Multiprozessor Task Scheduling

Entwicklung und Vergleich von Algorithmen zur optimalen Auslastung eines Parallelrechners

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von
Kai Baumgarten

aus Merseburg

1. Gutachter: Prof. Dr. Thomas Rauber
2. Gutachter: Prof. Dr. Wolf Zimmermann

Tag der Einreichung: 30. Juli 2013
Tag des Kolloquiums: 25. März 2014

An dieser Stelle möchte ich mich für die hervorragende Betreuung während meiner Promotionszeit durch Herrn Prof. Dr. Thomas Rauber bedanken. In vielen Diskussionen konnte ich ihm meine Probleme schildern und unzählige Ratschläge entgegennehmen.
Mein Dank geht auch an Herrn Prof. Dr. Wolf Zimmermann. Er eröffnete mir die Möglichkeit, mit anderen Forschern, welche an verwandten Themen arbeiten, auf der CIRM in Marseille in Verbindung zu treten.
Für seine Unterstützung bei der Suche nach den kleinen Unstimmigkeiten in meiner Promotion möchte ich Kristian Debrabant ein ganz großes Dankeschön aussprechen.
Außerdem haben meine Eltern viel Rücksicht auf meine Arbeit genommen und mich in allen Lebenslagen tatkräftig unterstützt, so daß ich die Promotion anfertigen konnte. Vielen Dank dafür.

Inhaltsverzeichnis

1 Einführung und Motivation 11
1.1 Aufgabenstellung 11
1.2 Einsatz der Verfahren 12
1.3 Anforderungen an Module und Parallelrechner 13
1.4 Visualisierung des Problems und ein Beispiel 14
1.5 Einordnung des Problems 15
2 Näherungsalgorithmen ohne Berücksichtigung von Modulabhängigkeiten 18
2.1 Optimierung von formbaren Modulen 18
2.1.1 Der flächenminimale Algorithmus „P1“ 18
2.1.2 Schrittweise Parallelisierung von Modulen 22
2.1.3 Verschlechterungen der Zwischenschedules zulassen 27
2.1.4 Strategien zur Positionierung aller Module 28
2.1.4.1 Sortierung nach der Modullaufzeit 28
2.1.4.2 Sortierung nach der Prozessoranzahl der Module 30
2.1.4.3 Sortierung nach der Modulfläche 33
2.1.4.4 Weitere Algorithmen für das Rechteck - Füll - Problem 33
2.1.5 Schrittweise Erhöhung der Gesamtfläche 33
2.1.6 Praktische Tests 39
2.1.7 Auswertung der Testergebnisse 40
2.1.8 Vergleich der erzielten Ergebnisse mit denen anderer Algorithmen 43
2.2 Optimierung von nicht formbaren Modulen 46
2.2.1 Der flächenminimierende Algorithmus „XP+MF“ 46
2.2.2 Praktische Tests 47
2.2.3 Auswertung der Testergebnisse 49
3 Näherungsalgorithmen beim Vorliegen eines serien - parallelen Modulabhängigkeitsgraphen 51
3.1 Optimierung von formbaren Modulen 51
3.1.1 Eigenschaften von serien - parallelen Graphen 51
3.1.2 Berechnung eines Konstruktionsbaumes 55
3.1.2.1 Auflösung von komplexen seriellen Vereinigungen 55
3.1.2.2 Der Konstruktionsbaum eines serien - parallelen Graphen ohne komplexe Vereinigungen 56
3.1.2.3 Der Konstruktionsbaum eines beliebigen serien - parallelen Graphen 73
3.1.3 Schrittweise Modulvereinigung gemäß des Konstruktionsbaumes 73
3.1.3.1 Der Algorithmus „2M1“ 73
3.1.3.2 Garantierte Ergebnisgüte 76
3.1.3.3 Weitere Eigenschaften des Schedules 81
3.1.4 Praktische Tests 82
3.1.5 Vergleich der erzielten Ergebnisse mit denen anderer Algorithmen 85
3.2 Optimierung von nicht formbaren Modulen 86
3.2.1 Der Algorithmus „2M1N“ 86
3.2.2 Praktische Tests 87
4 Näherungsalgorithmen beim Vorliegen eines beliebigen Modulabhängigkeitsgraphen 91
4.1 Optimierung von formbaren Modulen 91
4.1.1 Die schichtenorientierten Algorithmen „SchT“ und „SchHT“ 91

4.1.2 Der waldorientierte Algorithmus „WaSP“ 98
4.1.3 Der reduktionsbasierte Algorithmus „ReSP“ 100
4.1.4 Kritische Pfade 109
4.1.5 Praktische Tests 112
4.1.6 Vergleich der erzielten Ergebnisse mit denen anderer Algorithmen 114
4.2 Optimierung von nicht formbaren Modulen 116
4.2.1 Die Modifikation der Algorithmen 116
4.2.2 Praktische Tests 116
5 Vergleich aller Routinen 119
5.1 Matrixmultiplikation 119
5.2 Fast Fourier Transformation (FFT) 121
5.3 LR - Zerlegung mittels Gauß 123
5.4 Auswertung der Ergebnisse 125
6 Zusammenfassung und Ausblick 127
7 Index 131
8 Literaturverzeichnis 135
8.1 Vorliegendes Optimierungsproblem 135
8.2 Vorliegendes Optimierungsproblem mit speziellen Modulabhängigkeiten 136
8.3 Vorliegendes Optimierungsproblem ohne Modulabhängigkeiten 136
8.4 Rechteck - Füll - Problem 137
8.5 Shop - Scheduling 138
8.6 Scheduling zur Minimierung der durchschnittlichen Wartezeit 138
8.7 Weitere Schedulingverfahren 139
8.8 Arbeit mit Graphen und Suchalgorithmen 140
8.9 Serien - Parallele Graphen 140
9 Anhang - Zusatzinformationen 141
9.1 Minimumsuche in Linearzeit 141
9.2 Beispiel für „HMZ“ 142
9.3 Beispiel für „BMZ“ 144
9.4 Sortierung nach der Modulfläche 150
9.5 Kombination aus den Algorithmen „HMZ“ und „BMZ“ 151
9.6 Kleine Verbesserungen beim Rechteck - Füll - Problem 157
9.7 Auswahl des nächsten Moduls zur Parallelisierung 159
9.8 Sequentielle Positionierungsreihenfolge 161
9.9 Beispiel für „P1+MF“ 162
9.10 Beispielgenerierung für „P1+MF“ 163
9.10.1 Gleichverteilte Zufallszahlen 165
9.10.2 Exponentialverteilte Zufallszahlen 166
9.10.3 Normalverteilte Zufallszahlen 167
9.11 Implementierung des Algorithmus „P1+MF“ 169
9.12 Beispielgenerierung für „XP+MF“ 170
9.13 Implementierung des Algorithmus „XP+MF“ 171
9.14 Laufzeitanalyse von „GzuB“ 171
9.15 Eine verbesserte Minimumsuche für die parallele Vereinigung beim Algorithmus „2M1“ 173
9.16 Beispiele für den Algorithmus „2M1“ 175
9.17 Durchschnittliche Ergebnisgüte von „2M1“ 181
9.18 Möglichkeiten zur Verbesserung des Algorithmus „2M1” 192

9.19 Implementierung und Beispielgenerierung von „2M1“ 194
9.20 Beispielgenerierung für „2M1N“ 195
9.21 Der Algorithmus „SchHT“ 195
9.21.1 Obere Laufzeitschranke 195
9.21.2 Vergleich von „SchH“ und „SchHT“ 197
9.21.3 Beispiel für „SchT“ und „SchHT“ 198
9.22 Der Algorithmus „kGzuB“ 199
9.22.1 Der Pseudocode 199
9.22.2 Beispiel eines „kGzuB“ - Laufes 202
9.23 Beispiel für „KP+“ und „KPMF“ 204
9.24 Beispielgenerierung bei beliebigem Abhängigkeitsgraphen 205
9.25 Laufzeitberechnung für die Standardbeispiele 208
9.25.1 Matrixmultiplikation 208
9.25.2 Fast Fourier Transformation (FFT) 210
9.25.3 LR - Zerlegung mittels Gauß 214

Multiprozessor Task Scheduling
Entwicklung und Vergleich von Algorithmen zur optimalen Auslastung eines Parallelrechners

Zusammenfassung

Diese Arbeit befaßt sich mit der Berechnung eines möglichst optimalen Schedules für eine gegebene Anzahl an formbaren Modulen, wobei jedes Modul von einer beliebigen Anzahl an Prozessoren ausgeführt werden kann. Die Laufzeit eines Moduls hängt dabei von der zur Ausführung des Moduls verwendeten Anzahl an Prozessoren ab. Die Laufzeiten aller Module sind vor dem Start des Schedulingalgorithmus vollständig bekannt. Als weitere Eingabe nimmt ein Großteil der in dieser Arbeit vorgestellten Schedulingalgorithmen einen gerichteten azyklischen Graph entgegen. Dieser repräsentiert die Abhängigkeiten zwischen den einzelnen Modulen. Gehört der gegebene Graph zur Klasse der serien - parallelen Graphen, so wird diese zusätzliche Strukturinformation von einem Algorithmus zur Erstellung des Schedules ausgenutzt.
Als Ergebnis liefern alle Algorithmen einen Schedule, welcher eine möglichst geringe Gesamtlaufzeit zur Ausführung aller Module auf einer gegebenen Anzahl an Prozessoren benötigt. Zum Erreichen einer minimalen Gesamtlaufzeit versuchen alle Algorithmen die Parallelität zwischen verschiedenen Modulen (Taskparallelität) und die Parallelität innerhalb eines Moduls (Datenparallelität) optimal auszunutzen.
Im Gegensatz zu vielen anderen Arbeiten zu diesem oder einem verwandten Thema muß bei den meisten der hier vorgestellten Algorithmen der Aufwand zur Ausführung eines Moduls bei der Hinzunahme eines Prozessors nicht zwangsweise steigen. Dadurch sind die in dieser Arbeit angegebenen Schedulingalgorithmen unabhängig von den Werten der Eingabedaten nutzbar.
Bei der Entwicklung der Algorithmen wurde viel Wert auf ihre praktische Verwendbarkeit gelegt. Dies äußert sich insbesondere in einer geringen Laufzeit der Algorithmen zur Berechnung des Schedules, sowohl bei einer sehr großen Anzahl an gegebenen Modulen, als auch bei sehr vielen zur Verfügung stehenden Prozessoren. Die Laufzeit der meisten in dieser Arbeit vorgestellten Algorithmen wird durch die Sortierung der Module bezüglich ihrer Modullaufzeit dominiert. Es werden aber auch Algorithmen angegeben, deren Laufzeit in der Größenordnung der Mächtigkeit der Eingabedaten liegt. Die in verwandten Arbeiten vorgestellten Schedulingalgorithmen haben sehr häufig eine wesentlich höhere Laufzeit, so daß diese nur bedingt in der Praxis einsetzbar sind.
Selbstverständlich soll trotz der geringen Laufzeiten der Algorithmen ein sehr guter Schedule berechnet werden. Falls keine Abhängigkeiten zwischen den Modulen vorliegen, so wird ein Schedule erreicht, welcher im schlechtesten Fall die dreifache Gesamtlaufzeit gegenüber der Gesamtlaufzeit des optimalen Schedules benötigt. Liegen hingegen Modulabhängigkeiten vor, so richtet sich die garantierte Ergebnisgüte bei allen vorgestellten Schedulingalgorithmen nach den Eigenschaften der Eingabedaten.
Die in der Arbeit angegebenen Schedulingalgorithmen lassen sich nach ihrer Funktionsweise grob in drei Gruppen einteilen: Die erste Art partitioniert alle Module in mehrere Modulmengen, so daß zwischen den in einer Modulmenge enthaltenen Modulen keine Abhängigkeiten mehr bestehen. Für jede dieser Modulmengen wird dann ein Schedule errechnet. Diese Teilschedules werden am Ende zu einem Gesamtschedule zusammengefügt. Die zweite Art versucht, durch die gezielte Parallelisierung von Modulen, die auf dem kritischen Pfad liegen,

einen möglichst guten Schedule zu berechnen. Die dritte Art reduziert schrittweise die Modulanzahl. Dies geschieht durch die Zusammenlegung zweier oder mehrerer in bestimmten Abhängigkeiten stehender Module. Diese Art des Schedulings ist im Gegensatz zu den erstgenannten beiden Arten noch in keiner anderen bekannten Arbeit untersucht wurden.
Alle vorgestellten Schedulingalgorithmen wurden in C implementiert und anhand von zufällig erstellten Eingabedaten umfangreich getestet. Dabei lag die benötigte Gesamtlaufzeit des erzielen Schedule fast immer unter dem 1,3-fachen der grob nach unten abgeschätzten Gesamtlaufzeit des optimalen Schedules. In anderen Arbeiten wurden Schedulingalgorithmen mit pseudopolynomieller Laufzeit entwickelt, welche für den ermittelten Schedule gegenüber dem optimalen Schedule eine maximal um den Faktor 4,7 höhere Gesamtlaufzeit garantieren. Die praktisch ermittelten Ergebnisse der in dieser Arbeit vorgestellten Algorithmen liegen also in nahezu allen Fällen in dem von laufzeitintensiven Algorithmen garantierten Bereich, was wiederum für den Einsatz der schnellen Algorithmen in der Praxis spricht.
Weiterhin wurden die Schedulingalgorithmen an Beispielen, denen die LR - Zerlegung nach Gauß, die Matrixmultiplikation bzw. die Fast Fourier Transformation zugrunde liegt, getestet. Anhand dieser Beispiele wurden die Schedulingergebnisse der vorgestellten Algorithmen direkt miteinander verglichen.
Zusätzlich wird in dieser Arbeit für sehr viele Schedulingalgorithmen auch eine Version für nichtformbare Module vorgestellt. Dadurch können auch Module verarbeitet werden, die aufgrund ihrer internen Implementierung eine bestimmte Anzahl an Prozessoren voraussetzen.

Multiprocessor Task Scheduling
Design and Comparison of Algorithms for an Optimal Utilisation of Parallel Computers

Abstract

This thesis deals with the calculation of a schedule for an arbitrary, but fixed number of moldable parallel tasks, where each task can be executed using an arbitrary number of processors. Depending on the number of processors, different execution times may result for different tasks. The execution times of the tasks are assumed to be known before the calculation of the schedule starts. Moreover, a directed acyclic graph might be given. This graph represents the dependencies between the tasks. One presented scheduling algorithm is especially designed for graphs that belong to the class of serial - parallel graphs. This algorithm uses the additional information provided by the graph as a basis for the calculation of the schedule.

The goal of all algorithms presented in this thesis is to minimise the resulting makespan, which means that the overall completion time of the tasks should be a minimum. To get a good schedule, all algorithms try to use the parallelism between different tasks (task parallelism) and the parallelism within a single task (data parallelism) in an efficient way.

In contrast to other publications on the same or similar topics, this thesis does not generally assume that the work to execute a task increases if the number of processors used is increased. Most of the algorithms presented work without this limitation. That's why these algorithms can be used on arbitrary input data.

All algorithms presented are developed with respect to their practicality. This implies that all algorithms should have a minimal runtime, especially if the number of given tasks and/or available processors is high. The runtime of nearly all algorithms is dominated by the time used to sort the tasks. Additionally this thesis presents algorithms with a runtime that is bound linearly by the size of the input data. Algorithms in other publications usually have a much larger runtime. Therefore those algorithms can practically only be used for a quite small number of tasks.

Nevertheless, the short runtime of the presented algorithms does not imply that the calculated schedule is not close to the optimal one. If no dependencies between the tasks are given, it can be proven that the worst-case running time of the tasks using the calculated schedule is never more than three times the running time using the optimal schedule. In case of dependencies between the tasks, the quality of the schedule depends on the properties of the input data.

It is possible to categorise the algorithms presented in this thesis into three groups: The algorithms of the first group separate all tasks into different sets of tasks. There are no dependencies allowed between each pair of tasks within the same set. Afterwards the algorithms calculate a schedule for each set of tasks. These sub-schedules are finally put together for the overall schedule. The algorithms of the second group try to parallelize the tasks in the critical path to get a good schedule. The algorithms of the third group reduce, step by step, the number of tasks by unifying two or more tasks that are specially dependent on each other to a new task. The last group of algorithms is, in contrast to the first two groups, not part of any other known publication.

All presented algorithms have been implemented and checked extensively by the usage of randomly created input data. In nearly all cases the created schedule has a completion time which is less than 1.3-times of the lower bound of the completion time of the optimal sched-

ule. There are algorithms presented in other publications that guarantee a schedule which needs at most 4.7-times of the completion time of the optimal schedule. Unfortunately the runtime needed to create the schedule is pseudo-polynomial. Since the results computed with the fast algorithms presented in this thesis are almost always within the guaranteed range of the more time consuming algorithms, the fast algorithms are usually the better choice in practice.

Moreover all algorithms are additionally tested with non-random examples based on Gaussian elimination, matrix multiplication, and fast Fourier transformation. These examples provide a data basis to compare all algorithms presented in this thesis.

This thesis also presents a version for non-moldable tasks for many of the scheduling algorithms. Therefore it is also possible to schedule tasks that require a predefined minimum number of processors for their execution.

1 Einführung und Motivation

1.1 Aufgabenstellung

In dieser Arbeit werden verschiedene Näherungsalgorithmen zum statischen Scheduling von Modulen vorgestellt und analysiert. Ein Modul (in der Literatur manchmal auch Task genannt) ist dabei ein zusammenhängender Programmteil, welcher im allgemeinen mit einer beliebigen Anzahl an Prozessoren ausgeführt werden kann. Statisches Scheduling bedeutet, im Gegensatz zum dynamischen, daß die Verteilung der verfügbaren Ressourcen vor dem Start aller Module vorgenommen wird und während der Ausführung der Module unverändert bleibt. Um überhaupt eine Optimierung vornehmen zu können, müssen die Laufzeiten der Module bekannt sein. Im folgenden bezeichnet $T(x, i)$ die Laufzeit von Modul x bei der Verwendung von i Prozessoren. Die Funktion T kann anstatt eines mathematischen Ausdrucks auch als $M \times N$ Tabelle realisiert werden. Dabei steht M für die Anzahl der zu optimierenden Module, und N gibt die maximal nutzbare Gesamtprozessoranzahl an. Die N Prozessoren tragen die Nummern Eins bis N und sind in einem linearen Feld angeordnet. Die Module werden von Eins bis M durchnumeriert.
Ziel der nachfolgend vorgestellten Optimierungsalgorithmen ist es, allen Modulen $x \in [1; M]$ eine Startzeit $S(x)$, eine Prozessoranzahl $B(x)$ und einen Referenzprozessor $P(x)$ zuzuweisen. Dann wird x vom Zeitpunkt $S(x)$ bis zum Zeitpunkt $S(x) + T(x, B(x))$ auf den Prozessoren $P(x)$ bis $P(x) + B(x) - 1$ ausgeführt. Dies impliziert, daß ein Modul nur Prozessoren mit benachbarten Prozessornummern nutzen und während der Ausführung nicht unterbrochen oder auf andere Prozessoren umgelagert werden kann. Man spricht in diesem Fall von einem nonpreemptiven Scheduling. Das Ergebnis des Schedulingvorgangs (hier $S(x)$, $B(x)$ und $P(x)$) heißt Schedule. Logischerweise darf sich zu jedem Zeitpunkt auf jedem Prozessor maximal ein Modul in Arbeit befinden. Mit H bezeichnet man die Gesamtlaufzeit des erzielten Schedules, also die insgesamt benötigte Zeit

$$H = \max_{x=1}^{M} \{ S(x) + T(x, B(x)) \}$$

zur Ausführung aller M Module auf N Prozessoren. Als abkürzende Schreibweise steht $T(x)$ für die Laufzeit des Moduls x bei der Verwendung von $B(x)$ Prozessoren, also $T(x) = T(x, B(x))$.

Definition 1: *Ein Schedule A mit der Gesamtlaufzeit H_A ist besser als ein Schedule B mit der Gesamtlaufzeit H_B, wenn die Ungleichung $H_A < H_B$ gilt. Der beste erzielbare Schedule heißt optimal und hat die Gesamtlaufzeit*

$$O = min\{ H_A : A \text{ ist ein Schedule} \}.$$

Die Güte (oder Suboptimalität) eines ermittelten Schedules läßt sich mit Hilfe des Quotienten $G = \frac{H}{O}$ angeben.

Je näher G an Eins liegt, um so besser ist der Schedule. Nachfolgend wird G als Gütefaktor bezeichnet.

Definition 2: *Zur Modellierung der Datenströme zwischen den einzelnen Modulen wird ein gerichteter, zyklusfreier Modulabhängigkeitsgraph G = (V, E) verwendet. Die Knotenmenge V enthält alle Module, also $V = \{1; 2; ...; M\}$ und $|V| = M$. Benö-*

tigt das Modul y die Ergebnisse von Modul x, so geht eine Kante von x nach y oder in Zeichen (x, y) ∈ E.

Sollte G einen Zyklus enthalten, so existiert keine Möglichkeit, die einzelnen Module nacheinander auszuführen. Aus diesem Grund muß der Modulabhängigkeitsgraph zyklusfrei sein.
Eine umfassende Einführung zu der soeben vorgestellten Optimierungsaufgabe ist in [4] und in [5] angegeben.

Die in dieser Arbeit aufgeführten Algorithmen versuchen, die soeben vorgestellte Optimierungsaufgabe mit einem möglichst kleinen Gütefaktor G zu lösen. Da es sich um eine sehr komplexe Berechnung handelt, werden in den ersten beiden Kapiteln einige Spezialfälle betrachtet.

1.2 Einsatz der Verfahren

Bei vielen Berechnungen reicht ein Rechner mit nur einem Prozessor nicht mehr aus, um das Ergebnis in angemessener Zeit zu ermitteln. Aus diesem Grund wurden Parallelrechner entwickelt. Im Zuge der Entwicklung von Multi-Core-CPU's ist man zur optimalen Auslastung der gesamten CPU ebenfalls auf parallele Algorithmen angewiesen. Außerdem ist man bereits sehr nahe an die physikalisch mögliche Grenze der CPU-Taktfrequenz gekommen, so daß die wachsenden Anforderungen an die Software und die damit verbundene Erhöhung der Komplexität der Programme nicht mehr durch die schnellere Hardware ausgeglichen werden.
Leider ist die Transformation des sequentiellen Programms in ein effizientes paralleles Programm im allgemeinen nicht so einfach möglich. Zergliedert man den sequentiellen Algorithmus in seine Teilschritte (egal ob manuell oder automatisch), so hat man einerseits die Möglichkeit, unabhängige Teilschritte gleichzeitig auf verschiedenen Prozessorgruppen auszuführen. Man spricht in diesem Fall von der Ausnutzung der Taskparallelität eines Algorithmus. Andererseits müssen in einem Teilschritt manchmal die gleichen Operationen auf verschiedenen Daten ausgeführt werden. Dies ist zum Beispiel in Schleifen oft der Fall. Verwendet man mehrere Prozessoren, um die Operationen gleichzeitig auf verschiedenen Daten auszuführen, so spricht man von der Ausnutzung der Datenparallelität des Verfahrens. Meistens hat man aber nicht genug Prozessoren zur Verfügung, um alle Task- und Datenparallelitäten ausnutzen zu können, oder es lohnt sich für die gegebene Zielmaschine nicht, das gesamte Potential an Task- und Datenparallelität auszunutzen, oder es lohnt sich nur, eine spezielle Kombination von Task- und Datenparallelität auszunutzen. Man muß also entscheiden, in wieweit man Task- und Datenparallelitäten in jedem Teilschritt ausnutzt, um ein möglichst effizientes paralleles Programm zu erstellen.
In Anlehnung an die in der Aufgabenstellung eingeführten Begriffe, entspricht ein Teilschritt des Algorithmus einem Modul. Die Möglichkeit der Ausnutzung der Datenparallelität ist durch die Ausführung des Moduls mit einer beliebigen Prozessoranzahl gegeben. Die Taskparallelität resultiert aus der gleichzeitigen Ausführung zweier oder mehrerer Module. Zur Modellierung der Datenströme zwischen den Teilschritten kann man den Modulabhängigkeitsgraphen verwenden.
Nun kann es vorkommen, daß es bei einigen Modulen aufgrund von Schleifen, Cacheeffekten o.ä. günstig ist, dieses Modul mit mindestens i Prozessoren auszuführen, um einen effizienten Programmcode zu erstellen.

Definition 3: *Ein Modul x, welches zur Ausführung mindestens i Prozessoren ($i \geqslant 2$) benötigt, heißt nicht formbar (non-moldable). In diesem Fall liefert $T(x, j)$ für alle $j < i$ den Wert unendlich.*

Kann das Modul mit einer beliebigen Anzahl an Prozessoren und damit auch sequentiell, d.h., mit einem Prozessor, abgearbeitet werden, so wird es mit formbar (moldable) bezeichnet.
Die zugeordnete Prozessoranzahl bleibt in beiden Fällen während der gesamten Modulausführung konstant.

In der Literatur wird oft eine strengere Definition für nicht formbar verwendet, als dies in dieser Arbeit der Fall ist. Ein Modul x heißt dort nicht formbar, wenn eine Prozessoranzahl i existiert, so daß gilt:

$$T(x, j) = \infty \text{ für } 1 \leqslant j < i \quad \text{und} \quad T(x, k) = T(x, i) \text{ für } i < k \leqslant N.$$

Das Scheduling von nicht formbaren Modulen entspricht nach dieser strengeren Definition dem Rechteck - Füll - Problem (siehe Abschnitt 1.5).

Außerdem wird in der Literatur auch oft der Begriff malleable mit der gleichen Bedeutung wie moldable verwendet. In [53] ist der Unterschied aber sehr gut beschrieben: Einem Modul mit der Eigenschaft malleable kann auch während der Modulausführung eine andere Prozessoranzahl zugewiesen werden.

Wie die in dieser Arbeit angegebenen Optimierungsalgorithmen bei der Programmerstellung eingesetzt werden können, wird detailliert in [13] beschrieben.
In [4] wird anhand der Cholesky - Faktorisierung und der Modellierung von Ozeanströmungen der Einsatz von formbaren Modulen zur Bestimmung eines Schedules vorgeführt.

1.3 Anforderungen an Module und Parallelrechner

Im Abschnitt 1.1 wurde schon erwähnt, daß die Prozessoren des Parallelrechners linear angeordnet sein müssen. Natürlich ist es auch möglich Parallelrechner zu verwenden, die die Einbettung eines linearen Feldes ermöglichen. Diese Tatsache erfüllen fast alle Parallelrechner, denn ein lineares Feld ist eine der am einfachsten zu realisierende Ordnung auf einer Menge von Prozessoren. Eine Ausnahme bilden zum Beispiel diejenigen Parallelrechner mit einer sternförmigen Struktur. Aber auch diese Klasse an Parallelrechnern kann meistens eine lineare Anordnung sehr gut simulieren.
Ein auf einer linearen Anordnung basierender Parallelrechner heißt fragmentierbar, wenn es möglich ist, ein Modul auf nicht benachbarten Prozessoren zu starten. Aus der Aufgabenstellung folgt, daß der in dieser Arbeit angenommene Parallelrechner nicht fragmentierbar zu sein braucht.
Ein Parallelrechner verwendet genau dann Synchronisationsmarken zum Modulstart, wenn ein Modul nicht sofort auf einer freien Prozessorgruppe gestartet werden kann, sondern der Start des Moduls nur zu bestimmten, in regelmäßigen Abständen wiederkehrenden Synchronisationszeitpunkten möglich ist. Verwendet die Zielmaschine zum Modulstart Synchronisationsmarken, so müssen die Modullaufzeiten auf Vielfache des Abstandes zwischen zwei Synchronisationszeitpunkten aufgerundet werden. Dann ist gewährleistet, daß man ein neues Modul jederzeit auf einer freien Prozessorgruppe ohne weitere Berücksichtigung der Synchronisationszeitpunkte starten kann.

Definition 4: *Für ein formbares Modul x gilt genau dann die Flächenbedingung, wenn* $\forall i \in [1; N]$ *folgt, daß* $\frac{T(x, 1)}{T(x, i)} \leqslant i$ *ist.*

Erfüllt ein formbares Modul x nicht die Flächenbedingung, so gibt es mindestens ein $i \in [1; N]$ mit $\frac{T(x, 1)}{T(x, i)} > i$, d.h., der Geschwindigkeitsgewinn bei der Ausführung von x auf i Pro-

zessoren ist superlinear. In der Praxis kann ein superlinearer Geschwindigkeitsgewinn z.B. durch Cacheeffekte entstehen. Bezeichnet man das Produkt aus der Prozessoranzahl und Laufzeit als Fläche des Moduls, so ist für alle Module x, die die Flächenbedingung erfüllen, die benötigte Fläche bei der Ausführung von x mit mehr als einem Prozessor nie kleiner als die Fläche bei der sequentiellen Ausführung von x.
Außerdem muß bei einem festen Modul x die Funktion T monoton fallend sein, also aus $\forall i, j \in [1; N]$ mit $i \leqslant j$ die Eigenschaft $T(x, i) \geqslant T(x, j)$ folgen. Ist $T(x, i)$ tatsächlich echt kleiner als $T(x, j)$, so kann bei der Ausführung von x mit j Prozessoren auf den Algorithmus mit i Prozessoren zurückgegriffen werden. Die verbleibenden j - i Prozessoren haben dann einen „sinnvollen“ Leerlauf.
Ist das Modul x nicht formbar und benötigt mindestens i Prozessoren, so gilt laut Abschnitt 1.2 für alle $j < i$: $T(x, j) = \infty$. Aus diesem Grund gilt die Monotonieeigenschaft auch für nicht formbare Module.
Die Gültigkeit der Flächenbedingung und die Monotonie der Laufzeiten wird für alle Module in der gesamten Arbeit vorausgesetzt.

1.4 Visualisierung des Problems und ein Beispiel

Damit man sich schnell ein Bild vom erzielten Schedule machen kann, ist eine graphische Darstellung des Schedules sinnvoll. Dazu nutzt man am besten ein Prozessor - Zeit - Diagramm (auch Gantt - Diagramm genannt; siehe [17]). In diesem Diagramm wird jedes Modul als Rechteck entsprechend der Ausführungszeit und der verwendeten Prozessoren eingetragen (vgl. Abbildung 2).

Modullaufzeiten				
	Anzahl der Prozessoren			
Modulnummer	1	2	3	4
1	8	7,9	7,8	7,7
2	15	7,5	5	4,5
3	3	2,5	2,2	2
4	9	5	4	3,5
5	3,8	2	1,8	1,5
6	3	2,5	2	1

Zur Verdeutlichung der Aufgabenstellung soll das folgende Beispiel dienen: Der angenommene Parallelrechner habe 4 Prozessoren ($N = 4$). Weiterhin existieren 6 Module ($M = 6$), wobei Modul 1 erst nach Beendigung von Modul 6 gestartet werden kann, und Modul 2 auf die Ergebnisse der Module 3 und 5 wartet. Die Laufzeiten der Module sind aus der Tabelle entnehmbar. Die Abbildung 1 zeigt den in diesem Beispiel entstehenden Abhängigkeitsgraphen.

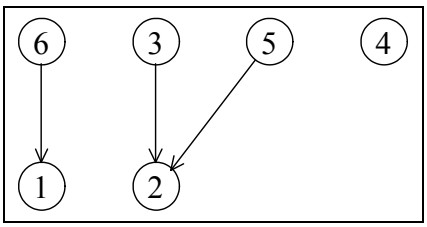


Abbildung 1 - Modulabhängigkeitsgraph des einführenden Beispiels.

Wie man erkennen kann, ist die in Abbildung 2(a) angegebene Möglichkeit zur Positionierung aller Module nicht zulässig, da das Modul 6 zum Startzeitpunkt von Modul 1 noch abgearbeitet wird. Die Versionen (b) und (c) erfüllen erst einmal alle Voraussetzungen und sind somit zulässige Schedules. Ein Vergleich der beiden Schedules zeigt aber, daß der in der Abbildung 2(c) dargestellte Schedule schon nach 11 Zeiteinheiten fertig ist. Der Schedule (b) hingegen benötigt noch 0,8 Zeiteinheiten mehr, d.h., der Schedule (c) ist besser als der Schedule (b).

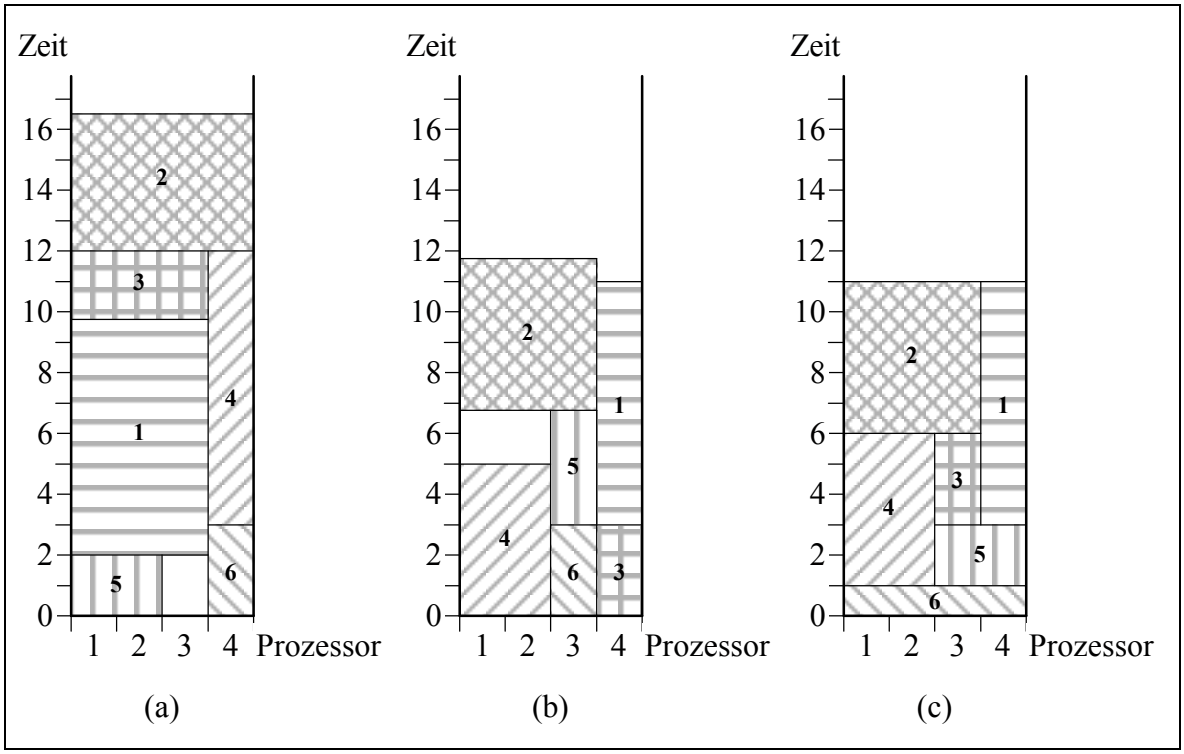


Abbildung 2 - Gantt - Diagramme für Beispiel 1.

Aber sind die 11 Zeiteinheiten nun das Minimum? Man könnte jetzt argumentieren, daß bis zum Zeitpunkt H alle Prozessoren ein Modul ausführen und die Gesamtlaufzeit des Schedules demzufolge minimal sein muß. Diese Schlußweise ist aber falsch, denn wenn man alle Module nacheinander auf jeweils allen Prozessoren starten würde, existiert auch kein Zeitpunkt, zu dem ein Prozessor kein Modul ausführt. In diesem Beispiel hätte man dann jedoch eine Gesamtlaufzeit von 20,2 Zeiteinheiten.

1.5 Einordnung des Problems

Zu der hier vorgestellten Optimierungsaufgabe gibt es natürlich auch verwandte Probleme. Das erste ist das Rechteck - Füll - Problem. Bei ihm hat man die Abmaße mehrerer Rechtecke gegeben. Diese Rechtecke müssen dann in ein großes Rechteck fester Breite gepackt werden, so daß seine Höhe minimal ist (vgl. [17]). Dabei darf man die kleinen Rechtecke nicht drehen, d.h., die Vertauschung von Breite und Höhe der kleinen Rechtecke ist nicht zulässig.

Lemma 1: *Das Rechteck - Füll - Problem ist ein Spezialfall der in dieser Arbeit betrachteten Optimierungsaufgabe.*

Beweis: Will man die im Lemma 1 aufgestellte Behauptung beweisen, muß man zuerst durch die Multiplikation mit einer hinreichend großen Zahl alle Rechteckbreiten (auch die des gro-

ßen Rechtecks) in den Bereich der natürlichen Zahlen konvertieren. Da die Zahlendarstellung im Computer nur eine endliche Genauigkeit hat, ist dies immer möglich. Im weiteren Beweis sei die Breite aller Rechtecke eine natürliche Zahl. Man setze nun N auf die Breite des großen Rechtecks und erzeuge für jedes einzusortierende Rechteck z ein Modul x mit folgenden Daten:

$$T(x, i) = \begin{cases} \infty & : \text{falls } 1 \leqslant i < \text{Breite von z} \\ \text{Höhe von z} & : \text{falls Breite von z} \leqslant i \leqslant N \end{cases}$$

Der Modulabhängigkeitsgraph enthält in diesem Fall keine Kanten. Nach der Berechnung der Startzeiten S(x) und Positionen P(x) aller Module kann über S(x) und P(x) die linke untere Ecke des zum Modul x gehörenden Rechtecks z entnommen werden.

■

Aus diesem Grund sind alle komplexitätstheoretischen Ergebnisse, die beim Rechteck - Füll - Problem erzielt wurden, auf das in dieser Arbeit betrachtete Problem übertragbar. In [18] ist ein Beweis angegeben, welcher zeigt, daß das Rechteck - Füll - Problem NP - hart ist. Damit folgt:

Lemma 2: *Die in dieser Arbeit betrachtete Optimierungsaufgabe ist NP - hart.*

Mit dieser Erkenntnis weiß man, daß der optimale Schedule nicht in vertretbarer Zeit berechenbar ist. Aus diesem Grund werden in dieser Arbeit nur Näherungsalgorithmen zur Berechnung eines Schedules betrachtet.

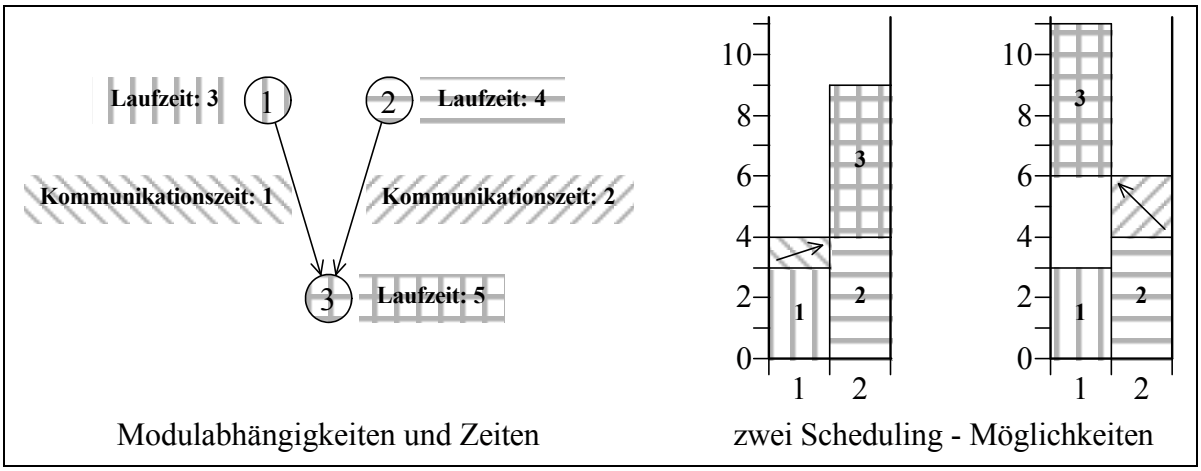


Abbildung 3 - DAG - Shop - Scheduling.

Eine zweite verwandte Aufgabenstellung wird als DAG - Shop - Scheduling bezeichnet. Dabei müssen, wie in dieser Arbeit, Module auf verschiedene Prozessoren so verteilt werden, so daß eine minimale Gesamtlaufzeit resultiert. Für diese Module ist ebenfalls die Angabe von Abhängigkeiten zulässig. Beim DAG - Shop - Scheduling benötigen jedoch alle Module zur Ausführung exakt einen Prozessor. Außerdem ist es dort zulässig, den Kanten des Modulabhängigkeitsgraphen Kosten zuzuordnen. Sie entsprechen der Zeit, welche der Datenaustausch zwischen den Modulen benötigt, wenn das nachfolgende Modul nicht auf dem gleichen Prozessor gestartet wird. In der Abbildung 3 ist diese Tatsache noch einmal dargestellt.
Unter der Voraussetzung, daß alle Kommunikationszeiten den Wert Null haben, erhält man das DAG - Shop - Scheduling als Spezialfall der in dieser Arbeit gestellten Aufgabe. In [35] findet man einen Beweis, welcher diesen Unterfall schon als NP - vollständig einordnet.

Die dritte verwandte Aufgabenstellung geht davon aus (siehe [40]), möglichst viele Module möglichst schnell abzuarbeiten. Es wird also nicht nach H optimiert, sondern es soll

$$\sum_{x=1}^{M} s(x) + t(x)$$

minimal sein. Diese Herangehensweise ist zum Beispiel wichtig, wenn die Ergebnisse der einzelnen Module hinterher getrennt weiterverarbeitet werden sollen. Untersuchungen zur Kombination der Optimierung nach der Gesamtausführungszeit H mit der schnellen Ausführung möglichst vieler Module findet man in [43] und [44]. Dabei dürfen in [43], im Gegensatz zu den Ausführungen in [44], keine Modulabhängigkeiten vorliegen.

Eine relativ umfangreiche Übersicht über weitere verwandte Schedulingprobleme ist in [4] angegeben. In [5] sind die Ergebnisse von komplexitätstheoretischen Untersuchungen zu einigen verwandten Schedulingproblemen aufgeführt. Weitere Komplexitätsbetrachtungen sind in [36] nachlesbar.
Die in dieser Arbeit behandelte Aufgabenstellung berücksichtigt keine Kommunikationszeiten zwischen den Modulen. Aus diesem Grund betrachte man die folgenden Parallelrechnerarten:

1) Alle Prozessoren des Parallelrechners greifen auf einen gemeinsamen Speicher zu. Die Zugriffszeit auf ein und dieselbe Speicherposition ist für alle Prozessoren gleich.
   Bei Parallelrechnern dieses Typs tritt keine Kommunikationszeit auf. Die in dieser Arbeit vorgestellten Schedulingalgorithmen können problemlos eingesetzt werden.
2) Jeder Prozessor hat einen Speicherbereich, auf welchen er schnell zugreifen kann. Ein Zugriff auf den Speicherbereich eines anderen Prozessors ist erheblich langsamer.
   Bei dieser Parallelrechnerart kann man bei der Berechnung der Modullaufzeiten die folgende Heuristik anwenden: Wird in einem Modul x erstmalig lesend auf eine Speicherzelle s zugegriffen, so benötigt dieser Zugriff maximal r + w Zeiteinheiten. Dabei steht r für die maximale Zeit, die zum Lesen einer Speicherzelle im Speicherbereich eines anderen Prozessors benötigt wird. Wird auf die Speicherzelle s in x nicht noch einmal zugegriffen, so gilt w = 0. Anderenfalls entspricht w der Zeit, die zum Schreiben des Inhalts der Speicherzelle s in den eigenen Speicherbereich notwendig ist. Alle nachfolgenden Zugriffe auf den Inhalt der Speicherzelle s sind dann Zugriffe auf die Kopie im eigenen Speicherbereich. Die so ermittelten Modullaufzeiten sind garantiert nicht kleiner als die realen Modullaufzeiten. Demzufolge ist die tatsächliche Gesamtlaufzeit aller Module höchstens geringer, als die vom Schedulingalgorithmus berechnete.
3) Zu jedem Prozessor gehört ein lokaler Speicher. Der direkte Zugriff auf den lokalen Speicher eines anderen Prozessors ist nicht möglich. Ein Datenaustausch zwischen den lokalen Speichern verschiedener Prozessoren kann nur mittels Sende- und Empfangsoperationen realisiert werden.
   Bei Parallelrechnern dieses Typs müssen zur Laufzeitberechnung der Module die folgenden zwei Zeiten berücksichtigt werden: die Zeit zum Empfangen der Eingangsdaten (inkl. der Datenlaufzeit vom Sender zum Empfänger) und die Zeit zum Versenden der berechneten Daten. Wählt man bei beiden Zeiten den maximal möglichen Wert, so liegen die berechneten Modullaufzeiten nie unter der tatsächlich benötigten Modullaufzeit.

2 Näherungsalgorithmen ohne Berücksichtigung von Modulabhängigkeiten

2.1 Optimierung von formbaren Modulen

2.1.1 Der flächenminimale Algorithmus „P1“

Im gesamten Kapitel 2.1 wird davon ausgegangen, daß die Kantenmenge im Modulabhängigkeitsgraphen leer ist, und alle Module auch mit nur einem Prozessor ausführbar sind. Demzufolge ist die Fläche eines jeden Moduls bei der Ausführung mit nur einem Prozessor minimal (vgl. Definition 4). Gelingt es nun, die Module derart zu positionieren, daß alle Prozessoren ohne Unterbrechung arbeiten und auf jedem Prozessor bis zum Zeitpunkt H ein Modul ausgeführt wird, so ist der erzielte Schedule sicher optimal. Die Bedingung, daß alle Prozessoren des Parallelrechners ohne Unterbrechung arbeiten, ist leicht zu realisieren: Dazu startet man die Module nacheinander auf dem Prozessor, welcher die aktuell kleinste Endzeit hat.

Definition 5: *Mit der Endzeit $E(i)$ eines Prozessors i wird der Zeitpunkt bezeichnet, an dem das letzte auf i ausgeführte Modul beendet ist. Wird auf i kein Modul abgearbeitet, so ist $E(i) = 0$.*

Man beachte, daß die Endzeit eines Prozessors während des Ablaufes des Algorithmus dynamisch verändert wird.
Somit hat man nur noch die Reihenfolge, in welcher die Module auf dem Prozessor mit der jeweils kleinsten Endzeit gestartet werden, als freien Parameter. Intuitiv beginnt man das Positionieren mit dem Modul, welches die größte sequentielle Laufzeit hat und fährt absteigend fort. Der Pseudocode des Algorithmus „P1“ ist in der Abbildung 4 dargestellt.

```
1)  ∀i ∈ [1; N]: E(i) = 0
2)  sortiere die Module so, daß aus x < y die Beziehung T(x, 1) ⩾ T(y, 1) folgt
3)  für x = 1 bis M
4)      suche den Prozessor i mit der kleinsten Endzeit heraus (gibt es mehrere, so wähle
        den mit der kleinsten Nummer), also i = min_{j=1}^{N} { j : E(j) = min_{k=1}^{N} { E(k) } }
5)      führe x auf i aus, also S(x) = E(i); B(x) = 1; P(x) = i; E(i) = E(i) + T(x, 1)
```

Abbildung 4 - Algorithmus „P1“.

Die Laufzeit von „P1“ setzt sich aus der Sortierung von M Modulen und anschließenden M Minimumbildungen über N Zahlen zusammen. Sie liegt somit in $O(M \cdot \log(M) + N + M \cdot \log(N))$. Zu diesem Ergebnis gelangt man, wenn die N Prozessorendzeiten in einem Heap sortiert gehalten werden. Der einmalige Aufbau des Heaps benötigt $O(N)$ Schritte. Das Aufsuchen und Löschen des kleinsten Elementes sowie das Eintragen des neuen Wertes ist dafür in $O(\log(N))$ Zeiteinheiten erledigt.
Da für die Modullaufzeiten der Datentyp `double` verwendet wird, ist im allgemeinen eine Sortierung in $O(M)$ mittels Bucketsort oder einem ähnlichen Verfahren nicht sinnvoll.

Zur Abschätzung der Güte des von „P1“ berechneten Schedules wird das folgende Lemma angegeben:

Lemma 3: *Die Module seien absteigend nach ihrer sequentiellen Laufzeit $T(x, 1)$ sortiert. Es gelte also:*

$$\forall x \in [1; M - 1]: T(x, 1) \geq T(x + 1, 1).$$

Wenn der vom Algorithmus „P1“ erstellte Schedule, in welchem zur Ausführung aller M Module auf N Prozessoren H Zeiteinheiten benötigt werden, eine der drei Bedingungen (a) bis (c) (vgl. Abbildung 5) erfüllt, wird von „P1“ ein Gütefaktor G kleiner oder gleich Zwei erreicht:

(a) *Das zuletzt eingefügte Modul ende zum Zeitpunkt H, d.h., $S(M) + T(M, 1) = H$. Dabei steht $S(M)$ für die von „P1“ berechnete Startzeit des Moduls M. Außerdem seien mindestens $\lceil ½ \cdot (N + 1) \rceil$ Module vorhanden.*

(b) *Das zuletzt eingefügte Modul ende nicht zum Zeitpunkt H, d.h., $S(M) + T(M, 1) < H$. Außerdem sei $S(M) \geq ½ \cdot H$.*

(c) *Es gelten*

$$S(M) + T(M, 1) < H, \quad S(M) < ½ \cdot H \quad \text{und} \quad T(1, 1) \leq \frac{2}{N - 1} \cdot \sum_{x = 2}^{M} T(x, 1).$$

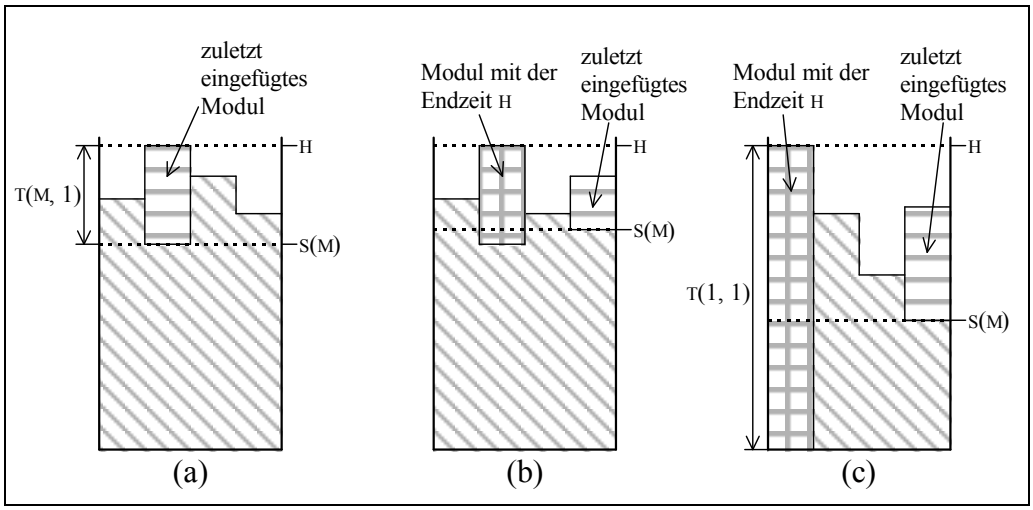


Abbildung 5 - Illustration der Einfügesituationen.

Beweis des Lemmas bei Gültigkeit von (a):

Fall 1: Das zuletzt eingefügte Modul beginne nicht zum Zeitpunkt Null, also $S(M) > 0$.

Da alle Module so zeitig wie möglich gestartet werden, sind im Prozessor - Zeit - Diagramm keine internen Lücken vorhanden. Es gibt also keinen Prozessor, welcher nach einem Zeitintervall ohne Ausführung eines Moduls noch ein neues Modul abarbeitet. Daraus kann man schließen, daß alle Prozessoren mindestens bis zum Zeitpunkt $S(M)$ ausgelastet sind. Da die Modulfläche bei der sequentiellen Ausführung minimal ist, kann die im optimalen Schedule belegte Fläche den Wert

$$(N - 1) \cdot S(M) + H = N \cdot S(M) + T(M, 1)$$

nicht unterschreiten. Aus diesem Grund ist die Gesamtlaufzeit des optimalen Schedules nie kleiner als

$$\frac{N \cdot S(M) + T(M, 1)}{N} = S(M) + \frac{T(M, 1)}{N}.$$

Die von „P1“ ermittelte Gesamtlaufzeit beträgt hingegen H. Für den Gütefaktor gilt also

$$G \leqslant \frac{H}{S(M) + \frac{T(M, 1)}{N}} = \frac{S(M) + T(M, 1)}{S(M) + \frac{T(M, 1)}{N}}. \qquad (1)$$

Je kleiner $S(M)$ ist, um so größer wird der Quotient. Da das Modul M mit der kleinsten Laufzeit nicht zum Zeitpunkt Null gestartet wird, wird auf $P(M)$ vor dem Modul M noch mindestens ein anderes Modul x mit der sequentiellen Laufzeit $T(x, 1)$ ausgeführt. Es gilt

$$S(M) \geqslant T(x, 1) \geqslant T(M, 1).$$

Setzt man dies in die Ungleichung (1) ein, so folgt für den Gütefaktor

$$G \leqslant \frac{T(M, 1) + T(M, 1)}{T(M, 1) + \frac{T(M, 1)}{N}} = \frac{2}{1 + \frac{1}{N}} < 2.$$

Fall 2: $S(M) = 0$
Unter dieser Voraussetzung beginnen alle Module zum Zeitpunkt Null. Da das Modul mit der kleinsten Laufzeit erst zum Zeitpunkt H endet, haben alle M Module die gleiche Laufzeit, also

$$T(1, 1) = T(2, 1) = ... = T(M, 1).$$

Die minimale Fläche für den optimalen Schedule beträgt somit $M \cdot T(1, 1)$. Setzt man nun die Voraussetzung ein, so kann man die im optimalen Schedule minimal belegte Fläche mittels

$$M \cdot T(1, 1) \geqslant \left\lfloor \tfrac{1}{2} \cdot (N + 1) \right\rfloor \cdot T(1, 1) \geqslant \tfrac{1}{2} \cdot N \cdot T(1, 1)$$

abschätzen. Die Gesamtlaufzeit des optimalen Schedules liegt demzufolge nie unter $\frac{1}{2} \cdot T(1, 1)$. Als schlechtesten Gütefaktor erhält man, da $T(1, 1) = H$ ist, den Wert Zwei.

Beweis des Lemmas bei Gültigkeit von (b):
Da mindestens ein Prozessor bis zum Zeitpunkt H arbeitet und alle anderen wie im Fall (a) mindestens bis zum Zeitpunkt $S(M)$ aktiv sind, beträgt der minimale Flächenbedarf für den optimalen Schedule $s(M) \cdot (N - 1) + H$. Demzufolge benötigt der optimale Schedule mindestens $\frac{S(M) \cdot (N - 1) + H}{N}$ Zeiteinheiten. Für den Gütefaktor gilt deshalb

$$G \leqslant \frac{H}{\frac{S(M) \cdot (N - 1) + H}{N}} \leqslant \frac{H \cdot N}{H + \frac{1}{2} \cdot H \cdot (N - 1)} = \frac{N}{\frac{1}{2} + \frac{1}{2} \cdot N} < 2.$$

Beweis des Lemmas bei Gültigkeit von (c):
Zuerst wird gezeigt, daß in diesem Fall $T(1, 1) = H$ gilt: Angenommen, das zuerst eingefügte Modul endet nicht zum Zeitpunkt H (also $T(1, 1) < H$), sondern ein anderes Modul x ($x \in [2; M - 1]$) mit der Laufzeit $T(x, 1) \leqslant T(1, 1)$. Die Startzeit $S(x)$ des Moduls x liegt demzufolge bei $H - T(x, 1)$. Da das Modul x vor dem letzten Modul eingefügt wird, gilt

$$H - T(x, 1) = S(x) \leqslant s(M) < \tfrac{1}{2} \cdot H.$$

Aus dieser Ungleichung läßt sich nun

$$T(x, 1) > \tfrac{1}{2} \cdot H \quad \text{bzw.} \quad 2 \cdot T(x, 1) > H$$

herleiten. Wie man sehen kann, gibt es keinen Prozessor, auf dem ein Modul y mit einer sequentiellen Laufzeit $T(y, 1) \geqslant T(x, 1)$ und das Modul x ausgeführt werden. Es müßte sonst

$$H \geqslant T(y, 1) + T(x, 1) \geqslant 2 \cdot T(x, 1) > H$$

gelten. Also beginnt das Modul x zum Zeitpunkt Null, und es folgt

$$T(x, 1) = H > T(1, 1)$$

und daraus mit $T(x, 1) \leqslant T(1, 1)$ ein Widerspruch.
Da $T(1, 1) = H$ ist, läßt sich für den Gütefaktor analog zu den ersten beiden Fällen

$$G \leqslant \frac{H}{\frac{1}{N} \cdot \sum_{x=1}^{M} T(x, 1)} = \frac{H \cdot N}{T(1, 1) + \sum_{x=2}^{M} T(x, 1)} \leqslant \frac{H \cdot N}{T(1, 1) + \frac{N - 1}{2} \cdot T(1, 1)} \quad \text{(nach Voraussetzung)}$$

$$= \frac{H}{T(1, 1)} \cdot \frac{N}{½ + ½ \cdot N} = \frac{N}{½ + ½ \cdot N} < 2$$

herleiten.
Auf gleiche Weise kann man die folgende Erweiterung zu diesem Fall zeigen: Gilt nur

$$T(1, 1) \leqslant \frac{a}{N - 1} \cdot \sum_{x=2}^{M} T(x, 1)$$

für ein beliebiges a mit $1 \leqslant a \leqslant N$, so kann der Gütefaktor mit $G \leqslant a$ abgeschätzt werden.

■

Der Vollständigkeit halber wird noch ein Beispiel angegeben, bei dem der eben vorgestellte Algorithmus keinen Schedule mit zufriedenstellendem Gütefaktor berechnet. Dazu existiere nur ein Modul, welches beim Start mit i Prozessoren die Laufzeit $\frac{1}{i}$ annimmt. Der von „P1“ ermittelte Schedule ist um den Faktor N schlechter als der optimale Schedule.
Leider ist auch eine hohe Modulanzahl keine Garantie für einen guten Schedule. Es seien M Module (M hinreichend groß) und ein kleines $\varepsilon \in (0; 1]$ gegeben. Diese M Module haben die folgenden Laufzeiten:

$$T(1, i) = \frac{1}{i} \quad (\forall i \in [1; N]) \qquad \text{sowie} \qquad T(x, i) = \frac{\varepsilon}{(M - 1) \cdot i} \quad (\forall i \in [1; N], \forall x \in [2; M]).$$

Man kann erkennen, daß der Algorithmus „P1“ in diesem Fall einen Schedule mit der Gesamtlaufzeit H = 1 erstellt. Der optimale Schedule hat aber nur eine Gesamtlaufzeit von

$$O = \frac{1}{N} + \frac{\varepsilon}{N}.$$

Trotz einer hohen Modulanzahl erreicht „P1“ bei diesem Beispiel nur einen Gütefaktor in der Nähe von N:

$$G = \frac{H}{O} = \frac{1}{\frac{1}{N} + \frac{\varepsilon}{N}} = \frac{N}{1 + \varepsilon}.$$

Lemma 4: *Es bezeichne H die Gesamtlaufzeit des von „P1“ ermittelten und O die des optimalen Schedules. Außerdem stehe T(x, 1) für die sequentielle Laufzeit des Moduls x. Insgesamt seien M Module vorhanden. Dann gilt*

$$H \leqslant max\left\{ 2 \cdot O; \max_{x=1}^{M}\{ T(x, 1) \} \right\}.$$

Beweis: Es wird gezeigt, daß aus

$$H \neq \max_{x=1}^{M}\{ T(x, 1) \}$$

die Beziehung $H \leqslant 2 \cdot O$ folgt.
Man betrachte den Prozessor i, welcher bis zum Zeitpunkt H aktiv ist. Auf i werden die Module $y_1, ..., y_k$ ausgeführt. Dabei gelte

$$s(y_1) < s(y_2) < ... < s(y_k).$$

Die Funktionsweise von „P1“ garantiert dann

$$T(y_1, 1) \geqslant T(y_2, 1) \geqslant ... \geqslant T(y_k, 1) \quad \text{und} \quad \forall j \in [1; k\text{-}1]: \; s(y_{j+1}) \geqslant T(y_j, 1).$$

Aus der Annahme

$$H \neq \max_{x=1}^{M} \{ T(x, 1) \}$$

folgt $k \geqslant 2$. Da das Modul y_k auf dem Prozessor i gestartet wird, arbeiten alle N Prozessoren mindestens bis zum Zeitpunkt $S(y_k)$. Die belegte Mindestfläche beträgt somit $N \cdot S(y_k)$. Für den Gütefaktor ergibt sich demzufolge

$$G < \frac{H}{\frac{N \cdot S(y_k)}{N}} = \frac{S(y_k) + T(y_k, 1)}{S(y_k)} = 1 + \frac{T(y_k, 1)}{S(y_k)} \leqslant 1 + \frac{T(y_{k-1}, 1)}{S(y_k)} \leqslant 1 + \frac{S(y_k)}{S(y_k)} = 2.$$

■

2.1.2 Schrittweise Parallelisierung von Modulen

Da der eben vorgestellte Algorithmus nur einen Gütefaktor größer als Zwei liefern kann, wenn das Modul mit der größten sequentiellen Laufzeit die Laufzeit H hat (siehe Lemma 4), wird dieser Fall noch genauer betrachtet. Um zu einem besseren Schedule zu gelangen, müssen die rechenzeitintensiven Module mit mehreren Prozessoren ausgeführt werden. Wie man bereits gesehen hat, bestimmt die Laufzeit dieser rechenzeitintensiven Module die Gesamtlaufzeit H. Demzufolge wählt man jeweils das Modul, welches zum Zeitpunkt H endet, aus und teilt diesem Modul einen Prozessor mehr zu. Anschließend positioniert man alle Module neu. Diese zwei Schritte (Parallelisierung eines Moduls und Positionierung aller Module) wiederholt man so oft, bis erstmalig eine Verschlechterung eintritt. Der Pseudocode des Algorithmus „P1+“ ist in der Abbildung 6 dargestellt.
Wie man aus der Abbildung 7 entnehmen kann, ist es ganz wichtig, daß der Algorithmus nicht abbricht, wenn die Gesamtlaufzeit des Schedules gleich bleibt.
Der Algorithmus „P1+“ benötigt am Anfang $O(M \cdot \log(M))$ Schritte zur Sortierung der einzelnen Module. Die beim Positionieren der Module verwendete Minimumsuche läuft in $O(N)$ (siehe Anhang 9.1). Demzufolge liegt jeder in Zeile 8) der Abbildung 6 erstellte Zwischenschedule nach jeweils $O(M \cdot N)$ Zeiteinheiten vor. Da der Algorithmus die zugeteilten Prozessoranzahlen nur erhöht, ist er spätestens nach $N \cdot M$ Positionierungen aller Module beendet. Als Gesamtlaufzeit resultiert somit $O(M \cdot \log(M) + (N \cdot M) \cdot (M \cdot N)) = O(M^2 \cdot N^2)$.
Zur weiteren Analyse des Algorithmus „P1+“ benötigt man ein neues Merkmal, welches sicher bei vielen Modulen vorhanden ist:

Definition 6: *Ein Modul x erfüllt genau dann die Flächenklausel, wenn für alle $i \in [1; N - 1]$ gilt: $T(x, i) \cdot i \leqslant T(x, i + 1) \cdot (i + 1)$.*

Die Flächenklausel impliziert, daß der Mehraufwand durch die Parallelisierung des Moduls beim Übergang von i zu i + 1 Prozessoren nicht geringer wird. Die Fläche des Moduls darf sich demzufolge bei einer Parallelisierung des Moduls nicht verkleinern (siehe auch [20]).
Man kann nachrechnen, daß die Flächenklausel eine stärkere Forderung als die Flächenbedingung (siehe Definition 4) ist. Im Gegensatz zur Flächenbedingung ist die Gültigkeit der Flächenklausel in dieser Arbeit aber keine globale Voraussetzung an alle Module. Wenn ein Modul die Flächenklausel erfüllen soll, wird es explizit gefordert.

Lemma 5: *Der Algorithmus „P1+“ garantiert einen Gütefaktor von Zwei, wenn alle zur Parallelisierung herangezogenen Module, also die Module, welche in Zeile 11) des Pseudocodes in Abbildung 6 ausgewählt werden, die Flächenklausel erfüllen.*

1) $\forall x \in [1; M]: B(x) = 1$
2) $h = \infty$
3) *sortiere die Module so, daß aus* $x < y$ *die Beziehung* $T(x) \geq T(y)$ *mit* $T(x) = T(x, B(x))$ *folgt*
4) $\forall i \in [1; N]: E(i) = 0$
5) ***für*** $x = 1$ ***bis*** M
6) *suche den Prozessor i, so daß* $\max_{j=i}^{i+B(x)-1} \{ E(j) \}$ *minimal ist (gibt es mehrere Möglichkeiten, so wähle das kleinste i), also bestimme*

$$i = \min_{k=1}^{N-B(x)+1} \left\{ k : \max_{j=k}^{k+B(x)-1} \{ E(j) \} = \min_{m=1}^{N+1-B(x)} \left\{ \max_{j=m}^{m+B(x)-1} \{ E(j) \} \right\} \right\}$$

7) $S(x) = \max_{j=i}^{i+B(x)-1} \{ E(j) \}$; $P(x) = i$; $\forall j \in [i; i + B(x) - 1]: E(j) = S(x) + T(x)$
8) $H = \max_{i=1}^{N} \{ E(i) \}$
9) ***ist*** $H > h$, ***so*** *stelle den zuletzt gespeicherten Schedule wieder her;* ***Ende***
10) $h = H$; *speichere den aktuellen Schedule*
11) *es sei x das Modul, welches bis zum Zeitpunkt* H *aktiv ist; gibt es mehrere, so wähle das, welches zuerst eingefügt wurde, also* $x = \min_{y=1}^{M} \{ y : S(y) + T(y) = H \}$
12) ***ist*** $B(x) = N$, ***so Ende***
13) $B(x) = B(x) + 1$
14) ***weiter*** *in Zeile 4)*

Abbildung 6 - Algorithmus „P1+“.

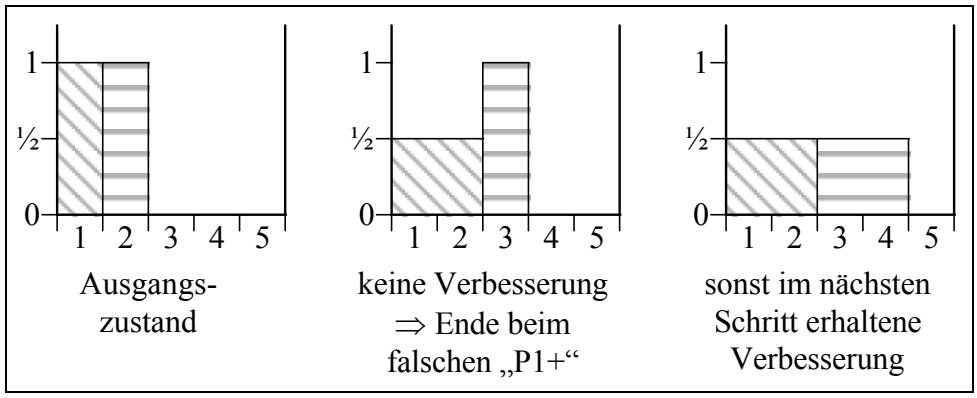


Abbildung 7 - Beispiel, bei dem „P1+“ versagen würde, wenn die Abbruchbedingung in Zeile 9) des Pseudocodes $H \geq h$ lauten würde.

Beweis: Damit man das Lemma 5 beweisen kann, muß man erst einmal eine Folgerung aus der Flächenklausel herleiten:

$$i \cdot T(x, i) \leq (i + 1) \cdot T(x, i + 1)$$

$$\Leftrightarrow \quad T(x, i) \leq \frac{i + 1}{i} \cdot T(x, i + 1)$$

$$\Rightarrow \quad T(x, i) \leq 2 \cdot T(x, i + 1). \qquad (1)$$

Außerdem benötigt man die Belegung einiger Variablen zu bestimmten Zeitpunkten des Ablaufes von „P1+“. Mit $B_a(x)$, $S_a(x)$, $P_a(x)$, $T_a(x)$ bzw. H_a werde die Belegung der Variablen $B(x)$, $S(x)$, $P(x)$, $T(x)$ bzw. H nach dem a-ten Durchlauf der Zeile 8) in Abbildung 6 notiert. Außerdem bezeichne x_a das beim a-ten Aufruf der Zeile 11) gewählte Modul x.

Es sei a das kleinste a, so daß $S_a(x_a) \neq 0$ ist. Sollte kein derartiges a existieren, so setze $a = \infty$.

Man suche das kleinste b, für welches

$$\sum_{\substack{x \in [1;\, M] \\ \text{mit } B_b(x) \geqslant 2}} B_b(x) > N$$

gilt. Mit b wird also die erste Positionierung aller Module bezeichnet, bei der nicht mehr alle Module mit einer zugeteilten Prozessoranzahl echt größer als Eins zum Zeitpunkt Null gestartet werden können. Auch hier gilt wieder $b = \infty$, wenn kein solches b existiert.

In c steht die Nummer des Positionierungsvorgangs, nach dem der Algorithmus in Zeile 9) beendet wird. Stoppt „P1+“ in Zeile 12), so ist $c = \infty$.

Fall 1: Es gelte $\min\{a; b; c\} = \infty$.

Man sieht sehr schnell, daß in diesem Fall $M = 1$ sein muß. Da der Algorithmus unter dieser Voraussetzung $B_N(1) = N$ setzt, erzielt „P1+“ den optimalen Schedule und es folgt $G = 1$.

Fall 2: Es gelte $\min\{a; b; c\} \neq \infty$.

Fall 2.1: Es sei $a = 1$.

Demzufolge werden alle Module mit einem Prozessor ausgeführt und belegen wegen der Flächenbedingung die minimale Fläche. Da alle Prozessoren bis zum Zeitpunkt $S_1(x_1)$ ausgelastet sind, beträgt die minimale Fläche aller Module $N \cdot S_1(x_1)$. Aus diesem Grund kann die Gesamtlaufzeit O des optimalen Schedules nicht kleiner als $S_1(x_1)$ sein. Da der Prozessor $P_1(x_1)$ vor x_1 noch mindestens ein Modul y mit $T_1(y) \geqslant T_1(x_1)$ ausführt (vgl. Definition von a), gilt

$$H_1 = S_1(x_1) + T_1(x_1) \leqslant S_1(x_1) + T_1(y) \leqslant 2 \cdot S_1(x_1).$$

Für den Gütefaktor folgt somit

$$G = \frac{H}{O} \leqslant \frac{H_1}{O} \leqslant 2.$$

Fall 2.2: Es sei $a \geqslant 2$.

Zuerst wird die Ungleichung $a \leqslant c$ gezeigt. Dazu muß man nachweisen, daß $S_c(x_c) \neq 0$ ist.

Aus dem Ablauf des Algorithmus folgen die Ungleichungen

$$H_c > H_{c-1} \quad \text{und} \quad \forall x \in [1; M]: T_{c-1}(x) \geqslant T_c(x).$$

Außerdem erfüllen alle Module x die Beziehung

$$H_{c-1} \geqslant T_{c-1}(x).$$

Demzufolge gilt für alle Module x:

$$H_c > T_c(x).$$

Aus der Definition von x_c folgt

$$S_c(x_c) + T_c(x_c) = H_c$$

$$\Rightarrow \qquad S_c(x_c) + T_c(x_c) > T_c(x_c)$$

$$\Rightarrow \qquad S_c(x_c) \neq 0.$$

Unter der Voraussetzung, daß der Fall 2.2 eintritt, wird der Algorithmus also nie vor der a-ten Positionierung aller Module in Zeile 9) beendet. Demzufolge ist die Gesamtlaufzeit H des erzielten Schedules nie größer als H_a.

Man setze $d = \min\{a; b\} - 1$. Dann garantiert die Wahl von d für alle $e \in [1; d]$:

$$S_e(x_e) = 0 \quad \text{und} \quad T_e(x_e) = H_e.$$

Demzufolge muß jedes Modul $y \in \{ x_e : e \in [1; d] \}$ in jedem Schedule mit einer Gesamtlaufzeit echt kleiner als H_d mit mindestens $B_{d+1}(y)$ Prozessoren ausgeführt werden. Die Flächenklausel garantiert dann, daß jeder Schedule mit einer Gesamtlaufzeit echt kleiner als H_d mindestens die folgende Fläche A belegt:

$$A = \sum_{x=1}^{M} T_{d+1}(x) \cdot B_{d+1}(x). \tag{2}$$

Sollte $O = H_d$ sein, so erzielt der Algorithmus „P1+“ den Gütefaktor Eins. Demzufolge wird nachfolgend nur der Fall $O < H_d$ betrachtet.

Fall 2.2.1: Es gelte $a < b$.

Demzufolge werden bei der a-ten Positionierung alle Module x mit $B_a(x) \geq 2$ zum Zeitpunkt Null gestartet. Da das Modul x_a nicht zum Zeitpunkt Null gestartet wird, gilt $B_a(x_a) = 1$. Außerdem gibt es keinen Prozessor, welcher zu einem Zeitpunkt t kein Modul abarbeitet und zu einem Zeitpunkt t’ mit $t' > t$ wieder ein Modul ausführt. Deshalb führen bis zum Zeitpunkt $S_a(x_a)$ alle Prozessoren ein Modul aus. Bei der a-ten Positionierung aller Module gilt also:

$$N \cdot S_a(x_a) \leq \sum_{x=1}^{M} T_a(x) \cdot B_a(x).$$

Mit der Aussage (2) gelangt man zu

$$N \cdot S_a(x_a) \leq A.$$

Aufgrund der Flächenbedingung ist dann

$$O \geq \frac{A}{N} \geq S_a(x_a). \tag{3}$$

Da $S_a(x_a) \neq 0$ und $B_a(x_a) = 1$ ist, muß vor x_a mindestens ein Modul y auf $P_a(x_a)$ ausgeführt werden.

Fall 2.2.1.1: Es sei $B_a(y) = 1$.

Da das Modul y vor x_a eingefügt wird und beide Module mit einem Prozessor ausgeführt werden, gilt $T_a(y) \geq T_a(x_a)$. Für die Gesamtlaufzeit H_a des im a-ten Schritt erzielten Schedules bedeutet das

$$H_a = S_a(x_a) + T_a(x_a) \leq S_a(x_a) + T_a(y) \leq 2 \cdot S_a(x_a).$$

Zusammen mit (3) ergibt sich für den Gütefaktor

$$G = \frac{H}{O} \leq \frac{H_a}{O} \leq 2.$$

Fall 2.2.1.2: Es gelte $B_a(y) \neq 1$.

Man bestimme das größte e, für das $B_e(y) + 1 = B_a(y)$ gilt. Da das Modul y zur Parallelisierung ausgewählt wurde, folgt $H_e = T_e(y)$. Mit Hilfe von (1) gelangt man zu $T_e(y) \leqslant 2 \cdot T_a(y)$. Außerdem ist $T_a(y) \leqslant S_a(x_a)$. Man erhält also $H_e \leqslant 2 \cdot S_a(x_a)$. Unter Verwendung der Ungleichung (3) ergibt sich

$$G = \frac{H}{O} \leqslant \frac{H_e}{O} \leqslant 2.$$

Fall 2.2.2: Es sei $a \geqslant b$.

Somit existiert für alle Module x mit $B_{b-1}(x) \geqslant 2$ ein maximales e_x, so daß $B_{e_x}(x) + 1 = B_{b-1}(x)$ ist. Wegen (1) gilt für alle x mit $B_{b-1}(x) \geqslant 2$:

$$T_{e_x}(x) \leqslant 2 \cdot T_{b-1}(x).$$

Aus dem Ablauf des Algorithmus folgt für alle x mit $B_{b-1}(x) \geqslant 2$:

$$T_{e_x}(x) \geqslant T_{e_x}(x_{b-1}).$$

Außerdem erhöht sich die Laufzeit des Moduls x_{b-1} während des Ablaufes von „P1+“ nicht, also $T_{e_x}(x_{b-1}) \geqslant T_{b-1}(x_{b-1})$. Aus den letzen drei Ungleichungen kann man für alle Module x mit $B_{b-1}(x) \geqslant 2$ auf

$$2 \cdot T_{b-1}(x) \geqslant T_{b-1}(x_{b-1})$$

schließen. Nun kann man die folgende Beziehung herleiten:

$$\sum_{x=1}^{M} T_{b-1}(x) \cdot B_{b-1}(x)$$

$$\geqslant T_{b-1}(x_{b-1}) \cdot B_{b-1}(x_{b-1}) + \sum_{\substack{x \in [1;\,M] \\ \text{mit } B_{b-1}(x) \geqslant 2 \\ \text{und } x \neq x_{b-1}}} T_{b-1}(x) \cdot B_{b-1}(x)$$

$$\geqslant T_{b-1}(x_{b-1}) \cdot B_{b-1}(x_{b-1}) + \sum_{\substack{x \in [1;\,M] \\ \text{mit } B_{b-1}(x) \geqslant 2 \\ \text{und } x \neq x_{b-1}}} \tfrac{1}{2} \cdot T_{b-1}(x_{b-1}) \cdot B_{b-1}(x)$$

$$\geqslant \tfrac{1}{2} \cdot T_{b-1}(x_{b-1}) \cdot \left(2 \cdot B_{b-1}(x_{b-1}) + \sum_{\substack{x \in [1;\,M] \\ \text{mit } B_{b-1}(x) \geqslant 2 \\ \text{und } x \neq x_{b-1}}} B_{b-1}(x) \right)$$

$$\geqslant \tfrac{1}{2} \cdot T_{b-1}(x_{b-1}) \cdot N \quad \text{(siehe Wahl von b)}$$

$$= \tfrac{1}{2} \cdot H_{b-1} \cdot N. \tag{4}$$

Die Aussage (2) entspricht

$$A = \sum_{x=1}^{M} T_b(x) \cdot B_b(x).$$

Aus der Flächenklausel und (4) folgt

$$A \geqslant \sum_{x=1}^{M} T_{b-1}(x) \cdot B_{b-1}(x) \geqslant \tfrac{1}{2} \cdot H_{b-1} \cdot N.$$

Für den Gütefaktor G ergibt sich

$$G = \frac{H}{O} \leqslant \frac{H_{b-1}}{O} \leqslant \frac{H_{b-1}}{\frac{A}{N}} \leqslant \frac{H_{b-1}}{\frac{1}{2} \cdot H_{b-1}} = 2.$$

■

Man kann erkennen, daß der beim ersten Durchlauf der Zeile 8) von „P1+“ erstellte Zwischenschedule dem von „P1“ berechneten Schedule entspricht. Aus diesem Grund liefert „P1+“ nie einen schlechteren Schedule als „P1“.
Es sei noch ein Beispiel angegeben, bei welchem „P1+“ kein zufriedenstellendes Resultat erzielt. Man nehme dazu zwei gleichartige Module die bei der Zuteilung von weniger als N Prozessoren die Laufzeit Eins haben. Für den Fall, daß ein Modul alle Prozessoren zur Verfügung hat, benötigt es nur $\frac{1}{N}$ Zeiteinheiten. Man sieht, daß der Algorithmus nie beide Module nacheinander ausführen wird, da er dazu immer eine Verschlechterung in Kauf nehmen muß. Sein ermittelter Schedule ist also um den Faktor ½ · N schlechter als der optimale Schedule.

Betrachtet man den eben notierten Beweis bezüglich der Laufzeit und läßt „P1+“ nach dem (min{a; b - 1})-ten Schritt abbrechen, so werden höchstens N - 1 Zwischenschedules erstellt. Der Algorithmus terminiert dann nach $O(M \cdot \log(M) + M \cdot N^2)$ Schritten. Ist $M \geqslant N$, so erlaubt die folgende Überlegung eine noch genauere Abschätzung der benötigten Rechenzeit. Es gibt maximal N Module, die eine Prozessoranzahl echt größer als Eins haben. Da nach dem einmaligen Aufbau eines Heaps in $O(N \cdot \log(N))$ die Startzeit eines jeden Moduls x mit $B(x) = 1$ in $O(\log(N))$ berechnet werden kann, erhält man als Algorithmuslaufzeit $O(M \cdot \log(M) + M \cdot N \cdot \log(N) + N^3)$.

Lemma 6: *Der Algorithmus „P1+“ hat unter den im Lemma 5 geforderten Voraussetzungen nach $O(M \cdot \log(M) + \min\{M \cdot N \cdot \log(N) + N^3; M \cdot N^2\})$ Schritten einen Schedule mit dem Gütefaktor von Zwei erstellt. Dabei bezeichnet wie bisher M die Anzahl der zu optimierenden Module und N die Gesamtanzahl an Prozessoren.*

2.1.3 Verschlechterungen der Zwischenschedules zulassen

Das Beispiel aus dem letzten Abschnitt zeigt sehr schön, daß ein Schedule mit einer zufriedenstellenden Gesamtlaufzeit in einigen verbleibenden Fällen nur gefunden werden kann, indem Verschlechterungen zugelassen werden. Eine mögliche Vorgehensweise wäre, den Algorithmus erst nach einer vorgegebenen Anzahl von Verschlechterungen nach der Berechnung des bisher besten ermittelten Schedules abbrechen zu lassen. Da bei dem vorgestellten Parallelisierungsverfahren sowieso nach M · N Schritten keine Modulmanipulation mehr möglich ist, braucht man den Algorithmus nur bis zum absoluten Ende durchlaufen lassen. (siehe auch Erklärung von ABBRUCH in Abschnitt 9.11)
Eine andere Verfahrensweise besteht darin, daß man nach i schlechteren Schedules wieder zum bisher besten Zwischenschedule zurückspringt und im weiteren Verlauf des Algorithmus das zuvor im nächsten Schritt parallelisierte Modul unbetrachtet läßt. Dieses Vorgehen ist jedoch sehr laufzeitintensiv.

2.1.4 Strategien zur Positionierung aller Module

In diesem Abschnitt wird das sogenannte Rechteck - Füll - Problem, welches ein Spezialfall der hier vorliegenden Aufgabenstellung ist, behandelt. Aus diesem Grund sind im ganzen Abschnitt 2.1.4 die Prozessoranzahlen $B(x)$ und Laufzeiten $T(x)$ aller Module fest vorgegeben und nicht variabel.

2.1.4.1 Sortierung nach der Modullaufzeit

Die an dieser Stelle vorgestellte Grundidee wurde aus [17] übernommen.
Man sortiere die Module vor dem Positionierungsvorgang absteigend nach ihrer aktuellen Laufzeit. Das Positionieren aller Module läuft ebenenweise ab: Die unterste Ebene beginnt zum Zeitpunkt 0. In diese Ebene werden die Module mit den größten Laufzeiten nebeneinander eingefügt, bis die Ebene voll ist. Eine Ebene heißt voll, wenn das Modul mit der nächstkleineren Laufzeit mehr Prozessoren benötigt, als in der Ebene noch zur Verfügung stehen. In diesem Fall wird eine neue Ebene eröffnet. Die Startzeit dieser neuen Ebene ist die aktuelle maximale Endzeit aller Prozessoren. Demzufolge haben alle Module, die zu einer Ebene gehören, die gleiche Startzeit. Der Pseudocode des Algorithmus „HMZ“ ist in der Abbildung 8 dargestellt.

```
1)  akt_start = 0; akt_proz = 0; ebenenhöhe = max_{x=1}^{M} { T(x) }
2)  sortiere die Module so, daß aus x < y die Beziehung T(x) ⩾ T(y) folgt
3)  für x = 1 bis M
4)      ist akt_proz + B(x) > N (Ebene voll), so akt_start = akt_start + ebenenhöhe;
        ebenenhöhe = T(x) und akt_proz = 0 (neue Ebene beginnen)
5)      führe x in der aktuellen Ebene aus, also P(x) = akt_proz + 1; S(x) = akt_start;
        akt_proz = akt_proz + B(x)
```

Abbildung 8 - Algorithmus „HMZ“.

Die Laufzeit dieses Algorithmus liegt in $O(M \cdot \log(M))$.
Die Abbildung 9 stellt eine Positionierung des Algorithmus „HMZ“ dar.

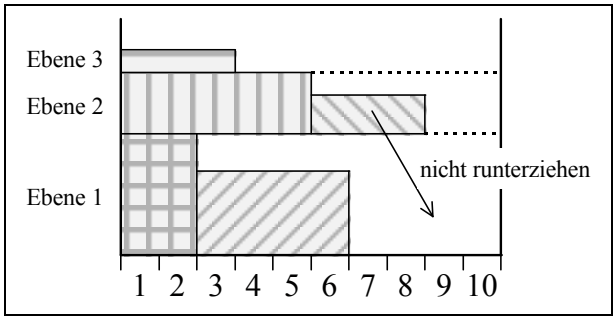


Abbildung 9 - Positionierung von „HMZ“.

Lemma 7: *Es bezeichne H die vom Algorithmus „HMZ“ erzielte Gesamtlaufzeit bei der Positionierung von M Modulen auf N Prozessoren. Wie bisher stehe $T(x)$ für die Laufzeit und $B(x)$ für die Prozessoranzahl des Moduls x. Außerdem sei das Modul 1 das Modul mit der größten Laufzeit. Dann gilt:*

$$\sum_{x=1}^{M} T(x) \cdot B(x) > \tfrac{1}{2} \cdot N \cdot (H - T(1)).$$

Beweis (vgl. auch [17]): Man sieht sehr schön, daß alle Module einer Ebene die gleiche Anfangszeit haben und die Ebenenlaufzeit der Laufzeit des Moduls entspricht, welches in der Ebene auf dem Prozessor 1 ausgeführt wird, d.h., $T(1)$ beschreibt die Laufzeit der untersten Ebene. Außerdem haben alle Module der i-ten Ebene entweder die gleiche oder eine größere Laufzeit als das Modul, welches in der (i + 1)-ten Ebene auf dem Prozessor 1 ausgeführt wird. Betrachtet man nun die Summe der Prozessoranzahlen der Module zweier aufeinander folgender Ebenen, so ist die Summe echt größer als N. Wäre das nicht der Fall, so würde der Algorithmus auch alle in beiden Ebenen enthaltenen Module in einer Ebene unterbringen.
Aus diesem Grund kann man, falls in der Ebene i (außer der obersten Ebene) nur $½ \cdot N - z$ Prozessoren ($z \geqslant 1$) zur Ausführung von Modulen benutzt werden, im Gantt - Diagramm die folgende flächenerhaltende Umformung durchführen: Man trennt vom Rechteck, welches das erste Modul in der (i + 1)-ten Ebene repräsentiert, genau z Spalten ab, und fügt sie mit in die i-te Ebene ein. Da die Summe der genutzten Prozessoren zweier aufeinander liegender Ebenen größer als N war, folgt, daß auch in der (i + 1)-ten Ebene noch mindestens $½ \cdot N$ Prozessoren in Benutzung sind (sofern es sich bei der Ebene i + 1 nicht um die oberste Ebene handelte). Nach dieser Konstruktion sieht man, daß in der i-ten Ebene (außer der obersten) mindestens die folgende Fläche belegt ist:

$$½ \cdot N \cdot \text{Höhe\_der\_(i + 1)-ten\_Ebene}.$$

Addiert man nun die Mindestflächen der einzelnen Ebenen, so ergibt sich $½ \cdot N \cdot (H - T(1))$. Man erhält also

$$\sum_{x=1}^{M} T(x) \cdot B(x) \geqslant ½ \cdot N \cdot (H - T(1)).$$

Das im Lemma angegebene „echt größer“ kann durch die Unterscheidung der folgenden beiden Fälle begründet werden:
Fall 1: Es gibt nur eine Ebene.
Hat die von „HMZ“ erstellte Positionierung nur eine Ebene, so gilt $H = T(1)$. Demzufolge wird der rechte Term der Ungleichung Null.
Fall 2: Die Module sind auf mindestens zwei Ebenen verteilt.
In diesem Fall gibt es eine oberste Ebene, deren Restfläche auch nach dem eventuellen Auffüllen der unter ihr liegenden Ebene ungleich Null ist. Diese Fläche wurde bei der Bestimmung der Mindestfläche für die optimale Positionierung nicht mit berücksichtigt, d.h., die Mindestfläche der optimalen Positionierung ist echt größer als bei der Herleitung der Beziehung angenommen wurde.

■

In der Abbildung 10 ist das eben beschriebene Vorgehen noch einmal dargestellt.

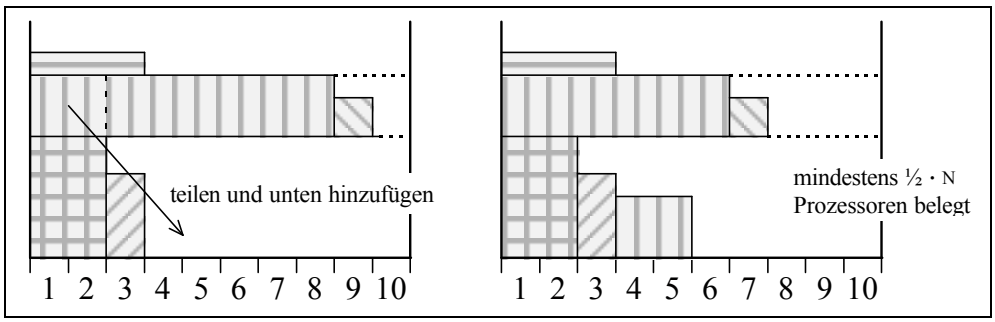


Abbildung 10 - Beweisveranschaulichung.

Lemma 8: *Es sei A ein Verfahren zur näherungsweisen Lösung des Rechteck - Füll - Problems. Die von A erzielte Positionierung aller M Module auf N Prozessoren habe die Gesamtlaufzeit H. Mit O werde die Gesamtlaufzeit der optimalen Positionierung bezeichnet. Wie bisher stehe T(x) für die Laufzeit und B(x) für die Prozessoranzahl des Moduls x. Das Modul mit der Nummer 1 sei das Modul mit der größten Laufzeit.*
Kann man für das Verfahren A die Ungleichung

$$\sum_{x=1}^{M} T(x) \cdot B(x) > \tfrac{1}{2} \cdot N \cdot (H - T(1)) \qquad (1)$$

zeigen, so gelten für das Verfahren A die folgenden zwei Ungleichungen:

$$O > \tfrac{1}{2} \cdot (H - T(1)) \quad \textit{und} \quad \tfrac{H}{O} = G < 3.$$

Beweis:
Fall 1: Es gelte $3 \cdot T(1) > H$.
Da das Modul mit der größten Laufzeit auch in der optimalen Positionierung enthalten ist, folgt für die Gesamtlaufzeit O der optimalen Positionierung $O \geqslant T(1)$. Durch Einsetzen in die Fallvoraussetzung ergibt sich:

$$3 \cdot O > H \quad \text{bzw.} \quad G = \frac{H}{O} < 3.$$

Aus der Gültigkeit von

$$\frac{1}{2} \cdot T(1) > \frac{1}{6} \cdot H \quad \text{und} \quad O > \frac{1}{3} \cdot H$$

kann man durch eine Addition

$$\frac{1}{2} \cdot T(1) + O > \frac{1}{6} \cdot H + \frac{1}{3} \cdot H \quad \Leftrightarrow \quad O > \tfrac{1}{2} \cdot (H - T(1))$$

herleiten.
Fall 2: Es sei $3 \cdot T(1) \leqslant H$.
Da in der optimalen Positionierung alle Module enthalten sein müssen, folgt

$$O \geqslant \frac{1}{N} \cdot \sum_{x=1}^{M} T(x) \cdot B(x).$$

Zusammen mit (1) erhält man $O > \tfrac{1}{2} \cdot (H - T(1))$. Verwendet man diese Ungleichung zum Abschätzen des Gütefaktors, so erhält man

$$G = \frac{H}{O} < \frac{H}{\tfrac{1}{2} \cdot (H - T(1))} = \frac{2 \cdot H}{H - T(1)}.$$

Mit Hilfe der Fallvoraussetzung läßt sich dieser Term durch

$$G < \frac{2 \cdot H}{H - \frac{H}{3}} = 3$$

von oben abschätzen.

■

Im Abschnitt 9.2 wird ein Beispiel angegeben, bei dem „HMZ“ tatsächlich nur den bewiesenen Gütefaktor von Drei erreicht.

2.1.4.2 Sortierung nach der Prozessoranzahl der Module

Der nächste Positionierungsalgorithmus „BMZ“ sortiert die einzelnen Module absteigend nach der benötigten Prozessoranzahl. Sollten zwei Module die gleiche Prozessoranzahl benö-

tigen, so können sie in beliebiger Reihenfolge stehen. Im Gegensatz zum eben vorgestellten Algorithmus „HMZ“ führt „BMZ“ wieder jedes Modul auf der Prozessorgruppe mit der kleinsten Endzeit aus. Sollte es dabei mehrere Möglichkeiten geben, so werden die Prozessoren mit den kleinsten Nummern gewählt. Der Pseudocode des Algorithmus „BMZ“ ist in der Abbildung 11 dargestellt.
Als Laufzeit resultiert hier $O(M \cdot N)$.

1) $\forall i \in [1; N]: E(i) = 0$
2) ***für*** $b = N$ ***bis*** *1 rückwärts zählend*
3) ***für*** *alle Module x mit der Prozessoranzahl b* $(B(x) = b)$
4) $$i = \min_{k=1}^{N-b+1} \left\{ k : \max_{j=k}^{k+b-1} \{ E(j) \} = \min_{m=1}^{N+1-b} \left\{ \max_{j=m}^{m+b-1} \{ E(j) \} \right\} \right\}$$
5) $$S(x) = \max_{j=i}^{i+b-1} \{ E(j) \};\ P(x) = i;\ \forall j \in [i; i+b-1]: E(j) = S(x) + T(x)$$

Abbildung 11 - Algorithmus „BMZ“.

Lemma 9: *Es bezeichne* H *für die vom Algorithmus „BMZ“ erzielte Gesamtlaufzeit bei der Positionierung von* M *Modulen auf* N *Prozessoren. Wie bisher stehe* $T(x)$ *für die Laufzeit und* $B(x)$ *für die Prozessoranzahl des Moduls x. Außerdem habe das Modul mit der Nummer 1 die größte Laufzeit. Dann gilt*

$$\sum_{x=1}^{M} T(x) \cdot B(x) > \tfrac{1}{2} \cdot N \cdot (H - T(1)).$$

Beweis:

Fall 1: Es gelte $H \cdot N \leqslant 2 \cdot \sum_{x=1}^{M} T(x) \cdot B(x)$.
Die Behauptung folgt durch Umstellen der Fallvoraussetzung.

Fall 2: Es sei $H \cdot N > 2 \cdot \sum_{x=1}^{M} T(x) \cdot B(x)$.
In diesem Fall liegt die durchschnittliche Auslastung der einzelnen Prozessoren echt unter 50%, d.h., im Mittel verbringt jeder Prozessor mehr Zeit ohne als mit Berechnungen. Demzufolge existiert ein frühester Zeitpunkt t mit $t < H$, zu welchem echt weniger als $\frac{1}{2} \cdot N$ Prozessoren aktiv sind.
Als nächstes wird gezeigt, daß im Intervall $(t; H]$ keine neuen Module mehr gestartet werden. Angenommen, es gibt ein Modul x, welches nach dem Zeitpunkt t gestartet wird. Dies ist nur möglich, wenn es zum Zeitpunkt t keine $B(x)$ nebeneinander liegenden Prozessoren gibt, welche kein Modul ausführen. Zum Zeitpunkt t seien die Module $y_1, ..., y_k$ aktiv.

Fall 2.a: Auf dem Prozessor 1 werde ein Modul ausgeführt.
Ohne Beschränkung der Allgemeinheit gelte $B(y_1) \leqslant B(y_i)\ \forall i \in [1; k]$. Daraus folgt

$$k \cdot B(y_1) \leqslant \sum_{i=1}^{k} B(y_i).$$

Weil die k Module weniger als die Hälfte der Prozessoren benutzen, muß

$$\sum_{i=1}^{k} B(y_i) < \tfrac{1}{2} \cdot N$$

sein. Außerdem weiß man noch, daß die Module absteigend nach der benötigten Prozessoranzahl sortiert sind, woraus $B(x) \leqslant B(y_1)$ resultiert. Aus den letzten drei Ungleichungen erhält man

$$k \cdot B(x) < \tfrac{1}{2} \cdot N \quad \text{bzw.} \quad B(x) < \frac{N}{2 \cdot k}.$$

Jetzt bleibt nur noch zu zeigen, daß es zum Zeitpunkt t eine Lücke von mindestens $\left\lfloor \frac{N}{2 \cdot k} \right\rfloor$ nebeneinander liegende Prozessoren gibt, welche kein Modul ausführen. Diese Prozessoren könnten sonst das Modul x bereits zum Zeitpunkt t ausführen. Da ein Modul auf dem Prozessor 1 ausgeführt wird, existieren zum Zeitpunkt t höchstens k Lücken. Die Summe aller Prozessoren, welche zum Zeitpunkt t kein Modul ausführen, war nach Voraussetzung mindestens $\left\lceil \frac{N}{2} \right\rceil$. Daraus läßt sich eine durchschnittliche Prozessoranzahl pro Lücke größer als $\left\lfloor \frac{N}{2 \cdot k} \right\rfloor$ ableiten. Aus diesem Grund muß sich auch eine Lücke über mindestens $\left\lfloor \frac{N}{2 \cdot k} \right\rfloor$ Prozessoren erstrecken. Somit wird in diesem Fall kein Modul nach t gestartet.

Fall 2.b: Auf dem Prozessor 1 werde kein Modul ausgeführt.

Es sei P die Menge aller zum Zeitpunkt t aktiven Prozessoren. Mit z bezeichne man das Modul, welches zum Zeitpunkt t unter anderem auf dem Prozessor min(P) ausgeführt wird. Man erkennt, daß $P(z) = \min(P)$ ist. Im Gantt - Diagramm ist z das zum Zeitpunkt t am weitesten links stehende Modul. Man betrachte nun den Startzeitpunkt von z. Da z nicht den Prozessor 1 nutzt, muß ein Modul y existieren, welches auf $B(y)$ benachbarten Prozessoren mit Prozessornummern aus der Menge $\{1; 2; ...; P(z) - 1\}$ ausgeführt wird. Im Gantt - Diagramm befindet sich y also links von z. Demzufolge gilt $B(y) \leqslant P(z) - 1$. Außerdem nimmt die Prozessoranzahl der Module im Laufe des Positionierungsvorgangs höchstens ab. Aus diesem Grund gilt $B(x) \leqslant B(y)$ und somit auch $B(x) \leqslant P(z) - 1$. Demzufolge kann auch im Fall 2.b das angenommene Modul x zum Zeitpunkt t gestartet werden. Damit ist die Zwischenbehauptung gezeigt.

Da nach dem Zeitpunkt t keine Module mehr gestartet werden, gilt

$$t + T(1) \geqslant H.$$

Fall 2.1: Es sei $t + T(1) > H$.

Aus der Definition von t folgt, daß die Mindestfläche aller Module nicht kleiner als $\tfrac{1}{2} \cdot N \cdot t$ ist:

$$\sum_{x=1}^{M} T(x) \cdot B(x) \geqslant \tfrac{1}{2} \cdot N \cdot t > \tfrac{1}{2} \cdot N \cdot (H - T(1)).$$

Fall 2.2: Es gelte $t + T(1) = H$.

In diesem Fall beträgt die belegte Mindestfläche

$$\sum_{x=1}^{M} T(x) \cdot B(x) \geqslant \tfrac{1}{2} \cdot N \cdot t + T(1) \cdot B(1) > \tfrac{1}{2} \cdot N \cdot t = \tfrac{1}{2} \cdot N \cdot (H - T(1)).$$

■

Mit Hilfe von Lemma 8 gelangt man auch für „BMZ“ zu $G < 3$.

Ein Beispiel, bei welchem „BMZ“ nur einen Gütefaktor von rund 2,69 erreicht, findet man im Anhang 9.3.

2.1.4.3 Sortierung nach der Modulfläche

Nachdem in den letzten zwei Abschnitten das Modul mit der größten Laufzeit bzw. mit der größten Prozessoranzahl zuerst eingefügt wurde, könnte man auf die Idee kommen, es auch in der durch die Flächen der Module vorgegebenen Reihenfolge zu versuchen. Das im Anhang 9.4 angegebene Beispiel zeigt, daß für dieses Vorgehen kein konstanter Gütefaktor bewiesen werden kann.

2.1.4.4 Weitere Algorithmen für das Rechteck - Füll - Problem

Das Verfahren „BHMZ“, welches auf der Kombination von „BMZ“ und „HMZ“ beruht, findet man im Anhang 9.5. Außerdem sind im Anhang 9.6 noch einige Verbesserungsvorschläge zu den in diesem Abschnitt vorgestellten Algorithmen notiert.
Einen interessanten rekursiven Algorithmus mit einem Gütefaktor von Zwei und einer Laufzeit von $O\left(\frac{M \cdot \log^2(M)}{\log(\log(M))}\right)$ findet man in [29]. Leider berechnet er am Anfang die Gesamtlaufzeit H mittels

$$H = \max\left\{ t; \frac{2 \cdot A}{N} + \max\left\{ 2 - \frac{N}{b}; 0 \right\} \cdot \max\left\{ t - \frac{A}{N}; 0 \right\} \right\},$$

wobei

$$A = \sum_{x=1}^{M} T(x) \cdot B(x), \quad t = \max_{x=1}^{M}\{ T(x) \} \quad \text{und} \quad b = \max_{x=1}^{M}\{ B(x) \}$$

ist. Dadurch gilt natürlich immer $H \geqslant \frac{2 \cdot A}{N}$, so daß von diesem Algorithmus nur im Fall $H \approx t$ ein Gütefaktor in der Nähe von Eins erreicht wird. Gilt $H \geqslant 2 \cdot t$, so erreicht der Algorithmus nur einen Gütefaktor von rund Zwei.
In [32] findet man einen Algorithmus, welcher unter der Annahme, daß die Zuordnung der Module zu den Prozessoren fest ist, in $O(M \cdot (\log(M) + \log(N)))$ ein Ergebnis mit einem Gütefaktor von maximal $3 \cdot \sqrt{M}$ berechnet.
Der Einsatz von Zufallszahlen zur Bestimmung einer Packordnung wird anhand von Beispielen in [33] untersucht. Alle dort vorgestellten Verfahren geben immer das beste innerhalb einer Zeitspanne ermittelte Ergebnis zurück.

Damit sind die Betrachtungen zum Rechteck - Füll - Problem abgeschlossen. In den nachfolgenden Abschnitten können die Module demzufolge wieder mit einer beliebigen Prozessoranzahl gestartet werden.

2.1.5 Schrittweise Erhöhung der Gesamtfläche

Im Anhang 9.7 werden erste Überlegungen zur Auswahl von Modulen, die für eine Parallelisierung in Frage kommen, gemacht. Die dort erläuterten schrittweisen Parallelisierungsheuristiken garantieren aber im Fall, daß die Flächenklausel nicht erfüllt ist, keinen zufriedenstellenden Gütefaktor. Auch ist die Erhöhung der Prozessoranzahl eines jeden Moduls um Eins sehr laufzeitintensiv. Zur Minimierung der Anzahl der Positionierungsvorgänge muß man demzufolge auch größere Schritte zulassen.
Bis jetzt wurden die Module einmal am Anfang sortiert und dann, auch nach Parallelisierungen, in der anfangs bestimmten Reihenfolge positioniert. Daß dieses Vorgehen nicht günstig ist, zeigt das im Anhang 9.8 dargestellte Beispiel.

Der neue Algorithmus „P1+MF“ beachtet die oben aufgeführten Erkenntnisse. Er läuft wie folgt ab:

1) $\forall x \in [1; M]: B(x) = 1$
2) $\forall i \in [1; N]: E(i) = 0$
3) *sortiere die Module so, daß aus* $x < y$ *die Beziehung* $T(x, 1) \geqslant T(y, 1)$ *folgt*
4) ***für*** $x = 1$ ***bis*** M
5) $i = \min_{j=1}^{N} \left\{ j : E(j) = \min_{k=1}^{N} \{ E(k) \} \right\}$
6) $S(x) = E(i); P(x) = i; E(i) = E(i) + T(x, 1)$
7) $H = \max_{i=1}^{N} \{ E(i) \}$
8) ***ist*** $H > \max_{x=1}^{M} \{ T(x, 1) \}$, ***so Ende***
9) $h = H$; *speichere den aktuellen Schedule*
10) $x = \min_{y=1}^{M} \{ y : T(y) = H \}$
11) ***ist*** $B(x) = N$, ***so Ende***
12) $B(x) = B(x) + 1$
13) $\forall i \in [1; N]: E(i) = 0$
14) ***für*** $x = 1$ ***bis*** M
15) $i = \min_{k=1}^{N - B(x) + 1} \left\{ k : \max_{j=k}^{k + B(x) - 1} \{ E(j) \} = \min_{m=1}^{N + 1 - B(x)} \left\{ \max_{j=m}^{m + B(x) - 1} \{ E(j) \} \right\} \right\}$
16) $S(x) = \max_{j=i}^{i + B(x) - 1} \{ E(j) \}; P(x) = i; \ \forall j \in [i; i + B(x) - 1]: E(j) = S(x) + T(x)$
17) $H = \max_{i=1}^{N} \{ E(i) \}$
18) ***ist*** $H < h$, ***so*** $h = H$; *speichere den aktuellen Schedule*
19) ***ist*** $H = \max_{x=1}^{M} \{ T(x) \}$, ***so weiter*** *in Zeile 10)*
20) ***für alle*** *Module x mit* $B(x) \geqslant 2$ ***tue***
21) $B(x) = \min_{i=B(x)}^{N} \left\{ i : i \cdot T(x,i) = \min_{j=B(x)}^{N} \{ j \cdot T(x, j) \} \right\}$.
22) *positioniere die Module mittels „BMZ“ oder „HMZ“; die Positionierung habe die Gesamtlaufzeit* H
23) ***ist*** $H < h$, ***so*** $h = H$; *speichere den aktuellen Schedule*
24) ***ist*** $N \cdot H < 3 \cdot \sum_{x=1}^{M} T(x) \cdot B(x)$, ***so*** *stelle den zuletzt gespeicherten Schedule wieder her;* ***Ende***
25) $x = \min_{y=1}^{M} \left\{ y : T(y) = \max_{z=1}^{M} \{ T(z) \} \right\}$
26) ***ist*** $B(x) = N$, ***so*** *stelle den zuletzt gespeicherten Schedule wieder her;* ***Ende***
27) $B(x) = \min_{i=B(x)+1}^{N} \left\{ i : i \cdot T(x,i) = \min_{j=B(x)+1}^{N} \{ j \cdot T(x, j) \} \right\}$.
28) ***weiter*** *in Zeile 22)*

Abbildung 12 - Algorithmus „P1+MF“.

Als ersten Schritt führt „P1+MF“ den Algorithmus „P1“ aus. Kann für das von „P1“ erstellte Scheduling bereits einen Gütefaktor von Zwei garantiert werden, so wird „P1+MF“ beendet. Anderenfalls führt „P1+MF“ den Algorithmus „P1+“ solange aus, bis das zur Parallelisierung ausgewählte Modul nicht mehr zum Zeitpunkt Null beginnt. Erfüllen alle Module die Flächenklausel, so hat „P1+MF“ an dieser Stelle einen Zwischenschedule mit einem Gütefaktor von maximal Zwei erstellt (siehe Lemma 5). Da „P1+MF“ keine Überprüfung auf die Gültigkeit der Flächenklausel vornimmt, wird der Algorithmus an dieser Stelle nie beendet. Im weiteren Verlauf ordnet „P1+MF“ jedem Modul so viele Prozessoren zu, daß die Fläche eines jeden Moduls möglichst klein ist. Anschließend positioniert „P1+MF“ alle Module mit einem Algorithmus zur näherungsweisen Lösung des Rechteck - Füll - Problems. Solange für den erstellten Zwischenschedule kein maximaler Gütefaktor von Drei garantiert werden kann, parallelisiert „P1+MF“ das Modul mit der aktuell größten Laufzeit flächenminimal. Ein Modul wird flächenminimal parallelisiert, indem man diesem Modul so viele Prozessoren mehr zuordnet, so daß die von diesem Modul belegte Fläche nur minimal größer wird. Anschließend werden wieder alle Module positioniert. Der Pseudocode des Algorithmus „P1+MF“ ist in der Abbildung 12 dargestellt.

Lemma 10: *Der vom Algorithmus „P1+MF“ erreichte Gütefaktor ist echt kleiner als Drei.*

Beweis: Wie im Beweis zu Lemma 5 bezeichne man die Belegung der Variablen $T(x)$, $B(x)$, $S(x)$, $P(x)$ bzw. H nach der a-ten Positionierung aller Module mit $T_a(x)$, $B_a(x)$, $S_a(x)$, $P_a(x)$ bzw. H_a. Im Pseudocode liegt an drei Stellen ein Zwischenschedule vor, nämlich in den Zeilen 8), 18) und 23). Es sei a die kleinste Zahl, so daß kein Modul x mit $T_a(x) = H_a$ existiert, d.h., nach der a-ten Positionierung aller Module bricht „P1+MF“ in der Zeile 8) ab oder springt von Zeile 19) nicht zur Zeile 10) zurück.
Fall 1: Es sei $a = 1$.
In diesem Fall wird der Algorithmus in Zeile 8) beendet und das Lemma 4 garantiert einen Gütefaktor von Zwei.
Fall 2: Es gelte $a \geqslant 2$.
Da für alle $b \in [1; a - 1]$ ein Modul x_b mit $T_b(x_b) = H_b$ existiert und nur dieses Modul im nächsten Positionierungsversuch mit einem Prozessor mehr ausgeführt wird, kann ein Schedule mit einer kleineren Gesamtlaufzeit als H_{a-1} nur erzielt werden, wenn alle Module x mit mindestens $B_a(x)$ Prozessoren gestartet werden. Angenommen für die Gesamtlaufzeit O des optimalen Schedules gilt $O < H_{a-1}$. Dann folgt für die Prozessoranzahlen $B^{opt}(x)$ im optimalen Schedule:

$$B^{opt}(x) \geqslant B_a(x). \tag{1}$$

Die Zeile 21) im Pseudocode garantiert, daß für jedes Modul x der Term $B_{a+1}(x) \cdot T_{a+1}(x)$ minimal ist. Dabei muß die Nebenbedingung $B_{a+1}(x) \geqslant B_a(x)$ gelten. Wegen (1) erhält man für alle Module x

$$B_{a+1}(x) \cdot T_{a+1}(x) = \min_{i = B_a(x)}^{N} \{ i \cdot T(x, i) \} \leqslant B^{opt}(x) \cdot T^{opt}(x).$$

Die soeben erstellte Ungleichung ist mit $b = a + 1$ der Induktionsanfang für die nachfolgende Induktion:
Es wird gezeigt, daß aus

$$\forall x \in [1; M]: \; B_b(x) \cdot T_b(x) \leqslant B^{opt}(x) \cdot T^{opt}(x) \qquad (2)$$

entweder

(α) „P1+MF“ hat einen Schedule mit einem Gütefaktor echt kleiner als Drei erstellt.

oder

(β) $\forall x \in [1; M]: \; B_{b+1}(x) \cdot T_{b+1}(x) \leqslant B^{opt}(x) \cdot T^{opt}(x)$ und $\exists y \in [1; M]$ mit $B_{b+1}(y) > B_b(y)$

folgt.

Da beim Eintreten von (β) der Algorithmus „P1+MF“ in Zeile 27) die Prozessoranzahl eines Moduls um mindestens Eins erhöht, kann (β) nur endlich oft eintreten. Demzufolge muß irgendwann (α) vorliegen.

Fall 2.1: Es gelte $N \cdot H_b < 3 \cdot \sum_{x=1}^{M} T_b(x) \cdot B_b(x)$.

Mit Hilfe von (2) erhält man

$$N \cdot H_b < 3 \cdot \sum_{x=1}^{M} T^{opt}(x) \cdot B^{opt}(x).$$

Außerdem gilt

$$\frac{1}{N} \cdot \sum_{x=1}^{M} T^{opt}(x) \cdot B^{opt}(x) \leqslant O.$$

Da der Algorithmus den besten ermittelten Zwischenschedule zurück liefert, folgt

$$H_b < 3 \cdot O \quad \text{bzw.} \quad G < 3.$$

Fall 2.2: Es gelte $N \cdot H_b \geqslant 3 \cdot \sum_{x=1}^{M} T_b(x) \cdot B_b(x)$.

Da $b \geqslant a + 1$ ist, wurden die Module in Zeile 22) mit dem Algorithmus „BMZ“ oder „HMZ“ positioniert. Aus Lemma 7 bzw. Lemma 9 folgt für beide Algorithmen

$$\sum_{x=1}^{M} T_b(x) \cdot B_b(x) > ½ \cdot N \cdot \left(H_b - T_b(y)\right).$$

Dabei ist y ein beliebiges Modul, für welches

$$T_b(y) = \max_{x=1}^{M} \{ T_b(x) \}$$

gilt. Mit Hilfe der letzten beiden Ungleichungen gelangt man zu

$$N \cdot H_b > \frac{3}{2} \cdot N \cdot \left(H_b - T_b(y)\right) \quad \Leftrightarrow \quad 3 \cdot T_b(y) > H_b. \qquad (3)$$

Fall 2.2.1: Es sei $3 \cdot O \leqslant H_b$.

Der Algorithmus hat also beim b-ten Positionierungsversuch noch keinen Gütefaktor $G < 3$ erzielt. Aus der Fallvoraussetzung und (3) erhält man

$$T_b(y) > O. \qquad (4)$$

Demzufolge wird das Modul y im optimalen Schedule mit echt mehr als $B_b(y)$ Prozessoren ausgeführt:

$$B_b(y) + 1 \leqslant B^{opt}(y).$$

Fall 2.2.1.1: Es gelte $B_b(y) < N$.

Die Wahl von $B_{b+1}(y)$ in Zeile 27) garantiert wiederum

$$B_{b+1}(y) \cdot T_{b+1}(y) \leqslant B^{opt}(y) \cdot T^{opt}(y).$$

Da die Prozessoranzahlen aller anderen Module unverändert bleiben, ist die Gültigkeit von (2) weiterhin gewährleistet.

Fall 2.2.1.2: Es sei $B_b(y) = N$.

Daß dieser Fall kann nicht eintreten kann, zeigt der folgende Widerspruch: Da das Modul y bereits mit allen N Prozessoren ausgeführt wird und im optimalen Schedule enthalten sein muß, gilt garantiert $T_b(y) \leqslant O$. Dies ist aber ein Widerspruch zur zuvor gewonnenen Aussage (4).

Fall 2.2.2: Es gilt $3 \cdot O > H$.

In diesem Fall hat der Algorithmus bereits einen Gütefaktor $G < 3$ erreicht, d.h., eigentlich könnte „P1+MF“ beendet werden. Da der Algorithmus nur eine untere Schranke für die Gesamtlaufzeit des optimalen Schedules verwendet, kann „P1+MF“ nicht exakt entscheiden, ob der erstellte Schedule schon einen Gütefaktor von Drei hat. Allerdings ist der Algorithmus so aufgebaut, daß er am Ende den besten ermittelten Zwischenschedule zurück gibt. Demzufolge verschlechtert sich der Endschedule nicht, wenn der Algorithmus wie im Fall 2.2.1 beschrieben weiter arbeitet.

■

Die Laufzeit von „P1+MF“ setzt sich aus der des verkürzten Algorithmus „P1+“ (siehe Lemma 6) $\left(O\left(M \cdot \log(M) + \min\left\{ M \cdot N \cdot \log(N) + N^3; M \cdot N^2 \right\}\right)\right)$, der Berechnung des Flächenminimums eines jeden bereits mit mehr als einem Prozessor ausgeführten Moduls $\left(O\left(N^2\right)\right)$ und der Laufzeit der unteren Schleife ab Zeile 22) zusammen.

Lemma 11: *Es bezeichne N die Gesamtanzahl der Prozessoren und M die Gesamtanzahl der Module. Mit $T(x, 1)$ werde die sequentielle Laufzeit des Moduls x notiert. Dann werden in Zeile 27) des Algorithmus „P1+MF“ (siehe Abbildung 12) nur Module z parallelisiert, für die*

$$N \cdot T(z, 1) > \sum_{x=1}^{M} T(x, 1)$$

gilt.

Beweis: Es wird angenommen, daß ein Modul z mit

$$N \cdot T(z, 1) \leqslant \sum_{x=1}^{M} T(x, 1)$$

in Zeile 27) parallelisiert wird. Aufgrund der Flächenbedingung gilt für jeden Positionierungsversuch b

$$N \cdot T(z, 1) \leqslant \sum_{x=1}^{M} T(x, 1) \leqslant \sum_{x=1}^{M} T_b(x) \cdot B_b(x). \qquad (5)$$

Da das Modul z zur Parallelisierung ausgewählt wurde, gibt es einen Durchlauf b der Schleife ab Zeile 22) mit

$$T_b(z) = \max_{x=1}^{M} \left\{ T_b(x) \right\}.$$

Demzufolge entspricht Lemma 7 bzw. Lemma 9 der Aussage

$$\sum_{x=1}^{M} T_b(x) \cdot B_b(x) > \tfrac{1}{2} \cdot N \cdot \left(H - T_b(z)\right) \quad \Leftrightarrow \quad N \cdot H - 2 \cdot \sum_{x=1}^{M} T_b(x) \cdot B_b(x) < N \cdot T_b(z).$$

Wegen $T_b(z) \leqslant T(z, 1)$ kann man mit Hilfe von (5) auf

$$N \cdot H - 2 \cdot \sum_{x=1}^{M} T_b(x) \cdot B_b(x) < \sum_{x=1}^{M} T_b(x) \cdot B_b(x) \quad \Leftrightarrow \quad N \cdot H < 3 \cdot \sum_{x=1}^{M} T_b(x) \cdot B_b(x)$$

schließen. Demzufolge würde der Algorithmus „P1+MF“ vor der Wahl von z in Zeile 24) abbrechen.

■

Lemma 12: *Es bezeichne N die Gesamtanzahl der Prozessoren und M die Gesamtanzahl der Module. Dann werden in der Zeile 27) des Algorithmus „P1+MF“ (siehe Abbildung 12) maximal $\min\{N; M\}$ verschiedene Module zur Parallelisierung ausgewählt.*

Beweis: Sollte $M \leqslant N$ sein, so ist die Behauptung trivial.
Im anderen Fall seien die Module absteigend nach ihrer sequentiellen Laufzeit sortiert, also aus $x < y$ folgt $T(x, 1) \geqslant T(y, 1)$. Unter Anwendung von Lemma 11 gilt für alle zur Parallelisierung ausgewählten Module z

$$N \cdot T(z, 1) > \sum_{x=1}^{M} T(x, 1) \geqslant \sum_{x=1}^{z} T(x, 1) \geqslant \sum_{x=1}^{z} T(z, 1) = z \cdot T(z, 1) \quad \Leftrightarrow \quad N > z.$$

Demzufolge können maximal $N - 1$ verschiedene Module zur Parallelisierung in Zeile 27) ausgewählt werden.

■

Aufgrund von Lemma 12 sind maximal $\min\{ N^2; M \cdot N \}$ Durchläufe der unteren Schleife ab Zeile 22) möglich. Auch wird bei jedem Schleifendurchlauf nur die Prozessoranzahl und Laufzeit eines Moduls geändert, so daß nicht alle Module neu nach ihrer Prozessoranzahl bzw. Laufzeit sortiert werden müssen. Das Einsortieren des einen veränderten Moduls ist nach $O(M)$ Schritten beendet. Die Laufzeit von „P1+MF“ unter Verwendung des Algorithmus „HMZ“, welcher abgesehen von der Sortierung der Module in $O(M)$ läuft, beträgt somit:

$$O\left(M \cdot \log(M) + \min\{ M \cdot N \cdot \log(N) + N^3; M \cdot N^2 \} + N^2 + \min\{ N^2; M \cdot N \} \cdot M\right)$$

$$= O\left(M \cdot \log(M) + M \cdot N^2\right).$$

Positioniert man die Module mit der größten Prozessoranzahl zuerst („BMZ“), so gibt es maximal $\min\{M; N\}$ Module mit einer Prozessoranzahl größer als Eins. Zum Positionieren eines Moduls mit einer beliebigen Prozessoranzahl benötigt man $O(N)$ Schritte. Nach dem Positionieren der $\min\{M; N\}$ Module mit der größten Prozessoranzahl verbleiben noch $M - \min\{M; N\}$. Jedes dieser Module benötigt zur Ausführung einen Prozessor und kann demzufolge nach dem einmaligen Aufbau des Heaps ($O(N \cdot \log(N))$ in jeweils $O(\log(N))$ Schritten auf dem Prozessor mit der kleinsten Endzeit positioniert werden. Man beachte, daß das Aufbauen des Heaps nur im Fall $M - \min\{M; N\} \geqslant 1$ notwendig ist. Demzufolge erhält man eine Laufzeit von

$$O\left(M \cdot \log(M) + \min\{ M \cdot N \cdot \log(N) + N^3; M \cdot N^2 \} + N^2 \right.$$

$$\left. + \min\{ N^2; M \cdot N \} \cdot \left(\min\{ N^2; M \cdot N \} + (N \cdot \log(N) + M \cdot \log(N)) \cdot \operatorname{sign}(M - \min\{M; N\})\right)\right)$$

$$= O\big(M \cdot \log(M) + \min\{\, M \cdot N^2 \cdot \log(N) + N^4;\, M^2 \cdot N^2 \}\big).$$

Die Funktion sign(x) ist dabei wie üblich definiert:

$$\text{sign}(x) = \begin{cases} 1 & \text{falls } x > 0 \\ 0 & \text{falls } x = 0 \\ -1 & \text{sonst} \end{cases}.$$

Im Fall $N \geqslant M$ werden also alle Module mit der in $O(N)$ laufenden Minimumsuche so tief wie möglich positioniert.

Im Anhang 9.9 ist ein Beispiel angegeben, bei dem der Algorithmus „P1+MF“ wirklich nur einen Schedule mit dem Gütefaktor 3 - ε erstellt.

2.1.6 Praktische Tests

Zusätzlich zu den theoretischen Ergebnissen zur Laufzeit und Ergebnisgüte wurden die Algorithmen an zufällig erzeugten Beispielen getestet. Die zufällige Erzeugung von Testbeispielen ist detailliert im Anhang 9.10 beschrieben. Einige Angaben zur Implementierung und der Testumgebung findet man im Abschnitt 9.11.
Zuerst wird die Ergebnisgüte betrachtet. Dabei sind zwei Faktoren entscheidend, nämlich der durchschnittliche und der schlechteste Wert. Allerdings muß zur exakten Bestimmung des Gütefaktors auch die Gesamtlaufzeit O des optimalen Schedules bekannt sein. Da die Berechnung des optimalen Schedules sehr rechenzeitintensiv gewesen wäre, wurde nur eine Abschätzung der Ergebnisgüte vorgenommen. Wie man nachvollziehen kann, gelten die folgenden beiden Aussagen:

1) $O \geqslant \frac{1}{N} \cdot \sum_{i=1}^{M} T(i, 1)$

2) $O \geqslant \max_{i=1}^{M} \{\, T(i, N) \,\}$

Demzufolge ist der vom Algorithmus erzielte Gütefaktor durch

$$G = \frac{H}{O} \leqslant \min\left\{ \frac{H \cdot N}{\sum_{i=1}^{M} T(i, 1)};\ \frac{H}{\max_{i=1}^{M} \{\, T(i, N) \,\}} \right\}$$

abschätzbar.
Alle in den nachfolgenden Graphiken aufgeführten Gütefaktoren wurden mit Hilfe dieser Formel berechnet. Man beachte, daß bei der Verwendung dieser einfachen Abschätzung auch Gütefaktoren größer als Drei möglich sind (vgl. Fall 2.2 im Beweis zu Lemma 10).

Damit die Werte vergleichbar sind, wurden mit jedem Algorithmus und jedem Beispiel 100 Testläufe durchgeführt. In jedem Testlauf wurden zufällig neue, zu optimierende Module erzeugt. Da bei der zufälligen Erzeugung von Testdaten Annahmen über die gewünschte Verteilung der sequentiellen Laufzeiten aller Module und deren Parallelisierungsgrade gemacht werden müssen, werden nachfolgend die Beispiele 1a und 1b betrachtet. Ausführliche Informationen zu den verwendeten Verteilungen bei der Beispielgenerierung können dem Anhang 9.10 entnommen werden.
In Abbildung 13 gibt die mittlere Markierung (Oberkante der dunklen Schraffierung) jeder Säule den durchschnittlichen und die obere (hell schraffiert) den schlechtesten Gütefaktor für die beim Beispiel 1a erzielten Schedules an. Die vier verschiedenen Säulen in einem Feld stehen für den verwendeten Algorithmus. Dabei beziehen sich die beiden linken auf „P1+MF“

mit „HMZ“. Bei der ganz links stehenden Säule lief der Algorithmus wie im Pseudocode angegeben. Die zweite Säule von links repräsentiert die Ergebnisse des gleichen Algorithmus ohne den Abbruch in Zeile 24) und den in Zeile 26) des Pseudocodes. Gilt in der Zeile 26) die Bedingung $B(x) = N$, so wählt der Algorithmus das Modul y mit der nächstkleineren Laufzeit zur Parallelisierung:

$$T(y) = \max_{z=1}^{v} \{ T(z) : T(z) \leqslant T(x) \text{ und } B(z) \neq N \}.$$

Erst wenn die v Module mit der größten sequentiellen Laufzeit mit allen N Prozessoren ausgeführt werden, terminiert der Algorithmus. Dabei steht v für die Anzahl der Module y für die

$$T(y, 1) \cdot N > \sum_{x=1}^{M} T(x, 1)$$

gilt.
Die zwei rechten Säulen eines jeden Diagramms stehen für die Ergebnisse von „P1+MF“ mit „BMZ“, je eine mit und eine ohne den Abbruch in den Zeilen 24) und 26) des Pseudocodes.
Als nächstes wird die Laufzeit der Algorithmen betrachtet. Die in der Abbildung 14 angegebenen Zeiten stellen ebenfalls einen Mittelwert aus 100 Läufen dar. Die Bedeutung der 4 Säulen ist entsprechend der Abbildung 13.
Unter den oben angegebenen Bedingungen erhält man für das Beispiel 1b die in der Abbildung 15 dargestellten Gütefaktoren und die in der Abbildung 16 dargestellten Laufzeiten.
Die Zahlenwerte der in den Abbildungen 13 bis 16 dargestellten Meßwerte können auf der beigelegten CD im Verzeichnis /res/bsp1a/ bzw. /res/bsp1b/ eingesehen werden. Dort befindet sich für jede Messung eine Textdatei sowie die Datei all.txt, welche nur die Laufzeiten und Ergebnisgüten der einzelnen Messungen enthält.

2.1.7 Auswertung der Testergebnisse

Man kann bei den praktisch ermittelten Ergebnissen drei verschiedene Bereiche bzgl. der Eingabedaten unterscheiden.
Der erste liegt bei $M \gg N$ vor. Dies ist sicherlich auch der praktisch relevante Fall. Mit einer durchschnittlichen Ergebnisgüte von rund Eins und einem schlechtesten Gütefaktor, welcher nur knapp über Eins liegt, erfüllt der gefundene Schedule sicherlich alle Ansprüche. Auch ist die Laufzeit, da der Schedule in vielen Fällen bereits vom Algorithmus „P1“ gefunden wird, sehr gering. Der zeitaufwendigste Teil des Verfahrens ist die Sortierung der Module in $O(M \cdot \log(M))$.
Gilt $M \approx N$, so liegt eine kritische Eingabe bzgl. der erzielten Ergebnisgüte vor. Im diesem Fall wird nur ein durchschnittlicher Gütefaktor von $G \approx 1{,}3$ erreicht. Das schlechteste Resultat innerhalb der 100 Läufe lag immer ungefähr bei Zwei. Dieser Wert ist sicherlich für fast alle praktischen Anwendungen ausreichend. Um diesen Schedule zu erzielen, mußte die Prozessoranzahl mehrerer Module geändert werden. Demzufolge sind auch mehrere Positionierungsversuche aller Module notwendig gewesen. Deshalb ist der Einfluß der Gesamtprozessoranzahl N auf die Laufzeit stärker als beim ersten Fall mit $M \gg N$.
Im letzten Fall mit $N \gg M$ wird die Laufzeitentwicklung von N dominiert. Der Grund dafür sind die vielen verschiedenen Prozessoranzahlen, die jedem Modul während des Programmablaufes zugeordnet werden müssen. Wird keine Parallelisierung der Module vorgenommen, so kann ein Großteil der Prozessoren zu vielen Zeitpunkten kein Modul ausführen. Die erzielte Ergebnisgüte ist an dieser Stelle wiederum sehr gut. So liegt der Gütefaktor im Durchschnitt sehr nah bei Eins. Auch der schlechteste Wert ist nur einen Bruchteil größer.

Die Abschätzung der Laufzeit des optimalen Schedules mit Hilfe der oben angegeben Formel liefert außerdem im Fall $M \gg N$ über die minimal benötigte Fläche und im Fall $N \gg M$ über die minimal mögliche Laufzeit des laufzeitintensivsten Moduls eine bessere untere Schranke als beim Vorliegen von $M \approx N$.
Vergleicht man die Ergebnisse von „P1+MF“ bei der Ausführung gemäß Pseudocode und Verwendung von „BMZ“ bzw. „HMZ“ miteinander, so kann man weder in der Laufzeit noch in der Ergebnisgüte erwähnenswerte Unterschiede feststellen. Im Fall $M \gg N$ sind beide Algorithmusvarianten sowieso gleich, da der Schedule bereits von „P1“ erstellt wird.
Das Weglassen des Abbruchs in den Zeilen 24) und 26) des Pseudocodes führt insbesondere im Fall $N \gg M$ zum Erstellen von besseren Schedules. Allerdings geht das stark auf Kosten einer erhöhten Laufzeit. Hier erkennt man auch, daß die Positionierung mittels „HMZ“ wesentlich schneller ist, als die Positionierung mit „BMZ“. Dafür sind die Ergebnisse von „BMZ“ aber etwas besser.
Vergleicht man die Ergebnisse der Beispiele 1a und 1b miteinander, so entstanden bei einer kleinen Modulanzahl im Beispiel 1b schlechtere Schedules als im Beispiel 1a. Für eine große Modulanzahl dreht sich das Verhältnis jedoch um. Im Beispiel 1b können bei kleiner Modulanzahl nur ein oder zwei Module mit sehr hoher sequentieller Laufzeit vorliegen, die aber gut parallelisierbar sind. Sie müssen also auf allen Prozessen ausgeführt werden um einen guten Schedule zu erzielen. Durch den Abbruch in den Zeilen 24) und 26) des Pseudocodes wird dieser Schedule aber nicht mehr erstellt, wenn zuvor ein Schedule mit zufriedenstellendem Gütefaktor berechnet wurde. Er wird erst durch das Weglassen der Abbruchbedingung gefunden. Beim Vorhandensein einer großen Modulanzahl können die im Beispiel 1b reichlich

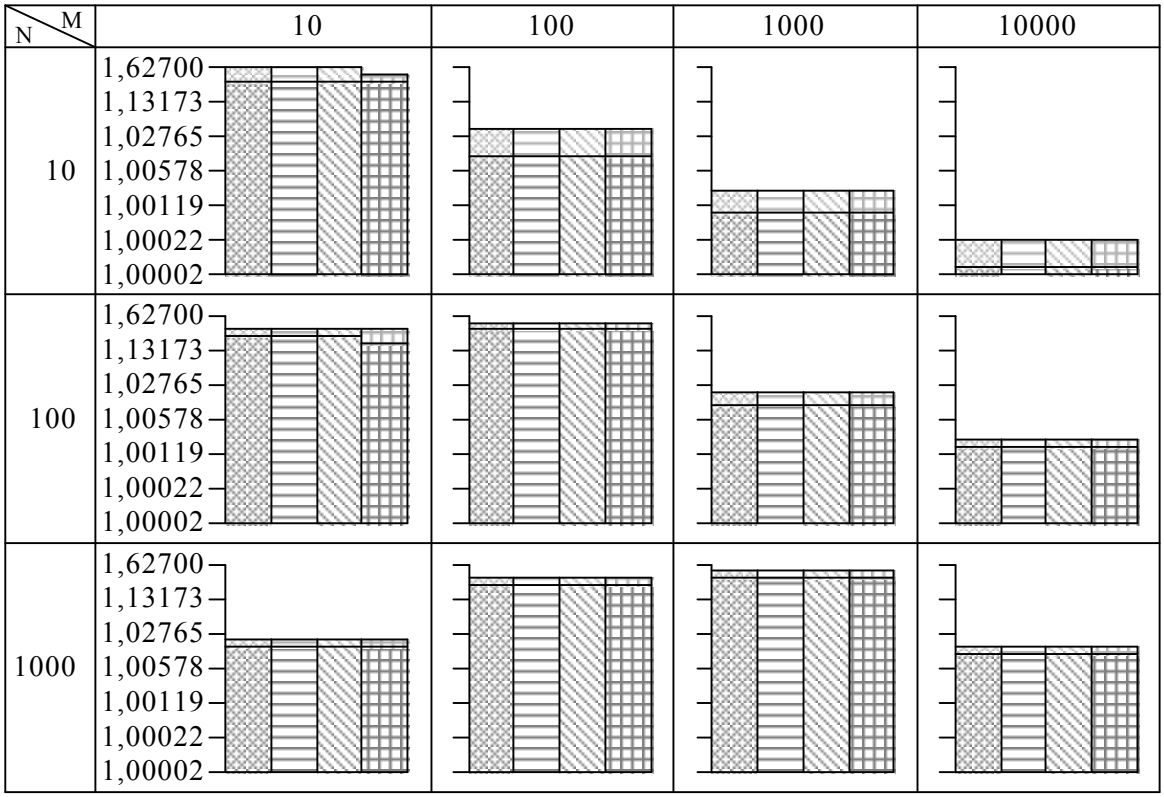


Abbildung 13 - durchschnittliche (mittlere Markierung) und schlechteste (obere Markierung) Gütefaktoren von Beispiel 1a.

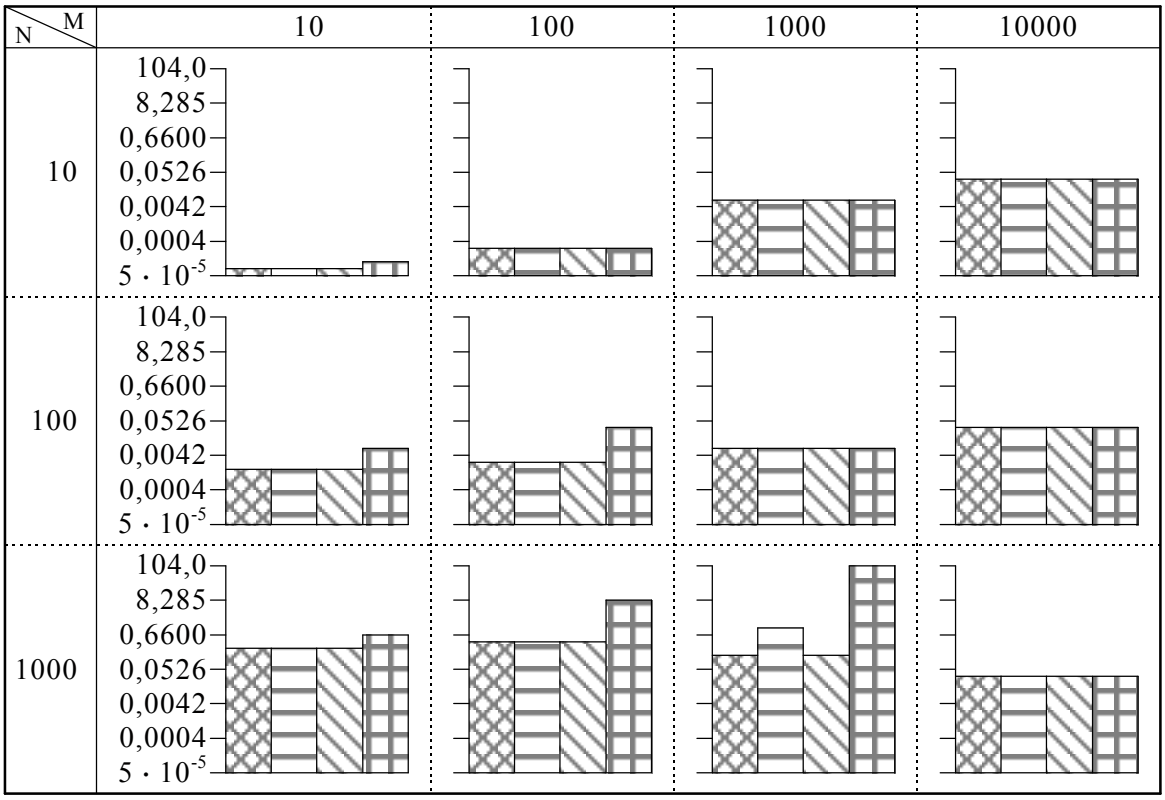


Abbildung 14 - durchschnittliche Laufzeit in Sekunden Beispiel 1a.

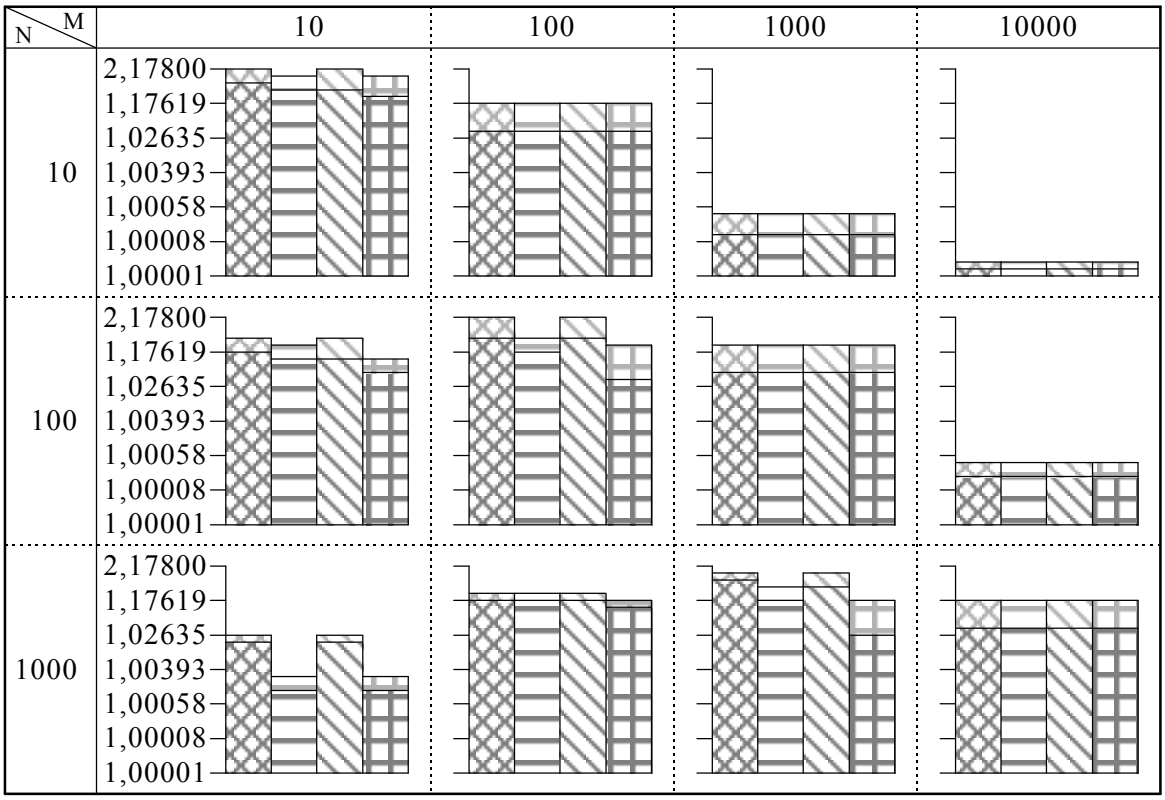


Abbildung 15 - durchschnittliche (mittlere Markierung) und schlechteste (obere Markierung) Gütefaktoren Beispiel 1b.

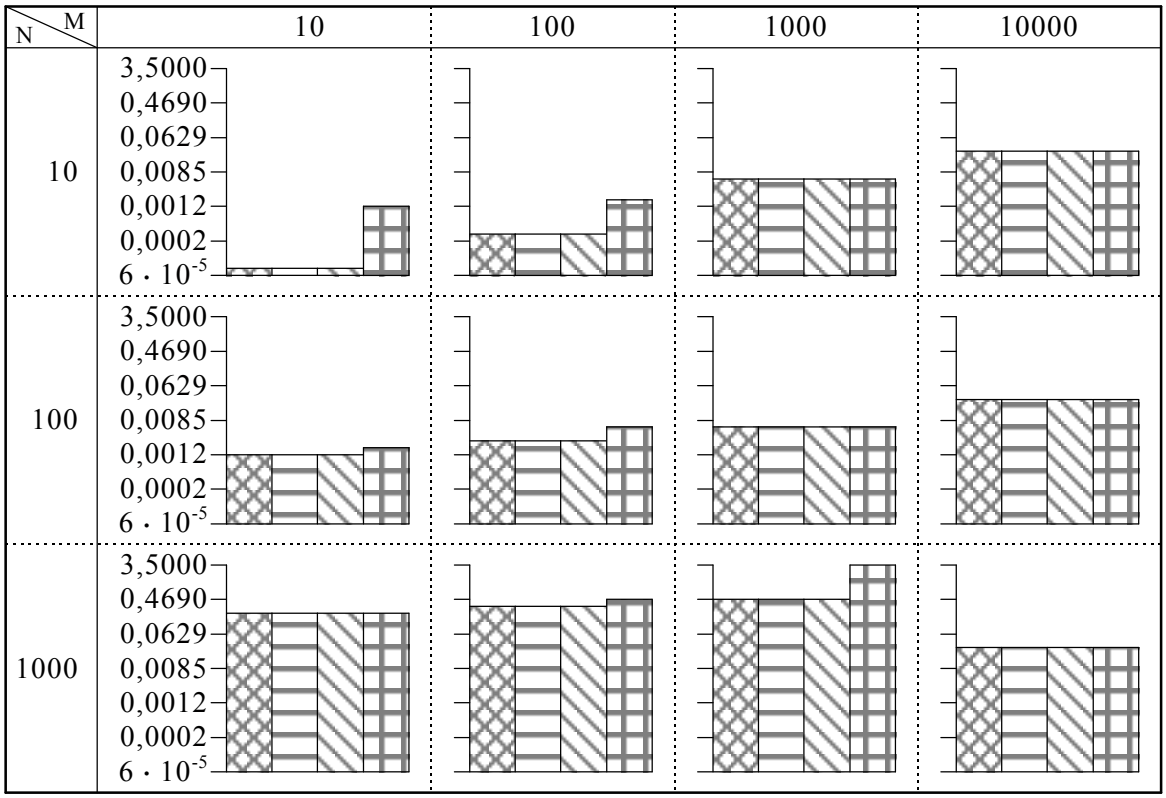


Abbildung 16 - durchschnittliche Laufzeiten in Sekunden Beispiel 1b.

vorhandenen Module mit kleiner Laufzeit sehr gut zur gleichmäßigen Auslastung aller Prozessoren verwendet werden. Dadurch werden in diesem Fall bessere Ergebnisse als im Beispiel 1a erzielt.
Bei der Laufzeit unterscheiden sich beide Beispiele nur um einen konstanten Faktor. Demzufolge mußten im Beispiel 1b durchschnittlich weniger Module parallelisiert werden als beim Beispiel 1a.
Insgesamt hat die Erstellung eines Schedules mit „P1+MF“ (gemäß Pseudocode) nie länger als eine Sekunde gedauert. Demzufolge kann der Algorithmus ohne Schwierigkeiten in der Praxis eingesetzt werden. Die Laufzeiteinsparung durch die Optimierung ist sicher größer, als die für die Optimierung benötigte Rechenzeit.

2.1.8 Vergleich der erzielten Ergebnisse mit denen anderer Algorithmen

Es gibt auch andere Algorithmen, welche die in diesem Abschnitt betrachtete Aufgabenstellung lösen. So garantiert der in [19] vorgestellte Algorithmus einen Gütefaktor von Zwei, wenn alle Module die Flächenklausel erfüllen. Seine Laufzeit ist mit $O(M^3 + M \cdot N)$ Schritten allerdings höher als die der hier vorgestellten Algorithmen.
Wenn man weiß, daß die Gesamtlaufzeit des optimalen Schedules den Wert Eins nicht überschreitet ($O \leqslant 1$), so findet man in [20] einen Algorithmus, welcher $H \leqslant \sqrt{3}$ garantiert. Man beachte jedoch, daß die Gesamtlaufzeit des optimalen Schedules im allgemeinen nicht bekannt ist und abgeschätzt werden muß. Dadurch entsteht wiederum ein Verlust der angegebenen Suboptimalität von $\sqrt{3}$. Bei diesem Algorithmus müssen außerdem alle Module die Flächenklausel erfüllen. Die Laufzeit des in [20] vorgestellten Algorithmus beträgt unter der An-

nahme, daß die Module in linearer Zeit sortiert werden können, nur $O(\min\{ M \cdot N; M^3 + M \cdot \log(N) \})$.
Ein ähnlicher Ansatz wird in [26] verfolgt. Auch hier muß $O \leqslant 1$ bekannt sein und die Flächenklausel gelten. Dann wird in $O(M \cdot N)$ ein Scheduling mit $H \leqslant 1{,}5$ berechnet.
Für den Fall, daß einige Module nicht die Flächenklausel erfüllen, wird in [17] ein Algorithmus mit einer polynomiellen Laufzeit von mindestens $O(M^2 \cdot N)$ und einem garantierten Gütefaktor von 2,5 vorgestellt. Dieser Algorithmus garantiert somit auf Kosten der Laufzeit einen besseren Schedule als die in dieser Arbeit vorgestellten Algorithmen.
In [21] findet man einen Algorithmus mit einer Ergebnisgüte von $1 + \varepsilon$ für ein festes $\varepsilon > 0$. Seine Laufzeit beträgt unter der Annahme, daß

$$M \geqslant 4 \cdot N^{N+1} \cdot (2 \cdot N)^{\left\lceil \frac{2 \cdot N}{\varepsilon} \right\rceil}$$

ist, nur $O(M)$. Dieser Algorithmus ist demzufolge nur für kleine N und nicht zu kleine ε verwendbar.
Ein weiterer Algorithmus, welcher das Scheduling Problem mit einem Gütefaktor von $G \leqslant 1 + \varepsilon$ löst, ist in [23] angegeben. Durch die Verwendung der dynamischen Programmierung beträgt seine Laufzeit $O\left(M \cdot \left(\frac{N}{\varepsilon}\right)^N\right)$. Dieser Algorithmus ist somit auch nur für kleine Prozessoranzahlen einsetzbar.

Der in [25] vorgestellte Algorithmus hat eine Laufzeit von mindestens $O\left(2^N \cdot M \cdot N^{\left(\frac{N}{\varepsilon}\right)}\right)$ und garantiert einen Gütefaktor von $G \leqslant 1 + \varepsilon$. Damit ist dieser Algorithmus ebenfalls nur für sehr kleine Prozessoranzahlen einsetzbar.
Es sei $T_{max} = \max_{x=1}^{M} \{ T(x, 1) \}$. Dann findet man in [24] einen Algorithmus, welcher in

$$O\left(N^3 \cdot \left(\ln(N) + \frac{1}{\varepsilon^2}\right) \cdot \min\left\{ M; \frac{N}{\varepsilon} \right\} \cdot \frac{1}{\varepsilon} \cdot \ln\left(\frac{N}{\varepsilon}\right) + N \cdot M\right)$$

einen Schedule mit der Eigenschaft

$$H \leqslant (1 + \varepsilon) \cdot O + 2 \cdot \left(1 + \frac{16}{\varepsilon}\right)^2 \cdot T_{max}$$

erstellt. Für $\varepsilon \geqslant 2$ erzielt dieser Algorithmus also noch nicht einmal einen garantierten Gütefaktor von Drei. Damit der Algorithmus einen garantierten Gütefaktor von Drei erzielt, muß

$$(1 + \varepsilon) \cdot O + 2 \cdot \left(1 + \frac{16}{\varepsilon}\right)^2 \cdot T_{max} \leqslant 3 \cdot O \quad \Leftrightarrow \quad 2 \cdot \left(1 + \frac{16}{\varepsilon}\right)^2 \cdot T_{max} \leqslant (2 - \varepsilon) \cdot O$$

$$\Leftrightarrow \quad \frac{T_{max}}{O} \leqslant \frac{(2 - \varepsilon) \cdot \varepsilon^2}{2 \cdot (\varepsilon + 16)^2}$$

gelten. Die rechte Seite der letzten Ungleichung nähert sich für ε gegen 0 und ε gegen 2 dem Wert 0. Ihr Maximum hat sie im Intervall (0; 2) bei folgendem ε:

$$0 = \left(\frac{(2 - \varepsilon) \cdot \varepsilon^2}{2 \cdot (\varepsilon + 16)^2}\right)' = (-\varepsilon^2 + (2 - \varepsilon) \cdot 2 \cdot \varepsilon) \cdot (2 \cdot (\varepsilon + 16)^2) - ((2 - \varepsilon) \cdot \varepsilon^2) \cdot (4 \cdot (\varepsilon + 16))$$

$$= (-\varepsilon + (2 - \varepsilon) \cdot 2) \cdot 2 \cdot (\varepsilon + 16) - (2 - \varepsilon) \cdot \varepsilon \cdot 4 \qquad (\varepsilon \neq 0; \varepsilon \neq -16)$$

$$= -2 \cdot \varepsilon^2 - 96 \cdot \varepsilon + 128 = \varepsilon^2 + 48 \cdot \varepsilon - 64$$

$$\varepsilon_{1/2} = -24 \pm \sqrt{24^2 + 64} = -24 \pm 8 \cdot \sqrt{10}$$

$\varepsilon = -24 + 8 \cdot \sqrt{10}$ (minus wegen $\varepsilon > 0$ nicht möglich)

$= 8 \cdot (\sqrt{10} - 3) \approx 1{,}298$

Für dieses ε ergibt sich

$$\frac{T_{max}}{O} \leqslant \frac{253 - 80 \cdot \sqrt{10}}{9} \approx 0{,}00198.$$

Dieses Ergebnis ist äquivalent zu

$$O \geqslant \frac{9}{253 - 80 \cdot \sqrt{10}} \cdot T_{max} = (253 + 80 \cdot \sqrt{10}) \cdot T_{max} > 505 \cdot T_{max}.$$

Anders ausgedrückt bedeutet das, daß der Prozessor, welcher im optimalen Schedule die Gesamtzeit bestimmt, mindestens 506 Module mit nur einem Prozessor ausführt. Man betrachte nun die Arbeitsweise von „P1“ bei dieser Eingabe. „P1“ erziele dabei einen Schedule mit der Gesamtlaufzeit H. Somit gilt für die minimal belegte Fläche A in diesem Schedule:

$$A \geqslant H + \left(H - T_{max}\right) \cdot (N - 1) > H + \left(H - \frac{1}{505} \cdot O\right) \cdot (N - 1) > \left(H - \frac{1}{505} \cdot O\right) \cdot N$$

Schätzt man nun die Gesamtlaufzeit des optimalen Schedules mit

$$O \geqslant \frac{A}{N} > H - \frac{1}{505} \cdot O \quad \Leftrightarrow \quad \frac{506}{505} \cdot O > H$$

ab, so erhält man für „P1“ einen garantierten Gütefaktor von

$$G = \frac{H}{O} < \frac{506}{505} \approx 1{,}002.$$

Somit ist unter den Voraussetzungen des in [24] vorgestellten Algorithmus schon „P1“ um Klassen besser.

Einen Algorithmus, welcher ähnlich zu dem hier vorgestellten „P1+“ arbeitet, ist in [6] gegeben. Unter der Annahme, daß alle Module die Flächenklausel erfüllen, liefert er in $O(M \cdot \log(M) + M \cdot N^2)$ ein Ergebnis mit einem Gütefaktor von $G \leqslant \frac{2}{1 + \frac{1}{N}}$.

In [27] wird der Spezialfall, daß alle Module die gleichen Ausführungszeiten haben und die Flächenklausel erfüllen, betrachtet. Der dort angegebene Algorithmus hat die Laufzeit $O(M)$ und garantiert den Gütefaktor $G \leqslant \frac{5}{4}$.

Falls für jedes Modul x eine Prozessoranzahl $B_{max}(x)$ mit der Eigenschaft

$$T(x, i) = \begin{cases} \left\lceil \frac{B_{max}(x)}{i} \right\rceil \cdot T\left(x, B_{max}(x)\right) & : \text{falls } i < B_{max}(x) \\ T\left(x, B_{max}(x)\right) & : \text{falls } i \geqslant B_{max}(x) \end{cases}$$

existiert, so wird in [28] ein Algorithmus vorgestellt, welcher in $O(N \cdot M \cdot \log(M))$ einen Schedule mit $H \leqslant 2 \cdot O + 1$ berechnet. Module mit dieser Eigenschaft werden als bulk - synchron bezeichnet.

Unter der Annahme, daß die Modulausführung unterbrochen und zu einem späteren Zeitpunkt auf einer anderen Prozessorgruppe fortgesetzt werden kann (preemptive Scheduling), ist in [22] ein Algorithmus mit pseudopolynomieller Laufzeit angegeben. Der von diesem Algorithmus erstellte Schedule garantiert den Gütefaktor von 1. Durch die Verwendung der linearen Programmierung kann dieser Algorithmus ebenfalls nur für kleine M und N genutzt werden.

2.2 Optimierung von nicht formbaren Modulen

2.2.1 Der flächenminimierende Algorithmus „XP+MF“

Der Algorithmus „XP+MF“ berechnet einen möglichen Schedule für formbare und nicht formbare Module. Er setzt voraus, daß keine Modulabhängigkeiten vorliegen. Seine Arbeitsweise lehnt sich an das im Abschnitt 2.1.5 beschriebenen Vorgehen an. Der Pseudocode des Algorithmus „XP+MF“ ist in der Abbildung 17 dargestellt.

Lemma 13: *Der Algorithmus „XP+MF“ garantiert einen Gütefaktor echt kleiner als Drei.*

Beweis: Mit $T_a(x)$ bzw. $B_a(x)$ wird die Belegung der Variablen $T(x)$ bzw. $B(x)$ beim a-ten Aufruf der Zeile 5) bezeichnet. Für die Prozessoranzahl des Moduls x in einem optimalen Schedule wird $B^{opt}(x)$ geschrieben. Außerdem gelte $T^{opt}(x) = T\left(x, B^{opt}(x)\right)$.
Die Zeile 2) garantiert

$$B_1(x) \cdot T_1(x) \leqslant B^{opt}(x) \cdot T^{opt}(x).$$

Analog zum Beweis von Lemma 10 kann man auch hier zeigen, daß aus

$$\forall x \in [1; M]: \; B_b(x) \cdot T_b(x) \leqslant B^{opt}(x) \cdot T^{opt}(x)$$

entweder

(α) „XP+MF“ hat einen Schedule mit einem Gütefaktor echt kleiner als Drei erstellt.

oder

(β) $\forall x \in [1; M]: \; B_{b+1}(x) \cdot T_{b+1}(x) \leqslant B^{opt}(x) \cdot T^{opt}(x)$ und $\exists y$ mit $B_{b+1}(y) > B_b(y)$

folgt.

■

1) ***für*** $x = 1$ ***bis*** M
2) $B(x) = \min_{i=1}^{N} \left\{ i : i \cdot T(x, i) = \min_{j=1}^{N} \{ j \cdot T(x, j) \} \right\}$
3) *positioniere die Module mittels „HMZ“ oder „BMZ“; die Positionierung habe die Gesamtlaufzeit h*
4) ***ist*** $h < H$, ***so*** $H = h$; *speichere den aktuellen Schedule*
5) *ist* $N \cdot H < 3 \cdot \sum_{x=1}^{M} T(x) \cdot B(x)$, ***so*** *stelle den zuletzt gespeicherten Schedule wieder her,* ***Ende***
6) $x = \min_{y=1}^{M} \left\{ y : T(y) = \max_{z=1}^{M} \{ T(z) \} \right\}$
7) ***ist*** $B(x) = N$, ***so*** *stelle den zuletzt gespeicherten Schedule wieder her,* ***Ende***
8) $B(x) = \min_{i=B(x)+1}^{N} \left\{ i : i \cdot T(x, i) = \min_{j=B(x)+1}^{N} \{ j \cdot T(x, j) \} \right\}$
9) ***weiter*** *in Zeile 3)*

Abbildung 17 - Algorithmus „XP+MF“.

Lemma 14: *Der Algorithmus „XP+MF“ führt die Zeile 3) höchstens* $(N \cdot \min\{N; M\})$*-mal aus.*

Beweis:
Fall 1: Es ist $M \leqslant N$.
Wie man sieht, sind nur $M \cdot N$ Parallelisierungen möglich. Anschließend werden alle Module voll parallel, d.h. mit allen N Prozessoren, ausgeführt.
Fall 2: Es gilt $M > N$.
Die Prozessoranzahl des Moduls x beim a-ten Aufruf der Zeile 5) wird mit $B_a(x)$ bezeichnet. Demzufolge enthält $B_1(x)$ die Anfangsprozessoranzahl (in Zeile 2) berechnet) von Modul x. $T_a(x)$ gibt die zur Prozessoranzahl $B_a(x)$ gehörende Laufzeit an, also $T_a(x) = T\left(x, B_a(x)\right)$.
Analog zu Lemma 11 kann man zeigen, daß „XP+MF“ in Zeile 6) nur Module y mit

$$N \cdot T_1(y) > \sum_{x=1}^{M} T_1(x) \cdot B_1(x) \qquad (1)$$

auswählt.
Im folgenden seien die Module absteigend bzgl. $T_1(x)$ sortiert:

$$\forall x \in [1; M - 1]: T_1(x) \geqslant T_1(x + 1).$$

Nun kann man den rechten Term von (1) abschätzen. Es gilt für alle in Zeile 6) ausgewählten Module

$$N \cdot T_1(y) > \sum_{x=1}^{M} T_1(x) \cdot B_1(x) \geqslant \sum_{x=1}^{y} T_1(x) \cdot B_1(x) \geqslant \sum_{x=1}^{y} T_1(x) \geqslant \sum_{x=1}^{y} T_1(y) = y \cdot T_1(y)$$

$$\text{bzw.} \quad N > y.$$

Diese Ungleichung garantiert, daß maximal $N - 1$ verschiedene Module zur Parallelisierung ausgewählt werden können.

■

Bei der Laufzeitabschätzung gehe man wie beim Algorithmus „P1+MF“ davon aus, daß die vollständige Sortierung der Module nach Laufzeit bzw. Prozessoranzahl nur einmal am Anfang vorgenommen wird. Dann braucht bei jedem Schleifendurchlauf ab Zeile 3) nur das Modul, dessen Prozessoranzahl geändert wurde, neu einsortiert werden, um die Sortierung aller Module nach Laufzeit bzw. Prozessoranzahl weiterhin zu gewährleisten. Unter Zuhilfenahme der Aussage von Lemma 14 erhält man bei der Verwendung des Positionierungsalgorithmus „HMZ“ die folgende obere Schranke:

$$O(M \cdot N + M \cdot \log(M) + (N \cdot \min\{N; M\}) \cdot M) = O(M \cdot \log(M) + M \cdot N \cdot \min\{N; M\}).$$

Beginnt man beim Positionieren mit dem Modul, welches die meisten Prozessoren benötigt („BMZ“), so resultiert bei der Verwendung von Bucket - Sort (es gibt nur N verschiedene Prozessoranzahlen) eine Laufzeit von

$$O(M \cdot N + (N \cdot \min\{N; M\}) \cdot (M \cdot N)) = O\left(N^2 \cdot M \cdot \min\{N; M\}\right).$$

2.2.2 Praktische Tests

Bei der zufälligen Erzeugung der Testbeispiele wurde die Grundstruktur von Beispiel 1a zur Erstellung von formbaren und nicht formbaren Modulen erweitert (siehe Beispiel 2 im Anhang 9.12). Bei den zufällig erzeugten Beispielen ist im Durchschnitt nur ein Achtel aller Module formbar. Die nicht formbaren Module haben eine Mindestprozessoranzahl zwischen 2 und $0{,}8 \cdot N$.

Nach jeweils 100 Durchläufen ergab sich für die Gütefaktoren das in der Abbildung 18 dargestellte Bild. Dabei beziehen sich die beiden linken Säulen in jedem Diagramm auf den Algorithmus „XP+MF“ bei der Verwendung des Positionierungsverfahrens „HMZ“, je eine mit und eine ohne Abbruch des Pseudocodes in den Zeilen 5) und 7). Der Algorithmus terminiert ohne die zwei Abbruchbedingungen erst, wenn alle Module mit N Prozessoren ausgeführt werden. Die anderen vier Säulen stehen für die Ergebnisse bei der Nutzung von „BMZ“. Bei den zwei rechten Säulen (je eine mit bzw. ohne Abbruch) werden die Module mit Bucket - Sort nur nach ihrer Breite sortiert. Das in der Mitte stehende Säulenpaar repräsentiert die Ergebnisse, wenn anstatt des Bucket - Sorts ein Merge - Sort verwendet wird, welches in bei gleicher Modulbreite zusätzlich nach der Modullaufzeit sortiert.
Bei den mit einem Stern markierten Säulen ist die benötigte Laufzeit sehr hoch, so daß nur 10 anstatt der üblichen 100 Testbeispiele berechnet wurden.
Die Berechnung einer unteren Schranke für die Gesamtlaufzeit O des optimalen Schedules erfolgte durch die folgenden Formeln:

1) $$O \geqslant \frac{1}{N} \cdot \sum_{x=1}^{M} \min_{i=1}^{N} \{ T(x, i) \cdot i \}$$

2) $$O \geqslant \max_{x=1}^{M} \{ T(x, N) \}$$

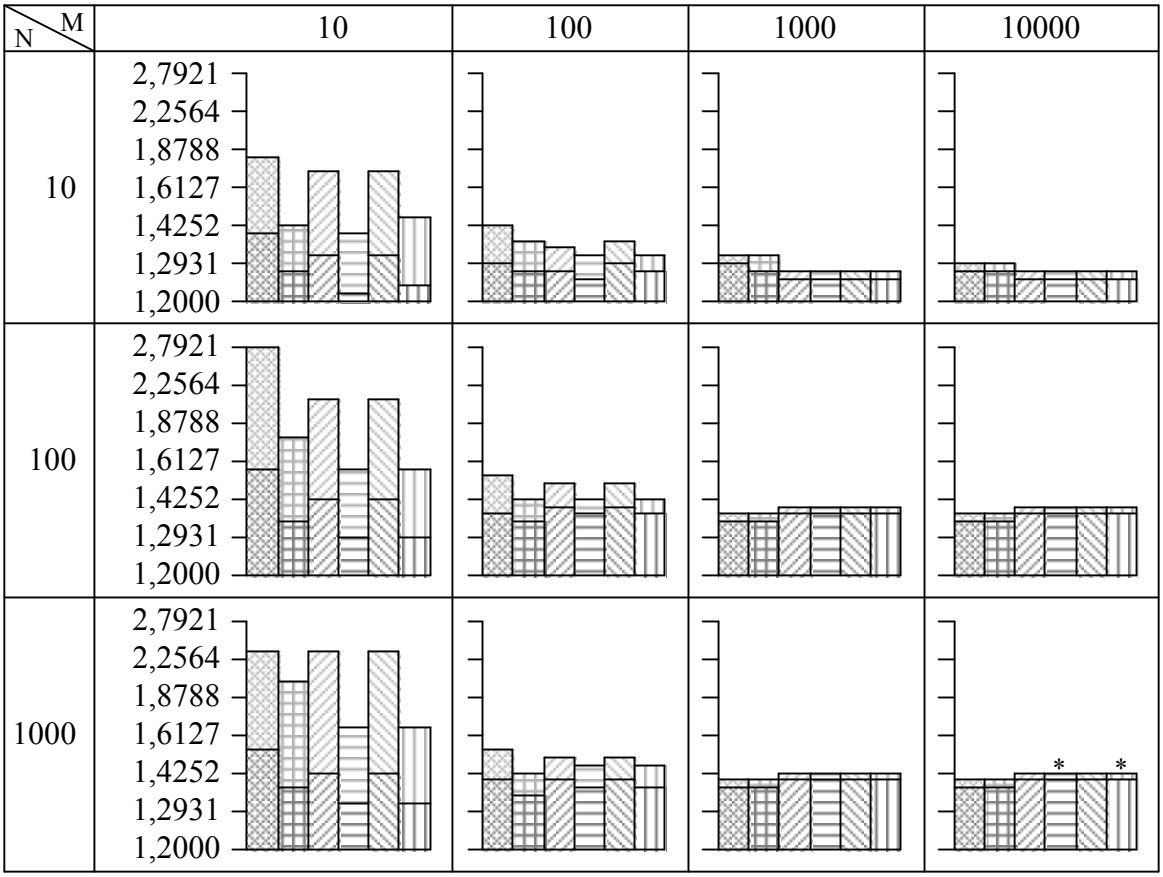


Abbildung 18 - durchschnittliche (mittlere Markierung) und schlechteste (obere Markierung) Gütefaktoren von Beispiel 2.

Mit Hilfe des Maximums der beiden Werte wird durch eine Division der Mindestgütefaktor bestimmt.
Man beachte jedoch, daß das Berechnungsverfahren für die untere Schranke der Gesamtlaufzeit des optimalen Schedules in einigen Fällen sehr grob ist.
Als Beispiel nehme man $M \geqslant 3$ gleichartige nicht formbare Module mit

$$T(x, i) = \begin{cases} \infty : \text{falls } i \leqslant \frac{1}{2} \cdot N \\ 1 \ : \text{sonst} \end{cases}.$$

Man sieht, daß „XP+MF“ den optimalen Schedule mit der Gesamtlaufzeit M erzielt. Die Abschätzung für den optimalen Schedule liefert aber nur

$$O \geqslant \frac{1}{N} \cdot M \cdot \lfloor \frac{1}{2} \cdot N + 1 \rfloor \approx \frac{1}{2} \cdot M.$$

Es wird bei diesem Beispiel also nur ein Gütefaktor von rund Zwei angegeben, obwohl ein Gütefaktor von Eins erreicht wurde.
Die ermittelten Laufzeiten sind in der Abbildung 19 dargestellt. Die den Diagrammen zu Grunde liegenden Zahlenwerte sind auf der CD im Verzeichnis /res/bsp2/ einsehbar.

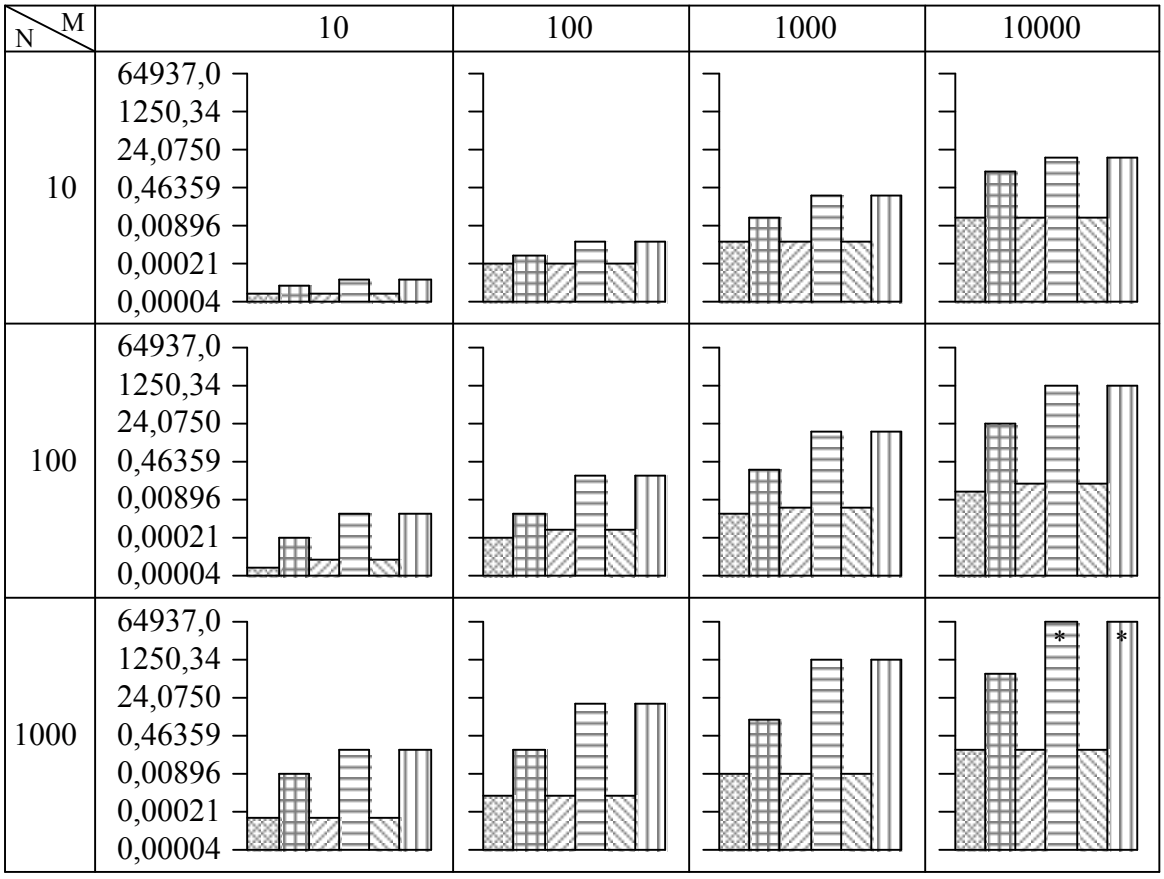


Abbildung 19 - durchschnittliche Laufzeit von Beispiel 2.

2.2.3 Auswertung der Testergebnisse

Als erstes werden die drei Versionen („HMZ“ und „BMZ“ mit Merge - bzw. Bucket - Sort), welche beim Erreichen eines Gütefaktors echt kleiner als Drei abbrechen, betrachtet. Vergleicht man die Laufzeiten miteinander, so kann man keine relevanten Unterschiede feststel-

len. Bei den Ergebnisgüten fällt hingegen sofort ein Unterschied auf: Liegt eine geringe Modulanzahl oder eine geringe Prozessoranzahl vor, so ist „HMZ“ immer schlechter als „BMZ“. Durch das ebenenweise Positionieren von „HMZ“ ist bei einer geringen Modulanzahl die Wahrscheinlichkeit sehr groß, daß zwei in einer Ebene positionierte Module stark verschiedene Laufzeiten haben und somit ein größerer Leerraum entsteht. Da die minimale Prozessoranzahl von allen nicht formbaren Modulen im Intervall von 2 bis $0{,}8 \cdot N$ liegt, ist bei kleinem N die relative Breite bzgl. N der Module größer als bei einem großem N. Das hat bei „HMZ“ zur Folge, daß die Anzahl der unbenutzten Prozessoren bei kleiner werdendem N in jeder Ebene steigt.
Bei der Verwendung von „BMZ“ beeinflußt die Sortierung der Module gleicher Breite nach ihrer Laufzeit die Ergebnisgüte nur minimal. Am größten ist der Unterschied der Ergebnisgüte bei kleiner Modul- und Prozessoranzahl. Betrachtet man in diesem Fall die Positionierung der letzten beiden Module mit gleicher Breite, so erkennt man, daß es aufgrund der geringen Prozessoranzahl oft genau eine halbwegs optimale Lücke gibt, welche eins der beiden Module aufnehmen kann. Das zweite Modul wird dann wesentlich später gestartet und hat durch die fehlende Sortierung auch noch eine größere Modullaufzeit. Bei einer größeren Modulanzahl ist die Gesamtlaufzeit H aller Module höher, so daß der Einfluß der schlechten Positionierung der letzten beiden Module geringer wird.
Läßt man die drei Versionen („HMZ“, „BMZ“ mit bzw. ohne Sortierung der Module gleicher Breite) des Verfahrens bis zum Zeitpunkt, bei welchem alle Module mit N Prozessoren ausgeführt werden, laufen, so erhöht sich die Laufzeit sehr stark. Man sieht außerdem, daß bei der wiederholten Ausführung von „BMZ“ mit der Minimumsuche in $O(N)$ eine wesentlich höhere Laufzeit entsteht, als durch den einmaligen Sortiervorgang von „HMZ“. Die oben geführten Überlegungen zu den Gütefaktoren gelten auch hier. Natürlich werden durch die Elimination der Abbruchbedingungen für Eingaben mit kleiner Modulanzahl wesentlich bessere Schedules erstellt.
In der Praxis sollte der Abbruch also nur umgangen werden, wenn der garantierte Gütefaktor z.B. noch über Zwei liegt. Eine Garantie für einen wesentlich besseren Gütefaktor ist die Löschung der Abbruchbedingung aber nicht.

Für die Kombination von formbaren und nicht formbaren Modulen ist in [17] ein Algorithmus mit einer polynomiellen Laufzeit von mindestens $O(M^2 \cdot N)$ und einem garantierten Gütefaktor von 2,5 angegeben. Dieser Algorithmus ist langsamer, aber erlaubt die Angabe einer besseren Schranke für den Gütefaktor als das hier vorgestellte Verfahren „XP+MF“.
Außerdem kann für kleine N der Algorithmus aus [21] zur Optimierung verwendet werden. Er benötigt dann nur $O(M)$ Schritte und garantiert eine Ergebnisgüte von $1 + \varepsilon$ für ein $\varepsilon > 0$.

3 Näherungsalgorithmen beim Vorliegen eines serien - parallelen Modulabhängigkeitsgraphen

3.1 Optimierung von formbaren Modulen

3.1.1 Eigenschaften von serien - parallelen Graphen

Nachdem bisher Näherungsalgorithmen zur Lösung des Schedulingproblems ohne Modulabhängigkeiten vorgestellt wurden, werden nun auch Datenabhängigkeiten zwischen den Modulen berücksichtigt. Bevor jedoch ein allgemeiner Abhängigkeitsgraph betrachtet wird, soll als Unterklasse die Menge der serien - parallelen Graphen bearbeitet werden.

Definition 7: *Ein gerichteter azyklischer Graph $G = (V, E)$ mit der Knotenmenge V und der Kantenmenge E heißt serien - parallel, wenn er eine der Bedingungen (a) bis (c) erfüllt (vgl. [34] und Abbildung 20):*

(a) $|V| = 1$ *und* $E = \varnothing$

(b) *Es existiert mindestens eine Zerlegung* $V = V_1 \cup V_2$ *mit* $V_1 \cap V_2 = \varnothing$ *und* $E_1 = E \cap (V_1 \times V_1)$ *sowie* $E_2 = E \cap (V_2 \times V_2)$ *mit* $E = E_1 \cup E_2$, *so daß* $G_1 = (V_1, E_1)$ *und* $G_2 = (V_2, E_2)$ *serien - parallele Graphen sind.*

Diesen Schritt bezeichnet man als parallele Vereinigung. In Zeichen wird $G = G_1 \parallel G_2$ *geschrieben.*

(c) *Es existiert mindestens eine Zerlegung* $V = V_1 \cup V_2$ *mit* $V_1 \cap V_2 = \varnothing$ *und* $E_1 = E \cap (V_1 \times V_1)$ *sowie* $E_2 = E \cap (V_2 \times V_2)$ *mit* $E = E_1 \cup E_2 \cup (S \times T)$, *so daß* $G_1 = (V_1, E_1)$ *und* $G_2 = (V_2, E_2)$ *serien - parallele Graphen sind. Dabei bezeichne*

$$S = \{ v : v \in V_1 \text{ und } \nexists x \in V_1 \text{ mit } (v, x) \in E_1 \}$$

die Senken von G_1. *Analog stehe*

$$T = \{ v : v \in V_2 \text{ und } \nexists x \in V_2 \text{ mit } (x, v) \in E_2 \}$$

für die Quellen von G_2.

Dieser Schritt heißt serielle Vereinigung und wird als $G = G_1 + G_2$ *notiert.*

Betrachtet man die serielle Vereinigung genauer, so lassen sich die folgenden Eigenschaften angeben:

Lemma 15: *Es gelte* $G = G_1 + G_2$, *wobei* $G_1 = (V_1, E_1)$ *und* $G_2 = (V_2, E_2)$ *serien - parallele Graphen sind. Außerdem sei* $x \in V_1$ *und* $y \in V_2$. *Dann gibt es in G einen Pfad von x nach y.*

Der **Beweis** dieses Lemmas unterteilt sich in drei Teile und verwendet die in der Definition 7 eingeführten Bezeichnungen:

Fall 1: Es sei $x \in S$ und $y \in T$ gegeben.

Da laut Definition $(S \times T) \subseteq E$ gilt, folgt $(x, y) \in E$.

Fall 2: Es liege $x \notin S$ und $y \in T$ vor.
Wenn x keine Senke ist, so gibt es eine Kante $(x, z_1) \in E_1 \subseteq E$. Ist $z_1 \notin S$ so gibt es eine weitere Kante $(z_1, z_2) \in E_1 \subseteq E$. Da G_1 zyklusfrei ist, wird irgendwann ein Knoten $z_i \in S$ erreicht. Man hat somit einen Pfad von x nach z_i gefunden. Wie im ersten Fall gezeigt wurde, gibt es einen Pfad von z_i nach y und deshalb auch von x nach y.

Fall 3: Es sei $y \notin T$.
Es wird wie eben ein Rückwärtspfad von y zu einem Knoten $z \in T$ gesucht. Dann hat man einen Pfad von z nach y und mit Hilfe des ersten oder zweiten Falles kann man die Behauptung ableiten.

■

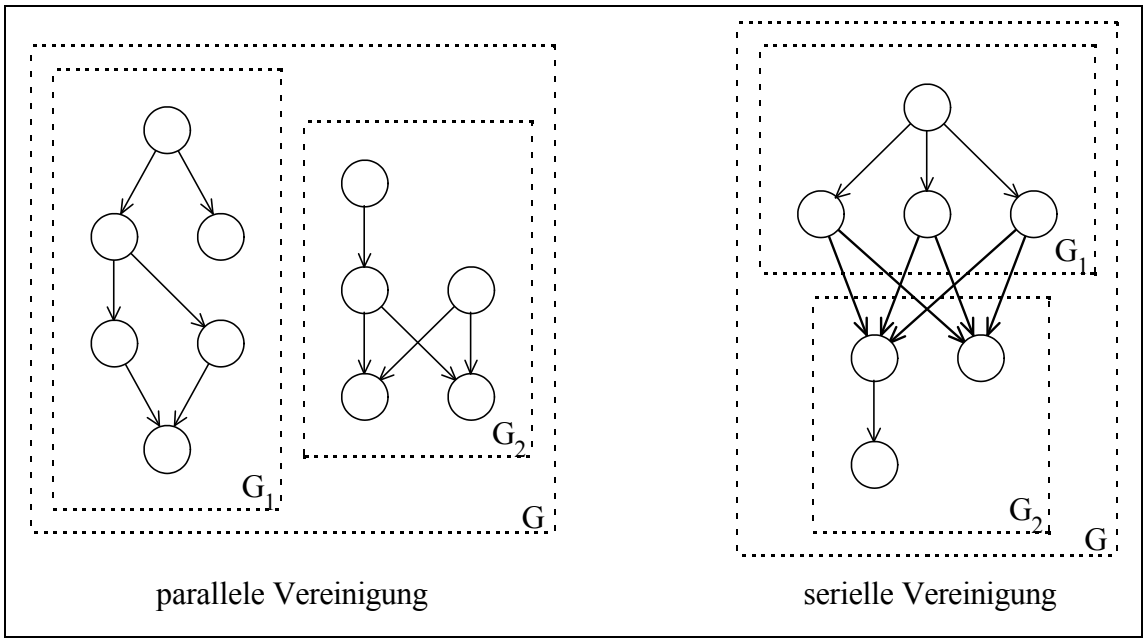


Abbildung 20 - Veranschaulichung der Definition 7.

Lemma 16: *Es sei $x \in V$ ein Knoten des serien - parallelen Graphen $G = (V, E)$.*

(1) Dann haben alle Kinder von x den gleichen Eingangsgrad, also $\forall y$, $\forall z$ mit $(x, y) \in E$ und $(x, z) \in E$ gilt $indeg(y) = indeg(z)$. Dabei bezeichne $indeg(y)$ den Eingangsgrad des Knotens y.

(2) Dann haben alle Vorgänger von x den gleichen Ausgangsgrad, also $\forall y$, $\forall z$ mit $(y, x) \in E$ und $(z, x) \in E$ gilt $outdeg(y) = outdeg(z)$. Dabei bezeichne $outdeg(y)$ den Ausgangsgrad des Knotens y.

Beweis: Es wird nur die Aussage (1) gezeigt. Die Aussage (2) folgt analog.
Aus der Definition 7 folgt, daß alle Kanten des Graphen G bei der seriellen Vereinigung zweier Teilgraphen entstanden sind. Außerdem werden nur ausgehende (eingehende) Kanten zu Knoten mit dem Ausgangsgrad (Eingangsgrad) Null hinzugefügt. Man betrachte nun den Zeitpunkt, zu dem zwei Teilgraphen G_1 und G_2 zu einem Graphen $G_3 = G_1 + G_2$ vereinigt werden. Es seien die Knoten y und z Quellen in G_2, also in der Notation aus der Definition 7: $\{y; z\} \subseteq T$. Der Graph G_3 besteht einerseits aus den Kanten von G_1 sowie G_2 und andererseits

aus der Kantenmenge ($S \times T$). Da die Knoten y und z in G_2 den Eingangsgrad Null haben und in G_1 gar nicht enthalten sind, führen nur Kanten von jedem Knoten aus der Menge S zu y bzw. z. Demzufolge gilt indeg(y) = indeg(z) = |S|.

■

Definition 8: *Ein binärer Baum B = (W, F) (W ≠ ∅) mit der Knotenmenge W, der Kantenmenge F und der Knotenbeschriftungsfunktion h : W → {#; +; ||} heißt Konstruktionsbaum zum serien - parallelen Graphen G = (V, E), wenn eine der Bedingungen (a) bis (c) erfüllt ist:*

(a) Es ist V = {a} und E = ∅.
Dann muß W = {a} mit h(a) = # und F = ∅ sein.

(b) G ist als parallele Vereinigung von G_1 und G_2 darstellbar $(G = G_1 \parallel G_2)$.
Es sei $B_i = (W_i, F_i)$ ein Konstruktionsbaum von G_i ($i \in \{1; 2\}$). Mit a $(a_1; a_2)$ werde die Wurzel von B $(B_1; B_2)$ bezeichnet. Dann muß h(a) = || und $W = \{a\} \cup W_1 \cup W_1$ gelten. Außerdem muß a als linken (rechten) Nachfolger den Knoten a_1 (a_2) haben: $F = \{(a, a_1, a_2)\} \cup F_1 \cup F_2$.

(c) G ist als serielle Vereinigung von G_1 und G_2 darstellbar $(G = G_1 + G_2)$.
Es sei $B_i = (W_i, F_i)$ ein Konstruktionsbaum von G_i ($i \in \{1; 2\}$). Mit a $(a_1; a_2)$ werde die Wurzel von B $(B_1; B_2)$ bezeichnet. Dann muß h(a) = + und $W = \{a\} \cup W_1 \cup W_1$ gelten. Außerdem muß a als linken (rechten) Nachfolger den Knoten a_1 (a_2) haben: $F = \{(a, a_1, a_2)\} \cup F_1 \cup F_2$.

Lemma 17: *Zu jedem Konstruktionsbaum B ist eindeutig ein serien - paralleler Graph G angebbar.*

Als **Beweis** wird die Konstruktionsvorschrift anhand der Struktur von B angegeben.
Es sei a die Wurzel von B.
Fall 1: Es sei h(a) = #.
Dann gilt V = {w} und E = ∅.
Fall 2: Es gelte $h(a) \in \{+, \|\}$.
Mit B_1 (B_2) wird der linke (rechte) Unterbaum von a bezeichnet. Außerdem sei G_i der zu B_i gehörende serien - parallele Graph ($i \in \{1; 2\}$).
Fall 2.1: Es sei h(a) = +.
Dann entsteht G aus der seriellen Vereinigung von G_1 und G_2 $(G = G_1 + G_2)$.
Fall 2.2: Es gilt h(a) = ||.
Dann entsteht G aus der parallelen Vereinigung von G_1 und G_2 $(G = G_1 \parallel G_2)$.

■

Damit nicht alle Konstruktionsbäume B mittels Knoten- und Kantenmenge notiert werden müssen, sollte man sie als Kantorowitsch - Bäume (siehe [57]) auffassen und als Konstruktionsterm K notieren:

Definition 9: *Der Konstruktionsterm K eines Konstruktionsbaums B mit der Wurzel a ist wie folgt definiert:*

Ist a ein Blatt des Baumes (B besteht also nur aus a), so lautet der Konstruktionsterm K = a.
Wenn a kein Blatt ist, so hat a den linken (rechten) Unterbaum B_1 (B_2). Der zu B_1 (B_2) gehörende Konstruktionsterm sei K_1 (K_2).
Gilt h(a) = ||, so lautet der Konstruktionsterm für den Baum B: K = (K_1) || (K_2).
Gilt h(a) = +, so lautet die Konstruktionsterm für den Baum B: K = (K_1) + (K_2).

Lemma 18: *Die Umkehrung von Lemma 17 gilt nicht. Demzufolge kann ein serien - paralleler Graph mehrere Konstruktionsbäume haben.*

Als **Beweis** dazu wird in der Abbildung 21 ein serien - paralleler Graph angegeben, welcher mindestens zwei verschiedene Konstruktionsbäume besitzt.

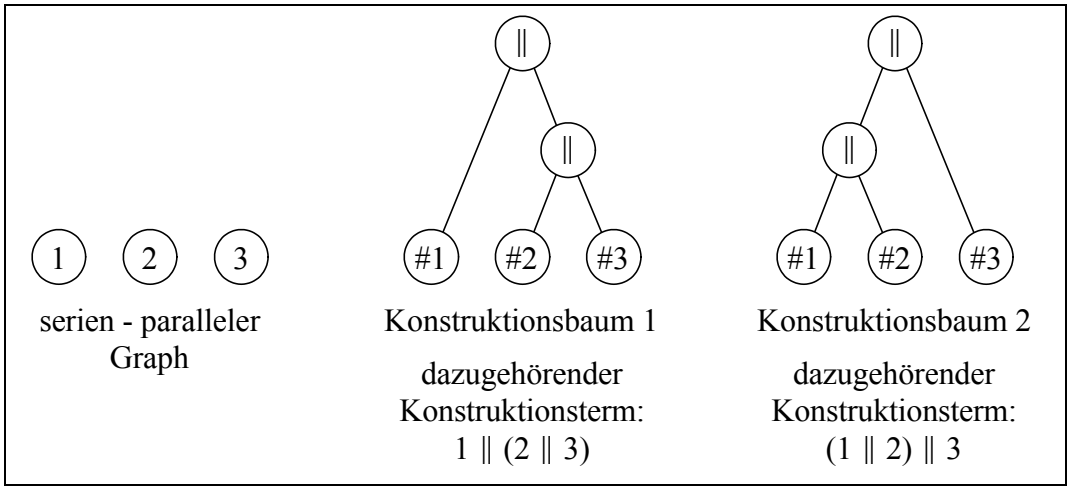


Abbildung 21 - Gegenbeispiel für Lemma 18.

■

Um in den Konstruktionstermen nicht immer alle Klammern mitschreiben zu müssen, gelten die in der Definition 10 angegebenen Prioritäten:

Definition 10: *Für die Auswertungsreihenfolge der Operationen + und || in einem Konstruktionsterm werden die folgenden Regeln definiert:*

(a) Die Operationen + und || haben die gleiche Priorität und sind linksassoziativ, d.h., für die Konstruktionsterme a, b und c gilt: a + b + c = (a + b) + c, a + b || c = (a + b) || c, a || b + c = (a || b) + c und a || b || c = (a || b) || c.

(b) *Es seien c und d natürliche Zahlen mit c < d. Dann kann eine mehrfache serielle Vereinigung der Konstruktionsterme a_c, ..., a_d wie folgt geschrieben werden:*

$$\sum_{i=d}^{c} a_i = \varnothing, \quad \sum_{i=c}^{c} a_i = a_c \quad \textit{und}$$

$$\sum_{i=c}^{d} a_i = \left(\sum_{i=c}^{d-1} a_i \right) + a_d$$
$$= (((\cdots((a_c + a_{c+1}) + a_{c+2}) + \cdots) + a_{d-1}) + a_d$$

(c) *Äquivalent zu (b) wird für die parallele Vereinigung definiert:*

$$\prod_{i=d}^{c} a_i = \varnothing, \quad \prod_{i=c}^{c} a_i = a_c \quad \textit{und}$$

$$\prod_{i=c}^{d} a_i = \left(\prod_{i=c}^{d-1} a_i \right) \parallel a_d = (((\cdots((a_c \parallel a_{c+1}) \parallel a_{c+2}) \parallel \cdots) \parallel a_{d-1}) \parallel a_d$$

Lemma 19: *Der Konstruktionsbaum B = (W, F) zum Modulabhängigkeitsgraphen G = (V, E) hat genau |V| Blätter und |V| - 1 innere Knoten, also |W| = 2 · |V| - 1.*

Beweis: In der Konstruktionsvorschrift zur Erstellung eines Konstruktionsbaumes B aus G (siehe Definition 8) wird nur im Fall (a) ein Blatt von B erstellt. In den Fällen (b) und (c) werden nur bestehende Teilgraphen von B zusammen gefügt. Dabei entsteht aber kein neues Blatt, sondern ein Knoten mit dem Ausgangsgrad Zwei. Fall (a) tritt aber nur ein, wenn der betrachtete Teilgraph von G genau aus einem Knoten besteht. Demzufolge muß der Fall (a) bei der Erstellung eines Konstruktionsbaumes genau |V| Mal eintreten. Ein binärer Baum mit |V| Blättern hat immer |V| - 1 innere Knoten und besteht insgesamt aus 2 · |V| - 1 Knoten. ■

3.1.2 Berechnung eines Konstruktionsbaumes

3.1.2.1 Auflösung von komplexen seriellen Vereinigungen

Da die Berechnung eines Konstruktionsbaumes für beliebige serien - parallele Graphen etwas komplexer ist, wird an dieser Stelle ein Dreischrittalgorithmus vorgestellt.

Definition 11: *Eine serielle Vereinigung zweier serien - paralleler Graphen G_1 und G_2 heißt genau dann komplex, wenn G_1 mindestens zwei Senken und G_2 mindestens zwei Quellen hat.*

Im ersten Schritt soll ein beliebiger serien - paralleler Graph durch das Einfügen zusätzlicher Knoten so vereinfacht werden, daß er keine komplexen seriellen Vereinigungen mehr enthält. Dazu wird eine komplexe serielle Vereinigung $G = G_1 + G_2$ in zwei nicht komplexe serielle Vereinigungen an einem Hilfsknoten a aufgeteilt. Die neue Vereinigungsvorschrift lautet somit $G' = G_1 + G_3 + G_2$, wobei $G_3 = (\{a\}, \varnothing)$ nur aus dem Hilfsknoten a besteht (vgl. Abbildung 22).

Das Aufsuchen von komplexen Vereinigungen in einem serien - parallelen Graphen ist durch die Betrachtung des Ausgangsgrades eines Knotens x und des Eingangsgrades eines Nachfolgers von x möglich. Da sich beim Auflösen der komplexen Vereinigungen die Kantenanzahl des Graphen höchstens verringert und jeder eingefügte Hilfsknoten mindestens zwei Vorgänger aus G = (V, E) hat, gelten

$$|E'| \leqslant |E| \quad \text{und} \quad |V'| \leqslant \tfrac{3}{2} \cdot |V|,$$

wobei G' = (V', E') durch die Auflösung aller komplexen Vereinigungen aus G hervorgegangen ist. Demzufolge benötigt der in der Abbildung 23 notierte Algorithmus „KoVer“ maximal O(|V| + |E|) Schritte.

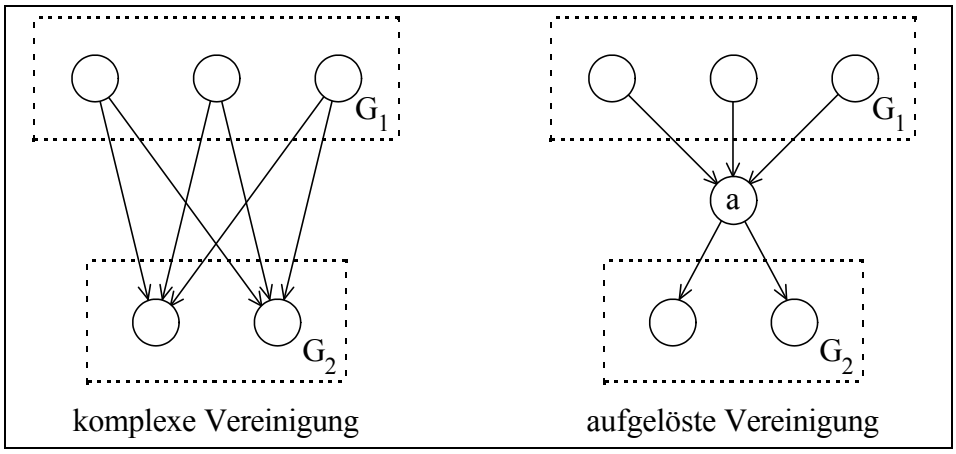


Abbildung 22 - Auflösen von komplexen Vereinigungen.

```
1)   a = |V| + 1
2)   für x = 1 bis |V|
3)        ist outdeg(x) ⩾ 2, so tue
4)             wähle y ∈ {z : z ∈ V und (x, z) ∈ E}
5)             ist indeg(y) ⩾ 2, so tue
6)                  I = {z : z ∈ V und (z, y) ∈ E}
7)                  J = {z : z ∈ V und (x, z) ∈ E}
8)                  E = E \ (I × J)
9)                  V = V ∪ {a}
10)                 E = E ∪ (I × {a}) ∪ ({a} × J)
11)                 a = a + 1
```

Abbildung 23 - Algorithmus „KoVer“.

3.1.2.2 Der Konstruktionsbaum eines serien - parallelen Graphen ohne komplexe Vereinigungen

Nach der Auflösung aller komplexen Vereinigungen im serien - parallelen Graphen G = (V, E) werden noch zwei weitere Hilfsknoten x und y in G eingefügt. Von x verlaufe je eine Kante zu jeder Quelle in G und zu y führe je eine Kante von jeder Senke in G. Somit besitzt der neue Graph exakt eine Quelle und eine Senke. Bei dieser Konstruktion erhöht sich die Kantenanzahl um maximal $2 \cdot |V|$.

Definition 12: *Es sei G = (V, E) ein serien - paralleler Graph mit nur einer Quelle z und ohne komplexe Vereinigungen. Der von $x \in V$ erzeugte Teilgraph $G_x = (V_x, E_x)$ ist dann wie folgt definiert:*
$V_x = \{y : y \in V$ und alle gerichteten Pfade von z nach y führen über x$\} \cup \{ x \}$
und $E_x = E \cap (V_x \times V_x)$.

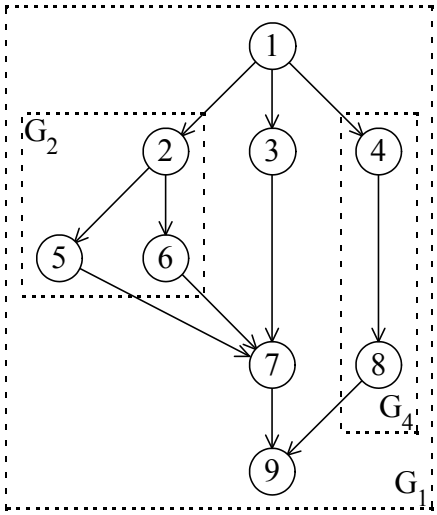
V_x enthält also alle Knoten des Graphen G, die von x dominiert werden sowie den Knoten x (siehe [58]).

Nachfolgend bezeichne $\mathcal{B}$ die Menge aller Konstruktionsbäume, inklusive des leeren Baumes. In der Abbildung 24 ist ein rekursiver Algorithmus angegeben, welcher zu einem serien - parallelen Graphen ohne komplexe Vereinigungen mit einer Quelle und einer Senke einen Konstruktionsbaum erstellt.

1) ***für*** $x = 1$ ***bis*** $|V|$
2) $eing(x) = indeg(x)$
3) $eing(\infty) = \infty$
4) *x sei die Quelle von G*
5) $(y, B) = baubaum(x)$
6) *der Konstruktionsbaum von G ist B*
7) ***Ende***
8) ***Unterroutine baubaum****(x)*
9) ***ist*** $outdeg(x) = 0$, ***so Rücksprung*** *unter Rückgabe von* (∞, x)
10) *definiere die Variable* $K = \{(y, \varnothing) : (x, y) \in E\}$ *lokal pro Unterroutinenaufruf*
11) ***solange*** *ein y mit* $|K \cap (\{y\} \times \mathcal{B})| = eing(y)$ *existiert*
12) $D = \prod_{\forall C \text{ mit } (y, C) \in K} C$
13) $K = K \setminus \left(\bigcup_{\forall C \text{ mit } (y, C) \in K} \{(y, C)\} \right)$
14) $(z, C) = baubaum(y)$
15) $C = D + C$
16) $K = K \cup \{(z, C)\}$
17) *wähle ein* $(y, C) \in K$
18) $eing(y) = eing(y) + 1 - |K \cap (\{y\} \times \mathcal{B})|$
19) $C = x + \prod_{\forall D \text{ mit } (y, D) \in K} D$
20) ***Rücksprung*** *unter Rückgabe von (y, C)*

Abbildung 24 - Algorithmus „GzuB“.

Die Unterroutine „baubaum(x)“ erstellt dabei den Konstruktionsbaum B_x zum Teilgraphen G_x (siehe Lemma 22). Dazu ermittelt die Unterroutine zuerst, ob Nachfolger y des Knotens x ebenfalls in V_x liegen. Sollte dies der Fall sein, so wird durch rekursives Aufrufen der Unterroutine der Konstruktionsbaum B_y zum Teilgraphen G_y berechnet. Anschließend überprüft die Unterroutine „baubaum(x)“, ob Nachfolger der Senken von G_y in V_x liegen. Wenn ja, so wird für diese Nachfolger ebenfalls die Unterroutine rekursiv aufgerufen. Die entstandenen Konstruktionsbäume der Teilgraphen werden im Laufe der Ausführung von „baubaum(x)“ zu einem Konstruktionsbaum zusammengesetzt, welcher am Ende von „baubaum(x)“ zurückgegeben wird.
Ein Beispiel zum Ablauf des Algorithmus ist in der Abbildung 25 dargestellt.



baubaum(1): K = { (2, ∅); (3; ∅); (4, ∅) }
↳ baubaum(2): K = { (5, ∅); (6, ∅) }
  ↳ baubaum(5): K = { (7, ∅) } ⇒ Rückgabe (7, 5)
  ↳ baubaum(6): K = { (7, ∅) } ⇒ Rückgabe (7, 6)
  baubaum(2): K = { (7, 5); (7, 6) } ⇒ eing(7) = 2;
    Rückgabe (7, 2+(5‖6))
↳ baubaum(3): K = { (7, ∅) } ⇒ Rückgabe (7, 3)
↳ baubaum(4): K = { (8, ∅) }
  ↳ baubaum(8): K = { (9, ∅) } ⇒ Rückgabe (9, 8)
  baubaum(4): K = { (9, 8) } ⇒ Rückgabe(9, 4+8)
baubaum(1): K = { (7, 2+(5‖6)); (7, 3); (9, 4+8) }
↳ baubaum(7): K = { (9, ∅) } ⇒ Rückgabe (9, 7)
baubaum(1): K = { (9, 4+8); (9, 2+(5‖6)‖3+7) }
↳ baubaum(9): Rückgabe (∞, 9)
baubaum(1): K = { (∞, (4+8)‖(2+(5‖6)‖3+7)+9) }
  Rückgabe (∞, 1+((4+8)‖(2+(5‖6)‖3+7)+9))

Abbildung 25 - Beispiel eines „GzuB“ - Laufes.

Lemma 20: *Es sei G = (V, E) ein serien - paralleler Graph ohne komplexe Vereinigungen mit nur einer Quelle z. Außerdem sei ein $x \in V$ gegeben. Dann gilt für jedes $y \in V_x \setminus \{x\}$: $V_y \subset V_x$.*

Beweis: Es wird zuerst gezeigt, daß für jedes $v \in V_y$ auch $v \in V_x$ gilt: Wegen $v \in V_y$ folgt, daß jeder Pfad von z nach v den Knoten y enthält: $z \rightarrow y \rightarrow v$. Da $y \in V_x \setminus \{x\}$ ist, enthält jeder Pfad von z nach y den Knoten x: $z \rightarrow x \rightarrow y \rightarrow v$. Demzufolge enthält auch jeder Pfad von z nach v den Knoten x. Es gilt also $V_y \subseteq V_x$.

Jetzt wird noch gezeigt, daß $x \notin V_y$ ist. Da $y \neq x$ ist und es einen Pfad von x nach y gibt, enthält der zyklusfreie Graph keinen Pfad von y nach x. Demzufolge existiert auch kein Pfad von z nach x welcher den Knoten y enthält. Da $x \in V_x$ ist, folgt $V_y \neq V_x$.

■

Lemma 21: *Es sei G = (V, E) ein serien - paralleler Graph ohne komplexe Vereinigungen mit nur einer Quelle. Dann gelten bei der Ausführung des in der Abbildung 24 angegebenen Algorithmus „GzuB“ die folgenden Aussagen:*

a) Wird beim Durchlaufen der Zeile 14) innerhalb der Unterroutine „baubaum(x)“ die Unterroutine „baubaum(y)“ aufgerufen, so existiert in G ein Pfad von x nach y.

b) In der Unterroutine „baubaum(x)“ existiert für alle Tupel $(y, C) \in K$ mit $y \neq \infty$ ein Pfad von x nach y. Dieser Pfad führt (abgesehen von den Knoten x und y) nur über in C enthaltene Knoten.

Beweis: Es sei $\mathfrak{B}$ die Menge aller Konstruktionsbäume inklusive des leeren Baumes.
Die Zeile 11) garantiert, daß die Unterroutine „baubaum(y)“ nur aufgerufen werden kann, wenn ein Tupel $(y, C) \in K$ mit $C \in \mathfrak{B}$ existiert. Es ist also ausreichend zu zeigen, daß in den Zeilen 10) und 16) nur Tupel (z, C) in K eingefügt werden, bei denen ein Pfad von x nach z

existiert, welcher nur über Knoten aus C führt. Die in der Zeile 10) eingefügten Tupel erfüllen diese Eigenschaft.
Angenommen es gibt während der Ausführung von „GzuB“ einen Zeitpunkt, zu dem in der Zeile 16) einer beliebigen Unterroutine „baubaum(x)“ ein Tupel (z, C) in K eingefügt wird, für welches kein Pfad von x nach z über die in C enthaltenen Knoten existiert. Gibt es mehrere dieser Zeitpunkte, so betrachte den ersten Zeitpunkt.
Das Tupel (z, C) kann nur in der Zeile 16) in K eingefügt werden, wenn der zuvor in der Zeile 14) erfolgte Aufruf der Unterroutine „baubaum(y)“ den Rückgabewert (z, D) mit $D \in \mathcal{B}$ geliefert hat. Da das Tupel (z, C) das erste Tupel in K ist, für welches kein Pfad von x nach z existiert, gab es in Zeile 12) für alle Tupel $(y, D_j) \in K$ einen Pfad von x nach y, welcher nur über Knoten aus D_j führt.
Man betrachte nun die Ausführung der Unterroutine „baubaum(y)“. Da diese Routine das Tupel (z, D) zurück liefert, muß in der Zeile 17) das Tupel (z, D') mit $D' \in \mathcal{B}$ in K enthalten sein. Demzufolge existiert ein Pfad von y nach z, welcher nur Knoten aus D' enthält. Wie man anhand der Erstellung des zurückgegebenen Konstruktionsbaums D in Zeile 19) erkennt, ist y als Quelle in D enthalten und es gibt einen Pfad von y nach z, welcher nur Knoten aus D enthält.
Nach dem Rücksprung zu „baubaum(x)“ ergibt sich der Konstruktionsbaum

$$C = \left(\prod_{(y, D_j)} (y, D_j) \right) + D.$$

Aus den Eigenschaften von D und der D_j folgt die Existenz eines Pfades von x nach z, welcher nur über in C enthaltene Knoten führt. Dies ist aber ein Widerspruch zur Annahme.

■

Lemma 22: *Es sei G = (V, E) ein serien - paralleler Graph ohne komplexe Vereinigungen mit nur einer Quelle und nur einer Senke. Dann liefert die in der Abbildung 24 angegebene Unterroutine „baubaum(x)“ als ersten Ergebnisparameter einen Knoten y mit (z, y) ∈ E für alle Senken z des Teilgraphen G_x und als zweiten Wert den zu G_x gehörenden Konstruktionsbaum B_x zurück. Gibt es kein derartiges y, d.h., die Senke von G_x ist gleich der Senke des Modulabhängigkeitsgraphen G, so wird von „baubaum(x)“ der Wert y = ∞ zurück geliefert.*

Beweis: Unter Verwendung der Variablen aus Abbildung 24 seien für jeden Zeitpunkt während der Ausführung des Algorithmus „GzuB“ die folgenden Mengen definiert:

Q = { x : die Unterroutine „baubaum(x)“ hat die Zeile 10) mindestens einmal durchlaufen }

R = { x : die Unterroutine „baubaum(x)“ hat die Zeile 20) mindestens einmal durchlaufen }

$$L_x = \begin{cases} \emptyset & : \text{falls } x \in R \\ K, \text{ welches lokal in „baubaum(x)“ definiert ist} & : \text{falls } x \in Q \setminus R \\ \{ (y, \emptyset) : (x, y) \in E \} & : \text{falls } x \notin Q \end{cases}$$

$$U = \bigcup_{x \in V} L_x.$$

Enthalten mehrere L_x das gleiche Element, so ist dieses Element mit der entsprechenden Vielfachheit in U enthalten. Zusätzlich bezeichne U_x^0 die Menge U beim ersten Durchlaufen der Zeile 8) der Unterroutine „baubaum(x)“. Mit U_x^i bezeichne man die Menge U beim i-ten Durchlauf der Zeile 11) in der Unterroutine „baubaum(x)“. Die Bedingung der Zeile 11) ist beim j-ten Durchlauf der Zeile 11) erstmals nicht mehr erfüllt. Dann sei für alle $i > j$ die Menge $U_x^i = U_x^j$. Außerdem stehe U_x^∞ für die Menge $U \cup \{(y, C)\}$ direkt nach dem erstmaligen Verlassen der Unterroutine „baubaum(x)“ (es gelte bereits $x \in R$) mit dem Rückgabewert (y, C). Es bezeichne $\mathcal{B}$ die Menge aller Konstruktionsbäume inklusive des leeren Baumes.

In Beweis stehen (α_x^i) bis (γ_x^i) für die folgenden Aussagen:

(α_x^i) Genau dann, wenn für den Knoten $y \in V$ die Unterroutine „baubaum(y)“ noch nicht aufgerufen wurde ($y \notin Q$), ist

$$\left|U_x^i \cap (\{y\} \times \mathcal{B})\right| = \text{eing}(y).$$

(β_x^i) Genau dann, wenn für den Knoten $y \in V$ die Unterroutine „baubaum(y)“ bereits aufgerufen wurde ($y \in Q$), ist

$$\left|U_x^i \cap (\{y\} \times \mathcal{B})\right| = 0.$$

(γ_x^i) Für alle Tupel $(y, C) \in U_x^i$ mit $C \neq \emptyset$ gilt: Der Knoten y ist in G Nachfolger aller Senken des durch C repräsentierten serien - parallelen Graphen. Sollte in G kein derartiger Knoten $y \in V$ existieren, so ist $y = \infty$.

Mit Hilfe einer Induktion über $|V_x|$ wird nun die folgende Behauptung gezeigt:

Gelten beim Start von „baubaum(x)“ die folgenden fünf Aussagen:

(α) Es gilt (α_x^0).

(β) Es gilt (β_x^0).

(γ) Es gilt (γ_x^0).

(δ) Gilt für einen Knoten $y \in V$ die Aussage $\text{indeg}(y) = 1$, so ist $\text{eing}(y) = 1$.

(ε) Gilt für einen Knoten $y \in V$ die Aussage $\text{indeg}(y) \geqslant 2$, so ist $\text{eing}(y) \geqslant 2$.

(ζ) Für alle Knoten $y \in V$, alle Tupel $(z, C) \in L_y$ und alle in C enthaltenen Knoten u gilt $u \in V_y$.

Dann erfüllt der Rückgabewert von „baubaum(x)“ die Behauptung von Lemma 22 und es gelten nach dem Verlassen von „baubaum(x)“ die folgenden Aussagen:

(α') Es gilt (α_x^∞).

(β') Es gilt (β_x^∞).

(γ') Es gilt (γ_x^∞).

(δ') Gilt für einen Knoten $y \in V$ die Aussage $\text{indeg}(y) = 1$, so ist $\text{eing}(y) = 1$.

(ε') Gilt für einen Knoten $y \in V$ die Aussage $\text{indeg}(y) \geqslant 2$, so ist $\text{eing}(y) \geqslant 2$.

(ζ') Für alle Knoten $y \in V$, alle Tupel $(z, C) \in L_y$ und alle in C enthaltenen Knoten u gilt $u \in V_y$.

Vor dem Induktionsbeweis wird gezeigt, daß die Aussagen (α) bis (ζ) nach dem Aufruf von „baubaum(x)" in Zeile 5) erfüllt sind:
Die Zeile 2) garantiert die Gültigkeit von (δ) und (ε). Man sieht, daß vor dem Start von „baubaum(x)" $Q = R = \emptyset$ ist. Somit gilt für alle $y \in V$ die Gleichung

$$L_y = \{ (z, \emptyset) : z \in V \text{ mit } (y, z) \in E \}.$$

Aus der Definition von U_x^0 folgt

$$U_x^0 = \{ (z, \emptyset) : y, z \in V \text{ mit } (y, z) \in E \}.$$

Da für alle Tupel (z, C) in U_x^0 die Aussage $C = \emptyset$ gilt, sind (γ) und (ζ) erfüllt. Außerdem gilt für alle $y \in V$

$$\left|U_x^0 \cap (\{y\} \times \mathcal{E})\right| = \left|\{ (y, \emptyset) : z \in V \text{ mit } (z, y) \in E \}\right| = \text{indeg}(y) = \text{eing}(y).$$

Somit ist (α) ebenfalls erfüllt. Da x die Quelle des serien - parallelen Graphen ist, gilt indeg(x) = 0 und somit auch (β).

Im **Induktionsanfang** wird der Fall $|V_x| = 1$ gezeigt:

Fall 1: Der Knoten x ist die Senke des Modulabhängigkeitsgraphen G = (V, E).
Somit gilt

$$U_x^\infty = U_x^0 \cup \{(\infty, x)\}.$$

Da $\infty \notin V$ ist und (α) sowie (β) nach Induktionsvoraussetzung erfüllt waren, gelten (α') und (β'). Man sieht, daß (γ) und (ζ) durch die Hinzunahme des Tupels (∞, x) nicht invalidiert werden. Somit gelten auch (γ') und (ζ'). Da das Feld eing nicht verändert wird, gelten (δ') und (ε'). Deshalb erfüllt die Unterroutine „baubaum(x)" die Aussage von Lemma 22.

Fall 2: Der Knoten x ist nicht die Senke von G.
Demzufolge hat x in G mindestens ein Kind. Es sei y ein beliebiger Knoten mit $(x, y) \in E$. Da $|V_x| = 1$ ist, muß $y \notin V_x$ gelten und somit $\text{indeg}(y) \geqslant 2$ sein. Wegen (ε) ist auch $\text{eing}(y) \geqslant 2$.
Nach dem Durchlaufen der Zeile 10) gilt $|K \cap (\{y\} \times \mathcal{E})| = 1$. Demzufolge ist die Bedingung in Zeile 11) für keinen Knoten y erfüllt und die Zeilen 12) bis 16) werden nicht durchlaufen. Da der Graph G keine komplexen Vereinigungen hat und $\text{indeg}(y) \geqslant 2$ ist, folgt $\text{outdeg}(x) = 1$. Demzufolge gilt in der Zeile 17) $K = \{(y, \emptyset)\}$. Aus diesem Grund verändert die Anweisung in Zeile 18) den Wert von eing(y) nicht, d.h., die Aussagen (δ') und (ε') sind erfüllt. In Zeile 19) wird C = x gesetzt, so daß in Zeile 20) der korrekte Wert zurück gegeben wird. Damit ist die Aussage von Lemma 22 erfüllt.
Man sieht sofort, daß die Ausführung der Zeile 10) die Menge U nicht ändert. Folglich ist $U_x^1 = U_x^0$.
Beim Rücksprung in Zeile 20) wird $L_x = \emptyset$ gesetzt. Demzufolge ist

$$U_{\text{nach Rücksprung}} = U_{\text{vor Rücksprung}} \setminus \{ (y, \emptyset) \}$$

und somit

$$U_x^{\infty} = \left(U_x^0 \setminus \{(y, \varnothing)\}\right) \cup \{(y, C)\}.$$

Aus diesem Grund sind (α') und (β') ebenfalls erfüllt. Wegen (x, y) ∈ E, outdeg(x) = 1 und C = x erfüllt das hinzugefügte Tupel (y, C) die Aussagen (γ') und (ζ').

Induktionsschritt: Es sei $|V_x| \geqslant 2$.

Betrachtung der Zeilen 8) bis 10):
Es seien $y_1, ..., y_i$ die Nachfolger von x im Modulabhängigkeitsgraphen G. Ist $i \geqslant 2$, so folgt aus der Tatsache, daß G keine komplexen Vereinigungen enthält: $\text{indeg}(y_j) = 1$ für alle $j \in \{1; ...; i\}$. Ist dagegen i = 1, so muß wegen $|V_x| \geqslant 2$ auch $y_1 \in V_x$ sein. Dies ist aber nur möglich, wenn $\text{indeg}(y_1) = 1$ ist.

Da $\text{indeg}(y_j) = 1$ für alle $j \in \{1; ...; i\}$ ist, folgt $y_j \in V_x$. Wegen (δ) erfüllen die Tupel $(y_j, \varnothing)$ garantiert die Bedingung in Zeile 11). Man sieht außerdem, daß nach dem Durchlaufen der Zeile 10) $U_x^1 = U_x^0$ gilt. Aus diesem Grund gelten die Aussagen (α_x^1), (β_x^1) und (γ_x^1).

Betrachtung der Zeilen 11) bis 16):
In diesem Abschnitt wird gezeigt, daß aus der Gültigkeit von (α_x^i), (β_x^i) und (γ_x^i) beim i-ten Durchlauf der Zeile 11) auch die Gültigkeit von (α_x^{i+1}), (β_x^{i+1}) und (γ_x^{i+1}) folgt. Außerdem wird dargelegt, daß die Hinzunahme des Tupels (z, C) zu K in Zeile 16) die Bedingung (ζ) nicht verletzt. Da in den Zeilen 11) bis 16) das Feld eing nicht verändert wird, ist die Gültigkeit von (δ) und (ε) gewährleistet.
Fall 1: In der Zeile 11) werde ein y_j mit $(x, y_j) \in E$ gewählt.

Nach (α_x^i), der Bedingung in Zeile 11) und den bereits gemachten Überlegungen zu den Nachfolgern von x ergibt sich

$$U_x^i \cap (y_j \times \mathfrak{B}) = K \cap (y_j \times \mathfrak{B}) = \{(y_j, \varnothing)\}.$$

In der Zeile 12) wird demzufolge D = ∅ gesetzt und nach dem Durchlaufen der Zeile 13) gilt

$$U_{y_j}^0 = U_x^i \setminus \{(y_j, \varnothing)\}.$$

Damit sind $(\alpha_{y_j}^0)$, $(\beta_{y_j}^0)$ und $(\gamma_{y_j}^0)$ erfüllt. In Lemma 20 wurde gezeigt, daß $V_{y_j} \subset V_x$ ist. Demzufolge arbeitet nach Induktionsvoraussetzung die in Zeile 14) aufgerufene Routine „baubaum(y_j)" korrekt und es gelten die Aussagen $(\alpha_{y_j}^{\infty})$, $(\beta_{y_j}^{\infty})$ und $(\gamma_{y_j}^{\infty})$.

Wegen D = ∅, bleibt C in Zeile 15) unverändert. In Zeile 16) wird der Rückgabewert (z, C) von „baubaum(y_j)" in K eingefügt. Da nach Induktionsvoraussetzung die Aussage von Lemma 22 gilt, ist C der Konstruktionsbaum von G_{y_j} und enthält somit nur Knoten aus V_{y_j}. Aufgrund von $V_{y_j} \subset V_x$ enthält C also nur Knoten, die auch in V_x enthalten sind. Demzufolge ist (ζ) weiterhin erfüllt. Außerdem gilt für U_x^{i+1} nach der Ausführung von Zeile 16)

$$U_x^{i+1} = \left(U_{y_j}^{\infty} \setminus \{(z, C)\}\right) \cup \{(z, C)\} = U_{y_j}^{\infty}.$$

Somit sind auch $\left(\alpha_x^{i+1}\right)$, $\left(\beta_x^{i+1}\right)$ und $\left(\gamma_x^{i+1}\right)$ erfüllt.

Fall 2: In der Zeile 11) werde ein y mit $(x, y) \notin E$ gewählt.
Nach der Voraussetzung von Fall 2 existieren in K die Tupel $\left(y, D_j\right)$ für alle $j \in \{1; ...; \text{eing}(y)\}$. Da $\text{eing}(\infty) = \infty$ ist, gilt $y \neq \infty$. Mit H_j werde der durch D_j repräsentierte serien - parallele Graph bezeichnet. Es sei Z die Menge aller Senken aller Graphen H_j:

$$Z = \left\{ z : z \text{ ist Senke in } H_j \text{ mit } 1 \leqslant j \leqslant \text{eing}(y) \right\}.$$

Fall 2.1: Es existiere ein Knoten $u \in V$ mit $(u, y) \in E$ und $u \notin Z$.
Fall 2.1.1: Es gelte $u \notin Q$.
Laut Definition gilt $L_u \supseteq \{ (y, \varnothing) \}$. Wegen $\left(\alpha_x^i\right)$ und $L_u \cup L_x \subseteq U_x^i$ folgt

$$\left|U_x^i \cap (\{y\} \times \mathcal{B})\right| = \text{eing}(y)$$

$$\Rightarrow \quad \left|\left(L_u \cup L_x\right) \cap (\{y\} \times \mathcal{B})\right| \leqslant \text{eing}(y)$$

$$\Rightarrow \quad \left|L_x \cap (\{y\} \times \mathcal{B})\right| \leqslant \text{eing}(y) - 1$$

$$\Leftrightarrow \quad |K \cap (\{y\} \times \mathcal{B})| \leqslant \text{eing}(y) - 1,$$

was ein Widerspruch zur Wahl von y in Zeile 11) ist. Demzufolge kann der Fall 2.1.1 nicht eintreten.
Fall 2.1.2: Es sei $u \in Q$.
Fall 2.1.2.1: Es gelte $\text{indeg}(y) = 1$.
Wegen (δ) ist auch $\text{eing}(y) = 1$. Somit hat nach der Voraussetzung von Fall 2.1 y nur u als Vorgänger. Da die Bedingung in Zeile 11) erfüllt ist, existiert genau ein Tupel $\left(y, D_1\right) \in K \cap (\{y\} \times \mathcal{B})$.

Fall 2.1.2.1.1: Es sei $D_1 \neq \varnothing$.

Aus $\left(y, D_1\right) \in K$ folgt laut Definition auch $\left(y, D_1\right) \in U_x^i$. Wegen $\left(\gamma_x^i\right)$ muß dann y Nachfolger aller Senken von H_1 sein. Da y nur u als Vorgänger hat, jedoch u nach Voraussetzung von Fall 2.1 keine Senke von H_1 ist, kann der Fall 2.1.2.1.1 nicht eintreten.

Fall 2.1.2.1.2: Es gelte $D_1 = \varnothing$.

Nach Voraussetzung von Fall 2 ist y nicht Nachfolger von x. Demzufolge wurde das Tupel $(y, \varnothing)$ nicht in Zeile 10) in K eingefügt. Es kommt also nur noch Zeile 16) als Einfügestelle in Frage. Damit in Zeile 16) $C = \varnothing$ gilt, muß in Zeile 15) $C = \varnothing$ gegolten haben. Dies ist aber nur der Fall, wenn der Rückgabewert der Routine „baubaum“ in Zeile 14) $C = \varnothing$ geliefert hat. Betrachtet man nun die Rücksprungstellen der Routine „baubaum“, so erkennt man, daß sowohl beim Verlassen in Zeile 9), als auch beim Verlassen in Zeile 20) $C \neq \varnothing$ ist. Aus diesem Grund kann der Fall 2.1.2.1.2 nicht eintreten.
Fall 2.1.2.2: Es gelte $\text{indeg}(y) \geqslant 2$.
Fall 2.1.2.2.1: Es sei $u \notin R$.
Da „baubaum(u)“ noch nicht beendet ist und zur Zeit „baubaum(x)“ abgearbeitet wird, kann man wiederum zwei Fälle unterscheiden.

Fall 2.1.2.2.1.1: Es gelte x = u.
Wegen (u, y) ∈ E gilt (x, y) ∈ E, was ein Widerspruch zur Voraussetzung von Fall 2 ist. Demzufolge kann der Fall 2.1.2.2.1.1 nicht eintreten.
Fall 2.1.2.2.1.2: Es gelte x ≠ u.
In diesem Fall muß bei der Ausführung von „baubaum(u)“ die Zeile 14) durchlaufen wurden sein, um irgendwann rekursiv „baubaum(x)“ starten zu können. Da (u, y) ∈ E und indeg(y) ⩾ 2 ist, gilt outdeg(u) = 1 und somit $V_u = \{u\}$. Im Induktionsanfang wurde aber gezeigt, daß im Fall $|V_u| = 1$ die Zeile 14) nicht durchlaufen wird. Demzufolge kann der Fall 2.1.2.2.1.2 nicht eintreten.
Fall 2.1.2.2.2: Es sei u ∈ R.
Da (u, y) ∈ E und indeg(y) ⩾ 2 ist, gilt outdeg(u) = 1 und somit $V_u = \{u\}$. Nach Induktionsvoraussetzung liefert „baubaum(u)“ das Tupel (y, u) als Rückgabewert. Die nachfolgende Operation in Zeile 15) modifiziert das Tupel (y, u) in das Tupel (y, C) mit C ∈ $\mathfrak{B}$. Dabei ist u eine Senke des durch C repräsentierten serien - parallelen Graphen. In der Zeile 16) wird das Tupel (y, C) zu K hinzugefügt, d.h. (y, C) ∈ U.
Da nach Voraussetzung von Fall 2.1 kein Tupel (y, D_j) existiert, wobei H_j den Knoten u als Senke enthält, muß es einen Zeitpunkt geben, an dem entweder (y, C) aus U gelöscht wird, oder C so geändert wird, daß u nicht mehr Senke des durch C repräsentierten serien - parallelen Graphen ist. Dieser Zeitpunkt sei während der Ausführung von „baubaum(v)“ mit v ∈ V.
Fall 2.1.2.2.2.1: Die Operation in Zeile 10) lösche oder ändere (y, C).
Wie bereits gezeigt wurde, gilt $U_v^1 = U_v^0$. Aus diesem Grund ändert sich die Menge U beim Durchlaufen der Zeile 10) nicht und der Fall 2.1.2.2.2.1 kann nicht eintreten.
Fall 2.1.2.2.2.2: Die Operation in Zeile 13) lösche oder ändere (y, C).
Wenn das Tupel (y, C) in Zeile 13) aus L_v und damit auch aus U gelöscht wurde, so muß anschließend in Zeile 14) „baubaum(y)“ gestartet wurden sein. Wegen (β_x^i) gilt dann

$$\Rightarrow \quad \begin{aligned} &|U_x^i \cap (\{y\} \times \mathfrak{B})| = 0 \\ &|L_x \cap (\{y\} \times \mathfrak{B})| = 0. \end{aligned}$$

Andererseits ist wegen (δ) und (ε) eing(y) ⩾ 1. Somit kann der Knoten y beim i-ten Durchlauf der Zeile 11) in „baubaum(x)“ nicht gewählt werden, was ein Widerspruch zur Voraussetzung von Fall 2 ist. Demzufolge kann der Fall 2.1.2.2.2.2 nicht eintreten.
Fall 2.1.2.2.2.3: Die Operation in Zeile 16) lösche oder ändere (y, C).
In der Zeile 16) wird nur ein Tupel zu K und damit auch zu U hinzugefügt. Demzufolge ändern sich alle bereits in U enthaltenen Tupel nicht. Deshalb kann der Fall 2.1.2.2.2.3 nicht eintreten.
Fall 2.1.2.2.2.4: Beim Verlassen von „baubaum(v)“ in der Zeile 20) werde (y, C) gelöscht oder verändert.
Fall 2.1.2.2.2.4.1: Es gelte (y, C) ∉ L_v.
Da beim Rücksprung nur alle in L_v enthaltenen Tupel aus U gelöscht werden, ist das Tupel (y, C) auch nach dem Rücksprung noch in U enthalten. Somit kann der Fall 2.1.2.2.2.4.1 nicht eintreten.

Fall 2.1.2.2.2.4.2: Es gelte $(y, C) \in L_v$.

Fall 2.1.2.2.2.4.2.1: In Zeile 17) werde ein Tupel (w, Q) mit $w \neq y$ und $Q \in \mathcal{B}$ gewählt.
Bei der Ausführung von Zeile 17) gelte

$$\text{eing}(y) = |U \cap (\{y\} \times \mathcal{B})| + k$$

für ein $k \in \mathbb{Z}$. [Anmerkung: Leider ist an dieser Stelle nicht zwingend $|V_v| < |V_x|$ gegeben. Somit kann die Induktionsvoraussetzung $\left(\alpha_x^i\right)$ nicht verwendet werden.]

Da in Zeile 19) nur eing(w) verändert und beim Rücksprung in Zeile 20) das Tupel (y, C) aus L_v und somit auch aus U gelöscht wird, gilt nach dem Verlassen der Unterroutine „baubaum(v)“

$$\text{eing}(y) \geq \left|U_v^\infty \cap (\{y\} \times \mathcal{B})\right| + k + 1.$$

Beim Durchlaufen der Zeile 4) und bei der Wahl von y beim i-ten Durchlauf der Zeile 11) in „baubaum(x)“ gilt jedoch

$$\text{eing}(y) = |U \cap (\{y\} \times \mathcal{B})|.$$

Da „baubaum(v)“ vor dem i-ten Durchlauf der Zeile 11) in „baubaum(x)“ beendet wurde, muß es einen Zeitpunkt geben, zu dem entweder eing(y) verkleinert oder ein Tupel (y, R) mit $R \in \mathcal{B}$ zu U hinzugefügt wird. Dieser Zeitpunkt liege während der Ausführung von „baubaum(r)“.

Fall 2.1.2.2.2.4.2.1.1: In Zeile 10) werde ein Tupel (y, R) zu U hinzugefügt.
Wie man bereits gesehen hat, wird U beim Durchlaufen der Zeile 10) nicht verändert. Somit kann der Fall 2.1.2.2.2.4.2.1.1 nicht eintreten.

Fall 2.1.2.2.2.4.2.1.2: In Zeile 16) werde ein Tupel (y, R) zu U hinzugefügt.
Damit in der Zeile 16) das Tupel (y, R) zu U hinzugefügt werden kann, muß die zuvor aufgerufene Unterroutine „baubaum(z)“ das Tupel (y, A) zurück liefern. Dies ist aber nur möglich, wenn in Zeile 17) von „baubaum(z)“ der Knoten y ausgewählt wurde. Beim Rücksprung aus „baubaum(z)“ werden genau $|L_z \cap (\{y\} \times \mathcal{B})|$ Tupel mit dem Knoten y als ersten Wert aus U gelöscht. Betrachtet man nun die Änderungen der Variable eing(y) in Zeile 18), so erkennt man, daß der Wert um die $|L_z \cap (\{y\} \times \mathcal{B})|$ gelöschten Tupel verkleinert und zusätzlich um Eins für das anschließend neu in Zeile 16) von „baubaum(r)“ eingefügte Tupel wieder erhöht wird. Da das Hinzufügen des Tupels (y, R) in Zeile 16) bereits in eing(y) berücksichtigt ist, kann der Fall 2.1.2.2.2.4.2.1.2 nicht eintreten.

Fall 2.1.2.2.2.4.2.1.3: In Zeile 18) werde eing(y) erhöht.
Die Wahl des Knotens y in der Zeile 17) von „baubaum(r)“ garantiert $|L_r \cap (\{y\} \times \mathcal{B})| \geq 1$. Dadurch kann anschließend eing(y) nie erhöht werden. Somit kann der Fall 2.1.2.2.2.4.2.1.3 nicht eintreten.

Fall 2.1.2.2.2.4.2.2: In Zeile 17) werde ein Tupel (y, Q) mit $Q \in \mathcal{B}$ gewählt.
Die Berechnung des neuen Konstruktionsbaumes R der Zeile 19) garantiert, daß u wiederum eine Senke des durch R repräsentierten serien - parallelen Graphen ist. Beim Verlassen der Unterroutine „baubaum(v)“ wird das Tupel (y, R) zurückgegeben. Nach dem Rücksprung werde die Unterroutine „baubaum(r)“ ausgeführt. Dort wird der Konstruktionsbaum R in Zeile 15) nochmals modifiziert. Dabei entstehe der Konstruktionsbaum A. Aber auch die Operation in der Zeile 15) garantiert, daß u weiterhin eine Senke des durch A repräsentierten serien - parallelen Graphen ist. In der Zeile 16) wird das Tupel (y, A) in L_r und somit auch

in U eingefügt. Demzufolge ist in U wieder ein Tupel (y, A) enthalten, bei dem u eine Senke des durch A repräsentierten serien - parallelen Graphen ist. Aus diesem Grund kann der Fall 2.1.2.2.2.4.2.2 nicht eintreten.
Da keiner der Unterfälle von Fall 2.1 eintreten kann, kann auch der Fall 2.1 nicht eintreten.

Fall 2.2: Für alle Knoten $u \in V$ mit $(u, y) \in E$ gelte $u \in Z$.
Wegen der Aussage (ζ) gilt für jeden Knoten v eines jeden Graphen H_j mit $1 \leqslant j \leqslant \text{eing}(y)$ die Beziehung $v \in V_x$. Demzufolge gilt auch für alle Knoten $u \in Z$ die Beziehung $u \in V_x$, d.h. $Z \subseteq V_x$. Nach der Voraussetzung von Fall 2.2 liegen alle Vorgänger von y in Z und somit muß auch $y \in V_x$ gelten. Außerdem ist $y \neq x$ und mittels Lemma 20 folgt $V_y \subset V_x$ und $|V_y| < |V_x|$.
Aufgrund der Bedingung in Zeile 11) und der Gültigkeit von (α_x^i) werden in der Zeile 13) alle Tupel der Form (y, D) mit $D \in \mathcal{E}$ aus K und somit auch aus U gelöscht. Es folgt also

$$U_y^0 = U_x^i \setminus (\{y\} \times \mathcal{E}).$$

Nach dem Start von „baubaum(y)“ in Zeile 14) ist die Aussage von (β_y^0) für den Knoten y erfüllt. Für alle anderen Knoten folgt aus der Gültigkeit von (α_x^i) bzw. (β_x^i) die Gültigkeit von (α_y^0) bzw. (β_y^0). Da in den Zeilen 12) und 13) keine Tupel zu U hinzugefügt werden, läßt sich von (γ_x^i) die Gültigkeit von (γ_y^0) ableiten. Außerdem sind (δ), (ε) und (ζ) weiterhin wahr. Somit arbeitet nach Induktionsvoraussetzung die in Zeile 14) aufgerufene Unterroutine „baubaum(y)“ korrekt. Als Rückgabewert erhalte man das Tupel (z, C). Dabei ist C ein Konstruktionsbaum des serien - parallelen Graphen G_y. In C sind wegen $V_y \subset V_x$ nur Knoten aus V_x enthalten.

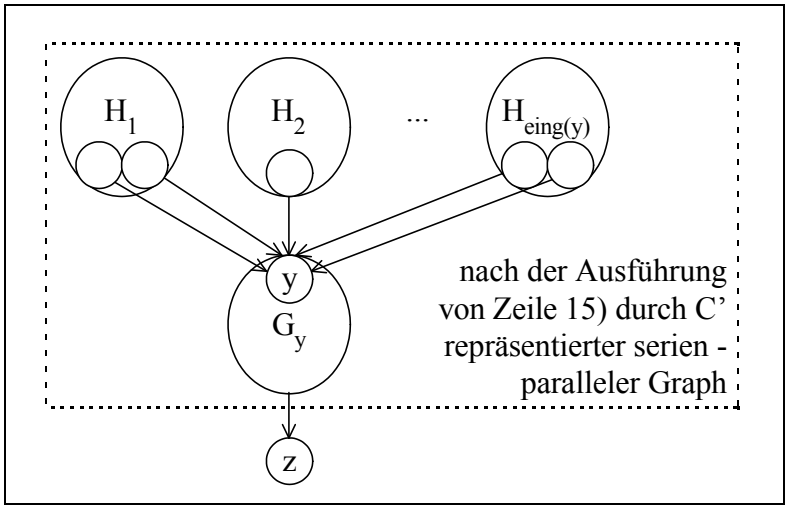


Abbildung 26 - serielle Vereinigung oberhalb vom Knoten y.

Betrachtet man nun den Modulabhängigkeitsgraphen G in der Umgebung von y, so hat er das in der Abbildung 26 dargestellte Aussehen. Die Zeile 12) nutzt diese Struktur, um die Konstruktionsbäume D_j mit $1 \leqslant j \leqslant \text{eing}(y)$ zu einem neuen Konstruktionsbaum D parallel zu

vereinigen. Dabei ist zu beachten, daß alle D_j wegen (ζ) nur Knoten aus V_x enthalten. Anschließend wird in Zeile 15) noch D und C seriell vereinigt. Dabei entstehe der Konstruktionsbaum C'. Dieser enthält wiederum nur Knoten aus V_x, was die Gültigkeit von (ζ) weiterhin garantiert. Nach dem Hinzufügen des Tupels (z, C') zu K in Zeile 16) gilt

$$U_x^{i+1} = \left(U_y^{\infty} \setminus (z, C)\right) \cup (z, C').$$

Wegen $\left(\alpha_y^{\infty}\right)$ und $\left(\beta_y^{\infty}\right)$ sind somit auch $\left(\alpha_x^{i+1}\right)$ und $\left(\beta_x^{i+1}\right)$ erfüllt. Die Tatsache, daß z eine Senke des durch C repräsentierten serien - parallelen Graphen ist und die Konstruktion von C' in Zeile 15) garantieren, daß z auch eine Senke des durch C' repräsentierten serien - parallelen Graphen ist. Somit folgt aus der Gültigkeit von $\left(\gamma_y^{\infty}\right)$ die Gültigkeit von $\left(\gamma_x^{i+1}\right)$. Die Aussagen (δ) und (ε) sind weiterhin gültig.

Betrachtung der Zeilen 17) bis 20):
Fall 3: Beim i-ten Durchlauf der Zeile 11) sei die Bedingung in der Zeile 11) für kein y mehr erfüllt.
Fall 3.1: Es existiere mindestens ein Knoten $y \in V_x$ mit $(y, C) \in K$.
Existieren mehrere Knoten $y_1, ..., y_k$ mit $y_j \in V_x$ und $\left(y_j, C_j\right) \in K$ $(1 \leqslant j \leqslant k)$, so setze $y = y_m$, wobei für alle $j \in \{1, ..., m - 1, m + 1, ..., k\}$ gilt: Es gibt in G keinen Pfad von y_j nach y_m.
Da $(y, C) \in K \subseteq U_x^i$ ist, folgt wegen $\left(\alpha_x^i\right)$

$$\left|U_x^i \cap (\{y\} \times \mathfrak{B})\right| = \text{eing}(y).$$

Zusammen mit der Voraussetzung von Fall 3, welche $|K \cap (\{y\} \times \mathfrak{B})| \neq \text{eing}(y)$ lautet, ergibt sich daraus

$$X = \left(U_x^i \cap (\{y\} \times \mathfrak{B})\right) \setminus (K \cap (\{y\} \times \mathfrak{B})) \neq \emptyset.$$

Es sei $(y, D) \in X$ mit $D \in \mathfrak{B}$ ein beliebiges Element von X. Wegen $X \subseteq U_x^i$ folgt mit Hilfe der Definition von U, daß es beim i-ten Durchlauf der Zeile 11) von „baubaum(x)" ein L_z mit $(y, D) \in L_z$ geben muß.
Fall 3.1.1: Es sei $z \notin Q$, d.h., die Zeile 10) der Unterroutine „baubaum(z)" wurde noch nicht durchlaufen.
Aus der Definition von L_z folgt, daß

$$(z, y) \in E \qquad (1)$$

eine Kante des Modulabhängigkeitsgraphen ist. Da $y \in V_x$ ist, muß $V_y \subseteq V_x \setminus \{x\}$ sein. Aufgrund von $y \in V_y$ und (1) folgt nun $z \in V_x$. Außerdem ist $x \in Q$, woraus man mit Hilfe der Voraussetzung von Fall 3.1.1 $z \neq x$ und somit $z \in V_x \setminus \{x\}$ folgern kann. Es sei

$$Z = \left\{ v : v \in V_x \setminus \{x\} \text{ und es gibt in G einen Pfad } v \rightarrow z \right\} \setminus \{z\}.$$

Fall 3.1.1.1: Beim i-ten Durchlauf der Zeile 11) von „baubaum(x)“ existiere ein (v, D) ∈ K mit v ∈ Z für ein beliebiges D ∈ $\mathcal{B}$.
Die Existenz von v ist ein Widerspruch zur Wahl von y, da es den Pfad v→z→y gibt. Demzufolge kann der Fall 3.1.1.1 nicht eintreten.
Fall 3.1.1.2: Es sei Z = ∅.
Da z ∈ V_x \ {x} ist und es keinen Knoten auf einem Pfad von x nach z gibt, folgt (x, z) ∈ E. Wie bereits am Anfang des Induktionsschrittes gezeigt wurde, kann in diesem Fall der Knoten z garantiert in Zeile 11) ausgewählt und „baubaum(z)“ gestartet werden. Da aber nach Voraussetzung von Fall 3.1.1 z ∉ Q ist und nach Voraussetzung von Fall 3 kein Knoten mehr in Zeile 11) ausgewählt werden kann, ergibt sich ein Widerspruch. Somit kann der Fall 3.1.1.2 nicht eintreten.
Fall 3.1.1.3: Beim i-ten Durchlauf der Zeile 11) von „baubaum(x)“ gelte für alle v ∈ Z und alle D ∈ $\mathcal{B}$ die Aussage (v, D) ∉ K. Außerdem sei Z ≠ ∅.
Da Z ≠ ∅ ist, existiert ein Pfad x→w→z welcher den Knoten w ∈ Z mit (x, w) ∈ E enthält. Demzufolge wurde beim Durchlaufen der Zeile 10) der Unterroutine „baubaum(x)“ das Tupel (w, ∅) in K eingefügt. Es existiert somit ein Durchlauf der Zeile 11) von „baubaum(x)“, bei dem in K ein Tupel (w, D) mit D ∈ $\mathcal{B}$ enthalten ist.
Beim j-ten Durchlauf der Zeile 11) von „baubaum(x)“ enthalte K letztmalig ein Tupel (v, D) mit v ∈ Z und D ∈ $\mathcal{B}$. Demzufolge gilt für alle k > j: Beim k-ten Durchlauf der Zeile 11) von „baubaum(x)“ ist

$$K \cap (Z \times \mathcal{B}) = \emptyset.$$

Die Voraussetzung von Fall 3.1.1.3 garantiert die Existenz der Zahl j < i.
Beim j-ten Durchlauf der Zeile 11) von „baubaum(x)“ werde der Knoten v in der Zeile 11) gewählt. Anschließend wird in der Zeile 14) „baubaum(v)“ gestartet. Nach Induktionsvoraussetzung arbeitet „baubaum(v)“ korrekt und liefert (u, D) zurück. In der Zeile 16) wird dann das Tupel (u, D’) zu K hinzugefügt.
Aus der Definition von Z und der Wahl von v ∈ Z folgt die Existenz des Pfades v→z. Der Pfad v→z enthalte die Knoten $v = w_1, w_2, ..., w_{k-1}, w_k = z$. Es sei m ∈ {1, ..., k} die größte Zahl, so daß $w_m \in V_v$ ist. Da Nach Voraussetzung von Fall 3.1.1 „baubaum(z)“ noch nicht gestartet wurde, kann z kein Knoten von G_v sein und es gilt m < k. Die Wahl von w_m garantiert, daß w_m eine Senke von G_v ist. Da „baubaum(v)“ das Tupel (u, D) zurück geliefert hat, ist u nach Induktionsvoraussetzung Nachfolger aller Senken von G_v. Somit gilt $(w_m, u) \in E$. Andererseits folgt aus $w_{m+1} \notin V_v$ die Beziehung $\text{indeg}(w_{m+1}) \geqslant 2$. Da G keine komplexen Vereinigungen enthält, muß $\text{outdeg}(w_m) = 1$ sein und w_m hat als einzigen Nachfolger den Knoten $w_{m+1} = u$.
Fall 3.1.1.3.1: Es gelte m + 1 < k.
Demzufolge existiert der Pfad u→z in G und es ist u ∈ Z. Da das Tupel (u, D’) beim (j + 1)-ten Durchlauf der Zeile 11) von „baubaum(x)“ in K enthalten ist, ergibt sich ein Widerspruch zur Wahl von j. Somit kann der Fall 3.1.1.3.1 nicht eintreten.
Fall 3.1.1.3.2: Es sei m + 1 = k.
In diesem Fall ist u = z. Demzufolge ist das Tupel (z, D’) ab dem (j + 1)-ten Durchlauf der Zeile 11) von „baubaum(x)“ in K enthalten. Da z ∉ Q ist, wird „baubaum(z)“ bis zum i-ten Durchlauf der Zeile 11) von „baubaum(x)“ nicht gestartet. Somit ist das Tupel (z, D’) auch

beim i-ten Durchlauf der Zeile 11) von „baubaum(x)“ in K enthalten. Wegen (1) existiert ein Pfad von z nach y, was mit (z, D') $\in$ K ein Widerspruch zur Wahl von y ist. Somit kann der Fall 3.1.1.3.2 nicht eintreten.

Fall 3.1.2: Es sei z $\in$ Q \ R, d.h., die Zeile 10) der Unterroutine „baubaum(z)“ wurde bereits durchlaufen, die Zeile 20) jedoch noch nicht.

Da „baubaum(z)“ noch nicht beendet ist und „baubaum(x)“ aktuell bearbeitet wird, wurde „baubaum(x)“ direkt oder indirekt von „baubaum(z)“ aufgerufen. Nach Lemma 21a) existiert dann in G ein Pfad von z nach x. Demzufolge gilt z $\notin V_x$.

Da L_z nur während der Ausführung „baubaum(z)“ verändert wird, und (y, D) $\in L_z$ ist, muß das Tupel (y, D) bereits vor dem Start von „baubaum(x)“ in L_z enthalten sein. Man betrachte nun den Zeitpunkt, zu dem das Tupel (y, D) in L_z eingefügt wurde.

Fall 3.1.2.1: Das Tupel (y, D) ist seit dem Start des Algorithmus in L_z enthalten.

In diesem Fall gilt (z, y) $\in$ E, d.h., es gibt in G einen Pfad von z nach y, welcher nicht über x führt. Somit gilt y $\notin V_x$, was ein Widerspruch zur Voraussetzung von Fall 3.1 ist. Deshalb kann der Fall 3.1.2.1 nicht eintreten.

Fall 3.1.2.2: Das Tupel (y, D) wird während der Ausführung von „baubaum(z)“ in L_z eingefügt.

Da sich L_z bei der Ausführung von Zeile 10) nicht ändert, muß das Tupel (y, D) in der Zeile 16) eingefügt wurden sein. Das Lemma 21b) garantiert dann, daß es einen Pfad von z nach y gibt, welcher nur über in D enthaltene Knoten führt. Die einzige Zeile, in welcher der Algorithmus „GzuB“ einen neuen Knoten zu einem Konstruktionsbaum hinzufügt, ist die Zeile 19). Demzufolge kann der Knoten x nur in einem Konstruktionsbaum enthalten sein, wenn „baubaum(x)“ bereits beendet wurde. Da dies nicht der Fall ist, kann der Konstruktionsbaum D den Knoten x nicht enthalten. Aus diesem Grund existiert ein Pfad von z nach y, welcher den Knoten x nicht enthält. Somit gilt y $\notin V_x$, was ein Widerspruch zur Voraussetzung von Fall 3.1 ist. Deshalb kann der Fall 3.1.2.2 nicht eintreten.

Fall 3.1.3: Es sei z $\in$ R.

Aus der Definition von L_z folgt, daß $L_z = \varnothing$ ist. Wegen (y, C) $\in L_z$ kann dieser Fall nicht eintreten.

Da alle Unterfälle von Fall 3.1 nicht eintreten können, kann auch der Fall 3.1 nie eintreten.

Fall 3.2: Für alle Tupel (y, C) $\in$ K gelte y $\notin V_x$.

Fall 3.2.1: In K existieren die Tupel (y, C) und (z, D) mit y $\neq$ z.

Man teile die Vorgänger von y in zwei Gruppen ein: Für die Vorgänger $a_1, ..., a_i$ gilt $a_k \in V_x$ und für die Vorgänger $b_1, ..., b_j$ gilt $b_k \notin V_x$. Da die Senken von C nach (ζ) in V_x liegen und y Nachfolger der Senken von C ist, folgt i $\geqslant$ 1. Andererseits gilt y $\notin V_x$. Demzufolge muß auch j $\geqslant$ 1 sein. Da G keine komplexen Vereinigungen enthält und indeg(y) = i + j $\geqslant$ 2 ist, folgt $\text{outdeg}(a_k) = 1$ (für alle k $\leqslant$ i) und $\text{outdeg}(b_k) = 1$ (für alle k $\leqslant$ j). Die gleichen Aussagen kann man für den Knoten z mit den Vorgängern $c_k \in V_x$ (k $\in \{1, ..., \hat{\imath})\}$ und $d_k \notin V_x$ (k $\in \{1, ..., \hat{\jmath})\}$ herleiten. Da alle a_k, b_k, c_k und d_k den Ausgangsgrad 1 haben, können y und z keinen gemeinsamen Vorgänger besitzen. Somit sind alle Vorgängerknoten von y und z verschieden.

Da G ein serien - paralleler Graph ist, werden Kanten nur bei seriellen Vereinigungen hinzugefügt. Demzufolge müssen unter anderem die folgenden zwei seriellen Vereinigungen durchgeführt werden (vgl. Abbildung 27):

a) Zum Zeitpunkt t_a wird der Teilgraph G_{1a} mit den Senken a_k (für alle $k \leqslant i$) und b_k (für alle $k \leqslant j$) mit dem Teilgraphen G_{2a} mit der Quelle y seriell vereinigt. Der Zeitpunkt t_a existiert, da alle Kanten zu einem Knoten innerhalb einer seriellen Vereinigung eingefügt werden müssen.
b) Zum Zeitpunkt t_b wird der Teilgraph G_{1b}, welcher den Knoten x enthält, mit dem Teilgraphen G_{2b}, welcher die Knoten a_k (für alle $k \leqslant i$) und c_k (für alle $k \leqslant \hat{\imath}$) enthält, seriell vereinigt. Die Existenz dieser Vereinigung kann man wie folgt nachweisen: Angenommen der Teilgraph G_{2b} enthält nicht alle Knoten a_k und c_k, sondern nur einen Teil von ihnen. Ohne Beschränkung der Allgemeinheit sei der Knoten a_1 in G_{2b} enthalten. Da es einen Pfad von x zu allen a_k (für alle $k \leqslant i$) und c_k (für alle $k \leqslant \hat{\imath}$) gibt, muß nachfolgend mindestens ein weiterer Teilgraph G_{3b}, welcher mindestens einen Knoten $e \in \{a_2, ..., a_i, c_1, ..., c_{\hat{\imath}}\}$ enthält, mit einem aus $G_{1b} + G_{2b}$ entstandenen Teilgraphen seriell vereinigt werden. Nach Lemma 15 existiert dann ein Pfad von a_1 zu e. Da a_1 als einzigen Nachfolger den Knoten y hat, führt auch ein Pfad von y zu e oder es gilt y = e. Da aber $y \notin V_x$ ist, folgt $e \notin V_x$. Dies ist ein Widerspruch zur Wahl der a_k und c_k.

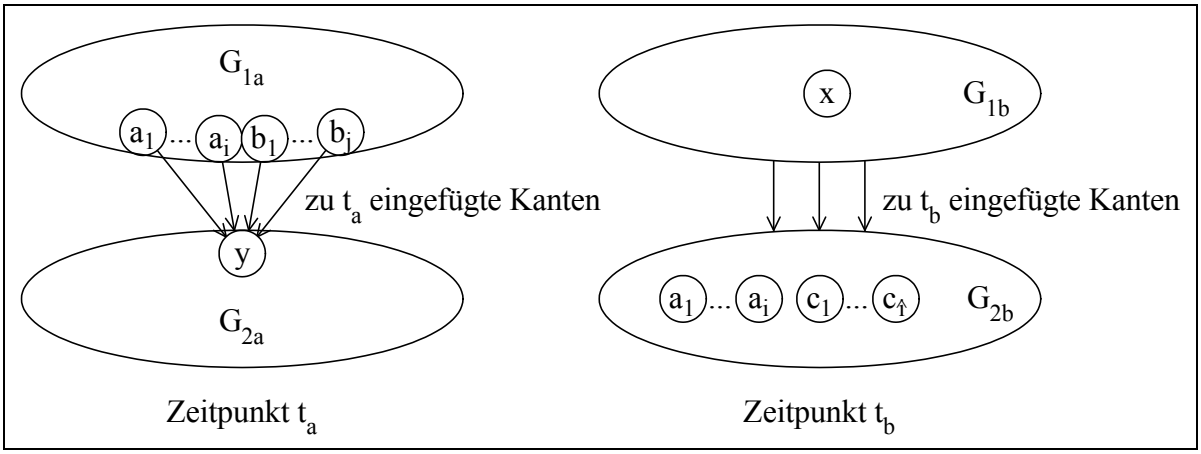


Abbildung 27 - zwei Vereinigungszeitpunkte.

Fall 3.2.1.1: Es gelte $t_a < t_b$.

Da der Knoten a_1 in G_{1a} und G_{2b} enthalten ist, muß wegen $t_a < t_b$ auch $G_{1a} + G_{2a} \subseteq G_{2b}$ gelten. Demzufolge ist b_1 in G_{2b} enthalten. Es sei e eine beliebige Quelle und f eine beliebige Senke in G_{1b}. Dann enthalten alle Pfade von e nach f den Knoten x. Wäre dies nicht der Fall, so gäbe es nach Lemma 15 in $G_{1b} + G_{2b}$ auch einen Pfad von e nach a_1, welcher nicht über x führt, d.h. $a_1 \notin V_x$. Demzufolge führen in $G_{1b} + G_{2b}$ auch alle Pfade von e nach b_1 über x. Dies ist aber ein Widerspruch zur Wahl von b_1. Somit kann der Fall 3.2.1.1 nicht eintreten.

Fall 3.2.1.2: Es gelte $t_a > t_b$.
Da die Knoten a_k (für alle $k \leqslant i$) im Teilgraphen G_{1a} Senken sind, müssen sie auch Senken im Teilgraphen G_{2b} sein.
Angenommen ein weiterer Knoten $e \neq a_k$ (für alle $k \leqslant i$) ist Senke in G_{2b}. Dann ist e ebenfalls Senke Teilgraphen G_{1a} und es existiert in $G_{1a} + G_{2a}$ und damit auch G eine Kante von e nach y. Demzufolge muß $e = b_k$ für ein $k \in \{1; ...; j\}$ sein. Analog zu den im Fall 3.2.1.1 gemachten Überlegungen erkennt man, daß dann auch alle Pfade von der Wurzel des Graphen G nach e über x führen. Dies ist aber ein Widerspruch zur Wahl der b_k. Somit hat G_{2b} nur die Knoten a_k als Senken.
Da c_1 keine Senke von G_{2b} ist, existiert in G_{2b} von c_1 ein Pfad zu mindestens einer Senke a_h (für ein $h \leqslant i$). Laut Voraussetzung hat c_1 als einzigen Nachfolger den Knoten z. Aus diesem Grund existiert ein Pfad von z nach a_h. Da $z \notin V_x$ und $a_h \in V_x$ gilt, muß der Pfad von z nach a_h über x führen. Es existiert also ein Pfad von c_1 nach x. Wegen $c_1 \in V_x$ existiert aber auch ein Pfad von x nach c_1. Da jeder serien - paralleler Graph zyklusfrei ist, ergibt sich auch im Fall $t_b < t_a$ ein Widerspruch. Somit kann der Fall 3.2.1.2 nicht eintreten.
Fall 3.2.1.3: Es gelte $t_a = t_b$.
In diesem Fall muß also $G_{1a} = G_{2a}$ und $G_{1b} = G_{2b}$ sein. Da aber $a_1 \in G_{1a}$, $a_1 \in G_{2b}$ und $G_{1b} \cap G_{2b} = \varnothing$ ist, kann dieser Fall auch nicht eintreten.

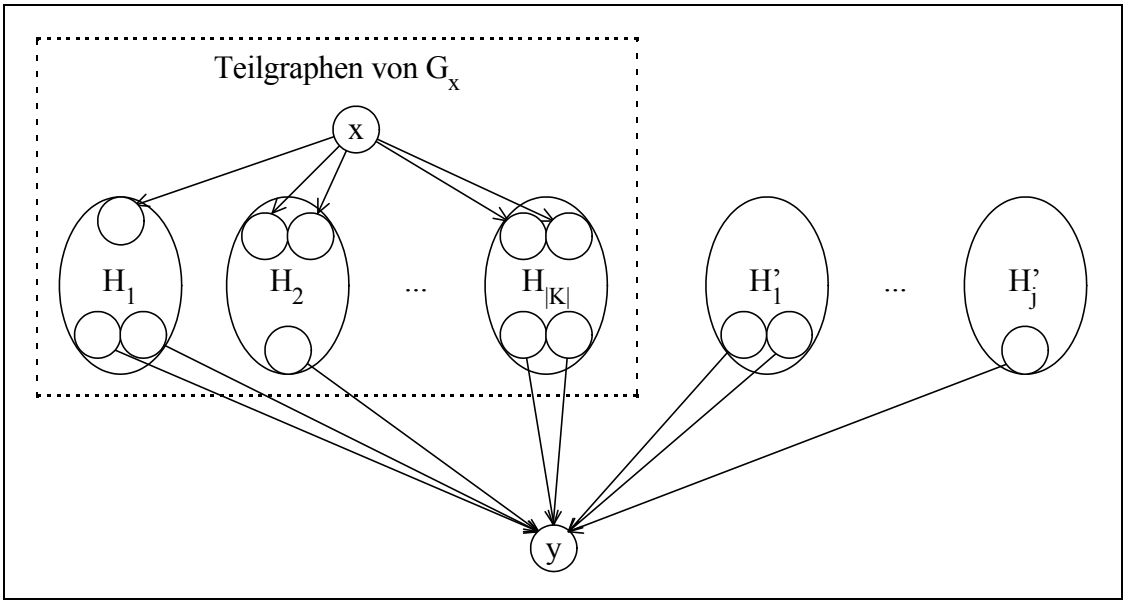


Abbildung 28 - parallele Vereinigung vor dem Verlassen von „baubaum(x)“.

Fall 3.2.2: Beim Durchlaufen der Zeile 17) haben alle Tupel in K den gleichen ersten Wert y. Es sei

$$K = \{ (y, D_1); (y, D_2); ...; (y, D_{|K|}) \}.$$

Mit H_k werde für alle $k \in [1; |K|]$ der von D_k repräsentierte serien - parallele Graph bezeichnet. Betrachtet man den Modulabhängigkeitsgraphen G in der Umgebung von y, so hat er das in der Abbildung 28 dargestellte Aussehen. Da $y \notin V_x$ ist, existieren außer den H_k weitere Teilgraphen H_1' bis H_j', deren Senken als Nachfolger den Knoten y haben.

Als nächstes wird gezeigt, daß die in der Zeile 19) durchgeführte parallele Vereinigung aller Teilgraphen H_k zur Erstellung des Konstruktionsbaumes C von $G_x = (V_x, E_x)$ führt.

Fall 3.2.2.1: Es existiere ein Knoten $z \in V_x \setminus \{x\}$, der in keinem der H_k enthalten ist.

Sollten mehrere Knoten in $V_x \setminus \{x\}$ existieren, die in keinem der H_k enthalten sind, so wähle den Knoten z derart, daß keiner der Vorgänger von z die für z geforderte Eigenschaft erfüllt.

Fall 3.2.2.1.1: Es gelte $(x, z) \in E$.

Wie gleich am Anfang des Induktionsschrittes gezeigt wurde (noch vor Fall 1), erfüllt das Tupel $(z, \emptyset)$ die Bedingung in der Zeile 11) und „baubaum(z)“ wird ausgeführt. Wie man bereits in Fall 1 gesehen hat, wird in Zeile 16) das Tupel (u, C) zu K hinzugefügt, wobei C nur den Knoten z enthält. Anhand der Operationen in den Zeilen 12), 15) und 16) erkennt man, daß in K immer ein Tupel (u', C') existiert, in dem der Knoten z im Konstruktionsbaum C' enthalten ist. Demzufolge ist z auch in einem D_k enthalten, was ein Widerspruch zur Wahl von z darstellt. Somit kann der Fall 3.2.2.1.1 nicht eintreten.

Fall 3.2.2.1.2: Es gelte $(x, z) \notin E$.

Man wähle einen Knoten u mit $(u, z) \in E$. Da $u \neq x$ und $z \in V_x \setminus \{x\}$ ist, gilt $u \in V_x \setminus \{x\}$. Die Wahl von z garantiert die Existenz eines $k \in [1; |K|]$, so daß u in H_k enthalten ist. Da z nicht in H_k enthalten ist, muß u eine Senke in H_k sein. Die Definition der H_k garantiert, daß das Tupel (y, D_k) in K enthalten ist. Da wegen (γ) y Nachfolger aller Senken von H_k ist, muß $y = z$ sein. Da der Fall 3.1 nicht eintreten kann, muß $y \notin V_x$ sein, was ein Widerspruch zur Wahl von z ist. Somit kann auch der Fall 3.2.2.1.2 nicht eintreten.

Fall 3.2.2.2: Jeder Knoten $z \in V_x \setminus \{x\}$ sei in mindestens einem der H_k enthalten.

Da ein neuer Knoten z nur in Zeile 19) von „baubaum(z)“ zu einem Konstruktionsbaum hinzugefügt wird und wegen (β), der Bedingung in Zeile 11) und (δ) bzw. (ε) die Routine „baubaum(z)“ höchstens einmal gestartet wird, kann der Knoten z nicht in zwei verschiedenen Teilgraphen H_k enthalten sein. Die Aussage (ζ) garantiert für alle $k \in [1; |K|]$ die Beziehung $H_k \subseteq G_x$. Somit sind alle in $V_x \setminus \{x\}$ enthaltenen Knoten in genau einem H_k enthalten. Zusammen mit der Eigenschaft, daß alle Senken der H_k den gleichen Nachfolger, nämlich den Knoten y, haben, gelangt man zu der Erkenntnis, daß der zu G_x gehörende Konstruktionsbaum in Zeile 19) korrekt erstellt wird.

Man betrachte weiterhin die Belegung der Variable K direkt vor dem Verlassen der Unterroutine „baubaum(x)“. Bekanntlich war die Bedingung in der Zeile 11) beim i-ten Durchlauf der Zeile 11) nicht mehr erfüllt. Man erkennt, daß

$$U_x^{\infty} = \left(U_x^{i} \setminus K\right) \cup (y, C)$$

ist. Demzufolge gilt

$$\left|U_x^{\infty} \cap (\{y\} \times \mathcal{B})\right| = \left|U_x^{i} \cap (\{y\} \times \mathcal{B})\right| - |K| + 1.$$

Wegen $K = K \cap (\{y\} \times \mathcal{B})$ könnte im Pseudocode die Zeile 18) auch

18) eing(y) = eing(y) + 1 - |K|

lauten.

Die Änderung von eing(y) in der Zeile 18) garantiert, die Gültigkeit von $\left(\alpha_x^\infty\right)$ bzgl. des Knotens y. Für alle anderen Knoten folgt die Gültigkeit von $\left(\alpha_x^\infty\right)$ direkt aus $\left(\alpha_x^i\right)$. Außerdem sind wegen $\left(\beta_x^i\right)$ und $\left(\gamma_x^i\right)$ auch $\left(\beta_x^\infty\right)$ und $\left(\gamma_x^\infty\right)$ erfüllt.

Da die Bedingung in der Zeile 11) für y nicht erfüllt war, galt vor dem Durchlaufen der Zeile 18) eing(y) > |K| ⩾ 1. Somit ist eing(y) ⩾ 2 und eing(y) - |K| ⩾ 1 bzw. eing(y) - |K| + 1 ⩾ 2. Demzufolge sind (δ) und (ε) auch nach dem Durchlaufen der Zeile 18) gültig.

In den Zeilen 17) bis 20) werden keine neuen Tupel in L_x eingefügt. Somit ist (ζ) ebenfalls weiterhin gültig.

■

Bei der Verwendung geeigneter Datenstrukturen (siehe Anhang 9.14) kann „GzuB“ in O(|E| + |V|) Schritten ausgeführt werden.

In [59] findet man ebenfalls einen Algorithmus zur Berechnung des Konstruktionsgraphen eines serien - parallelen Graphen ohne komplexe Vereinigungen in O(|E| + |V|). Aufgrund des Ablaufs des in [59] vorgestellten Algorithmus müssen die Hilfsknoten zum Auflösen der komplexen Vereinigungen vor dem Start des Algorithmus eingefügt und hinterher wieder gelöscht werden. Der Algorithmus „GzuB“ kann im Gegensatz dazu so implementiert werden, daß er ohne explizites Einfügen der Hilfsknoten den Konstruktionsbaum berechnet. Dadurch wird ein Lauf über den Modulabhängigkeitsgraphen und ein Lauf über den Konstruktionsbaum eingespart.

3.1.2.3 Der Konstruktionsbaum eines beliebigen serien - parallelen Graphen

Nachdem im Abschnitt 3.1.2.1 gezeigt wurde, wie ein beliebiger serien - paralleler Graph G = (V, E) in einen serien - parallelen Graphen G' = (V', E') ohne komplexe Vereinigungen überführt werden kann, und man für G' mit Hilfe des Algorithmus aus dem Abschnitt 3.1.2.2 ein Konstruktionsbaum B' berechnet hat, behandelt dieser Abschnitt den Übergang von B' zum Konstruktionsbaum B von G.

Es bezeichne K = V' \ V die Menge aller beim Übergang von G zu G' hinzugefügten Hilfsknoten. Die Konstruktion aus dem Abschnitt 3.1.2.1 garantiert für alle x ∈ K, daß zwei serien - parallele Teilgraphen G_1' und G_2' von G' existieren, so daß $G_1' + (\{x\}, \varnothing) + G_2'$ wiederum ein Teilgraph von G' ist. Es sei B_i' der zu G_i' ($\forall i \in \{1; 2\}$) erstellte Konstruktionsbaum. Dann ist entweder $\left(B_1' + x\right) + B_2'$ oder $B_1' + \left(x + B_2'\right)$ der zu $G_1' + (\{x\}, \varnothing) + G_2'$ gehörende Konstruktionsbaum. Man sieht nun, daß dann $B_1' + B_2'$ ein Konstruktionsbaum zu $G_1' + G_2'$ ist. Auf diese Weise kann man alle Knoten x ∈ K aus B' löschen und erhält somit B.

3.1.3 Schrittweise Modulvereinigung gemäß des Konstruktionsbaumes

3.1.3.1 Der Algorithmus „2M1“

Nachdem man zu dem serien - parallelen Graphen G einen Konstruktionsbaum B berechnet hat, kann man den in den Abbildungen 29 bis 31 notierten rekursiven Optimierungsalgorithmus „2M1“ zur Berechnung eines Schedules aller Module starten.

```
1)  es sei B der bereits berechnete Konstruktionsbaum zum übergebenen Graphen G
2)  starte verein_modul( B ) (siehe Zeile 5))
3)  starte zeiten( B, 1, N, 0 ) (siehe Zeile 27))
4)  Ende
```

Abbildung 29 - Algorithmus „2M1“.

```
5)  Unterroutine verein_modul( B ):
6)  es sei B = (W, F), d.h., B habe die Knotenmenge W und die Kantenmenge F
7)  wenn |W| = 1 ist, so tue
8)      es sei W = {z}
9)      für i = 1 bis N
10)         d(i) = T(z, i)
11) anderenfalls
12)     es sei z die Wurzel von B
13)     es sei x das linke und y das rechte Kind von z
14)     es sei B_x der Unterbaum mit der Wurzel x
15)     es sei B_y der Unterbaum mit der Wurzel y
16)     a = verein_modul( B_x )
17)     c = verein_modul( B_y )
18)     ist z mit + beschriftet, so tue
19)         für i = 1 bis N
20)             d(i) = a(i) + c(i)
21)     anderenfalls
22)         für i = 1 bis N
23)             d(i) = a(i) + c(i); typ(z, i) = 0
24)             für j = 1 bis i - 1
25)                 ist d(i) > max{ a(j); c(i - j) }, so d(i) = max{ a(j); c(i - j) };
                                                        typ(z, i) = j
26) Rücksprung unter Rückgabe der N-dimensionalen Variable d
```

Abbildung 30 - Algorithmus „2M1“ - Unterroutine „verein_modul“.

Zum besseren Verständnis des Algorithmus sollte man nur die erste Unterroutine „verein_modul“ ab Zeile 5) ohne die Operationen auf der Variable typ betrachten. Dann erkennt man, daß zu jedem Knoten des Konstruktionsbaumes ein Vektor d der Dimension N berechnet wird. Dabei bezeichnet N die Anzahl der Prozessoren des Parallelrechners. Für die Blätter des Baumes enthält dieser Vektor die Laufzeiten des dortigen Moduls mit den verschiedenen Prozessoranzahlen (siehe Zeile 10). Werden zwei Module A und C mittels + oder || verknüpft, so bildet der Algorithmus aus ihnen ein neues zusammengesetztes Modul D. Bei der seriellen Vereinigung ergibt sich D aus der Nacheinanderausführung von A und C mit gleicher Prozessoranzahl, d.h., der Vektor d, welcher die Laufzeiten des zusammengesetzten Moduls D enthält, entsteht durch Addition der beiden zu A und C gehörenden Vektoren a und c (siehe Zeile 20)). Sollte es sich hingegen um eine parallele Verknüpfung handeln, so ist es egal, ob die beiden Module hintereinander oder parallel auf verschiedenen Prozessorgruppen ausgeführt werden. Der Algorithmus entscheidet sich dann für die Möglichkeit, bei welcher die Laufzeit minimal ist:

```
27) Unterroutine zeiten( B, li, re, st ):
28) es sei B = (W, F), d.h., B habe die Knotenmenge W und die Kantenmenge F
29) wenn |W| = 1 ist, so tue
30)       es sei W = {z}
31)       s(z) = st, p(z) = li und b(z) = re - li + 1
32)       st = st + t(z, re - li + 1)
33) anderenfalls
34)       es sei z die Wurzel von B
35)       es sei x das linke und y das rechte Kind von z
36)       es sei B_x der Unterbaum mit der Wurzel x
37)       es sei B_y der Unterbaum mit der Wurzel y
38)       ist z mit + beschriftet oder typ(z, re - li + 1) = 0, so tue
39)             st = zeiten( B_x, li, re, st )
40)             st = zeiten( B_y, li, re, st )
41)       anderenfalls
42)             st_x = zeiten( B_x, li, li + typ(z, re - li + 1) - 1, st )
43)             st_y = zeiten( B_y, li + typ(z, re - li + 1), re, st )
44)             st = max{ st_x; st_y }
45) Rücksprung unter Rückgabe von st
```

Abbildung 31 - Algorithmus „2M1“ - Unterroutine „zeiten“.

$$\forall i \in [1; N]: d(i) = \min\left\{ a(i) + c(i); \min_{j=1}^{i-1}\{ \max\{ a(j); c(i-j) \} \} \right\}.$$

Das neu entstandene zusammengesetzte Modul D wird im weiteren Verlauf des Algorithmus wie ein normales Modul behandelt. Zur Unterscheidung zwischen den Modultypen wird nachfolgend die Bezeichnung zusammengesetzt für Module, die inneren Knoten des Konstruktionsbaumes entsprechen, benutzt. Man spricht von einem einfachen Modul, wenn es in der Menge der gegebenen Module enthalten ist.
Die Abbildung 32 stellt die Möglichkeiten zur Durchführung einer parallelen Vereinigung bei vier Prozessoren graphisch dar.

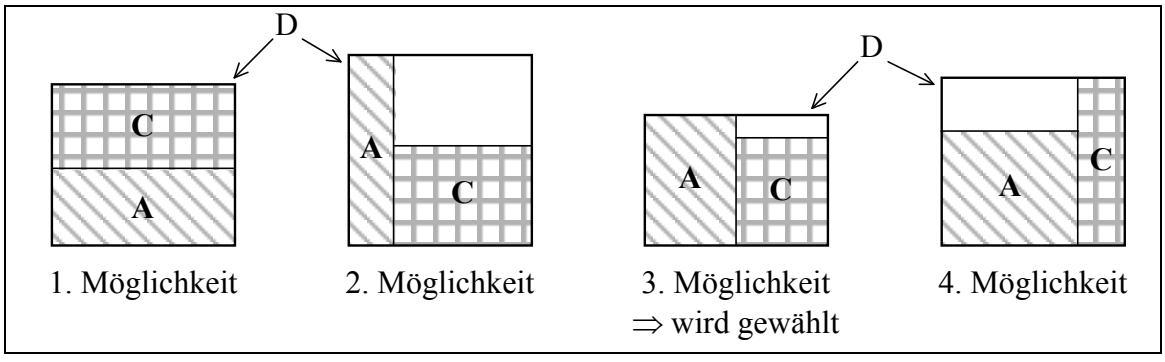


Abbildung 32 - Erstellen von Modul D = A ‖ C zur Ausführung auf vier Prozessoren.

Da aber nicht nur die Gesamtlaufzeit aller Module, sondern der gesamte Schedule gesucht wird, muß der Algorithmus noch weitere Informationen zwischenspeichern. Dies geschieht

bei allen Knoten, die eine parallele Vereinigung darstellen. Dort merkt sich das Verfahren in der Variable typ, welche der N Verknüpfungsmöglichkeiten gewählt wurde. Die Unterroutine „zeiten“ verwendet diese Daten, um die Prozessoren li bis re, auf denen ein Modul (egal ob einfach oder zusammengesetzt) ausgeführt wird, zu bestimmen. Zusammen mit der Anfangszeit st kann so der Schedule erstellt werden. Man beachte, daß die Variable st in den Zeilen 42) und 43) den gleichen Wert hat, jedoch in den Zeilen 39) und 40) verändert wird.

Da jede der beiden Unterroutinen mit jedem Knoten des Baumes einmal aufgerufen wird, entstehen je O(|W|) Sprünge zur Zeile 5) und Zeile 27). Ein Unterroutinendurchlauf benötigt, da die Unterbäume B_x und B_y durch das Umsetzen von Zeigern aus B erstellt werden können, höchstens $O(N^2)$ Zeiteinheiten. Insgesamt erhält man unter Anwendung von Lemma 19 als Laufzeitabschätzung des Algorithmus (ohne die Berechnung des Konstruktionsbaumes) $O(|W| \cdot N^2)$. Die Laufzeit von $O(N^2)$ je Unterroutinendurchlauf entsteht nur bei der parallelen Vereinigung zweier Module. Gestaltet man die in den Zeilen 22) bis 25) durchgeführte Minimumsuche wie im Anhang 9.15 dargestellt, so gelangt man auch bei der parallelen Vereinigung auf eine Laufzeit von O(N) je Unterroutinendurchlauf. Dann benötigt der Algorithmus „2M1“ ohne die Berechnung des Konstruktionsbaumes nur noch $O(|W| \cdot N)$ Zeiteinheiten. Die Berechnung des Konstruktionsbaumes benötigt (vgl. Abschnitt 3.1.2) O(|V| + |E|) Schritte. Insgesamt ist der Algorithmus „2M1“ also nach

$$O(|V| + |E| + |W| \cdot N) = O(|V| \cdot N + |E|) \qquad \text{(siehe Lemma 19)}$$
$$= O(M \cdot N + |E|)$$

Zeiteinheiten beendet. Dabei steht M für die Gesamtanzahl aller Module.

3.1.3.2 Garantierte Ergebnisgüte

Leider ist bei dem eben vorgestellten Algorithmus die garantierte Ergebnisgüte sehr gering.

Lemma 23: *Es bezeichne $H(A, i)$ die durch den Algorithmus „2M1“ ermittelte Laufzeit des (einfachen oder zusammengesetzten) Moduls A bei der Ausführung von A auf i Prozessoren. Im Gegensatz zu $T(A, i)$ ist $H(A, i)$ auch für zusammengesetzte Module A definiert. Mit $O(A, i)$ wird die Laufzeit des optimalen Schedules der in A enthaltenen einfachen Module notiert. Dabei steht $i \in [1; N]$ für die Anzahl der verwendeten Prozessoren. Außerdem sei noch der garantierte Gütefaktor*

$$G(A, i) = \frac{H(A, i)}{O(A, i)}$$

definiert.
(a) *Falls D ein einfaches Modul ist, kann man $G(D, i) = 1$ setzen.*
(b) *Ist $D = A + C$, so gilt $G(D, i) \leqslant \max\{ G(A, i), G(C, i) \}$.*
(c) *Sollte D aus $A \parallel C$ entstehen, so folgt für den garantierten Gütefaktor*

$$G(D, i) \leqslant \begin{cases} \tfrac{1}{2} \cdot (i + G(A, i)) : \text{falls } H(A, 1) \geqslant H(C, 1) \\ \tfrac{1}{2} \cdot (i + G(C, i)) : \text{falls } H(A, 1) < H(C, 1) \end{cases}.$$

Wenn keine Aussage über die Größe von $H(A, 1)$ oder $H(C, 1)$ möglich ist, gilt

$$G(D, i) \leqslant \tfrac{1}{2} \cdot (i + \max\{ G(A, i); G(C, i) \}).$$

Beweis:
Fall (a): D sei ein einfaches Modul.
Dieser Fall ist trivial, da ein einfaches Modul nur in einer Weise optimal auf i Prozessoren ausgeführt werden kann. Diese findet der Algorithmus „2M1“ garantiert.

Fall (b): Es sei D = A + C.
Nach Definition gelten

$$H(A, i) = G(A, i) \cdot O(A, i) \quad \text{und} \quad H(C, i) = G(C, i) \cdot O(C, i). \tag{1}$$

Aus Lemma 15 folgt, daß auch im optimalen Schedule kein einfaches Modul aus C vor dem Ende aller einfachen Module aus A gestartet werden kann. Es gibt demzufolge einen Zeitpunkt, bis zu dem alle einfachen Module von A beendet sind und noch kein einfaches Modul aus C begonnen wurde. Deshalb ist

$$O(D, i) = O(A, i) + O(C, i).$$

Setzt man in diese Beziehung die Gleichungen (1) ein, so erhält man

$$O(D, i) = \frac{H(A, i)}{G(A, i)} + \frac{H(C, i)}{G(C, i)} \geqslant \frac{H(A, i) + H(C, i)}{\max\{ G(A, i); G(C, i) \}}.$$

Da der Algorithmus den neuen Laufzeitvektor mittels

$$H(D, i) = H(A, i) + H(C, i)$$

berechnet, folgt

$$\max\{ G(A, i); G(C, i) \} \cdot O(D, i) \geqslant H(D, i), \quad \text{also} \quad G(D, i) \leqslant \max\{ G(A, i); G(C, i) \}.$$

Fall (c): Es sei D = A ‖ C.
Fall (c).1: Es gelte i = 1.
In diesem Fall werden A und C nacheinander auf dem einen Prozessor ausgeführt. Da A und C ebenfalls mit „2M1“ erstellt wurden, werden auch die Module, aus denen A bzw. C entstanden ist, nacheinander ausgeführt. Setzt man diese Überlegung rekursiv fort, so erkennt man, daß alle in A, C und D enthaltenen einfachen Module nacheinander ausgeführt werden. Man sieht auch, daß dies der optimale Schedule ist. Demzufolge gelten:

$$O(A, 1) = H(A, 1),\ O(C, 1) = H(C, 1),\ O(D, 1) = H(D, 1) \text{ und } G(A, 1) = G(C, 1) = G(D, 1) = 1.$$

Diese Gütefaktoren erfüllen die im Lemma Punkt (c) angegebenen Formeln.
Fall (c).2: Es gelte i ⩾ 2.
Da die Operation ‖ kommutativ ist, kann man ohne Beschränkung der Allgemeinheit

$$H(A, 1) \geqslant H(C, 1)$$

voraussetzen. Aus der Flächenbedingung folgt

$$O(D, i) \geqslant \frac{O(D, 1)}{i}.$$

Wie man eben gesehen hat, gilt

$$O(D, 1) = H(D, 1).$$

Weiterhin ergibt sich

$$H(D, 1) = H(A, 1) + H(C, 1).$$

Setzt man die letzten drei Beziehungen zu einer zusammen, so erhält man

$$O(D, i) \geqslant \frac{H(A, 1) + H(C, 1)}{i}. \tag{2}$$

Auf der anderen Seite wird die Gesamtlaufzeit des optimalen Schedules bei der Hinzunahme von Modulen nicht kleiner, also

$$O(D, i) \geqslant O(A, i).$$

Mit Hilfe von (1) kann man daraus

$$O(D, i) \geqslant \frac{H(A, i)}{G(A, i)} \tag{3}$$

ableiten. Setzt man (2) und (3) zusammen, so erhält man:

$$O(D, i) \geqslant \max\left\{ \frac{H(A, 1) + H(C, 1)}{i}; \frac{H(A, i)}{G(A, i)} \right\}.$$

Aus der Arbeitsweise des Algorithmus folgt

$$H(D, i) \leqslant \min\{ H(A, i) + H(C, i); \max\{ H(A, 1); H(C, i - 1) \} \}$$

und daraus mittels $H(A, 1) \geqslant H(C, 1) \geqslant H(C, i - 1)$

$$H(D, i) \leqslant \min\{ H(A, i) + H(C, i); H(A, 1) \}. \tag{4}$$

Fall (c).2.1: Es sei $i \cdot H(A, i) \leqslant G(A, i) \cdot (H(A, 1) + H(C, 1))$. (5)
Fall (c).2.1.1: Es gelte $H(A, 1) \leqslant H(A, i) + H(C, i)$. (6)
Wegen $H(C, i) \leqslant H(C, 1)$ folgt aus (6)

$$H(A, 1) \leqslant H(A, i) + H(C, 1).$$

Durch Erweitern erhält man

$$i \cdot H(A, 1) \leqslant i \cdot H(A, i) + i \cdot H(C, 1).$$

Setzt man nun (5) ein, so ergibt sich

$$i \cdot H(A, 1) \leqslant G(A, i) \cdot (H(A, 1) + H(C, 1)) + i \cdot H(C, 1)$$

$$\Leftrightarrow \quad 2 \cdot i \cdot H(A, 1) \leqslant i \cdot H(A, 1) + G(A, i) \cdot (H(A, 1) + H(C, 1)) + i \cdot H(C, 1)$$

$$\Leftrightarrow \quad H(A, 1) \leqslant \frac{i + G(A, i)}{2} \cdot \frac{H(A, 1) + H(C, 1)}{i}.$$

Mit (2) und (4) erhält man

$$H(D, i) \leqslant \tfrac{1}{2} \cdot (i + G(A, i)) \cdot O(D, i) \quad \text{bzw.} \quad G(D, i) \leqslant \tfrac{1}{2} \cdot (i + G(A, i)).$$

Fall (c).2.1.2: Es sei $H(A, 1) > H(A, i) + H(C, i)$. (7)
Aus $H(C, i) \leqslant H(C, 1)$ folgt

$$i \cdot H(C, i) \leqslant i \cdot H(C, 1) \quad \text{sowie} \quad i \cdot H(A, i) + i \cdot H(C, i) \leqslant i \cdot H(C, 1) + i \cdot H(A, i).$$

Setzt man nun (5) ein, so erhält man

$$i \cdot H(A, i) + i \cdot H(C, i) \leqslant i \cdot H(C, 1) + G(A, i) \cdot (H(A, 1) + H(C, 1))$$

und somit

$$i \cdot H(A, 1) + i \cdot H(A, i) + i \cdot H(C, i) \leqslant i \cdot H(A, 1) + i \cdot H(C, 1) + G(A, i) \cdot (H(A, 1) + H(C, 1)).$$

Mit Hilfe von (7) kann man auf

$$i \cdot (H(A, i) + H(C, i)) + i \cdot H(A, i) + i \cdot H(C, i) < (i + G(A, i)) \cdot (H(A, 1) + H(C, 1))$$

$$\Leftrightarrow \quad H(A, i) + H(C, i) < \frac{i + G(A, i)}{2} \cdot \frac{H(A, 1) + H(C, 1)}{i}$$

schließen. Aus dieser Beziehung ist dann mittels (2) und (4)

$$H(D, i) \leqslant \tfrac{1}{2} \cdot (i + G(A, i)) \cdot O(D, i)$$

und somit

$$G(D, i) \leqslant \tfrac{1}{2} \cdot (i + G(A, i))$$

ableitbar.

Fall (c).2.2: Es gelte $i \cdot H(A, i) > G(A, i) \cdot (H(A, 1) + H(C, 1))$. (8)

Fall (c).2.2.1: Es sei $H(A, 1) \leqslant H(A, i) + H(C, i)$. (9)

Setzt man in (9) die Beziehung

$$H(C, i) \leqslant H(C, 1)$$

ein, so erhält man

$$H(A, 1) \leqslant H(C, 1) + H(A, i)$$

$$\Leftrightarrow \quad 2 \cdot H(A, 1) \leqslant H(A, 1) + H(C, 1) + H(A, i)$$

$$\Leftrightarrow \quad 2 \cdot G(A, i) \cdot H(A, 1) \leqslant G(A, i) \cdot (H(A, 1) + H(C, 1)) + G(A, i) \cdot H(A, i).$$

Unter Verwendung der Ungleichung (8) kann man auf

$$2 \cdot G(A, i) \cdot H(A, 1) < i \cdot H(A, i) + G(A, i) \cdot H(A, i)$$

$$\Leftrightarrow \quad H(A, 1) < \frac{i + G(A, i)}{2} \cdot \frac{H(A, i)}{G(A, i)}$$

schließen. Mit Hilfe von (3) und (4) ergibt sich

$$H(D, i) < \tfrac{1}{2} \cdot (i + G(A, i)) \cdot O(D, i)$$

und demzufolge auch

$$G(D, i) < \tfrac{1}{2} \cdot (i + G(A, i)).$$

Fall (c).2.2.2: Es gelte $H(A, 1) > H(A, i) + H(C, i)$. (10)

Wendet man auf (8) die Beziehung $H(C, i) \leqslant H(C, 1)$ an, so kommt man auf

$$G(A, i) \cdot H(A, 1) + G(A, i) \cdot H(C, i) < i \cdot H(A, i).$$

Setzt man nun gleich noch (10) ein, so ergibt sich

$$G(A, i) \cdot (H(A, i) + H(C, i)) + G(A, i) \cdot H(C, i) < i \cdot H(A, i)$$

$$\Leftrightarrow \quad G(A, i) \cdot H(A, i) + 2 \cdot G(A, i) \cdot H(C, i) < i \cdot H(A, i)$$

$$\Leftrightarrow \quad 2 \cdot G(A, i) \cdot H(A, i) + 2 \cdot G(A, i) \cdot H(C, i) < i \cdot H(A, i) + G(A, i) \cdot H(A, i)$$

$$\Leftrightarrow \quad H(A, i) + H(C, i) < \frac{i + G(A, i)}{2} \cdot \frac{H(A, i)}{G(A, i)}.$$

Mit den oben aufgeführten Ungleichungen (3) und (4) kann man auch hier auf

$$H(D, i) < \tfrac{1}{2} \cdot (i + G(A, i)) \cdot O(D, i) \quad \text{und} \quad G(D, i) < \tfrac{1}{2} \cdot (i + G(A, i))$$

schließen.

■

Im Anhang 9.16 werden Beispiele konstruiert, bei denen der Algorithmus „2M1“ tatsächlich nur die im Lemma 23 angegebenen Gütefaktoren erreicht.

Unter Verwendung der zuvor bewiesenen Lemmata kann man die folgenden, wenn auch sehr schwachen Abschätzungen angeben:

Lemma 24: *Es sei M die Anzahl aller zu optimierenden einfachen Module. N bezeichne die insgesamt verfügbare Gesamtprozessoranzahl. Dann gelten für den vom Algorithmus „2M1“ erzielten Gütefaktor G die folgenden zwei Aussagen:*

(a) $G \leqslant \frac{(2^{M-1} - 1) \cdot N + 1}{2^{M-1}}$.

(b) *Zu jedem $\varepsilon > 0$ kann man ein Beispiel mit einem Gütefaktor größer oder gleich $N - \varepsilon$ konstruieren. Dazu muß nur ein hinreichend großes M gewählt werden.*

Beweis von Punkt **(a)**:
Zum Beweis der oberen Schranke wird zuerst die serielle Vereinigung von zwei zusammengesetzten Modulen A und C zu einem neuen Modul D betrachtet.
Mit Hilfe der Flächenbedingung $O(A, N) \geqslant \frac{O(A, 1)}{N}$ folgt

$$G(A, N) = \frac{1}{2} \cdot \left(\frac{H(A, N)}{O(A, N)} + G(A, N) \right) \leqslant \frac{1}{2} \cdot \left(N \cdot \frac{H(A, N)}{O(A, 1)} + G(A, N) \right) \leqslant \frac{1}{2} \cdot \left(N \cdot \frac{H(A, 1)}{O(A, 1)} + G(A, N) \right).$$

Unter Verwendung von Lemma 23(a) gelangt man zu

$$G(A, N) \leqslant ½ \cdot (N + G(A, N)).$$

Diese Ungleichung kann man äquivalent auch für $G(C, N)$ aufstellen.
Setzt man die beiden Ungleichungen in die Aussage von Lemma 23(b) ein, so erhält man

$$G(D, N) \leqslant \max\{ ½ \cdot (N + G(A, N)); ½ \cdot (N + G(C, N)) \} = ½ \cdot (N + \max\{ G(A, N); G(C, N) \}).$$

Für die parallele Vereinigung von A und C zu D (Lemma 23(c)) gilt ebenfalls

$$G(D, N) \leqslant ½ \cdot (N + \max\{ G(A, N); G(C, N) \}). \quad (1)$$

Demzufolge gilt die Ungleichung (1) für die parallele und für die serielle Vereinigung von Modul A und Modul C zu Modul D.

Nun wird eine Induktion über die Tiefe des Konstruktionsbaumes durchgeführt. Als Induktionsvoraussetzung gilt für die resultierende Ergebnisgüte $G_{1 \circ 2}$ beim Vorliegen eines Konstruktionsbaumes mit Tiefe x ($\forall x \leqslant y$) die folgende Ungleichung:

$$G_{1 \circ 2} \leqslant \frac{(2^{x-1} - 1) \cdot N + 1}{2^{x-1}}.$$

Sollte y = 1 sein (Induktionsanfang), so steht zur Optimierung nur ein Modul zur Verfügung. Es gilt nach Lemma 23(a): $G_{1 \circ 2} = 1$. Dieses Ergebnis erhält man auch unter Verwendung der in der Induktionsvoraussetzung angegebenen Formel.
Hat der Konstruktionsbaum nun eine Tiefe von y + 1 mit $y \geqslant 1$ (Induktionsschritt), so besitzt die Wurzel zwei durch ihre Kinder erzeugte Unterbäume. Einer von ihnen hat mindestens die Tiefe y. Sein Gütefaktor G_1 erfüllt nach der Induktionsvoraussetzung

$$G_1 \leqslant \frac{(2^{y-1} - 1) \cdot N + 1}{2^{y-1}}. \quad (2)$$

Für den anderen Unterbaum gilt bei einer Tiefe von $x \leqslant y$ ebenfalls

$$G_2 \leqslant \frac{(2^{x-1} - 1) \cdot N + 1}{2^{x-1}} = N - \frac{N - 1}{2^{x-1}} \leqslant N - \frac{N - 1}{2^{y-1}} = \frac{(2^{y-1} - 1) \cdot N + 1}{2^{y-1}}. \quad (3)$$

Mit Hilfe von (1), (2) und (3) erhält man

$$G_{1 \circ 2} \leqslant ½ \cdot (N + \max\{ G_1; G_2 \}) \leqslant \frac{1}{2} \cdot \left(N + \frac{(2^{y-1} - 1) \cdot N + 1}{2^{y-1}} \right) = \frac{1}{2} \cdot \frac{(2^{y-1} + 2^{y-1} - 1) \cdot N + 1}{2^{y-1}}$$

$$= \frac{(2^{(y+1)-1} - 1) \cdot N + 1}{2^{(y+1)-1}}.$$

Da der Konstruktionsbaum genau M Blätter hat, ist seine maximale Tiefe auf M beschränkt. Für den garantierten Gütefaktor heißt das, daß er den Wert $\frac{(2^{M-1} - 1) \cdot N + 1}{2^{M-1}}$ nicht übersteigen kann.

Beweis von Punkt **(b)**:
Die unter Punkt (b) angegebene Aussage ist ein Teilergebnis der im Anhang 9.16 durchgeführten Konstruktion zur Erstellung des Moduls A bei der parallelen Vereinigung D = A ‖ C.
■

3.1.3.3 Weitere Eigenschaften des Schedules

Nachdem bisher die obere Schranke der Ergebnisgüte des Algorithmus „2M1“ unabhängig von den Modullaufzeiten berechnet wurde, nutzt man in diesem Abschnitt noch Informationen über die Parallelisierbarkeit der einzelnen Module. Dadurch kann man in vielen Fällen zu einer besseren Abschätzung des Gütefaktors gelangen.

Lemma 25: *Der mittels „2M1“ erzielte Schedule mit der Gesamtlaufzeit H ist nie schlechter als der eines trivialen Algorithmus, welcher alle M Module nacheinander auf allen N Prozessoren ausführt:*

$$H \leqslant \sum_{x=1}^{M} T(x, N).$$

Dabei stehe $T(x, N)$ für die Laufzeit von Modul x bei der Ausführung auf allen N Prozessoren.
Dieser triviale Algorithmus wird als Gang - Scheduling (vgl. [48] und [49]) bezeichnet.

Zum **Beweis** dieser Aussage ist es ausreichend, wenn man sich überlegt, ob bei der von „2M1“ durchgeführten Optimierung auch der angesprochene Spezialfall behandelt wird. Löscht man im Algorithmus „2M1“ in der Abbildung 30 die Zeilen 24) und 25), so entsteht der triviale Algorithmus. Man sieht, daß der triviale Algorithmus bei der parallelen Vereinigung der Module A und C zu einem neuen Modul D die Formel

$$\forall i \in [1; N] : H(D, i) = H(A, i) + H(C, i)$$

verwendet. Da

$$H(A, i) + H(C, i) \geqslant \min\left\{ H(A, i) + H(C, i); \min_{j=1}^{i-1}\{ \max\{ H(A, j); H(C, i - j) \} \} \right\}$$

ist, sind die vom trivialen Algorithmus berechneten Laufzeiten für das Modul D nie kleiner als die von „2M1“ ermittelten. Auch wirkt sich bei „2M1“ ein Teilschedule mit kleineren Laufzeiten nicht verschlechternd auf den Schedule, welcher alle M Module umfaßt, aus.
■

Die nächste Eigenschaft stellt einen Zusammenhang zwischen der Parallelisierbarkeit der einfachen Module und der Ergebnisgüte von „2M1“ dar.

Lemma 26: *Es wird der schlechteste Parallelisierbarkeitsgrad aller M einfachen Module x durch*

$$a = \max_{x=1}^{M} \left\{ \frac{T(x, N)}{T(x, 1)} \right\}$$

definiert. Dabei bezeichne wie bisher $T(x, i)$ die Laufzeit des Moduls x bei der Ausführung auf i Prozessoren. Dann wird bei der Optimierung der einfachen

Module x mittels „2M1“ garantiert der Gütefaktor $G \leqslant N \cdot a$ erreicht. Dabei stehe N für die Anzahl aller Prozessoren.

Beweis: Aus der Wahl von a läßt sich für alle einfachen Module $x \in [1; M]$

$$a \geqslant \frac{T(x, N)}{T(x, 1)} \quad \text{bzw.} \quad T(x, 1) \geqslant \frac{T(x, N)}{a} \tag{1}$$

ableiten. Mit Hilfe der Flächenbedingung folgt, daß für die Gesamtlaufzeit O des optimalen Schedules

$$O \geqslant \frac{\sum_{x=1}^{M} T(x, 1)}{N} \tag{2}$$

gilt. Verknüpft man die Ungleichungen (1), (2) und die aus Lemma 25, so erhält man:

$$O \geqslant \frac{\sum_{x=1}^{M} T(x, 1)}{N} \geqslant \frac{\sum_{x=1}^{M} T(x, N)}{N \cdot a} \geqslant \frac{H}{N \cdot a}.$$

■

Einige Überlegungen zur Berechnung einer durchschnittlichen Ergebnisgüte für „2M1“ findet man im Anhang 9.17. Im Anhang 9.18 werden kleine Modifikationen des Algorithmus „2M1“ diskutiert.

3.1.4 Praktische Tests

Das Laufzeitverhalten und die Ergebnisgüte des Algorithmus „2M1“ wurde zusätzlich zu den theoretischen Ergebnissen mit Hilfe von 100 zufällig erzeugten Beispielen untersucht (siehe Beispiel 3 im Anhang 9.19). Das Beispiel 3 nutzt die Modullaufzeiten von Beispiel 1a. Die Erstellung eines serien - parallelen Modulabhängigkeitsgraphen basiert auf der zufälligen parallelen oder seriellen Vereinigung der erzeugten Module.
Zur Berechnung einer unteren Schranke für die Gesamtlaufzeit O des optimalen Schedules benutzt der Algorithmus wiederum die Flächenbedingung

$$O \geqslant \frac{1}{N} \cdot \sum_{x=1}^{M} T(x, 1). \tag{1}$$

Anstatt wie im Abschnitt 2.1.7 nur die Laufzeit des Moduls mit der größten Laufzeit bzgl. der voll parallelen Ausführung zu betrachten, kann man nun die maximale Summe der voll parallelen Modullaufzeiten aller auf einem Pfad im Modulabhängigkeitsgraphen G liegenden Module nutzen:

$$O \geqslant \max\left\{ \sum_{i=1}^{k} T\left(x_i, N\right) \;:\; x_1, ..., x_k \text{ ist ein Pfad in } G \right\}. \tag{2}$$

Da bei der parallelen Vereinigung mehrerer serien - paralleler Graphen die Anwendung des Assoziativgesetzes nur zu verschiedenen Konstruktionsbäumen, aber zum gleichen serien - parallelen Graphen führt (siehe Lemma 18), kann man bei der Berechnung eines Konstruktionsbaumes die Reihenfolge aufeinanderfolgender paralleler Vereinigungen frei wählen. Der implementierte Algorithmus erstellt entweder eine lineare Kette oder einen balancierten Binärbaum als Konstruktionsbaum für mehrere aufeinanderfolgende parallele Vereinigungen (siehe Anhang 9.19). In den Abbildungen zur Ergebnisgüte (Abbildung 33) und Laufzeit (Abbildung 34) entsprechen die linken drei Säulen der kettenförmigen und die rechten drei Säulen der balancierten Vereinigungsstruktur. Die linke (rechte) Säule jeder der beiden Dreiergruppen entstand bei der Übergabe eines serien - parallelen Graphen mit viermal so vielen

parallelen (seriellen) Vereinigungen wie seriellen (parallelen) Vereinigungen. In der mittleren Säule jeder Dreiergruppe ist das Verhältnis von paralleler zu serieller Vereinigung im Mittel Eins.

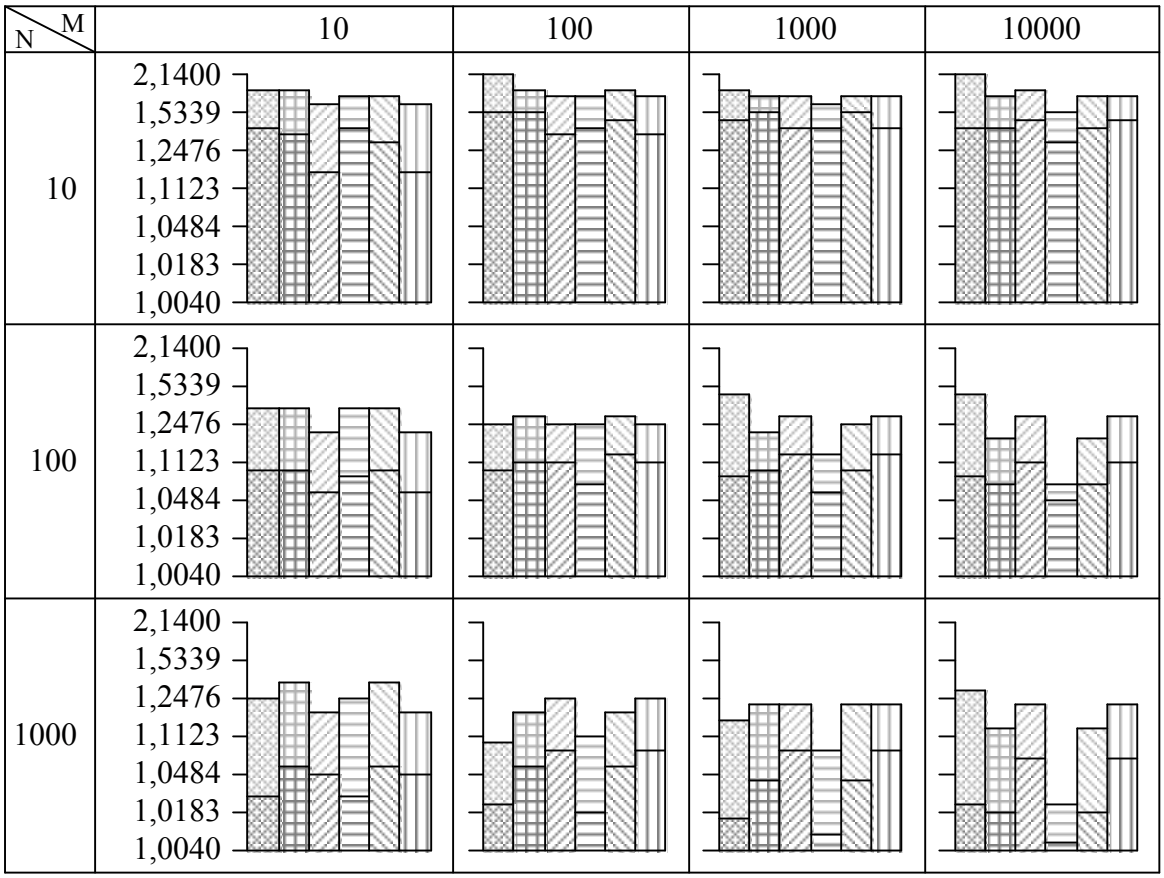


Abbildung 33 - schlechtester (oberste Markierung jeder Säule) und durchschnittlicher Gütefaktor (mittlere Markierung) beim Beispiel 3.

Die Laufzeit des Algorithmus „2M1“ liegt sogar bei großen Beispielen mit zehntausend Modulen und eintausend Prozessoren noch weit unter einer Sekunde und ist demzufolge sehr gering. In der Abbildung 34 sieht man sehr schön, daß die Laufzeit bei konstantem N (M) nur linear mit M (N) wächst. Man erkennt außerdem einen geringen Einfluß des Verhältnisses zwischen paralleler und serieller Vereinigung im übergebenen Graphen auf die Laufzeit des Algorithmus. Die parallele Vereinigung ist durch den doch etwas höheren Rechenaufwand minimal langsamer als die serielle Vereinigung zweier (fiktiver) Module. Weiterhin liegt nur ein sehr geringer Mehraufwand für die Erzeugung eines balancierten Modulabhängigkeitsgraphen vor.
Die Ergebnisgüte des Algorithmus (siehe Abbildung 33) ist von der Modulanzahl relativ unabhängig. Der Algorithmus kann also sehr gut die Task- und die Datenparallelität der Eingabe ausnutzen. Mit steigender Gesamtprozessoranzahl wird die Qualität des von „2M1“ berechneten Schedules immer besser.
Der Einfluß der Gesamtprozessoranzahl soll an einem Beispiel dargelegt werden: Man betrachte die parallele Vereinigung der beiden in der nachfolgenden Tabelle angegebenen Module.

$N_1 = 4$					$N_2 = 8$								
i	1	2	3	4	i	1	2	3	4	5	6	7	8
$T_1(A, i)$	18	16	14	12	$T_2(A, i)$	19	18	17	16	15	14	13	12
$T_1(C, i)$	20	18	16	14	$T_2(C, i)$	21	20	19	18	17	16	15	14

Die Laufzeiten der Module A und C wurden unabhängig von der Gesamtprozessoranzahl gewählt, d.h., für zwei verschiedene Gesamtprozessoranzahlen N_1 und N_2 folgen aus $\frac{i}{N_1} = \frac{j}{N_2}$ die Beziehungen $T_1(A, i) = T_2(A, j)$ und $T_1(C, i) = T_2(C, j)$. Dabei bezeichnet T_i die Laufzeiten der Module bei der Gesamtprozessoranzahl N_i. Diese Voraussetzung garantiert, daß der Mehraufwand bei der Parallelisierung der Module unabhängig von der Gesamtprozessoranzahl N_i ist. Dadurch wird ein Vergleich der Vereinigungsergebnisse ermöglicht.
Das zusammengesetzte Modul A ‖ C hat bei $N_1 = 4$ Prozessoren die voll parallele Laufzeit 18. Im Fall $N_2 = 8$ beträgt sie nur 17.
Man kann sich verdeutlichen, daß aus $N_2 = k \cdot N_1$ mit $k \in \mathbb{N}$ und Modulen A und C mit Modullaufzeiten unabhängig von der Gesamtprozessoranzahl immer die Beziehung

$$H_1(A \parallel C, N_1) \geqslant H_2(A \parallel C, N_2)$$

folgt. Dabei steht $H_i(A \parallel C, N_i)$ für die Laufzeit des zusammengesetzten Moduls A ‖ C bei der Ausführung auf allen N_i Prozessoren.

Betrachtet man die durchschnittliche Ergebnisgüte in Abhängigkeit von der Wahrscheinlichkeit v, daß eine Vereinigung innerhalb des Modulabhängigkeitsgraphen eine serielle Vereinigung ist, so kann man die folgenden Eigenschaften erkennen: Liegen nur wenige Module vor, so existieren im Fall v = 0,8 fast ausschließlich serielle Vereinigungen. Da bei seriellen Vereinigungen auch im optimalen Schedule nur die Nacheinanderausführung der Module möglich ist, erzielt „2M1“ sehr gute Schedules (siehe Abbildung 33). Außerdem kann die Gesamtlaufzeit O des optimalen Schedules mit Hilfe der Abschätzung (2) relativ exakt ermittelt werden. Liegt bei einer großen Modulanzahl die Verteilungswahrscheinlichkeit v = 0,8 vor, so liefert die Abschätzung von O mittels (2) aufgrund der größeren Anzahl an parallelen Vereinigungen nur noch eine grobe Schranke. Da in diesem Fall im Schedule viele Module mit einer hohen Prozessoranzahl ausgeführt werden, ist wegen dem Mehraufwand durch die Parallelisierung der einfachen Module auch die mittels (1) berechnete Schranke nicht sehr genau. Dies äußert sich dann in schlechteren Gütefaktoren in der Abbildung 33.
Für den Fall v = 0,2 wurde bereits dargelegt, daß sich die Qualität des von „2M1“ ermittelten Schedules mit steigender Gesamtprozessoranzahl N erhöht. Gilt außerdem $N \ll M$, so werden sicherlich im optimalen Schedule viele einfache Module nur mit einem Prozessor ausgeführt. Demzufolge liefert die Abschätzung von O mittels (1) einen relativ exakten Wert, was wiederum zu kleinen Gütefaktoren in Abbildung 33 führt.
Aus diesen Aussagen folgt, daß der näherungsweise ermittelte Gütefaktor nur für kleine Produkte $M \cdot N$ mit steigendem v kleiner wird. Für große Produkte $M \cdot N$ erhält man bei großem v auch den größeren Gütefaktor.
Bei einem hohen Anteil an parallelen Vereinigungen erkennt man einen Unterschied zwischen der Verwendung des kettenförmigen oder balancierten Vereinigungsbaums. Im Durchschnitt erstellt „2M1“ bei der Verwendung eines balancierten Vereinigungsbaums den besseren Schedule. Wenn man davon ausgeht, daß die Module alle ähnliche Laufzeiten aufweisen, so er-

kennt man, daß bei der Verwendung des balancierten Vereinigungsbaums bei jeder parallelen Vereinigung, also auch bei der Vereinigung von zusammengesetzten Modulen, immer Module mit ungefähr gleichen Laufzeiten aufeinander treffen. Dadurch wird sicherlich sehr oft eine gleichzeitige Ausführung der beiden Module als Optimum gewählt. Bei der Verwendung eines linearen Vereinigungsbaums trifft hingegen ein bei jeder Vereinigung laufzeitintensiver werdendes Modul auf ein Modul mit kurzer Laufzeit. Dadurch ergibt sich irgendwann als Optimum nur noch die sequentielle Ausführung beider Module. Diese Modulanordnung kann dann die Datenparallelität nur noch bedingt ausnutzen und es kommt zu einem schlechteren Schedule.

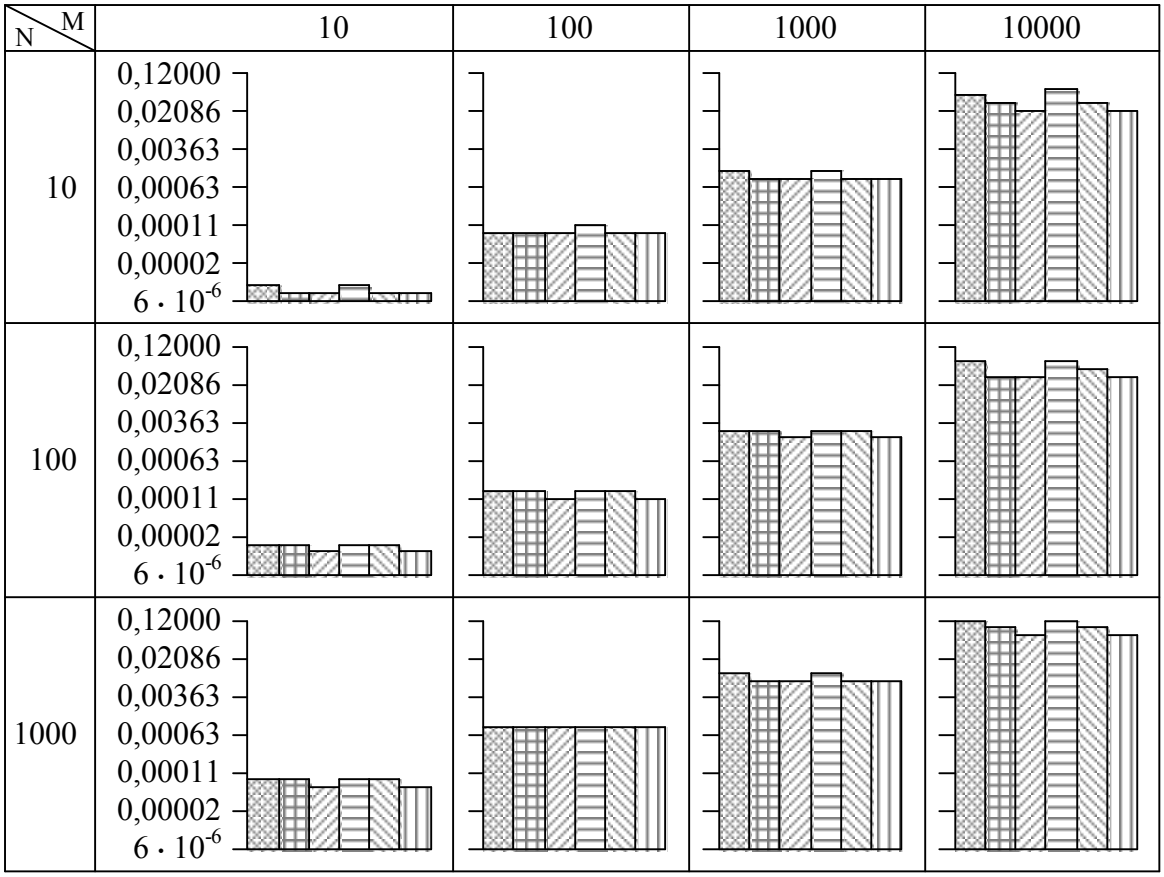


Abbildung 34 - durchschnittliche Laufzeit in Sekunden (Beispiel 3).

In der Abbildung 33 sieht man auch, daß die Differenz zwischen schlechtestem und durchschnittlichem Gütefaktor relativ groß ist. Da für „2M1“ keine konstante Schranke für den Gütefaktor bewiesen werden kann, entstanden natürlich auch ein paar Testbeispiele (im Durchschnitt 4%), bei denen der erreichte Gütefaktor etwas stärker vom ermittelten durchschnittlichen Gütefaktor abwich.

3.1.5 Vergleich der erzielten Ergebnisse mit denen anderer Algorithmen

Einen anderen Algorithmus zur Optimierung von formbaren Modulen mit einem serien - parallelen Modulabhängigkeitsgraphen findet man in [2]. Dort wird vorausgesetzt, daß alle Module die Flächenklausel erfüllen und ihre Laufzeiten ganzzahlig sind. Er garantiert dann einen Gütefaktor von $\frac{3+\sqrt{5}}{2} \approx 2{,}62$. Leider liegt seine Laufzeit bei der Optimierung von M Modulen auf N Prozessoren nicht unter $O(M \cdot N \cdot H)$, wobei H wie üblich für die Gesamtlaufzeit des er-

zielten Schedules steht. Dadurch ist seine praktische Nutzbarkeit natürlich stark eingeschränkt.
Eine Teilmenge der serien - parallelen Graphen, nämlich die Bäume, betrachtet der in [12] angegebene Algorithmus. Auch er setzt voraus, daß alle Module die Flächenklausel erfüllen. Seine obere Schranke für die Ergebnisgüte liegt bei $4 \cdot (1 + \varepsilon)$, für ein festes $\varepsilon > 0$. Durch die Verwendung der dynamischen Programmierung hat er eine mit $O\left(\log\left(\frac{N}{\varepsilon}\right) \cdot \frac{N^2 \cdot M}{\varepsilon}\right)$ für kleine ε viel zu große Laufzeit.
In [14] wird der Spezialfall, daß der Modulabhängigkeitsgraph ein Baum ist, nur zwei Prozessoren vorhanden sind ($N = 2$) und alle Module unabhängig von der Prozessoranzahl die Laufzeit Eins haben ($\forall x, i: T(x, i) = 1$) behandelt. Der vorgestellte Algorithmus läuft in $O(M)$ und ermittelt das optimale Ergebnis.
Ein ähnlicher Spezialfall wird in [15] vorgestellt. Dort wird angenommen, daß alle Module unabhängig von der Prozessoranzahl die Laufzeit Eins haben und der Modulabhängigkeitsgraph nur Knoten mit einem Ein- und Ausgangsgrad von maximal Eins aufweist. Der Modulabhängigkeitsgraph entartet also zu einer Menge von Ketten. Unter diesen Annahmen kann der optimale Schedule in $O(M)$ berechnet werden.
Aufgrund der hohen Laufzeit der in [2] und [12] angegebenen Algorithmen, sollten diese Algorithmen nur genutzt werden, wenn schnelle Algorithmen ohne garantierte Ergebnisgüte, wie z.B. „2M1“, keinen zufriedenstellenden Schedule finden.

3.2 Optimierung von nicht formbaren Modulen

3.2.1 Der Algorithmus „2M1N“

Der im Abschnitt 3.1.3.1 vorgestellte Algorithmus „2M1“ kann direkt zur Optimierung von nicht formbaren Modulen verwendet werden. Zur Kennzeichnung, daß der Algorithmus „2M1“ mit nicht formbaren Modulen ausgeführt wird, wird der Algorithmus für nicht formbare Module mit „2M1N“ bezeichnet.
Im Beweis zu Lemma 25 wird nicht vorausgesetzt, daß die Module formbar sind. Somit ist das Lemma 25 auch für nicht formbare Module gültig. Das Lemma 26 kann zu einem neuen Lemma abgeändert werden:

Lemma 27: *Wie bisher sei N die Anzahl aller Prozessoren, M die Anzahl aller Module und $T(x, i)$ die Laufzeit des Moduls x bei der Ausführung auf i Prozessoren. Außerdem sei*

$$a = \max_{x=1}^{M} \left\{ \frac{T(x, N)}{\min_{i=1}^{N} \{i \cdot T(x, i)\}} \right\}$$

der schlechteste Parallelisierungsfaktor. Dann gilt für den von „2M1“ erzielten Gütefaktor: $G \leqslant a \cdot N$.

Beweis: Durch Erweitern der im Lemma 25 aufgestellten Ungleichung gelangt man zu

$$H \leqslant \left(\sum_{x=1}^{M} \min_{i=1}^{N} \{i \cdot T(x, i)\} \cdot \frac{T(x, N)}{\min_{i=1}^{N} \{i \cdot T(x, i)\}} \right).$$

Schätzt man den Bruch nach oben ab, so erhält man

$$H \leqslant \left(\sum_{x=1}^{M} \min_{i=1}^{N} \{ i \cdot T(x, i) \} \right) \cdot \max_{x=1}^{M} \left\{ \frac{T(x, N)}{\min_{i=1}^{N} \{ i \cdot T(x, i) \}} \right\}$$

$$\leqslant \left(\sum_{x=1}^{M} \min_{i=1}^{N} \{ i \cdot T(x, i) \} \right) \cdot a. \qquad (1)$$

Nutzt man die Flächenbedingung zur Bestimmung einer unteren Schranke für die Gesamtlaufzeit O des optimalen Schedules, so ergibt sich

$$O \geqslant \frac{1}{N} \cdot \sum_{x=1}^{M} \min_{i=1}^{N} \{ i \cdot T(x, i) \}. \qquad (2)$$

Aus (1) und (2) resultiert

$$G = \frac{H}{O} \leqslant N \cdot a.$$

■

3.2.2 Praktische Tests

Mit Hilfe von zufällig erzeugten Testdaten (siehe Beispiel 4 im Anhang 9.20) entstanden beim Algorithmus „2M1N“ die in den Abbildungen 35 und 36 dargestellten Ergebnisse.

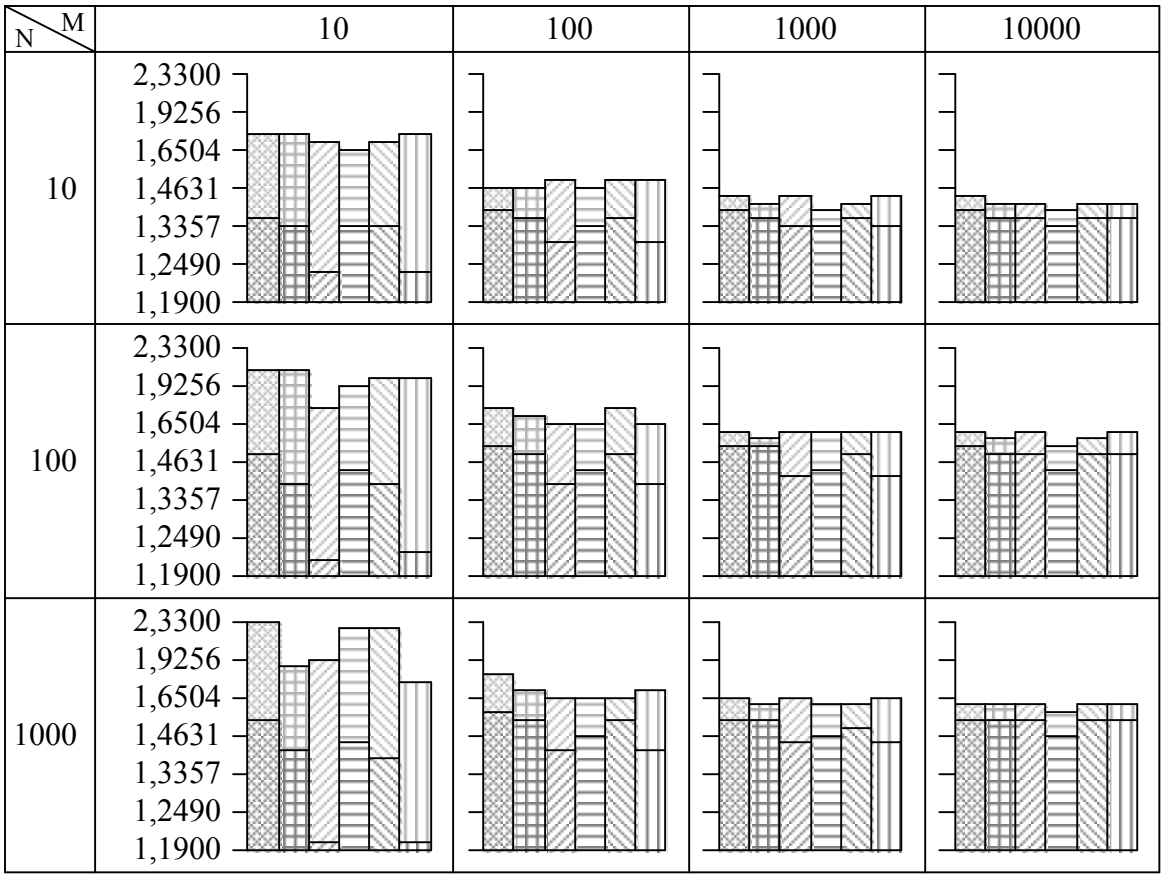


Abbildung 35 - schlechtester (oberste Markierung jeder Säule) und durchschnittlicher Gütefaktor (mittlere Markierung) beim Beispiel 4.

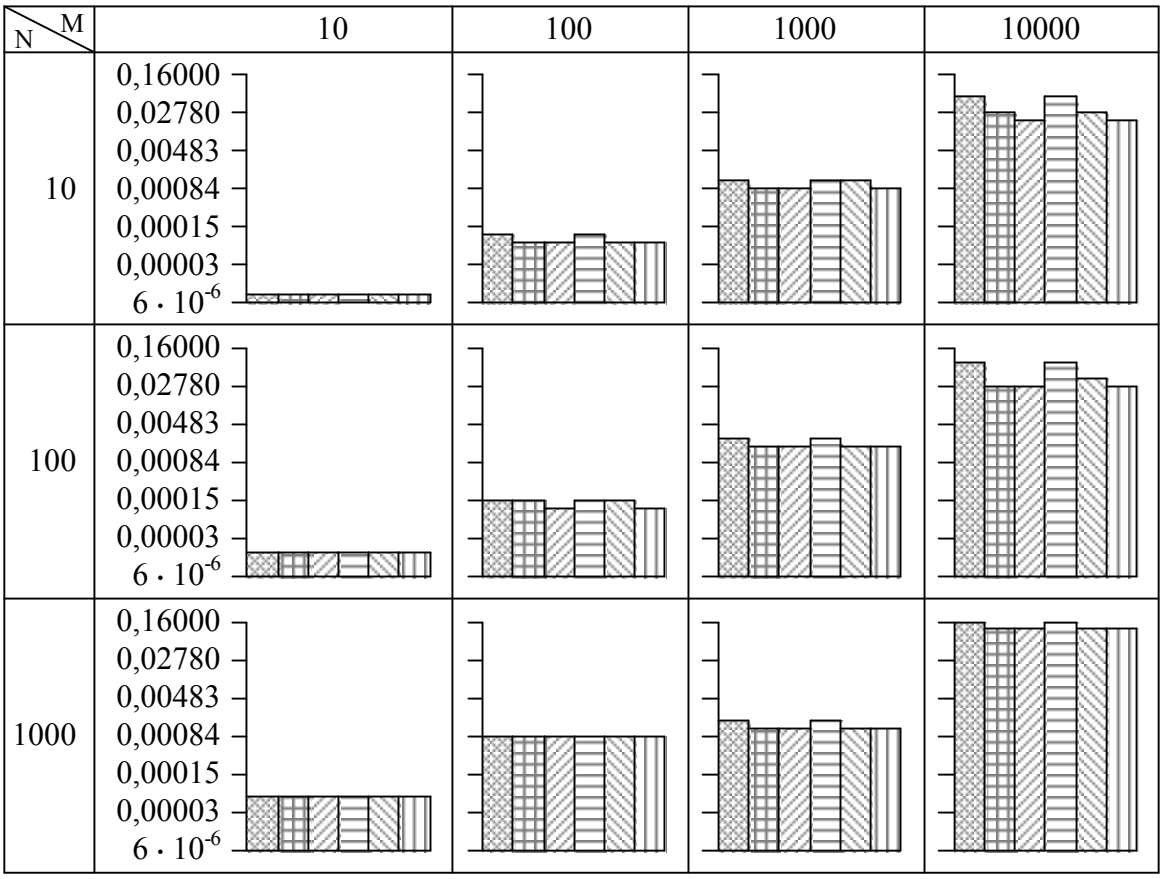


Abbildung 36 - durchschnittliche Laufzeit in Sekunden (Beispiel 4).

Das Beispiel 4 nutzt die Modullaufzeiten von Beispiel 2 und erstellt den Modulabhängigkeitsgraphen analog zu Beispiel 3. In den Abbildungen 35 und 36 wurde bei den linken (rechten) drei Säulen ein kettenförmiger (balancierter) Konstruktionsbaum für aufeinanderfolgende parallele Vereinigungen verwendet. In jeder Dreiergruppe steht die linke (rechte) Säule für einen serien - parallelen Graphen welcher aus ca. viermal so vielen parallelen wie seriellen (seriellen wie parallelen) Vereinigungen hervorgegangen ist. Bei der mittleren Säule jeder Dreiergruppe war das Verhältnis von paralleler zu serieller Vereinigung ca. Eins. Die in den Tests ermittelten Zahlenwerte können auf der CD im Verzeichnis /res/bsp4/ eingesehen werden.
Vergleicht man die Abbildungen 34 und 36 miteinander, so erkennt man, daß „2M1N“ sogar noch etwas schneller als „2M1“ ist. Diese Verbesserung läßt sich durch die Bearbeitung der nicht formbaren Module x bei Prozessoranzahlen i mit $T(x, i) = \infty$ erklären.
Der durchschnittliche Gütefaktor ist bei fast allen Läufen gleich und liegt bei 1,4. Entsteht der serien - parallele Graph aus vielen seriellen Vereinigungen, so ist der durchschnittlich erreichte Gütefaktor immer etwas kleiner als beim Vorliegen vieler paralleler Vereinigungen. Da bei der parallelen Vereinigung von zwei nicht formbaren Modulen x und y im Fall

$$B_{min}(x) + B_{min}(y) > N \quad \text{mit} \quad B_{min}(z) = \min_{i=1}^{N} \{ i : T(z, i) \neq \infty \}$$

nur die Nacheinanderausführung beider Module möglich ist, kann der Algorithmus sehr oft nur die Datenparallelität ausnutzen. Diesen Nachteil beachtet aber die Methode zur Bestimmung einer unteren Schranke für die Gesamtlaufzeit des optimalen Schedules nicht, d.h., es

wird beim Vorliegen vieler paralleler Vereinigungen nur eine grobe Abschätzung des Gütefaktors vorgenommen (siehe Abschnitt 2.2.2).
Entsteht der Modulabhängigkeitsgraph aus vielen seriellen Vereinigungen, so erreicht man oft mittels

$$O \geqslant \max\left\{ \sum_{i=1}^{k} T(x_i, N) \;:\; x_1, ..., x_k \text{ ist ein Pfad in } G \right\}$$

eine gute Abschätzung für die Gesamtlaufzeit des optimalen Schedules.
In Abbildung 35 erkennt man außerdem, daß die Differenz zwischen optimalen und schlechtesten Gütefaktor mit steigendem M abnimmt. Angenommen ein Modul x benötigt zur Ausführung i Prozessoren. Dann ist auch jedes zusammengesetzte Modul, welches x enthält, nur mit mindestens i Prozessoren ausführbar. Durch die hohe Modulanzahl ist es bei den erzeugten Testbeispielen sehr wahrscheinlich, daß viele einfache Module mehr als $½ \cdot N$ Prozessoren benötigen und demzufolge nur noch nacheinander ausführbar sind.

4 Näherungsalgorithmen beim Vorliegen eines beliebigen Modulabhängigkeitsgraphen

4.1 Optimierung von formbaren Modulen

Nachdem bisher Module mit leerem bzw. serien - parallelen Modulabhängigkeitsgraphen betrachtet wurden, befaßt sich dieser Abschnitt mit der näherungsweisen Optimierung von Modulen beim Vorhandensein eines beliebigen Modulabhängigkeitsgraphen. Im ersten Teil wird wieder davon ausgegangen, daß alle Module formbar sind.

4.1.1 Die schichtenorientierten Algorithmen „SchT“ und „SchHT“

Bei diesen Algorithmen wird der vorliegende Modulabhängigkeitsgraph G = (V, E) in mehrere Teilgraphen G_1, ..., G_d zerlegt, so daß die $G_i = (V_i, E_i)$ keine Kanten enthalten und jeder Knoten aus G in exakt einem Teilgraphen vorhanden ist. Existiert in G eine Kante von x nach y mit $x \in V_i$ und $y \in V_j$, so muß für die Zerlegung $i < j$ gelten. Eine derartige Zerlegung von G wird im folgenden als disjunkte Zerlegung bezeichnet.
Für die anschließende Anwendung eines Schedulingalgorithmus ist es zur Erstellung eines sehr guten Schedules in vielen Fällen vorteilhaft, wenn d möglichst klein ist.
Im folgenden wird mit tief(x) (hoch(x)) die Tiefe (Höhe) des Knotens x in G = (V, E) bezeichnet:

$$\text{tief}(x) = \begin{cases} 1 & : x \text{ ist Quelle in } G \\ 1 + \max\{ \text{tief}(y) : (y, x) \in E \} & : \text{sonst} \end{cases},$$

$$\text{hoch}(x) = \begin{cases} 1 & : x \text{ ist Senke in } G \\ 1 + \max\{ \text{hoch}(y) : (x, y) \in E \} & : \text{sonst} \end{cases}.$$

Außerdem steht maxtief für die Tiefe des Graphen G, also

$$\text{maxtief} = \max\{ \text{tief}(x) : x \in V \}.$$

Man erkennt, daß die folgenden beiden Zerlegungen des Modulabhängigkeitsgraphen G = (V, E) disjunkt sind:

a) Für alle $x \in V$: Ordne x dem Graphen $G_{\text{tief}(x)}$ zu, also $x \in V_{\text{tief}(x)}$.

b) Für alle $x \in V$: Ordne x dem Graphen $G_{\text{maxtief} + 1 - \text{hoch}(x)}$ zu, also $x \in V_{\text{maxtief} + 1 - \text{hoch}(x)}$.

Bei beiden Zerlegungen entstehen d = maxtief verschiedene Teilgraphen, d.h., d ist minimal.
Da die Graphen G_1 bis G_d keine Kanten enthalten, liegen bei der Optimierung der in einem G_i enthaltenen Module auch keine Modulabhängigkeiten vor. Demzufolge kann man den Algorithmus „P1+MF“ aus dem Abschnitt 2.1.5 nacheinander auf die Teilgraphen anwenden und erhält d Teilschedules mit den Laufzeiten H_1 bis H_d. Durch die bei der Zerlegung von G geforderte sequentielle Ordnung kann man nun die Startzeit von Modul $x \in V_i$ auf

$$S(x) = S_i(x) + \sum_{j=1}^{i-1} H_j$$

setzen. Dabei steht $S_i(x)$ für die bei der Optimierung der Knoten V_i berechnete Startzeit des Moduls x. Definiert man $B_i(x)$ und $P_i(x)$ äquivalent, so gilt

$$B(x) = B_i(x) \quad \text{und} \quad P(x) = P_i(x).$$

Die Abbildung 37 verdeutlicht diese Konstruktion noch einmal. Der Pseudocode des Algorithmus „SchT“ ist in der Abbildung 38 dargestellt. Eine Zerlegung des Modulabhängigkeitsgraphen in Schichten findet man auch in [16].

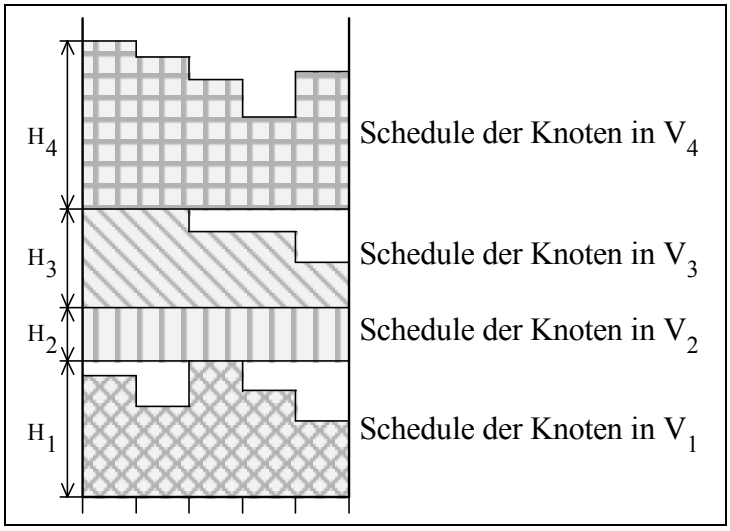


Abbildung 37 - Verdeutlichung der Positionierung der Teilschedules.

```
1)  bestimme für jeden Knoten x die Tiefe tief(x) im Modulabhängigkeitsgraphen G
2)  maxtief = max{ tief(x) : x ∈ V }
3)  schichtstart = 0
4)  für i = 1 bis maxtief
5)      berechne für alle Knoten x mit tief(x) = i unter Verwendung des Algorithmus
        „P1+MF“ den Teilschedule i; man erhält die Modulbreiten B(x), die Referenzpro-
        zessoren P(x), die Startzeiten S_i(x) und die Gesamtlaufzeit des Teilschedules H_i
6)      für alle Knoten x mit tief(x) = i
7)          S(x) = S_i(x) + schichtstart
8)      schichtstart = schichtstart + H_i
```

Abbildung 38 - Algorithmus „SchT“.

Die Laufzeit des Algorithmus setzt sich aus der Bestimmung der Tiefe jedes Knotens und den Abläufen des Algorithmus „P1+MF“ zusammen. Sie liegt also in

$$O(|V| + |E| + M \cdot \log(M) + M \cdot N^2) = O(|E| + M \cdot \log(M) + M \cdot N^2)$$

bei der Verwendung des Positionierungsalgorithmus „HMZ“ bzw. in

$$O(|V| + |E| + M \cdot \log(M) + \min\{ M \cdot N^3; M^2 \cdot N^2\})$$

$$= O(|E| + M \cdot \log(M) + \min\{ M \cdot N^3; M^2 \cdot N^2\})$$

bei der Nutzung von „BMZ“. Dabei bezeichnet $M = |V|$ die Modulanzahl und N die Gesamtprozessoranzahl.

Lemma 28: *Es sei A ein Schedulingalgorithmus, welcher die folgenden drei Schritte ausführt:*

1) *Der Algorithmus A zerlegt den Modulabhängigkeitsgraphen G = (V, E) in mehrere Teilgraphen G_1, ..., G_d mit den folgenden drei Eigenschaften:*
   a) *Alle $G_i = (V_i, E_i)$ enthalten keine Kanten.*
   b) *Jeder Knoten aus G ist exakt in einem Teilgraphen G_i vorhanden.*
   c) *Existiert in G eine Kante von x nach y mit $x \in V_i$ und $y \in V_j$, so muß für die Zerlegung $i < j$ gelten.*
2) *Der Algorithmus A nutzt zur Berechnung aller d Teilschedules für die Teilgraphen G_1, ..., G_d den Algorithmus „P1+MF“.*
3) *Der Schedule aller in G enthaltener Module erstellt der Algorithmus A aus den Teilschedules gemäß Abbildung 37.*

Dann gilt für die garantierte Ergebnisgüte G des von A erstellten Schedules

$$G \leqslant 1 + \max_{i=1}^{d} \left\{ \frac{N \cdot \max_{x \in V_i} \{ T(x, 1) \}}{\sum_{x \in V_i} T(x, 1)} \right\}.$$

Wie bisher stehe dabei N für die Gesamtanzahl der Prozessoren und T(x, 1) für die sequentielle Laufzeit des Moduls x.

Beweis (siehe auch Beweis zu Lemma 3): Zur besseren Lesbarkeit werden die folgenden Abkürzungen verwendet:

$$t_i = \max_{x \in V_i} \{ T(x, 1) \} \quad \text{und} \quad f_i = \sum_{x \in V_i} T(x, 1).$$

Im ersten Teil des Beweises wird gezeigt, daß für jeden Teilschedule i ($1 \leqslant i \leqslant d$) die Ungleichung

$$\frac{f_i}{N} \cdot \left(1 + \frac{N \cdot t_i}{f_i}\right) \geqslant H_i \quad \Leftrightarrow \quad \frac{f_i}{N} + t_i \geqslant H_i$$

gilt.
Die Gesamtlaufzeit des vom Algorithmus „P1“ berechneten Schedules aller in V_i enthaltener Module werde mit h_i bezeichnet. Da der Algorithmus „P1+MF“ nie einen schlechteren Schedule als der Algorithmus „P1“ erstellt (siehe Abschnitt 2.1.5), folgt $h_i \geqslant H_i$. Es reicht also,

$$\frac{f_i}{N} + t_i \geqslant h_i$$

zu zeigen.
Mit x bezeichne man ein beliebiges Modul, welches im von „P1“ berechneten Teilschedule i erst zum Zeitpunkt h_i beendet wird. Der Algorithmus „P1“ positioniert x auf dem Prozessor, welcher die aktuell kleinste Endzeit hat (siehe Abschnitt 2.1.1). Da der Algorithmus „P1“ alle Module mit nur einem Prozessor ausführt, sind alle N Prozessoren bis zum Startzeitpunkt von x mit der Ausführung anderer Module beschäftigt. Aus der Wahl von x folgt, daß x im von „P1“ berechneten Teilschedule i zum Zeitpunkt $h_i - T(x, 1)$ gestartet wird. Aus diesem Grund ist im von „P1“ berechneten Teilschedule i eine Mindestfläche von $(h_i - T(x, 1)) \cdot N$ belegt. Es gilt also

$$(h_i - T(x, 1)) \cdot N \leqslant f_i \quad \Rightarrow \quad (h_i - t_i) \cdot N \leqslant f_i \quad \Rightarrow \quad \frac{f_i}{N} + t_i \geqslant h_i.$$

Damit hat man die Gültigkeit von

$$\frac{f_i}{N} \cdot \left(1 + \frac{N \cdot t_i}{f_i}\right) \geqslant H_i$$

für $1 \leqslant i \leqslant d$ gezeigt. Addiert man nun die d verschiedenen Ungleichungen, so erhält man

$$\sum_{i=1}^{d} \frac{f_i}{N} \cdot \left(1 + \frac{N \cdot t_i}{f_i}\right) \geqslant \sum_{i=1}^{d} H_i \quad \Rightarrow \quad \sum_{i=1}^{d} \frac{f_i}{N} \cdot \max_{j=1}^{d}\left\{1 + \frac{N \cdot t_j}{f_j}\right\} \geqslant \sum_{i=1}^{d} H_i$$

$$\Leftrightarrow \quad \max_{j=1}^{d}\left\{1 + \frac{N \cdot t_j}{f_j}\right\} \cdot \sum_{i=1}^{d} \frac{f_i}{N} \geqslant H \quad \Leftrightarrow \quad \max_{j=1}^{d}\left\{1 + \frac{N \cdot t_j}{f_j}\right\} \cdot \frac{1}{N} \cdot \sum_{x=1}^{M} T(x, 1) \geqslant H.$$

Bekanntlich ist

$$O \geqslant \frac{1}{N} \cdot \sum_{x=1}^{M} T(x, 1),$$

wobei O die Gesamtlaufzeit des optimalen Schedules bezeichnet. Es ergibt sich also

$$\max_{j=1}^{d}\left\{1 + \frac{N \cdot t_j}{f_j}\right\} \cdot O \geqslant H \quad \text{bzw.} \quad \max_{j=1}^{d}\left\{1 + \frac{N \cdot t_j}{f_j}\right\} \geqslant G.$$

■

Folgerung aus Lemma 28: *Der Modulabhängigkeitsgraph G = (V, E) habe die maximale Tiefe maxtief. Mit tief(x) werde die Tiefe des Knotens x in G bezeichnet. Außerdem stehe N für die Gesamtprozessoranzahl und T(x, 1) für die sequentielle Laufzeit des Moduls x. Dann erreicht der Algorithmus „SchT" garantiert einen Gütefaktor G von*

$$G \leqslant 1 + \max_{i=1}^{maxtief} \left\{ \frac{N \cdot \max\limits_{\substack{x \in V \\ tief(x) = i}} \{ T(x, 1) \}}{\sum\limits_{\substack{x \in V \\ tief(x) = i}} T(x, 1)} \right\}.$$

Bei der Betrachtung von Lemma 28 erkennt man sofort, daß die garantierte Ergebnisgüte G des in Lemma 28 definierten Algorithmus A von der Zerlegung des Modulabhängigkeitsgraphen abhängt. Um einen möglichst kleine obere Schranke für G zu erreichen, muß der Wert $\max_{j=1}^{d}\left\{1 + \frac{N \cdot t_j}{f_j}\right\}$ minimal sein. Der nachfolgende Algorithmus „SchHT" versucht den Modulabhängigkeitsgraphen so in maxtief Teilgraphen zu zerlegen, daß $\max_{j=1}^{maxtief}\left\{1 + \frac{N \cdot t_j}{f_j}\right\}$ möglichst klein ist.

Damit nicht mehr als maxtief Teilgraphen entstehen, ist es notwendig, alle Knoten x, die in G auf einem Pfad der Länge maxtief liegen, zu $G_{tief(x)}$ zuzuordnen. Für diese Knoten x gilt

$$tief(x) = maxtief + 1 - hoch(x).$$

Für alle anderen Knoten y wird anschließend die Menge der Teilgraphen bestimmt, zu welchen der Knoten y hinzugefügt werden kann, ohne daß insgesamt mehr als maxtief Teilgraphen entstehen oder die im Lemma 28 notierten Anforderungen an die Teilgraphen verletzt werden. Man erkennt sofort, daß unter diesen Bedingungen der Knoten y nur in die Teilgraphen $G_{tief(y)}$ bis $G_{maxtief + 1 - hoch(y)}$ eingefügt werden kann.

Im Verlauf des Algorithmus „SchHT“ soll durch geeignete Bestimmung der Werte ti(y) und ho(y) zu jedem Zeitpunkt gelten, daß ein noch nicht zu einem Teilgraphen hinzugefügter Knoten y immer zu einem beliebigen Teilgraphen $G_{ti(y)}$ bis $G_{maxtief + 1 - ho(y)}$ hinzugefügt werden kann. Wie man bereits gesehen hat, gilt nach der Zuordnung aller Knoten x mit

$$tief(x) = maxtief + 1 - hoch(x)$$

zum Teilgraphen $G_{tief(x)}$ für alle noch nicht einem Teilgraphen zugeordneten Knoten y

$$ti(y) = tief(y) \quad \text{und} \quad ho(y) = hoch(y).$$

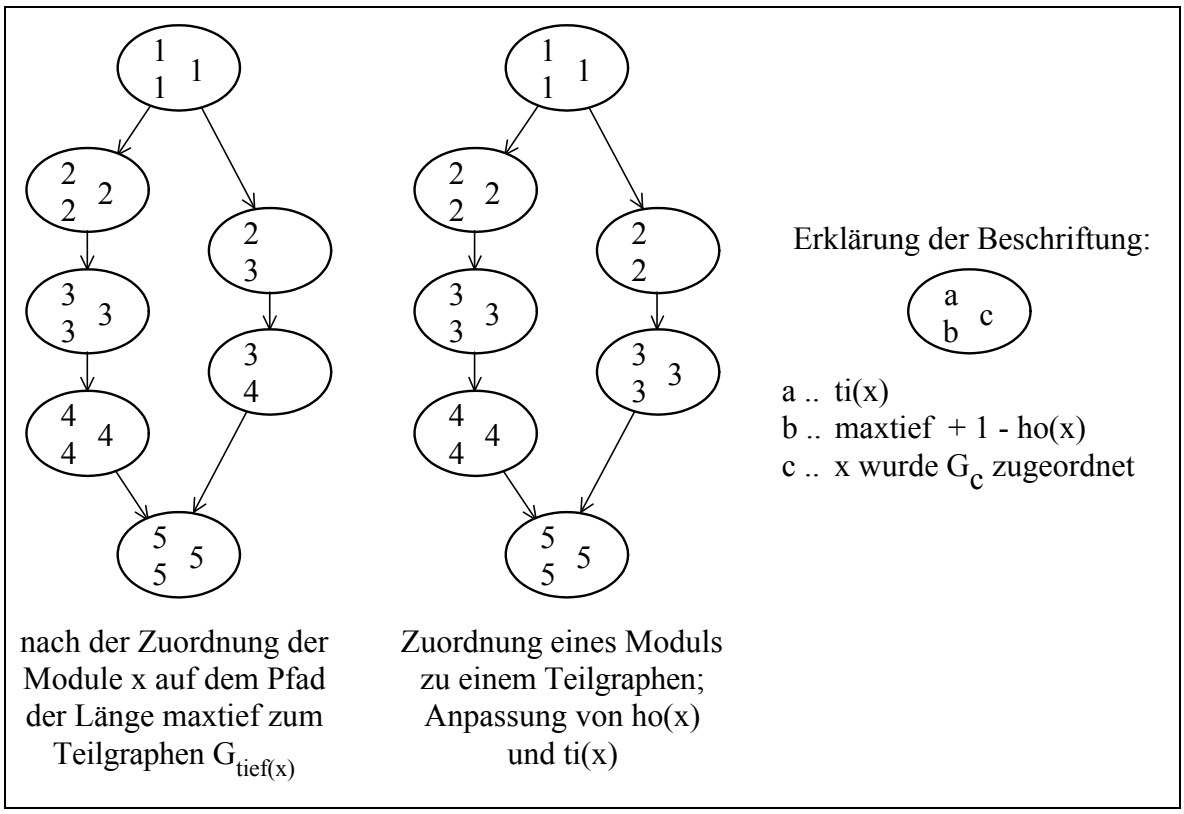


Abbildung 39 - Festlegen des Teilgraphen für Module, die nicht auf dem längsten Pfad liegen.

Wird nun ein Knoten y zum Teilgraphen G_i zugeordnet, so müssen ti(z) und ho(z) für alle noch nicht zu einem Teilgraphen hinzugefügten Knoten z angepaßt werden, damit diese Knoten weiterhin zu einem beliebigen Teilgraphen $G_{ti(z)}$ bis $G_{maxtief + 1 - ho(z)}$ hinzugefügt werden können. Diese Eigenschaft ist erfüllt, wenn für nach der Zuordnung von y zu G_i die folgenden Ungleichungen gelten:

$$ti(z) \geqslant i + 1 \ \forall \text{Nachfolger z von y} \quad \text{und} \quad ho(z) \geqslant maxtief + 2 - i \ \forall \text{Vorgänger z von y.}$$

Ändert sich ti (z) bzw. ho(z) eines Knotens z, so müssen auch alle Nachfolger bzw. Vorgänger dieses Knotens über diese Änderung informiert werden, damit die Eigenschaft

$$ti\ (w) > ti\ (z)\ \forall \text{Nachfolger w von z} \quad \text{und} \quad ho(w) > ho(z)\ \forall \text{Vorgänger w von z}$$

erhalten bleibt. Die Abbildung 39 zeigt diese Tatsache an einem Beispiel.

Wird der Knoten y zu einem Teilgraphen G_i zugeordnet, so ändern sich die Werte von t_i und f_i. Da es im Modulabhängigkeitsgraphen G zu jedem $i \in [1; \text{maxtief}]$ mindestens einen Knoten x mit

$$i = ti(x) = \text{maxtief} + 1 - ho(x)$$

gibt, sind vor dem Einfügen des ersten Knotens y mit

$$ti(y) \neq \text{maxtief} + 1 - ho(y)$$

alle t_j und f_j ungleich Null.
Der Algorithmus „SchHT“ versucht nun bei der Zuordnung eines Knotens y mit

$$ti(y) \neq \text{maxtief} + 1 - ho(y)$$

zu einem Teilgraphen, den Wert $\max_{j=1}^{\text{maxtief}} \left\{ 1 + \frac{N \cdot t_j}{f_j} \right\}$ zu minimieren. Dazu ist es ausreichend, daß der maximale Quotient $\frac{t_j}{f_j}$ aller Teilgraphen G_j möglichst klein ist.

Man sieht sofort, daß sich der Quotient $\frac{t_j}{f_j}$ nur beim Hinzufügen eines Moduls y mit $T(y, 1) > t_j$ zu G_j vergrößern kann. Aus diesem Grund ist es sicherlich vorteilhaft, die Module mit der größten sequentiellen Laufzeit zuerst zu einem Teilgraphen hinzuzufügen. Beim Hinzufügen von Modul y wird der Teilgraph G_j mit dem größten Quotienten $\frac{t_j}{f_j}$ gewählt, bei dem nach der Hinzunahme des Moduls y keine Vergrößerung des Quotienten $\frac{t_j}{f_j}$ eintritt. Sollte kein derartiger Teilgraph existieren, so entscheidet sich „SchHT“ für den Teilgraphen, für den nach der Hinzunahme von y der kleinste Quotient $\frac{t_j}{f_j}$ resultiert.
Der Pseudocode des Algorithmus „SchHT“ ist in den Abbildungen 40 bis 42 dargestellt.

Leider ist die Laufzeit des Algorithmus „SchHT“ im schlechtesten Fall um Größenordnungen höher als die von „SchT“. Die Festlegung des Teilgraphen für alle Knoten benötigt $O(M \cdot \log(M) + \text{maxtief} \cdot M + \text{maxtief} \cdot |E|)$ Schritte. Der erste Summand entsteht durch die Sortierung der Module. Der Term $\text{maxtief} \cdot M$ ergibt sich aus den Schleifendurchläufen ab Zeile 9) und Zeile 11). Dabei wurde die Zeit für die rekursiven Aufrufe der Unterroutinen info_kind(x) und info_vorgänger(x) nicht mit berücksichtigt. Deshalb muß man noch die Anzahl der Unterroutinenaufrufe von info_kind(x) und info_vorgänger(x) bzgl. eines beliebigen Knotens x betrachten. Vor jedem Start einer dieser beiden Unterroutinen mit dem Parameter x verringert sich der Wert von

$$\text{maxtief} + 1 - ho(x) - ti(x)$$

um mindestens Eins. Da der Maximalwert dieses Terms maxtief - 1 ist, können die zum Knoten x adjazenten Kanten höchstens O(maxtief) - mal betrachtet werden. Man gelangt somit zu maximal $O(\text{maxtief} \cdot |E|)$ Aufrufen der Unterroutinen „info_kind“ und „info_vorgänger“.
Das diese Laufzeit tatsächlich erreicht werden kann, zeigt das im Anhang 9.21.1 angegebene Beispiel.
Die Gesamtlaufzeit von „SchHT“ beträgt demzufolge

$$O(\text{maxtief} \cdot (|E| + M) + M \cdot \log(M) + M \cdot N^2)$$

1) $\forall x \in [1; M]$: $ti(x) = tief(x)$; $ho(x) = hoch(x)$
2) $maxtief = max\{ ti(x) : x \in V \}$; $W = \varnothing$
3) **für** $i = 1$ **bis** $maxtief$: $t_i = 0$; $f_i = 0$; $V_i = \varnothing$
4) **für alle** Knoten x
5) **ist** $ti(x) = maxtief + 1 - ho(x)$, **so** tue
6) $V_{ti(x)} = V_{ti\,(x)} \cup \{x\}$; $f_{ti(x)} = f_{ti(x)} + T(x, 1)$
7) **ist** $t_{ti\,(x)} < T(x, 1)$, **so** $t_{ti(x)} = T(x, 1)$
8) **anderenfalls** $W = W \cup \{x\}$
9) **für alle** Knoten $x \in W$ in absteigender Reihenfolge bzgl. $T(x, 1)$
10) $k = ti\,(x)$
11) **für** $i = ti(x) + 1$ **bis** $maxtief + 1 - ho(x)$
12) **ist** $\frac{t_k}{f_k} < \frac{t_i}{f_i}$, **so** tue
13) **ist** $\frac{max\{ t_i; T(x, 1) \}}{f_i + t(x, 1)} \leqslant \frac{t_i}{f_i}$, **so** $k = i$
14) **anderenfalls**
15) **ist** $\frac{max\{ t_k; T(x, 1) \}}{f_k + t(x, 1)} > max\left\{ \frac{t_k}{f_k}; \frac{max\{ t_i; T(x, 1) \}}{f_i + t(x, 1)} \right\}$, **so** $k = i$
16) **anderenfalls**
17) **ist** $\frac{max\{ t_k; T(x, 1) \}}{f_k + t(x, 1)} > \frac{t_k}{f_k}$, **so** tue
18) **ist** $\frac{max\{ t_i; T(x, 1) \}}{f_i + t(x, 1)} \leqslant \frac{t_i}{f_i}$, **so** $k = i$
19) **anderenfalls**
20) **ist** $\frac{max\{ t_k; T(x, 1) \}}{f_k + t(x, 1)} > \frac{max\{ t_i; T(x, 1) \}}{f_i + t(x, 1)}$, **so** $k = i$
21) **ist** $t_k < T(x, 1)$, **so** $t_k = T(x, 1)$
22) $V_k = V_k \cup \{x\}$; $f_k = f_k + T(x, 1)$
23) **ist** $k > ti(x)$, **so** tue
24) $ti(x) = k$
25) **starte** die Unterroutine info_kinder(x) in Zeile 35)
26) **ist** $k < maxtief + 1 - ho(x)$, **so** tue
27) $ho(x) = maxtief + 1 - k$
28) **starte** die Unterroutine info_vorgänger(x) in Zeile 41)
29) $schichtstart = 0$
30) **für** $i = 1$ **bis** $maxtief$
31) berechne für alle Knoten $x \in V_i$ unter Verwendung des Algorithmus „P1+MF“ den Teilschedule i; man erhält die Modulbreiten $B(x)$, die Referenzprozessoren $P(x)$, die Startzeiten $S_i(x)$ und die Gesamtlaufzeit des Teilschedules H_i
32) **für alle** Knoten $x \in V_i$: $S(x) = S_i(x) + schichtstart$
33) $H_i = max\{ S_i(x) + T(x) : x \in [1; M]$ und $schicht(x) = i \}$
34) $schichtstart = schichtstart + H_i$

Abbildung 40 - Algorithmus „SchHT“.

```
35) Unterroutine info_kinder(x)
36) für alle Kinder y des Knotens x, also ∀y mit (x, y) ∈ E
37)       ist ti(y) < ti(x)+ 1, so tue
38)             ti(y) = ti(x) + 1
39)             starte info_kinder(y)
40) Rücksprung aus der Unterroutine
```

Abbildung 41 - Algorithmus „SchHT“ - Unterroutine „info_kinder“.

```
41) Unterroutine info_vorgänger(x)
42) für alle Vorgänger y des Knotens x, also ∀y mit (y, x) ∈ E
43)       ist ho(y) < ho(x) + 1, so tue
44)             ho(y) = ho(x) + 1
45)             starte info_vorgänger(y)
46) Rücksprung aus der Unterroutine
```

Abbildung 42 - Algorithmus „SchHT“ - Unterroutine „info_vorgänger“.

bei der Verwendung des Positionierungsalgorithmus „HMZ“ bzw.

$$O\left(\text{maxtief} \cdot (|E| + M) + M \cdot \log(M) + \min\left\{ M \cdot N^2 \cdot \log(N) + N^4; M^2 \right\} \cdot N^2 \right\})$$

bei der Nutzung von „BMZ“. Ist die Tiefe maxtief des Modulabhängigkeitsgraphen nicht bekannt, so kann sie durch

$$\text{maxtief} \leqslant \min\{M; |E| + 1\}$$

abgeschätzt werden.

Leider ist der Mehraufwand von „SchHT“ bzgl. „SchT“ kein Garant für die Erstellung eines besseren Schedules. Dazu ist im Anhang 9.21.2 ein kleines Beispiel angegeben.
Obwohl „SchT“ und „SchHT“ sicherlich im praktischen Einsatz sehr gute Schedules erstellen (siehe Abschnitt 4.1.5), kann man auch ein Beispiel angeben, bei welchem sie nur einen Gütefaktor von $N - \varepsilon$ mit $\varepsilon \in [N - 1; 0)$ erreichen. Ein solches Beispiel wird im Anhang 9.21.3 vorgestellt.

An dieser Stelle sei noch auf das Lemma 1 in [31] verwiesen. Dort wird gezeigt, daß für jeden Algorithmus, der den Anforderungen von Lemma 28 genügt, ein Beispiel existiert, bei welchem nur ein Gütefaktor von $N - \varepsilon$ für alle $\varepsilon > 0$ erzielt wird. Diese Aussage gilt sogar für Module mit fester Prozessoranzahl (wie beim Rechteck - Füll - Problem).

4.1.2 Der waldorientierte Algorithmus „WaSP“

Nachdem der Modulabhängigkeitsgraph bisher nur in Teilgraphen, welche keine Abhängigkeiten mehr enthielten, zerlegt wurde, soll er jetzt in mehrere serien - parallele Teilgraphen untergliedert werden. Dabei beschränkt man sich an dieser Stelle auf eine Untergruppe der serien - parallele Graphen, nämlich Wälder. Ein Graph $G_i = (V_i, E_i)$ ist genau dann ein Wald, wenn j ($j \geqslant 1$) Bäume $G_{i_k} = (V_{i_k}, E_{i_k})$ mit

$$V_i = \bigcup_{k=1}^{j} V_{i_k}, \quad E_i = \bigcup_{k=1}^{j} E_{i_k} \quad \text{und} \quad V_{i_k} \cap V_{i_m} = \varnothing \quad (\forall k, m \text{ mit } 1 \leqslant k < m \leqslant j)$$

existieren.

Bei der Zerlegung des Modulabhängigkeitsgraphen G = (V, E) in mehrere Wälder G_1, ..., G_k mit $G_i = (V_i, E_i)$ muß die folgende Eigenschaft beachtet werden: Ist (x, y) ∈ E, so muß entweder x ∈ V_i und y ∈ V_j mit i < j, oder {x, y} ⊆ V_i und es gibt in G_i einen gerichteten Pfad von x nach y, folgen.

```
1)  i = 1; K = {x : x ∈ G und indeg(x) = 0}; G_1 = ∅
2)  solange K ≠ ∅ ist, tue
3)      solange K ≠ ∅ ist, tue
4)          wähle x ∈ K
5)          K = K \ {x}; V_i = V_i ∪ {x}
6)          für alle y mit (x, y) ∈ E
7)              ist indeg(y) = 1, so E_i = E_i ∪ {(x, y)}; K = K ∪ {y}
8)      E = E \ (V_i × V); V = V \ V_i
9)      i = i + 1; K = {x : x ∈ G und indeg(x) = 0}; G_i = ∅
10) für j = 1 bis i - 1
11)     nutze den Algorithmus „2M1“, um die Module aus G_j zu positionieren; das Ergebnis liege in B(x), P(x) und S_j(x) sowie H_j
12) für x = 1 bis M
13)     es sei x ∈ V_i
14)     S(x) = S_i(x) + Σ_{j=1}^{i-1} H_j
Die Funktion indeg(x) bezieht sich dabei immer auf den Graphen G in seinem aktuellen Zustand.
```

Abbildung 43 - Algorithmus „WaSP“.

Der Algorithmus „WaSP“ (siehe Abbildung 43) übernimmt am Anfang alle Quellen des Modulabhängigkeitsgraphen G in den ersten Wald $G_1 = (V_1, E_1)$. Anschließend überprüft „WaSP“ für alle in V_1 enthaltenen Knoten x, ob im Graphen G ein Kind y von x den Eingangsgrad Eins hat. In diesem Fall wird y zu V_1 und (x, y) zu E_1 hinzugefügt. Durch diesen Schritt bleibt G_1 garantiert ein Wald. Kann kein Kind y eines bereits nach V_1 übernommenen Knotens x mehr zu G_1 hinzugefügt werden, so löscht der Algorithmus alle nach V_1 übernommenen Knoten aus G. Dadurch entstehen in G neue Quellen, und es kann ein zweiter Wald gebildet werden.
Da G irgendwann leer ist, terminiert der Algorithmus. Im Anschluß werden die einzelnen Wälder als serien - parallele Graphen mit Hilfe von „2M1“ optimiert. Den Schedule aller Module erhält man, indem man die einzelnen Teilschedules wie im Abschnitt 4.1.1 zusammenfügt.
Man sieht, daß das Erzeugen der einzelnen Wälder in O(M + |E|) geschehen kann. Aufgrund der geringen Laufzeit von „2M1“ benötigt „WaSP“ auch nur O(M · N + |E|) Schritte.
Leider kann man für die Ergebnisgüte von „WaSP“ durch die Verwendung von „2M1“ keine konstante Obergrenze angeben. Man erkennt jedoch auch für „WaSP“, daß der Schedule,

welcher allen Modulen N Prozessoren zuordnet (Gang - Scheduling), mit bearbeitet wird. Demzufolge gelten für „WaSP“ die Aussagen von Lemma 25 und Lemma 26.

4.1.3 Der reduktionsbasierte Algorithmus „ReSP“

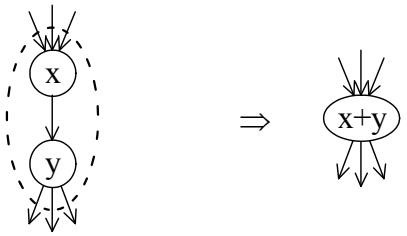
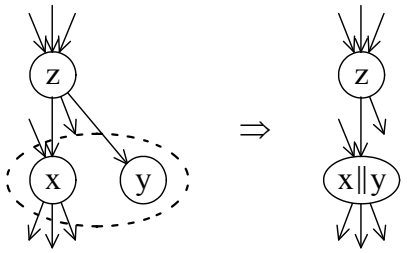
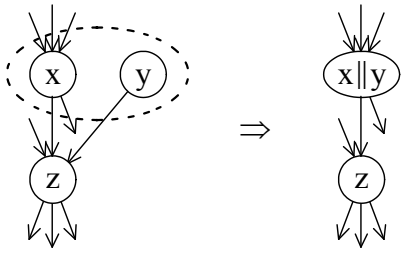
Nachdem bei „WaSP“ nur ein spezieller serien - paralleler Graph (nämlich eine Menge von Wäldern) erzeugt wurde, soll nun der Modulabhängigkeitsgraph G durch Hinzunahme bzw. Löschung von möglichst wenigen Kanten in einen serien - parallelen Graphen G’ überführt werden. Die Überführung von G zu G’ geschieht mit Hilfe des Transformationsalgorithmus „kGzuB“:

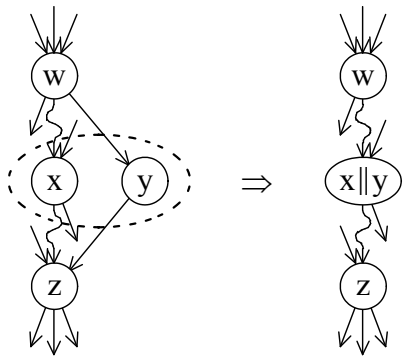
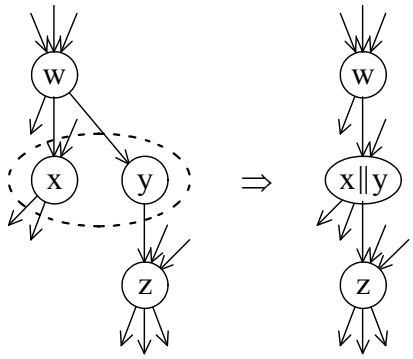
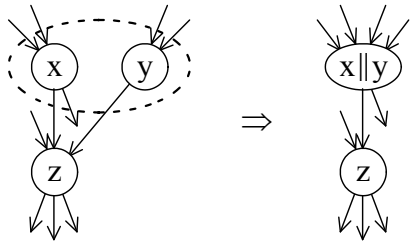
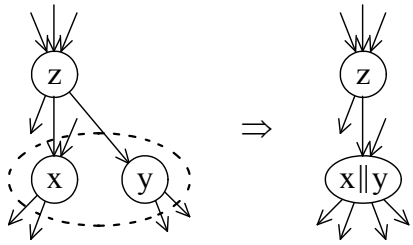
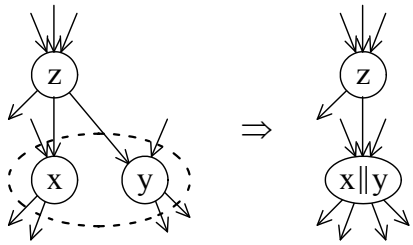
Man definiere die Funktion k, welche jedem Knoten $x \in V$ des Modulabhängigkeitsgraphen $G = (V, E)$ ein Konstruktionsterm (siehe Definition 9) zuordnet, mittels

$$k(x) = x.$$

Der Algorithmus „kGzuB“ reduziert mit Hilfe der Transformationsregeln 1) bis 7) den Modulabhängigkeitsgraph auf einen Graphen ohne Kanten. Man beachte, daß sich G und k bei der Anwendung der Transformationsregeln ändern und immer vom aktuellen Graphen $G = (V, E)$ und der aktuellen Definition der Funktion k ausgegangen wird.

Bei der Auswahl einer Transformationsregel hat immer die Transformationsregel mit der kleinsten Nummer die höchste Priorität. Die Transformationsregel 2) wird also nur verwendet, wenn an keiner Stelle des Graphen G die Transformationsregel 1) anwendbar ist. Ob die Transformation 4a) oder 4b) genutzt wird, entscheidet der Algorithmus erst, nachdem er den Knoten y festgelegt und die Existenz eines zweiten Pfades von w nach z überprüft hat. Ist die gleiche Transformation an verschiedenen Stellen des Graphen möglich, so wird zufällig eine Stelle ausgewählt.

1)		Voraussetzungen: $x, y \in V$; $outdeg(x) = 1$; $indeg(y) = 1$; $(x, y) \in E$ Transformation: $k(x) = (k(x)) + (k(y))$; $E = E \cup \{ (x, v) : (y, v) \in E \}$; $E = E \setminus \{ (y, v) : v \in V \}$; $E = E \setminus \{ (x, y) \}$; $V = V \setminus \{y\}$
2)		Voraussetzungen: $x, y, z \in V$; $indeg(y) = 1$; $outdeg(y) = 0$; $(z, x) \in E$; $(z, y) \in E$ Transformation: $k(x) = (k(x)) \parallel (k(y))$; $E = E \setminus \{ (z, y) \}$; $V = V \setminus \{y\}$
3)		Voraussetzungen: $x, y, z \in V$; $indeg(y) = 0$; $outdeg(y) = 1$; $(x, z) \in E$; $(y, z) \in E$ Transformation: $k(x) = (k(x)) \parallel (k(y))$; $E = E \setminus \{ (y, z) \}$; $V = V \setminus \{y\}$

4a)		Voraussetzungen: w, x, y, z ∈ V; indeg(y) = 1; outdeg(y) = 1; (w, y) ∈ E; (y, z) ∈ E; Pfad w→x; Pfad x→z Transformation: k(x) = (k(x)) ‖ (k(y)); E = E \ { (w, y); (y, z) }; V = V \ {y}
4b)		Voraussetzungen: w, x, y, z ∈ V; indeg(y) = 1; outdeg(y) = 1; (w, y) ∈ E; (y, z) ∈ E; (w, x) ∈ E Transformation: k(x) = (k(x)) ‖ (k(y)); E = E ∪ { (x, z) }; E = E \ { (w, y); (y, z) }; V = V \ {y}
5)		Voraussetzungen: x, y, z ∈ V; outdeg(y) = 1; (x, z) ∈ E; (y, z) ∈ E Transformation: k(x) = (k(x)) ‖ (k(y)); E = E ∪ { (v, x) : (v, y) ∈ E }; E = E \ { (v, y) : v ∈ V }; E = E \ { (y, z) }; V = V \ {y}
6)		Voraussetzungen: x, y, z ∈ V; indeg(y) = 1; (z, x) ∈ E; (z, y) ∈ E Transformation: k(x) = (k(x)) ‖ (k(y)); E = E ∪ { (x, v) : (y, v) ∈ E }; E = E \ { (y, v) : v ∈ V }; E = E \ { (z, y) }; V = V \ {y}
7)		Voraussetzungen: x, y, z ∈ V; (z, x) ∈ E; (z, y) ∈ E Transformation: k(x) = (k(x)) ‖ (k(y)); E = E ∪ { (v, x) : (v, y) ∈ E }; E = E ∪ { (x, v) : (y, v) ∈ E }; E = E \ { (v, y) : v ∈ V }; E = E \ { (y, v) : v ∈ V }; V = V \ {y}

Ein Beispiel eines „kGzuB“ - Laufes und der vollständige Pseudocode ist im Anhang 9.22 angegeben.

Definition 13: *Eine Kante (x, y) eines Graphen G heißt genau dann transitiv, wenn es auch nach der Löschung von (x, y) noch einen Pfad von x nach y in G gibt.*

Eine transitive Kante spezifiziert im Modulabhängigkeitsgraphen keine neue Abhängigkeit zwischen zwei Modulen. Aus diesem Grund löscht der Algorithmus „kGzuB“ vor der ersten Transformation alle im übergebenen Modulabhängigkeitsgraphen enthaltenen transitiven Kanten.

Lemma 29: *Es sei G = (V, E) ein Modulabhängigkeitsgraph ohne transitive Kanten. Der Graph G' = (V', E') entstehe durch die Anwendung einer der Transformationsregeln 1) bis 4a) auf G. Dann enthält G' keine transitiven Kanten.*

Beweis: Bei der Anwendung der Transformationsregel 1) auf die Kante $(x, y) \in E$ mit outdeg(x) = 1 und indeg(y) = 1 gelte

$$E' \setminus E = \{ (x, v_1), ..., (x, v_k) \}.$$

(a) Angenommen eine Kante $(x, v_i) \in E'$ mit $1 \leqslant i \leqslant k$ ist Teil des Pfades $w \rightarrow x \rightarrow v_i \rightarrow z$ in G' welcher die Kante $(w, z) \in E'$ als transitive Kante kennzeichnet. Dann wäre die Kante $(w, z) \in E$ aufgrund des Pfades $w \rightarrow x \rightarrow y \rightarrow v_i \rightarrow z$ in G bereits eine transitive Kante in G gewesen.

(b) Angenommen eine Kante $(x, v_i) \in E'$ mit $1 \leqslant i \leqslant k$ ist selbst transitiv, d.h., es gibt eine Kante $(x, w) \in E'$ und einen Pfad $w \rightarrow v_i$ in G'. Da in G outdeg(x) = 1 mit $(x, y) \in E$ gilt und $(x, y) \notin E'$ ist, muß ein $j \in [1; k]$ mit $i \neq j$ und $w = v_j$ existieren. Demzufolge enthielt G bereits die transitive Kante $(y, v_i) \in E$, denn es gab die Kante $(y, w) \in E$ und den Pfad $w \rightarrow v_i$ in G.

Aus (a) und (b) folgt, daß bei der Ausführung der Transformationsregel 1) keine transitive Kante entsteht.
Bei der Anwendung der Transformationsregeln 2) bis 4a) gilt $E' \subset E$. Da G keine transitiven Kanten enthält, enthält auch G' keine transitiven Kanten.

■

Es kann also nur nach der Anwendung einer der Transformationsregeln 4b) bis 7) zur Existenz von transitiven Kanten kommen. Diese löscht der Algorithmus „kGzuB“ sofort wieder.

Lemma 30: *Es sei G = (V, E) ein zyklusfreier Modulabhängigkeitsgraph ohne transitive Kanten. Der Graph G' = (V', E') entstehe durch die Anwendung einer der Transformationsregeln 1) bis 7) auf G. Dann enthält G' keinen Zyklus.*

Beweis: Bei der Anwendung der Transformationsregel 1) auf die Kante $(x, y) \in E$ mit outdeg(x) = 1 und indeg(y) = 1 gelte

$$E' \setminus E = \{ (x, v_1), ..., (x, v_k) \}.$$

Angenommen die Kante $(x, v_i) \in E'$ mit $1 \leqslant i \leqslant k$ ist Teil eines Zyklus in G', d.h., es gibt einen Pfad $v_i \rightarrow x$ in G'. Da G' nach Lemma 29 keine transitiven Kanten enthält, liegt auf dem Pfad $v_i \rightarrow x$ in G' keiner der Knoten v_j mit $1 \leqslant j \leqslant k$ und $j \neq i$. Demzufolge existierte bereits in G der Pfad $v_i \rightarrow x$. Da $(x, y) \in E$ und $(y, v_i) \in E$ gilt, existierte bereits in G ein Zyklus.

Bei der Anwendung der Transformationsregeln 2) bis 4a) gilt $E' \subset E$. Da G keinen Zyklus enthält, enthält auch G' keinen Zyklus.
Die Transformationsregel 4b) ist ein Spezialfall der Transformationsregel 6) mit outdeg(y) = 1. Aus diesem Grund können bei der Anwendung der Transformationsregel 4b) nur Zyklen entstehen, wenn dies bei der Transformationsregel 6) möglich ist.
Die Anwendung der Transformationsregel 5) auf G kann mit der Anwendung der Transformationsregel 6) auf $\overline{G} = (V, \overline{E})$ mit $\overline{E} = \{ (w, v) : (v, w) \in E \}$ verglichen werden. Wenn $\overline{G}$ nach der Anwendung der Transformationsregel 6) zyklusfrei ist, ist es auch G' nach der Anwendung der Transformationsregel 5).
Die Transformationsregel 6) ist ein Spezialfall der Transformationsregel 7) mit indeg(y) = 1. Aus diesem Grund können bei der Anwendung der Transformationsregel 6) nur Zyklen entstehen, wenn dies bei der Transformationsregel 7) möglich ist.
Bei der Anwendung der Transformationsregel 7) auf die Knoten x, y und z mit $(z, x) \in E$ und $(z, y) \in E$ gelte

$$E' \setminus E = \{ (v_1, x), ..., (v_a, x), (x, w_1), ..., (x, w_b) \}.$$

Es wird angenommen, daß G' einen Zyklus Z enthält. Sollte G' mehrere Zyklen enthalten, so wähle den Zyklus Z mit folgenden Eigenschaft:
(α) Es gibt in G' keinen Zyklus mit geringerer Länge, d.h. über weniger Knoten, als Z.
Fall 1: Der Zyklus Z enthalte die Kanten (v_i, x) und (x, w_j) für ein $i \in [1; a]$ und ein $j \in [1; b]$.
Demzufolge existiert der Pfad $w_j \rightarrow v_i$ in G'. Dieser Pfad enthält außer dem Start- und Endknoten keinen der Knoten x, v_1, ..., v_a, w_1, ..., w_b, da sonst die Bedingung (α) verletzt wäre. Aus diesem Grund existiert der Pfad $w_j \rightarrow v_i$ bereits in G. Da $(v_i, y) \in E$ und $(y, w_j) \in E$ gilt, hatte G bereits einen Zyklus.
Fall 2: Der Zyklus Z enthalte die Kante (v_i, x) für ein $i \in [1; a]$, jedoch keine der Kanten (x, w_j) für alle $j \in [1; b]$.
Demzufolge existiert der Pfad $x \rightarrow v_i$ in G'. Dieser Pfad enthält außer dem Start- und Endknoten keinen der Knoten x, v_1, ..., v_a, w_1, ..., w_b, da sonst die Bedingung (α) verletzt wäre. Aus diesem Grund existiert der Pfad $x \rightarrow v_i$ bereits in G. Da $(v_i, y) \in E$ und $(z, x) \in E$ gilt, existiert in G der Pfad $z \rightarrow x \rightarrow v_i \rightarrow y$. Demzufolge enthält G die transitive Kante $(z, y) \in E$.
Fall 3: Der Zyklus Z enthalte die Kante (x, w_j) für ein $j \in [1; b]$, jedoch keine der Kanten (v_i, x) für alle $i \in [1; a]$.
Demzufolge existiert der Pfad $w_j \rightarrow x$ in G'. Dieser Pfad enthält außer dem Start- und Endknoten keinen der Knoten x, v_1, ..., v_a, w_1, ..., w_b, da sonst die Bedingung (α) verletzt wäre. Aus diesem Grund existiert der Pfad $w_j \rightarrow x$ bereits in G.
Fall 3.1: Der in G liegende Pfad $w_j \rightarrow x$ enthalte den Knoten z.
Demzufolge existiert in G der Pfad $w_j \rightarrow z$. Da $(z, y) \in E$ und $(y, w_j) \in E$ gilt, hatte G bereits einen Zyklus.

Fall 3.2: Der in G liegende Pfad $w_j \rightarrow x$ enthalte den Knoten z nicht.

Da $(z, y) \in E$ und $(y, w_j) \in E$ gilt, existiert in G der Pfad $z \rightarrow y \rightarrow w_j \rightarrow x$. Demzufolge enthält G die transitive Kante $(z, x) \in E$.

Fall 4: Der Zyklus Z enthält keine der in E \ E' liegenden Kanten.

In diesem Fall führt der Zyklus nur über Kanten, die bereits in E enthalten sind. Demzufolge enthält G einen Zyklus.

■

Lemma 31: *Der Algorithmus „kGzuB" gelangt bei jedem Modulabhängigkeitsgraphen G durch die wiederholte Anwendung der Transformationsregeln 1) bis 7) und dem Löschen aller transitiver Kanten zu einem Graphen mit leerer Kantenmenge.*

Beweis: Aus Lemma 30 folgt, daß nach der Anwendung einer Transformationsregel auf einen Modulabhängigkeitsgraphen wiederum ein zyklusfreier Graph entsteht. Da bei der Anwendung einer Transformationsregel immer mindestens ein Knoten, nämlich y, gelöscht wird, gelangt man durch die wiederholte Anwendung der Transformationsregeln spätestens wenn der Graph nur noch einen Knoten enthält, zu einem Graphen ohne Kanten. Demzufolge ist es ausreichend zu zeigen, daß auf alle Modulabhängigkeitsgraphen G = (V, E) ohne transitive Kanten mit $E \neq \emptyset$ mindestens eine Transformationsregel anwendbar ist.

Es sei $(a, b) \in E$ eine beliebige Kante im Modulabhängigkeitsgraphen G.

Fall 1: Es sei indeg(b) = 1.

Fall 1.1: Es sei outdeg(a) = 1.

In diesem Fall ist die Transformationsregel 1) mit x = a und y = b anwendbar.

Fall 1.2: Es sei outdeg(a) $\geqslant$ 2.

Da outdeg(a) $\geqslant$ 2 ist und G keine transitiven Kanten enthält, existiert ein Knoten $c \in V$ mit $c \neq b$ und $(a, c) \in E$.

Fall 1.2.1: Es sei outdeg(b) = 0.

In diesem Fall ist die Transformationsregel 2) mit x = c, y = b und z = a anwendbar.

Fall 1.2.2: Es sei outdeg(b) = 1.

Da outdeg(b) = 1 ist, existiert ein Knoten $d \in V$ mit $(b, d) \in E$.

Fall 1.2.2.1: Es existiere ein Pfad $a \rightarrow d$, welcher nicht über b führt.

Der nicht über b führende Pfad $a \rightarrow d$ enthält mindestens einen Knoten $e \in V$ mit $e \neq a$ und $e \neq d$. Würde e nicht existieren, so gäbe es aufgrund des Pfad $a \rightarrow b \rightarrow d$ die transitive Kante (a, d). Somit ist die Transformationsregel 4a) mit w = a, x = e, y = b und z = d anwendbar.

Fall 1.2.2.2: Alle Pfade $a \rightarrow d$ führen über den Knoten b.

In diesem Fall ist die Transformationsregel 4b) mit w = a, x = c, y = b und z = d anwendbar.

Fall 1.2.3: Es sei outdeg(b) $\geqslant$ 2.

In diesem Fall ist die Transformationsregel 6) mit x = c, y = b und z = a anwendbar.

Fall 2: Es sei indeg(b) $\geqslant$ 2.

Da indeg(b) $\geqslant$ 2 ist und G keine transitiven Kanten enthält, existiert ein Knoten $c \in V$ mit $c \neq a$ und $(c, b) \in E$.

Fall 2.1: Es sei outdeg(a) = 1.

Fall 2.1.1: Es sei indeg(a) = 0.

In diesem Fall ist die Transformationsregel 3) mit x = c, y = a und z = b anwendbar.

Fall 2.1.2: Es sei indeg(a) $\geqslant$ 1.

In diesem Fall ist die Transformationsregel 5) mit x = c, y = a und z = b anwendbar.

Fall 2.2: Es sei outdeg(a) $\geqslant$ 2.
Da outdeg(a) $\geqslant$ 2 ist und G keine transitiven Kanten enthält, existiert ein Knoten d $\in$ V mit d $\neq$ b und (a, d) $\in$ E. Somit ist in diesem Fall die Transformationsregel 7) mit x = b, y = d und z = a anwendbar.

■

Lemma 32: *Es sei G = (V, E) ein Modulabhängigkeitsgraph ohne transitive Kanten. Jedem Knoten x $\in$ V sei der Konstruktionsterm k(x) zugeordnet. Der Graph G' = (V', E') gehe durch die Anwendung einer der Transformationsregeln 1) bis 7) aus G hervor. In G' sei jedem Knoten y $\in$ V' der Konstruktionsterm k'(y) zugeordnet. Der Knoten a sei im Konstruktionsterm k(b) mit b $\in$ V' enthalten. Dann enthält der Konstruktionsterm k'(b) ebenfalls den Knoten a.*

Beweis:
Fall 1: Es gelte k(b) = k'(b).
In diesem Fall folgt die Behauptung sofort.
Fall 2: Es gelte k(b) $\neq$ k'(b).
Es sei v $\in$ V \ V'. Da bei der Anwendung einer der Transformationsregeln 1) bis 7) immer genau ein Knoten gelöscht wird, ist v eindeutig gegeben. Außerdem ist k(b) $\neq$ k'(b) und b $\in$ V'. Demzufolge gilt bei der Anwendung einer Transformationsregel x = b und y = v.
Fall 2.1: Es wurde die Transformationsregel 1) angewandt.
In diesem Fall gilt k'(b) = k(b) + k(v). Somit ist der Knoten a auch in k'(b) enthalten.
Fall 2.2: Es wurde eine der Transformationsregeln 2) bis 7) angewandt.
In diesem Fall gilt k'(b) = k(b) $\|$ k(v). Somit ist der Knoten a auch in k'(b) enthalten.

■

Folgerung aus Lemma 32: *Da beim Start des Algorithmus „kGzuB" für alle Knoten x des übergebenen Modulabhängigkeitsgraphen k(x) = x gesetzt wird, enthält der Konstruktionsterm des Knotens x auch nach der Anwendung einer beliebigen Anzahl an Transformationsregeln den Knoten x.*

Lemma 33: *Es sei G = (V, E) ein Modulabhängigkeitsgraph und a$\rightarrow$b mit a, b $\in$ V ein beliebiger Pfad in G. Jedem Knoten x $\in$ V sei der Konstruktionsterm k(x) zugeordnet. Der Graph G' = (V', E') gehe durch die Anwendung einer der Transformationsregeln 1) bis 7) aus G hervor. In G' sei jedem Knoten y $\in$ V' der Konstruktionsterm k'(y) zugeordnet.*

- *(α) Es gelte a, b $\in$ V'. Dann gibt es in G' einen Pfad von a nach b.*
- *(β) Es gelte a $\notin$ V'. Dann gibt es einen Knoten c $\in$ V', in dessen Konstruktionsterm k'(c) der Knoten a enthalten ist. Außerdem existiert ein Pfad von c nach b in G'.*
- *(γ) Es gelte b $\notin$ V'. Dann gibt einen Knoten c $\in$ V', in dessen Konstruktionsterm k'(c) der Knoten b enthalten ist. Außerdem existiert entweder ein Pfad von a nach c in G' oder es ist a = c. Sollte a = c sein, so gilt zusätzlich k'(c) = k(a) + k(b).*

Beweis von Lemma 33(α):
Es sei v $\in$ V \ V'. Da bei der Anwendung einer der Transformationsregeln 1) bis 7) immer genau ein Knoten gelöscht wird, ist v eindeutig gegeben.

Fall 1: Der in G liegende Pfad $a \rightarrow b$ enthält v nicht.
Da bei der Anwendung einer Transformationsregel nur alle zu v adjazenten Kanten gelöscht werden und der Pfad $a \rightarrow b$ in G nicht über v führt, existiert in G' der gleiche Pfad von a nach b.
Fall 2: Der in G liegende Pfad $a \rightarrow b$ enthält den Knoten v.
Da $a, b \in V'$ ist, gilt $v \neq a$ und $v \neq b$. Demzufolge existieren die Knoten c und d mit:
(a) Es ist $a = c$ oder in G existiert der Pfad $a \rightarrow c$.
(b) Es gilt $(c, v) \in E$ und $(v, d) \in E$.
(c) Es ist $d = b$ oder in G existiert der Pfad $d \rightarrow b$.
Da G zyklusfrei ist, enthält weder der Pfad $a \rightarrow c$ noch der Pfad $d \rightarrow b$ den Knoten v. Aus diesem Grund existieren nach Fall 1 die Pfade $a \rightarrow c$ und $d \rightarrow b$ auch in G'. Es ist also ausreichend zu zeigen, daß in G' der Pfad $c \rightarrow d$ existiert.
Fall 2.1: Beim Übergang von G zu G' wurde die Transformationsregel 1) angewandt.
Man erkennt sofort, daß die Transformationsregel 1) auf die Knoten $x = c$ und $y = v$ angewandt wurde. Wegen $(v, d) \in E$ folgt, daß $(c, d) \in E'$ ist. Somit existiert in G' der Pfad $c \rightarrow d$.
Fall 2.2: Beim Übergang von G zu G' wurde die Transformationsregel 2) oder 3) angewandt.
Dieser Fall kann nicht eintreten, weil für den Knoten v in G $indeg(v) \geqslant 1$ und $outdeg(v) \geqslant 1$ gilt.
Fall 2.3: Beim Übergang von G zu G' wurde die Transformationsregel 4a) angewandt.
Demzufolge existiert ein Knoten $e \in V$, so daß die Transformationsregel 4a) auf die Knoten $w = c$, $x = e$, $y = v$ und $z = d$ angewandt wurde. Aus diesem Grund existieren in G die Pfade $c \rightarrow e$ und $e \rightarrow d$. Beide Pfade enthalten den Knoten v nicht, so daß nach Fall 1 die Pfade $c \rightarrow e$ und $e \rightarrow d$ auch in G' existieren. Somit existiert in G' der Pfad $c \rightarrow e \rightarrow d$.
Fall 2.4: Beim Übergang von G zu G' wurde die Transformationsregel 4b) angewandt.
Demzufolge existiert ein Knoten $e \in V$, so daß die Transformationsregel 4b) auf die Knoten $w = c$, $x = e$, $y = v$ und $z = d$ angewandt wurde. Aus diesem Grund gilt $(c, e) \in E$ und somit auch $(c, e) \in E'$. Wegen $(v, d) \in E$ folgt $(e, d) \in E'$. Es existiert also ein Pfad von c nach d in G'.
Fall 2.5: Beim Übergang von G zu G' wurde die Transformationsregel 5) angewandt.
Demzufolge existiert ein Knoten $e \in V$, so daß die Transformationsregel 5) auf die Knoten $x = e$, $y = v$ und $z = d$ angewandt wurde. Aus diesem Grund gilt $(e, d) \in E$ und somit auch $(e, d) \in E'$. Wegen $(c, v) \in E$ folgt $(c, e) \in E'$. Es existiert also ein Pfad von c nach d in G'.
Fall 2.6: Beim Übergang von G zu G' wurde die Transformationsregel 6) angewandt.
Demzufolge existiert ein Knoten $e \in V$, so daß die Transformationsregel 6) auf die Knoten $x = e$, $y = v$ und $z = c$ angewandt wurde. Aus diesem Grund gilt $(c, e) \in E$ und somit auch $(c, e) \in E'$. Wegen $(v, d) \in E$ folgt $(e, d) \in E'$. Es existiert also ein Pfad von c nach d in G'.
Fall 2.7: Beim Übergang von G zu G' wurde die Transformationsregel 7) angewandt.
Demzufolge existieren zwei Knoten $e, f \in V$, so daß die Transformationsregel 7) auf die Knoten $x = e$, $y = v$ und $z = f$ angewandt wurde. Wegen $(c, v) \in E$ folgt $(c, e) \in E'$ und wegen $(v, d) \in E$ folgt $(e, d) \in E'$. Es existiert also ein Pfad von c nach d in G'.

Beweis von Lemma 33(β):
Da $a \in V \setminus V'$ ist, gilt bei der Anwendung einer Transformationsregel $y = a$. Wegen $outdeg(a) \geqslant 1$ kann die Transformationsregel 2) nicht angewandt werden.
Die Folgerung aus Lemma 32 besagt, daß der Knoten a im Konstruktionsterm k(a) enthalten ist. Es sei $c \in V$ der Knoten, so daß bei der Anwendung einer Transformationsregel $x = c$ ist.

Wegen k'(c) = k(c) + k(a) bzw. k'(c) = k(c) ‖ k(a) folgt, daß der Knoten a im Konstruktionsterm k'(c) enthalten ist.
Der Pfad a→b in G enthalte die Kante (a, d) ∈ E.
Fall 1: Es sei d = b.
Bei der Anwendung einer der Transformationsregeln 1) und 3) bis 7) folgt aus (a, d) ∈ E, daß (c, d) ∈ E' ist. Demzufolge existiert in G' die Kante (c, b) ∈ E'.
Fall 2: Es sei d ≠ b.
Da G zyklusfrei ist, führt der Pfad d→b in G nicht über a. Nach Punkt (α) Fall 1 existiert der Pfad d→b auch in G'. Bei der Anwendung einer der Transformationsregeln 1) und 3) bis 7) folgt aus (a, d) ∈ E, daß (c, d) ∈ E' ist. Demzufolge existiert in G' der Pfad c→d→b.

Beweis von Lemma 33(γ):
Da b ∈ V \ V' ist, gilt bei der Anwendung einer Transformationsregel y = b. Wegen indeg(b) ⩾ 1 kann die Transformationsregel 3) nicht angewandt werden.
Die Folgerung aus Lemma 32 besagt, daß der Knoten b im Konstruktionsterm k(b) enthalten ist. Es sei c ∈ V der Knoten, so daß bei der Anwendung einer Transformationsregel x = c ist. Wegen k'(c) = k(c) + k(b) bzw. k'(c) = k(c) ‖ k(b) folgt, daß der Knoten b im Konstruktionsterm k'(c) enthalten ist.
Der Pfad a→b in G enthalte die Kante (d, b) ∈ E.
Fall 1: Es sei a = d.
Fall 1.1: Es wurde die Transformationsregel 1) angewandt.
Da die Kante (a, b) ∈ E ist, gilt bei der Anwendung der Transformationsregel 1) x = c = a und somit k'(c) = k(a) + k(b).
Fall 1.2: Es wurde eine der Transformationsregeln 2) und 4a) bis 7) angewandt.
Bei der Anwendung einer der Transformationsregeln folgt aus (d, b) ∈ E, daß (d, c) ∈ E' ist. Wegen a = d existiert in G' die Kante (a, c) ∈ E'.
Fall 2: Es sei a ≠ d.
Da G zyklusfrei ist, führt der Pfad a→d in G nicht über b. Nach Punkt (α) Fall 1 existiert der Pfad a→d auch in G'.
Fall 2.1: Es wurde die Transformationsregel 1) angewandt.
Bei der Anwendung der Transformationsregel 1) gilt x = c = d. Demzufolge existiert in G' der Pfad a→c.
Fall 2.2: Es wurde eine der Transformationsregeln 2) und 4a) bis 7) angewandt.
Bei der Anwendung einer der Transformationsregeln folgt aus (d, b) ∈ E, daß (d, c) ∈ E' ist. Demzufolge existiert in G' der Pfad a→d→c.

■

Folgerungen aus Lemma 33:

(1) Gilt im Fall (γ) die Beziehung a = c, so folgt mit Hilfe von Lemma 15 aus k'(c) = k(a) + k(b), daß es im durch k'(c) repräsentierten serien - parallelen Graphen einen Pfad von a nach b gibt.

(2) Der Graph G' = (V', E') gehe aus der wiederholten Anwendung der Transformationsregeln 1) bis 7) aus dem Modulabhängigkeitsgraph G = (V, E) hervor. In G' sei jedem Knoten x ∈ V' der Konstruktionsterm k'(x) zugeordnet. Es sei a→b mit a, b ∈ V ein Pfad in G. Dann gilt genau eine der folgenden Aussagen:

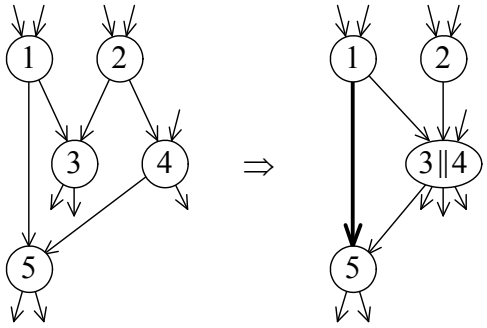
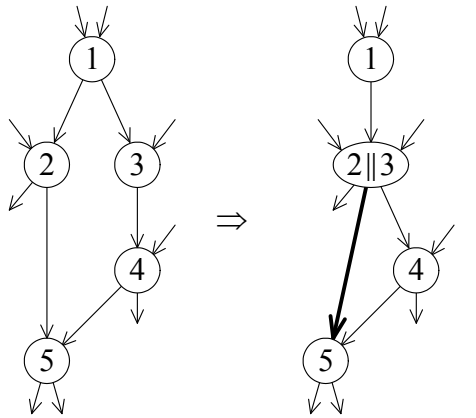
(α) Es existieren zwei Knoten c, d ∈ V' mit den folgenden Eigenschaften: Der Konstruktionsterm k'(c) enthält den Knoten a. Der Konstruktionsterm k'(d) enthält den Knoten b. Es gibt in G' den Pfad c→d.

(β) Es existiert ein Knoten c ∈ V' mit den folgenden Eigenschaften: Der Konstruktionsterm k'(c) enthält die Knoten a und b. In dem durch k'(c) repräsentierten serien - parallelen Graphen existiert ein Pfad von a nach b.

Nachdem der Algorithmus „kGzuB" den Modulabhängigkeitsgraphen G = (V, E) mit Hilfe der Transformationsregeln 1) bis 7) in den Graphen G' = (V', E') mit E' = ∅ und den Konstruktionstermen k'(x) für alle x ∈ V' überführt hat, erstellt „kGzuB" den Konstruktionsterm

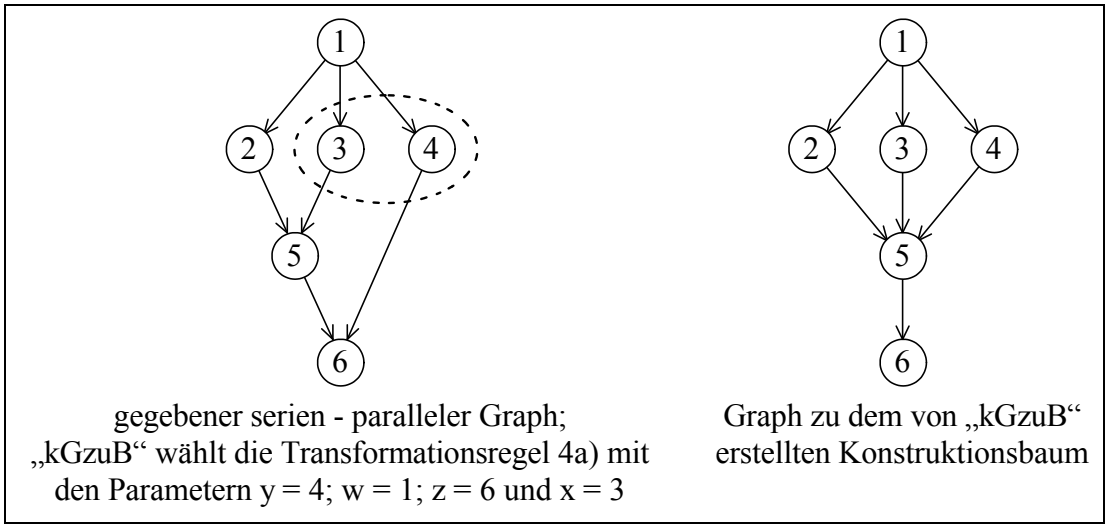
$$K = \prod_{x \in V'} k'(x).$$

Die Folgerung (2) aus dem Lemma 33 garantiert, daß es für jeden Pfad a→b in G auch in dem durch K repräsentierten serien - parallelen Graphen einen Pfad von a nach b gibt.
Nach der Berechnung des Konstruktionsterms K mittels „kGzuB" kann der Algorithmus „2M1" zur Berechnung eines Schedules für den durch K repräsentierten serien - parallelen Graphen verwendet werden. Die Nacheinanderausführung von „kGzuB" und „2M1" wird nachfolgend als Algorithmus „ReSP" bezeichnet.
Zur Laufzeitanalyse von „kGzuB" sind die folgenden Überlegungen notwendig: Da in jedem Transformationsschritt die Knotenanzahl um Eins verringert wird, führt der Algorithmus „kGzuB" maximal |V| - 1 Transformationen durch. In allen Unterroutinen werden maximal O(|V|) Kanten gelöscht bzw. eingefügt. Bei der Entscheidung, ob die Transformationsregel 4a) oder 4b) angewandt werden kann, sucht der Algorithmus zusätzlich einen Pfad zwischen zwei Knoten. Dies kann in O(|V| + |E|) Schritten erfolgen. Außerdem sind bei den Transformationsregeln 4b) bis 7) noch transitive Kanten zu löschen. Bei der Anwendung der Transformationsregel 7) auf die Knoten x, y und z gibt es exakt zwei Stellen, an welchen transitive Kanten entstehen können:

a) 	Eine neu eingefügte Kante ist Teil eines Pfades, der eine transitive Kante hervorruft. Demzufolge ist der Knoten x in diesem Pfad enthalten. Der Test, ob eine Kante (v, w) und die Pfade v→x sowie x→w existieren, ist mit Hilfe zweier DFS - Läufe von x aus (in und entgegen der Kantenrichtung) in O(|V| + |E|) möglich.
b) 	Eine neu eingefügte Kante ist selbst transitiv. Diese Kante beginnt oder endet im Knoten x. Man muß also testen, ob es zwischen einem Vorgänger v bzw. Nachfolger w des Knotens x und dem Knoten x einen Pfad, welcher über mindestens einen von x, v und w verschiedenen Knoten führt, gibt. Dieser Test ist ebenfalls mit zwei DFS - Läufen realisierbar und benötigt somit O(|V| + |E|) Schritte.

Da die Transformationsregeln 4b) bis 6) Spezialfälle der Transformationsregel 7) sind, kann auch bei diesen Transformationsregeln die Löschung aller transitiver Kanten in O(|V| + |E|) vorgenommen werden kann.
Insgesamt benötigt der Algorithmus „kGzuB“ maximal O(|V| · (|V| + |E|)) Zeiteinheiten. Zusammen mit der anschließenden Optimierung des Konstruktionsbaumes mittels „2M1“ resultiert eine Laufzeit von O(|V| · (|V| + |E| + N)) = O(M · (M + |E| + N)) für „ReSP“.

Wie beim Algorithmus „WaSP“ sind für die Ergebnisgüte von „ReSP“ nur die im Lemma 25 und Lemma 26 gemachten Angaben möglich. Wird dem Algorithmus „kGzuB“ ein serien - paralleler Graph G übergeben, so ist nicht garantiert, daß der von „kGzuB“ erstellte Konstruktionsbaum K ein Konstruktionsbaum des Graphen G ist. Es ist nur garantiert, daß es für jeden Pfad in G auch einen Pfad in dem durch K repräsentierten serien - parallelen Graphen gibt. Ein Beispiel ist in der Abbildung 44 angegeben.



gegebener serien - paralleler Graph;
„kGzuB“ wählt die Transformationsregel 4a) mit den Parametern y = 4; w = 1; z = 6 und x = 3

Graph zu dem von „kGzuB“ erstellten Konstruktionsbaum

Abbildung 44 - Beispiel, bei dem „kGzuB“ einen serien - parallelen Graphen nicht richtig auflöst.

4.1.4 Kritische Pfade

Die in diesem Abschnitt angegebenen Algorithmen „KP+“ und „KPMF“ lehnen sich stark an die z.B. in [35] zum DAG - Shop - Scheduling vorgestellten Verfahren an. Sie basieren auf der bevorzugten Behandlung der Module, die sich auf dem kritischen Pfad befinden.

Definition 14: *Ein Pfad* $(x_1, x_2, ..., x_k)$ *des Modulabhängigkeitsgraphen G = (V, E) heißt kritisch, wenn* $\sum_{i=1}^{k} T(x_i)$ *maximal bzgl. aller Pfade in G ist. Dabei bezeichne T(x) die Laufzeit des Moduls x bei der Ausführung mit B(x) Prozessoren.*

Da sich beide Verfahren am DAG - Shop - Scheduling orientieren, setzen sie die Prozessoranzahl aller Module am Anfang auf Eins. Im Verlauf des Algorithmus wird die Prozessoranzahl der Module variiert, so daß nacheinander mehrere Schedules berechnet werden. Bei der Berechnung eines Schedules bleibt die Prozessoranzahl der Module aber fest.
Zur Berechnung eines Schedules ordnen die Algorithmen „KP+“ und „KPMF“ jedem Modul x eine Priorität

$$\text{prio}(x) = T(x) + \max(\{0\} \cup \{ \text{prio}(y) : y \in V \text{ mit } (x, y) \in E \})$$

Der gewünschte Gütefaktor wird mit g bezeichnet.

1) $h = \infty$
2) $\forall x \in [1; M]: B(x) = 1$
3) $\forall x \in [1; M]: f(x) = 0;\ eing(x) = indeg(x);\ prio(x) = -1$
4) ***für*** $x = 1$ ***bis*** M
5) ***ist*** $prio(x) = -1$, ***so starte*** *Unterroutine berechne_prio(x) in Zeile 33)*
6) $\forall i \in [1; N]: E(i) = 0$
7) $X = \{ x : x \in V \text{ mit } eing(x) = 0 \};\ Z = \emptyset$
8) ***solange*** $X \neq \emptyset$
9) *wähle ein beliebiges* $x \in X$ *mit* $prio(x) = max\{ prio(y) : y \in X \}$
10) $X = X \setminus \{x\}$
11) $i = \min\limits_{j=1}^{N - B(x) + 1} \left\{ j : \max\limits_{k=j}^{j + B(x) - 1} \{ E(k) \} = \min\limits_{m=1}^{N + 1 - B(x)} \left\{ \max\limits_{k=m}^{m + B(x) - 1} \{ E(k) \} \right\} \right\}$
12) $P(x) = i;\ S(x) = \max \left\{ \max\limits_{k=i}^{i + B(x) - 1} \{ E(k) \};\ f(x) \right\}$
13) $\forall j \in [i; i + B(x) - 1]: E(j) = S(x) + T(x)$
14) ***für alle*** $y \in V$ *mit* $(x, y) \in E$
15) $eing(y) = eing(y) - 1$
16) ***ist*** $f(y) < E(i)$, ***so*** $f(y) = E(i)$
17) ***ist*** $eing(y) = 0$, ***so*** $Z = Z \cup \{y\}$
18) ***für alle*** $x \in Z$ *mit* $f(x) \leqslant \min\limits_{i=1}^{N} \{ E(i) \}$
19) $Z = Z \setminus \{x\};\ X = X \cup \{x\}$
20) ***ist*** $X = \emptyset$, ***so tue***
21) $X = \{ x : x \in Z \text{ mit } f(x) = min\{ f(y) : y \in Z \} \}$
22) $Z = Z \setminus X$
23) $H = \max\limits_{i=1}^{N} \{ E(i) \}$
24) ***ist*** $\frac{H \cdot N}{M} \leqslant \sum\limits_{x=1} T(x, 1)$, ***so*** *gib den aktuellen Schedule zurück;* ***Ende***
25) ***ist*** $H < h$, ***so*** *speichere den aktuellen Schedule;* $h = H$
26) $flag = 0;\ X = \{ x : x \in V \text{ mit } indeg(x) = 0 \}$
27) ***solange*** $X \neq \emptyset$
28) *wähle* $z \in \{ x : x \in X \text{ und } prio(x) = max\{ prio(y) : y \in X \} \}$
29) ***ist*** $B(z) \neq N$, ***so*** $B(z) = B(z) + 1;\ flag = 1$
30) $X = \{ x : x \in V \text{ mit } (z, x) \in E \}$
31) ***ist*** $flag = 0$, ***so*** *stelle den zuletzt gespeicherten Schedule wieder her und gib ihn zurück;* ***Ende***
32) *weiter in Zeile 3)*

Abbildung 45 - Algorithmus „KP+“.

```
33) Unterroutine berechne_prio( x )
34) prio(x) = 0
35) für alle y ∈ V mit (x, y) ∈ E
36)       ist prio(y) = -1, so starte Unterroutine berechne_prio( y ) in Zeile 33)
37)       ist prio(y) > prio(x), so prio(x) = prio(y)
38) prio(x) = prio(x) + T(x, B(x))
39) Rücksprung aus der Unterroutine
```

Abbildung 46 - Algorithmus „KP+“ - Unterroutine „berechne_prio“.

zu. Man kann erkennen, daß nach dem Start von x noch mindestens prio(x) Zeiteinheiten vergehen müssen, bevor alle Module abgearbeitet sind.
Beide Verfahren positionieren die Module nacheinander auf den einzelnen Prozessoren. Mit E(i) wird die aktuelle Endzeit von Prozessor i bezeichnet (siehe Definition 5). Außerdem bedeutet S(x) = ∞, daß das Modul x noch nicht gestartet wurde. Ansonsten enthält S(x) den Startzeitpunkt des Moduls x.
Zur Auswahl des Moduls, welches als nächstes positioniert wird, bestimmen die Algorithmen „KP+“ und „KPMF“ die Menge

$$X = \left\{ x : x \in V \text{ mit } S(x) = \infty \text{ und } \forall y \in V \text{ mit } (y, x) \in E \text{ gilt } S(y) + T(y) \leqslant \min_{i=1}^{N} \{ E(i) \} \right\}$$

der wartenden Module. Da zum Zeitpunkt

$$t = \min_{i=1}^{N} \{ E(i) \}$$

alle Vorgänger der in X enthaltenen Module beendet sind und alle Prozessoren bis zum Zeitpunkt t Module ausführen, können die in X enthaltenen Module sofort auf den nächsten freien Prozessoren gestartet werden. Sollte die Menge X leer sein, so wählen „KP+“ bzw. „KPMF“ für X alle Module, die zum frühesten Zeitpunkt gestartet werden können. Es wird also

$$X = \{ x : x \in V \text{ mit } S(x) = \infty \text{ und } \nexists y \in V \text{ mit } S(y) = \infty \text{ und } f(y) < f(x) \}$$

gesetzt. Dabei liefert

$$f(x) = \begin{cases} \max\{ S(y) + T(y) : y \in V \text{ mit } (y, x) \in E \} & : x \text{ ist keine Quelle in G} \\ 0 & : \text{sonst} \end{cases}$$

den ersten Zeitpunkt, zu dem ein Modul x gestartet werden kann.
Anschließend suchen die beiden Algorithmen in X ein Modul x mit maximaler Priorität. Dieses Modul wird nun so zeitig wie möglich ausgeführt. Erzielt der Algorithmus bei verschiedenen Prozessoren bzw. Prozessorgruppen die gleiche Startzeit, so wählt er die mit den kleinsten Prozessornummern.
Nachdem für alle Module eine Startzeit und ein Prozessor oder eine Prozessorgruppe bestimmt wurden, kann die Laufzeit H des berechneten Schedules angegeben werden:

$$H = \max\{ S(x) + T(x) : \forall \text{ Module } x \}.$$

Außerdem wird der garantiert erreichte Gütefaktor

$$a = \frac{H \cdot N}{\sum_{x=1}^{M} T(x, 1)}$$

mit Hilfe der Flächenbedingung ermittelt. Ist a kleiner oder gleich einer vorgegebenen Schranke, so bricht der Algorithmus ab. Anderenfalls werden die Prozessoranzahlen aller

Module auf dem kritischen Pfad erhöht. An dieser Stelle unterscheidet sich das Vorgehen der beiden Verfahren. Bei „KP+“ wird die Prozessoranzahl aller Module auf dem kritischen Pfad exakt um Eins vergrößert. Der Algorithmus „KPMF“ erhöht bei jedem Modul x auf dem kritischen Pfad die Prozessoranzahl soweit, daß die Flächenvergrößerung von x minimal ist:

$$B_{neu}(x) = \min_{i = B(x)+1}^{N} \left\{ i : i \cdot T(x, i) = \min_{j = B(x)+1}^{N} \{ j \cdot T(x, j) \} \right\}.$$

Ist die Parallelisierung eines Moduls auf dem kritischen Pfad nicht mehr möglich, so wird es unverändert in den nächsten Positionierungsschritt übernommen. Erst wenn alle Module auf dem kritischen Pfad die Prozessoranzahl N haben, bricht der Algorithmus ab und gibt den besten erstellten Schedule zurück.
Man beachte, daß sich nach jeder Parallelisierung die Prioritäten und der kritische Pfad ändern können.
Der Pseudocode des Algorithmus „KP+“ ist in den Abbildungen 45 und 46 dargestellt. Um im Pseudocode aus dem Algorithmus „KP+“ das Verfahren „KPMF“ zu machen, muß man die Zeile 29) in

$$29) \qquad \textbf{\textit{ist}}\ B(z) \neq N,\ \textbf{\textit{so}}\ B(z) = \min_{i = B(z)+1}^{N} \left\{ i : i \cdot T(z, i) = \min_{j = B(z)+1}^{N} \{ j \cdot T(z, j) \} \right\};\ flag = 1$$

umwandeln.

Im schlechtesten Fall wird die Schleife ab Zeile 3) $(M \cdot N)$-mal durchlaufen. Implementiert man die Mengen X und Z als Heap, so benötigt eine Positionierung aller Module $O(|E| + M \cdot (\log(M) + N))$ Schritte. Man gelangt also zu einer Laufzeit für „KP+“ und „KPMF“ von $O\left(M \cdot N \cdot |E| + M^2 \cdot N \cdot (\log(M) + N)\right)$. In vielen Fällen müssen sicherlich nur wenige Module mit mehr als einem Prozessor ausgeführt werden, um einen Schedule mit einem Gütefaktor kleiner oder gleich dem gegebenen Gütefaktor g zu erzielen. In diesem Fall brechen die Algorithmen „KP+“ und „KPMF“ schon nach wenigen Positionierungen aller Module in Zeile 24) ab.

4.1.5 Praktische Tests

Auch die in diesem Abschnitt vorgestellten Algorithmen wurden mit Hilfe von zufällig erzeugten Beispielen getestet (siehe Beispiel 5 im Anhang 9.24). Die Modullaufzeiten von Beispiel 5 basieren auf denen von Beispiel 3. Der Modulabhängigkeitsgraph wird zufällig erzeugt, indem für je zwei Knoten i und j mit j > i entschieden wird, ob eine Kante von i nach j führt.
In den Abbildungen 47 und 48 steht die linke Säule in jedem Diagramm für die Ergebnisse des Algorithmus „WaSP“. Danach folgen zwei Säulen für den Algorithmus „KP+“ mit dem gewünschten Gütefaktor Eins bzw. Zwei. Im Anschluß daran sind die Ergebnisse von „KPMF“ mit dem gewünschten Gütefaktor Eins bzw. Zwei dargestellt. Die sechste Säule von links repräsentiert die Resultate des Verfahrens „ReSP“. Danach folgen noch die Säulen für die Algorithmen „SchHT“ und „SchT“. Die Zahlenwerte der Messungen sind auf der CD im Verzeichnis /res/bsp5/ einsehbar. Bei den mit einem Stern gekennzeichneten Säulen war die Laufzeit so hoch, daß anstatt der üblichen 100 Testläufe nur 10 Testläufe durchgeführt wurden.
Die besten Schedules liefert das Verfahren „ReSP“. Auch ist die Laufzeit von „ReSP“ relativ unabhängig von der Gesamtprozessoranzahl N. Allerdings ist „ReSP“, im Vergleich zu den anderen Algorithmen, bei wenigen Prozessoren und vielen Modulen wesentlich langsamer.

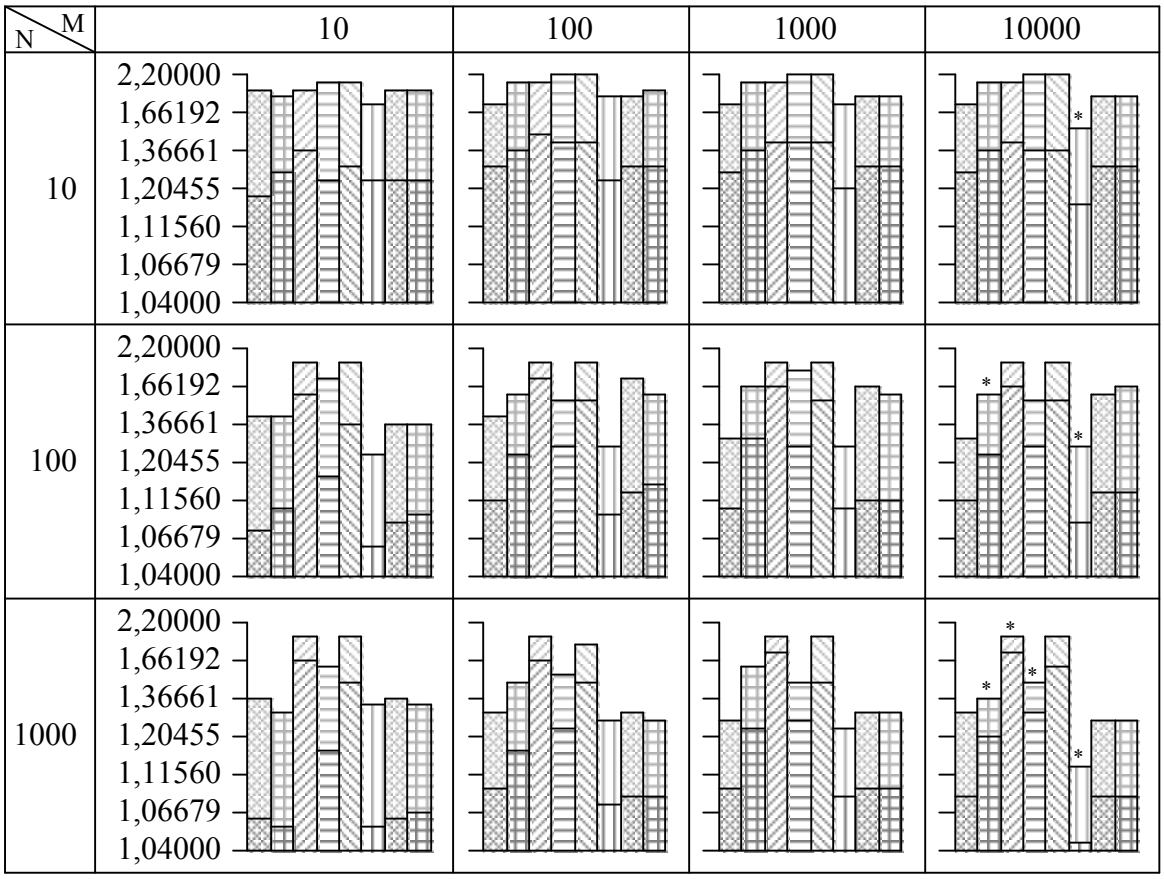


Abbildung 47 - schlechtester und durchschnittlicher Gütefaktor (Beispiel 5).

Man sieht sehr schön, daß die von „WaSP“ erzeugten Schedules nur minimal schlechter sind als die von „ReSP“. Dafür ist „WaSP“ aber auch wesentlich schneller als „ReSP“.
Sind wesentlich weniger Prozessoren als Module vorhanden, so ist die Verwendung der Verfahren „SchT“ und „SchHT“ ganz sinnvoll. Hier liefert der nur minimal langsamere Algorithmus „SchHT“ im Durchschnitt die besseren Schedules. Es wurden aber auch Beispiele erzeugt (z.B. $N = 10$, $M = 10$), bei denen der schlechteste Schedule von „SchHT“ schlechter ist als der schlechteste Schedule von „SchT“. Stehen viele Prozessoren und nur wenige Module zur Verfügung, so müssen beide Algorithmen die Prozessoranzahl vieler Module variieren und mehrere Positionierungsversuche durchführen. Dadurch erhöht sich die Laufzeit nicht unwesentlich (siehe Algorithmus „P1+MF“).
Falls die Modulanzahl wesentlich größter ist als die Prozessoranzahl, so werden auch im optimalen Schedule die meisten Module mit nur einem Prozessor ausgeführt. Da die Algorithmen „KP+“ und „KPMF“ am Anfang jedem Modul die Prozessoranzahl Eins zuordnen, erzielen sie bereits bei der ersten Positionierung einen zufriedenstellenden Schedule. Dieser Schedule wird unabhängig vom gewünschten Gütefaktor erstellt. Aus diesem Grund sind die Schedules bei einem gewünschten Gütefaktor von Zwei nur minimal schlechter als die bei einem gewünschten Gütefaktor von Eins. Bei großen Prozessoranzahlen müssen jedoch viele Module mit mehr als einem Prozessor ausgeführt werden, so daß bei einem gewünschten Gütefaktor von Zwei auch tatsächlich nur ein Gütefaktor von knapp unter Zwei erreicht wird. Wird der gewünschte Gütefaktor auf Eins gesetzt, so erhält man im Durchschnitt einen wesentlich besseren Schedule als bei einem gewünschten Gütefaktor von Zwei, hat aber auch eine um mindestens eine Größenordnung höhere Laufzeit. Da „KPMF“ um einiges schneller

ist als „KP+“ und „KP+“ in einigen Fällen sogar einen schlechteren Schedule erstellt, sollte man „KPMF“ unbedingt den Vorzug geben.

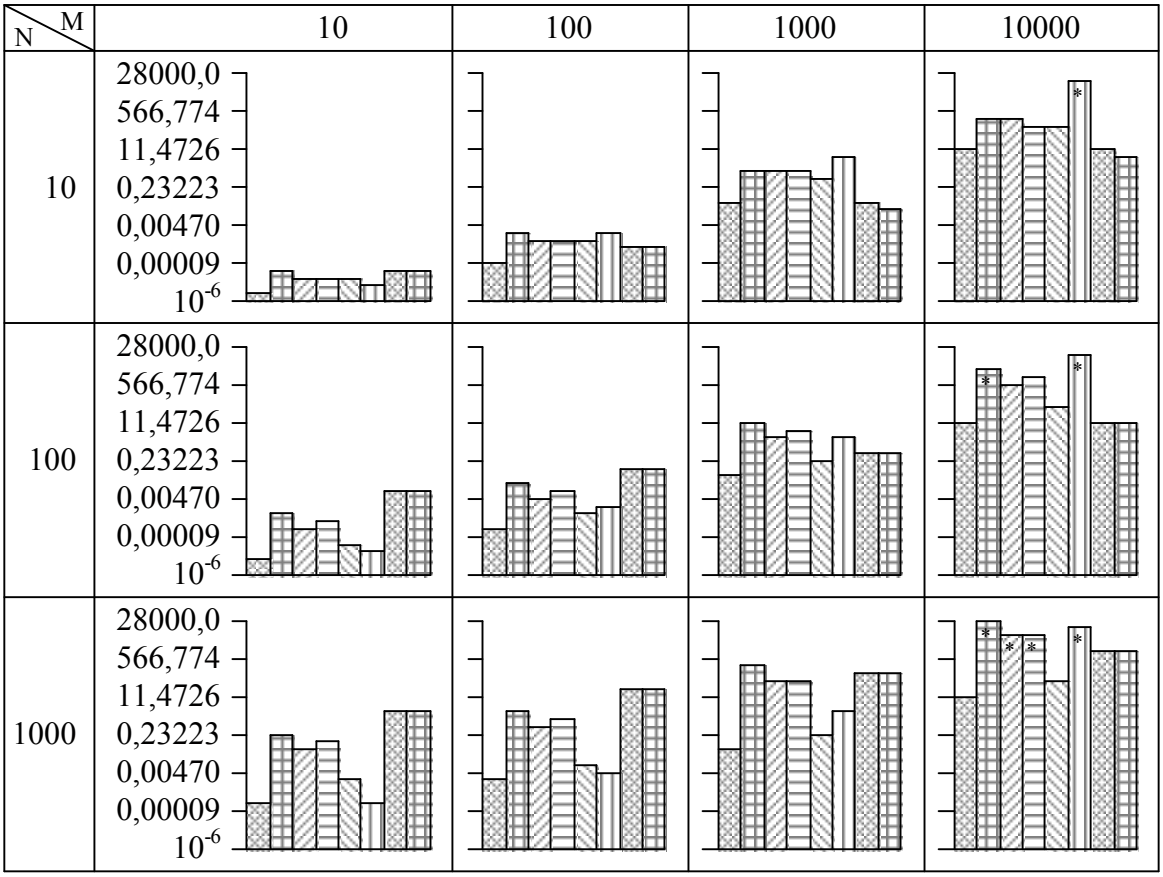


Abbildung 48 - durchschnittliche Laufzeit in Sekunden (Beispiel 5).

4.1.6 Vergleich der erzielten Ergebnisse mit denen anderer Algorithmen

In [2] findet man einen Algorithmus, welcher ebenfalls formbare Module mit beliebigem Modulabhängigkeitsgraphen optimiert. Dort wird allerdings vorausgesetzt, daß alle Module die Flächenklausel erfüllen und ihre Laufzeiten ganzzahlig sind. Dann kann für den Schedule, welcher in pseudopolynomieller Laufzeit berechnet wird, der Gütefaktor $3 + \sqrt{5} \approx 5{,}24$ bewiesen werden. Allerdings sollte man das Verfahren aus [2] aufgrund der hohen Laufzeit nur verwenden, wenn die hier angegebenen schnellen Algorithmen keinen brauchbaren Schedule erzielen.

Eine Verbesserung des in [2] vorgestellten Algorithmus findet man in [9]. Das dort angegebene Verfahren hat ähnliche Laufzeiteigenschaften und garantiert den Gütefaktor

$$100 \cdot \left(\frac{1}{43} + \frac{\sqrt{4349} - 7}{2451}\right) \approx 4{,}73.$$

Ein auf der Parallelisierung der Module, welche auf dem kritischen Pfad liegen (vgl. Algorithmus „KP+“), basierendes Verfahren wird in [3] vorgestellt. Es benötigt $O(|E| \cdot M^2 \cdot N + M^3 \cdot N \cdot (\log(M) + N \cdot \log(N)))$ Schritte und ist demzufolge um den Faktor $O(M)$ langsamer als „KP+“. Anhand von praktischen Tests wurde ein durchschnittlicher Gütefaktor von rund Zwei angegeben. Allerdings müssen bei dem in [3] vorgestellten Algorithmus die Module nicht auf benachbarten Prozessoren ausgeführt werden.

Ein weiterer, den kritischen Pfad optimierender Algorithmus, ist in [10] angegeben. Seine Laufzeit ist mit $O(M^3 \cdot N^2 + M \cdot N^2 \cdot |E|)$ ebenfalls höher als die Laufzeit von „KP+“. Diese zusätzliche Laufzeit wird benötigt, um einen berechneten Schedule, welcher womöglich nur ein lokales Optimum darstellt, wieder zu verwerfen und durch die Parallelisierung anderer Module einen besseren Schedule zu erstellen. In [11] findet man weitere Überlegungen zur Verbesserung der Algorithmen, die den kritischen Pfad als Optimierungskriterium verwenden.
Für den Fall, daß kein Modul parallelisierbar ist ($\forall x \in [1; M]\ \forall i \in [1; N]: T(x, i) = T(x, 1)$), können die Algorithmen zur Lösung des DAG - Shop - Problems genutzt werden. Das List - Scheduling Verfahren aus [35] garantiert dabei einen Gütefaktor von $2 - \frac{1}{N}$ bei einer Laufzeit von nur $O(M + N + |E|)$.
In [1] wird der Modulabhängigkeitsgraph solange transformiert bis nur noch voneinander unabhängige Module übrig sind. Das Scheduling dieser Module wird dann über die Bildung aller möglicher balancierter Binärbäume mit M Kindern durchgeführt. Durch die Nutzung der dynamischen Programmierung können zur Laufzeitreduzierung Module mit gleichen Laufzeiten zu einem Modul (mit der entsprechenden Vielfachheit) zusammengefaßt werden. Bei einer größeren Modulanzahl ist das in [1] vorgestellte Verfahren aufgrund seiner hohen Laufzeit nur bedingt anwendbar.
Unter der Annahme, daß die Module alle nur mit einem Prozessor ausgeführt werden können, ist in [51] ein schichtenorientierter Algorithmus angegeben. Der Algorithmus bricht ab, wenn er einen Schedule mit einem vorgegebenen Gütefaktor erzielt hat. Sollte er diesen Gütefaktor nicht erreichen, so terminiert das Verfahren nie.
In [16] wird ebenfalls davon ausgegangen, daß alle Module nur sequentiell ausgeführt werden können und dabei eine ganzzahlige Laufzeit haben. Sollten nun noch alle Module mit der gleichen Tiefe im Modulabhängigkeitsgraphen die gleiche Laufzeit haben, so garantiert der dort angegebene Algorithmus einen Gütefaktor von 4.
Insgesamt sind die in dieser Arbeit vorgestellten Algorithmen schneller als die in der Literatur angegebenen Verfahren mit einer ähnlichen Funktionalität. Insbesondere der Algorithmus „WaSP“ erreicht mit einer sehr geringen Laufzeit den sehr guten durchschnittlichen Gütefaktor von maximal 1,3.
Ausführliche praktische Tests von Schedulingalgorithmen, die analog zu „KP+“ und „KPMF“ die Länge des kritischen Pfades als Kriterium zur Modulparallelisierung heranziehen, findet man in [8]. Die dortigen Ergebnisse verdeutlichen unter anderem, daß bei vielen Algorithmen die Laufzeiten sehr schnell steigen. So konnten z.B. zwei der dort untersuchten Algorithmen aufgrund ihrer hohen Laufzeiten noch nicht einmal einen Schedule für 1000 Module und 16 Prozessoren erstellen. Bei einem Algorithmus mit pseudolinearer Laufzeit wurden die gegebenen Modullaufzeiten auf relativ wenig verschiedene Werte, die allesamt Vielfache von a mit $a \in \mathbb{R}$ waren, nach oben aufgerundet. Da die Laufzeit dieses Schedulingalgorithmus von $\frac{H}{a}$ abhängt, konnte nur mit dieser Diskretisierung ein Schedule berechnet werden. Man erkennt jedoch sofort, daß zur Erstellung eines brauchbaren Schedules a möglichst klein sein muß, was sich wiederum negativ auf die Algorithmuslaufzeit auswirkt.
Algorithmen, welche analog zu „SchT“ und „SchHT“ den Modulabhängigkeitsgraphen in mehrere Schichten zerlegen, werden in [7] anhand von praktischen Beispielen untersucht. Dabei wurde in allen Algorithmen das gleiche Verfahren zur Bildung der Schichten eingesetzt. Demzufolge hängt die Ergebnisgüte des Schedules direkt vom verwendeten Schedulingalgorithmus für unabhängige Module ab. Läßt man die trivialen Schedulingalgorithmen unbeachtet, so hatte der schlechteste berechnete Schedule eine gerade mal um 25% höhere Ge-

samtlaufzeit als der vom besten Algorithmus erstellte. Es ist also durchaus sinnvoll, wie hier mit „SchT“ und „SchHT“ geschehen, auch unterschiedliche Zerlegungen des Modulabhängigkeitsgraphen zu verwenden, um zu einem besseren Schedule zu gelangen.

4.2 Optimierung von nicht formbaren Modulen

4.2.1 Die Modifikation der Algorithmen

Die Algorithmen „SchT“, „SchHT“, „WaSP“ und „ReSP“ zerlegen den Konstruktionsbaum und verwenden dann das Verfahren „P+MF“ oder „2M1“ zur Optimierung der Module. Demzufolge braucht man bei diesen vier Algorithmen nur „P+MF“ durch „XP+MF“ bzw. „2M1“ durch „2M1N“ zu ersetzen, um die Versionen „SchTN“, „SchHTN“, „WaSPN“ und „ReSPN“ für nicht formbare Module zu erhalten.
Die Erweiterung der Algorithmen „KP+“ und „KPMF“ auf nicht formbare Module geschieht durch die Änderung von Zeile 2) im Pseudocode (Abbildung 45). Bei „KP+N“ wird als Anfangsprozessoranzahl eines Moduls die minimal benötigte Prozessoranzahl gewählt:

2) $\forall x \in [1; M]: B(x) = min\{ i : i \in [1; N] \text{ und } T(x, i) \neq \infty \}$

Die flächenminimale Version „KPMFN“ sucht gleich am Anfang die Prozessoranzahl, bei der das Modul den geringsten Flächenbedarf hat:

2) $\forall x \in [1; M]: B(x) = min\{ i : i \in [1; N] \text{ und } i \cdot T(x, i) = \min_{j=1}^{N}\{ j \cdot T(x, j) \} \}$

4.2.2 Praktische Tests

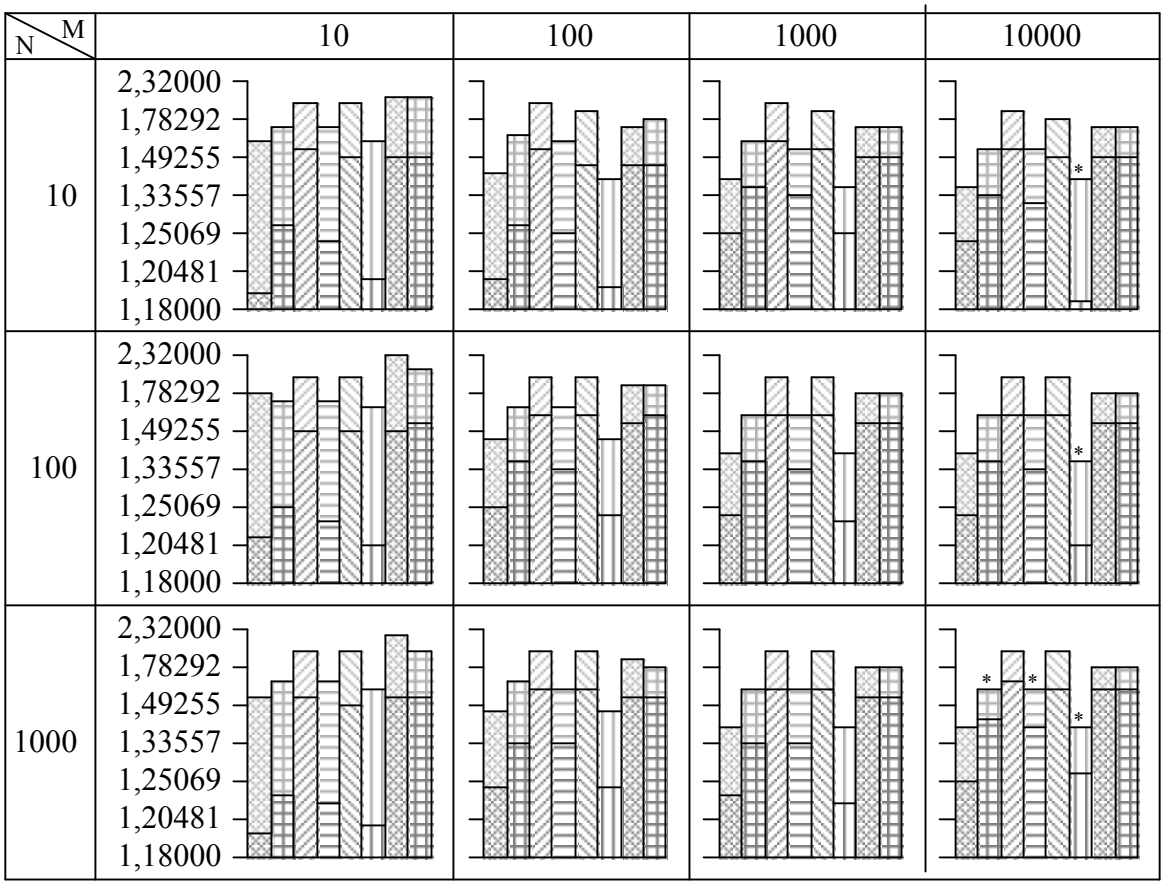


Abbildung 49 - schlechtester und durchschnittlicher Gütefaktor (Beispiel 6).

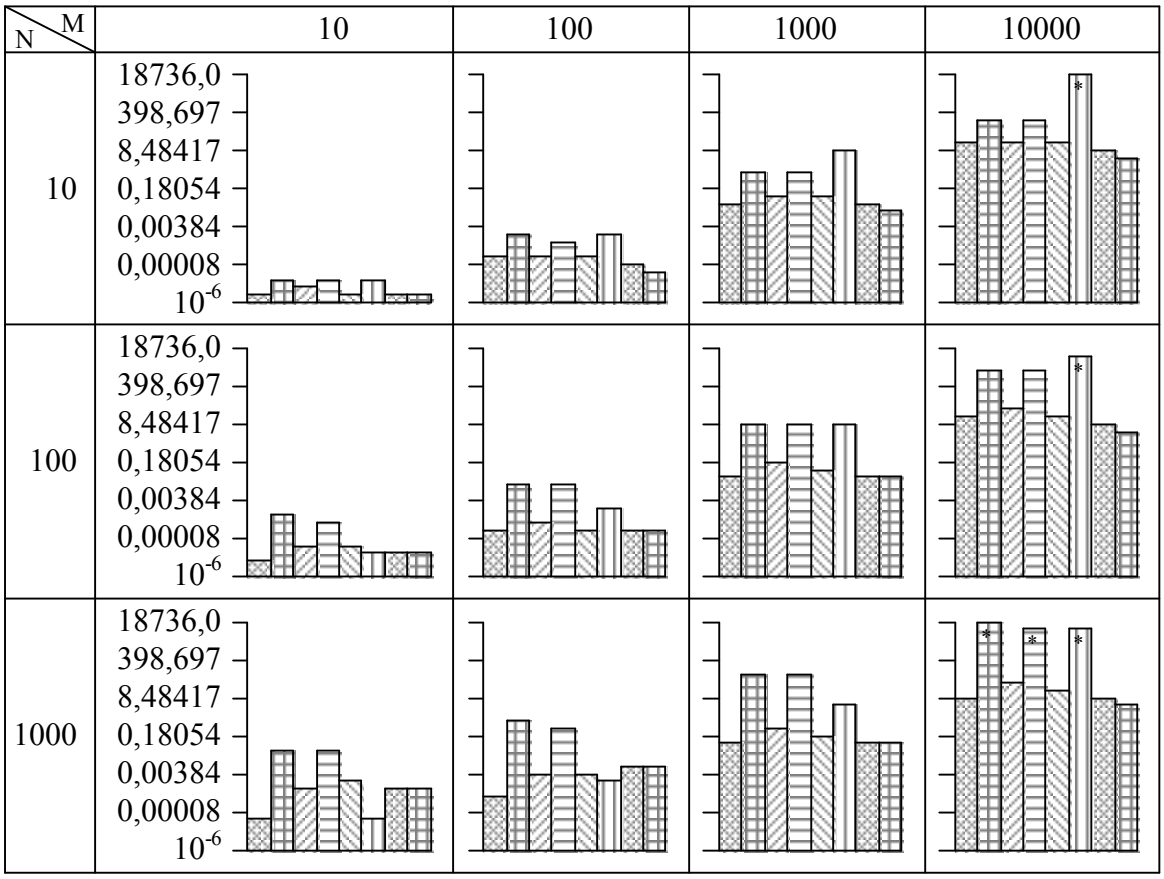


Abbildung 50 - durchschnittliche Laufzeit in Sekunden (Beispiel 6).

Bei den Testläufen mit zufällig erzeugten Beispielen (siehe Beispiel 6 Anhang 9.24) ergaben sich die in den Abbildungen 49 und 50 dargestellten Ergebnisse. Das Beispiel 6 nutzt die Modullaufzeiten von Beispiel 4 und erstellt den Modulabhängigkeitsgraphen analog zu Beispiel 5. Die Säulen wurden wie in den Abbildungen 47 und 48 angeordnet. Die genauen Meßwerte befinden sich auf der CD im Verzeichnis /res/bsp6/. Die Sterne markieren Messungen, bei denen nur 10 Testläufe ausgeführt wurden.
Für nicht formbare Module ist der Einsatz der Verfahren „KP+N“ und „KPMFN“ nicht besonders zu empfehlen. Beide Algorithmen haben eine hohe Laufzeit und liefern auch nur Schedules von befriedigender Qualität. Man sieht also sehr schön, daß „KP+N“ und „KPMFN“ ursprünglich zur Lösung des DAG - Shop - Problems gedacht waren, d.h., sie sind Spezialisten, wenn möglichst alle Module mit einem Prozessor ausgeführt werden. Bei diesen Testläufen existieren jedoch nur wenige Module, welche überhaupt mit einem Prozessor ausgeführt werden können.
Im Gegensatz zu „SchT“ und „SchHT“ ordnen die Algorithmen „SchTN“ und „SchHTN“ gleich jedem Modul x die Prozessoranzahl zu, bei welcher das Modul x eine minimale Fläche belegt. Aus diesem Grund ist die Laufzeit von „SchTN“ und „SchHTN“ im Fall $M \leqslant N$ wesentlich geringer als die der Algorithmen „SchT“ und „SchHT“. Da die Laufzeit von „WaSPN“ bzw. „ReSPN“ in etwa der von „WaSP“ bzw. „ReSP“ entspricht, sind die schichtenorientierten Verfahren bei der Optimierung von nicht formbaren Modulen in vielen Fällen schneller als die Algorithmen „WaSPN“ und „ReSPN“. Allerdings sind die Schedules von „WaSPN“ und „ReSPN“ im Durchschnitt besser als die von „SchTN“ und „SchHTN“.

Ein Algorithmus für das Rechteck - Füll - Problem mit Abhängigkeiten zwischen den Rechtecken ist in [31] gegeben. Das Verfahren zerlegt den Abhängigkeitsgraphen wie der Algorithmus „SchT“ entsprechend der Knotentiefe, so daß am Ende nur das normale Rechteck - Füll - Problem gelöst werden muß. Anhand von Testläufen wird in [31] ein Gütefaktor von rund 1,3 ermittelt.
In [38] findet man einen weiteren Algorithmus für das Rechteck - Füll - Problem mit Abhängigkeiten. Die Laufzeit des dortigen Verfahrens beträgt $O(p^2 + M + |E|)$, wobei p die Länge des kritischen Pfades angibt:

$$p = \max_{P \in \mathcal{P}} \left\{ \sum_{x \in P} T(x) \right\} \quad \text{mit} \quad \mathcal{P} \text{ ist die Menge aller Pfade in G.}$$

5 Vergleich aller Routinen

Zusätzlich zu den zufällig erzeugten Beispielen wurden alle Algorithmen für formbare Module noch mit den Standardbeispielen Matrixmultiplikation, Fast Fourier Transformation und LR - Zerlegung getestet. Da zufällig erzeugte Eingabewerte in vielen Fällen gutartig sind, d.h., den zu testenden Algorithmus nicht in seinen Grenzbereich bringen, soll an den drei Standardbeispielen überprüft werden, ob die von den Algorithmen bei zufälligen Eingaben ermittelten Ergebnisse auch bei praktischen Anwendungen erzielt werden.

5.1 Matrixmultiplikation

Das erste Standardbeispiel ist die Berechnung einer möglichst schnellen Version zur Ausführung der parallelen Matrixmultiplikation. In der folgenden Tabelle sind die Größen der zu multiplizierenden Matrizen angegeben:

Beispiel „M1“			Beispiel „M2“			Beispiel „M3“		
Größe der Matrix		Viel-	Größe der Matrix		Viel-	Größe der Matrix		Viel-
A	B	fachheit	A	B	fachheit	A	B	fachheit
1 × 114	114 × 1	1	3 × 12	12 × 4	1	1 × 90	90 × 2	2
						3 × 18	18 × 4	1
						2 × 7	7 × 2	3
						3 × 3	3 × 4	2
						4 × 1	1 × 7	1

Die Größe der Matrizen wurde so gewählt, daß bei der Verwendung von 100 Prozessoren bereits viele Module mit mehr als einem Prozessor ausgeführt werden müssen, um einen Schedule mit möglichst geringer Laufzeit zu erstellen. Für kleine Prozessoranzahlen kann man das Verhalten der Algorithmen bei der Ausführung von nahezu allen Modulen auf einem Prozessor analysieren.
Außerdem werden bei jedem Beispiel („M1“, „M2“ und „M3“) noch drei Komplexitäten unterschieden. In der ersten Komplexitätsstufe stellt ein Modul die komplette Multiplikation von zwei Matrizen dar. Demzufolge existieren im Beispiel „M3“ exakt 9 Module. Bei der zweiten Komplexitätsstufe erhält man ein Modul für jedes Element in der Ergebnismatrix, also 12 Module im Beispiel „M2“. Wird die Komplexität auf Drei gesetzt, so entspricht jede Operation (Addition bzw. Multiplikation zweier Zahlen) einem Modul. Im Beispiel „M1“ sind das 227 Module. Die Laufzeit einer Multiplikation im Verhältnis zu einer Addition wurde mit Hilfe von /src/addmul/addmul.c bestimmt. Dabei ergab sich, daß eine Multiplikation genauso viel Zeit benötigt, wie 0,75 Additionen. Ausführliche Angaben zu den Laufzeiten der Module und den Modulabhängigkeitsgraphen bei allen Standardbeispielen findet man im Anhang 9.25.
In den Abbildungen 51 bis 53 sind die Schedulingergebnisse der einzelnen Algorithmen in Abhängigkeit der Gesamtprozessoranzahl N dargestellt. Der Parallelisierungsmehraufwand ist dabei der Quotient aus der erzielten Gesamtlaufzeit H und der minimalen Laufzeit bei gleichmäßiger Auslastung aller Prozessoren. Der Quotient entspricht also den N-ten Teil der minimal benötigten Fläche. Zur Unterscheidung der drei Komplexitäten wurden die Kürzel K1 bis K3 vor den Algorithmusnamen notiert.

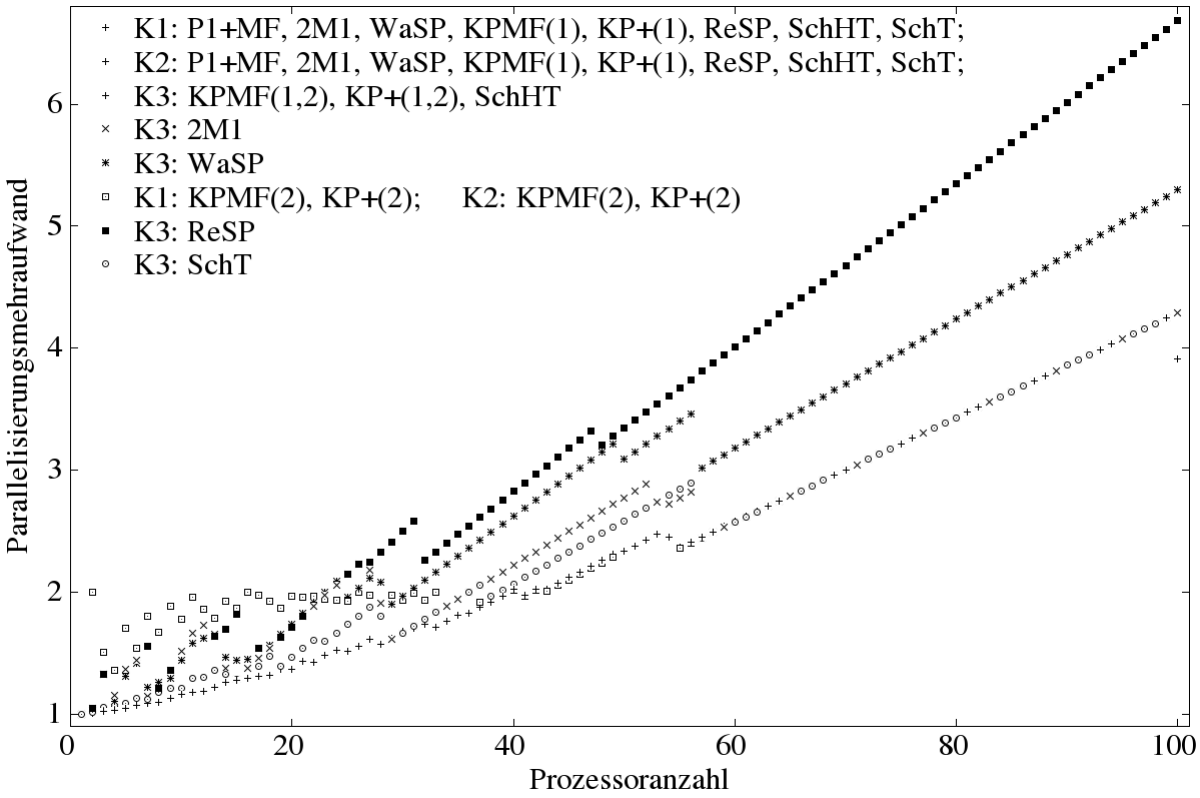


Abbildung 51 - Beispiel „M1“.

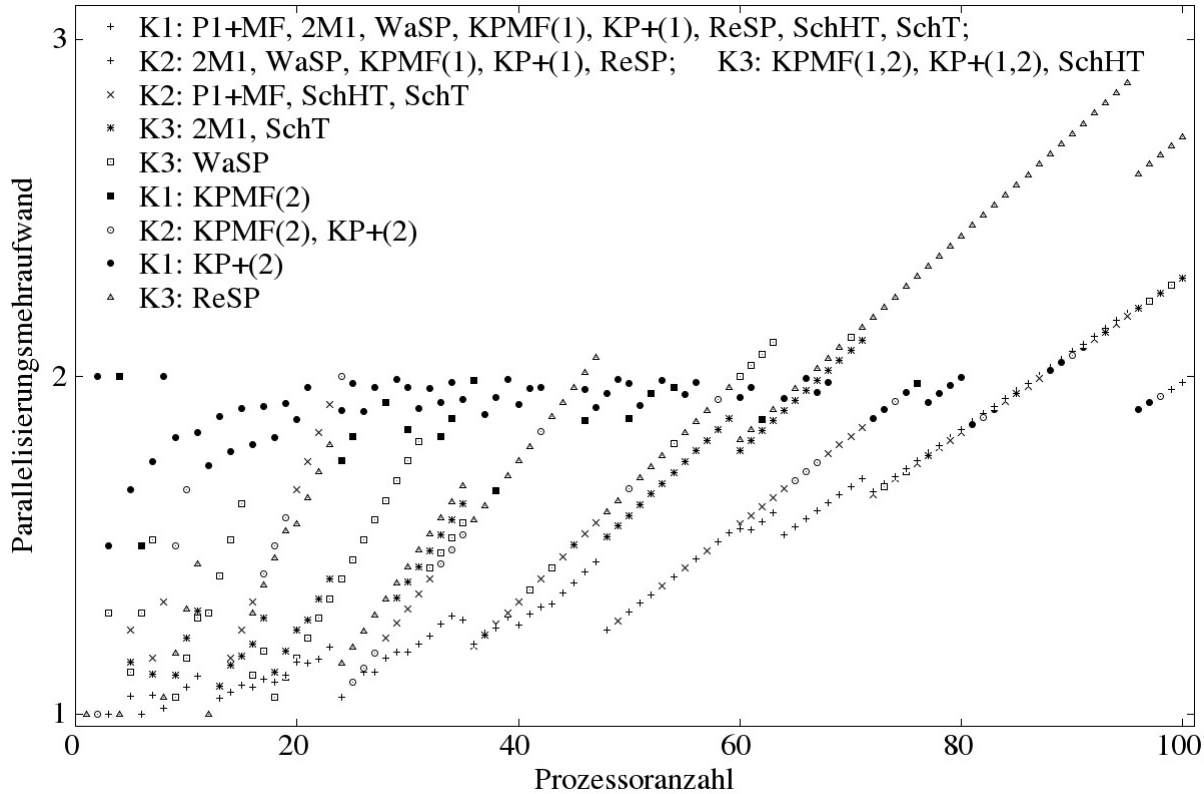


Abbildung 52 - Beispiel „M2“.

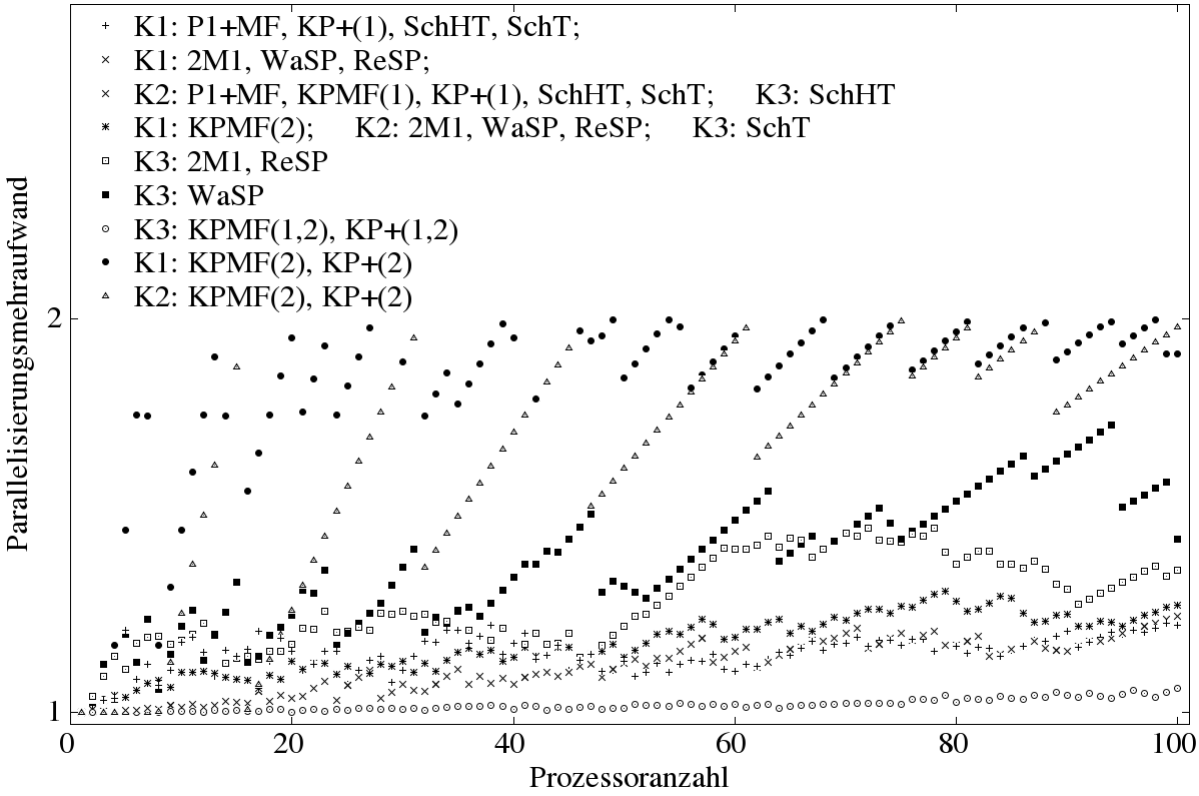


Abbildung 53 - Beispiel „M3“.

Damit nicht 31 Linien in das Diagramm eingezeichnet werden müssen, wurden Algorithmen mit ähnlichen Schedulingergebnissen zu einem Eintrag zusammengefaßt. Dies geschah mit Hilfe des Programms /src/stdplot/stdplot.c. Es vereinigt solange die zwei Meßreihen mit der geringsten quadratischen Abweichung, bis maximal nur noch 8 Meßreihen übrig sind. Dabei ergab sich, daß die Meßreihen von „P1+MF“ bei der Verwendung von „HMZ“ und „BMZ“ immer zusammen fielen. Die Zahl in Klammern hinter „KPMF“ und „KP+“ gibt den gewünschten Gütefaktor an.
Eine Auswertung der Ergebnisse aller Standardbeispiele ist im Kapitel 5.4 zu finden.

5.2 Fast Fourier Transformation (FFT)

Als zweites Standardbeispiel wurde die FFT gewählt. Als Dimensionsangaben sind nur Zweierpotenzen echt größer als Eins erlaubt. Die genauen Angaben zu den verwendeten Beispielen sind in der folgenden Tabelle enthalten:

Beispiel „F1“		Beispiel „F2“		Beispiel „F3“	
Dimension r	Vielfachheit	Dimension r	Vielfachheit	Dimension r	Vielfachheit
64	1	16	2	32	2
		8	3	8	10
		2	2	4	20

Die Komplexität Eins erstellt für jede FFT ein Modul, also beim Beispiel „F3“ genau 32 Stück. Wird die Komplexität auf Zwei gesetzt, so entstehen für jede FFT zwei balancierte Binärbäume mit je r - 1 Modulen. Bei der Komplexität Drei entspricht jede Operation einem

Modul, d.h., aus einer Multiplikation im Bereich der komplexen Zahlen werden sechs Module. Die Ergebnisse sind in den Abbildungen 54 bis 56 dargestellt.

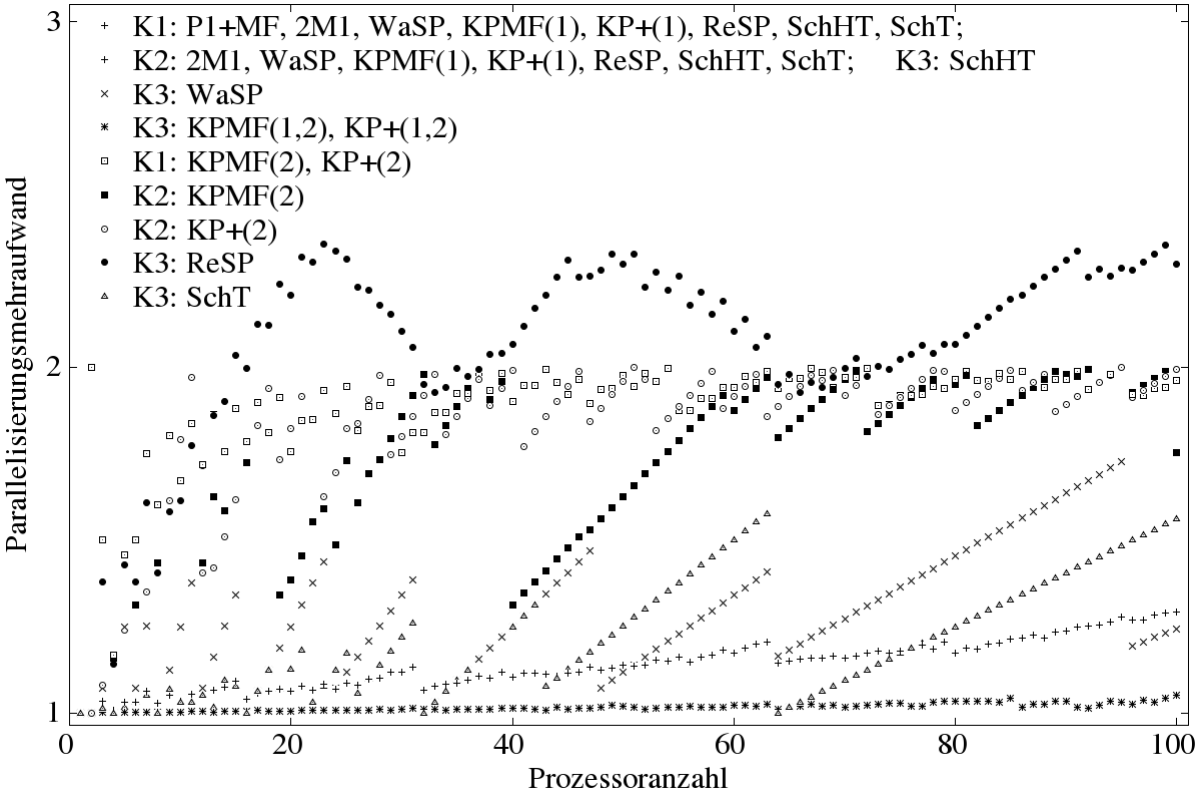


Abbildung 54 - Beispiel „F1“.

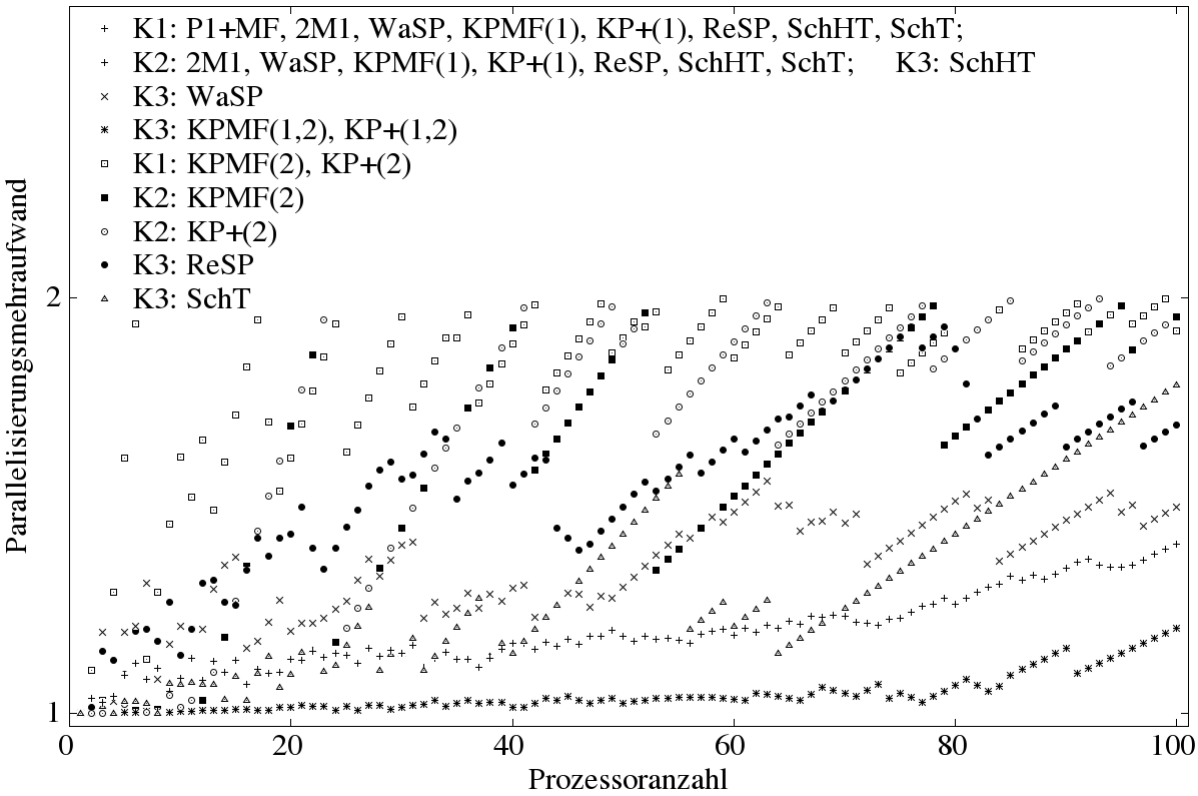


Abbildung 55 - Beispiel „F2“.

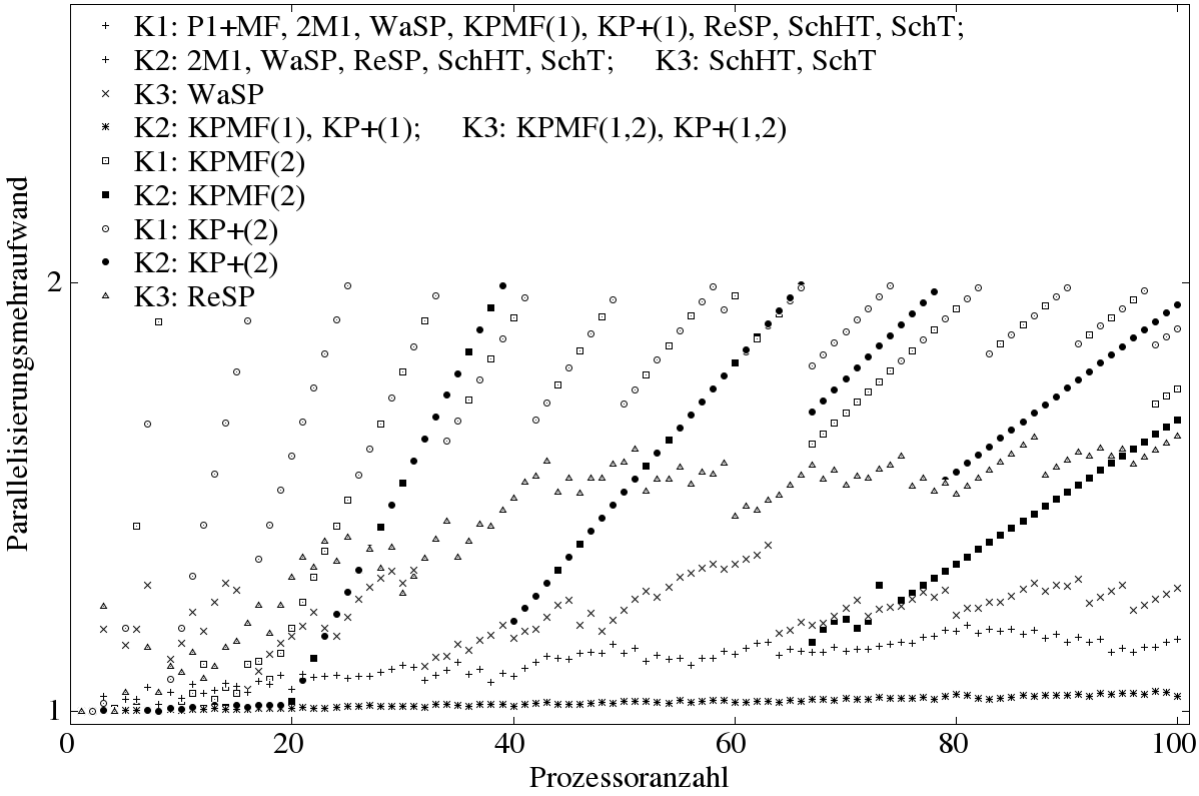


Abbildung 56 - Beispiel „F3“.

5.3 LR - Zerlegung mittels Gauß

Beim Gaußalgorithmus wird die Laufzeit einer Division benötigt. Mit Hilfe von /src/addmul/addmul.c wurde die Laufzeit einer Division als Laufzeit von 4,75 Additionen bestimmt. Unter dieser Voraussetzung war es möglich, die Schedulingalgorithmen anhand der folgenden Beispiele zu testen:

Beispiel „G1“		Beispiel „G2“		Beispiel „G3“	
Dimension r	Vielfachheit	Dimension r	Vielfachheit	Dimension r	Vielfachheit
20	1	6	4	10	6
		4	16	5	10
		3	6	4	13
		2	1		

Wie auch in den vorangegangenen Beispielen steht die Komplexität Eins für die Abbildung einer LR - Zerlegung auf ein Modul. Bei der Komplexität Zwei wird jede Division und die nachfolgenden Multiplikationen und Subtraktionen zu einem Modul zusammengefaßt. Der entstehende Modulabhängigkeitsgraph wird durch die Hinzunahme weiterer Kanten zu einem serien - parallelen Graphen erweitert. Die Komplexität Drei bedeutet wiederum, daß jede Operation durch ein Modul repräsentiert wird. Im Anhang 9.25.3 finden sich weitere Details zur Erstellung der Eingabedaten.
Die Abbildungen 57 bis 59 stellen die Ergebnisse graphisch dar.

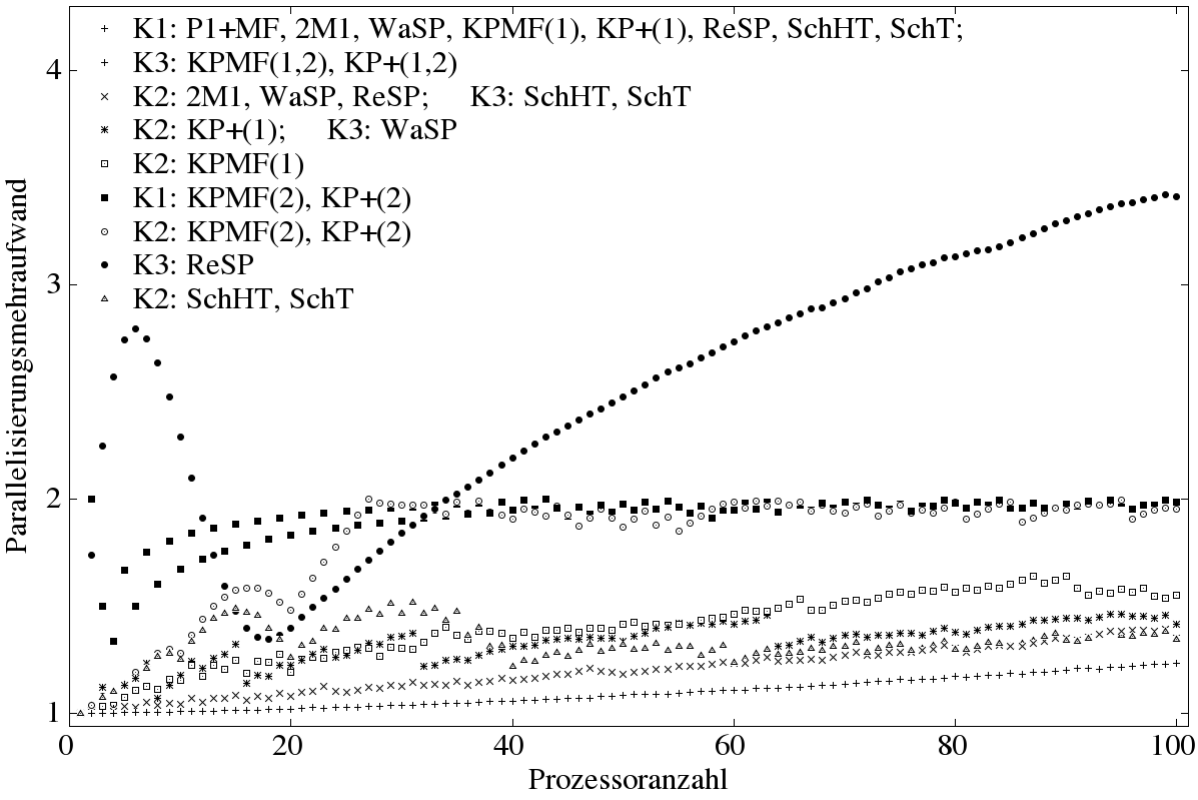


Abbildung 57 - Beispiel „G1“.

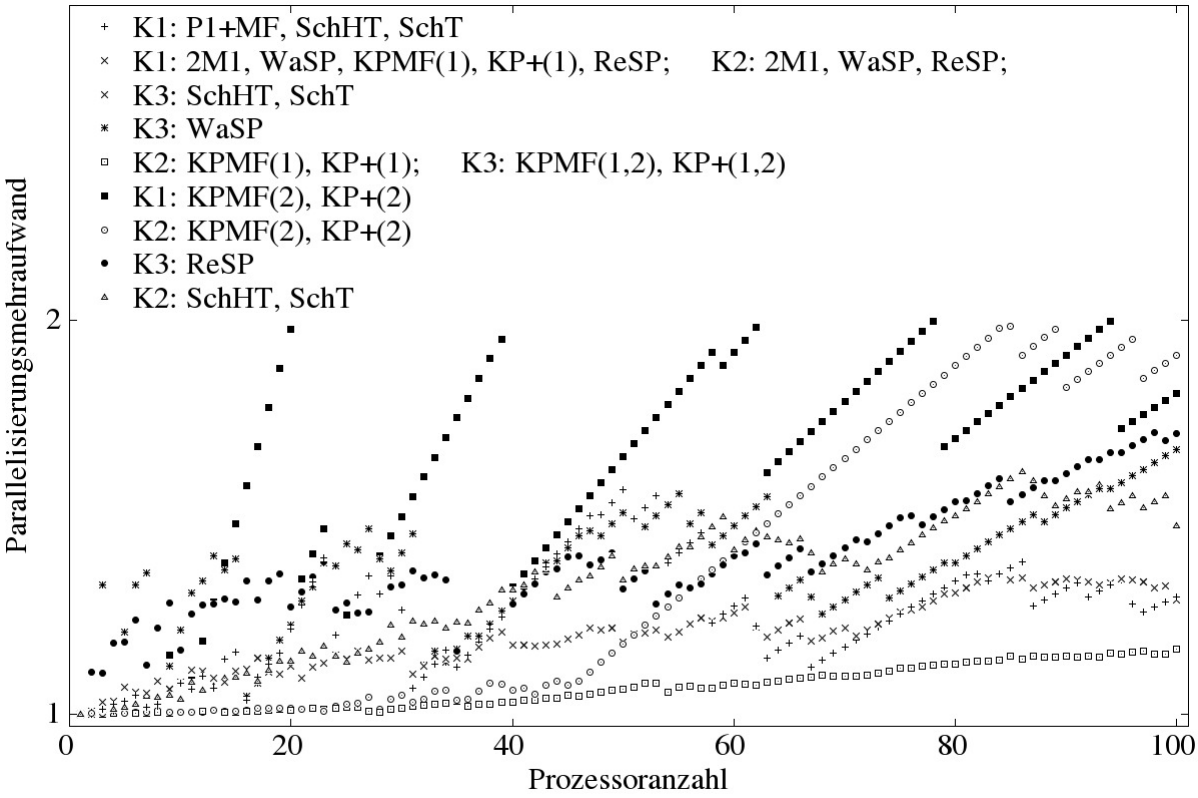


Abbildung 58 - Beispiel „G2“.

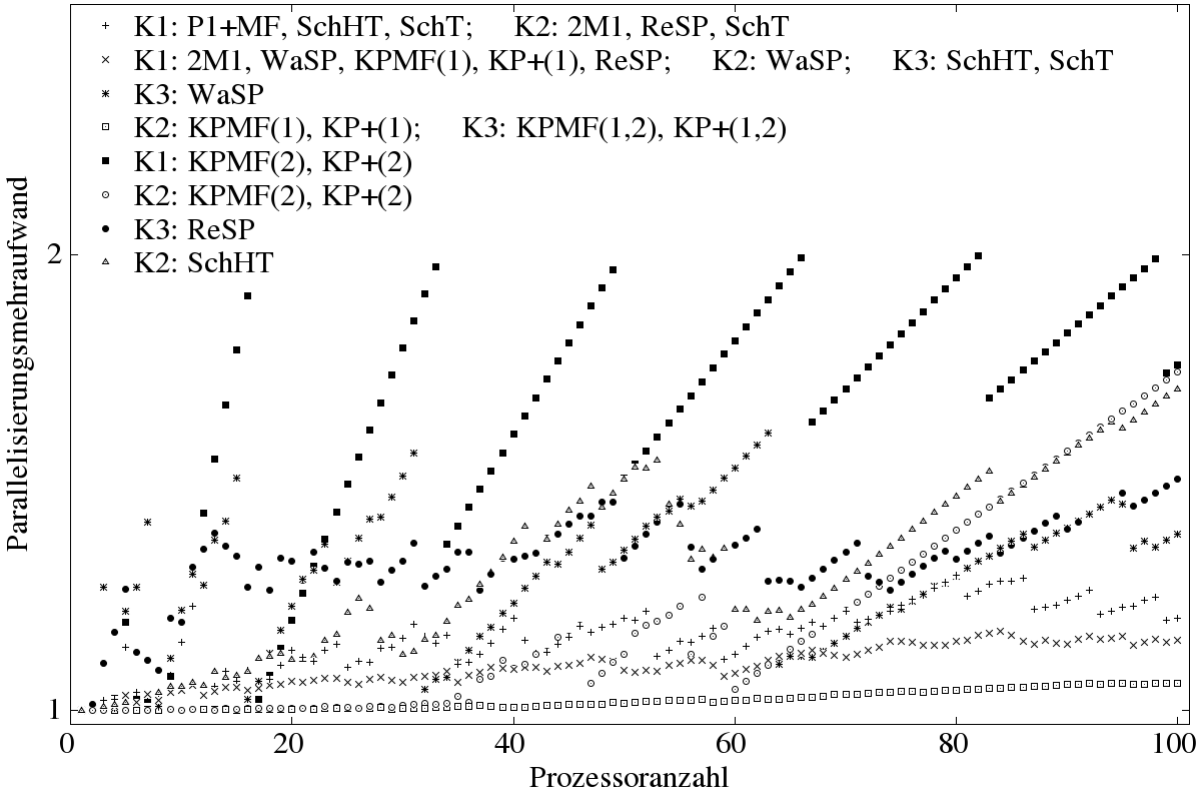


Abbildung 59 - Beispiel „G3“.

5.4 Auswertung der Ergebnisse

Da die Algorithmen „KP+(2)“ und „KPMF(2)“ beim Erreichen des Gütefaktors Zwei abbrechen, sind ihre Schedules bei geringer Modulanzahl (Komplexität 1) auch meistens schlechter als die der anderen Algorithmen. Wird dagegen die Komplexität auf Drei gesetzt, so liefern sie sehr gute Schedules. Dann entspricht die Eingabe auch einem DAG - Shop - Scheduling Problem, wofür „KP+“ und „KPMF“ ursprünglich nur konzipiert waren. Wird der gewünschte Gütefaktor bei „KP+“ und „KPMF“ auf Eins gesetzt, so erreichen sie bei geringer Modulanzahl ein zufriedenstellendes Ergebnis. Bei einer hohen Modulanzahl sind die Ergebnisse von der Wahl des gewünschten Gütefaktors praktisch unabhängig, da bereits die erste Positionierung ein sehr gutes Endergebnis erstellt. Allerdings kann es bei der Verwendung von „KP+“ vorkommen, daß für N_1 Prozessoren ein Schedule mit einer größeren Gesamtlaufzeit als für N_2 $(N_2 < N_1)$ Prozessoren erstellt wird. Die Gesamtlaufzeit bildet also in Abhängigkeit von der Gesamtprozessoranzahl keine monoton fallende Funktion. Ein Beispiel dafür ist der Übergang zu $N = 77$ bei der Anwendung von „KP+“ mit dem gewünschten Gütefaktor Eins auf das Beispiel „G2“ in der Komplexität Zwei. „KP+“ und „KPMF“ sind sehr gut im Fall $d \gg N$ verwendbar. Dabei steht d für den minimalen Durchmesser des Modulabhängigkeitsgraphen.
Das Verfahren „ReSP“ erzielt bei schlecht parallelisierbaren Modulen (Komplexität 3) keine Schedules mit zufriedenstellenden Gütefaktoren. Im Beispiel „G1“ (Komplexität 3) ist „ReSP“ sogar um Größenordnungen schlechter als die anderen Algorithmen. Stehen jedoch nur wenige, aber gut parallelisierbare Module zur Verfügung (Komplexität 1), so liefert dieser Algorithmus in vielen Fällen den besten Schedule.
Da es bei allen Beispielen kaum Module x mit

$$tief(x) + hoch(x) \neq maxtief + 1$$

gibt, erzeugen die Algorithmen „SchT“ und „SchHT“ fast immer den gleichen Schedule. Auch sie gehören in vielen Fällen zu den Algorithmen, die sehr gute Resultate liefern. Gegenüber Änderungen der Komplexität der Beispiele sind beide Verfahren ebenfalls robust. „SchT“ und „SchHT“ sind also bei diesen Beispielen die beste Wahl, da sie eine wesentlich geringere Laufzeit aufweisen als „KP+“ und „KPMF“ bei einem gewünschten Gütefaktor von Eins.
Ein Algorithmus mit guten Schedulingergebnissen unabhängig vom gewählten Beispiel ist „WaSP“. Er gehört zwar nicht immer zu den Verfahren, die das beste Resultat liefern, aber auch nie zu den schlechteren Algorithmen. Obwohl er ein viel primitiveren Algorithmus zur Überführung des Modulabhängigkeitsgraphen auf einen serien - parallelen Graphen als „ReSP“ hat, erzielt er z.B. bei der FFT durchweg bessere Ergebnisse.
Wenn „2M1“ einsetzbar ist, so entsprechen die von „2M1“ berechneten Schedules in der Regel denen von „WaSP“ oder sind etwas besser.
Die Schedulingergebnisse von „P1+MF“ decken sich fast immer mit den Schedulingergebnissen von „SchT“ bzw. „SchHT“. Je nach Verwendung von „BMZ“ oder „HMZ“ für „P1+MF“ erhält man aber z.B. bei „F3“ (Komplexität 1) für $N = 9$ geringfügig verschiedene Gesamtlaufzeiten.

Insgesamt sollte man also immer versuchen „P1+MF“ oder wenigstens „2M1“ zu verwenden, damit die Eigenschaften des Modulabhängigkeitsgraphen vollständig ausgenutzt werden können. Ist die Anwendung dieser Algorithmen nicht möglich, oder ist man mit der von ihnen erreichten Ergebnisgüte nicht zufrieden, so kann man auf die anderen Algorithmen zurückgreifen. Da alle vorgestellten Verfahren Stärken und Schwächen haben, ist die Angabe eines besten Algorithmus für den allgemeinen Fall nicht möglich.
Aufgrund der unterschiedlichen Laufzeiten der Algorithmen ist es gerade bei größeren Schedulingproblemen sinnvoll, zuerst einen Schedule mit einem schnellen Verfahren zu berechnen. Sollte man mit der Gesamtlaufzeit des Schedules dieses Algorithmus nicht zufrieden sein, so kann man anschließend noch mit einem zweiten Verfahren versuchen, ein besseres Resultat zu erzielen.
Bei der Auswahl des zweiten Algorithmus ist zu beachten, daß man drei Gruppen an Algorithmen bzgl. ihrer Arbeitsweise unterscheiden kann. Die erste Gruppe bilden die Verfahren, welche auf der Optimierung unabhängiger Modulmengen beruhen, also „P1+MF“, „SchT“ und „SchHT“. Die auf der Basis von serien - parallelen Graphen arbeitenden Algorithmen „2M1“, „WaSP“ und „ReSP“ bilden die zweite Gruppe. Die Verfahren „KP+“ und „KPMF“ lehnen sich an die Lösungsalgorithmen zum DAG - Shop - Problem an und bilden somit die dritte Gruppe. Es ist in fast allen Fällen sinnvoll, den zweiten Algorithmus aus einer anderen Algorithmusgruppe zu wählen, um eine vollständig andere Ergebnisstruktur zu erhalten. Nur so kann verhindert werden, daß bei der Berechnung eines Schedules die gleichen Fehlentscheidungen zur Positionierung eines Moduls wieder getroffen werden.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurden verschiedene Algorithmen zur Berechnung eines möglichst optimalen Schedules für eine gegebene Menge an parallelisierbaren Modulen vorgestellt. Dabei wurden zuerst spezielle Modulabhängigkeitsgraphen, wie der leere Graph oder die Menge der serien - parallelen Graphen, betrachtet. Unter der Ausnutzung der dabei erzielten Ergebnisse wurden Algorithmen zur Berechnung eines Schedules beim Vorliegen eines beliebigen Modulabhängigkeitsgraphen angegeben.
Beim Design aller Algorithmen wurde großen Wert auf eine möglichst geringe Algorithmuslaufzeit gelegt. Nur dadurch kann der Schedulingalgorithmus in der Praxis effektiv eingesetzt werden, denn die Berechnung eines Schedules darf nicht länger dauern, als der durch die Anwendung des Schedulingalgorithmus erzielte Laufzeitgewinn ist. Aus diesem Grund wurde versucht, für die Schedulingalgorithmen eine Laufzeit in der Größenordnung der Sortieralgorithmen zu erreichen. Dieser Wert konnte sogar bei den Schedulingalgorithmen „2M1“ und „WaSP“ noch unterboten werden.
Trotz der geringen Algorithmuslaufzeit sollte der berechnete Schedule eine möglichst gute Suboptimalität aufweisen. Für den Fall, daß der Modulabhängigkeitsgraph leer ist, konnte dies für die vorgestellten Algorithmen bewiesen werden. Bei allen anderen Schedulingalgorithmen war leider immer die Angabe eines Beispiels, bei welchem der Schedulingalgorithmus nur einen Gütefaktor von $N - \varepsilon$ erzielt, möglich. Dabei steht N für die Anzahl der Prozessoren des Parallelrechners und ε ist eine beliebig kleine positive Zahl. Neben der theoretischen Analyse der Suboptimalität wurden die Algorithmen praktischen Tests unterzogen. Als Eingabedaten dienten einerseits zufällig erzeugte Modullaufzeiten und Modulabhängigkeitsgraphen, andererseits die LR - Zerlegung nach Gauß, die Matrixmultiplikation und die FFT. Bei diesen Beispielen ergab sich, daß die in dieser Arbeit vorgestellten Algorithmen eine gute bis sehr gute Suboptimalität liefern. Allerdings war es nicht möglich einen Schedulingalgorithmus anzugeben, welcher bei allen Testeingaben das beste Resultat erzielt.

In der Literatur findet man ebenfalls einige Schedulingalgorithmen für das in dieser Arbeit betrachtete Schedulingproblem. Für das Scheduling von Modulen mit leerem Modulabhängigkeitsgraphen werden in [17], [19], [21], [23] und [25] Optimierungsalgorithmen vorgestellt. Alle dort angegebenen Algorithmen erzielen einen garantierten Gütefaktor von maximal 2,5. Sie liefern also im schlechtesten Fall ein besseres Ergebnis als der in dieser Arbeit vorgestellte Algorithmus „P1+MF“ mit einem garantierten Gütefaktor von Drei. Allerdings brauchen die in der Literatur angegebenen Algorithmen mindestens eine Größenordnung mehr an Laufzeit als „P1+MF“, um diesen etwas besseren Gütefaktor zu erzielen. Sollte der Gütefaktor in der Nähe von Eins liegen, so werden sogar pseudopolynomielle Laufzeiten benötigt.
In [2] und [12] werden Algorithmen zur näherungsweisen Berechnung eines Schedules beim Vorhandensein eines serien - parallelen oder baumartigen Modulabhängigkeitsgraphen angegeben. Diese Algorithmen garantieren relativ gute Gütefaktoren von maximal $4 + \varepsilon$. Leider sind sie aufgrund der pseudopolynomiellen Laufzeit nur bedingt in der Praxis einsetzbar.
Für beliebige Modulabhängigkeitsgraphen ist in [2] ein Algorithmus mit pseudopolynomieller Laufzeit und einem maximalen Gütefaktor von rund 5,2 angegeben. In [3] findet man einen Algorithmus ohne garantierten Gütefaktor welcher in polynomieller Zeit arbeitet. Er ist allerdings mindestens eine Größenordnung langsamer als die in dieser Arbeit angegebenen Algorithmen.

Zusammenfassend läßt sich sagen, daß die in dieser Arbeit vorgestellten Verfahren wesentlich schneller sind als die in der Literatur angegebenen Algorithmen. Die von „2M1“ und „WaSP“ benötigte Laufzeit liegt bereits in der Größenordnung der Eingabedaten, d.h., ein Algorithmus mit einer kleineren asymptotischen Laufzeit kann nur erstellt werden, wenn nicht alle Eingabedaten gelesen werden. Durch die geringe Laufzeit der in dieser Arbeit vorgestellten Verfahren konnten sie auch mit größeren Modul- und Prozessoranzahlen praktisch getestet werden. Dies war bei vielen in der Literatur angegebenen Algorithmen wegen der hohen Laufzeit nicht möglich, was ihren Einsatz in der Praxis relativ stark einschränkt.
In der Literatur wird sehr häufig eine Ergebnisgüte von $1 + \varepsilon$ bewiesen. An diese Schranke kommt keines der in dieser Arbeit vorgestellten Verfahren heran. In den praktischen Tests für formbare Module wurde als schlechtester Gütefaktor der Wert 2,6 von allen in dieser Arbeit vorgestellten Algorithmen nicht überschritten. Es sei nun t_1 die Laufzeit eines Algorithmus mit dem Gütefaktor $1 + \varepsilon$ und t_2 die Laufzeit eines in dieser Arbeit vorgestellten Algorithmus. Für diesen Algorithmus wird angenommen, daß der von ihm erzielte Schedule den maximalen Gütefaktor von 2,6 hat. Mit O werde wie bisher die Laufzeit des optimalen Schedules bezeichnet. Außerdem stehe i für die Anzahl der zu erwartenden Ausführungen des mit Hilfe der Schedulingalgorithmen zu optimierenden Verfahrens. Man erkennt, daß sich der Einsatz des Algorithmus mit dem Gütefaktor $1 + \varepsilon$ nur lohnt, wenn

$$t_1 + O \cdot (1 + \varepsilon) \cdot i < t_2 + O \cdot 2{,}6 \cdot i \quad \Leftrightarrow \quad t_1 < t_2 + O \cdot i \cdot (1{,}6 - \varepsilon)$$

ist. Da t_2 polynomiell von der Modulanzahl M und der Gesamtprozessoranzahl N abhängt ergibt sich

$$t_1 < O(i \cdot \text{poly}(M, N, O)).$$

Andererseits hängt t_1 in allen in der Literatur angegebenen Algorithmen mit dem Gütefaktor $1 + \varepsilon$ exponentiell von N, M oder O ab. Aus diesem Grund lohnt sich der Einsatz der Algorithmen mit dem Gütefaktor $1 + \varepsilon$ höchstens bei einem sehr großem i. Somit sind alle in dieser Arbeit vorgestellten Verfahren in der Regel wesentlich besser in der Praxis einsetzbar, als viele in der Literatur angegeben Algorithmen.

Ziel nachfolgender Untersuchungen der in dieser Arbeit betrachteten Problemstellung sollte es sein, einen Schedulingalgorithmus mit bewiesenem maximalen Gütefaktor G und polynomieller Laufzeit bzgl. der Modulanzahl M und der Gesamtprozessoranzahl N zu entwickeln. Es ist natürlich auch denkbar, daß ein Beweis gefunden wird, der die Existenz eines solchen Algorithmus widerlegt.
Weiterhin kann die Problemstellung um die Betrachtung von Kommunikationszeiten bei der Datenübergabe von einem Modul an ein nachfolgendes Modul erweitert werden. Dadurch wäre das DAG - Shop - Scheduling ein Spezialfall der zu betrachtenden Problemstellung. Durch die Aufnahme von Kommunikationszeiten könnten z.B. Schedules für Message - Passing - Systeme wesentlich exakter berechnet werden.
Aufgrund der hohen Laufzeit der Schedulingalgorithmen kann auch die Entwicklung von parallelen Schedulingalgorithmen günstig sein. Betrachtet man die Algorithmen „SchT“ und „SchHT“ hinsichtlich ihrer Parallelisierbarkeit, so können die Teilschedules unabhängig voneinander mittels „P1+MF“ berechnet werden. Zur parallelen Berechnung des Konstruktionsbaumes eines serien - parallelen Graphen ist in [61] ein Algorithmus angegeben. Somit bilden einige der in dieser Arbeit vorgestellten Algorithmen bereits eine gute Grundlage zur Erstellung von parallelen Schedulingalgorithmen. Außerdem kann man die vorgestellten Schedu-

lingalgorithmen auch auf sich selbst anwenden, um zu parallelen Schedulingalgorithmen zu gelangen.

7 Index

2M1 ..... Schedulingalgorithmus für formbare Module mit serien - parallelem Modulabhängigkeitsgraphen .... *Seite 73*

2M1N ..... Schedulingalgorithmus für nicht formbare Module mit serien - parallelem Modulabhängigkeitsgraphen ..... *Seite 86*

B(x) ..... Prozessoranzahl des Moduls x im Schedule ..... *Seite 11*

BHMZ ..... Algorithmus für das Rechteck - Füll - Problem ..... *Seite 33*

BZM ..... Algorithmus für das Rechteck - Füll - Problem ..... *Seite 30*

DAG Shop Scheduling ..... Scheduling Verfahren ..... *Seite 16*

E(i) ..... Endzeit des Prozessors i ..... *Seite 18*

Endzeit ..... Zeitpunkt ab dem ein Prozessor im Leerlauf ist ..... *Seite 18*

Flächenbedingung ..... Moduleigenschaft, kein super linearer Geschwindigkeitsgewinn bei Parallelisierung ..... *Seite 13*

Flächenklausel ..... Moduleigenschaft, Mehraufwand zur parallelen Modulausführung steigt bei Hinzunahme von Prozessoren ..... *Seite 22*

formbar ..... Moduleigenschaft, Modulausführung ist auch mit einem Prozessor möglich ..... *Seite 12*

G ..... Gütefaktor eines Schedules ..... *Seite 11*

Gang Scheduling ..... Scheduling Verfahren ..... *Seite 81*

Gantt - Diagramm ..... Graphische Darstellung eines Schedules ..... *Seite 14*

Gesamtlaufzeit ..... Benötigte Zeit zur Ausführung aller Module in einem berechneten Schedule ..... *Seite 11*

Gütefaktor ..... Qualität eines berechneten Schedules ..... *Seite 11*

H ..... Gesamtlaufzeit des berechneten Schedules ..... *Seite 11*

hoch(x) ..... Höhe des Moduls x bzgl. der Pfade im Modulabhängigkeitsgraphen ..... *Seite 91*

Höhe ..... Maximale Entfernung eines Moduls von den Senken des Modulabhängigkeitsgraphen ..... *Seite 91*

indeg(x) ..... Eingangsgrad des Moduls x ..... *Seite 52*

komplexe serielle Vereinigung .. Spezialfall der seriellen Vereinigung bei serien - parallelen Graphen ..... *Seite 55*

Konstruktionsbaum ..... Beschreibung eines serien - parallelen Graphen in Form eines Baumes ..... *Seite 53*

Konstruktionsterm ..... Beschreibung eines serien - parallelen Graphen in Form eines mathematischen Terms ..... *Seite 53*

KoVer ..... Algorithmus zur Auflösung komplexer Vereinigungen in einem serien - parallelen Graphen ..... *Seite 56*

KPMF ..... Schedulingalgorithmus für formbare Module und einem beliebigen Modulabhängigkeitsgraphen ..... *Seite 109*

KPMFN ..... Schedulingalgorithmus für nicht formbare Module und einem beliebigen Modulabhängigkeitsgraphen ..... *Seite 116*

KP+ ..... Schedulingalgorithmus für formbare Module und einem beliebigen Modulabhängigkeitsgraphen ..... *Seite 109*

KP+N ..... Schedulingalgorithmus für nicht formbare Module und einem beliebigen Modulabhängigkeitsgraphen ..... *Seite 116*

kritischer Pfad ..... Pfad im Modulabhängigkeitsgraphen, welcher in Summe die größten Modullaufzeiten enthält ..... *Seite 109*

HMZ ..... Algorithmus für das Rechteck - Füll - Problem ..... *Seite 28*

M ..... Anzahl aller Module ..... *Seite 11*

moldable ..... Moduleigenschaft, Modulausführung ist auch mit einem Prozessor möglich ..... *Seite 12*

maxtief ..... Tiefe des Modulabhängigkeitsgraphen ..... *Seite 91*

Modulabhängigkeitsgraph ..... Beschreibt den Datenfluß zwischen den Modulen ..... *Seite 11*

Modulfläche ..... Produkt aus Modullaufzeit und verwendeter Prozessoranzahl ..... *Seite 14*

N ..... Anzahl an verfügbaren Prozessoren ..... *Seite 11*

non-moldable ..... Moduleigenschaft, Modulausführung ist nicht mit einem Prozessor möglich ..... *Seite 12*

O ..... Laufzeit des optimalen Schedules ..... *Seite 11*

outdeg(x) ..... Ausgangsgrad des Moduls x ..... *Seite 52*

P(x) ..... Referenzprozessor des Moduls x im Schedule ..... *Seite 11*

P1 ..... Schedulingalgorithmus für formbare Module ohne Modulabhängigkeiten ..... *Seite 18*

P1+ ..... Schedulingalgorithmus für formbare Module ohne Modulabhängigkeiten ..... *Seite 22*

P1+MF ..... Schedulingalgorithmus für formbare Module ohne Modulabhängigkeiten ..... *Seite 33*

prio(x) ..... Ausführungspriorität des Moduls x ..... *Seite 109*

Prozessoranzahl ..... Anzahl an Prozessoren die auf dem Parallelrechner verfügbar sind oder zur Ausführung eines Moduls verwendet werden ..... *Seite 11*

Rechteck - Füll - Problem ..... Positionierungsproblem von Rechtecken ..... *Seite 15 sowie Seite 28*

Referenzprozessor ..... Linker Prozessor welcher in einem Schedule zur Ausführung eines Moduls verwendet wird ..... *Seite 11*

ReSP ..... Schedulingalgorithmus für formbare Module und einem beliebigen Modulabhängigkeitsgraphen ..... *Seite 100*

ReSPN ..... Schedulingalgorithmus für nicht formbare Module und einem beliebigen Modulabhängigkeitsgraphen ..... *Seite 116*

S(x) ..... Startzeit des Moduls x in einem Schedule ..... *Seite 11*

SchHT ..... Schedulingalgorithmus für formbare Module und einem beliebigen Modulabhängigkeitsgraphen ..... *Seite 94*

SchHTN ..... Schedulingalgorithmus für nicht formbare Module und einem beliebigen Modulabhängigkeitsgraphen ..... *Seite 116*

SchT ..... Schedulingalgorithmus für formbare Module und einem beliebigen Modulabhängigkeitsgraphen ..... *Seite 92*

SchTN ..... Schedulingalgorithmus für formbare Module und einem beliebigen Modulabhängigkeitsgraphen ..... *Seite 116*

serien - parallel ..... Eigenschaft eines Graphen ..... *Seite 51*
seiten - paralleler Teilgraph ..... Von einem Knoten aufgespannter Teilgraph eines serien - parallelen Graphen ..... *Seite 56*
Startzeit ..... Zeitpunkt ab dem ein Modul in einem Schedule ausgeführt wird ..... *Seite 11*
T(x) ..... Ausführungszeit von Modul x bei der Verwendung von B(x) Prozessoren ..... *Seite 11*
T(x, i) ..... Ausführungszeit von Modul x bei der Verwendung von i Prozessoren ..... *Seite 11*
tief(x) ..... Tiefe des Moduls x bzgl. der Pfade im Modulabhängigkeitsgraphen ..... *Seite 91*
Tiefe ..... Maximale Entfernung eines Moduls von den Quellen des Modulabhängigkeitsgraphen ..... *Seite 91*
transitiv ..... Kanteneigenschaft im Graphen ..... *Seite 102*
WaSP ..... Schedulingalgorithmus für formbare Module und einem beliebigen Modulabhängigkeitsgraphen ..... *Seite 98*
WaSPN ..... Schedulingalgorithmus für nicht formbare Module und einem beliebigen Modulabhängigkeitsgraphen ..... *Seite 116*
XP+MF ..... Schedulingalgorithmus für nicht formbare Module ohne Modulabhängigkeiten ..... *Seite 46*

8 Literaturverzeichnis

8.1 Vorliegendes Optimierungsproblem

[1] T. Rauber, and G. Rünger; Tree-based scheduling of data parallel modules for scientific computing; *Proceedings of 9th SIAM Conference on Parallel Processing for Scientific Computing* 1999

[2] R. Lepère, D. Trystram, and G. J. Woeginger; Approximation algorithms for scheduling malleable tasks under precedence constraints; *Proceedings of the 17th annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* 2005; Seiten 86 - 95

[3] A. Rădulescu, C. Nicolescu, A. J. C. van Gemund, and P. P. Jonker; Mixed task and data parallel scheduling for distributed systems; *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS)* 2001; Seiten 69 - 76

[4] R. Lepère, G. Mounié, D. Trystram, and B. Robic; Malleable Tasks: An efficient model for solving actual parallel applications; *Proceedings of Parallel Computing (ParCo)* 1999

[5] M. Drozdowski; Scheduling Multiprocessor Tasks - An Overview; *European Journal of Operational Research (EJOR)*; Vol. 94, 1996; Seiten 215 - 230

[6] K. P. Belkhale, and P. Banerjee; A scheduling algorithm for parallelizable dependent tasks; *Proceedings of the International Parallel Processing Symposium* 1991; Seiten 500 - 506

[7] J. Dümmler, R. Kunis, and G. Rünger; Layer-Based Scheduling Algorithms for Multiprocessor-Tasks with Precedence Constraints; *Proceedings of the Parallel Computing (ParCo)*; Advances in Parallel Computing, Band 15, IOS Press 2007; Seiten 321 - 328

[8] J. Dümmler, R. Kunis, and G. Rünger; A Comparison of Scheduling Algorithms for Multiprocessortasks with Precedence Constraints; *Proceedings of the High Performance Computing and Simulation Conference (HPCS)* 2007; Seiten 663 - 669

[9] K. Jansen, and H. Zhang; An Approximation Algorithm for Scheduling Malleable Tasks under General Precedence Constraints; *ACM Transactions on Algorithms*; Vol. 2, Nr. 3, Juli 2006; Seiten 416 - 434

[10] N. Vydyanathan, S. Krishnamoorty, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz; An Integrated Approach for Processor Allocation and Scheduling of Mixed - Parallel Applications; *Technical Report*; OSU-CISRC-2/06-TR20, 2006

[11] T. N'Takpé, F. Suter, and H. Casanova; A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms; *6th International Symposium on Parallel and Distributed Computing (ISPDC)* 2007; Seiten 250 - 257

8.2 Vorliegendes Optimierungsproblem mit speziellen Modulabhängigkeiten

[12] R. Lepère, G. Mounié, and D. Trystram; An approximation algorithm for scheduling trees of malleable tasks; *European Journal of Operational Research (EJOR)*; Vol. 142, 2002; Seiten 242 - 249

[13] T. Rauber, and G. Rünger; A transformation approach to derive efficient parallel implementations; *IEEE Transactions on Software Engineering*; Vol. 26, Nr. 4; April 2000; Seiten 315 - 339

[14] J. Blazewicz, Paolo dell'Olmo, and Maciej Drozdowski; Scheduling multiprocessor tasks on two parallel processors; *RAIRO - Operations Research (RO)*; Vol. 36, Nr. 1, 2002; Seiten 37 - 51

[15] A. Goldman, G. Mounié, and D. Trystram; 1-optimality of static BSP computations: Scheduling independent chains as a case study; *Theoretical Computer Science*; Vol. 290, Nr. 1, 2003; Seiten 1331 - 1359

[16] W. Lowe, and W. Zimmermann; Scheduling balanced task-graphs to LogP-machines; *Parallel Computing*; Vol. 26, 2000; Seiten 1083 - 1108

sowie in [2]

8.3 Vorliegendes Optimierungsproblem ohne Modulabhängigkeiten

[17] J. Turek, J. L. Wolf, and P. S. Yu; Approximate algorithms for scheduling parallelizable tasks; *Proceedings of the 4th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* 1992; Seiten 323 - 332

[18] J. Du, and J. Leung; Complexity of scheduling parallel task systems; *SIAM Journal on Discrete Mathematics*; Vol. 2, Nr. 4, November 1989; Seiten 473 - 487

[19] J. Turek, W. Ludwig, J. L. Wolf, L. Fleischer, P. Tiwari, J. Glasgow, U. Schwiegelshohn, and P. S. Yu; Scheduling parallelizable tasks to minimize average response time; *Proceedings of the 6th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* 1994; Seiten 200 - 209

[20] G. Mounié, C. Rapine, and D. Trystram; Efficient approximation algorithms for scheduling malleable tasks; *Proceedings of the 11th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* 1999; Seiten 23 - 32

[21] K. Jansen, and L. Porkolab; Linear - time approximation schemes for scheduling malleable parallel tasks; *Proceedings of 10th annual ACM - SIAM Symposium on Discrete Algorithms (SODA)* 1999; Seiten 490 - 498

[22] K. Jansen, and L. Porkolab; Preemptive parallel task scheduling in O(n) + poly(m) time; *Proceedings of the International Symposium on Algorithms and Computation (ISAAC)* 2000; Seiten 398 - 409

[23] K. Jansen, and L. Porkolab; Improved approximation schemes for scheduling unrelated parallel machines; *Proceedings of the 31st annual ACM Symposium on Theory of Computing (STOC)* 1999; Seiten 408 - 417

[24] K. Jansen; Scheduling malleable parallel tasks: An asymptotic fully polynomial time approximation scheme; *Algorithmica*; Vol. 39, Nr. 1, Januar 2004; Seiten 59 - 81

[25] J. Chen, and A. Miranda; A polynomial time approximation scheme for general multiprocessor job scheduling; *Proceedings of the 31st annual ACM Symposium on Theory of Computing (STOC)* 1999; Seiten 418 - 427

[26] G. Mounié, C. Rapine, and D. Trystram; A 3/2-dual approximation algorithm for scheduling independent monotonic malleable tasks; *SIAM Journal on Computing*; Vol. 37, Nr. 2, 2007; Seiten 401 - 412

[27] T. Decker, T. Lücking and B. Monien; A 5/4-approximation algorithm for scheduling identical malleable tasks; *Theoretical Computer Science*; Vol. 361, Nr. 2, 2006; Seiten 226 - 240

[28] P. Dutot, M. A. S. Netto, A. Goldman, and F. Kon; Scheduling Moldable BSP Tasks; *Proceedings of the 11th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) 2005*; Seiten 157 - 172

außerdem in [6]

8.4 Rechteck - Füll - Problem

[29] A. Steinberg; A strip - packing algorithm with absolute performance bound 2; *SIAM Journal on Computing*; Vol. 26, Nr. 2, 1997; Seiten 401 - 409

[30] K. Li; Analysis of an approximation algorithm for scheduling independent parallel tasks; *Diskrete Mathematics and Theoretical Computer Science* ; Vol. 3, Nr. 4, 1999; Seiten 155 - 166

[31] K. Li, and Y. Pan; Probabilistic analysis of scheduling precedence constrained parallel tasks on multicomputers with contiguous processor allocation; *IEEE Transactions on Computers*; Vol. 49, Nr. 10, Oktober 2000; Seiten 1021 - 1030

[32] E. B. Massimiliano; Scheduling of independent dedicated multiprocessor tasks; *Lecture Notes In Computer Science*; Vol. 2518, 2002; Seiten 391 - 402

[33] N. Lesh, J. Marks, A. McMahon, and M. Mitzenmacher; New heuristic and interactive approaches to 2D rectangular strip packing; *Proceedings of the IJCAI Workshop on Stochastic Search Algorithms 2003*; MERL Technical Report TR2003-18

8.5 Shop - Scheduling

[34] P. Brucker; *Scheduling Algorithms*; Springer 1998

[35] A. Darte, Y. Robert, and Frédéric Vivien; *Scheduling and Automatic Parallelization*; Birkhäuser 2000

[36] M. R. Garey, and D. S. Johnson; *Computers and Intractability – A Guide to the Theorie of NP - Completeness*; W.H. Freemann and Company 1979

[37] D. B. Shmoys, C. Stein, and J. Wein; Improved approximation algorithms for shop scheduling problems; *SIAM Journal on Computing*; Vol. 23, Nr. 3, 1994; Seiten 617 - 632

[38] K. K. Jain, and V. Rajaraman; Lower and upper bounds on time for multiprocessor optimal schedules; *IEEE Transactions on Parallel and Distributed Systems*; Vol. 5, Nr. 8, August 1994; Seiten 879 - 886

[39] K. Jansen, R. Solis-Oba, and M. Sviridenko; Makespan minimization in job shops: A polynomial time approximation scheme; *Proceedings of the 31st annual ACM Symposium on Theory of Computing (STOC)* 1999; Seiten 394 - 399

8.6 Scheduling zur Minimierung der durchschnittlichen Wartezeit

[40] F. Afrati, E. Bampis, A. V. Fishkin, K. Jansen, and C. Kenyon; Scheduling to minimize the average completion time of dedicated tasks; *Lecture Notes in Computer Science*; Vol. 1974, 2000; Seiten 454 - 464

[41] A. V. Fishkin, K. Jansen, and L. Porkolab; On minimizing average weighted completion time of multiprocessor tasks with release dates; *Lecture Notes in Computer Science*; Vol. 2076, 2001; Seiten 875 - 886

[42] A. V. Fishkin, and G. Zhang; On maximizing the throughput of multiprocessor tasks; *Theoretical Computer Science*; Vol. 302, Nr. 1, 2003; Seiten 319 - 335

[43] P.-F. Dutot, and D. Trystram; A best-compromise bicriteria scheduling algorithm for malleable tasks; *Workshop on Efficient Algorithms*; CTI Press, 2005

[44] T. N'Takpé, and F. Suter; Concurrent Scheduling of Parallel Task Graphs on Multi-Clusters Using Constrained Resource Allocations; *IEEE International Symposium on Parallel and Distributed Processing* 2009; Seiten 1 - 8

8.7 Weitere Schedulingverfahren

[45] M. R. Garey, and R. L. Graham; Bounds for multiprocessor scheduling with resource constraints; *SIAM Journal on Computing*; Vol. 4, Nr. 2, 1975; Seiten 187 - 200

[46] D. S. Hochbaum, and D. B. Shmoys; Using dual approximation algorithms for scheduling problems: theoretical and practical results; *Journal of the Association for Computing Machinery*; Vol. 34, Nr. 1; Januar 1987; Seiten 144 - 162

[47] P. Dutot, and D. Trystram; Scheduling on hierarchical clusers using malleable tasks; *Proceedings of the 13th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* 2001; Seiten 199 - 208

[48] A. Batat, and D. G. Feitelson; Gang scheduling with memory considerations; *Proceedings of the 14th International Symposium on Parallel and Distributed Processing* 2000; Seiten 109 - 114

[49] D. G. Feitelson; Packing schemes for gang scheduling; *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing* 1996; Seiten 89 - 110

[50] K. Jansen, and L. Porkolab; Preemptive scheduling on dedicated processors: Applications of fractional graph coloring; *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS)* 2000; Seiten 446 - 455

[51] T. Ma, and R. Buyya; Critical-Path and Priority Based Algorithms for Scheduling Workflows with Parameter Sweep Tasks on Global Grids; *Proceedings of the 17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* 2005; Seiten 251 - 258

[52] P.-F. Dutot, T. N'Takpé, F. Suter, and H. Casanova; Scheduling Parallel Task Graphs on (Almost) Homogeneous Multicluster Platforms; *IEEE Transactions on Parallel and Distributed Systems*; Vol. 20, Nr. 7, Juli 2009; Seiten 940 - 952

[53] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong; Theory and Practice in Parallel Job Scheduling; *Lecture Notes in Computer Science*; Vol. 1291 1997; Seiten 1 - 34

8.8 Arbeit mit Graphen und Suchalgorithmen

[54] Robert Sedgewick, and Phillippe Flajolet; *Analysis of Algorithms*; Addison - Wesley 1996

[55] Thomas H. Cormen, Charles E. Leiserson, Roland L. Rivest; *Introduction to Algorithms*; The MIT Press 1997

[56] Kurt Mehlhorn; *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*; Springer 1994

[57] Gerhard Goos, Wolf Zimmermann; *Vorlesungen über Informatik 1 – Grundlagen und funktionales Programmieren*; Springer 2005

[58] A. V. Aho, R. Sethi, J. D. Ullman; *Compilerbau (Teil 2)*; Addison Wesley 1995

8.9 Serien - Parallele Graphen

[59] J. Valdez, R. E. Tarjan, and E. L. Lawler; The recognition of series parallel graphs; *Proceedings of the 11th annual ACM Symposium on Theory of Computing (STOC)* 1979; Seiten 1 - 12

[60] M. Juvan, B. Mohar, and R. Thomas; List edge - colorings of series - parallel graphs; *Electronic Journal of Combinatorics*; Vol. 6, Nr. 1, 1999, Research Paper 42

[61] H. L. Bodlaender, and B. L. E. van Antwerpen - de Fluiter; Parallel algorithms for series parallel graphs; *Proceedings of the European Symposium on Algorithms (ESA)* 1996; Seiten 277 - 289

[62] J. Löwe; Zufälliges Erzeugen und Zeichnen serien - paralleler Graphen; *Projektarbeit im Studienfach Informatik an der Martin - Luther - Universität Halle - Wittenberg*; 1999

außerdem in [34]

9 Anhang - Zusatzinformationen

9.1 Minimumsuche in Linearzeit

Bei der Minimumsuche für Module mit einer Prozessoranzahl echt größer als Eins wird ein aus N Knoten bestehender Baum mit den folgenden Eigenschaften erstellt: Im linken (rechten) Unterbaum eines Knotens i befinden sich nur Knoten j (k) mit j < i (i < k). Außerdem gilt für alle Knoten j (k) die Beziehung E(j) ⩽ E(i) (E(k) < E(i)). Mit E(i) wird dabei die aktuelle Endzeit des Prozessors i bezeichnet (siehe Definition 5). Man erkennt, daß der Baum am Anfang (alle E(i) sind Null) zu einer nach links geneigten Kette entartet.

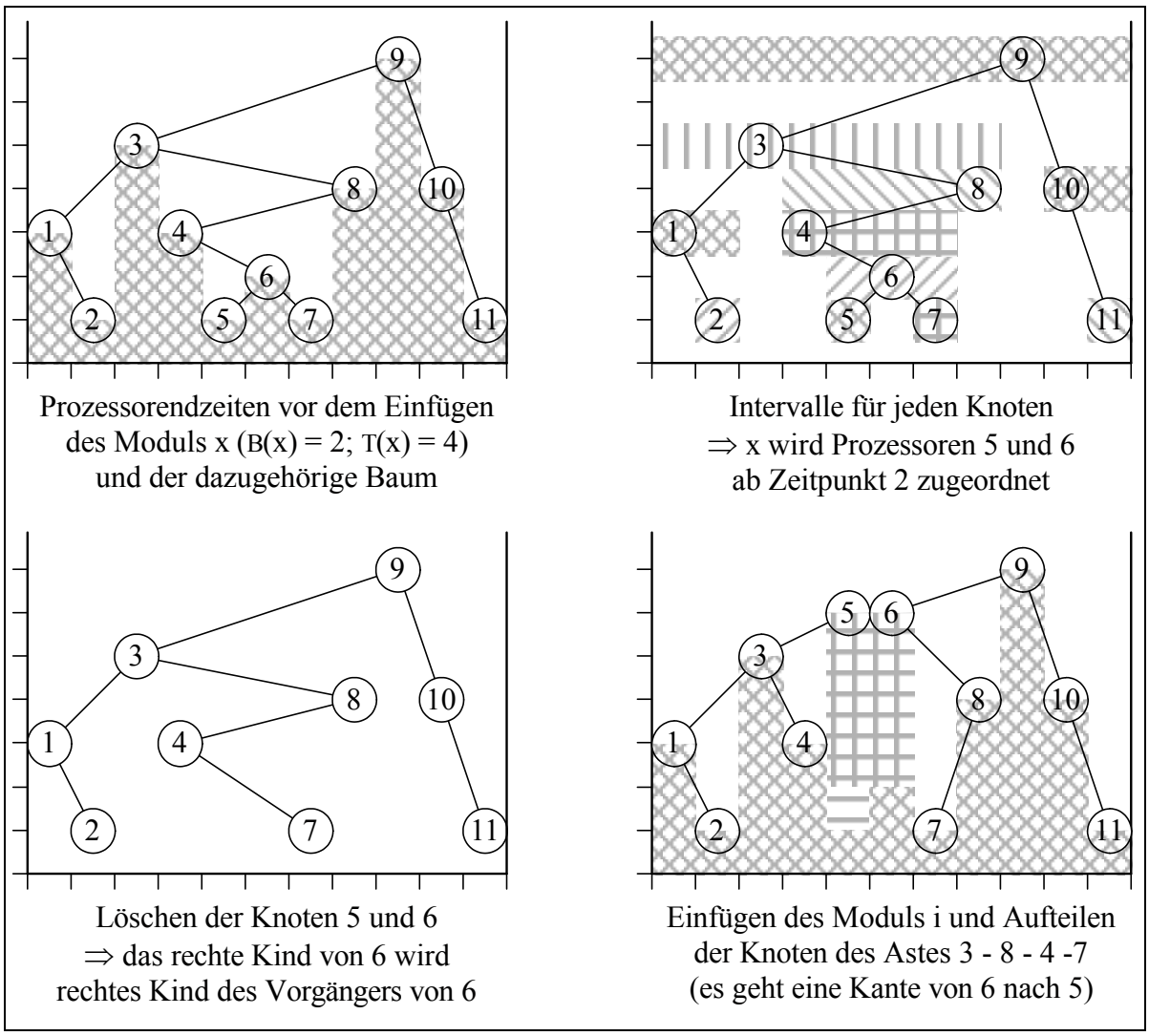


Abbildung 60 - Bestimmung der Einfügeposition in O(N).

Das Aufsuchen einer Einfügeposition geschieht mit Hilfe eines DFS - Laufes über den Baum. Dabei wird für jeden Knoten i ein Intervall [a, b] berechnet, welches unter der Bedingung, daß b - a + 1 ⩾ B(x) ist, das einzufügende Modul x ab dem Zeitpunkt E(i) aufnehmen kann. Für die Wurzel des Baumes ergibt sich das Intervall [1; N]. Wurde dem Knoten i das Intervall [a; b] zugeordnet, so lautet es für das linke (rechte) Kind [a; i - 1] ([i + 1; b]). Beim Durchlaufen des Baumes merkt man sich das Intervall mit der kleinsten Startzeit (bei gleicher Start-

zeit wählt man das weiter links liegende), in welches das einzufügende Modul x bzgl. seiner benötigten Prozessoranzahl B(x) paßt. In dieses Intervall wird x dann ganz links eingefügt und alle von x belegten Knoten k werden aus dem Baum gelöscht. Da die Knoten k nach dem Einfügen des Moduls x alle die gleiche Endzeit haben, bilden sie wiederum eine nach links geneigte Kette, welche mit einem Tiefenlauf in den Baum eingefügt wird. Dabei ist unter Umständen ein Aufteilen der Knoten unterhalb der Einfügeposition notwendig, damit die oben notierte Baumeigenschaft erhalten bleibt. Dieser Schritt kann aber ebenfalls in O(N) realisiert werden. Die Abbildung 60 veranschaulicht einen Arbeitsschritt des beschriebenen Einfügealgorithmus.

9.2 Beispiel für „HMZ“

In diesem Abschnitt wird ein Beispiel angeführt, bei dem der Algorithmus „HMZ“ zur Lösung des Rechteck - Füll - Problems nur den in Lemma 8 bewiesenen Gütefaktor von Drei erreicht.

Es seien $N \geqslant 3$ Prozessoren vorhanden. Außerdem sei k eine hinreichend große natürliche Zahl. Dann existieren $M = 2 \cdot k + 3$ Module mit den folgenden Prozessoranzahlen und Laufzeiten:

Modulnummer x	Prozessoranzahl B(x)	Modullaufzeit T(x)
$x = 1$	1	$k + \left\lceil \frac{k+1}{\left\lfloor \frac{N-1}{2} \right\rfloor} \right\rceil$
$x = 2$	N	1
$3 \leqslant x \leqslant 2 \cdot k + 3$; x ungerade	2	1
$4 \leqslant x \leqslant 2 \cdot k + 2$; x gerade	N - 1	1

Da bis auf das Modul 1 alle Module die gleiche Laufzeit haben, gehe man davon aus, daß sie entsprechend ihrer Modulnummer sortiert sind. Für $N = 5$ und $k = 3$ sind die Positionierungen in der Abbildung 61 dargestellt.

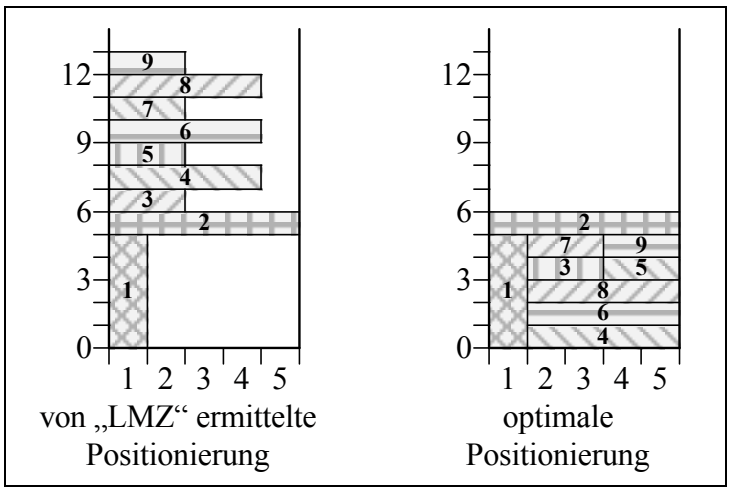


Abbildung 61 - schlechtes Beispiel beim Algorithmus „HMZ“.

Wie man erkennen kann, gibt es genau k Module mit der Prozessoranzahl N - 1. Sie liegen auch in der optimalen Positionierung übereinander. Von den k + 1 Modulen, welche zwei Prozessoren benötigen, können maximal $\left\lfloor \frac{N-1}{2} \right\rfloor$ Module in einer „Zeile“ mit der Laufzeit Eins und

der Prozessoranzahl N - 1 untergebracht werden. Insgesamt benötigen die Module 3 bis $2 \cdot k + 3$ eine Laufzeit von $k + \left\lceil \frac{k+1}{\lfloor \frac{N-1}{2} \rfloor} \right\rceil$ bei der Verwendung von N - 1 Prozessoren. Sie können also immer zeitgleich mit dem Modul 1 ausgeführt werden. Die Gesamtlaufzeit O der optimalen Positionierung beträgt somit

$$k + \left\lceil \frac{k+1}{\lfloor \frac{N-1}{2} \rfloor} \right\rceil + 1.$$

Die Gesamtlaufzeit der von „HMZ“ ermittelten Positionierung ist die Summe aller Modullaufzeiten, also

$$H = k + \left\lceil \frac{k+1}{\lfloor \frac{N-1}{2} \rfloor} \right\rceil + 1 + 2 \cdot k + 1.$$

Demzufolge beträgt der Gütefaktor

$$\begin{aligned} G &= \frac{H}{O} \\ &= \frac{k + \left\lceil \frac{k+1}{\lfloor \frac{N-1}{2} \rfloor} \right\rceil + 1 + 2 \cdot k + 1}{k + \left\lceil \frac{k+1}{\lfloor \frac{N-1}{2} \rfloor} \right\rceil + 1} \\ &= \frac{3 \cdot k + 2 + \left\lceil \frac{k+1}{\lfloor \frac{N-1}{2} \rfloor} \right\rceil}{k + 1 + \left\lceil \frac{k+1}{\lfloor \frac{N-1}{2} \rfloor} \right\rceil} \\ &\geqslant \frac{3 \cdot k + 2 + \frac{k+1}{\lfloor \frac{N-1}{2} \rfloor}}{k + 1 + \frac{k+1}{\lfloor \frac{N-1}{2} \rfloor} + 1} \\ &\geqslant \frac{3 \cdot k + 2 + \frac{k+1}{\frac{N-1}{2}}}{k + 2 + \frac{k+1}{\frac{N-1}{2} - \frac{1}{2}}} \\ &= \frac{k \cdot (3 \cdot N^2 - 7 \cdot N + 2) + 2 \cdot N^2 - 4 \cdot N}{k \cdot (N^2 - N) + 2 \cdot N^2 - 4 \cdot N + 2}. \end{aligned}$$

Läßt man k gegen unendlich gehen, so erhält man:

$$\lim_{k \to \infty} \frac{k \cdot (3 \cdot N^2 - 7 \cdot N + 2) + 2 \cdot N^2 - 4 \cdot N}{k \cdot (N^2 - N) + 2 \cdot N^2 - 4 \cdot N + 2} = \frac{3 \cdot N^2 - 7 \cdot N + 2}{N^2 - N} = 3 - \frac{2}{N} - \frac{2}{N - 1}.$$

Man kann sich also dem Gütefaktor von Drei beliebig nähern, sofern genug Prozessoren zur Verfügung stehen.
Dieses Beispiel zeigt außerdem sehr schön, daß man bei der Verwendung des Modulpositionierungsalgorithmus von „P1+“ (Zeilen 6) und 7) in Abbildung 6), welcher jedes Modul so

zeitig wie möglich startet, auch keine bessere Abschätzung vornehmen kann. Er würde hier die gleiche Positionierung liefern, wie das in „HMZ“ benutzte ebenenbasierte Verfahren.

9.3 Beispiel für „BMZ“

Zum Erstellen eines Beispiels mit einem Gütefaktor, welcher größer als 2,69 ist, betrachte man zunächst die Folge

$$a_1 = 1 \quad \text{und} \quad a_{i+1} = a_i \cdot (a_i + 1).$$

die ersten Glieder der soeben definierten Folge						
i	1	2	3	4	5	6
a_i	1	2	6	42	1806	3263442

Die Reihe der reziproken Werte der soeben definierten Folge bezeichne man mit

$$s_i = \sum_{j=1}^{i} \frac{1}{a_j}.$$

Wie man durch die Aufsummierung der ersten sechs Glieder von $\frac{1}{a_i}$ erkennen kann, gilt

$$s_\infty = \lim_{i \to \infty} s_i > 1{,}691.$$

Zur Berechnung einer oberen Schranke ist es notwendig, die folgende Eigenschaft mittels Induktion zu beweisen:

$$\frac{1}{a_{i+1}} = 1 - \sum_{j=1}^{i} \frac{1}{a_j + 1}. \tag{1}$$

Für i = 1 ist die Formel gültig. Im Induktionsschritt wird nun angenommen, daß sie für ein beliebiges aber festes i gilt. Man erhält also

$$\frac{1}{a_{i+1}} = 1 - \sum_{j=1}^{i} \frac{1}{a_j + 1}$$

$$\Leftrightarrow \quad \frac{1}{a_{i+1}} - \frac{1}{a_{i+1} + 1} = 1 - \sum_{j=1}^{i+1} \frac{1}{a_j + 1}$$

$$\Leftrightarrow \quad \frac{1}{a_{i+1} \cdot (a_{i+1} + 1)} = 1 - \sum_{j=1}^{i+1} \frac{1}{a_j + 1}$$

$$\Leftrightarrow \quad \frac{1}{a_{i+2}} = 1 - \sum_{j=1}^{i+1} \frac{1}{a_j + 1}.$$

■

Mit Hilfe von (1) ist es möglich

$$\sum_{j=1}^{\infty} \frac{1}{a_j + 1} = \lim_{i \to \infty} \sum_{j=1}^{i} \frac{1}{a_j + 1} = \lim_{i \to \infty} 1 - \frac{1}{a_{i+1}} = 1 \tag{2}$$

zu bestimmen. Nun kann man eine oberen Schranke von

$$
\begin{aligned}
s_\infty &= \left(\sum_{j=1}^{i-1} \frac{1}{a_j}\right) + \sum_{j=i}^{\infty} \frac{1}{a_j} \\
&= \left(\sum_{j=1}^{i-1} \frac{1}{a_j}\right) + \sum_{j=i}^{\infty} \frac{2}{a_j + a_j} \\
&\leqslant \left(\sum_{j=1}^{i-1} \frac{1}{a_j}\right) + 2 \cdot \sum_{j=i}^{\infty} \frac{1}{a_j + 1} \\
&= \left(\sum_{j=1}^{i-1} \frac{1}{a_j}\right) + 2 \cdot \left(\left(\sum_{j=1}^{\infty} \frac{1}{a_j + 1}\right) - \left(\sum_{j=1}^{i-1} \frac{1}{a_j + 1}\right)\right) \\
&= \left(\sum_{j=1}^{i-1} \frac{1}{a_j}\right) + 2 \cdot \left(1 - \sum_{j=1}^{i-1} \frac{1}{a_j + 1}\right)
\end{aligned}
$$

bestimmen. Setzt man i = 7, so ergibt sich die folgende Abschätzung:

$$s_\infty \leqslant 1{,}693.$$

Doch nun zum Beispiel, bei dem der Algorithmus „BMZ“ nur einen Gütefaktor von rund 2,69 erzielt. Es sei $N \geqslant 16$ und k die größte natürliche Zahl, so daß

$$1 + 2 \cdot k + \sum_{i=1}^{k} \left\lfloor \frac{N}{a_i + 1} \right\rfloor \leqslant N$$

gilt. Zusätzlich sei noch eine hinreichend kleine Zahl $\varepsilon > 0$ gegeben.
Das Modul mit der Nummer 1 benötige

$$N - 2 \cdot k - \sum_{i=1}^{k} \left\lfloor \frac{N}{a_i + 1} \right\rfloor$$

Prozessoren und habe eine Laufzeit von Eins. Außerdem existieren zu jeder natürlichen Zahl $j \in [1; k]$ noch die folgenden Module, wobei alle Module mit der gleichen Laufzeit und der gleichen Prozessoranzahl zu einer Modulgruppe zusammengefaßt werden:

Modullaufzeiten und Prozessoranzahlen			
Modulgruppe	Modulanzahl	Laufzeit $T(x)$	Prozessoranzahl $B(x)$
A_j	a_j	$\frac{1}{a_j}$	$\left\lfloor \frac{N}{a_j + 1} \right\rfloor + 2$
B_j	$N - \left\lfloor \frac{N}{a_j + 1} \right\rfloor \cdot (a_j + 1)$	ε	$\left\lfloor \frac{N}{a_j + 1} \right\rfloor + 1$
C_j	$\left\lfloor \frac{N}{a_j + 1} \right\rfloor \cdot (a_j + 1) + a_j + 1 - N$	ε	$\left\lfloor \frac{N}{a_j + 1} \right\rfloor$

Die Abbildungen 62 und 63 zeigen die Wahl der oben definierten Modullaufzeiten und Prozessoranzahlen am Beispiel $N = 300$. Wie man ausrechnen kann, muß $k = 3$ gesetzt werden. Die Zahl in den Rechtecken gibt diesmal nicht die Modulnummer sondern die Prozessoranzahl des Moduls an.

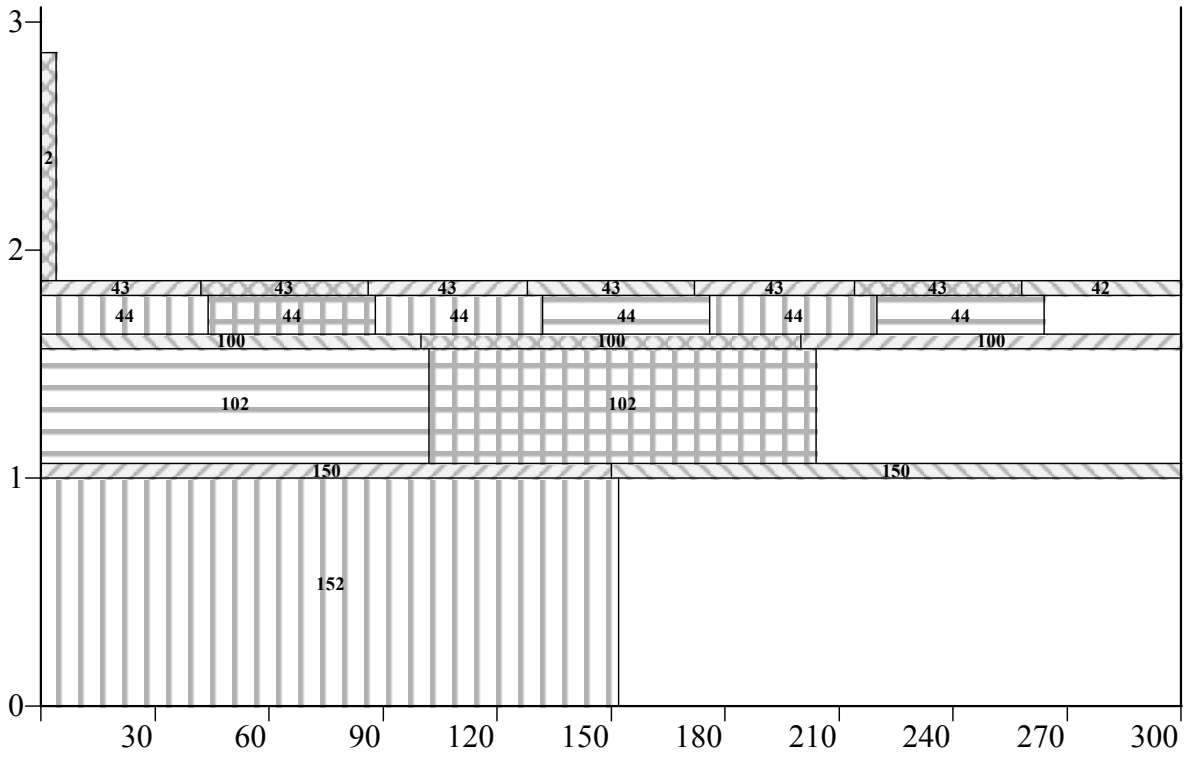


Abbildung 62 - Beispiel, bei welchem der Algorithmus „BMZ“ nur einen Gütefaktor von 2,69 erzielt.

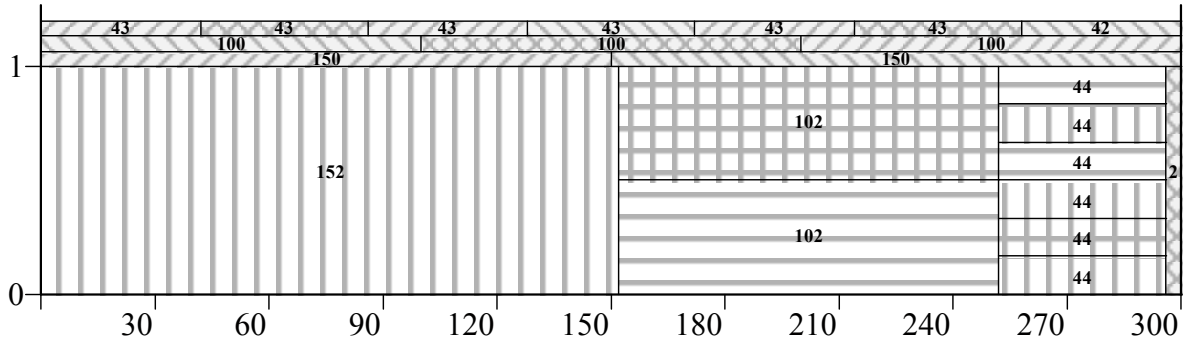


Abbildung 63 - Optimale Positionierung des Beispiels, bei welchem der Algorithmus „BMZ“ nur einen Gütefaktor von 2,69 erzielt.

Zur Berechnung der Gesamtlaufzeit H der von „BMZ“ berechneten Positionierung muß man die Struktur der Positionierung analysieren. Dazu benötigt man eine Folgerung aus der Wahl von k:

$$N \geqslant 1 + 2 \cdot k + \sum_{i=1}^{k} \left\lfloor \frac{N}{a_i + 1} \right\rfloor$$

$$\geqslant 1 + k + \sum_{i=1}^{k} \frac{N}{a_i + 1}$$

$$= 1 + k + N \cdot \sum_{i=1}^{k} \frac{1}{a_i + 1}$$

$$= 1 + k + N \cdot \left(1 - \frac{1}{a_{k+1}}\right) \quad \text{(wegen (1))}.$$

Durch Umstellen erhält man

$$(1 + k) \cdot a_{k+1} \leqslant N.$$

Es sei $j \in [1; k]$. Dann kann man wegen $k \geqslant 1$ und $a_{j+1} \leqslant a_{k+1}$ die Ungleichung

$$2 \cdot a_{j+1} \leqslant N \tag{3}$$

folgern.
Zur Analyse der Struktur der von „BMZ“ erstellten Positionierung wird zuerst die Positionierungsreihenfolge der Module bestimmt. Diese ist durch die Prozessoranzahlen der Module gegeben. Man erkennt sofort, daß für ein festes $j \in [1; k]$ jedes Modul in der Modulgruppe A_j mehr Prozessoren benötigt als ein Modul in der Modulgruppe B_j und jedes Modul in der Modulgruppe B_j wiederum mehr Prozessoren benötigt als ein Modul in der Modulgruppe C_j. Als nächstes wird gezeigt, daß für alle $j \in [1; k - 1]$ jedes Modul in der Modulgruppe C_j mehr Prozessoren benötigt als ein Modul in der Modulgruppe A_{j+1}. Diese Aussage entspricht der Ungleichung

$$\left\lfloor \frac{N}{a_{j+1} + 1} \right\rfloor + 2 < \left\lfloor \frac{N}{a_j + 1} \right\rfloor.$$

Zum Beweis dieser Ungleichung müssen die folgenden Fälle unterschieden werden:
Fall 1: Es sei $j = 1$.
Wegen $15 < N$ gilt

$$2 \cdot N + 12 < 3 \cdot N - 3 \quad \Leftrightarrow \quad \frac{N}{3} + 2 < \frac{N - 1}{2} \quad \Rightarrow \quad \left\lfloor \frac{N}{a_2 + 1} \right\rfloor + 2 < \left\lfloor \frac{N}{a_1 + 1} \right\rfloor.$$

Fall 2: Es sei $j = 2$.
Aus $14 < N$ folgt

$$3 \cdot N + 42 < 7 \cdot N - 14 \quad \Leftrightarrow \quad \frac{N}{7} + 2 < \frac{N - 2}{3} \quad \Rightarrow \quad \left\lfloor \frac{N}{a_3 + 1} \right\rfloor + 2 < \left\lfloor \frac{N}{a_2 + 1} \right\rfloor.$$

Fall 3: Es sei $j \geqslant 3$.
Mit Hilfe der Ungleichung (3) erhält man

$$2 \cdot \left(a_j + 1\right) \cdot a_j \leqslant N$$

$$\Leftrightarrow \qquad 2 \cdot \frac{a_j^4 + a_j^3}{a_j^2} \leqslant N$$

$$\Rightarrow \qquad 3 \cdot \frac{a_j^3 + 2 \cdot a_j^2 + 2 \cdot a_j + 1}{a_j^2} \leqslant N \quad \left(\text{da } j \geqslant 3 \text{ und somit } a_j \geqslant 6\right)$$

$$\Leftrightarrow \qquad 3 \cdot \frac{((a_j + 1) \cdot a_j + 1) \cdot (a_j + 1)}{a_j^2} \leqslant N$$

$$\Leftrightarrow \qquad 3 \leqslant \frac{N \cdot a_j^2 + N \cdot a_j + N - N \cdot a_j - N}{((a_j + 1) \cdot a_j + 1) \cdot (a_j + 1)}$$

$$\Leftrightarrow \qquad \frac{N}{(a_j + 1) \cdot a_j + 1} + 2 \leqslant \frac{N}{a_j + 1} - 1$$

$$\Rightarrow \qquad \left\lfloor \frac{N}{a_{j+1} + 1} \right\rfloor + 2 < \left\lfloor \frac{N}{a_j + 1} \right\rfloor.$$

Es verbleibt noch das Modul mit der Nummer 1, welches

$$N - 2 \cdot k - \sum_{i=1}^{k} \left\lfloor \frac{N}{a_i + 1} \right\rfloor$$

Prozessoren benötigt. Es wird gezeigt, daß das Modul 1 weniger Prozessoren benötigt als alle anderen Module, d.h., es benötigt weniger Prozessoren als ein Modul in der Modulgruppe C_k:

$$N - 2 \cdot k - \sum_{i=1}^{k} \left\lfloor \frac{N}{a_i + 1} \right\rfloor < \left\lfloor \frac{N}{a_k + 1} \right\rfloor. \tag{4}$$

Dazu betrachte man

$$\underbrace{1 - k}_{\leqslant 0} \leqslant \underbrace{N \cdot \frac{a_k - 1}{a_k \cdot (a_k + 1)}}_{\geqslant 0}$$

$$\Leftrightarrow \qquad N \cdot \frac{1}{a_k \cdot (a_k + 1)} - k \leqslant \frac{N}{a_k + 1} - 1$$

$$\Leftrightarrow \qquad N \cdot \left(1 - \sum_{i=1}^{k} \frac{1}{a_i + 1} \right) - k \leqslant \frac{N}{a_k + 1} - 1 \quad \text{(siehe (1))}$$

und folgere die Ungleichung (4).

Im nächsten Schritt der Analyse der Struktur der Positionierung wird gezeigt, daß für alle $j \in [1; k]$ die Summe der Prozessoranzahlen aller Module in den Modulgruppen B_j und C_j gleich N ist:

$$\underbrace{\left(N - \left\lfloor \frac{N}{a_j + 1} \right\rfloor \cdot (a_j + 1) \right)}_{\text{Modulanzahl in } B_j} \cdot \underbrace{\left(\left\lfloor \frac{N}{a_j + 1} \right\rfloor + 1 \right)}_{\text{Proz.anzahl}} + \underbrace{\left(\left\lfloor \frac{N}{a_j + 1} \right\rfloor \cdot (a_j + 1) + a_j + 1 - N \right)}_{\text{Modulanzahl in } C_j} \cdot \underbrace{\left\lfloor \frac{N}{a_j + 1} \right\rfloor}_{\text{Proz.anzahl}}$$

$$= N - \left\lfloor \frac{N}{a_j + 1} \right\rfloor \cdot (a_j + 1) + (a_j + 1) \cdot \left\lfloor \frac{N}{a_j + 1} \right\rfloor$$

$$= N.$$

Man definiere

$$e(j) = s_j + j \cdot \varepsilon = \left(\sum_{i=1}^{j} \frac{1}{a_i} \right) + j \cdot \varepsilon.$$

Jetzt wird mittels Induktion über j gezeigt, daß in der von „BMZ“ berechneten Positionierung alle Module der Modulgruppen B_j und C_j zum Zeitpunkt e(j) beendet werden.

Induktionsvoraussetzung: Es sei $j = 0$.

Da die Module der Modulgruppe A_1 die größte Prozessoranzahl haben, wurden vor diesen Modulen keine anderen Module positioniert und alle Prozessoren des Parallelrechners haben die Endzeit $0 = e(0)$.

Induktionsschritt: Es gelte $j \geqslant 1$.
Da die Summe der Prozessoranzahlen aller Module in den Modulgruppen B_{j-1} und C_{j-1} gleich N ist und alle Module in den Modulgruppen B_{j-1} und C_{j-1} nach Induktionsvoraussetzung zum Zeitpunkt $e(j-1)$ beendet werden, haben vor der Positionierung eines Moduls aus A_j alle Prozessoren des Parallelrechners die aktuelle Endzeit $e(j-1)$.
Es wird gezeigt, daß alle Module der Modulgruppe A_j zum Zeitpunkt $e(j-1)$ gestartet werden können, da sie insgesamt maximal N Prozessoren benötigen. Aus (3) folgt:

$$2 \cdot a_j \leqslant \frac{N}{a_j + 1}$$

$$\Leftrightarrow \quad 2 \cdot a_j + \frac{N \cdot a_j}{a_j + 1} \leqslant N$$

$$\Rightarrow \quad \underbrace{a_j}_{\text{Modulanzahl}} \cdot \underbrace{\left(\left\lfloor \frac{N}{a_j + 1} \right\rfloor + 2 \right)}_{\text{Proz.anzahl}} \leqslant N$$

Die Module der Modulgruppe A_j sind demzufolge zum Zeitpunkt $e(j-1) + \frac{1}{a_j}$ beendet.
Jetzt wird gezeigt, daß weder ein Modul der Modulgruppe B_j noch ein Modul der Modulgruppe C_j zum Zeitpunkt $e(j-1)$ gestartet werden kann, da nicht mehr genügend Prozessoren zur Ausführung des Moduls zur Verfügung stehen.
Aus $a_j \geqslant 1$ folgt

$$\frac{N}{a_j + 1} \cdot \left(a_j + 1 \right) + a_j - 1 \geqslant N$$

$$\Leftrightarrow \quad a_j \cdot \left(\frac{N}{a_j + 1} + 1 \right) + \frac{N}{a_j + 1} - 1 \geqslant N$$

$$\Rightarrow \quad \underbrace{a_j \cdot \left(\left\lfloor \frac{N}{a_j + 1} \right\rfloor + 2 \right)}_{\substack{\text{von den Modulen in } A_j \\ \text{belegte Prozessorenanzahl}}} + \underbrace{\left\lfloor \frac{N}{a_j + 1} \right\rfloor}_{\substack{\text{von einem Modul in } C_j \\ \text{benötigte Prozessorenanzahl}}} > N.$$

Demzufolge werden alle Module der Modulgruppen B_j und C_j zum Zeitpunkt $e(j-1) + \frac{1}{a_j}$ gestartet und sind somit zum Zeitpunkt

$$e(j-1) + \frac{1}{a_j} + \varepsilon = e(j)$$

beendet.
Damit wäre die Induktion beendet.

Man sieht nun, daß das Modul mit der Nummer 1 zum Zeitpunkt $e(k)$ gestartet wird. Demzufolge beträgt die Gesamtlaufzeit H der von „BMZ“ berechneten Positionierung

$$H = e(k) + 1 = s_k + k \cdot \varepsilon + 1.$$

In der optimalen Positionierung werden für jedes $j \in [1; k]$ alle Module der Modulgruppe A_j nacheinander auf den Prozessoren

$$\left(\sum_{i=1}^{j-1}\left\lfloor\frac{N}{a_i+1}\right\rfloor+2\right)+1 \quad \text{bis} \quad \left(\sum_{i=1}^{j}\left\lfloor\frac{N}{a_i+1}\right\rfloor+2\right)$$

ausgeführt. Die Wahl von k garantiert, daß auf den verbleibenden Prozessoren noch das Modul 1 ausgeführt werden kann. Die Nacheinanderausführung aller Module einer Modulgruppe A_j benötigt eine Zeiteinheit. Da das Modul 1 ebenfalls eine Zeiteinheit benötigt, sind alle Prozessoren bis zum Zeitpunkt 1 ausgelastet. Wie man bereits gesehen hat, benötigen die Module in den Modulgruppen B_j und C_j bei der Ausführung auf allen Prozessoren für jedes $j \in [1; k]$ exakt ε Zeiteinheiten. Für die Gesamtlaufzeit O der optimalen Positionierung ergibt sich somit

$$O = 1 + k \cdot \varepsilon.$$

Insgesamt erhält man einen Gütefaktor von

$$G = \frac{s_k + k \cdot \varepsilon + 1}{1 + k \cdot \varepsilon}.$$

Für $\varepsilon \to 0$ geht dieser Wert gegen $1 + s_k$. Da mit steigendem N auch k größer wird (siehe (2)), liegt der Grenzwert des Gütefaktors bei $1 + s_\infty > 2{,}69$.

9.4 Sortierung nach der Modulfläche

Das in diesem Abschnitt angegebene Beispiel zeigt, daß beim Rechteck - Füll - Problem eine Sortierung nach der Modulfläche nicht sinnvoll ist. Es existieren die folgenden $2 \cdot N - 1$ Module: Ist die Modulnummer ungerade, so habe das Modul eine Laufzeit von N und eine Prozessoranzahl von Eins. Anderenfalls benötige es N Prozessoren und habe eine Laufzeit von Eins. Da die Module alle die gleiche Fläche belegen, entscheide die Modulnummer über die Positionierungsreihenfolge. Für $N = 5$ entsteht das in der Abbildung 64 dargestellte Bild.

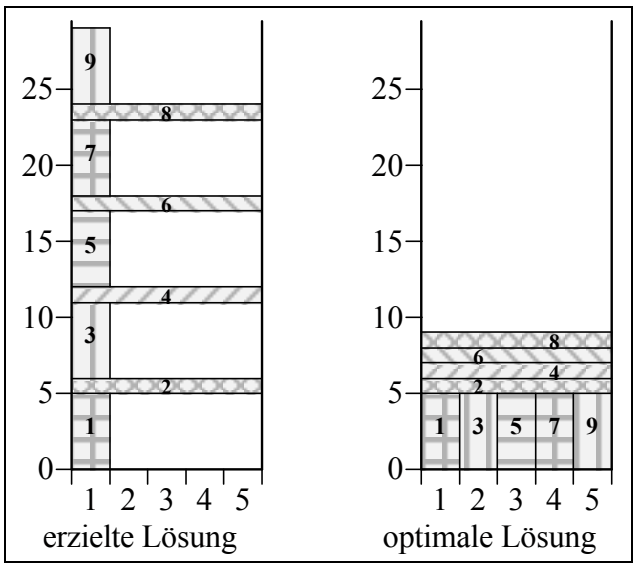


Abbildung 64 - Beispiel, welches zeigt, daß eine Sortierung nicht der Modulfläche nicht sinnvoll ist.

Wie man nachvollziehen kann, beträgt in diesem Beispiel der Gütefaktor

$$G = \frac{H}{O} = \frac{N^2 + N - 1}{N + N - 1} = \frac{N}{2} + \frac{3}{4} - \frac{1}{8 \cdot N - 4} \geqslant \tfrac{1}{2} \cdot N + \tfrac{1}{2}.$$

9.5 Kombination aus den Algorithmen „HMZ“ und „BMZ“

Betrachtet man den Beweis zur Ergebnisgüte des Algorithmus „HMZ“ (Lemma 7), so erkennt man, daß man mit einer Zusatzvoraussetzung eine bessere Abschätzung liefern kann. Dazu definiere man

$$z = \max_{x=1}^{M} \{ B(x) \}.$$

Demzufolge sind in jeder Ebene mindestens $N - z + 1$ Prozessoren aktiv. Ohne Beschränkung der Allgemeinheit seien auch hier die Module absteigend nach ihrer Laufzeit sortiert, also aus $x < y$ folgt $T(x) \geqslant T(y)$. Sollte nun $z \leqslant ½ \cdot N$ sein, so kann analog zum Beweis von Lemma 7 die folgende Ungleichung hergeleitet werden:

$$\sum_{x=1}^{M} T(x) \cdot B(x) > (N - z + 1) \cdot (H - T(1)).$$

Daraus folgt

$$O \cdot N > (N - z + 1) \cdot (H - T(1)).$$

Lemma 34: *Es bezeichne N die Anzahl aller Prozessoren. Benötigt bei der Verwendung des Algorithmus „HMZ“ kein Modul mehr als $z \leqslant ½ \cdot N$ Prozessoren, so erzielt „HMZ“ einen Gütefaktor von $G < \frac{N}{N - z + 1} + 1$.*

Beweis:

Fall 1: Es gelte $\left(\frac{N}{N - z + 1} + 1\right) \cdot T(1) > H$.

Da $O \geqslant T(1)$ gilt, folgt die Behauptung.

Fall 2: Es sei $\left(\frac{N}{N - z + 1} + 1\right) \cdot T(1) \leqslant H$.

Die Abschätzung des Gütefaktors ergibt hier:

$$G = \frac{H}{O} < \frac{H}{\frac{(N - z + 1) \cdot (H - T(1))}{N}}$$

$$= \frac{H \cdot N}{(H - T(1)) \cdot (N - z + 1)} \leqslant \frac{H \cdot N}{\left(H - \frac{H}{\frac{N}{N - z + 1} + 1}\right) \cdot (N - z + 1)} \quad \text{(siehe Fallvoraussetzung)}$$

$$= \frac{N}{\left(1 - \frac{N - z + 1}{2 \cdot N - z + 1}\right) \cdot (N - z + 1)} = \frac{N}{\frac{N}{2 \cdot N - z + 1} \cdot (N - z + 1)}$$

$$= \frac{2 \cdot N - z + 1}{N - z + 1} = \frac{N}{N - z + 1} + 1.$$

■

Der Algorithmus „BHMZ“ positioniert am Anfang alle Module, welche mehr als z Prozessoren benötigen, mit Hilfe des Algorithmus „BMZ“. Die Laufzeit der mittels „BMZ“ erzielten Teilpositionierung aller Module x mit $B(x) > z$ betrage H_b. Dabei sind, selbst wenn man alle Module nacheinander ausführt, zu jedem Zeitpunkt mindestens $z + 1$ Prozessoren belegt. Da die gleiche Fläche auch in der optimalen Positionierung benötigt wird, gelangt man zu einem Gütefaktor von $\frac{N}{z + 1}$.

Anschließend positioniert „BHMZ“ alle Module x mit $B(x) \leqslant z$ mit dem Algorithmus „HMZ“ ab dem Zeitpunkt H_b. Der Gesamtgütefaktor G wird dann vom Gütefaktor der schlechtesten Teilpositionierung dominiert. Er liegt somit maximal bei

$$G \leqslant \max\left\{ \frac{N}{z+1}; \frac{N}{N-z+1} + 1 \right\}.$$

Dieser Schritt ist aber nur dann korrekt, wenn bei beiden Verfahren die Flächenbedingung zur Berechnung des Gütefaktors verwendet wird. Das trifft allerdings nicht zu, wenn im Beweis von Lemma 34 der Fall 1 eintritt. In der Abbildung 65 ist ein solches Beispiel dargestellt. Wenn im Lemma 34 der Fall 2 verwendet wird, kann man mittels

$$\frac{N}{z+1} = \frac{N}{N-z+1} + 1$$

einen optimalen Wert für z berechnen:

$$z_{1/2} = \frac{3}{2} \cdot N \pm \sqrt{\frac{5}{4} \cdot N^2 + N + 1}.$$

Da $z \leqslant N$ gelten muß, ist das „+“ eine Scheinlösung. Es ergibt sich also

$$z = \frac{3}{2} \cdot N - \sqrt{\frac{5}{4} \cdot N^2 + N + 1}.$$

Diese Lösung ist zulässig, weil z immer im Intervall $[0; ½ \cdot N]$ liegt. Nun kann man den Gütefaktor durch

$$G = \frac{N}{z+1} = \frac{N}{\frac{3}{2} \cdot N - \sqrt{\frac{5}{4} \cdot N^2 + N + 1} + 1} = \frac{N}{\frac{3}{2} \cdot N + 1 - \sqrt{\left(\frac{\sqrt{5}}{2} \cdot N + 1\right)^2 - (\sqrt{5} - 1) \cdot N}}$$

$$\leqslant \frac{N}{\frac{3}{2} \cdot N + 1 - \sqrt{\left(\frac{\sqrt{5}}{2} \cdot N + 1\right)^2}} = \frac{N}{\frac{3}{2} \cdot N - \frac{\sqrt{5}}{2} \cdot N} = \frac{3 + \sqrt{5}}{2} < 2{,}62$$

abschätzen.

Lemma 35: *Es bezeichne N die Anzahl aller Prozessoren und T(x) die Laufzeit des Moduls x. Außerdem setze man*

$$z = \frac{3}{2} \cdot N - \sqrt{\frac{5}{4} \cdot N^2 + N + 1}.$$

Mit der Nummer 1 werde das Modul bezeichnet, welches die größte Laufzeit hat. Dann folgt aus

$$\left(\frac{N}{N-z+1} + 1\right) \cdot T(1) \leqslant H,$$

daß der Algorithmus „BHMZ“ einen Gütefaktor kleiner als 2,62 erzielt.

Abschließend wird noch ein Beispiel angegeben, bei welchem der zusammengesetzte Algorithmus aufgrund der Beziehung

$$\left(\frac{N}{N-z+1} + 1\right) \cdot T(1) > H$$

die eben ermittelte Schranke nicht einhält, obwohl z gemäß Lemma 35 gewählt wird. Es sei $N = 38 + k$ mit $k \geqslant 0$. Setzt man diese Beziehung nun in die Berechnungsvorschrift für z ein, so erhält man:

$$z = ½ \cdot \left(3 \cdot k + 114 - \sqrt{5 \cdot k^2 + 384 \cdot k + 7376}\right) = \frac{1}{2} \cdot \left(3 \cdot k + 114 - \sqrt{\left(\sqrt{5} \cdot k + \frac{192}{\sqrt{5}}\right)^2 + \frac{16}{5}}\right)$$

$$< \frac{1}{2} \cdot \left(3 \cdot k + 114 - \sqrt{\left(\sqrt{5} \cdot k + \frac{192}{\sqrt{5}} \right)^2} \right) = \frac{1}{2} \cdot \left(\left(3 - \sqrt{5} \right) \cdot k + 114 - \frac{192}{\sqrt{5}} \right)$$

$$< ½ \cdot (k + 29) = ½ \cdot (N - 9).$$

Es werden also alle Module, die mindestens $\left\lfloor \frac{N}{2} \right\rfloor - 3$ Prozessoren $\left(\left\lfloor \frac{N}{2} \right\rfloor - 3 \geqslant z + 1 \right)$ benötigen, vom Algorithmus „BMZ“ behandelt. Die Modulgrößen sind aus der folgenden Tabelle entnehmbar. Dabei sei $\varepsilon > 0$ wie immer eine hinreichend kleine Zahl.

Modullaufzeiten		
Modulnummer	Prozessoranzahl	Laufzeit
1	$\left\lfloor \frac{N}{2} \right\rfloor + 2$	1
2	$\left\lceil \frac{N}{2} \right\rceil$	ε
3	$\left\lfloor \frac{N}{2} \right\rfloor$	ε
4	$\left\lfloor \frac{N}{2} \right\rfloor - 3$	1
5	1	1

Für $N = 38$ entsteht das in der Abbildung 65 dargestellte Bild.

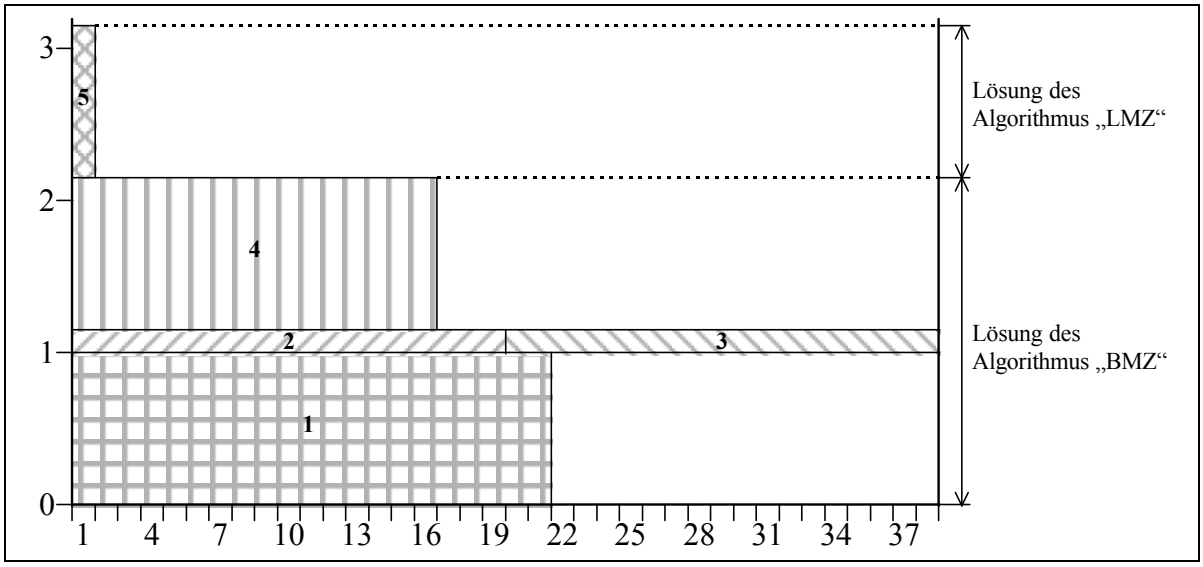


Abbildung 65 - Beispiel, bei dem die Flächenbedingung nicht in beiden Teilalgorithmen angewendet wird.

Wie man sieht, entsteht bei diesem Beispiel ein Gütefaktor von $\frac{3 + \varepsilon}{1 + \varepsilon}$. Durch die Wahl von ε kann man sich also beliebig dem Wert 3 nähern.

Lemma 36: *Mit N werde die Anzahl aller Prozessoren bezeichnet. Dann erzielt das kombinierte Verfahren „BHMZ“ einen garantierten Gütefaktor G echt kleiner als Drei, sofern $z \geqslant \lceil ½ \cdot N \rceil$ gesetzt wird. Ist $z < \lceil ½ \cdot N \rceil$, so erhält man*

$$G < 5 - \frac{4 \cdot z}{N}.$$

Beweis: Im Beweis stehe H_h (H_b) für die Laufzeit der Teilpositionierung, welche der Algorithmus „HMZ“ („BMZ“) erzielt hat. Die Gesamtlaufzeit H der Positionierung läßt sich demzufolge als $H = H_h + H_b$ darstellen. Mit i werde die Laufzeit des Moduls, welches die größte Laufzeit hat und nicht mehr als z Prozessoren (hier sei ohne Beschränkung der Allgemeinheit $z \in \mathbb{N}$) benötigt, angegeben, also

$$i = \max_{x=1}^{M} \{ T(x) : B(x) \leqslant z \}.$$

Wie man bereits im Beweis zum Gütefaktor des Algorithmus „BMZ“ gesehen hat, gibt es in der von diesem Algorithmus ermittelten Teilpositionierung einen Zeitpunkt $H_b - j$ (im Fall 1 des dortigen Beweises ist j gleich Null), ab welchem weniger als $½ \cdot N$ Prozessoren aktiv sind. Da alle mittels „BMZ“ zu positionierenden Module x die Mindestprozessoranzahl z + 1 ($B(x) \geqslant z + 1$) haben, sind auch im Intervall $[H_b - j; H_b)$ wenigstens z + 1 Prozessoren aktiv. Die Variable k stehe für die maximale Laufzeit aller Module:

$$k = \max_{x=1}^{M} \{ T(x) \}.$$

Man erkennt, daß $k \geqslant i$ gilt. Aber es ist auch $k \geqslant j$. Dazu betrachte man nochmals den im Abschnitt 2.1.4.2 angegebenen Beweis zur Ergebnisgüte des Algorithmus „BMZ“. Dort wurde gezeigt, daß nach dem Zeitpunkt $H_b - j$ kein neues Modul mehr gestartet wird.

Die Abbildung 66 stellt noch einmal die Bedeutung der einzelnen Variablen dar.

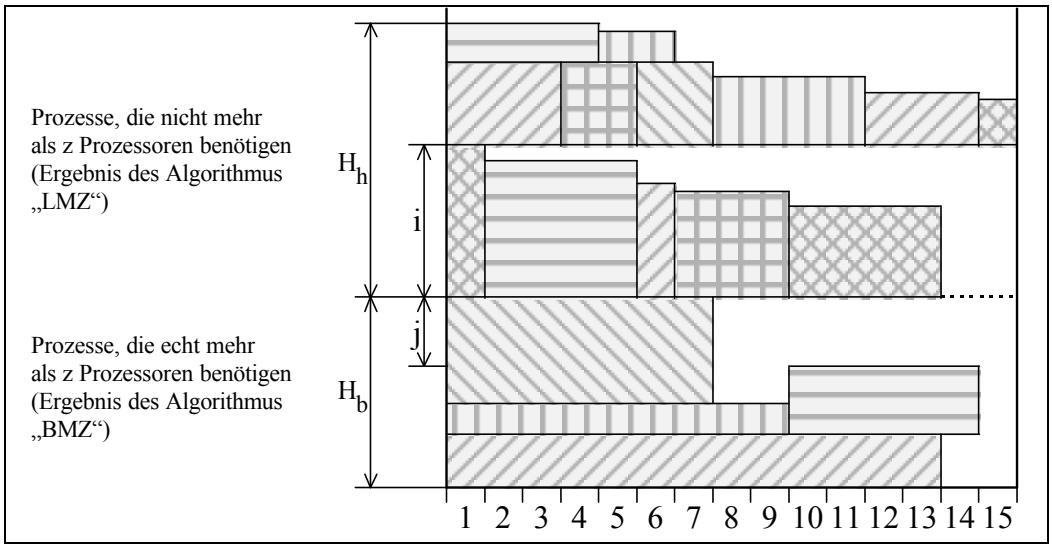


Abbildung 66 - verwendete Bezeichnungen.

Fall 1: Es sei $½ \cdot N - 1 \geqslant z$.

Fall 1.1: Es gelte $\left(5 - \frac{4 \cdot z}{N}\right) \cdot k > H$.

Wegen $O \geqslant k$ folgt die Behauptung.

Fall 1.2: Es sei $\left(5 - \frac{4 \cdot z}{N}\right) \cdot k \leqslant H$.

Die Mindestfläche A der optimalen Positionierung läßt sich mittels

$$A = \underbrace{(H_h - i) \cdot (N - z + 1)}_{\substack{\text{Summe der belegten} \\ \text{Fläche (bis auf oberste} \\ \text{Ebene) bei „HMZ“}}} + \underbrace{j \cdot (z + 1)}_{\substack{\text{belegte Fläche} \\ \text{bei „BMZ“ ab} \\ \text{Zeitpunkt } H_b - j}} + \underbrace{(H_b - j) \cdot (½ \cdot N)}_{\substack{\text{belegte Fläche} \\ \text{bei „BMZ“ bis zum} \\ \text{Zeitpunkt } H_b - j}}$$

$$= H_h \cdot (N - z + 1) + H_b \cdot ½ \cdot N - i \cdot (N - z + 1) - j \cdot (½ \cdot N - z - 1)$$

abschätzen. Wegen $i \leqslant k$, $j \leqslant k$, $N - z + 1 \geqslant 0$ und $½ \cdot N - z - 1 \geqslant 0$ kann man die Abschätzung mit

$$A \geqslant H_h \cdot (N - z + 1) + H_b \cdot ½ \cdot N - k \cdot (N - z + 1) - k \cdot (½ \cdot N - z - 1)$$

$$\geqslant H_h \cdot (N - z + 1) + H_b \cdot \frac{1}{2} \cdot N - \frac{H}{5 - \frac{4 \cdot z}{N}} \cdot \left(\frac{3}{2} \cdot N - 2 \cdot z\right) \quad \text{(siehe Fallvoraussetzung)}$$

$$\geqslant H_h \cdot \left(N - \left(\frac{1}{2} \cdot N - 1\right) + 1\right) + H_b \cdot \frac{1}{2} \cdot N - \frac{H \cdot N}{5 \cdot N - 4 \cdot z} \cdot \left(\frac{3}{2} \cdot N - 2 \cdot z\right)$$

$$> \left(H_h + H_b\right) \cdot \frac{1}{2} \cdot N - \frac{H \cdot N}{5 \cdot N - 4 \cdot z} \cdot \left(\frac{3}{2} \cdot N - 2 \cdot z\right)$$

$$= \frac{H \cdot N}{5 \cdot N - 4 \cdot z} \cdot N$$

fortsetzen. Als Obergrenze für den Gütefaktor ergibt sich somit

$$G = \frac{H}{O} \leqslant \frac{H \cdot N}{A} \leqslant \frac{N \cdot H}{\frac{N \cdot H}{5 \cdot N - 4 \cdot z} \cdot N} = 5 - \frac{4 \cdot z}{N}.$$

Fall 2: Es gelte $z > ½ \cdot N - 1$.
Fall 2.1: Es sei $3 \cdot k > H$.

Da $O \geqslant k$ ist, folgt $G = \frac{H}{O} \leqslant 3$.

Fall 2.2: Es gelte $3 \cdot k \leqslant H$.

An dieser Stelle kann beim Algorithmus „HMZ“ für das Zeitintervall der Länge $H_h - i$ wieder nur eine Auslastung von $½ \cdot N$ Prozessoren vorausgesetzt werden. Auf der anderen Seite sind in der vom Algorithmus „BMZ“ erstellten Positionierung zu jedem Zeitpunkt mindestens $z + 1$ Prozessoren aktiv. Man erhält demzufolge eine Minimalfläche von

$$A \geqslant \underbrace{(H_h - i) \cdot ½ \cdot N}_{\text{Fläche „HMZ“}} + \underbrace{H_b \cdot (z + 1)}_{\text{Fläche „BMZ“}}$$

$$> H_h \cdot ½ \cdot N - i \cdot ½ \cdot N + H_b \cdot ½ \cdot N$$

$$= H \cdot ½ \cdot N - i \cdot ½ \cdot N$$

$$\geqslant H \cdot ½ \cdot N - k \cdot ½ \cdot N$$

$$\geqslant H \cdot \frac{1}{2} \cdot N - \frac{H}{6} \cdot N$$

$$= \frac{1}{3} \cdot H \cdot N.$$

Für der Gütefaktor ergibt sich somit

$$G = \frac{H}{O} \leqslant \frac{H \cdot N}{A} < 3.$$

■

Für $z = \lfloor ½ \cdot N \rfloor$ wird noch ein Beispiel angegeben, bei welchem nur der bewiesene Gütefaktor von Drei erreicht wird. Damit die Prozessoranzahlen und Laufzeiten der Module berechenbar sind und mindestens fünf Module entstehen, muß $N \geqslant 13$ gelten.

Modulgrößen		
Modulnummer x	Prozessoranzahl B(x)	Laufzeit T(x)
x = 1	$\lfloor ½ \cdot N \rfloor + 1$	$\left\lfloor \frac{\lceil \frac{N}{2} \rceil - 2}{5} \right\rfloor + 2$
x = 2	1	$\left\lfloor \frac{\lceil \frac{N}{2} \rceil - 2}{5} \right\rfloor + 2$
$3 \leqslant x \leqslant 2 \cdot \left\lfloor \frac{\lceil \frac{N}{2} \rceil - 2}{5} \right\rfloor + 3$ und x ungerade	$\lceil ½ \cdot N \rceil - 2$	1
$3 \leqslant x \leqslant 2 \cdot \left\lfloor \frac{\lceil \frac{N}{2} \rceil - 2}{5} \right\rfloor + 3$ und x gerade	5	1

Wie schon in vielen anderen Beispielen entscheide auch hier bei gleicher Modullaufzeit die Modulnummer über die Reihenfolge in der die Module positioniert werden. Für $N = 23$ entsteht das in der Abbildung 67 dargestellte Bild.

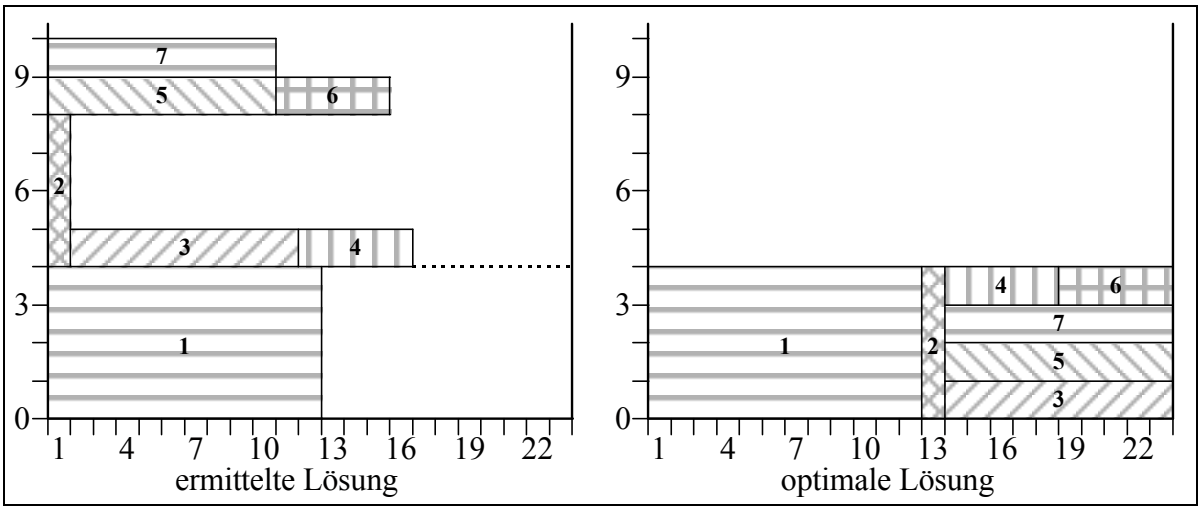


Abbildung 67 - Beispiel, bei dem nur der bewiesene Gütefaktor erreicht wird.

Man sieht, daß nur das Modul 1 mit „BMZ“ positioniert wird:

$$H_b = \left\lfloor \frac{\lceil \frac{N}{2} \rceil - 2}{5} \right\rfloor + 2.$$

In der von „HMZ“ ermittelten Teilpositionierung ist die Laufzeit der untersten Ebene durch die Laufzeit des Moduls 2 gegeben. Da nicht zwei Module mit ungerader Nummer größer als 2 und eines mit der Prozessoranzahl 5 in eine Ebene passen, werden noch $\left\lfloor \frac{\lceil \frac{N}{2} \rceil - 2}{5} \right\rfloor$ Ebenen mit der Laufzeit Eins angelegt. Als Gesamtlaufzeit H erhält man somit

$$H = H_b + H_h = 3 \cdot \left\lfloor \frac{\left\lceil \frac{N}{2} \right\rceil - 2}{5} \right\rfloor + 4.$$

Im Gegensatz dazu benötigt die optimale Positionierung nur $\left\lfloor \frac{\left\lceil \frac{N}{2} \right\rceil - 2}{5} \right\rfloor + 2$ Einheiten. Man sieht, daß die $\left\lfloor \frac{\left\lceil \frac{N}{2} \right\rceil - 2}{5} \right\rfloor + 1$ Module mit ungerader Nummer größer als 2 gleichzeitig mit den Modulen 1 und 2 ausgeführt werden können. Da nur $\left\lfloor \frac{\left\lceil \frac{N}{2} \right\rceil - 2}{5} \right\rfloor + 1$ Module mit ungerader Nummer größer als 2 existieren, können auf den gleichen Prozessoren ab dem Zeitpunkt $\left\lfloor \frac{\left\lceil \frac{N}{2} \right\rceil - 2}{5} \right\rfloor + 1$ alle Module mit gerader Nummer größer als 3 ausgeführt werden. Demzufolge erhält man einen Gütefaktor von

$$G = \frac{3 \cdot \left\lfloor \frac{\left\lceil \frac{N}{2} \right\rceil - 2}{5} \right\rfloor + 4}{\left\lfloor \frac{\left\lceil \frac{N}{2} \right\rceil - 2}{5} \right\rfloor + 2} = 3 - \frac{2}{\left\lfloor \frac{\left\lceil \frac{N}{2} \right\rceil - 2}{5} \right\rfloor + 2} \geqslant 3 - \frac{2}{\frac{\left\lceil \frac{N}{2} \right\rceil - 2}{5} - \frac{4}{5} + 2} \geqslant 3 - \frac{2}{\frac{\frac{N}{2} - 2}{5} - \frac{4}{5} + 2} = 3 - \frac{20}{N + 8}.$$

Man kann sich also dem Gütefaktor Drei beliebig weit annähern, indem man N hinreichend groß wählt.

9.6 Kleine Verbesserungen beim Rechteck - Füll - Problem

Beim Algorithmus „BMZ“ kann man eine bessere Positionierung erzielen, wenn man ein nachfolgendes Modul nicht immer so zeitig wie möglich bzgl. der Prozessorendzeiten startet, sondern zuerst versucht, es in die bereits vorhandenen Lücken einzufügen. Dabei kann man wiederum zwei Taktiken verfolgen: Einmal nutzt man zur Aufnahme des Moduls die unterste mögliche Lücke (first fit). Man kann aber auch alle Lücken durchsuchen und jene wählen, in die das Modul am besten paßt (best fit). Jetzt muß man nur noch definieren, was „am besten“ bedeutet. Die beiden Standardherangehensweisen beurteilen die Restlücke zuerst bzgl. der verfügbaren Prozessoranzahl und dann bzgl. des verfügbaren Zeitintervalls bzw. umgekehrt. Man kann aber auch die verbleibende freie Fläche als Kriterium wählen.
Fügt man die Module mit dem Algorithmus „HMZ“ ein, so ist natürlich auch die Anwendung der eben vorgestellten Verfahren möglich. Dabei muß man allerdings beachten, daß sich ein Modul, welches von unten in die aktuelle Ebene hineinragt, direkt neben dem zuvor in der aktuellen Ebene positionierten Modul befindet. Die Abbildung 68 soll dies verdeutlichen.
Da bei diesem Vorgehen die Ebenenstruktur der Positionierung zerstört wird, ist es vielleicht günstiger, eine andere Herangehensweise zu verfolgen: Man verwaltet eine Liste der freien Rechtecke. Am Anfang enthält sie nur ein Eintrag ab Zeitpunkt Null, Prozessor 1 mit der Gesamtprozessoranzahl N und der Laufzeit unendlich. Jedes nachfolgende Modul Z wird in die linke untere Ecke eines mittels first fit oder best fit ausgesuchten Rechtecks R eingefügt. R wird dann aus der Liste gelöscht. Nach dem Einfügen entstehen, abhängig von der gewählten Strategie, die unten angegebenen neuen Rechtecke (vgl. auch Abbildung 69). Es ist jeweils die untere linke und die obere rechte Ecke im Format (Zeitpunkt, Prozessornummer) angegeben. Sollte ein zu erstellendes Rechteck die Laufzeit oder die Prozessoranzahl Null haben, so wird es nicht erzeugt. Das Ausgangsrechteck habe nachfolgend die Bezeichnung

R = ((a, b), (c, d)). Vom Modul Z weiß man, daß es x Einheiten hoch ist und y Prozessoren benötigt.
Version 1: ((a + x, b), (c, d)) und ((a, b + y), (a + x, d))
Version 2: ((a + x, b), (c, b + y)) und ((a, b + y), (c, d))
Bei der Version 2 sollte (im Sinne des Algorithmus „HMZ“) bei einer Prozessoranzahl von N natürlich die als Version 1 beschriebene Teilung durchgeführt werden.
Da im weiteren Verlauf des Algorithmus die Laufzeit der Rechtecke abnimmt, jedoch über die Prozessoranzahl keine Angabe gemacht werden kann, liefert die Version 1 sicherlich in vielen Fällen das bessere Resultat.

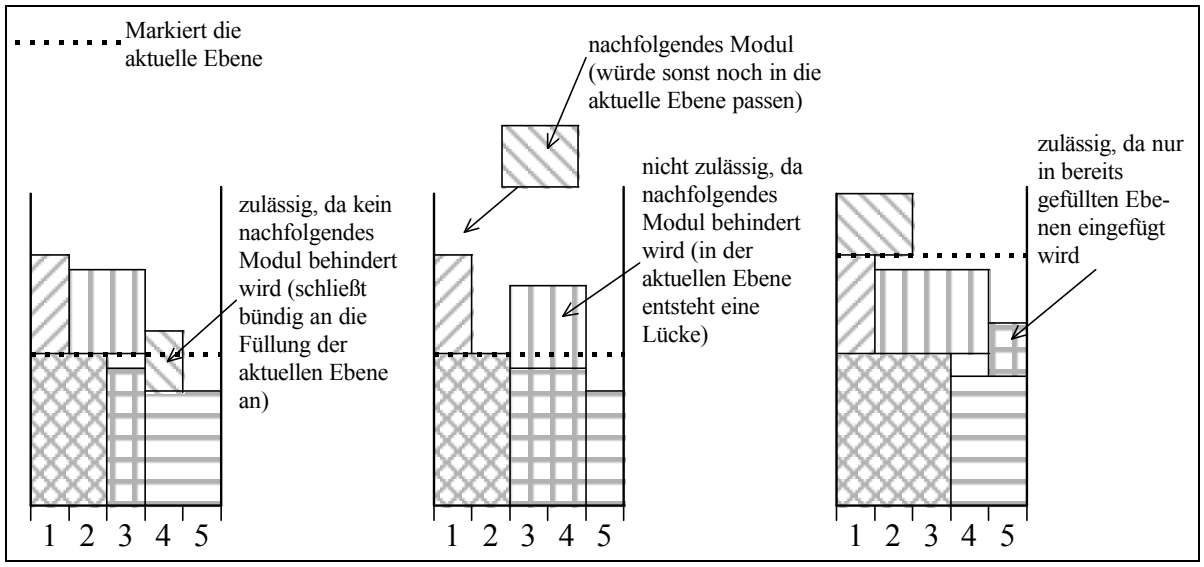


Abbildung 68 - erlaubte und verbotene Einfügepositionen.

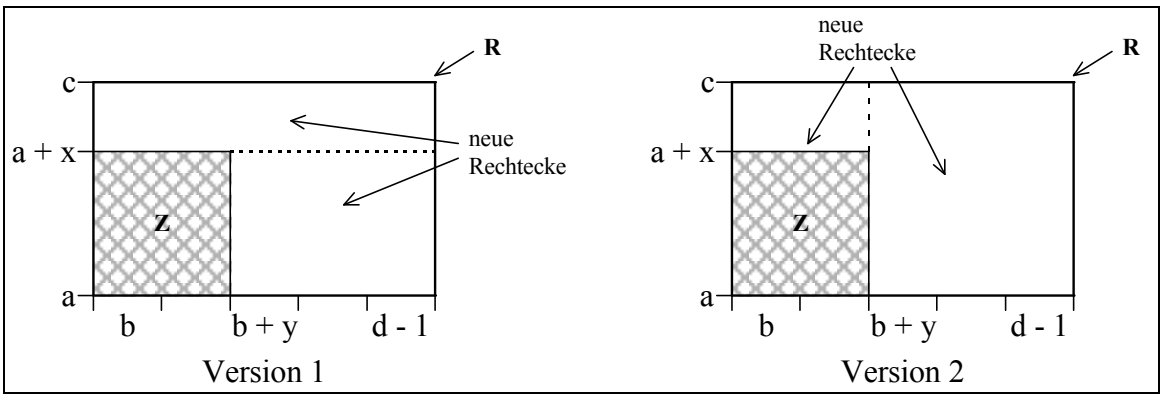


Abbildung 69 - Rechteckaufteilungsverfahren.

Für den Algorithmus „HMZ“ kann man noch eine weitere Verbesserung angeben: Dazu wechselt man in jeder Ebene die Einfügerichtung und läßt jedes Modul so zeitig wie möglich auf den zugeteilten Prozessoren starten. Man nutzt also die ursprüngliche Ebenenstruktur nur noch zur Berechnung der Einfügeposition (vgl. Abbildung 70). Demzufolge wird das Modul mit größten Laufzeit innerhalb einer Ebene immer abwechselnd auf dem Prozessor 1 bzw. auf dem Prozessor N ausgeführt. Allerdings wird durch dieses Vorgehen die Ebenenstruktur vollständig zerstört. Die restlichen Lücken können beim eben beschriebenen schlangenförmigen

Einfügen natürlich auch noch durch eins der oben angegebenen Verbesserungsverfahren gefüllt werden.

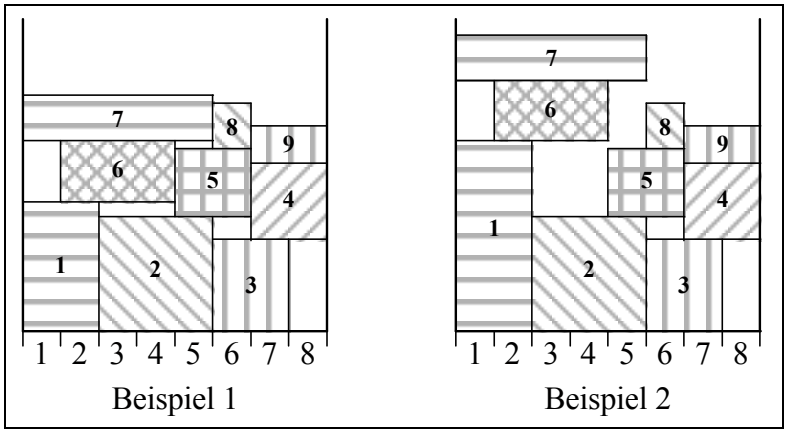


Abbildung 70 - schlangenförmiges Einfügen.

9.7 Auswahl des nächsten Moduls zur Parallelisierung

Ein weiteres Problem ist die Ermittlung eines geeigneten Moduls zur Parallelisierung. Geht man nach dem in Algorithmus „P1+“ praktizierten Schema vor, so entsteht schon bei dem in der Abbildung 71 dargestellten Beispiel nur ein Gütefaktor von $\frac{N}{3}$. Die drei Module haben eine Laufzeit von 1 bei der Zuteilung von weniger als N Prozessoren. Werden sie voll parallel ausgeführt, so sind sie nach $\frac{1}{N}$ Schritten fertig. Damit der optimale Schedule nur eine Gesamtlaufzeit von $O = \frac{3}{N}$ Einheiten hat, muß es mindestens 3 Prozessoren geben. Sofern sich der Algorithmus den besten erstellten Zwischenschedule merkt, gelangt er zu einer Gesamtlaufzeit von $H = 1$.
Obwohl im $(N - 2)$-ten und im $(2 \cdot N - 2)$-ten Schritt jeder Prozessor bis zu seinem Endzeitpunkt ein Modul ausführt, wird nur ein Gütefaktor von

$$G = \frac{1}{\frac{3}{N}} = \frac{N}{3}$$

erreicht. Es muß also auch bei Modulen, welche nicht zum Zeitpunkt H ausgeführt werden, die Prozessoranzahl erhöht werden.

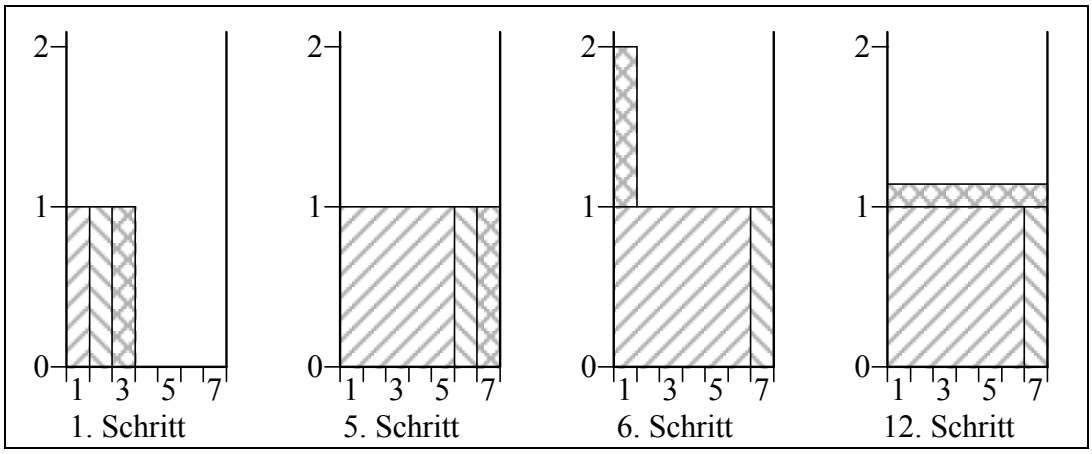


Abbildung 71 - Beispiel zur Verdeutlichung des Auswahlproblems.

Ein Ansatz zur Auswahl eines Moduls wäre die folgende Vorgehensweise: Solange bei der Parallelisierung eines erst zum Zeitpunkt H beendeten Moduls keine Verschlechterung auftritt, wird der alte Algorithmus beibehalten. Anschließend wählt man das Modul i, bei welchem der wie folgt definierte Zahlenwert maximal ist:
Version 1: $T(i) - T(i, B(i) + 1)$
Version 2: $T(i) \cdot B(i) - T(i, B(i) + 1) \cdot (B(i) + 1)$
Man entscheidet sich also für das Modul, bei dem die Laufzeitreduzierung bzw. die Flächeneinsparung ein Maximum annimmt. Sollten mehrere Module den gleichen Wert liefern, so wählt man aus ihnen das Modul, welches zuerst eingefügt wird. Allerdings liefert auch dieses Vorgehen (egal welche Version) im allgemeinen keinen zufriedenstellenden Schedule.
Man betrachte dazu das folgende Beispiel: Der Einfachheit halber sei $N \geqslant 9$ eine Quadratzahl. Außerdem stehe ε für eine hinreichend kleine Zahl echt größer als Null. Die Laufzeiten der einzelnen Module sind aus der Tabelle entnehmbar:

Modullaufzeiten			
Modul 1		Module 2 bis $\sqrt{N} + 1$	
Prozessoranzahl i	Laufzeit	Prozessoranzahl i	Laufzeit
$1 \leqslant i \leqslant N - 1$	$1 + \left(N - \sqrt{N} + 1 - i\right) \cdot \varepsilon$	$1 \leqslant i \leqslant \sqrt{N} - 1$	1
$i = N$	$\frac{1 + \left(N - \sqrt{N}\right) \cdot \varepsilon}{N}$	$\sqrt{N} \leqslant i \leqslant N$	$\frac{1}{\sqrt{N}} - \left(i - \sqrt{N}\right) \cdot \varepsilon$

In der Abbildung 72 ist für $N = 9$ der Ablauf des Algorithmus bei der Verwendung des Einfügeverfahrens „BMZ“ dargestellt.
Der Schedule nach dem $\left(N - \sqrt{N}\right)$-ten Schritt hat eine Gesamtlaufzeit von $1 + \varepsilon$. Er ist unter der Voraussetzung, daß

$$\varepsilon \leqslant \frac{1}{\sqrt{N} + N^2 \cdot \sqrt{N} - N^2}$$

gilt, besser als der nach der letzten Parallelisierung erhaltene Schedule (alle Module werden mit N Prozessoren ausgeführt), welcher

$$\frac{1 + \left(N - \sqrt{N}\right) \cdot \varepsilon}{N} + \sqrt{N} \cdot \left(\frac{1}{\sqrt{N}} - \left(N - \sqrt{N}\right) \cdot \varepsilon\right) = 1 + \frac{1}{N} + \varepsilon \cdot \left(1 - \frac{1}{\sqrt{N}} + N - N \cdot \sqrt{N}\right)$$

Zeiteinheiten benötigt.
In einem besseren Schedule werden die $\sqrt{N}$ gleichartigen Module gleichzeitig auf verschiedenen Prozessorgruppen mit je $\sqrt{N}$ Prozessoren ausgeführt. Demzufolge erhält man eine Gesamtlaufzeit von

$$\frac{1 + \left(N - \sqrt{N}\right) \cdot \varepsilon}{N} + \frac{1}{\sqrt{N}} \geqslant O$$

als obere Schranke für die Gesamtlaufzeit O des optimalen Schedules. Als Gütefaktor resultiert somit

$$G = \frac{H}{O} \geqslant \frac{1 + \varepsilon}{\frac{1 + \left(N - \sqrt{N}\right) \cdot \varepsilon}{N} + \frac{1}{\sqrt{N}}} = \frac{N + N \cdot \varepsilon}{1 + \sqrt{N} + \varepsilon \cdot \left(N - \sqrt{N}\right)}.$$

Aufgrund der oben angegebenen Bedingung für ε ist

$$G \geqslant \frac{N}{1 + \sqrt{N} + \frac{N - \sqrt{N}}{N^2 \cdot \sqrt{N} - N^2 + \sqrt{N}}} \geqslant \frac{N}{1 + \sqrt{N} + \frac{N - \sqrt{N}}{N^2 \cdot \sqrt{N} - N^2}} = \frac{N}{1 + \sqrt{N} + \frac{1}{N \cdot \sqrt{N}}}$$

$$\geqslant \frac{N}{\sqrt{N}+2} = \sqrt{N} - 2 + \frac{4}{\sqrt{N}+2} \geqslant \sqrt{N} - 2.$$

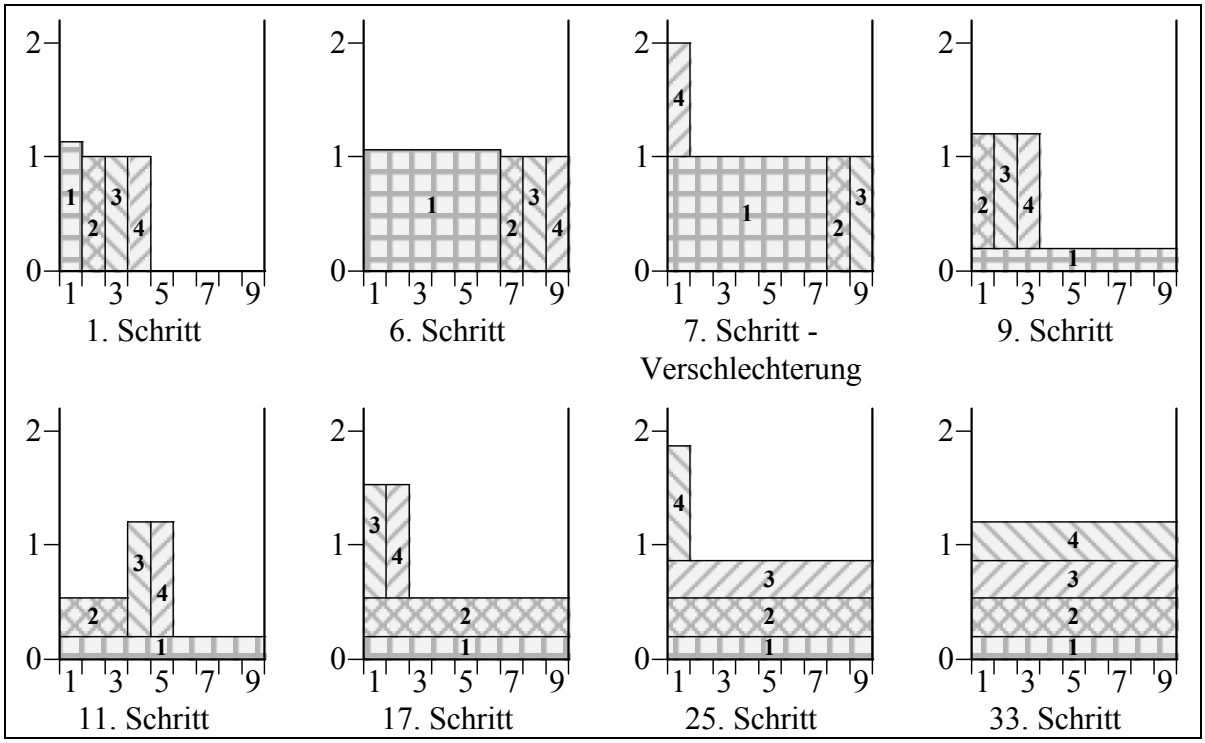


Abbildung 72 - Schlechte Wahl des Moduls zur Parallelisierung.

9.8 Sequentielle Positionierungsreihenfolge

Das nachfolgende Beispiel soll zeigen, daß die Verwendung der mittels

$$x < y \Rightarrow T(x, 1) \geqslant T(y, 1)$$

definierten Modulreihenfolge, beim Rechteck - Füll - Problem im allgemeinen nicht zu zufriedenstellenden Positionierungen führt. Damit die Rundungsklammern weggelassen werden können, muß $N + 1 \geqslant 9$ eine Quadratzahl sein. Es seien insgesamt $2 \cdot \sqrt{N+1} - 1$ Module mit den in der nachfolgenden Tabelle angegebenen Laufzeiten vorhanden.

Modullaufzeiten			
Modulnummer ist ungerade		Modulnummer ist gerade	
Prozessoranzahl i	Laufzeit	Prozessoranzahl i	Laufzeit
$1 \leqslant i \leqslant \sqrt{N+1} - 2$	1	$1 \leqslant i \leqslant N - 1$	1
$\sqrt{N+1} - 1 \leqslant i \leqslant N$	$\frac{1}{\sqrt{N+1} - 1}$	$i = N$	$\frac{1}{N}$

Da alle Module am Anfang die Laufzeit Eins haben, seien sie entsprechend ihrer Modulnummer sortiert. Für $N = 8$ ergibt sich das in der Abbildung 73 dargestellte Bild.
Man kann erkennen, daß eine Positionierung bei der Beibehaltung der Positionierungsreihenfolge mit einer kleineren Gesamtlaufzeit als Eins nicht erreicht werden kann.
Da

$$\sqrt{N+1} \cdot \left(\sqrt{N+1} - 1\right) = N + 1 - \sqrt{N+1} \leqslant N$$

ist, können die $\sqrt{N+1}$ Module mit ungerader Nummer bei Ausführung auf je $\sqrt{N+1}$ - 1 Prozessoren gleichzeitig abgearbeitet werden. Demzufolge liegt die Gesamtlaufzeit h einer möglichen Positionierung ($h \geqslant O$) bei

$$h = \frac{1}{\sqrt{N+1} - 1} + \left(\sqrt{N+1} - 1\right) \cdot \frac{1}{N} = 2 \cdot \frac{\sqrt{N+1}}{N}.$$

Es ergibt sich somit ein Gütefaktor von

$$G = \frac{H}{O} \geqslant \frac{H}{h} = \frac{1}{2 \cdot \frac{\sqrt{N+1}}{N}} = \frac{N}{2 \cdot \sqrt{N+1}} = ½ \cdot \sqrt{N+1} - \frac{1}{2 \cdot \sqrt{N+1}} \geqslant ½ \cdot \sqrt{N+1} - ½.$$

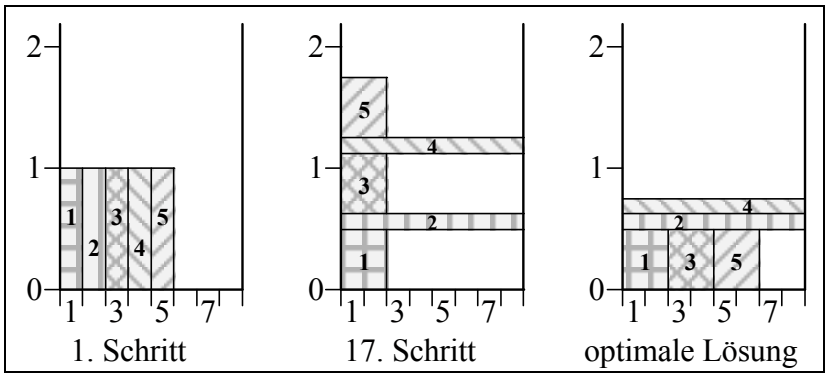


Abbildung 73 - Beispiel, bei dem eine Änderung der Reihenfolge besser wäre.

9.9 Beispiel für „P1+MF“

In diesem Abschnitt wird ein Beispiel angegeben, bei welchem der Algorithmus „P1+MF“ wirklich nur einen Gütefaktor von 3 - ε erzielt. Dazu sei $M = 2$, $N \geqslant 6$ und $\varepsilon > 0$ eine hinreichend kleine Zahl. Das Modul 1 habe bei der Benutzung von $i < N$ Prozessoren die Laufzeit $N - (i - 1) \cdot \varepsilon$. Wird es auf allen N Prozessoren ausgeführt, so sei es nach einer Zeiteinheit fertig. Das Modul 2 habe die Laufzeit $\frac{2 \cdot N - N \cdot \varepsilon}{i \cdot (N - 3 + \varepsilon)}$, wobei i die verwendete Prozessoranzahl bezeichnet. Wenn man ε so wählt, daß

$$N - ((N - 1) - 1) \cdot \varepsilon > \frac{2 \cdot N - N \cdot \varepsilon}{N - 3 + \varepsilon} \qquad (1)$$

gilt, parallelisiert der Algorithmus „P1+MF“ in Zeile 12) das Modul 1 solange, bis es mit allen Prozessoren ausgeführt wird, d.h., $B(1) = N$ ist. Die anschließende Bestimmung des Flächenminimums eines jeden Moduls bringt keine Änderung der Prozessoranzahlen B(1) und B(2). Beim Positionieren beider Module (egal ob mit „HMZ“, „BMZ“ oder „BHMZ“) resultiert eine Gesamtlaufzeit von

$$H = 1 + \frac{2 \cdot N - N \cdot \varepsilon}{N - 3 + \varepsilon} = (3 - \varepsilon) \cdot \frac{N - 1}{N - 3 + \varepsilon}.$$

Damit im (N - 1)-ten Schritt, also mit $B(1) = N - 1$ und $B(2) = 1$, kein besserer Schedule als im zuvor betrachteten N-ten Schritt erzielt wird, muß man die Bedingung (1) für ε auf

$$N - (N - 1 - 1) \cdot \varepsilon > 1 + \frac{2 \cdot N - N \cdot \varepsilon}{N - 3 + \varepsilon} \qquad (2)$$

verschärfen. Wählt man

$$\varepsilon < \frac{N^2 - 6 \cdot N + 3}{(N - 3) \cdot (N - 2)},$$

so ist (2) garantiert erfüllt. Die belegte Fläche im N-ten Schritt beträgt

$$\sum_{i=1}^{M} T(i) \cdot B(i) = N + \frac{2 \cdot N - N \cdot \varepsilon}{N - 3 + \varepsilon} = N \cdot \frac{N - 1}{N - 3 + \varepsilon}.$$

Wie man sieht, ist die Bedingung in Zeile 24) erfüllt, und der Algorithmus bricht ab. Da der optimale Schedule, welcher durch die Nacheinanderausführung der beiden Module auf allen Prozessoren entsteht, die Gesamtlaufzeit

$$O = 1 + \frac{2 \cdot N - N \cdot \varepsilon}{N \cdot (N - 3 + \varepsilon)} = \frac{N - 1}{N - 3 + \varepsilon}$$

hat, ergibt sich insgesamt auch nur ein Gütefaktor von 3 - ε.

Die Abbildung 74 veranschaulicht das oben angeführte Beispiel für $N = 6$ und $\varepsilon = \frac{1}{6}$.

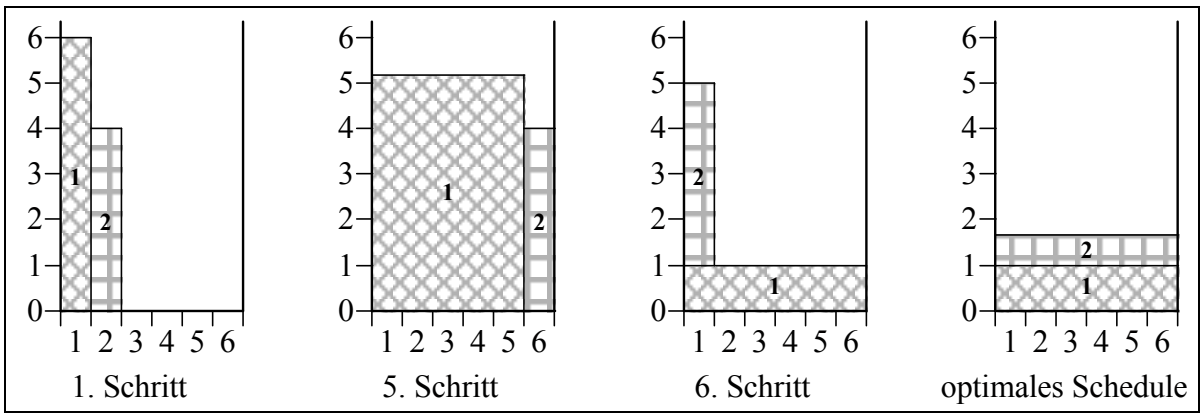


Abbildung 74 - Beispiel mit Faktor Drei $\left(N = 6; \varepsilon = \frac{1}{6}\right)$.

9.10 Beispielgenerierung für „P1+MF“

Da die direkte Eingabe der Modullaufzeiten bei größerer Modulanzahl sehr aufwendig ist, wurde ein Hilfsprogramm zur Beispielgenerierung erstellt. Der Quellcode kann in der Datei /src/prg1/bsp_bau1.c auf der beiliegenden CD eingesehen werden.

Die Beispielgenerierung geschieht in mehreren Schritten. Als erstes muß die Gesamtanzahl der Module bestimmt werden. Dies geschieht durch die Ermittlung einer ganzen Zufallszahl, wobei die folgenden Verteilungen zur Verfügung stehen:

a) konstant x: Dabei steht x für die zurückgegebene Zahl.
b) gleich x y: Es wird eine gleichverteilte Zufallszahl aus dem Intervall [x; y] gewählt.
c) exponential x y λ: Hiermit wird eine mit dem Parameter $\lambda > 0$ exponentialverteilte Zufallszahl zwischen x und y bestimmt. Man kann auch $\lambda < 0$ wählen. In diesem Fall wird zuerst ein z mittels exponential x y (-λ) erzeugt und anschließend der Wert y - z + x zurückgegeben. Auf diesem Weg erhält man eine an der Senkrechten durch ½ · (x + y) gespiegelte Exponentialverteilung.
d) normal x y α σ: Es wird die Glockenkurve als Dichtefunktion zugrunde gelegt und eine Zufallszahl aus dem Intervall [x; y] zurückgegeben.

Weitere Details zur Zufallszahlenerzeugung sind aus den nachfolgenden Abschnitten entnehmbar.

Steht die Modulanzahl fest, so muß der Algorithmus die Laufzeiten der einzelnen Module erzeugen. Dazu wird zuerst die sequentielle Laufzeit T(x, 1) mit Hilfe einer der vier Vertei-

lungen a) bis d) festgelegt. Bei der anschließenden Erzeugung der parallelen Laufzeiten $T(x, i)\ \forall i \in [2; N]$ nutzt das Verfahren die folgende Beziehung aus:

$$T(x, i - 1) \geqslant T(x, i) \geqslant \frac{1}{i} \cdot T(x, 1).$$

Der Algorithmus ermittelt also eine Zufallszahl z zwischen 0 und 1 setzt

$$T(x, i) = \frac{1}{i} \cdot T(x, 1) + z \cdot \left(T(x, i - 1) - \frac{1}{i} \cdot T(x, 1)\right).$$

Damit nicht die Laufzeiten aller Module nach der gleichen Verteilung erstellt werden, können mehrere Modulgattungen angegeben werden. Es sei k die Anzahl der angegebenen Modulgattungen. Dann wird jeder Modulgattung ein Wichtungsfaktor w_i ($\forall i \in [1; k]$) zugeordnet. Die Auswahlwahrscheinlichkeit für die Modulgattung i liegt dann bei $\frac{w_i}{\sum_{j=1}^{k} w_j}$. Wird für w_i kein konstanter Wert eingetragen, so legt das Beispielgenerierungsprogramm diesen am Anfang entsprechend der angegebenen Verteilung fest. Die w_i sind anschließend für die gesamte Generierung eines Beispiels fest.

Ein Beispiel soll die Funktion des eben vorgestellten Algorithmus, sowie den Aufbau der Eingabedatei verdeutlichen. Der Inhalt von /make/prg1/bsp1a.txt ist in der Abbildung 75 dargestellt. Der gebogene Pfeil ⍥ am Zeilenende symbolisiert, daß der Inhalt der nachfolgenden Zeile noch in die aktuelle Zeile gehört und nur aus Platzgründen umgebrochen wurde.

Der erste Integerwert in der Datei steht für die Gesamtprozessoranzahl. An dieser Stelle darf keine Verteilung angegeben werden. Aus der nachfolgenden Verteilung ermittelt die Routine die Modulanzahl. Die Erkennung der Verteilung geschieht nur durch den ersten Buchstaben, also ‚k', ‚g', ‚e' bzw. ‚n'. Aus diesem Grund dürfen als Kommentar nur große Buchstaben verwendet werden. Im Anschluß an die Modulanzahlverteilung folgen in je einer Zeile die k Modulgattungen. Die erste Verteilung in einer Zeile gibt dabei das w_i an. Aus der nächsten Verteilung wird die sequentielle Laufzeit der Module bestimmt. Danach muß eine natürliche Zahl j echt größer als Eins folgen. Die zur Ermittlung der $T(x, 2)$ bis $T(x, j)$ notwendige Zufallszahl zwischen Null und Eins (es findet keine Überprüfung der angegebenen Intervallgrenzen statt) wird entsprechend der sich anschließenden Verteilung erstellt. Sollte j kleiner als N sein, so legt der Algorithmus durch das Einlesen einer weiteren natürlichen Zahl ein neues j fest. Bis zu diesem Wert wird die nächste gelesene Verteilung benutzt. Alle zu einer Modulgattung gehörenden Befehle müssen in einer Zeile stehen. Dies wird in Abbildung 75 durch den gebogenen Pfeil ⍥ am Zeilenende angedeutet.

Zum Test des Algorithmus „P1+MF" wurden zwei verschiedene Beispielgenerierungsdateien verwendet. Sie sind in ihrer allgemeinen Form in den Abbildungen 76 und 77 angegeben. Dabei müssen die in geschweiften Klammern geschriebenen Formeln durch die entsprechenden Zahlenwerte ersetzt werden.

Man sieht sehr schön, daß bei den mit Hilfe von /make/prg1/bsp1a.txt (siehe Abbildung 76) erzeugten Beispielen zwei Modulgattungen mit stark unterschiedlichen Parallelisierungseigenschaften vorliegen. Die Module aus der ersten Gattung können bis zur Verwendung von $0{,}4 \cdot N$ Prozessoren relativ gut parallelisiert werden. Anschließend ist trotz Hinzunahme weiterer Prozessoren keine Laufzeitverbesserung möglich. Die Elemente der zweiten Modulgattung haben sicherlich einen geringen Parallelisierungsmehraufwand bei der Ausführung mit allen N Prozessoren. Werden sie hingegen mit nur 70% der verfügbaren Prozessoren gestartet, so entsteht ein relativ hoher Mehraufwand. Das Testbeispiel ist also so konzipiert, daß der Schedule nicht immer vom Algorithmus „P1+" erstellt wird.

```
PROZESSORANZAHL: 10

MODULANZAHL: konstant 10

MODULWAHRSCH  LAUFZEIT_MIT_EINEM_PROZ  PARALLELISIERBARKEIT
gleich 1 3    exponential 1 100 1      BIS 4 PROZ: normal 0 1 0.5 0.2;
                                       BIS 10 PROZ: konstant 0
konstant 3.5  normal 0.5 4.7 2.2 1     BIS 7 PROZ: exponential 0 1 0.5;
                                       BIS 10 PROZ: konstant 0.5
```

Abbildung 75 - Datei /make/prg1/bsp1a.txt.

```
PROZESSORANZAHL: {N}

MODULANZAHL: konstant {M}

MODULWAHRSCH LAUFZEIT_MIT_EIN_PROZ  PARALLELISIERBARKEIT
gleich 1 3   exponential 1 100 1    BIS {0,4·N} PROZ: normal 0 1 0.5 0.2;
                                    BIS {N} PROZ: konstant 0
konstant 3.5 normal 0.5 4.7 2.2 1   BIS {0,7·N} PROZ: exponential 0 1 0.5;
                                    BIS {N} PROZ: konstant 0.5
```

Abbildung 76 - Datei /make/prg1/bsp1a.txt - allgemeine Form.

```
PROZESSORANZAHL: {N}

MODULANZAHL: konstant {M}

MODULWAHRSCH LAUFZEIT_MIT_EIN_PROZ     PARALLELISIERBARKEIT
gleich 1 4   normal 1 2 1.5 0.2        BIS {N} PROZ: expo 0 1 0.7
gleich 1 4   normal 10 11 10.5 0.2     BIS {N} PROZ: expo 0 1 0.7
gleich 1 4   normal 100 101 100.5 0.2  BIS {N} PROZ: expo 0 1 0.7
```

Abbildung 77 - Datei /make/prg1/bsp1b.txt - allgemeine Form.

Die Beispielgenerierungsdatei /make/prg1/bsp1b.txt (siehe Abbildung 77) erstellt Optimierungseingaben, bei denen die sequentielle Laufzeit sehr stark variiert. Ein guter Schedule liegt also vor, wenn die Module mit hoher Laufzeit mit möglichst allen Prozessoren ausgeführt werden.

9.10.1 Gleichverteilte Zufallszahlen

Da zur Beispielgenerierung sehr viele Zufallszahlen benötigt werden, hat man schnell die Grenzen der Standardfunktion `rand()` erreicht. Diese erzeugt nur gleichverteilte Integerwerte im Intervall [0; `RAND_MAX`]. Benötigt man nun zufällige ganze Zahlen im Bereich [0; a], so muß man sogar bei a $\leqslant$ `RAND_MAX` aufpassen. Damit die kleineren Zahlen keine höhere Wahrscheinlichkeit als die größeren haben, reicht es im allgemeinen nicht aus, diese mittels `rand() % (a + 1)` zu erzeugen. Eine korrekte Vorgehensweise besteht darin, vor der Modulo - Operation alle von `rand()` erzeugten Zufallszahlen, die nicht kleiner als

$$(a + 1) \cdot \left\lfloor \frac{RAND\_MAX + 1}{a + 1} \right\rfloor$$

sind, zu verwerfen. In diesem Fall muß `rand()` erneut ausgeführt werden.
Ist nun a > `RAND_MAX`, so muß die gesuchte Zufallszahl aus mehreren Zufallszahlen zusammengesetzt werden. Angenommen es gilt

$$(\texttt{RAND\_MAX} + 1)^i \leqslant a < (\texttt{RAND\_MAX} + 1)^{i+1}.$$

Dann werden zuerst i zufällige Zahlen z_j ($0 \leqslant j < i$) aus dem Intervall [0; RAND_MAX] erzeugt. Anschließend bestimmt der implementierte Algorithmus noch z_i zwischen 0 und $\left\lfloor \frac{a}{(\texttt{RAND\_MAX} + 1)^i} \right\rfloor$. Die zurückgelieferte Zufallszahl beträgt

$$\sum_{j=0}^{i} (\texttt{RAND\_MAX} + 1)^j \cdot z_j,$$

sofern sie kleiner oder gleich a ist. Sollte dies nicht der Fall sein, so werden alle i + 1 mittels `rand()` bestimmten Zahlen neu erzeugt.

Daß bei dieser Vorgehensweise jede Zahl die gleiche Wahrscheinlichkeit hat, ist sofort einzusehen, wenn man die z_j als die Ziffern einer (i + 1)-stelligen (RAND_MAX + 1)-nären Zahl auffaßt, deren erste Stelle kleiner oder gleich $\left\lfloor \frac{a}{(\texttt{RAND\_MAX} + 1)^i} \right\rfloor$ sein soll. Aus diesem Grund ist es ausreichend zu zeigen, daß die Zahl a erstellt werden kann. Dann sind auch alle kleineren Werte möglich. Dazu setze man $z_j = \texttt{RAND\_MAX}\ \forall j \in [0; i)$ und $z_i = \left\lfloor \frac{a}{(\texttt{RAND\_MAX} + 1)^i} \right\rfloor$. Ermittelt man nun das dazugehörige Endergebnis, so erhält man

$$\left(\sum_{j=0}^{i-1} (\texttt{RAND\_MAX} + 1)^j \cdot \texttt{RAND\_MAX}\right) + (\texttt{RAND\_MAX} + 1)^i \cdot \left\lfloor \frac{a}{(\texttt{RAND\_MAX} + 1)^i} \right\rfloor$$

$$= (\texttt{RAND\_MAX} + 1)^i - 1 + (\texttt{RAND\_MAX} + 1)^i \cdot \left\lfloor \frac{a}{(\texttt{RAND\_MAX} + 1)^i} \right\rfloor$$

$$\geqslant (\texttt{RAND\_MAX} + 1)^i - 1 + (\texttt{RAND\_MAX} + 1)^i \cdot \left(\frac{a - ((\texttt{RAND\_MAX} + 1)^i - 1)}{(\texttt{RAND\_MAX} + 1)^i} \right)$$

$$= a.$$

9.10.2 Exponentialverteilte Zufallszahlen

Da die Exponentialverteilung über dem Intervall [0; ∞) und nicht über [a; b] definiert ist, müssen alle mit Hilfe der Dichtefunktion

$$p(x) = \lambda \cdot e^{-\lambda \cdot x} \quad (\lambda > 0)$$

ermittelten Wahrscheinlichkeiten durch eine lineare Abbildung in den gewünschten Bereich umgerechnet werden. Man erhält demzufolge als Verteilungsfunktion

$$F(t) = \frac{\int_a^t \lambda \cdot e^{-\lambda \cdot x}\, dx}{\int_a^b \lambda \cdot e^{-\lambda \cdot x}\, dx},$$

wobei $t \in [a; b]$ sein soll. Um nun die exponentialverteilte Zahl t zu erzeugen, gibt man sich eine gleichverteilte Zahl für F(t) vor und berechnet daraus t. Die dazu nötige Formel erhält man durch Umstellen der Verteilungsfunktion:

$$F(t) = \frac{e^{-\lambda \cdot a} - e^{-\lambda \cdot t}}{e^{-\lambda \cdot a} - e^{-\lambda \cdot b}} \quad \Leftrightarrow \quad e^{-\lambda \cdot t} = e^{-\lambda \cdot a} - F(t) \cdot \left(e^{-\lambda \cdot a} - e^{-\lambda \cdot b}\right)$$

$$\Leftrightarrow \quad t = -\frac{1}{\lambda} \cdot \ln\left(e^{-\lambda \cdot a} - F(t) \cdot \left(e^{-\lambda \cdot a} - e^{-\lambda \cdot b}\right)\right).$$

Der Wert für F(t) wird mit Hilfe einer gleichverteilten ganzen Zufallszahl aus dem Intervall [0; z] mit anschließender Division durch z gewonnen. Dabei ist z hinreichend groß zu wählen. Im Algorithmus wird z = `ULONG_MAX` gesetzt.
Liegt der für λ angegebene Wert unter Null, so bildet der Algorithmus den Betrag und erzeugt wie eben die Zufallszahl t. Diese gibt das Verfahren dann aber nicht direkt zurück, sondern es rechnet sie erst mittels a + b - t um. Dadurch erreicht man eine Spiegelung der Dichtefunktion.
Damit aus der stetigen eine punktförmige Verteilung wird, bildet man eine im Intervall [t; t+1) mit $t \in \mathbb{N} \cup \{0\}$ liegende zufällige Fließkommazahl auf den Wert t ab. Um bei diesem Vorgehen auch den Wert b zu erreichen, erweitert man das Intervall vorher um die Länge Eins über b hinaus. Sollte der Algorithmus tatsächlich einmal die Zahl b + 1 erzeugen, so wird sie verworfen und eine neue bestimmt.

9.10.3 Normalverteilte Zufallszahlen

Hier kann man im Prinzip genauso vorgehen wie im letzten Abschnitt, nur mit einer anderen Dichtefunktion:

$$p(x) = \frac{1}{\sigma \cdot \sqrt{2 \cdot \pi}} \cdot e^{\left(-\frac{(x - \alpha)^2}{2 \cdot \sigma^2}\right)}.$$

Man erhält demzufolge auch

$$F(t) = \frac{\int_a^t p(x)\,dx}{\int_a^b p(x)\,dx}.$$

Nun sind aber die Integrale an dieser Stelle nicht lösbar. Aus diesem Grund kann die Fläche unter der Dichtefunktion nur mittels der Trapezregel näherungsweise bestimmt werden. Die verwendete Anzahl an Trapezen ist durch `#define STUETZ 100` vorgegeben. Im Intervall [a; t] muß somit die Fläche

$$f \approx F(t) \cdot \frac{b - a}{\mathtt{STUETZ}} \cdot \left(\frac{p(a) + p(b)}{2} + \sum_{i=1}^{\mathtt{STUETZ} - 1} p\left(a + \frac{(b - a) \cdot i}{\mathtt{STUETZ}}\right) \right)$$

unter p(x) vorhanden sein. Um nun t zu berechnen, sucht der Algorithmus zuerst eine ganze Zahl $k \in [0; \mathtt{STUETZ}]$ mit

$$\frac{1}{2} \cdot \frac{b - a}{\mathtt{STUETZ}} \cdot \left(\sum_{i=0}^{k-1} p\left(a + \frac{(b - a) \cdot i}{\mathtt{STUETZ}}\right) + p\left(a + \frac{(b - a) \cdot (i + 1)}{\mathtt{STUETZ}}\right) \right) \leqslant f$$

$$< \frac{1}{2} \cdot \frac{b - a}{\mathtt{STUETZ}} \cdot \left(\sum_{i=0}^{k} p\left(a + \frac{(b - a) \cdot i}{\mathtt{STUETZ}}\right) + p\left(a + \frac{(b - a) \cdot (i + 1)}{\mathtt{STUETZ}}\right) \right)$$

heraus. Entsprechend der Abbildung 78 wird

$$g = \frac{1}{2} \cdot \frac{b - a}{\mathtt{STUETZ}} \cdot \left(\sum_{i=0}^{k-1} p\left(a + \frac{(b - a) \cdot i}{\mathtt{STUETZ}}\right) + p\left(a + \frac{(b - a) \cdot (i + 1)}{\mathtt{STUETZ}}\right) \right),$$

$$y = a + \frac{(b - a) \cdot k}{\mathtt{STUETZ}} \quad \text{und} \quad z = a + \frac{(b - a) \cdot (k + 1)}{\mathtt{STUETZ}}$$

gesetzt. Wie man sieht, gilt nun:

$$f \approx g + (t - y) \cdot \frac{1}{2} \cdot \left(p(y) + p(y) + \frac{p(z) - p(y)}{z - y} \cdot (t - y)\right).$$

Stellt man diese Gleichung unter der Voraussetzung $p(z) \neq p(y)$ nach t um, so erhält man:

$$t^2 \cdot \frac{1}{2} \cdot \frac{p(z) - p(y)}{z - y} + t \cdot \left(p(y) - \frac{p(z) - p(y)}{z - y} \cdot y\right) - f + g - y \cdot p(y) + \frac{1}{2} \cdot \frac{p(z) - p(y)}{z - y} \cdot y^2 \approx 0$$

$$\Leftrightarrow\ t^2 + t \cdot \left(2 \cdot p(y) \cdot \frac{z - y}{p(z) - p(y)} - 2 \cdot y\right) + 2 \cdot \frac{z - y}{p(z) - p(y)} \cdot (g - f) - 2 \cdot \frac{z - y}{p(z) - p(y)} \cdot y \cdot p(y) + y^2 \approx 0$$

$$\Rightarrow\ t_{1/2} = y - p(y) \cdot \frac{z - y}{p(z) - p(y)} \pm \sqrt{(p(y))^2 \cdot \left(\frac{z - y}{p(z) - p(y)}\right)^2 + 2 \cdot \frac{z - y}{p(z) - p(y)} \cdot (f - g)}.$$

Damit die Scheinlösung ausgeschlossen werden kann, muß man an dieser Stelle zwei Fälle unterscheiden:

Fall 1: Es sei $p(z) - p(y) > 0$.

Wie man sieht, gilt auch

$$p(y) \cdot \frac{z - y}{p(z) - p(y)} > 0.$$

Nun wurde aber y so gewählt, daß $t \geqslant y$ ist, was durch das Minuszeichen nicht erreicht werden kann.

Fall 2: Es gelte $p(z) - p(y) < 0$.

Da die Wurzel nie negativ ist, folgt

$$t_1 \geqslant y - p(y) \cdot \frac{z - y}{p(z) - p(y)} = \frac{z \cdot p(y) - y \cdot p(z)}{p(y) - p(z)} > \frac{z \cdot p(y) - z \cdot p(z)}{p(y) - p(z)} = z.$$

Dies ist aber ein Widerspruch zur Wahl von z. Demzufolge erzeugt das Pluszeichen die Scheinlösung.

Jetzt bleibt nur noch der schon weiter oben ausgeschlossene Fall $p(y) = p(z)$ zur Betrachtung übrig. Aus

$$f = g + (t - y) \cdot \frac{1}{2} \cdot \left(p(y) + p(y) + \frac{p(z) - p(y)}{z - y} \cdot (t - y)\right)$$

läßt sich dann

$$f = g + (t - y) \cdot ½ \cdot (p(y) + p(y))$$

ableiten. Durch Umstellen erhält man

$$t = \frac{f - g}{p(y)} + y.$$

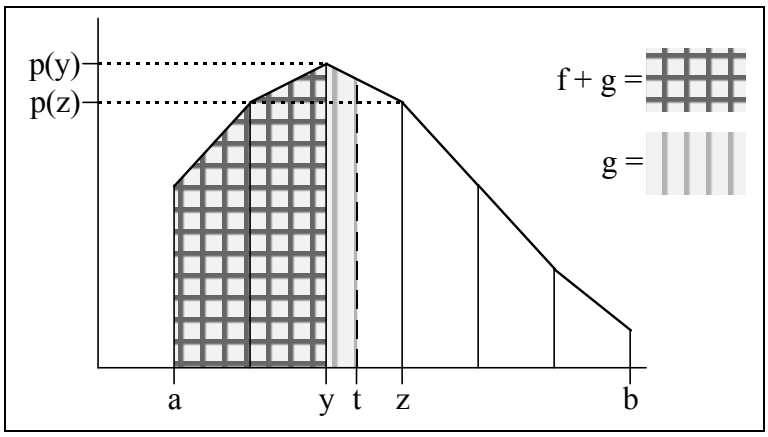


Abbildung 78 - verwendete Bezeichnungen.

9.11 Implementierung des Algorithmus „P1+MF“

Alle praktischen Tests fanden auf einem PC mit einem AMD Athlon X2 Prozessor statt. Durch die Verwendung eines normalen PCs waren die Einflüsse von parallel laufenden Prozessen auf die Laufzeit des Testprozesses minimal, so daß die Meßergebnisse sehr stabil ausgefallen sind. Als Compiler wurde der GCC 4.4 unter Linux eingesetzt.
Der in Pseudocode vorgestellte Algorithmus wurde in C - Funktionen umgesetzt. Diese können unter /src/prg1/opti1h.c bzw. /src/prg1/opti1b.c, je nach verwendetem Positionierungsalgorithmus („HMZ“ oder „BMZ“), auf der beiliegenden CD eingesehen werden. Alle gemeinsam genutzten Hilfsroutinen befinden sich in den Dateien im Verzeichnis /src/prgx/. Unter /make/prg1 liegt ein Makefile, welches zur Erstellung der einzelnen Testprogramme verwendet wurde.
In allen Fällen erfolgte die Sortierung der Module mittels Merge - Sort. Zur Minimierung der Laufzeit wurde außerdem ausgenutzt, daß beim flächenminimierenden Algorithmus immer nur das parallelisierte Modul verschoben werden muß, damit wieder alle Module sortiert sind.
Mit Hilfe von /src/suchvgl/suchvgl.c wurden Testreihen aufgenommen, die belegen, daß auf dem verwendeten Rechner der im Abschnitt 9.1 beschriebene Algorithmus zur Bestimmung des Minimums bei einer Prozessoranzahl echt größer als Eins langsamer ist als ein Verfahren, welches ein Fenster der mit gegebenen Prozessoranzahl über die einzelnen Elemente bewegt. Aus diesem Grund enthält die Implementierung die fensterbasierte Methode mit einer asymptotischen Laufzeit von $O(N^2)$. Die Vergleichsergebnisse für das Einfügen von Modulen mit gleichverteilter Breite zwischen 1 und N sind unter /res/suchvgl/breite_n.txt abgelegt. Beschränkt man die Modulbreiten auf das Intervall von 1 bis $\frac{N}{2}$, so ergeben sich die in /res/suchvgl/breite_n_halbe.txt Zeiten.
Außerdem ist es möglich, den in Pseudocode angegebenen Algorithmus noch zu verbessern. In diesem Fall sollte `#define ABBRUCH` nicht gesetzt sein. Dadurch wird die Überprüfung der Flächenbedingung (vgl. Zeile 24) im Pseudocode) ausgeschaltet. Auch wird der Algorithmus nicht beendet, wenn das Modul mit der aktuell größten Laufzeit voll parallel ausgeführt wird (Zeile 26)). In diesem Fall wählt der Algorithmus das nächstkleinere Modul, dessen sequentielle Laufzeit oberhalb von

$$\frac{1}{N} \cdot \sum_{i=1}^{M} T(i, 1)$$

liegt. Der abgeänderte Algorithmus terminiert erst, wenn alle Module, die nach der eben angegebenen Klausel parallelisierbar sind, mit N Prozessoren ausgeführt werden. Durch diese Erweiterung bleibt die Größenordnung der Laufzeit des Algorithmus unverändert.

Die von der Optimierungsroutine benötigten Daten können auf zwei Wegen übergeben werden. Eine Möglichkeit ist die Angabe eines Dateinamens, welche in Tabellenform die einzelnen Laufzeiten enthält. Dabei steht der i-te Eintrag in der x-ten Zeile für die Laufzeit des x-ten Moduls mit i Prozessoren. Eventuell vorhandene Zeilen, in denen nicht mindestens eine Zahl steht, werden dabei nicht mitgezählt. Bei der Angabe eines Exponenten ist nur die Verwendung des kleinen ‚e‘ zulässig.
Im anderen Fall wird ein Zeiger auf eine Funktion vom Typ

```
double laufzeit( Modulnummer, Prozessorenanzahl )
```

übergeben. Die Zählung der Module geschieht fortlaufend und beginnt bei Eins.
Der berechnete Schedule wird entweder in einer Datei oder in den mit zu übergebenen Variablen `breite[]`, `linker[]` und `startzeit[]` ausgegeben. Der Aufbau der Ergebnisdatei kann

aus der Abbildung 79 entnommen werden. Bei der Verwendung der drei Variablen befinden sich die zu Modul x gehörenden Daten an der Feldposition x - 1. Der Speicherplatz für die Variablen wird mit `malloc` von der Optimierungsroutine reserviert.

```
Modul 002: von +0.0000e+000 bis +2.3163e+000 auf Prozessoren 001 bis 005
Modul 005: von +0.0000e+000 bis +1.9617e+000 auf Prozessoren 006 bis 008
Modul 003: von +0.0000e+000 bis +1.0138e+000 auf Prozessoren 009 bis 010
Modul 001: von +1.0138e+000 bis +1.7691e+000 auf Prozessoren 010 bis 010
Modul 006: von +1.0138e+000 bis +1.6155e+000 auf Prozessoren 009 bis 009
Modul 004: von +1.6155e+000 bis +1.9972e+000 auf Prozessoren 009 bis 009
```

Abbildung 79 - Aussehen der Ergebnisdatei.

9.12 Beispielgenerierung für „XP+MF“

Die automatische Erzeugung von Testbeispielen geschieht fast äquivalent zu der im Abschnitt 9.10 beschriebenen Vorgehensweise. Der erste Unterschied besteht im Aufbau der Eingabedatei. Sie enthält eine Spalte mehr, nämlich die Information, wie viele Prozessoren das Modul mindestens benötigt. Außerdem ändert sich die Bedeutung der ehemals zweiten (jetzt dritten) Spalte. Da die Laufzeit mit einem Prozessor nicht immer vorliegt, gibt sie die Laufzeit der Module bei der Ausführung mit der minimalen Prozessoranzahl an. Der Inhalt der Datei /make/prg2/bsp2.txt kann der Abbildung 80 entnommen werden.

```
PROZESSORENZAHL: {N}

MODULANZAHL: konstant {M}

MODULW   MIN_PROZ     LAUFZ_MIN_PROZ    PARALLELISIERBARKEIT
gl 1 3   gl 2 {0,8·N}  ex 1 100 1        BIS {0,4·N} P: no 0 {2/N} {1/N} {4/(10·N)};
                                         BIS {N} P: gl 0 {1/N}
ko 1.5   gl 2 {0,8·N}  no 0.5 4.7 2.2 1  BIS {0,7·N} P: ex 0 1 {N};
                                         BIS {N} P: gl 0 {2/N}
ko 0.5   ko 1          ex 1 100 0.7      BIS {N} P: no 0 1 0.4 0.3
```

Abbildung 80 - Datei /make/prg2/bsp2.txt - allgemeines Aussehen.

Die zweite Änderung befindet sich in der Routine zur Erzeugung der Modullaufzeiten. Ist die Ausführung des Moduls mit einem Prozessor möglich, so stimmt die Funktion von /src/prg2/bsp_bau2.c mit der von /src/prg1/bsp_bau1.c überein. Da im anderen Fall keine sequentielle Laufzeit vorhanden ist, kann die bisher verwendete Formel nicht weiter benutzt werden. Aus diesem Grund bestimmt der Algorithmus zuerst die Laufzeit $T(x, min\_Proz)$ unter Verwendung der minimal möglichen Prozessoranzahl aus der in der dritten Spalte angegebenen Verteilung. Alle weiteren Laufzeiten werden nach der Gleichung $T(x, i + 1) = (1 - z) \cdot T(x, i)$ berechnet. Dabei steht z für den ermittelten Parallelisierbarkeitsparameter. Wie bei den formbaren Modulen bewirkt z = 0, daß keine Laufzeitverbesserung eintritt. Wird allerdings z = 1 gesetzt, so erhält man hier als nächste Laufzeit den Wert Null. Da dies sinnlos ist, reagiert das Programm wie folgt: Sowie die ermittelte Laufzeit kleiner als `EPSILON` ist, wird sie auf `EPSILON` gesetzt. Standardmäßig ist `EPSILON` mittels `#define EPSILON DBL_MIN` auf den kleinstmöglichen Wert gesetzt. Durch die unterschiedliche Bedeutung des Wertes Eins bei der Parallelisierung ist es in fast allen Fällen nicht sinnvoll, mit einer Modulgattung formbare und nicht formbare Module zu erzeugen.

Auch beim Beispiel 2 ist die erste Modulgattung relativ gut bis zur Nutzung von 40% der vorhandenen Prozessoren parallelisierbar. Sollte ein Modul aus der ersten Modulgattung bereits eine Mindestprozessoranzahl von mehr als $0{,}4 \cdot N$ Prozessoren haben, so kann es überhaupt nicht parallelisiert werden. Im Gegensatz dazu steht die zweite Modulgattung. Ihre Module sind erst ab $0{,}7 \cdot N$ Prozessoren besser parallelisierbar. Es ist also nicht möglich, je ein Modul aus den ersten beiden Modulgattungen mit minimalen Parallelisierungsmehraufwand gleichzeitig auszuführen. Die formbaren Module der dritten Modulgattung sind nur mittelmäßig parallelisierbar. Sie werden demzufolge mit relativ hoher Wahrscheinlichkeit sequentiell ausgeführt. Aufgrund der relativ geringen Wahrscheinlichkeit für formbare Module, kann man mit ihnen nur sehr wenige Zeitintervalle, in denen ein einzelner Prozessor kein Modul ausführt, überbrücken.

9.13 Implementierung des Algorithmus „XP+MF“

In der Datei /src/prg2/opti2h.c auf der beiliegenden CD befindet sich der Quellcode des Verfahrens „XP+MF“ unter Verwendung des Positionierungsalgorithmus „HMZ“. Wie bei /src/prg1/opti1h.c ist auch hier die Einstellung `ABBRUCH` möglich. Die Eingabe der einzelnen Modullaufzeiten geschieht ebenfalls wahlweise über eine Textdatei oder eine Funktion. Zur Kennzeichnung, daß es sich um ein nicht formbares Modul handelt, welches mindestens k Prozessoren benötigt, muß als Laufzeit mit weniger als k Prozessoren der Wert Null in die Datei eingetragen oder von der Funktion zurückgegeben werden. Die Ausgabe des berechneten Schedules geschieht analog zu der im Abschnitt 9.11 vorgestellten Methode.
Der Quellcode des Algorithmus „XP+MF“ unter Verwendung des Positionierungsalgorithmus „BMZ“ ist unter /src/prg2/opti2b.c einsehbar. Die Anwendung des dort implementierten Verfahrens entspricht der soeben beschriebenen. Zusätzlich besteht noch die Möglichkeit, mittels `#define NICHTSORTIEREN` die Sortierung der Module nach der Laufzeit bei gleicher Prozessoranzahl abzustellen. Nur durch das Setzen von `NICHTSORTIEREN` erreicht /src/prg2/opti2b.c die im Abschnitt 2.2.1 angegebene Laufzeit. Anderenfalls wird eine Sortierung der Module mit gleicher Prozessoranzahl nach ihrer sequentiellen Laufzeit vorgenommen, was die Laufzeit des Algorithmus um $O(M \cdot \log(M))$ erhöht.

9.14 Laufzeitanalyse von „GzuB“

Um eine Angabe über die Laufzeit des Algorithmus „GzuB“ machen zu können, muß der Aufbau der einzelnen Datenstrukturen etwas exakter angegeben werden. In der Abbildung 81 ist deshalb der Algorithmus „GzuB“ nochmals angegeben.
In dieser ausführlichen Notation des Algorithmus übernimmt baum(x) die Funktion des zweiten Parameters der in K gespeicherten Tupel. Dadurch ist die parallele Vereinigung der einzelnen, zuvor erstellten Konstruktionsbäume (vgl. Zeile 12) in Abbildung 24) in kürzerer Zeit möglich. Anstatt in eing(x) den Eingangsgrad eines Knotens herabzuzählen, wird die Anzahl der bereits erstellten Instanzen in anz(x) aufsummiert. Der Wert in teiler(x) dient dazu, daß die Unterroutine „baubaum(x)“ nicht sofort nach dem Einfügen der letzten Instanz des Knotens x (in K) gestartet wird, sondern erst nachdem sich der Algorithmus wieder in der Routine baubaum(y) mit $x \in V_y$ und es gibt kein $z \in V_y$, so daß $x \in V_z$ ist, befindet. In teiler(x) steht also immer eine Vermutung zu welchem Teilgraph $G_{teiler(x)}$ der Knoten x gehört. Im Stack P liegen alle Knoten, welche nicht bearbeitet werden konnten, da noch nicht alle Vorgänger behandelt wurden. Wenn ein Knoten auf P geschrieben wird, ist noch nicht sicher, ob er tatsächlich nicht abgearbeitet werden kann. Sicher ist nur, daß sich der Rückgabewert am Ende ganz oben auf dem Stack befindet. Es müssen aber alle übersprungenen

Werte zwischengespeichert werden, damit man vor dem Verlassen der Routine die entsprechende Anzahl an parallelen Vereinigungen (vgl. Zeile 19) in Abbildung 24) durchführen kann.

```
1)  für x = 1 bis M
2)        anz(x) = 0; teiler(x) = 0; baum(x) = ∅
3)  anz(∞) = 0; definiere indeg(∞) = 1; teiler(∞) = 0
4)  x = min{ y : y ∈ V und indeg(y) = 0}
5)  baubaum( x )
6)  der Konstruktionsbaum zu G liegt in baum(∞)
7)  Ende
8)  Unterroutine baubaum( x )
9)  ist outdeg(x) = 0, so tue
10)       baum(∞) = Baum der nur aus x besteht
11)       Rücksprung unter Rückgabe von ∞
12) definiere Variablen K = ∅; P = ∅ lokal pro Unterroutinenaufruf
13) für alle y mit (x, y) ∈ E
14)       füge y hinten in K ein
15)       füge das Element (∅, ∅) (leerer Baum) hinten in baum(y) ein
16)       ist teiler(y) = 0, so teiler(y) = x
17)       anz(y) = anz(y) + 1
18) solange K ≠ ∅
19)       y = erstes Element von K; lösche das erste Element von K
20)       ist teiler(y) = x und anz(y) = indeg(y), so tue
21)            z = baubaum( y )
22)            C = letztes Element von baum(z); lösche das letzte Element von baum(z)
23)            C = ( ∏ D, alle Elemente D in baum(y) ) + C
24)            füge das Element C hinten in baum(z) ein
25)            ist teiler(z) = y, so teiler(z) =x
26)            anz(y) = -1
27)            füge z hinten in K ein
28)       anderenfalls
29)            ist anz(y) ≠ -1, so füge y hinten in P ein
30) y = letztes Element von P; lösche das letzte Element von P
31) C = letztes Element von baum(y); lösche das letzte Element von baum(y)
32) solange P ≠ ∅
33)       z = letztes Element von P; lösche das letzte Element von P
34)       ist z = y, so tue
35)            D = letztes Element von baum(y); lösche das letzte Element von baum(y)
36)            C = C ∥ D
37) C = x + C
38) füge das Element C hinten in baum(y) ein
39) Rücksprung unter Rückgabe von y
```

Abbildung 81 - Algorithmus „GzuB“ (ausführliche Form).

Zur Abschätzung der Laufzeit dieser Routine nutzt man die maximale Anzahl der in K eingefügten Elemente aus. In Zeile 14) der Abbildung 81 werden jeweils outdeg(x) Elemente erstellt. Da die Unterroutine „baubaum(x)“ mit jedem Knoten nur einmal gestartet wird (anschließend ist anz(x) = -1), sind dies maximal |E| Stück. In Zeile 27) wird noch ein Element in K eingefügt, insgesamt also maximal |V| Stück.
Da nur in Zeile 15) ein zusätzlicher Baum in baum(y) eingefügt wird, können bei der parallelen Vereinigung der Bäume in Zeile 23) auch jeweils nur indeg(y) Einzelbäume verknüpft werden. Insgesamt sind das also O(|E|) Elemente.
Der Algorithmus „GzuB“ hat somit eine Laufzeit von O(|E| + |V|). Bei der Implementierung des Verfahrens wurde zur Laufzeitreduzierung auf das explizite Einfügen der Hilfsknoten zum Auflösen der komplexen Vereinigungen verzichtet. Die notwendigen Informationen legt der Algorithmus in einer eindimensionalen Hilfsvariable ab. Dadurch ist nur ein Lauf über den Graphen notwendig.

9.15 Eine verbesserte Minimumsuche für die parallele Vereinigung beim Algorithmus „2M1“

Es sei a der Laufzeitvektor des Moduls A und c der von C. Wie bereits im Abschnitt 3.1.3.1 dargestellt wurde, ist nun ein Vektor d mittels

$$\forall i \in [1; N]: d(i) = \min \left\{ a(i) + c(i); \min_{j=1}^{i-1} \{ \max \{ a(j); c(i-j) \} \} \right\}$$

zu berechnen. Den laufzeitkritischen Teil dieser Formel stellt dabei die Bestimmung von

$$\min_{j=1}^{i-1} \{ \max \{ a(j); c(i-j) \} \} \qquad (1)$$

dar. Zuerst wird der Fall i = N betrachtet.
Aus der Voraussetzung weiß man, daß

$$a(1) \geqslant a(2) \geqslant ... \geqslant a(N-1) \geqslant a(N) \quad \text{und} \quad c(1) \geqslant c(2) \geqslant ... \geqslant c(N-1) \geqslant c(N)$$

gilt. Demzufolge ist die Bestimmung des k mit

$$\begin{array}{ccc} a(k) & \leqslant & a(k-1) \\ \wedge & & \vee \\ c(N-k) & \geqslant & c(N-k+1) \end{array}$$

in logarithmischer Zeit möglich. Sollte dabei $a(1) \leqslant c(N-1)$ sein, so wird die rechte Ungleichung ignoriert und k = 1 gesetzt. Die Lösung von (1) lautet demzufolge c(N - 1). Äquivalent verfährt man bei a(N - 1) > c(1). In diesem Fall setze man k = N. Für (1) erhält man somit a(N - 1).
Nach der Berechnung von k (mit $2 \leqslant k \leqslant N - 1$) kann man (1) durch einen Vergleich bestimmen. Ist

$$c(N-k) \geqslant a(k-1), \qquad (2)$$

so ergibt sich als Minimum a(k - 1). Anderenfalls lautet das Ergebnis c(N - k).
Hat man das k für ein i berechnet, so ist die Ermittlung von k’ für i - 1 sehr schnell möglich.
Fall 1: Für $2 \leqslant k \leqslant i - 1$ weiß man, daß

$$\begin{array}{ccc} a(k) & \leqslant & a(k-1) \\ \wedge & & \vee \\ c(i-k) & \geqslant & c(i-k+1) \end{array}$$

gilt. Das k’ soll nun

$$\begin{array}{ccc} a(k') & \leqslant & a(k' - 1) \\ \wedge & & \vee \\ c(i - 1 - k') & \geqslant & c(i - 1 - k' + 1) = c(i - k') \end{array}$$

erfüllen.

Fall 1.1: Es sei $k' \geqslant k + 1$.

Aus $k' \leqslant i - 1$ folgt $k \leqslant i - 2$. Außerdem gelten

$$a(k) \geqslant a(k' - 1) \quad \text{und} \quad c(i - k') \geqslant c(i - (k + 1)).$$

Die Wahl von k' garantiert

$$a(k' - 1) > c(i - k').$$

Setzt man die letzten drei Ungleichungen zu einer zusammen, so erhält man

$$a(k) > c(i - (k + 1)). \tag{3}$$

Da k so gewählt wurde, daß es

$$a(k) \leqslant c(i - k) \leqslant c(i - (k + 1))$$

erfüllt, ergibt sich ein Widerspruch zu (3).

Fall 1.2: Es sei $k' \leqslant k - 2$.

Aus $k' \geqslant 1$ folgt damit $k \geqslant 3$. Aufgrund der Ordnung der Werte in den Vektoren a und c gelten

$$a(k - 2) \leqslant a(k') \quad \text{und} \quad c(i - (k' + 1)) \leqslant c(i - (k - 1)).$$

Außerdem wurde für k'

$$a(k') \leqslant c(i - 1 - k') = c(i - (k' + 1))$$

gefordert. Mit Hilfe der drei soeben aufgestellten Ungleichungen folgt

$$a(k - 2) \leqslant c(i - (k - 1)) = c(i - k + 1). \tag{4}$$

Auf der anderen Seite gilt für k

$$a(k - 2) \geqslant a(k - 1) > c(i - k + 1),$$

was einen Widerspruch zu (4) darstellt.

Demzufolge muß $k' = k$ oder $k' = k - 1$ gesetzt werden. Betrachtet man die Bedingungen für k', so kann $k' = k$ nur gewählt werden, wenn

$$a(k - 1) > c(i - k)$$

ist. Andererseits muß bei der Wahl von $k' = k - 1$ die Ungleichung

$$a(k - 1) \leqslant c(i - 1 - (k - 1)) = c(i - k)$$

erfüllt sein. Die Bestimmung von k' ist somit in konstanter Zeit möglich.

Fall 2: Es sei $k = 1$.

Da bereits $a(1) \leqslant c(i - 1)$ gilt, kann k' ebenfalls auf Eins gesetzt werden.

Fall 3: Es gelte $k = i$.

Hier weiß man, daß $c(1) < a(i - 1)$ ist. Demzufolge muß man $k' = i - 1 = k - 1$ wählen.

Insgesamt benötigt man unter Verwendung des vorgestellten Verfahrens zur Berechnung des Vektors d bei der parallelen Vereinigung nur $O(N)$ Schritte.

9.16 Beispiele für den Algorithmus „2M1“

In diesem Abschnitt werden einige Beispiele angeführt, bei denen der Algorithmus „2M1“ tatsächlich nur die im Lemma 23 angegebenen Schranken zur Güte der Algorithmusschritte erreicht.

Zur besseren Lesbarkeit wird ohne Beschränkung der Allgemeinheit davon ausgegangen, daß das zusammengesetzte Modul D mit der Prozessoranzahl N (anstatt i im Lemma 23) als Ergebnis einer seriellen bzw. parallelen Vereinigung der Module A und C erstellt werden soll.

Da der im Punkt (a) von Lemma 23 angegebene Wert für $G(D, N)$ nicht unterboten werden kann, braucht hier kein Beispiel angegeben zu werden.

Bei der seriellen Vereinigung (Punkt (b)) hat man mehrere Möglichkeiten, ein Beispiel zu konstruieren, bei welchem nur der im Lemma 23 angegebene Gütefaktor $G(D, N)$ erreicht wird.

Bei der ersten Version wähle man die Module A und C so, daß $H(A, N) = H(C, N)$ und $O(A, N) = O(C, N)$ gilt. Dies ist z.B. für A = C erfüllt. Wie man sieht, ergibt sich dann für D = A + C:

$$H(D, N) = H(A, N) + H(C, N) = 2 \cdot H(A, N) \quad \text{und} \quad O(D, N) = O(A, N) + O(C, N) = 2 \cdot O(A, N).$$

Demzufolge ist auch

$$G(D, N) = \frac{H(D, N)}{O(D, N)} = \frac{H(A, N)}{O(A, N)} = G(A, N).$$

Äquivalent erhält man $G(D, N) = G(C, N)$. Somit existiert ein Beispiel, für welches

$$G(D, N) = \max\{ G(A, N); G(C, N) \}$$

gilt.

In einem zweiten Beispiel wird gezeigt, daß auch für $G(A, N) \neq G(C, N)$ ein $G(D, N)$ in der Größenordnung $\max\{ G(A, N); G(C, N) \}$ erreicht werden kann. Man wähle dazu zwei Module mit stark verschieden Laufzeiten, z.B. $H(A, N) = 1$ und $H(C, N) = \varepsilon$ mit hinreichend kleinem $\varepsilon > 0$, wobei der Gütefaktor des Moduls mit der kleineren Laufzeit nicht größer als der des anderen Moduls sei. In diesem Beispiel gelte also $G(A, N) \geqslant G(C, N)$. Die Berechnung des Ergebnisgütefaktors ergibt

$$G(D, N) = \frac{H(D, N)}{O(D, N)} = \frac{H(A, N) + H(C, N)}{O(A, N) + O(C, N)} = \frac{H(A, N) + H(C, N)}{\frac{H(A, N)}{G(A, N)} + \frac{H(C, N)}{G(C, N)}} = \frac{(H(A, N) + H(C, N)) \cdot G(A, N) \cdot G(C, N)}{H(A, N) \cdot G(C, N) + H(C, N) \cdot G(A, N)}.$$

Läßt man nun ε bzw. $H(C, N)$ gegen Null gehen (dabei muß $G(C, N)$ konstant bleiben), so erhält man:

$$\lim_{H(C, N) \to 0} \frac{(H(A, N) + H(C, N)) \cdot G(A, N) \cdot G(C, N)}{H(A, N) \cdot G(C, N) + H(C, N) \cdot G(A, N)} = \frac{H(A, N) \cdot G(A, N) \cdot G(C, N)}{H(A, N) \cdot G(C, N)} = G(A, N).$$

Wegen $G(A, N) \geqslant G(C, N)$ ist nun

$$G(D, N) \approx \max\{ G(A, N); G(C, N) \}.$$

Bei der parallelen Vereinigung (Punkt (c)) wird zuerst gezeigt, wie man ein zusammengesetztes Modul A konstruieren kann, für welches bei der Optimierung der in A enthaltenen einfachen Module mittels „2M1“ nur ein vorzugebender Gütefaktor von $G(A, N) \in \mathbb{Q}$ erreicht wird. Das vorzugebende $G(A, N)$ muß dazu im Intervall $[1; N)$ liegen. Um die Terme übersichtlich zu halten, werden die folgenden Abkürzungen eingeführt:

$$a = \frac{G(A, N) - 1}{G(A, N) \cdot (N - 1)}, \quad b = \frac{N - G(A, N)}{2 \cdot G(A, N) \cdot (N - 1)} - \varepsilon \quad \text{und} \quad c = \frac{N - G(A, N)}{2 \cdot G(A, N) \cdot (N - 1)}.$$

Dabei sei $\varepsilon > 0$ eine hinreichend kleine Zahl, welche die angegebenen vier Eigenschaften

$$\varepsilon < \frac{c}{N}, \quad \varepsilon < \frac{1}{N-2} \text{ (sofern } N \geq 3), \quad \frac{a}{\varepsilon} \in \mathbb{N} \quad \text{und} \quad \frac{c}{\varepsilon} \in \mathbb{N}$$

erfüllt. Diese Bedingungen garantieren unter anderem, daß $\varepsilon < c$ und somit $b > 0$ ist. Da $G(A, N) \in \mathbb{Q}$ gewählt wurde, liegen auch a und c im Bereich der rationalen Zahlen und gewährleisten die Existenz eines ε.
Als nächstes werden die einfachen Module I, J, L und P mit folgenden Laufzeiten definiert:

$$T(I, i) = \begin{cases} N \cdot b & : \text{falls } i < N \\ b & : \text{falls } i = N \end{cases} \qquad T(J, i) = (N + 1 - i) \cdot \varepsilon$$

$$T(L, i) = \varepsilon \qquad T(P, i) = c.$$

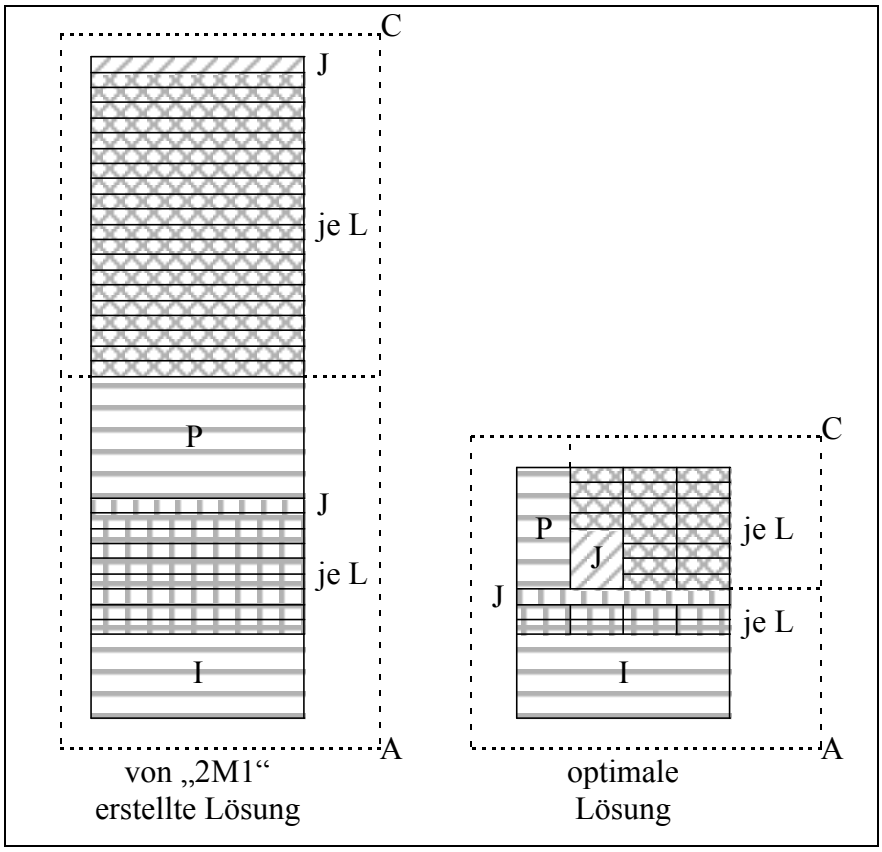


Abbildung 82 - Aussehen der unter D = A ‖ C (Punkt (c)) beschriebenen Module für N = 4.

Nun kann man A mittels

$$A = I + \left(J \parallel \prod_{j=1}^{\frac{N \cdot a}{\varepsilon}} L \right) + P$$

definieren.
Im nächsten Schritt werden die von „2M1“ ermittelten Laufzeiten für das zusammengesetzte Modul A bestimmt. Dazu betrachte man zuerst den Unterausdruck $J \parallel \prod_{j=1}^{\frac{N \cdot a}{\varepsilon}} L$. Durch Induktion über k wird gezeigt, daß

$$H\left(J \parallel \prod_{j=1}^{k} L, i\right) = (k + N + 1 - i) \cdot \varepsilon \qquad (1)$$

ist.
Setzt man k = 0, so erhält man die Laufzeiten für das Modul J.
Unter der Voraussetzung, daß (1) für k korrekt ist, wird nun die Gültigkeit von (1) für k + 1 gezeigt. Bei der parallelen Vereinigung vom zusammengesetzten Modul $J \parallel \prod_{j=1}^{k} L$ mit dem einfachen Modul L berechnet „2M1“ die Laufzeiten vom Modul $J \parallel \prod_{j=1}^{k+1} L$ nach folgender Formel:

$$H\left(J \parallel \prod_{j=1}^{k+1} L, i\right)$$

$$= \min\left\{ H\left(J \parallel \prod_{j=1}^{k} L, i\right) + H(L, i); \min_{h=1}^{i-1}\left\{ \max\left\{ H\left(J \parallel \prod_{j=1}^{k} L, h\right); H(L, i - h) \right\}\right\}\right\}$$

$$= \min\left\{ (k + N + 1 - i) \cdot \varepsilon + \varepsilon; \min_{h=1}^{i-1}\{ \max\{ (k + N + 1 - h) \cdot \varepsilon; \varepsilon \} \} \right\}$$

$$= \min\left\{ ((k + 1) + N + 1 - i) \cdot \varepsilon; \min_{h=1}^{i-1}\{ (k + N + 1 - h) \cdot \varepsilon \} \right\} \qquad (\text{da } h \leqslant i - 1 < N)$$

$$= \min\{ ((k + 1) + N + 1 - i) \cdot \varepsilon; (k + N + 1 - (i - 1)) \cdot \varepsilon \}$$

$$= ((k + 1) + N + 1 - i) \cdot \varepsilon.$$

Damit ist die Induktion beendet und die Gültigkeit von (1) gezeigt. Man sieht, daß von „2M1“ für $H\left(J \parallel \prod_{j=1}^{k+1} L, i\right)$ kein kleiner Wert als $H\left(J \parallel \prod_{j=1}^{k} L, i\right) + H(L, i)$ gefunden werden kann. Überträgt man diese Erkenntnis auf den Pseudocode des Algorithmus „2M1“ (siehe Abbildung 30), so erkennt man, daß die Bedingung in Zeile 25) nie erfüllt ist. Demzufolge werden die zwei Module $J \parallel \prod_{j=1}^{k} L$ und L nacheinander auf allen i Prozessoren ausgeführt.
Nun kann man sofort die von „2M1“ ermittelten Laufzeiten für das Modul A angeben: Für $i \in [1; N - 1]$ ist

$$H(A, i) = H(I, i) + H\left(J \parallel \prod_{j=1}^{\frac{N \cdot a}{\varepsilon}} L, i\right) + H(P, i)$$

$$= N \cdot b + \left(\frac{N \cdot a}{\varepsilon} + N + 1 - i\right) \cdot \varepsilon + c$$

$$= N \cdot \left(\frac{N - G(A, N)}{2 \cdot G(A, N) \cdot (N - 1)} - \varepsilon\right) + N \cdot \frac{G(A, N) - 1}{G(A, N) \cdot (N - 1)} + (N + 1 - i) \cdot \varepsilon + \frac{N - G(A, N)}{2 \cdot G(A, N) \cdot (N - 1)}$$

$$= \frac{N^2 - N \cdot G(A, N) + 2 \cdot N \cdot G(A, N) - 2 \cdot N + N - G(A, N)}{2 \cdot G(A, N) \cdot (N - 1)} - (i - 1) \cdot \varepsilon$$

$$= \frac{N^2 + N \cdot G(A, N) - N - G(A, N)}{2 \cdot G(A, N) \cdot (N - 1)} - (i - 1) \cdot \varepsilon$$

$$= \frac{(N - 1) \cdot (N + G(A, N))}{2 \cdot G(A, N) \cdot (N - 1)} - (i - 1) \cdot \varepsilon$$

$$= \frac{1}{2} \cdot \left(\frac{N}{G(A, N)} + 1 \right) - (i - 1) \cdot \varepsilon.$$

Für $H(A, N)$ erhält man

$$H(A, N) = H(I, N) + H\left(J \parallel \prod_{j=1}^{\frac{N \cdot a}{\varepsilon}} L, N \right) + H(P, N)$$

$$= b + \left(\frac{N \cdot a}{\varepsilon} + N + 1 - N \right) \cdot \varepsilon + c$$

$$= \frac{N - G(A, N)}{2 \cdot G(A, N) \cdot (N - 1)} - \varepsilon + N \cdot \frac{G(A, N) - 1}{G(A, N) \cdot (N - 1)} + \varepsilon + \frac{N - G(A, N)}{2 \cdot G(A, N) \cdot (N - 1)}$$

$$= \frac{N - G(A, N)}{G(A, N) \cdot (N - 1)} + N \cdot \frac{G(A, N) - 1}{G(A, N) \cdot (N - 1)}$$

$$= \frac{N - G(A, N) + N \cdot G(A, N) - N}{G(A, N) \cdot (N - 1)}$$

$$= 1.$$

Als nächstes wird die Gesamtlaufzeit $O(A, N)$ des optimalen Schedules bestimmt. Aus der Definition von A folgt, daß das einfache Modul I vor allen anderen in A enthaltenen einfachen Modulen ausgeführt werden muß. Es wird somit im optimalen Schedule auf N Prozessoren ausgeführt. Das einfache Modul P muß nach allen anderen in A enthaltenen einfachen Modulen ausgeführt werden. Zur Berechnung von $O(A, N)$ muß demzufolge nur noch das zusammengesetzte Modul $J \parallel \prod_{j=1}^{\frac{N \cdot a}{\varepsilon}} L$ betrachtet werden. Die minimale Fläche dieses zusammengesetzten Moduls beträgt

$$N \cdot \varepsilon + \frac{N \cdot a}{\varepsilon} \cdot \varepsilon = N \cdot \varepsilon + N \cdot a.$$

Wegen der Flächenbedingung kann die optimale Laufzeit dieses Moduls nicht unter $\varepsilon + a$ liegen. Wenn man jeweils N Module des Typs L gleichzeitig ausführt, entstehen $\frac{a}{\varepsilon}$ „Zeilen“ mit der Laufzeit ε. Dabei ist zu beachten, daß ε so gewählt wurde, daß $\frac{a}{\varepsilon} \in \mathbb{N}$ gilt. Anschließend startet man noch das einfache Modul J auf allen N Prozessoren. Die daraus resultierende Laufzeit entspricht der zuvor mit Hilfe der Flächenbedingung ermittelten und ist somit optimal. Insgesamt erhält man nun

$$O(A, N) = b + a + \varepsilon + c$$

$$= \frac{N - G(A, N)}{2 \cdot G(A, N) \cdot (N - 1)} - \varepsilon + \frac{G(A, N) - 1}{G(A, N) \cdot (N - 1)} + \varepsilon + \frac{N - G(A, N)}{2 \cdot G(A, N) \cdot (N - 1)}$$

$$= \frac{N - G(A, N) + G(A, N) - 1}{G(A, N) \cdot (N - 1)}$$

$$= \frac{1}{G(A, N)}.$$

Wie man sieht, liefert die angegebene Konstruktion für das Modul A den geforderten Gütefaktor von

$$\frac{H(A, N)}{O(A, N)} = G(A, N).$$

Nachdem das Modul A konstruiert wurde, kann man zu C übergehen. Unter Verwendung der oben definierten Hilfsmodule setze man

$$C = J \parallel \prod_{j=1}^{\frac{(N-1)\cdot c - N\cdot\varepsilon}{\varepsilon}} L.$$

Die Optimierung der in C enthaltenen einfachen Module mittels „2M1“ ergibt eine Laufzeit von

$$H(C, i) = \left(\frac{(N-1)\cdot c - N\cdot\varepsilon}{\varepsilon} + N + 1 - i\right)\cdot\varepsilon = \frac{1}{2}\cdot\left(\frac{N}{G(A, N)} - 1\right) - (i-1)\cdot\varepsilon.$$

Da

$$\varepsilon < \frac{c}{N}$$

gewählt wurde, ist für $N \geqslant 2$

$$H(C, i) > 0.$$

Wie man erkennen kann, gilt bei der angegebenen Wahl von A und C immer

$$H(A, 1) > H(C, 1).$$

Als nächstes wird die vom Algorithmus „2M1“ erzeugte Laufzeit von A ‖ C bei der Verwendung aller N Prozessoren berechnet:

$$H(A \parallel C, N) = \min\left\{ H(A, N) + H(C, N); \min_{i=1}^{N-1}\{ \max\{ H(A, i); H(C, N-i)\}\}\right\}$$

$$= \min\left\{ 1 + \frac{1}{2}\cdot\left(\frac{N}{G(A, N)} - 1\right) - (N-1)\cdot\varepsilon; \min_{i=1}^{N-1}\left\{ \max\left\{ \frac{1}{2}\cdot\left(\frac{N}{G(A, N)} + 1\right) - (i-1)\cdot\varepsilon; \right.\right.\right.$$

$$\left.\left.\left. \frac{1}{2}\cdot\left(\frac{N}{G(A, N)} - 1\right) - (N-i-1)\cdot\varepsilon \right\}\right\}\right\}.$$

Sollte $N \leqslant 2$ sein, so ist

$$1 \geqslant (N-2)\cdot\varepsilon$$

garantiert. Im Fall $N \geqslant 3$ folgt die Gültigkeit der soeben notierten Beziehung aus der Wahl von ε (siehe Forderungen an ε). Durch Erweitern der Ungleichung erhält man

$$1 \geqslant (2\cdot(N-1) - N)\cdot\varepsilon.$$

Außerdem weiß man, daß

$$i \leqslant N - 1$$

ist. Deshalb gilt auch

$$1 \geqslant (2\cdot i - N)\cdot\varepsilon$$

$$\Leftrightarrow \quad ½ - (i-1)\cdot\varepsilon \geqslant -½ - (N-i-1)\cdot\varepsilon$$

$$\Leftrightarrow \quad \frac{1}{2}\cdot\left(\frac{N}{G(A, N)} + 1\right) - (i-1)\cdot\varepsilon \geqslant \frac{1}{2}\cdot\left(\frac{N}{G(A, N)} - 1\right) - (N-i-1)\cdot\varepsilon.$$

Aus diesem Grund kann man

$$H(A \parallel C, N) = \min\left\{ 1 + \frac{1}{2}\cdot\left(\frac{N}{G(A, N)} - 1\right) - (N-1)\cdot\varepsilon; \min_{i=1}^{N-1}\left\{ \frac{1}{2}\cdot\left(\frac{N}{G(A, N)} + 1\right) - (i-1)\cdot\varepsilon \right\}\right\}$$

folgern. Das innere Minimum nimmt bei $i = N - 1$ den kleinsten Wert an, und es ergibt sich

$$H(A \parallel C, N) = \min\left\{ \frac{1}{2} \cdot \left(\frac{N}{G(A, N)} + 1\right) - (N - 1) \cdot \varepsilon; \frac{1}{2} \cdot \left(\frac{N}{G(A, N)} + 1\right) - (N - 1 - 1) \cdot \varepsilon \right\}$$

$$= \frac{1}{2} \cdot \left(\frac{N}{G(A, N)} + 1\right) - (N - 1) \cdot \varepsilon.$$

Bei der parallelen Vereinigung der zusammengesetzten Module A und C ermittelt „2M1“ die Nacheinanderausführung von A und C auf jeweils allen N Prozessoren als beste Möglichkeit. Für den Fall $N = 4$ ist der von „2M1“ erstellte Schedule in der Abbildung 82 angegeben.

Zum Abschluß der Konstruktion eines Gegenbeispiels bleibt noch die Bestimmung der optimalen Laufzeit von A ‖ C. Dazu ist es ausreichend zu zeigen, daß das Modul C bei der Verwendung von $N - 1$ benachbarten Prozessoren die Laufzeit c nicht überschreitet. In diesem Fall kann das Modul C gleichzeitig mit dem in Modul A enthaltenen einfachen Modul P ausgeführt werden, und es gilt

$$O(A \parallel C, N) = O(A, N) = \frac{1}{G(A, N)}.$$

Für den Gütefaktor $G(A \parallel C, N)$ folgt dann

$$G(A \parallel C, N) = \frac{H(A \parallel C, N)}{O(A \parallel C, N)} = \frac{\frac{1}{2} \cdot \left(\frac{N}{G(A, N)} + 1\right) - (N - 1) \cdot \varepsilon}{\frac{1}{G(A, N)}} = ½ \cdot (N + G(A, N)) - (N - 1) \cdot G(A, N) \cdot \varepsilon.$$

Läßt man ε gegen Null laufen, so ist eine beliebige Annäherung an die im Beweis angegebene Schranke möglich.

Es bleibt noch zu zeigen, daß die im Modul C enthaltenen einfachen Module auf $N - 1$ Prozessoren in maximal c Zeiteinheiten ausgeführt werden können. Dazu starte man das in C enthaltene einfache Modul J auf einem Prozessor. Da es $N \cdot \varepsilon$ Zeiteinheiten benötigt, können gleichzeitig auf jedem der $N - 2$ verbleibenden Prozessoren jeweils N Module des Typs L laufen. Zum Zeitpunkt $N \cdot \varepsilon$ verbleiben also noch

$$\frac{(N - 1) \cdot c - N \cdot \varepsilon}{\varepsilon} - (N - 2) \cdot N = \frac{c}{\varepsilon} \cdot (N - 1) - N - N^2 + 2 \cdot N = \frac{c}{\varepsilon} \cdot (N - 1) - N \cdot (N - 1) = \left(\frac{c}{\varepsilon} - N\right) \cdot (N - 1)$$

Module des Typs L. Da ε so gewählt wurde, daß

$$\frac{c}{\varepsilon} > N$$

gilt, ist die Anzahl der übrigen Module auch positiv. Außerdem ist $\frac{c}{\varepsilon}$ eine natürliche Zahl und somit auch $\frac{c}{\varepsilon} - N \in \mathbb{N}$. Da zum Zeitpunkt $N \cdot \varepsilon$ das Modul J beendet wird, stehen anschließend $N - 1$ Prozessoren zur Ausführung der verbleibenden $\left(\frac{c}{\varepsilon} - N\right) \cdot (N - 1)$ Module des Typs L zur Verfügung. Demzufolge können die verbleibenden Module in $\frac{c}{\varepsilon} - N$ „Zeilen“ mit der Laufzeit ε ausgeführt werden. Das zusammengesetzte Modul C kann also in

$$N \cdot \varepsilon + \left(\frac{c}{\varepsilon} - N\right) \cdot \varepsilon = c$$

Zeiteinheiten auf $N - 1$ Prozessoren ausgeführt werden.

Die im Lemma 23(c) angegebene Schranke kann für $N = 2$ sogar exakt erreicht werden. Dazu definiere man die einfachen Module C, I und J mit folgenden Laufzeiten:

$H(C, 1) = 1$; $H(C, 2) = 1$; $H(I, 1) = 1$; $H(I, 2) = 1$; $H(J, 1) = 2$; $H(J, 2) = 1$.

Das zusammengesetzte Modul A wird mittels A = I + J definiert. Man erkennt, daß

$$H(A, 1) = 3; \qquad H(A, 2) = 2 \qquad \text{und} \qquad O(A, 2) = 2$$

ist. Demzufolge ergibt sich $G(A, N) = 1$. Bildet man nun D = A ‖ C, so erhält man

$$H(D, 2) = \min\{ H(A, 2) + H(C, 2); \max\{ H(A, 1); H(C, 1) \} \} = 3.$$

Der optimale Schedule für die im zusammengesetzten Modul D enthaltenen einfachen Module ist in Abbildung 83 dargestellt. Es gilt also

$$O(D, 2) = 2 \qquad \text{und} \qquad G(D, 2) = \frac{H(D, 2)}{O(D, 2)} = \frac{3}{2}.$$

Die Ungleichung im Lemma 23(c) liefert:

$$G(D, 2) \leqslant ½ \cdot (2 + G(A, 2)) = \frac{3}{2}.$$

Somit existiert ein Beispiel, bei dem die in Lemma 23(c) angegebene Schranke sogar exakt erreicht wird.

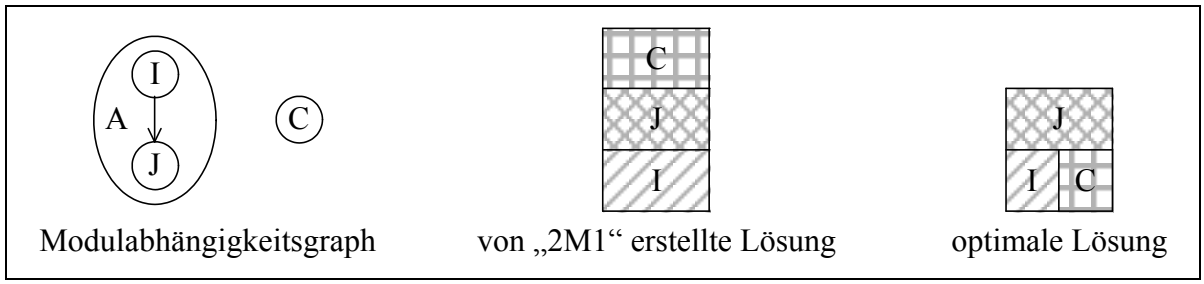


Abbildung 83 - Beispiel, bei welchem die angegebene Schranke exakt erreicht wird.

9.17 Durchschnittliche Ergebnisgüte von „2M1“

In diesem Abschnitt soll versucht werden, eine obere Schranke $\overline{G}(D, N)$ für die vom Algorithmus „2M1“ durchschnittlich erzielte Ergebnisgüte $\widetilde{G}(D, N)$ anzugeben. Da der Algorithmus „2M1“ bei $N = 1$ den optimalen Schedule findet, wird im folgenden von $N \geqslant 2$ ausgegangen.

Wie bereits im Beweis von Lemma 23(b) gezeigt wurde, gilt bei der seriellen Vereinigung von A und C zum Modul D die folgende Gleichung:

$$G(D, N) = \frac{H(A, N) + H(C, N)}{O(A, N) + O(C, N)} = \frac{H(A, N) + H(C, N)}{\frac{H(A, N)}{G(A, N)} + \frac{H(C, N)}{G(C, N)}} = \frac{G(A, N) \cdot G(C, N) \cdot (H(A, N) + H(C, N))}{H(A, N) \cdot G(C, N) + H(C, N) \cdot G(A, N)}$$

$$= \frac{G(A, N) \cdot G(C, N) \cdot \left(1 + \frac{H(C, N)}{H(A, N)}\right)}{G(C, N) + \frac{H(C, N)}{H(A, N)} \cdot G(A, N)}.$$

Aus der Flächenbedingung und Lemma 23(a) folgt

$$O(A, 1) \leqslant N \cdot O(A, N) \quad \Leftrightarrow \quad H(A, 1) \leqslant N \cdot O(A, N)$$

$$\Leftrightarrow \quad H(A, 1) \leqslant N \cdot \frac{H(A, N)}{G(A, N)} \quad \Leftrightarrow \quad H(A, N) \geqslant \frac{H(A, 1) \cdot G(A, N)}{N}.$$

Analog erhält man

$$H(C, N) \geqslant \frac{H(C, 1) \cdot G(C, N)}{N}.$$

Außerdem sind die Aussagen $H(A, N) \leqslant H(A, 1)$ und $H(C, N) \leqslant H(C, 1)$ immer erfüllt. Nun kann man $G(D, N)$ mittels

$$G(D, N) \leqslant \frac{G(A, N) \cdot G(C, N) \cdot \left(1 + \frac{H(C, 1) \cdot N}{H(A, 1) \cdot G(A, N)}\right)}{G(C, N) + \frac{H(C, 1) \cdot G(C, N)}{N \cdot H(A, 1)} \cdot G(A, N)} = \frac{N \cdot \left(H(A, 1) \cdot G(A, N) + H(C, 1) \cdot N\right)}{H(A, 1) \cdot N + H(C, 1) \cdot G(A, N)}$$

von oben abschätzen. Äquivalent erhält man

$$G(D, N) \leqslant \frac{N \cdot \left(H(C, 1) \cdot G(C, N) + H(A, 1) \cdot N\right)}{H(C, 1) \cdot N + H(A, 1) \cdot G(C, N)}.$$

Zur Minimierung des Rechenaufwandes überlege man sich, wann welche der beiden angegebenen Schranken den kleineren Wert liefert. Im ersten Fall sei $H(C, 1) \leqslant H(A, 1)$. Dann gilt

$$H(C, 1)^2 \cdot \left(N^2 - G(A, N) \cdot G(C, N)\right) \leqslant H(A, 1)^2 \cdot \left(N^2 - G(A, N) \cdot G(C, N)\right)$$

$$\Leftrightarrow \quad H(A, 1)^2 \cdot G(A, N) \cdot G(C, N) + H(C, 1)^2 \cdot N^2 \leqslant H(C, 1)^2 \cdot G(A, N) \cdot G(C, N) + H(A, 1)^2 \cdot N^2$$

$$\Leftrightarrow H(A, 1) \cdot G(A, N) \cdot N \cdot H(C, 1) + H(A, 1) \cdot G(A, N) \cdot H(A, 1) \cdot G(C, N)$$
$$+ H(C, 1) \cdot N \cdot N \cdot H(C, 1) + H(C, 1) \cdot N \cdot H(A, 1) \cdot G(C, N)$$
$$\leqslant H(C, 1) \cdot G(C, N) \cdot N \cdot H(A, 1) + H(C, 1) \cdot G(C, N) \cdot H(C, 1) \cdot G(A, N)$$
$$+ H(A, 1) \cdot N \cdot N \cdot H(A, 1) + H(A, 1) \cdot N \cdot H(C, 1) \cdot G(A, N)$$

$$\Leftrightarrow \quad \frac{H(A, 1) \cdot G(A, N) + H(C, 1) \cdot N}{N \cdot H(A, 1) + H(C, 1) \cdot G(A, N)} \leqslant \frac{H(C, 1) \cdot G(C, N) + H(A, 1) \cdot N}{N \cdot H(C, 1) + H(A, 1) \cdot G(C, N)}.$$

Aus Symmetriegründen kann die gleiche Schlußfolgerung auch mit vertauschten Modulen A und C durchgeführt werden.
Die für $G(D, N)$ erstellte Abschätzung hängt nur noch von den bereits erzielten Gütefaktoren und der sequentiellen Laufzeit der Eingangsmodule ab. Geht man nun davon aus, daß die Modullaufzeiten bei der Verwendung von einem Prozessor einer bestimmten Verteilung v unterliegen, bleiben nur noch $G(A, N)$ und $G(C, N)$ als Unbekannte übrig und man erhält:

$\widetilde{G}(D, N) \leqslant \overline{G}(D, N)$

$$= \lim_{z \to \infty} \frac{1}{z^2} \cdot \sum_{a=1}^{z} \sum_{c=1}^{z} \min \left\{ \frac{N \cdot \left(f\left(\frac{a}{z}\right) \cdot G(A, N) + f\left(\frac{c}{z}\right) \cdot N\right)}{f\left(\frac{a}{z}\right) \cdot N + f\left(\frac{c}{z}\right) \cdot G(A, N)}; \frac{N \cdot \left(f\left(\frac{c}{z}\right) \cdot G(C, N) + f\left(\frac{a}{z}\right) \cdot N\right)}{f\left(\frac{c}{z}\right) \cdot N + f\left(\frac{a}{z}\right) \cdot G(C, N)} \right\}.$$

In der angegebenen Ungleichung steht f für eine monoton steigende Funktion, mit welcher aus einer über (0; 1] gleichverteilten Zufallszahl eine zufällige Modullaufzeit, die der Verteilung v unterliegt, berechnet werden kann. Mit Hilfe der im Abschnitt 9.10.2 gewonnenen Resultate ist bei einer Exponentialverteilung

$$f(x) = -\frac{1}{\lambda} \cdot \ln(1 - x)$$

gegeben. Dazu wurde das Zielintervall auf (0; ∞) gesetzt. Will man den gleichen Bereich mit der Normalverteilung abdecken, so kann man die unter 9.10.3 angegebene Formel nicht verwenden, sondern muß mit Hilfe eines Näherungsverfahrens eine neue erstellen. Hier wird das Newtonverfahren genutzt. Dazu bezeichne

$$q(t) = \frac{1}{\sigma \cdot \sqrt{2 \cdot \pi}} \cdot e^{\left(-\frac{(t - \alpha)^2}{2 \cdot \sigma^2}\right)}$$

die Dichtefunktion der Normalverteilung. Wie im Abschnitt 9.10.3 lautet auch hier der Ansatz

$$x = \frac{\int_0^{f(x)} q(t)\,dt}{\int_0^{\infty} q(t)\,dt}.$$

Da die Lösung des Nennerintegrals an dieser Stelle mit der Trapezregel nicht möglich ist, sind erst noch einige Umformungen des Terms notwendig:

$$\int_0^{\infty} q(t)\,dt = \int_{-\alpha}^{\infty} q(t+\alpha)\,dt = \frac{1}{2} + \int_{-\alpha}^{0} q(t+\alpha)\,dt.$$

Man erhält somit

$$0 = \int_0^{f(x)} q(t)\,dt - x \cdot \left(\frac{1}{2} + \int_{-\alpha}^{0} q(t+\alpha)\,dt\right)$$

als Ausgangspunkt für das Newtonsche Näherungsverfahren. Die iterative Berechnung von f(x) kann demzufolge nach folgender Formel geschehen:

$$f(x)_{i+1} = f(x)_i - \frac{\int_0^{f(x)_i} q(t)\,dt - x \cdot \left(\frac{1}{2} + \int_{-\alpha}^{0} q(t+\alpha)\,dt\right)}{q(f(x)_i)}.$$

Zur Bestimmung des Nenners wurde die Beziehung

$$\frac{\partial}{\partial b}\left(\int_a^b q(t)\,dt\right) = q(b)$$

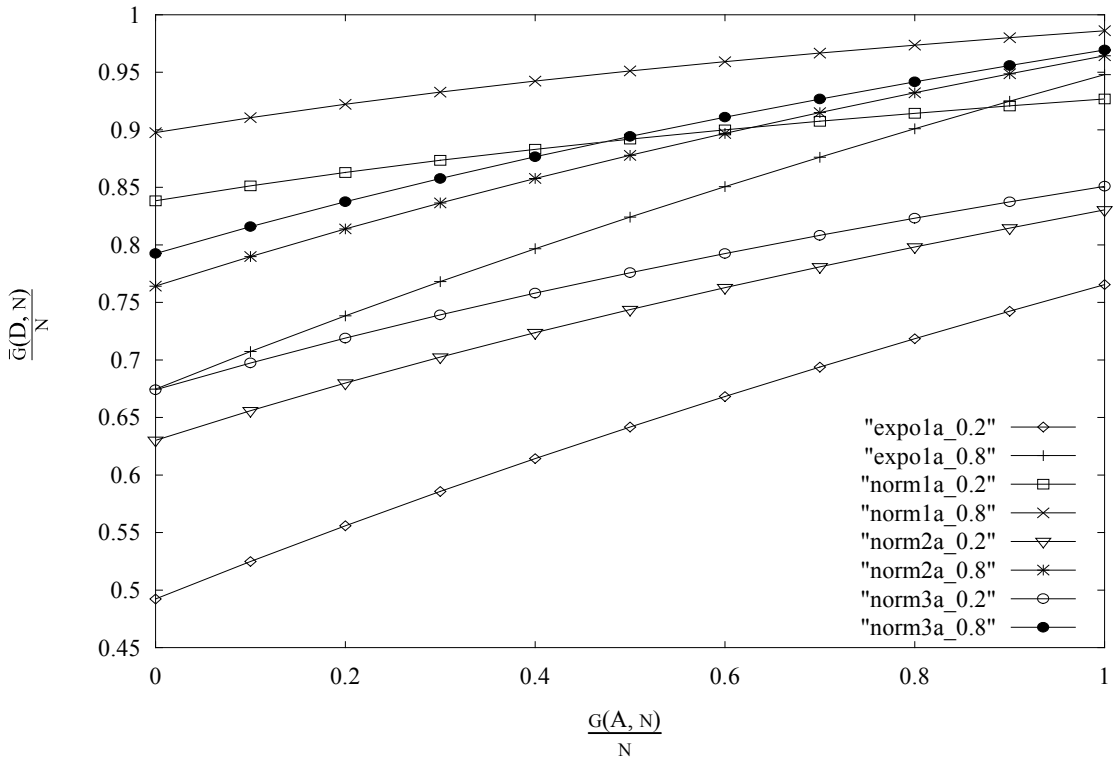


Abbildung 84 - serielle Vereinigung (Rückführung auf sequentielle Laufzeiten).

verwendet. Die Iteration bricht ab, wenn

$$\left| f(x)_{i+1} - f(x)_i \right| < \texttt{EPSILON}$$

ist. Alle verbliebenen Integrale können nun mit Hilfe der Trapezregel ermittelt werden. Die Anzahl der dazu verwendeten Stützstellen ist durch `#define INTEG 100` vorzugeben.
Man erkennt, daß eine allgemeine Lösung der erstellten Abschätzung nicht ohne weiteres möglich ist. Aus diesem Grund wurde ein kleines Programm /src/guete3/guete3.c (siehe beiliegende CD) geschrieben, welches bei vorgegebenem z (`#define SCHRITT 100`) eine entsprechende Wertetabelle zurückliefert. Die Darstellung der Gütefaktoren erfolgt als Vielfache von N (es wird also $\frac{1}{N} \cdot G(A, N)$ verwendet). Dadurch spielt die Gesamtprozessoranzahl bei der Rechnung keine Rolle.
Ersetzt man den Funktionsaufruf f(x) in der Formel zur Berechnung von $\overline{G}(D, N)$ durch die Exponentialverteilung, so kann man den Parameter λ kürzen. Demzufolge erhält man für alle λ das gleiche Resultat. Dieses ist unter /res/guete3/expo1*.txt auf der beigefügten CD abgelegt. Verwendet man hingegen die Normalverteilung, so muß man Werte für α und σ angeben. Für die Beispiele wurden die in der Tabelle angegebenen Werte genutzt.

Name der Datei	α	σ
/res/guete3/norm1*.txt	0,5	0,1
/res/guete3/norm2*.txt	0,3	0,2
/res/guete3/norm3*.txt	1,0	0,5

Um die Werte zweidimensional darstellen zu können, wurde $\frac{G(C, N)}{N}$ auf 0,2 bzw. 0,8 gesetzt. Eine vollständige Tabelle ist auf der CD enthalten. Der Wert $\frac{G(A, N)}{N} = 0$ ist in der Praxis nicht erreichbar, kann aber als Grenzwert mit Hilfe der angegebenen Formel berechnet werden.

In der Abbildung 84 erkennt man, daß die für den Wert $\frac{\tilde{G}(D, N)}{N}$ berechnete obere Schranke $\frac{\overline{G}(D, N)}{N}$ sehr grob ist. In vielen Fällen gilt sogar

$$\frac{\overline{G}(D, N)}{N} \geqslant \max\left\{ \frac{G(A, N)}{N}; \frac{G(C, N)}{N} \right\}.$$

Außerdem ist der Einfluß der verwendeten Verteilungsfunktion sehr hoch. Gerade bei kleinem $G(A, N)$ bzw. kleinem $G(C, N)$ spielt in der erstellten Formel zur Berechnung von $\overline{G}(D, N)$ der Quotient $\frac{H(C, 1)}{H(A, 1)}$ bzw. sein reziproker Wert $\frac{H(A, 1)}{H(C, 1)}$ eine dominierende Rolle. Aus diesem Grund ergeben sich bei den Verteilungen mit großer Streuung bessere Abschätzungen für $\overline{G}(D, N)$. Auch vergrößert sich der Wert $\overline{G}(D, N)$ bei der Verwendung von großen Modullaufzeiten. Demzufolge liefert ein großer Erwartungswert α eine schlechtere Abschätzung als ein kleiner. Die Veränderung des Parameters $G(C, N)$ bewirkt hingegen nur eine Verschiebung der Kurve.
Entsteht das Modul D über eine parallele Vereinigung, so kann für $G(D, N)$ die folgende obere Schranke notiert werden:

$$G(D, N) = \frac{H(D, N)}{O(D, N)} \leqslant \frac{\min\{ H(A, N) + H(C, N); \max\{ H(A, 1); H(C, 1) \} \}}{\max\left\{ \frac{H(A, 1) + H(C, 1)}{N}, \frac{H(A, N)}{G(A, N)}, \frac{H(C, N)}{G(C, N)} \right\}} \qquad (1)$$

(vgl. Gleichungen (2), (3) und (4) im Beweis zu Lemma 23(c)). Zum Abschätzen dieses Terms wird wiederum die Monotonieeigenschaft $H(A, N) \leqslant H(A, 1)$ verwendet. Es gilt somit

$$G(D, N) \leqslant \frac{H(A, N) + H(C, N)}{\max\left\{ \frac{H(A, 1) + H(C, 1)}{N}, \frac{H(A, N)}{G(A, N)}, \frac{H(C, N)}{G(C, N)} \right\}}$$

$$\leqslant \frac{H(A, N)}{\max\left\{ \frac{H(A, 1) + H(C, 1)}{N}, \frac{H(A, N)}{G(A, N)} \right\}} + \frac{H(C, N)}{\max\left\{ \frac{H(A, 1) + H(C, 1)}{N}, \frac{H(C, N)}{G(C, N)} \right\}}$$

$$= \min\left\{ \frac{N \cdot H(A, N)}{H(A, 1) + H(C, 1)}; G(A, N) \right\} + \min\left\{ \frac{N \cdot H(C, N)}{H(A, 1) + H(C, 1)}; G(C, N) \right\}$$

$$\leqslant \min\left\{ \frac{N \cdot H(A, 1)}{H(A, 1) + H(C, 1)}; G(A, N) \right\} + \min\left\{ \frac{N \cdot H(C, 1)}{H(A, 1) + H(C, 1)}; G(C, N) \right\}.$$

Außerdem ist nach (1)

$$G(D, N) \leqslant \frac{\max\{ H(A, 1); H(C, 1) \}}{\frac{H(A, 1) + H(C, 1)}{N}}.$$

Insgesamt erhält man somit

$$G(D, N) \leqslant \min\left\{ \min\left\{ \frac{N \cdot H(A, 1)}{H(A, 1) + H(C, 1)}; G(A, N) \right\} + \min\left\{ \frac{N \cdot H(C, 1)}{H(A, 1) + H(C, 1)}; G(C, N) \right\}; \right.$$

$$\left. \frac{N \cdot \max\{ H(A, 1); H(C, 1) \}}{H(A, 1) + H(C, 1)} \right\},$$

bzw. nach dem Einsetzen der Verteilungsfunktion:

$$\tilde{G}(D, N) \leqslant \overline{G}(D, N) = \lim_{z \to \infty} \frac{1}{z^2} \cdot \sum_{a=1}^{z} \sum_{c=1}^{z} \min\left\{ \min\left\{ \frac{N \cdot f\left(\frac{a}{z}\right)}{f\left(\frac{a}{z}\right) + f\left(\frac{c}{z}\right)}; G(A, N) \right\} \right.$$

$$\left. + \min\left\{ \frac{N \cdot f\left(\frac{c}{z}\right)}{f\left(\frac{a}{z}\right) + f\left(\frac{c}{z}\right)}; G(C, N) \right\}; \frac{N \cdot \max\left\{ f\left(\frac{a}{z}\right); f\left(\frac{c}{z}\right) \right\}}{f\left(\frac{a}{z}\right) + f\left(\frac{c}{z}\right)} \right\}.$$

Dieser Term wurde ebenfalls mit dem Programm /src/guete3/guete3.c unter Verwendung der bereits oben notierten Beispiele berechnet. Es ergab sich das in der Abbildung 85 dargestellte Bild.

Hat die Verteilungsfunktion nur eine geringe Streuung, also $H(A, 1) \approx H(C, 1)$, so liefert der letzte Term den Wert $\frac{1}{2} \cdot N$. Deshalb liegen viele Punkte in der Abbildung 85 im Intervall von 0,5 bis 0,6. Sind dagegen die Gütefaktoren $G(A, N)$ und $G(C, N)$ klein, so entsteht $\overline{G}(D, N)$ aus ihrer Summe. Das erklärt den starken Anstieg der Kurven mit $\frac{G(C, N)}{N} = 0{,}2$ bei kleinem $G(A, N)$. Bei großen, aber verschiedenen Laufzeiten der Module A und C erreicht man mit dem letzten Term in der Abschätzung einen kleineren Quotienten als bei kleinen $H(A, 1)$ und $H(C, 1)$ mit der gleichen Differenz.

Anstatt die einzelnen Modullaufzeiten auf $H(A, 1)$ bzw. $H(C, 1)$ zurückzuführen, kann man auch $H(A, N)$ und $H(C, N)$ nutzen. In diesem Fall bezieht sich die über die Funktion f vorgegebene Verteilung nicht auf die sequentiellen, sondern auf die voll parallelen Laufzeiten. Die Abschätzung für D = A + C gestaltet sich einfach, da man die bereits oben notierte Formel direkt verwenden kann:

$$G(D, N) = \frac{G(A, N) \cdot G(C, N) \cdot (H(A, N) + H(C, N))}{H(A, N) \cdot G(C, N) + H(C, N) \cdot G(A, N)},$$

$$\text{bzw.} \quad \tilde{G}(D, N) = \overline{G}(D, N) = \lim_{z \to \infty} \frac{1}{z^2} \cdot \sum_{a=1}^{z} \sum_{c=1}^{z} \frac{G(A, N) \cdot G(C, N) \cdot \left(f\left(\frac{a}{z}\right) + f\left(\frac{c}{z}\right) \right)}{f\left(\frac{a}{z}\right) \cdot G(C, N) + f\left(\frac{c}{z}\right) \cdot G(A, N)}.$$

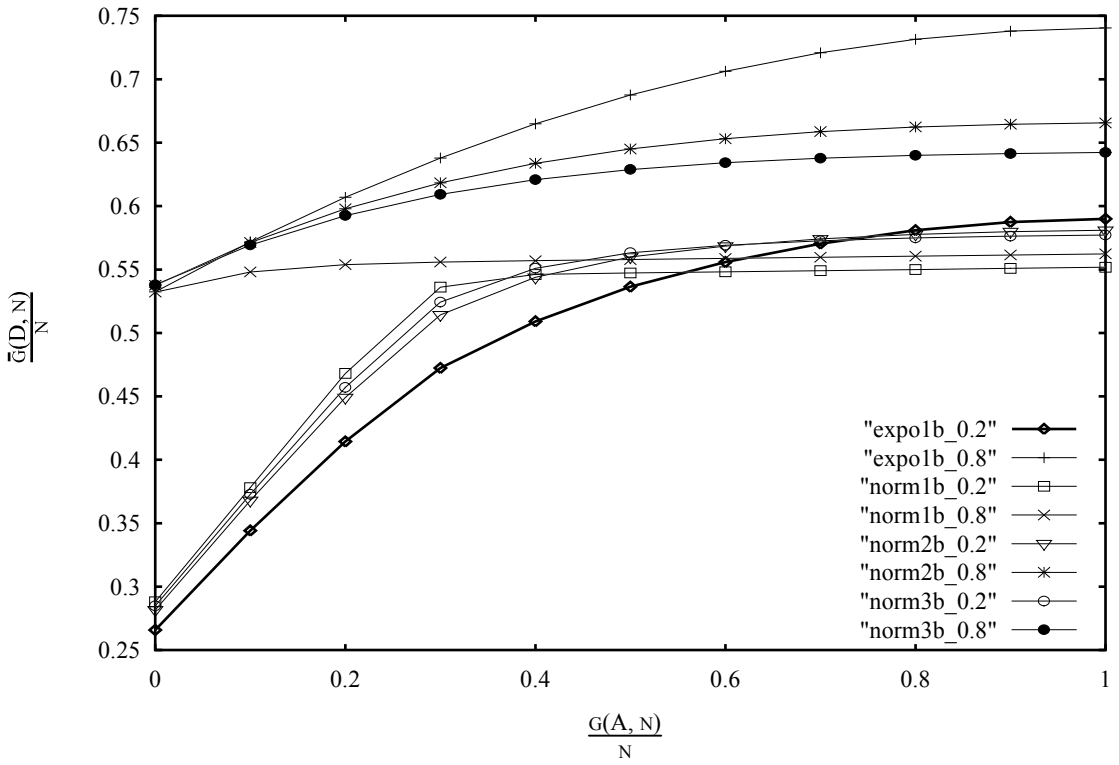


Abbildung 85 - parallele Vereinigung (Rückführung auf sequentielle Laufzeiten).

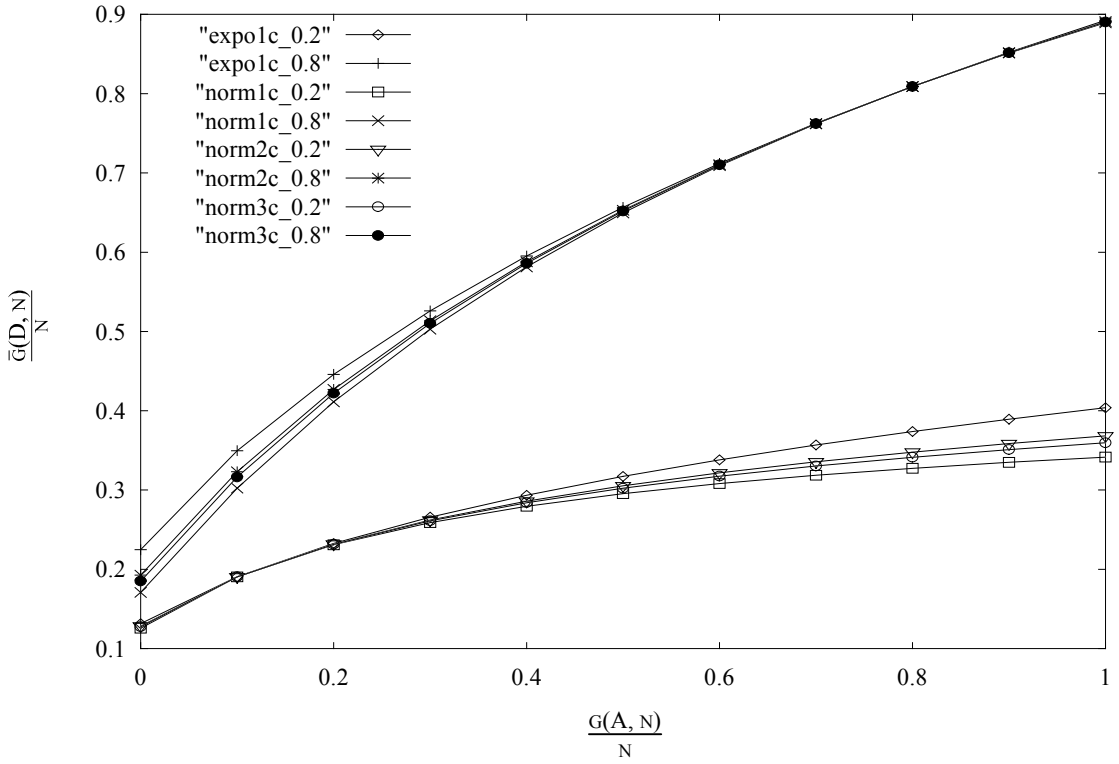


Abbildung 86 - serielle Vereinigung (Rückführung auf parallele Laufzeiten).

Die Abbildung 86 stellt die soeben aufgestellte Beziehung graphisch dar. Man sieht sehr schön, daß im Fall $G(A, N) = G(C, N)$ unabhängig von der Verteilung $\overline{G}(D, N) = G(A, N)$ gilt. Im Fall $G(A, N) \neq G(C, N)$ liegt $\overline{G}(D, N)$ immer zwischen $G(A, N)$ und $G(C, N)$. Dies läßt sich auch nachweisen: Da die Formel zur Berechnung von $G(D, N)$ symmetrisch bzgl. der Module A und C ist, gelte ohne Beschränkung der Allgemeinheit $G(A, N) \leqslant G(C, N)$. Somit erhält man:

$$G(A, N) \leqslant G(C, N)$$

$$\Leftrightarrow \quad H(A, N) \cdot G(C, N) + H(C, N) \cdot G(A, N) \leqslant H(A, N) \cdot G(C, N) + H(C, N) \cdot G(C, N)$$

$$\Leftrightarrow \qquad 1 \leqslant \frac{G(C, N) \cdot (H(A, N) + H(C, N))}{H(A, N) \cdot G(C, N) + H(C, N) \cdot G(A, N)}$$

$$\Leftrightarrow \qquad G(A, N) \leqslant \frac{G(A, N) \cdot G(C, N) \cdot (H(A, N) + H(C, N))}{H(A, N) \cdot G(C, N) + H(C, N) \cdot G(A, N)} = G(D, N)$$

und

$$G(A, N) \leqslant G(C, N)$$

$$\Leftrightarrow \qquad H(A, N) \cdot G(A, N) + H(C, N) \cdot G(A, N) \leqslant H(A, N) \cdot G(C, N) + H(C, N) \cdot G(A, N)$$

$$\Leftrightarrow \qquad \frac{G(A, N) \cdot (H(A, N) + H(C, N))}{H(A, N) \cdot G(C, N) + H(C, N) \cdot G(A, N)} \leqslant 1$$

$$\Leftrightarrow \qquad G(D, N) = \frac{G(A, N) \cdot G(C, N) \cdot (H(A, N) + H(C, N))}{H(A, N) \cdot G(C, N) + H(C, N) \cdot G(A, N)} \leqslant G(C, N).$$

Im Vergleich zur Abbildung 84 liefert diese Darstellung wesentlich kleinere Werte für die resultierende Ergebnisgüte des Moduls D.

Als letzter Fall wird noch D = A || C unter Rückführung auf parallele Laufzeiten behandelt. Wie man bereits am Anfang dieses Abschnittes gesehen hat, gelten die folgenden Ungleichungen:

$$H(A, 1) \leqslant N \cdot \frac{H(A, N)}{G(A, N)}, \quad H(A, 1) \geqslant H(A, N) \quad \text{und} \quad (1).$$

Demzufolge kann man

$$G(D, N) \leqslant \frac{H(A, N)}{\max\left\{ \frac{H(A, 1) + H(C, 1)}{N}; \frac{H(A, N)}{G(A, N)}; \frac{H(C, N)}{G(C, N)} \right\}} + \frac{H(C, N)}{\max\left\{ \frac{H(A, 1) + H(C, 1)}{N}; \frac{H(A, N)}{G(A, N)}; \frac{H(C, N)}{G(C, N)} \right\}}$$

$$\leqslant \min\left\{ \frac{N \cdot H(A, N)}{H(A, 1) + H(C, 1)}; G(A, N); \frac{H(A, N) \cdot G(C, N)}{H(C, N)} \right\} + \min\left\{ \frac{N \cdot H(C, N)}{H(A, 1) + H(C, 1)}; \frac{H(C, N) \cdot G(A, N)}{H(A, N)}; G(C, N) \right\}$$

$$\leqslant \min\left\{ \frac{N \cdot H(A, N)}{H(A, N) + H(C, N)}; G(A, N); \frac{H(A, N) \cdot G(C, N)}{H(C, N)} \right\} + \min\left\{ \frac{N \cdot H(C, N)}{H(A, N) + H(C, N)}; \frac{H(C, N) \cdot G(A, N)}{H(A, N)}; G(C, N) \right\}$$

ableiten. Außerdem gilt wegen (1)

$$G(D, N) \leqslant \frac{\max\{ H(A, 1); H(C, 1) \}}{\max\left\{ \frac{H(A, 1) + H(C, 1)}{N}; \frac{H(A, N)}{G(A, N)}; \frac{H(C, N)}{G(C, N)} \right\}}$$

$$\leqslant \max\left\{ \min\left\{ \frac{N \cdot H(A, 1)}{H(A, 1) + H(C, 1)}; \frac{H(A, 1) \cdot G(C, N)}{H(C, N)} \right\}; \min\left\{ \frac{N \cdot H(C, 1)}{H(A, 1) + H(C, 1)}; \frac{H(C, 1) \cdot G(A, N)}{H(A, N)} \right\} \right\}$$

$$= \max\left\{ \min\left\{ \frac{N}{1 + \frac{H(C, 1)}{H(A, 1)}}; \frac{H(A, 1) \cdot G(C, N)}{H(C, N)} \right\}; \min\left\{ \frac{N}{\frac{H(A, 1)}{H(C, 1)} + 1}; \frac{H(C, 1) \cdot G(A, N)}{H(A, N)} \right\} \right\}$$

$$\leqslant \max\left\{ \min\left\{ \frac{N}{1+\frac{H(C,N)\cdot G(A,N)}{N\cdot H(A,N)}}; \frac{\frac{H(A,N)\cdot N}{G(A,N)}\cdot G(C,N)}{H(C,N)} \right\}; \right.$$

$$\left. \min\left\{ \frac{N}{\frac{H(A,N)\cdot G(C,N)}{N\cdot H(C,N)}+1}; \frac{\frac{H(C,N)\cdot N}{G(C,N)}\cdot G(A,N)}{H(A,N)} \right\} \right\}$$

$$= \max\left\{ \min\left\{ \frac{N^2\cdot H(A,N)}{N\cdot H(A,N)+H(C,N)\cdot G(A,N)}; \frac{N\cdot H(A,N)\cdot G(C,N)}{H(C,N)\cdot G(A,N)} \right\}; \right.$$

$$\left. \min\left\{ \frac{N^2\cdot H(C,N)}{N\cdot H(C,N)+H(A,N)\cdot G(C,N)}; \frac{N\cdot H(C,N)\cdot G(A,N)}{H(A,N)\cdot G(C,N)} \right\} \right\}.$$

Insgesamt erhält man

$$\tilde{G}(D,N) \leqslant \overline{G}(D,N) = \lim_{z\to\infty} \frac{1}{z^2}\cdot \sum_{a=1}^{z}\sum_{c=1}^{z} \min\left\{ \min\left\{ \frac{N\cdot f\left(\frac{a}{z}\right)}{f\left(\frac{a}{z}\right)+f\left(\frac{c}{z}\right)}; G(A,N); \frac{f\left(\frac{a}{z}\right)\cdot G(C,N)}{f\left(\frac{c}{z}\right)} \right\} \right.$$

$$+ \min\left\{ \frac{N\cdot f\left(\frac{c}{z}\right)}{f\left(\frac{a}{z}\right)+f\left(\frac{c}{z}\right)}; \frac{f\left(\frac{c}{z}\right)\cdot G(A,N)}{f\left(\frac{a}{z}\right)}; G(C,N) \right\}; \max\left\{ \min\left\{ \frac{N^2\cdot f\left(\frac{a}{z}\right)}{N\cdot f\left(\frac{a}{z}\right)+f\left(\frac{c}{z}\right)\cdot G(A,N)}; \frac{N\cdot f\left(\frac{a}{z}\right)\cdot G(C,N)}{f\left(\frac{c}{z}\right)\cdot G(A,N)} \right\}; \right.$$

$$\left. \left. \left. \min\left\{ \frac{N^2\cdot f\left(\frac{c}{z}\right)}{N\cdot f\left(\frac{c}{z}\right)+f\left(\frac{a}{z}\right)\cdot G(C,N)}; \frac{N\cdot f\left(\frac{c}{z}\right)\cdot G(A,N)}{f\left(\frac{a}{z}\right)\cdot G(C,N)} \right\} \right\} \right\}.$$

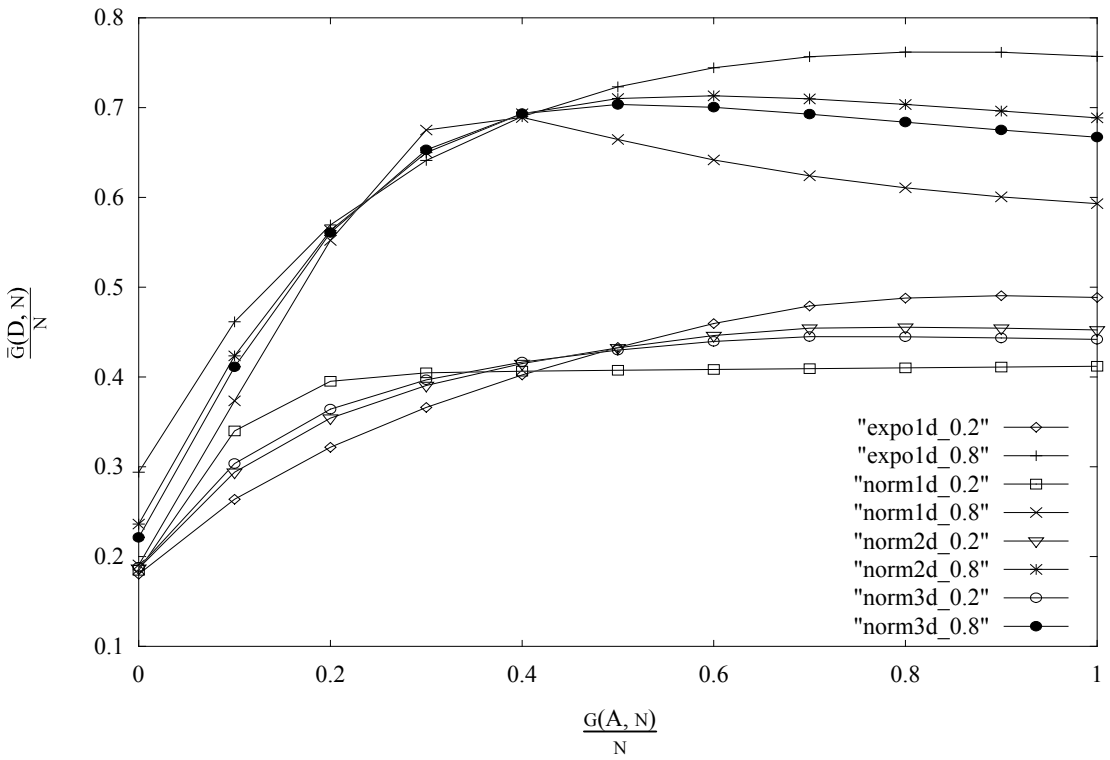


Abbildung 87 - parallele Vereinigung (Rückführung auf parallele Laufzeiten).

In der Abbildung 87 erkennt man, ähnlich wie in der Abbildung 85, einen Knick, an welchem die Dominanz von $G(A, N) + G(C, N)$ zur Berechnung von $\overline{G}(D, N)$ endet. Wie auch bei der Rückführung auf sequentielle Laufzeiten liefern Vereinigungen mit geringer Streuung sehr gute Ergebnisse. Liegen in diesem Fall kleine Gütefaktoren vor, so beeinflußt $\min\{ 2 \cdot G(A, N); 2 \cdot G(C, N) \}$ die obere Schranke. Bei zwei großen Gütefaktoren liefert die Maximumbildung einen Wert in der Größenordnung von $\frac{1}{2} \cdot N$. In der Abbildung 87 äußert sich diese Eigenschaft im absteigenden Verlauf der Kurven für $\frac{G(C, N)}{N} = 0{,}8$ ab einem bestimmten Punkt $\frac{G(A, N)}{N}$.

Die bei der parallelen Vereinigung erstellten Kurven werfen die Frage nach der Existenz von Fixpunkten auf. Bei diesen Punkten muß aus $G(A, N) = G(C, N)$ auch $\overline{G}(D, N) = G(A, N)$ folgen. Für die unterschiedlichen Verteilungen wurden sie mittels /src/guete3/guete3.c berechnet. Sie lauten:

Fixpunkte $\frac{1}{N} \cdot \overline{G}_{fix}(D, N)$		
	Rückführung auf	
Verteilung	sequentielle Laufzeiten	parallele Laufzeiten
/res/guete3/expo1?_fix.txt	0,7061	0,7567
/res/guete3/norm1?_fix.txt	0,5555	0,6554
/res/guete3/norm2?_fix.txt	0,6459	0,7151
/res/guete3/norm3?_fix.txt	0,6279	0,7031

In der Praxis gelangt man zu solchen Fixpunkten, wenn man eine hohe Anzahl an parallelen Vereinigungen hintereinander ausführt. Dabei muß natürlich ein balancierter Binärbaum als Verknüpfungsstruktur gewählt werden (vgl. Abschnitt 9.19). Die in der Fixpunkttabelle angegebenen Werte sind allerdings mit rund $0{,}7 \cdot N$ für eine mittlere Ergebnisgüte viel zu hoch. Man sollte eigentlich von N unabhängige Werte anstreben. Aus diesem Grund wird nun doch noch die gesamte Optimierung und nicht nur ein Einzelschritt betrachtet. Dazu erzeugt man einen zufälliger Konstruktionsgraphen. Ist A ein Blatt des Konstruktionsbaumes so gilt bekanntlich $G(A, N) = 1$. Geht man von diesen Werten aus, so kann mit Hilfe der in diesem Abschnitt hergeleiteten vier Formeln eine obere Schranke für die mittlere Güte für alle inneren Knoten bis zur Wurzel des Konstruktionsbaumes berechnet werden. Diese Art der Betrachtung ist nun leider nicht mehr unabhängig von der Modul- und Gesamtprozessoranzahl.

Damit bei der Erzeugung eines beliebigen Konstruktionsbaumes mit M Blättern (entspricht $M - 1$ inneren Knoten) alle möglichen Bäume die gleiche Wahrscheinlichkeit haben, darf man zueinander symmetrische Bäume nicht doppelt zählen. Zwei Bäume sind symmetrisch, wenn man sie nur durch (eventuell wiederholtes) Vertauschen der linken und rechten Unterbäume eines inneren Knotens ineinander überführen kann (vgl. Abbildung 88). Es sei A ein innerer Knoten des Konstruktionsbaumes. Dann bezeichne l(A) die Anzahl der inneren Knoten im linken Unterbaum und r(A) die im rechten. Verwendet man nur Bäume, bei welchen für alle inneren Knoten A die Beziehung $l(A) \geqslant r(A)$ gilt, so sind keine zwei zueinander symmetrischen Bäume vorhanden.

Damit ein rekursives Erstellen der Bäume möglich ist, muß man sich noch überlegen, wie viele verschiedene Konstruktionsbäume mit $M - 1$ inneren Knoten existieren. Dieser Wert wird von der Funktion $a(M - 1)$ berechnet. Für $M = 1$ ist das einer, also $a(0) = 1$. Hat man im Knoten A nun noch $x \geqslant 1$ innere Knoten unterzubringen, so benötigt man erst einmal einen für A und kann somit noch $x - 1$ Knoten auf die Unterbäume von A verteilen, also

$$l(A) + r(A) = x - 1.$$

Wegen $l(A) \geqslant r(A)$ folgt

$$x - 1 \geqslant l(A) \geqslant \lceil ½ \cdot (x - 1) \rceil = \lfloor ½ \cdot x \rfloor.$$

Da es für die Unterbäume ebenfalls wieder verschiedene Möglichkeiten geben kann und jede Kombination von linken und rechten Nachfolgern berücksichtigt werden muß, gilt

$$a(x) = \sum_{i = \lfloor ½ \cdot x \rfloor}^{x - 1} a(i) \cdot a(x - 1 - i).$$

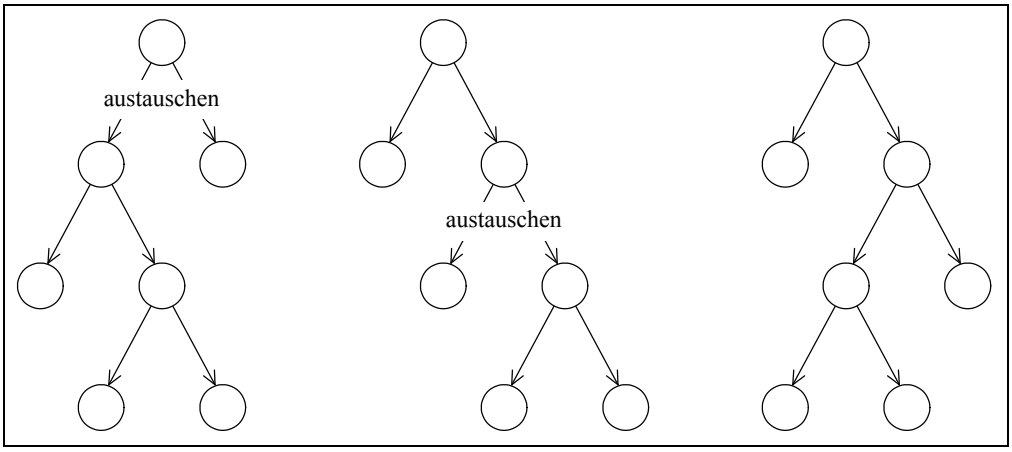


Abbildung 88 - Drei zueinander symmetrische Bäume.

Die Berechnung dieser Funktion kann vor der Baumerzeugung mittels bottom - up erfolgen. Bei der Baumerzeugung können die Werte dann schnell aus einer Tabelle ausgelesen werden. Zu guter Letzt muß man nun noch die Wahrscheinlichkeit bestimmen, daß in A die Verteilung

$$l(A) = i \quad \text{und} \quad r(A) = x - 1 - i$$

auftritt. Sie ist $\frac{a(i) \cdot a(x - 1 - i)}{a(x)}$. Somit reicht die Bestimmung einer gleichverteilten Zufallszahl aus dem Intervall [0; 1), um in jedem inneren Knoten A die Werte l(A) und r(A) bestimmen zu können. Anschließend behandelt man die beiden Kinder genauso. Hat man den Baum erzeugt, so beschriftet man die einzelnen inneren Knoten mit einer Wahrscheinlichkeit von v (`#define VERHAELTNIS v`) mit + (serielle Vereinigung). Anderenfalls erhalten sie ||.
Auch dieses Vorgehen wurde in /src/guete3/guete3.c implementiert (`#define ENDERGEBNIS`). Da es in vertretbarer Rechenzeit nicht möglich ist, alle Bäume zu betrachten, wird eine zufällige Auswahl von 100 Exemplaren (`#define DURCHLAEUFE 100`) erzeugt. Bei einigen Testläufen mit exponential verteilten (`#define EXPONENTIAL`) sequentiellen Modullaufzeiten (`#define SEQUENTIELL`) ergaben sich die in den Abbildungen 89 und 90 dargestellten Kurven. In der Legende von der Abbildung 89 steht die erste Zahl für die Gesamtprozessoranzahl und die zweite für die Anzahl der Module, also „verhae N . M “.
Wählt man bei der Darstellung der Ergebnisse ein konstantes Verhältnis v von paralleler und serieller Verteilung, so kann man die ermittelten Werte in Abhängigkeit der Modulanzahl darstellen. In der Legende der Abbildung 90 steht der erste Wert für die Gesamtprozessoranzahl N und der zweite Wert für das Verhältnis v, also „proz N_v“.
Leider werden auch mit diesem Ansatz nur sehr grobe obere Schranken für die mittlere Güte erreicht. Dies liegt vor allem daran, daß bei den vorgenommenen Abschätzungen kein Bezug auf die konstante Gesamtprozessoranzahl genommen wird. In der Praxis erreicht man jedoch

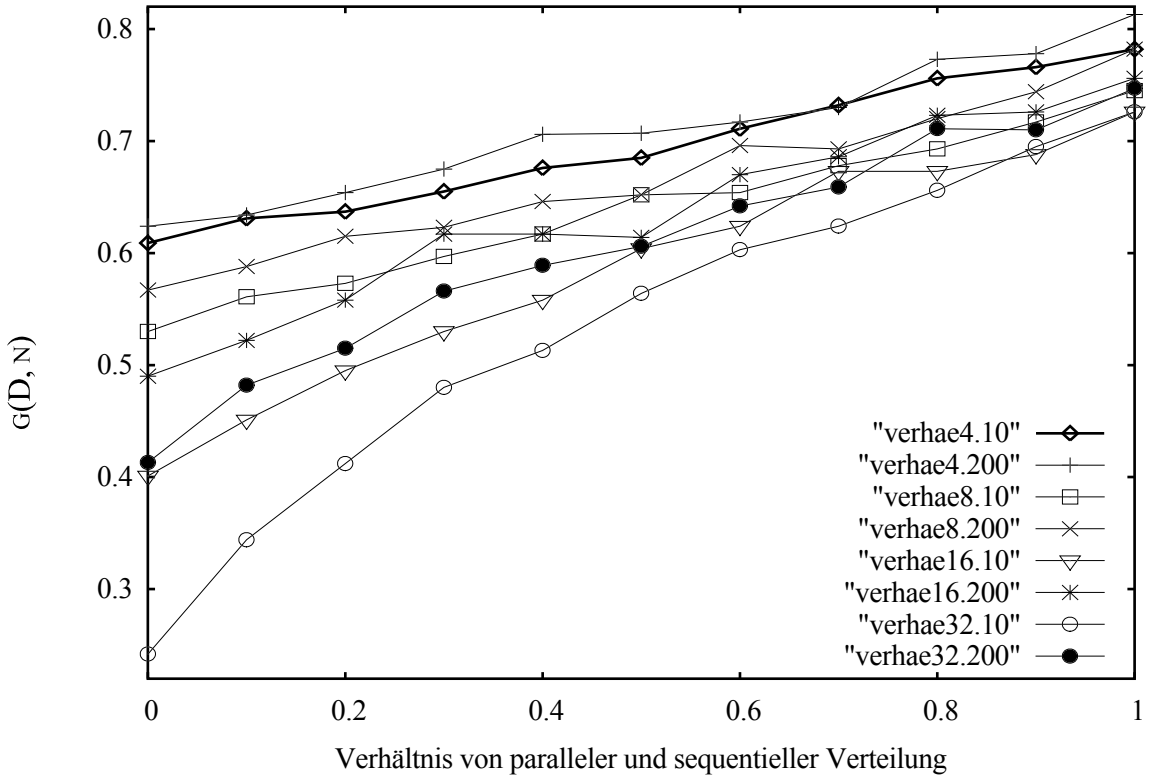


Abbildung 89 - mittlere Güte bei variablen Verhältnissen.

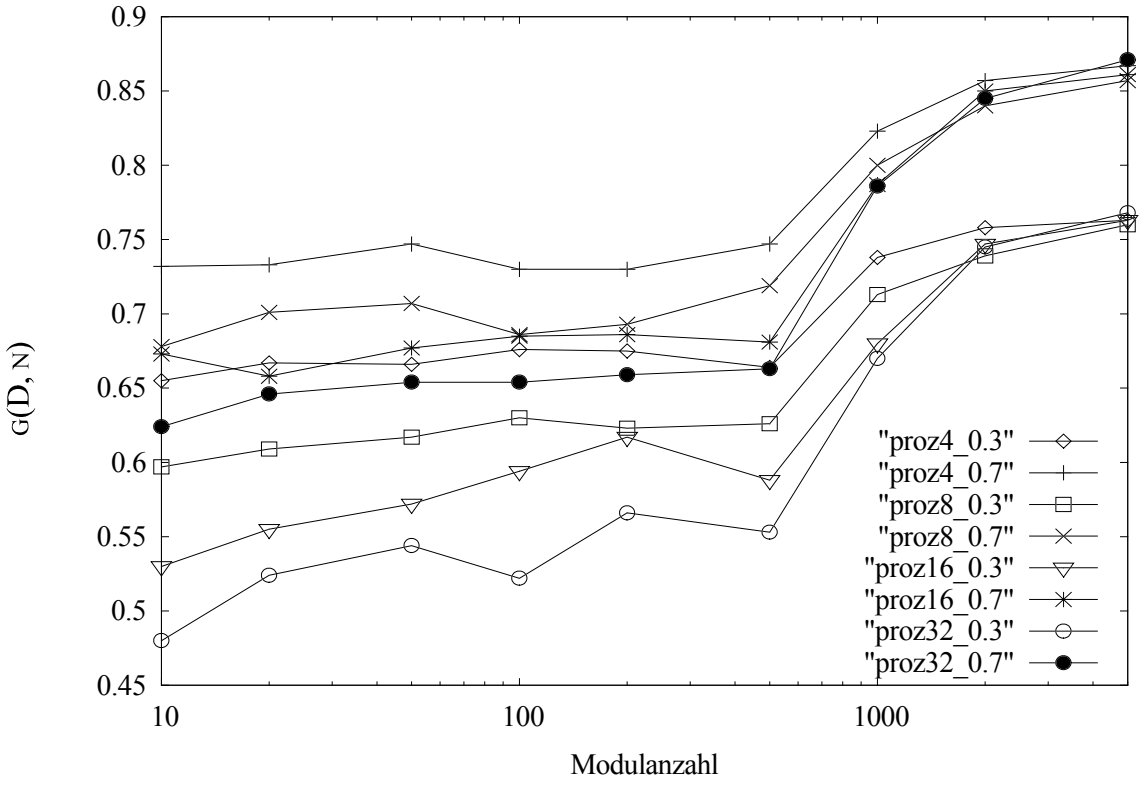


Abbildung 90 - mittlere Güte bei variabler Modulanzahl.

schon nach einer geringen Anzahl von Schritten einen Zustand, bei welchem durch wiederholte parallele Vereinigungen fast alle Module mit nur einem Prozessor ausgeführt werden.
In diesem Fall ist der Parallelisierungsmehraufwand sicher minimal, was ein Zeichen für einen relativ guten Schedule ist. Dominieren im Konstruktionsbaum die seriellen Vereinigungen, so werden auch im optimalen Schedule sehr viele Module mit allen Prozessoren ausgeführt, d.h., die Gesamtlaufzeit des optimalen Schedules ist größer als bei der oben verwendeten Abschätzung angenommen wurde.

9.18 Möglichkeiten zur Verbesserung des Algorithmus „2M1"

Ein erster Einfall basiert auf den im Abschnitt 2.1 gemachten Überlegungen. Dort war es sinnvoll, die Module nach ihrer sequentiellen Laufzeit zu sortieren. Aufgrund der Abhängigkeiten zwischen den Modulen können solche Umsortierungen nur bei parallelen Vereinigungen vorgenommen werden. Das im Abschnitt 9.16 zur parallelen Vereinigung angegebene Beispiel zeigt sehr schön, daß es nicht unbedingt günstig ist, mit dem Modul mit der größten sequentiellen Laufzeit zu beginnen. Aus diesem Grund vereinige man immer die beiden Module mit der kleinsten sequentiellen Laufzeit miteinander. Ein kleines Beispiel zeigt jedoch, daß diese Änderung der Vereinigungsreihenfolge nicht unbedingt zu besseren Schedules führen muß. Dazu setze man $N = 3$. Der Ausgangskonstruktionsbaum laute (A || D) || C. Nach der Sortierung erhält man (vgl. nachfolgende Tabelle): (A || C) || D.

Beispiel, daß Sortierung bzgl. sequentieller Laufzeiten keine Verbesserung bringt

i	H(A, i)	H(C, i)	H(D, i)	H(A || D, i)	H((A || D) || C, i)	H(A || C, i)	H((A || C) || D, i)
1	10	15	18	28	43	25	43
2	5	10	9	14	24	15	24
3	4	8	6	10	15	10	16

In gleicher Weise findet man auch ein Gegenbeispiel für die Sortierung nach den voll parallelen Laufzeiten:

Beispiel, daß Sortierung bzgl. paralleler Laufzeiten keine Verbesserung bringt

i	H(A, i)	H(C, i)	H(D, i)	H(A || D, i)	H((A || D) || C, i)	H(A || C, i)	H((A || C) || D, i)
1	18	45	20	38	83	63	83
2	14	38	17	20	45	45	62
3	10	15	16	18	33	25	41

Eine weitere Möglichkeit zur Erzeugung des fiktiven Moduls bei mehreren parallelen Vereinigungen ist die Verwendung des Algorithmus „P1+MF". Anstatt die zu vereinigenden Module nacheinander zu betrachten, werden sie auf einmal zu einem neuen Modul verknüpft. Doch auch bei dieser Vorgehensweise gibt es Beispiele, bei welchen gegenüber dem Originalalgorithmus schlechtere Schedules erzielt werden. Es sei $N = 2$ und der Konstruktionsbaum laute (A || C) || D:

Beispiel, daß die Verwendung von „P1+MF" keine Verbesserung bringt

i	H(A, i)	H(C, i)	H(D, i)	H(A || C, i)	H((A || C) || D, i)	„P1+MF"
1	2	2	2	4	6	6
2	2	2	1	2	3	4 (Ende in Zeile 8))

Natürlich existieren auch Beispiele, bei welchen dieses Vorgehen besser als die schrittweise Vereinigung ist. Man betrachte dazu die Konstruktion von Modul C im Abschnitt 9.16 bei der Erstellung des Gegenbeispiels für die parallele Vereinigung. Die an dieser Stelle ausgenutzte negative Eigenschaft der parallelen Vereinigung kann durch die gleichzeitige Einsortierung aller Module beseitigt werden. Man beachte jedoch, daß die Verwendung von „P1+MF“ zur Berechnung von hintereinander auszuführenden parallelen Vereinigungen die Laufzeit von „2M1“ erhöht. Im allgemeinen wird aber die im Abschnitt 9.16 verwendete Eingabekonstellation sehr selten auftreten, so daß sich der Mehraufwand in der Praxis nicht rentiert.
Außerdem ist es möglich ein Beispiel zu konstruieren, bei welchem nur ein Gütefaktor in der Nähe von N erzielt wird, obwohl alle parallelen Vereinigungen das optimale Ergebnis liefern. Wie man sieht, findet die in „2M1“ genutzte Methode den optimalen Schedule bei der parallelen Vereinigung von nur zwei einfachen Modulen. Aus diesem Grund kann sie auch im folgenden Beispiel verwendet werden. Dazu sei z eine hinreichend große natürliche Zahl und ein Vielfaches von N, also $\frac{z}{N} \in \mathbb{N}$. Man definiere sich die folgenden drei einfachen Module A, C und D mittels

$$\forall i \in [1; N]: \; H(A, i) = 1, \quad H(C, i) = \frac{1}{z} \quad \text{und} \quad H(D, i) = N - i + 1.$$

Der Konstruktionsbaum laute

$$\underbrace{C + (A \parallel (C + (A \parallel (C + \cdots \; (A \parallel (C + (A \parallel D))) \cdots))))}_{\text{z-mal das Modul C}}$$

Der dazugehörige Modulabhängigkeitsgraph, das gefundene sowie der optimale Schedule sind in der Abbildung 91 dargestellt.

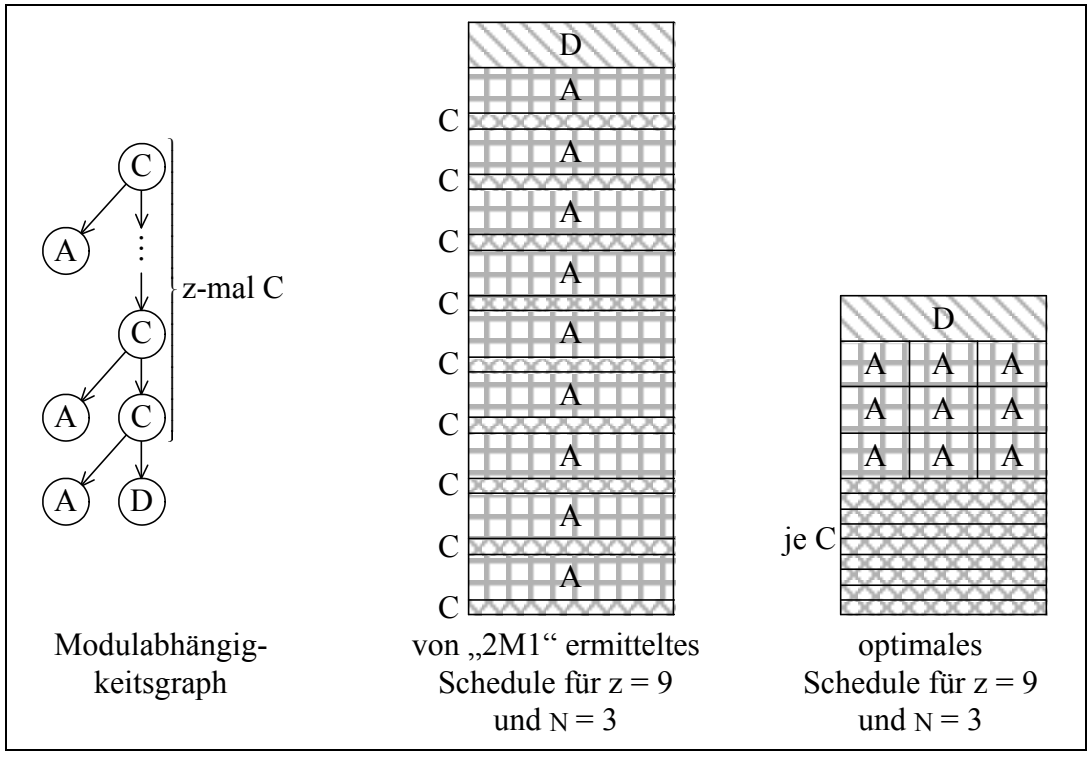


Abbildung 91 - Beispiel mit wechselndem Vereinigungstyp.

Analog zu den im Abschnitt 9.16 durchgeführten Überlegungen kann man auch hier zeigen, daß der von „2M1“ ermittelte Schedule alle Module voll parallel ausführt. Man gelangt somit zu einer Gesamtlaufzeit von

$$H = 1 + \left(\frac{1}{z} + 1\right) \cdot z = z + 2.$$

Im optimalen Schedule können aber immer N Module vom Typ A gleichzeitig abgearbeitet werden. Aus diesem Grund erhält man

$$O = \frac{1}{z} \cdot z + \frac{z}{N} + 1 = \frac{z}{N} + 2.$$

Läßt man nun z gegen unendlich laufen, so gilt:

$$\lim_{z \to \infty} G = \lim_{z \to \infty} \frac{H}{O} = \lim_{z \to \infty} \frac{N \cdot (z+2)}{z + 2 \cdot N} = N.$$

Demzufolge kann man trotz abwechselnder paralleler und serieller Vereinigung ein beliebig schlechtes Beispiel konstruieren. Es ist also nicht ausreichend, nur Verbesserungen bei der parallelen Vereinigung vorzunehmen, sondern es sind auch Umsortierungen in der Modulausführungsreihenfolge notwendig.

9.19 Implementierung und Beispielgenerierung von „2M1“

Der Algorithmus „2M1“ wurde wieder in C implementiert (siehe /src/prg3/opti3.c auf der beiliegenden CD). Der Aufbau der Graphendatei orientiert sich an der Speicherung von Graphen mit Hilfe von Adjazenzlisten. Der erste Wert einer jeden Zeile gibt den Bezugsknoten an. Anschließend folgen, durch Leerzeichen getrennt, alle Knoten, von welchen eine Kante zum Bezugsknoten führt. Dies sind alle Module, die vor dem Bezugsmodul beendet sein müssen. Hat ein Knoten keine Vorgänger, so muß er trotzdem als Bezugsknoten mit leerer Liste aufgeführt sein.
Wie man im Lemma 18 gesehen hat, ist der Konstruktionsbaum bei mehreren aufeinander folgenden parallelen Vereinigungen nicht eindeutig bestimmt. Aus diesem Grund kann man mittels `#define BALANCIERT` einen ausbalancierten Teilbaum erstellen lassen. Wird dieser Schalter nicht gesetzt, so entsteht eine lineare Kette, d.h., die parallelen Vereinigungen werden alle nacheinander ausgeführt. In der Abbildung 92 sind die beiden Möglichkeiten nochmals dargestellt.

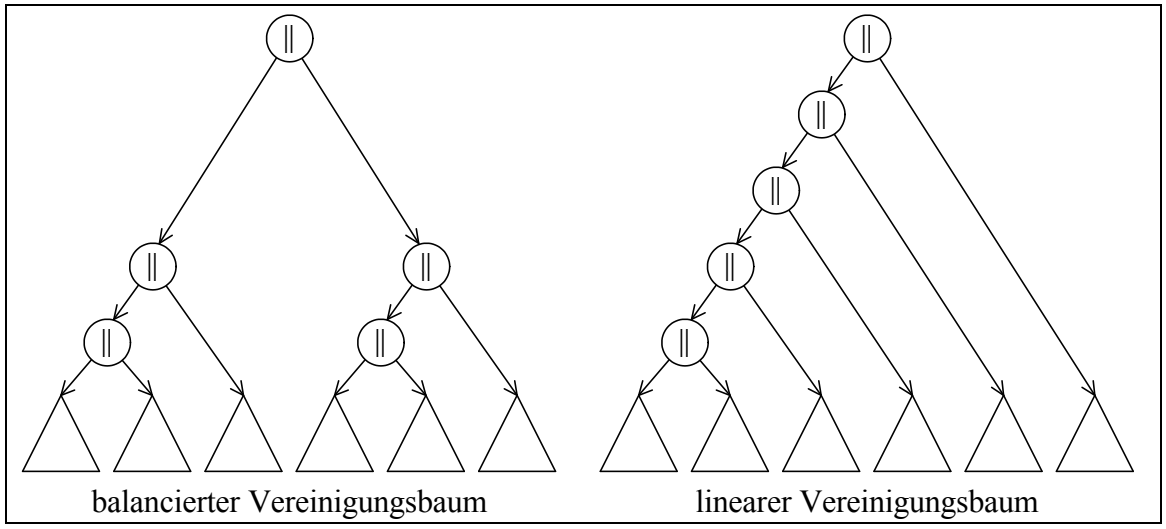


Abbildung 92 - Vereinigungsbäume.

Da die Reihenfolge der Vorgängerknoten in der Adjazenzliste einen Einfluß auf den erstellten Konstruktionsgraphen hat, kann für eigentlich gleiche Eingabewerte ein unterschiedlicher Schedule berechnet werden.
Als Testbeispiel erweitere man das Beispiel 1a aus dem Abschnitt 9.10 noch um einen Verteilungsparameter (vgl. Abschnitt 9.17) für die Erzeugung des Konstruktionsbaumes. Diesen Konstruktionsbaum rechnet der Beispielgenerierungsalgorithmus dann in einen serien - parallelen Graphen um und übergibt diesen dem Algorithmus „2M1“. Die Datei /make/prg3/bsp3.txt hat somit den in der Abbildung 93 dargestellten Inhalt. Der Parameter v nimmt nacheinander die Werte 0,2; 0,5 und 0,8 an.

```
PROZESSORANZAHL: {N}

MODULANZAHL: konstant {M}

WAHRSCHEINLICHKEIT EINER SEQUENTIELLEN VEREINIGUNG: normal 0 1 {v} 0.15

MODULWAHRSCH  LAUFZ_MIT_EIN_PROZ   PARALLELISIERBARKEIT
gleich 1 3    expo 1 100 1         BIS {0,4·N} PROZ: normal 0 1 0.5 0.2;
                                   BIS {N} PROZ: konstant 0
konstant 3.5  norm 0.5 4.7 2.2 1   BIS {0,7·N} PROZ: expo 0 1 0.5;
                                   BIS {N} PROZ: konstant 0.5
```

Abbildung 93 - Datei /make/prg3/bsp3.txt - allgemeine Form.

9.20 Beispielgenerierung für „2M1N“

Das zum Testen von Algorithmus „2M1N“ verwendete Beispiel 4 (/make/prg4/bsp4.txt, siehe Abbildung 94) basiert auf dem Beispiel 2 (siehe Abschnitt 9.12).

```
PROZESSORANZAHL: {N}

MODULANZAHL: konstant {M}

WAHRSCHEINLICHKEIT EINER SEQUENTELLEN VEREINIGUNG: normal 0 1 {v} 0.15

MODULW   MIN_PROZ      LAUFZ_MIN_PROZ     PARALLELISIERBARKEIT
gl 1 3   gl 2 {0,8·N}  ex 1 100 1         BIS {0,4·N} P: no 0 {2/N} {1/N} {4/(10·N)};
                                          BIS {N} P: gl 0 {1/N}
ko 1.5   gl 2 {0,8·N}  no 0.5 4.7 2.2 1   BIS {0,7·N} P: ex 0 1 {N};
                                          BIS {N} P: gl 0 {2/N}
ko 0.5   ko 1          ex 1 100 0.7       BIS {N} P: no 0 1 0.4 0.3
```

Abbildung 94 - Datei /make/prg4/bsp4.txt - allgemeine Form.

9.21 Der Algorithmus „SchHT“

9.21.1 Obere Laufzeitschranke

Das in diesem Abschnitt angegebene Beispiel zeigt, daß die für den Algorithmus „SchHT“ angegebene obere Laufzeitschranke wirklich erreicht werden kann.

Es sei $i \in 2\mathbb{N}$, also eine positive gerade Zahl. Der Modulabhängigkeitsgraph bestehe in diesem Beispiel aus $\frac{3}{2} \cdot i + 1$ Knoten. Außerdem sollen die folgenden Kanten existieren:

Kanten des Modulabhängigkeitsgraphen	
$(k, k + 1)$	$\forall k \in [1; i]$
$(1, i + 2)$	
$(k, k + 1)$	$\forall k \in \left[i + 2; \frac{3}{2} \cdot i\right]$
$\left(\frac{3}{2} \cdot i + 1; i + 1\right)$	
(k, j)	$\forall k \in [2; ½ \cdot i]$ und $\forall j \in \left[k + i + 1; \frac{3}{2} \cdot i + 1\right]$

Die sequentiellen Laufzeiten $T(x, 1)$ der Module x seien so gewählt, daß die Module, welche nicht auf dem längsten Pfad liegen, in absteigender Reihenfolge bzgl. ihrer Nummer eingefügt werden. Der Algorithmus beginnt also mit Modul $\frac{3}{2} \cdot i + 1$ und endet bei Nummer $i + 2$. Dabei soll er das Modul x dem Teilgraphen G_k mit

$$k = \begin{cases} 2 & : \text{falls } x = i + 2 \\ 2 \cdot x - 2 \cdot i - 3 & : \text{falls } x \geqslant i + 3 \end{cases}$$

zuordnen.
Für $i = 6$ entsteht somit der in der Abbildung 95 dargestellte Graph.

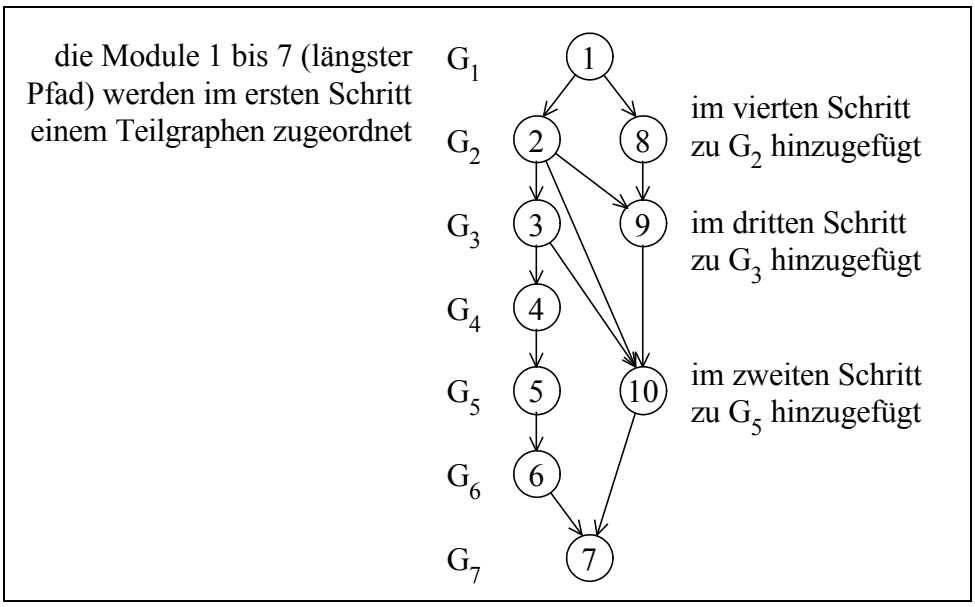


Abbildung 95 - laufzeitkritisches Beispiel.

Man erkennt, daß nach der Zuordnung des Moduls x ($x \geqslant i + 2$) zum oben angegebenen Teilgraphen der Wert in ho(y) aller Module $y \in [i + 2; x - 1]$ um Eins vergrößert wird. Demzufolge läuft der Algorithmus über alle Kanten, die die Module $i + 2$ bis x als Endknoten haben. Dies sind exakt

$$\sum_{k=1}^{x-i-1} k = \frac{x - i}{2} \cdot (x - i - 1)$$

Kanten. Summiert man nun die vom Algorithmus betrachtete Kantenanzahl bei allen Knoten x ($x \geqslant i + 2$), so erhält man

$$\sum_{x = i + 2}^{\frac{3}{2} \cdot i + 1} \frac{x - i}{2} \cdot (x - i - 1) = \sum_{x = 1}^{½ \cdot i} \frac{x + 1}{2} \cdot x$$

$$= \left(\frac{1}{2} \cdot \sum_{x = 1}^{½ \cdot i} x^2 \right) + \left(\frac{1}{2} \cdot \sum_{x = 1}^{½ \cdot i} x \right)$$

$$= \frac{1}{2} \cdot \frac{½ \cdot i \cdot (½ \cdot i + 1) \cdot (i + 1)}{6} + \frac{1}{2} \cdot \frac{½ \cdot i + 1}{2} \cdot ½ \cdot i$$

$$= \frac{1}{48} \cdot i^3 + \frac{1}{8} \cdot i^2 + \frac{1}{6} \cdot i.$$

Somit benötigt der Algorithmus zur Berechnung der Teilgraphen $\Theta(i^3)$ Schritte.
Der Modulabhängigkeitsgraphen hat

$$|E| = i + 1 + \frac{3}{2} \cdot i - (i + 2) + 1 + 1 + \left(\sum_{k = 2}^{½ \cdot i} \frac{3}{2} \cdot i + 1 - (k + i + 1) + 1 \right)$$

$$= \frac{3}{2} \cdot i + 1 + \left(\sum_{k = 1}^{½ \cdot i - 1} ½ \cdot i - k \right)$$

$$= \frac{3}{2} \cdot i + 1 + \sum_{k = 1}^{½ \cdot i - 1} k$$

$$= \frac{3}{2} \cdot i + 1 + \frac{½ \cdot i - 1}{2} \cdot ½ \cdot i$$

$$= \frac{1}{8} \cdot i^2 + \frac{5}{4} \cdot i + 1$$

Kanten und eine Tiefe von

$$\text{maxtief} = i + 1.$$

Bei diesem Beispiel liefert die Laufzeitabschätzung zur Berechnung der Teilgraphen eine Laufzeit von $O(\text{maxtief} \cdot (|E| + M) + M \cdot \log(M)) = O(i^3)$. Da die Modulanzahl in O(i) liegt, zeigt dieses Beispiel auch, daß in der Laufzeitabschätzung das |E| nicht durch M ersetzt werden kann.

9.21.2 Vergleich von „SchH“ und „SchHT“

Das nachfolgende Beispiel zeigt, daß „SchHT“ sogar einen schlechteren Schedule als „SchT“ liefern kann.
Es existieren fünf Module (M = 5) mit den in der Tabelle angegebenen Laufzeiten. Der Modulabhängigkeitsgraph ist aus der Abbildung 96 entnehmbar.

Modullaufzeiten, bei denen „SchHT“ schlechter ist als „SchT“		
Modulnummer	Laufzeit mit weniger als N Prozessoren	Laufzeit mit N Prozessoren
1; 2	1	1
3; 5	N	N
4	N + 1	$1 + \frac{1}{N}$

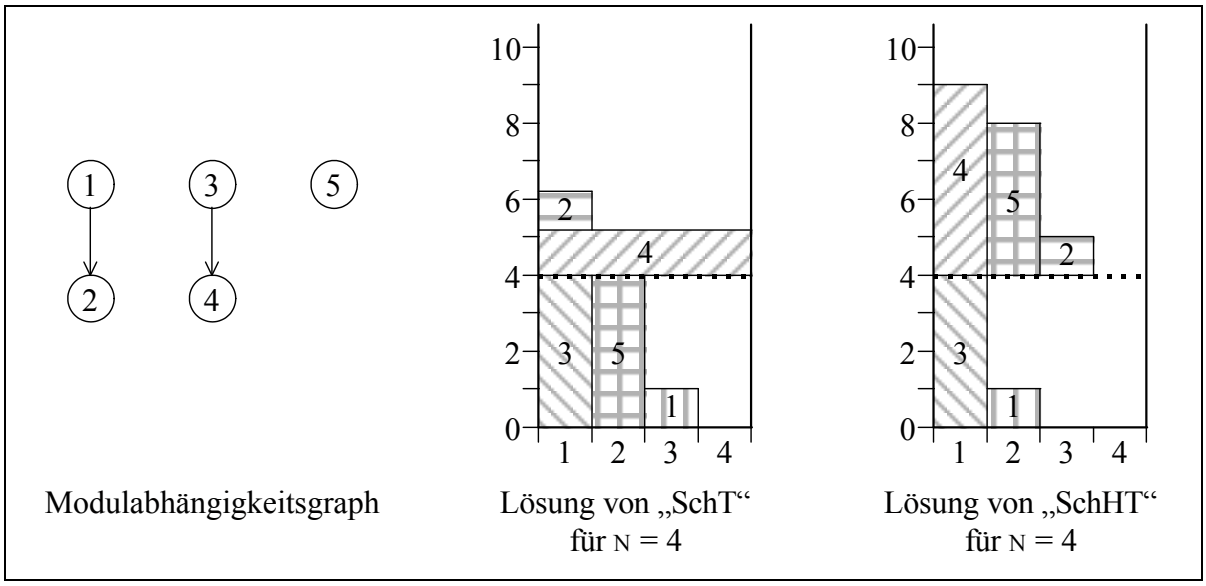


Abbildung 96 - Beispiel, bei welchem „SchT“ besser als „SchHT“ ist.

Die Modullaufzeiten sind bei diesem Beispiel so gewählt, daß bei „SchHT“ das Modul 5 in den Teilgraphen G_2 einsortiert wird. Der Algorithmus „SchT“ ordnet es hingegen in G_1 ein. Die Positionierung der Module in den Schedules von „SchT“ und „SchHT“ ist in der Abbildung 96 dargestellt. Die gepunktete Linie kennzeichnet die Grenze der beiden Teilschedules.

9.21.3 Beispiel für „SchT“ und „SchHT“

In diesem Abschnitt wird ein Beispiel angegeben, bei dem die Algorithmen „SchT“ und „SchHT“ nur einen Gütefaktor von $N - \varepsilon$ für ein beliebiges ε mit $N - 1 \geqslant \varepsilon > 0$ erzielen.

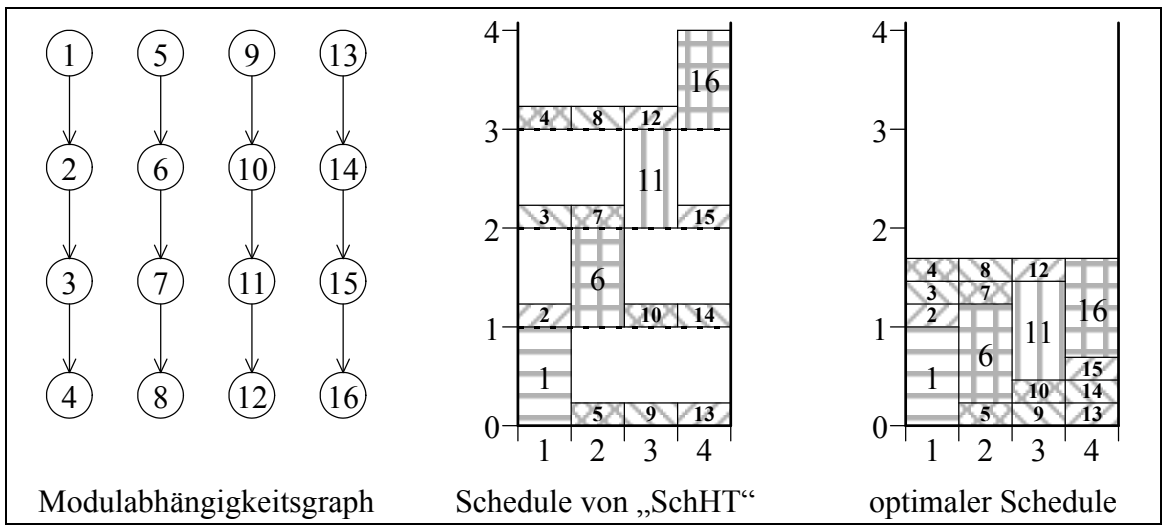


Abbildung 97 - Beispiel, bei welchem „SchT“ und „SchHT“ einen schlechten Schedule erstellen.

Dazu verwende man $M = N^2$ Module. Im folgenden steht $\delta > 0$ als Abkürzung für

$$\delta = \frac{\varepsilon}{(N - \varepsilon) \cdot (N - 1)}.$$

Die Laufzeiten aller Module x seien unabhängig von der Prozessoranzahl $B(x)$. Es gelte

$$\forall i \in [1; N]: T(x, i) = \begin{cases} 1 : x \bmod (N + 1) = 1 \\ \delta : \text{sonst} \end{cases}$$

Im Modulabhängigkeitsgraphen existiere genau dann die Kante (x, x + 1), wenn x kein Vielfaches von N ist, also

$$x \bmod N \neq 0.$$

In der Abbildung 97 sind für den Fall N = 4 der Modulabhängigkeitsgraph, der von „SchHT“ ermittelte und der optimale Schedule dargestellt.
Man erkennt, daß bei „SchHT“ in jedem der N Teilgraphen ein Modul die Laufzeit Eins hat, d.h., die erzielte Gesamtlaufzeit H beträgt N. Der optimale Schedule hingegen hat nur eine Gesamtlaufzeit von O = 1 + δ · (N - 1). Als Gütefaktor resultiert demzufolge

$$G = \frac{H}{O} = \frac{N}{1 + \delta \cdot (N - 1)} = \frac{N}{1 + \frac{\varepsilon}{N - \varepsilon}} = N - \varepsilon.$$

9.22 Der Algorithmus „kGzuB“

9.22.1 Der Pseudocode

In den Abbildungen 98 bis 105 ist der Pseudocode des Algorithmus „kGzuB“ dargestellt.

```
1)  lösche alle transitiven Kanten in G
2)  m = max{x : x ∈ V}; W = V; F = ∅
3)  ∀w ∈ W: h(w) = #
4)  solange E ≠ ∅
5)      m = m + 1
6)      wenn (x, y) ∈ E mit outdeg(x) = 1 und indeg(y) = 1 existiert
7)          starte Unterroutine trafo1(x, y) in Zeile 33)
8)          weiter in Zeile 4)
9)      wenn y ∈ V mit indeg(y) = 1 und outdeg(y) = 0 existiert
10)         starte Unterroutine trafo2(y) in Zeile 40)
11)         weiter in Zeile 4)
12)     wenn y ∈ V mit indeg(y) = 0 und outdeg(y) = 1 existiert
13)         starte Unterroutine trafo3(y) in Zeile 49)
14)         weiter in Zeile 4)
15)     wenn y ∈ V mit indeg(y) = 1 und outdeg(y) = 1 existiert
16)         starte Unterroutine trafo4(y) in Zeile 58)
17)         weiter in Zeile 4)
18)     wenn y ∈ V mit outdeg(y) = 1 existiert
19)         starte Unterroutine trafo5(y) in Zeile 78)
20)         weiter in Zeile 4)
21)     wenn y ∈ V mit indeg(y) = 1 existiert
22)         starte Unterroutine trafo6(y) in Zeile 89)
23)         weiter in Zeile 4)
24)     wähle z ∈ V mit outdeg(z) ⩾ 2
25)     starte Unterroutine trafo7(z) in Zeile 100)
```

26) ***solange*** *$|V| \geqslant 2$*
27) *$m = m + 1$*
28) *wähle $x, y \in V$ mit $x \neq y$*
29) *$W = W \cup \{m\}$; $F = F \cup \{(m, x, y)\}$; $h(m) = \|$*
30) *$V = V \cup \{m\}$*
31) *$V = V \setminus \{x; y\}$*
32) ***Ende***
Der zum Start von Algorithmus „2M1“ benötigte Konstruktionsbaum ist B und die Knotenbeschriftungsfunktion h.

Abbildung 98 - Algorithmus „kGzuB“.

33) ***Unterroutine trafo1(x, y)***
34) *$W = W \cup \{m\}$; $F = F \cup \{(m, x, y)\}$; $h(m) = +$ (beachte: x ist linkes Kind von m)*
35) *$V = V \cup \{m\}$*
36) *$E = E \cup \{(v, m) : v \in V$ mit $(v, x) \in E\} \cup \{(m, v) : v \in V$ mit $(y, v) \in E\}$*
37) *$E = E \setminus (\{(v, x) : v \in V\} \cup \{(y, v) : v \in V\})$*
38) *$V = V \setminus \{x; y\}$*
39) ***Rücksprung*** *aus der Unterroutine*

Abbildung 99 - Algorithmus „kGzuB“ - Unterroutine „trafo1“.

40) ***Unterroutine trafo2(y)***
41) *wähle $z \in \{v : v \in V$ mit $(v, y) \in E\}$*
42) *wähle $x \in \{v : v \in V \setminus \{y\}$ mit $(z, v) \in E\}$*
43) *$W = W \cup \{m\}$; $F = F \cup \{(m, x, y)\}$; $h(m) = \|$*
44) *$V = V \cup \{m\}$*
45) *$E = E \cup \{(v, m) : v \in V$ mit $(v, x) \in E\} \cup \{(m, v) : v \in V$ mit $(x, v) \in E\}$*
46) *$E = E \setminus (\{(v, x) : v \in V\} \cup \{(x, v) : v \in V\} \cup \{(z, y)\})$*
47) *$V = V \setminus \{x; y\}$*
48) ***Rücksprung*** *aus der Unterroutine*

Abbildung 100 - Algorithmus „kGzuB“ - Unterroutine „trafo2“.

49) ***Unterroutine trafo3(y)***
50) *wähle $z \in \{v : v \in V$ mit $(y, v) \in E\}$*
51) *wähle $x \in \{v : v \in V \setminus \{y\}$ mit $(v, z) \in E\}$*
52) *$W = W \cup \{m\}$; $F = F \cup \{(m, x, y)\}$; $h(m) = \|$*
53) *$V = V \cup \{m\}$*
54) *$E = E \cup \{(v, m) : v \in V$ mit $(v, x) \in E\} \cup \{(m, v) : v \in V$ mit $(x, v) \in E\}$*
55) *$E = E \setminus (\{(v, x) : v \in V\} \cup \{(x, v) : v \in V\} \cup \{(y, z)\})$*
56) *$V = V \setminus \{x; y\}$*
57) ***Rücksprung*** *aus der Unterroutine*

Abbildung 101 - Algorithmus „kGzuB“ - Unterroutine „trafo3“.

58) ***Unterroutine trafo4(y)***
59) *wähle $w \in \{v : v \in V$ mit $(v, y) \in E\}$*
60) *wähle $z \in \{v : v \in V$ mit $(y, v) \in E\}$*
61) *$E = E \setminus \{(w, y); (y, z)\}$; $V = V \setminus \{y\}$*
62) ***wenn** in $G = (V, E)$ ein Pfad $(w, x_1, \ldots x_k, z)$ existiert*
63) *wähle $i \in [1; k]$ und setze $x = x_i$*
64) *$W = W \cup \{m\}$; $F = F \cup \{(m, x, y)\}$; $h(m) = \|$*
65) *$V = V \cup \{m\}$*
66) *$E = E \cup \{(v, m) : v \in V$ mit $(v, x) \in E\} \cup \{(m, v) : v \in V$ mit $(x, v) \in E\}$*
67) *$E = E \setminus (\{(v, x) : v \in V\} \cup \{(x, v) : v \in V\})$*
68) *$V = V \setminus \{x\}$*
69) ***Rücksprung** aus der Unterroutine*
70) *wähle $x \in \{v : v \in V$ mit $(w, v) \in E\}$*
71) *$W = W \cup \{m\}$; $F = F \cup \{(m, x, y)\}$; $h(m) = \|$*
72) *$V = V \cup \{m\}$*
73) *$E = E \cup \{(v, m) : v \in V$ mit $(v, x) \in E\} \cup \{(m, v) : v \in V$ mit $(x, v) \in E\} \cup \{(m, z)\}$*
74) *$E = E \setminus (\{(v, x) : v \in V\} \cup \{(x, v) : v \in V\})$*
75) *$V = V \setminus \{x\}$*
76) *lösche alle transitiven Kanten in G*
77) ***Rücksprung** aus der Unterroutine*

Abbildung 102 - Algorithmus „kGzuB“ - Unterroutine „trafo4“.

78) ***Unterroutine trafo5(y)***
79) *wähle $z \in \{v : v \in V$ mit $(y, v) \in E\}$*
80) *wähle $x \in \{v : v \in V \setminus \{y\}$ mit $(v, z) \in E\}$*
81) *$W = W \cup \{m\}$; $F = F \cup \{(m, x, y)\}$; $h(m) = \|$*
82) *$V = V \cup \{m\}$*
83) *$E = E \cup \{(v, m) : v \in V$ mit $(v, x) \in E\} \cup \{(m, v) : v \in V$ mit $(x, v) \in E\}$*
84) *$E = E \cup \{(v, m) : v \in V$ mit $(v, y) \in E\}$*
85) *$E = E \setminus (\{(v, x) : v \in V\} \cup \{(x, v) : v \in V\} \cup \{(v, y) : v \in V\} \cup \{(y, z)\})$*
86) *$V = V \setminus \{x; y\}$*
87) *lösche alle transitiven Kanten in G*
88) ***Rücksprung** aus der Unterroutine*

Abbildung 103 - Algorithmus „kGzuB“ - Unterroutine „trafo5“.

89) ***Unterroutine trafo6(y)***
90) *wähle $z \in \{v : v \in V$ mit $(v, y) \in E\}$*
91) *wähle $x \in \{v : v \in V \setminus \{y\}$ mit $(z, v) \in E\}$*
92) *$W = W \cup \{m\}$; $F = F \cup \{(m, x, y)\}$; $h(m) = \|$*
93) *$V = V \cup \{m\}$*
94) *$E = E \cup \{(v, m) : v \in V$ mit $(v, x) \in E\} \cup \{(m, v) : v \in V$ mit $(x, v) \in E\}$*
95) *$E = E \cup \{(m, v) : v \in V$ mit $(y, v) \in E\}$*

96) E = E \ ({(v, x) : v ∈ V} ∪ {(x, v) : v ∈ V} ∪ {(y, v) : v ∈ V} ∪ {(z, y)})
97) V = V \ {x; y}
98) lösche alle transitiven Kanten in G
*99) **Rücksprung** aus der Unterroutine*

Abbildung 104 - Algorithmus „kGzuB“ - Unterroutine „trafo6“.

*100) **Unterroutine trafo7(z)***
101) wähle x, y ∈ {v : v ∈ V mit (z, v) ∈ E} mit x ≠ y
102) W = W ∪ {m}; F = F ∪ {(m, x, y)}; h(m) = ||
103) V = V ∪ {m}
104) E = E ∪ {(v, m) : v ∈ V mit (v, x) ∈ E} ∪ {(m, v) : v ∈ V mit (x, v) ∈ E}
105) E = E ∪ {(v, m) : v ∈ V mit (v, y) ∈ E} ∪ {(m, v) : v ∈ V mit (y, v) ∈ E}
106) E = E \ ({(v, x) : v ∈ V} ∪ {(x, v) : v ∈ V})
107) E = E \ ({(v, y) : v ∈ V} ∪ {(y, v) : v ∈ V})
108) V = V \ {x; y}
109) lösche alle transitiven Kanten in G
*110) **Rücksprung** aus der Unterroutine*

Abbildung 105 - Algorithmus „kGzuB“ - Unterroutine „trafo7“.

9.22.2 Beispiel eines „kGzuB“ - Laufes

In diesem Abschnitt wird ein „kGzuB“ - Lauf an einem Beispiel vorgeführt.
Mit $k_i(x)$ wird der Konstruktionsterm des Knotens x nach der i-ten Anwendung einer Transformationsregel notiert. In der Abbildung 107 sind die einzelnen Transformationsschritte unter Angabe der verwendeten Transformationsregel dargestellt. Die Abbildung 106 enthält einen Vergleich zwischen dem ursprünglichen Modulabhängigkeitsgraphen und dem erstellten serien - parallelen Graphen.

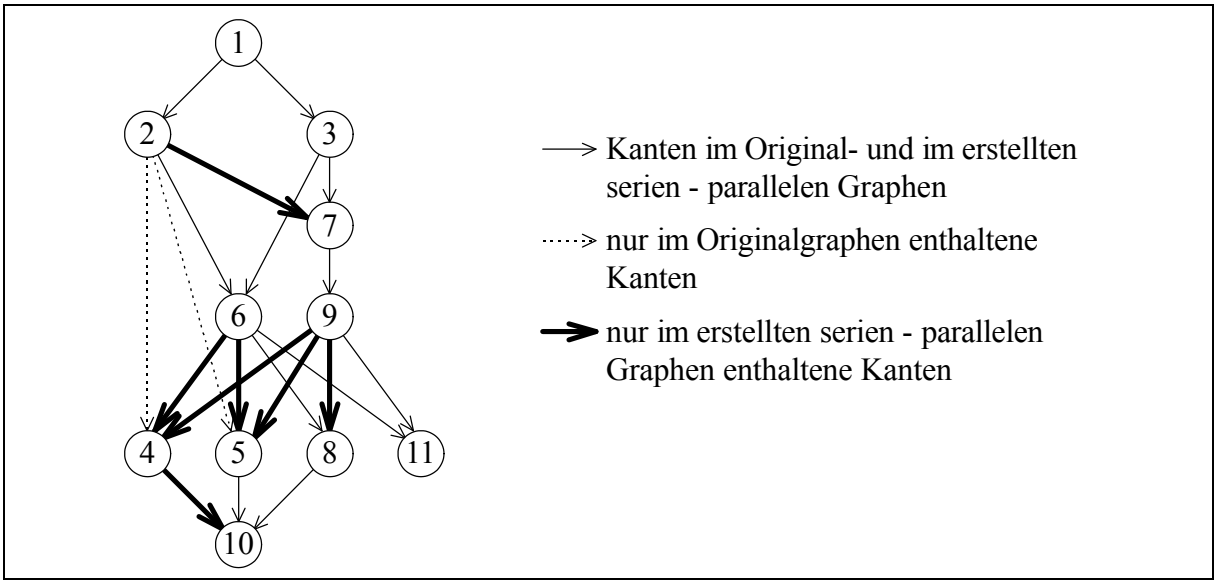


Abbildung 106 - Ergebnis eines „kGzuB“ - Laufes im Vergleich zum Ausgangsgraphen.

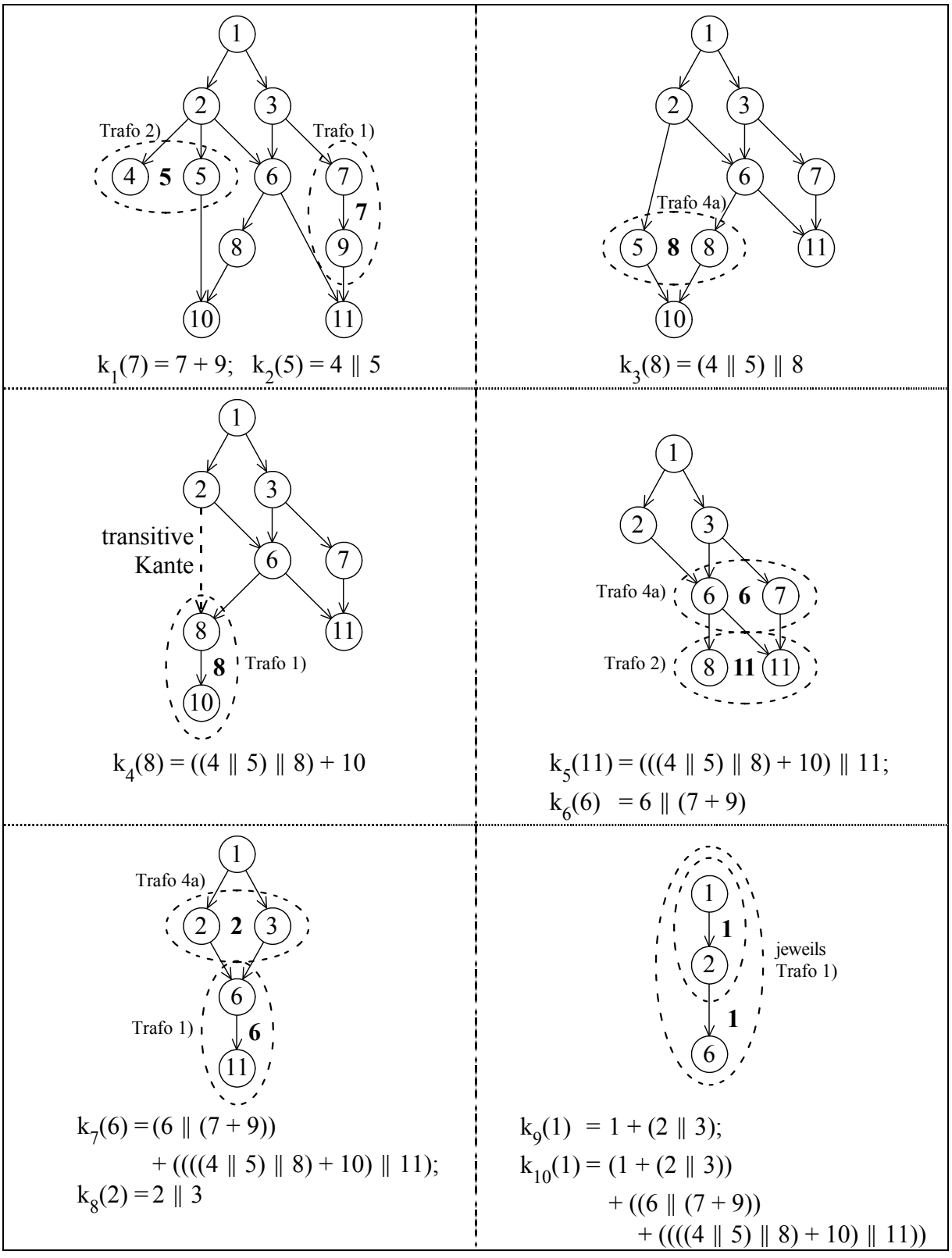


Abbildung 107 - Beispiel eines „kGzuB“ - Laufes.

9.23 Beispiel für „KP+“ und „KPMF“

In diesem Abschnitt wird für $N \geq 3$ ein Beispiel angegeben, bei dem die Algorithmen „KP+“ und „KPMF“ nur einen Gütefaktor von $G \approx \frac{N}{3}$ erzielen. Für $N = 5$ ist dieses Beispiel in der Abbildung 108 dargestellt.

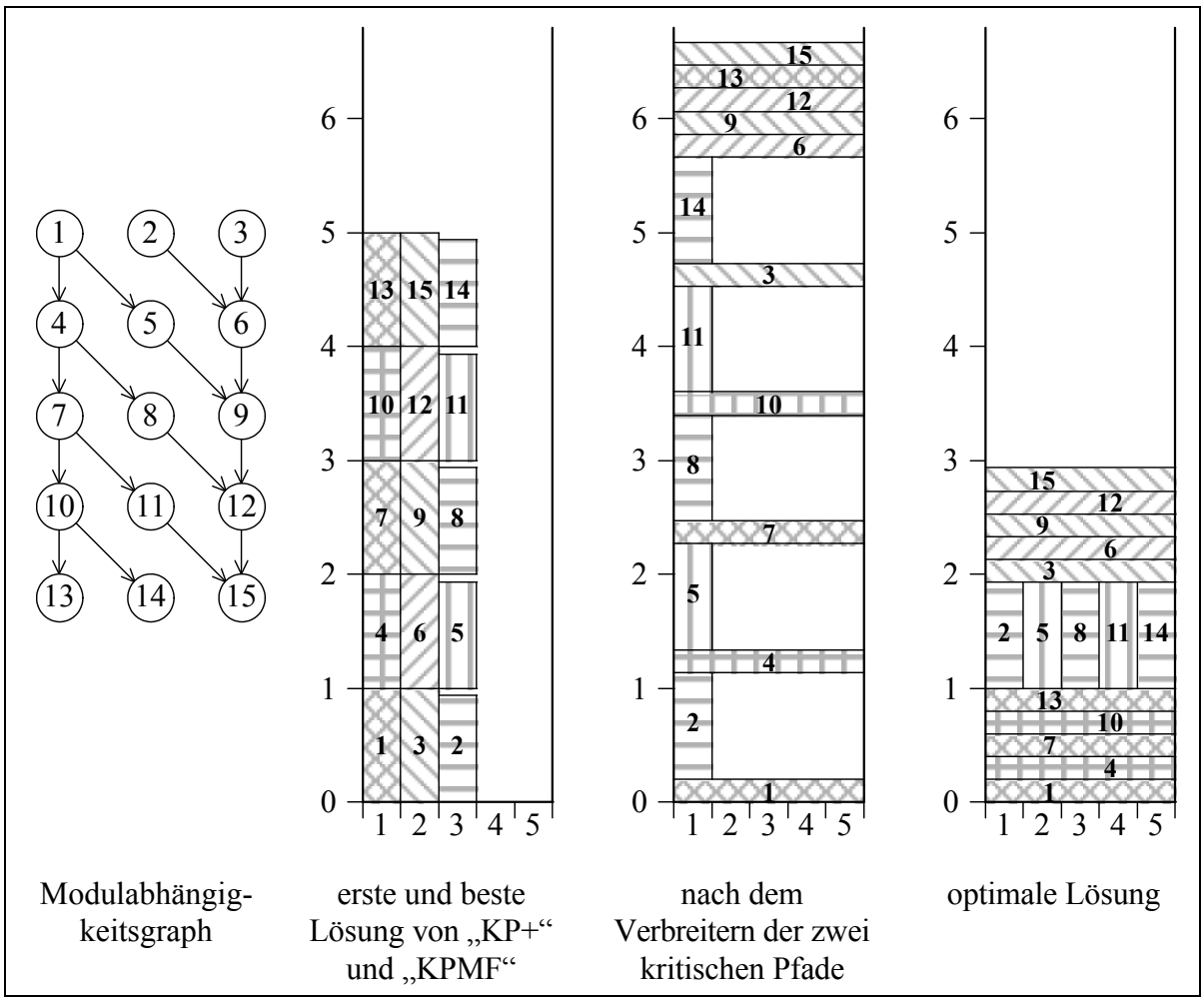


Abbildung 108 - Beispiel, bei dem „KP+“ und „KPMF“ einen schlechten Schedule berechnen.

Es sei ein positives $\varepsilon < \frac{1}{N}$ vorgegeben. Die Laufzeiten und der Modulabhängigkeitsgraph sind in der nachfolgenden Tabelle notiert. Haben mehrere Module die gleiche Priorität, so werde von „KP+“ bzw. „KPMF“ zuerst das Modul mit der kleinsten Modulnummer positioniert.

Beispiel, bei dem „KP+“ und „KPMF“ einen schlechten Schedule berechnen			
Modulnummer $x \in [1; 3 \cdot N]$ mit	Kante von x nach (sofern $x \leq 3 \cdot N - 3$)	Laufzeit von x mit weniger als N Proz.	Laufzeit von x mit N Prozessoren
x mod 3 = 1	x + 3 und x + 4	1	$\frac{1}{N}$
x mod 3 = 2	x + 4	1 - ε	1 - ε
x mod 3 = 0	x + 3	1	$\frac{1}{N}$

Der erste von „KP+“ bzw. „KPMF“ erstellte Schedule hat als Gesamtlaufzeit den Wert N. Da bei gleicher Priorität das Modul mit der kleinsten Nummer zuerst positioniert wird, werden nie zwei Module $x \neq y$ mit $x \bmod 3 = 2$ und $y \bmod 3 = 2$ zum gleichen Zeitpunkt gestartet. Demzufolge entsteht nach der Parallelisierung aller Module x mit $x \bmod 3 \neq 2$ ein Zwischenschedule mit der Laufzeit

$$2 \cdot N \cdot \frac{1}{N} + N \cdot (1 - \varepsilon) = 2 + N - N \cdot \varepsilon > 2 + N - 1 = N + 1.$$

Unter der Voraussetzung, daß immer der gleiche kritische Pfad gefunden wird, erhöhen am Ende beide Algorithmen nur noch die Prozessoranzahl eines Moduls x mit $x \bmod 3 = 2$ und terminieren. Die Algorithmen „KP+“ und „KPMF“ erstellen bei diesem Beispiel also einen Schedule mit der Gesamtlaufzeit $H = N$.
Der optimale Schedule hat hingegen nur eine Gesamtlaufzeit von $O = 3 - \varepsilon$, was einem Gütefaktor von

$$G = \frac{H}{O} = \frac{N}{3 - \varepsilon} > \frac{N}{3}$$

entspricht.

9.24 Beispielgenerierung bei beliebigem Abhängigkeitsgraphen

Zur zufälligen Erstellung eines Modulabhängigkeitsgraphen werden die folgenden Parameter verwendet: Knotenanzahl M, Kantenanzahl k und Knotenabstand q. Da jeder zyklusfreie Graph topologisch sortiert werden kann, kann man davon ausgehen, daß nur Kanten vom Knoten i zum Knoten j mit $j > i$ existieren. Die Größe q gibt den maximalem Knotenabstand an, d.h., der Knoten i kann nur mit den Knoten $i + 1, ..., i + q$ verbunden werden. Insgesamt können somit bei gegebenem $q < M$ maximal

$$(M - q) \cdot q + \sum_{i=1}^{q-1} i = (M - q) \cdot q + \frac{q - 1}{2} \cdot q = ½ \cdot q \cdot (2 \cdot M - q - 1)$$

Kanten eingefügt werden. Ist dieser Wert kleiner als k, so wird k reduziert. Die Auswahl einer Kante geschieht mittels einer Zufallszahl z aus dem Intervall $[0; ½ \cdot q \cdot (2 \cdot M - q - 3)]$. Nachfolgend stehe v als Abkürzung für den Term

$$v = (M - q) \cdot q.$$

Ist

$$z < v,$$

so wählt der Algorithmus die Kante $(z \operatorname{div} q + 1, z \operatorname{div} q + z \bmod q + 2)$. Im Fall

$$z \geqslant v$$

kann man die zu z gehörende Kante nicht so leicht ermitteln. Sie beginnt im Knoten $i + M - q$ mit

$$\sum_{j=1}^{i-1} q - j \leqslant z - v < \sum_{j=1}^{i} q - j$$

$$\Leftrightarrow \quad (i - 1) \cdot q - \frac{i - 1}{2} \cdot i \leqslant z - v < i \cdot q - \frac{i + 1}{2} \cdot i$$

$$\Leftrightarrow \quad i^2 + i \cdot (-1 - 2 \cdot q) + 2 \cdot q + 2 \cdot (z - v) \geqslant 0 \quad \text{und} \quad i^2 + i \cdot (1 - 2 \cdot q) + 2 \cdot (z - v) < 0$$

$$\Leftrightarrow \left(i - \left(\frac{1}{2} + q + \sqrt{\frac{1}{4} + q + q^2 - 2 \cdot q - 2 \cdot (z - v)}\right)\right) \qquad (1a)$$

$$\cdot \left(i - \left(\tfrac{1}{2} + q - \sqrt{\tfrac{1}{4} + q + q^2 - 2 \cdot q - 2 \cdot (z - v)}\right)\right) \geqslant 0 \quad (1b)$$

und $\left(i - \left(-\tfrac{1}{2} + q + \sqrt{\tfrac{1}{4} - q + q^2 - 2 \cdot (z - v)}\right)\right)$ (2a)

$$\cdot \left(i - \left(-\tfrac{1}{2} + q - \sqrt{\tfrac{1}{4} - q + q^2 - 2 \cdot (z - v)}\right)\right) < 0. \quad (2b)$$

Da $i \leqslant q$ ist, wird der Faktor (1a) immer negativ. Demzufolge muß

$$i \leqslant \tfrac{1}{2} + q - \sqrt{\tfrac{1}{4} + q + q^2 - 2 \cdot q - 2 \cdot (z - v)} = \tfrac{1}{2} + q - \sqrt{\tfrac{1}{4} - q + q^2 - 2 \cdot (z - v)} \quad (3)$$

sein. Außerdem gilt

$$\tfrac{q - 1}{2} \cdot q \geqslant z - v.$$

$$\Leftrightarrow \quad \tfrac{1}{4} - q + q^2 - 2 \cdot (z - v) \geqslant \tfrac{1}{4}$$

$$\Rightarrow \quad \sqrt{\tfrac{1}{4} - q + q^2 - 2 \cdot (z - v)} \geqslant \tfrac{1}{2}$$

$$\Leftrightarrow \quad -\tfrac{1}{2} + q + \sqrt{\tfrac{1}{4} - q + q^2 - 2 \cdot (z - v)} \geqslant q$$

$$\Rightarrow \quad -\tfrac{1}{2} + q + \sqrt{\tfrac{1}{4} - q + q^2 - 2 \cdot (z - v)} \geqslant i$$

Somit ist der Faktor (2a) nie positiv. Aus diesem Grund muß

$$i > -\tfrac{1}{2} + q - \sqrt{\tfrac{1}{4} - q + q^2 - 2 \cdot (z - v)} \quad (4)$$

sein. Mit Hilfe der Ungleichungen (3) und (4) gelangt man zu

$$i = \left\lfloor \tfrac{1}{2} + q - \sqrt{\tfrac{1}{4} - q + q^2 - 2 \cdot (z - v)} \right\rfloor.$$

Wenn

$$z - v = \sum_{j=1}^{i-1} q - j \quad \Leftrightarrow \quad z - v - (i - 1) \cdot q + \tfrac{i - 1}{2} \cdot i = 0$$

ist, wählt der Algorithmus den Knoten $i + M - q + 1$ als Ende der Kante. Im allgemeinen wird also die Kante

$$\left(i + M - q,\ i + M - q + 1 + z - v - (i - 1) \cdot q + \tfrac{i - 1}{2} \cdot i\right)$$

eingefügt.

Da es an dieser Stelle sicher günstig ist, die Kantenanzahl als Funktion der Modulanzahl anzugeben, kann man in der Beispielgenerierungsdatei über die Variable m auf die Anzahl der Module zugreifen. Bei der Festlegung des Knotenabstands q ist zusätzlich die Angabe von k für die Kantenanzahl möglich. Als Rechenoperationen werden +, –, *, / und ^ erkannt. Werden als Parameter natürliche Zahlen erwartet, so ist das kleinste zulässige Ergebnis bei einer Subtraktion Null. Ebenso wird in diesem Fall nur eine ganzzahlige Division durchgeführt. Zusätzlich ist die Verwendung von Klammern ( ) möglich. Außerdem liefert {x; y} das Minimum beliebig vieler Zahlen. Mit [x] kann man die größte ganze Zahl kleiner oder gleich x ermitteln. Dabei wird x immer, also auch wenn eine natürliche Zahl verlangt ist, als Fließkommazahl behandelt. Man beachte, daß die Leerzeichen als Trennsymbole für die Verteilungsparameter fungieren. Demzufolge dürfen die Formeln keine Leerzeichen enthalten. Die

Beispielgenerierungsdatei /make/prg5/bsp5.txt (siehe Abbildung 109) basiert auf dem Beispiel 3.

```
PROZESSORANZAHL: {N}

MODULANZAHL: konstant {M}

ANZAHL DER KANTEN IM MODULABHÄNGIGKEITSGRAPH: gleich 0 (m*(m-1))/2
KNOT I MIT NÄCHST ? KNOT VERBIND: [(2*m-1-(4*m^2-4*m+1-8*k)^(1/2))/2] m-1

MODULWAHRSCH LAUFZEIT_MIT_EIN_PROZ  PARALLELISIERBARKEIT
gleich 1 3   exponential 1 100 1    BIS {0,4·N} PROZ: normal 0 1 0.5 0.2;
                                    BIS {N} PROZ: konstant 0
konstant 3.5 normal 0.5 4.7 2.2 1   BIS {0,7·N} PROZ: exponential 0 1 0.5;
                                    BIS {N} PROZ: konstant 0.5
```

Abbildung 109 - /make/prg5/bsp5.txt - allgemeine Form.

Bei den Testläufen wurden die in der folgenden Tabelle angegebenen Einstellungen gewählt. Die angegebenen Dateien befinden sich auf der beiliegenden CD.

Algorithmus	Dateiname	Bemerkungen
„SchT“	/src/prg5/opti5t.c	
„SchHT“	/src/prg5/opti5s.c	
„WaSP“	/src/prg5/opti5b.c	Der Konstruktionsbaum wird bei mehreren aufeinander folgenden parallelen Vereinigungen ausbalanciert.
„ReSP“	/src/prg5/opti5r.c	Siehe „WaSP“.
„KP+“	/src/prg5/opti5j.c	Zur Wahl des gewünschten Gütefaktors wird `GUETEFAKTOR 1` bzw. `GUETEFAKTOR 2` gesetzt. Zwei Zahlen werden als gleich angesehen, wenn ihre Differenz kleiner als `EPSILON 0.00001` ist.
„KPMF“	/src/prg5/opti5j.c	Siehe „KP+“. Zur Abgrenzung von „KP+“ wird noch `MINFLAE` gesetzt.

Läßt man auch nicht formbare Module zu, so kann man das in der Abbildung 110 angegebene Beispiel 6 zum Testen der Algorithmen verwenden.

```
PROZESSORANZAHL: {N}

MODULANZAHL: konstant {M}

ANZAHL DER KANTEN IM MODULABHÄNGIGKEITSGRAPH: gleich 0 (m*(m-1))/2
KNOT I MIT NÄCHST ? KNOT VERBIND: [(2*m-1-(4*m^2-4*m+1-8*k)^(1/2))/2] m-1

MODULW  MIN_PROZ     LAUFZ_MIN_PROZ    PARALLELISIERBARKEIT
gl 1 3  gl 2 {0,8·N} ex 1 100 1        BIS {0,4·N} P: no 0 {2/N} {1/N} {4/(10·N)};
                                       BIS {N} P: gl 0 {1/N}
ko 1.5  gl 2 {0,8·N} no 0.5 4.7 2.2 1  BIS {0,7·N} P: ex 0 1 {N};
                                       BIS {N} P: gl 0 {2/N}
ko 0.5  ko 1         ex 1 100 0.7      BIS {N} P: no 0 1 0.4 0.3
```

Abbildung 110 - /make/prg6/bsp6.txt - allgemeine Form.

9.25 Laufzeitberechnung für die Standardbeispiele

9.25.1 Matrixmultiplikation

Damit die verschiedenen Optimierungsalgorithmen gestartet werden können, muß man die Laufzeit der Module berechnen und den Modulabhängigkeitsgraphen erstellen. Dabei werden drei verschiedene Komplexitäten unterschieden:

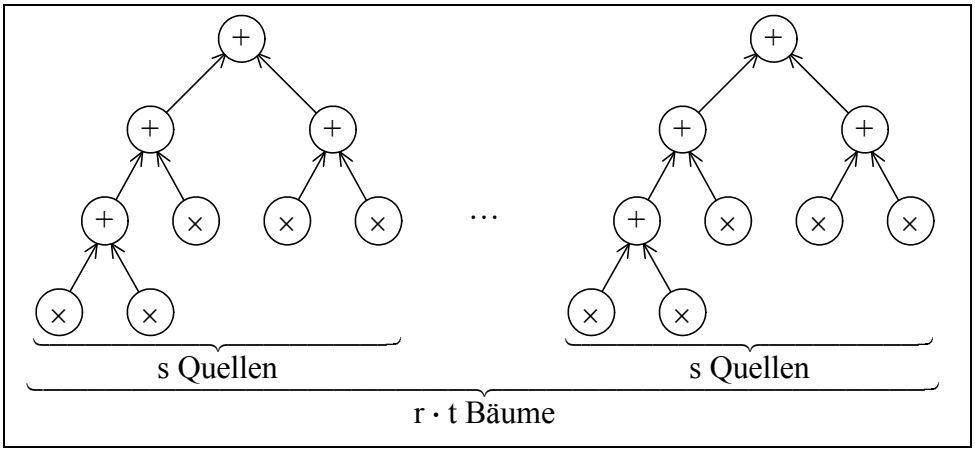


Abbildung 111 - Operationsgraph Matrixmultiplikation.

Bei der Komplexität Eins wird die gesamte Multiplikation einer r × s - Matrix mit einer s × t - Matrix auf ein Modul x abgebildet. Es entsteht also ein Modulabhängigkeitsgraph ohne Kanten. Zur Berechnung der Laufzeit des Moduls x benötigt man den Operationsgraphen eines Algorithmus zur Matrixmultiplikation (siehe Abbildung 111). Der Operationsgraph stellt die zur Berechnung des Matrixprodukts notwendigen Operationen (Additionen und Multiplikationen zweier reeller Zahlen) und den zwischen den Operationen vorhandenen Datenfluß dar.
In der nachfolgenden Berechnung der Laufzeit eines Schedules werden die Kanten des Baumes umgedreht, d.h., es werden zuerst die Additionen und dann die Multiplikationen ausgeführt.
Der Operationsgraph hat eine maximale Tiefe von $v = \lceil \log_2(s) + 1 \rceil$, wobei es in der Schicht i mit $1 \leqslant i < v$ genau $2^{i-1} \cdot r \cdot t$ Knoten gibt. In der Schicht v sind noch

$$w = r \cdot t \cdot (2 \cdot s - 1 - (2^{v-1} - 1)) = r \cdot t \cdot (2 \cdot s - 2^{v-1})$$

Multiplikationen enthalten. Es sei k die erste Schicht, in der mehr Operationen ausgeführt werden als Prozessoren verfügbar sind.
Fall 1: Es gibt kein derartiges k, d.h. in allen Schichten sind maximal N Operationen auszuführen.
Die optimale Laufzeit für die komplette Matrixmultiplikation ist somit (v - 1) + y. Dabei benötigt eine Addition eine Zeiteinheit und eine Multiplikation y Zeiteinheiten. Die Definition von y gelte auch für alle folgenden Fälle.
Fall 2: Alle Operationsknoten der Tiefe k sind Multiplikationen.
Da nur die Blätter des Graphen Multiplikationen darstellen, muß k = v gelten. Somit ist die optimale Laufzeit der Matrixmultiplikation $(v - 1) + \left\lceil \frac{w}{N} \right\rceil \cdot y$.
Fall 3: In der Schicht k existiert mindestens eine Addition.
Zuerst muß man sich verdeutlichen, daß jede ausgeführte Addition die Ausführung von 2 weiteren Operationen ermöglicht. Da schon in Schicht k mehr Operationen zur parallelen

Ausführung bereit stehen als Prozessoren verfügbar sind, gilt das auch für alle nachfolgenden Schichten j mit $k < j < v$. Führt man also bevorzugt die Additionen in der Schicht mit der niedrigsten Nummer aus, so stehen bis zum Start der letzten Addition immer genug Additionen zur Auslastung aller Prozessoren zur Verfügung.
Insgesamt gibt es $a = r \cdot t \cdot (s - 1)$ Additionen. Davon sind bis einschließlich Schicht $k - 1$ schon $b = 2^{k-1} - 1$ Stück in $k - 1$ Zeiteinheiten ausgeführt wurden. Die restlichen $a - b$ Additionen benötigen $c = \left\lfloor \frac{a-b}{N} \right\rfloor$ Mal alle N Prozessoren und anschließend noch einmal $d = a - b - c \cdot N$ Prozessoren.
Fall 3.1: Es gilt $d = 0$.
In diesem Fall können alle $s \cdot r \cdot t$ Multiplikationen auf allen N Prozessoren ausgeführt werden. Es ergibt sich als Gesamtlaufzeit $(k - 1 + c) + \left\lceil \frac{s \cdot r \cdot t}{N} \right\rceil \cdot y$. Man beachte, daß dies unter Umständen nicht der optimale Schedule ist.
Fall 3.2: Es ist $d \neq 0$.
Auf den verbleibenden $N - d$ Prozessoren können parallel zu den d Additionen maximal $e = \left\lceil \frac{1}{y} \right\rceil \cdot (N - d)$ Multiplikationen ausgeführt werden. Dabei wird keine Multiplikation nach dem Ende der Additionen mehr neu gestartet. Die d Additionen haben als Nachfolger noch $2 \cdot d$ Multiplikationen. Die restlichen $f = s \cdot r \cdot t - 2 \cdot d$ Multiplikationen sind jedoch sofort ausführbar.
Fall 3.2.1: Es ist $f \leqslant e - (N - d)$.
In diesem Fall sind alle N Prozessoren zum Endzeitpunkt der Additionen frei und es müssen noch $h = s \cdot r \cdot t - f$ Multiplikationen ausgeführt werden. Dies kann in $\left\lceil \frac{h}{N} \right\rceil \cdot y$ Zeiteinheiten geschehen. Somit beträgt die Gesamtlaufzeit eines guten Schedules $(k - 1 + c) + \left\lceil \frac{h}{N} \right\rceil \cdot y$.
Fall 3.2.2: Es gilt $e - (N - d) < f < e$.
Von den $N - d$ Prozessoren können zum Endzeitpunkt der letzten Addition $e - f$ Prozessoren keine Multiplikation mehr ausführen. Es sind also $g = d + e - f$ Prozessoren frei. Außerdem müssen noch $h = s \cdot r \cdot t - f$ Multiplikationen ausgeführt werden.
Fall 3.2.3: Es ist $e \leqslant f$.
Somit können die $N - d$ Prozessoren vollständig ausgelastet werden. Zum Endzeitpunkt der letzen Addition sind also $g = d$ Prozessoren frei und es müssen noch $h = s \cdot r \cdot t - e$ Multiplikationen ausgeführt werden.
Ab hier werden die Fälle 3.2.2 und 3.2.3 wieder gemeinsam betrachtet. Zur Ausführung der h Multiplikationen benötigt man $i = \left\lfloor \frac{h}{N} \right\rfloor$ Mal alle N Prozessoren. Gilt $h - N \cdot i = 0$, so erhält man als Gesamtlaufzeit $(k - 1 + c) + \left(\left\lceil \frac{1}{y} \right\rceil + i \right) \cdot y$. Ist $0 < h - N \cdot i \leqslant g$, so können die verbleibenden $h - N \cdot i$ Multiplikationen auf den g Prozessoren ausgeführt werden, die zum Ende der Additionen keine Multiplikation ausführen, und es ergibt sich als Gesamtlaufzeit des Schedules $(k - 1 + c + 1) + (i + 1) \cdot y$. Sollte $h - N \cdot i > g$ sein, so dominieren die Prozessoren die parallel zu den letzten Additionen schon Multiplikationen ausführen, die Gesamtlaufzeit des Schedules: $(k - 1 + c) + \left(\left\lceil \frac{1}{y} \right\rceil + i + 1 \right) \cdot y$.
Die Abbildung 112 verdeutlicht die soeben geführten Überlegungen noch einmal.
Bei der Komplexität Zwei wird für jeden der $r \cdot t$ Bäume ein Modul erstellt. Die Laufzeiten der Module erhält man mit den Formeln für die Komplexität Eins, indem man dort $t = r = 1$ setzt.

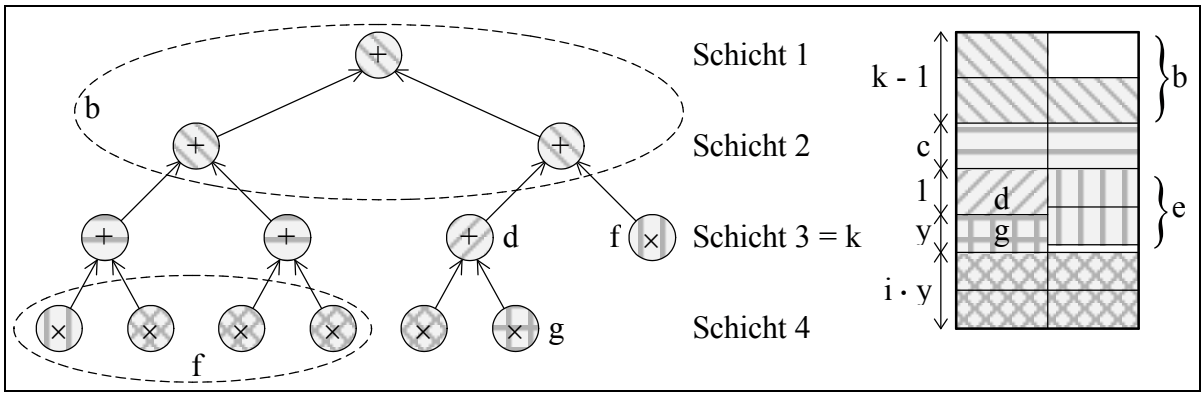


Abbildung 112 - Matrixmultiplikation: Schedule für N = 2, s = 7, r = t = 1.

Wird die Komplexität Drei verwendet, so erhält der Optimierungsalgorithmus r · t balancierte Binärbäume mit je s Quellen. Jeder innere Knoten dieser Bäume stellt ein Modul mit der Laufzeit Eins dar. Für jedes Blatt wird ein Modul mit der Laufzeit einer Multiplikation (relativ zur Addition) angelegt.

9.25.2 Fast Fourier Transformation (FFT)

Bei der Erstellung des Operationsgraphen wird die anfängliche Umsortierung des Eingabevektors vernachlässigt, da diese Operation keine Berechnung zur Folge hat. Außerdem wird nur eine Zweierpotenz als Dimension r ⩾ 2 zugelassen.

Die Komplexität Eins bedeutet auch hier, daß die gesamte FFT auf ein Modul abgebildet wird. Leider ist durch die Mischung von Additionen und Multiplikationen die Berechnung der möglichst minimalen Laufzeiten des Moduls nicht trivial. Aus diesem Grund wird der Operationsgraph wie bei der Komplexität Drei (siehe unten) erstellt und mit Hilfe eines Greedy - Algorithmus ein Schedule der einzelnen Operationen berechnet. Dieser Schedule definiert dann die Laufzeit des Moduls. Der verwendete Greedy - Algorithmus versucht alle Operationen so zeitig wie möglich auszuführen. Dabei haben Operationen in einer kleineren Schicht eine höhere Priorität.

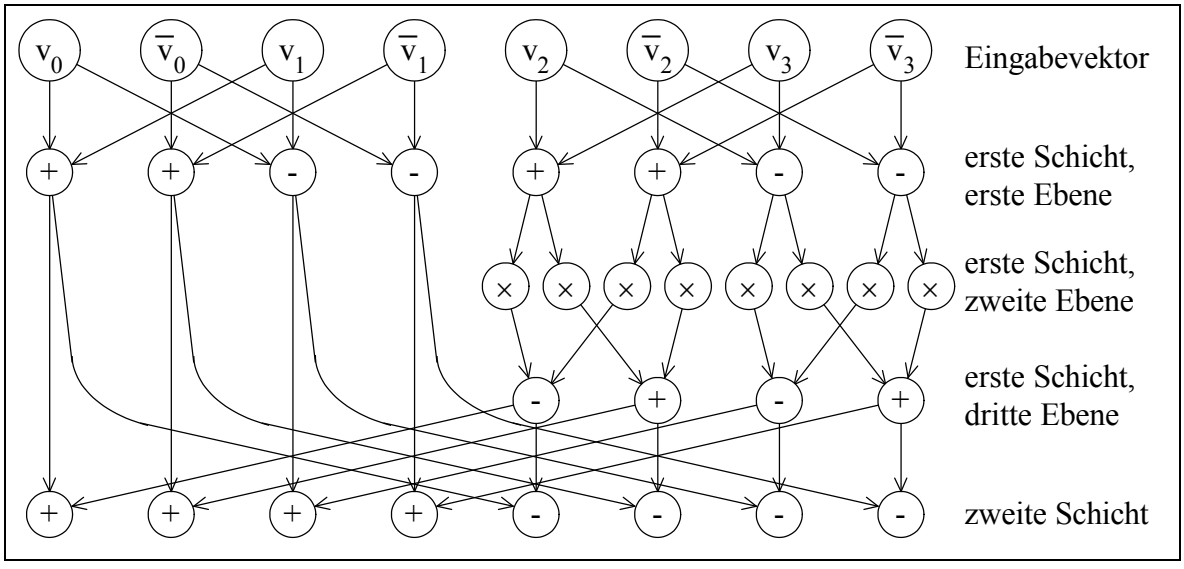


Abbildung 113 - Operationsgraph FFT für r = 4.

Bei der Komplexität Zwei erstellt der Algorithmus für jede FFT zwei balancierte Binärbäume mit der Gesamttiefe $\log_2(r)$. Dazu wird wie schon bei der Matrixmultiplikation die Richtung des Datenflusses umgedreht, d.h., die Wurzeln der Bäume stellen die Operationen in der Schicht mit der höchsten Nummer dar. Jede Wurzel repräsentiert also $½ \cdot r$ komplexzahlige Additionen und Subtraktionen. Zur besseren Lesbarkeit wird ab jetzt auch bei Subtraktionen nur noch von Additionen geredet. Die Laufzeit beider Operationen ist sowieso gleich.

Die Baumknoten mit der Tiefe b repräsentieren $\frac{r}{2^b}$ komplexzahlige Operationen der Schicht $\log_2(r) - b + 1$. Ist der Baumknoten ein linkes Kind, so handelt es sich bei den Operationen um Additionen. Im Falle eines rechten Kindes um Multiplikationen.

In der Abbildung 114 sind die Wurzeln der beiden Bäume ganz unten und haben die Nummern 13 und 14. Weiterhin haben die Knoten des einen Modulabhängigkeitsbaums gerade und die des anderen ungerade Nummern. Die Elementaroperationen, welche einen Baumknoten bilden, sind mit einer zusammenhängenden Schraffur gekennzeichnet.

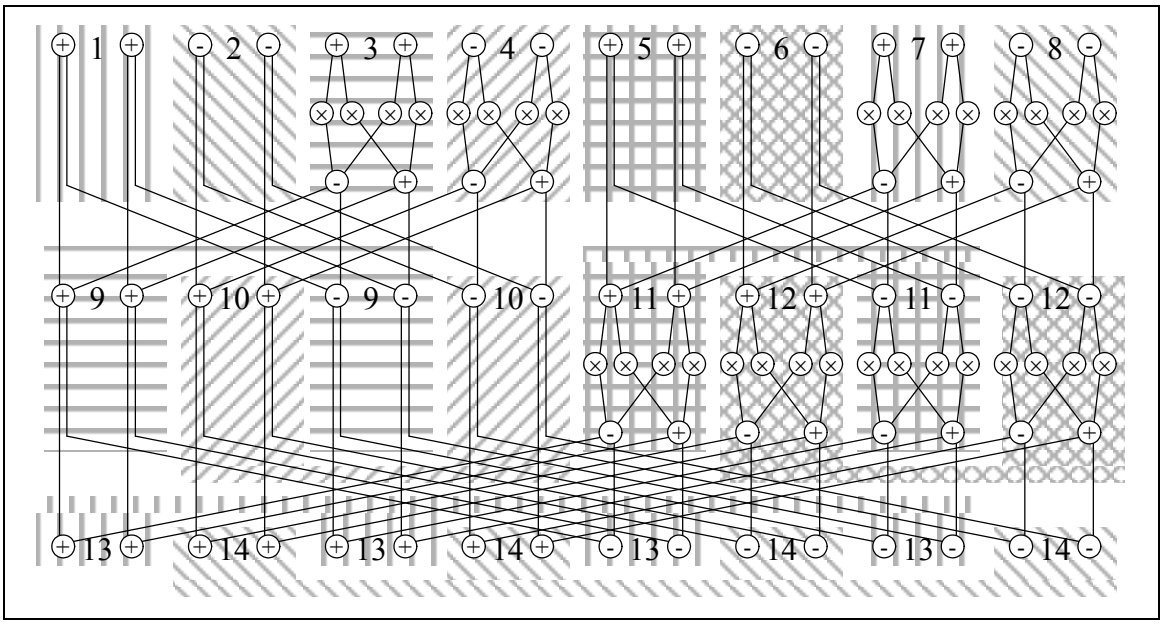


Abbildung 114 - Operationsgraph FFT und die beiden Modulabhängigkeitsgraphen.

Die Laufzeit eines Knotens, welcher i komplexe Additionen darstellt, kann wie folgt berechnet werden: Die i komplexen Additionen zerfallen in $2 \cdot i$ unabhängige normale Additionen. Der Knoten hat also eine Laufzeit von $\left\lceil \frac{2 \cdot i}{N} \right\rceil$.

Wesentlich komplizierter ist die Berechnung der Laufzeit von i komplexen Multiplikationen. Deshalb wird bei den folgenden Überlegungen ausgenutzt, daß $½ \leqslant y < 1$ gilt. Der Wert y gibt dabei die Laufzeit einer Multiplikation an, wenn die Laufzeit einer Addition Eins ist.

Bei einer komplexen Multiplikation können zuerst die $2 \cdot i$ unabhängigen Additionen der ersten Ebene ausgeführt werden. Dazu führen alle N Prozessoren $a = \left\lfloor \frac{2 \cdot i}{N} \right\rfloor$ Additionen aus und anschließend noch einmal $b = 2 \cdot i - N \cdot a$ Prozessoren eine Addition aus.

Fall 1: Es ist $a = 0$.

In diesem Fall können alle Additionen der ersten und dritten Ebene jeweils parallel ausgeführt werden. Da nach der letzen Multiplikation noch mindestens eine Addition der dritten Ebene

ausgeführt werden muß, ergibt sich als optimale Gesamtlaufzeit $2 + y \cdot \left\lceil \frac{4 \cdot i}{N} \right\rceil$.

Fall 2: Es gilt $a \geqslant 1$ und $b = 0$.

Demzufolge stehen zur Ausführung der $4 \cdot i$ Multiplikationen wieder alle N Prozessoren zur Verfügung. Da bereits $2 \cdot i$ ein Vielfaches von N war, ist auch $4 \cdot i$ ein Vielfaches von N und die Multiplikationen benötigen genau $y \cdot \frac{4 \cdot i}{N}$ Zeiteinheiten. Zusammen mit den nachfolgenden $2 \cdot i$ Additionen hat der optimale Schedule eine Gesamtlaufzeit von $2 \cdot a + y \cdot \frac{4 \cdot i}{N}$.

Fall 3: Es ist $a \geqslant 1$ und $2 \cdot b \leqslant N$.

In diesem Fall können die jeweils b Restadditionen der ersten und dritten Ebene auf unterschiedlichen Prozessoren ausgeführt werden. Die Abbildung 115 verdeutlicht die Anordnung der einzelnen Operationen.

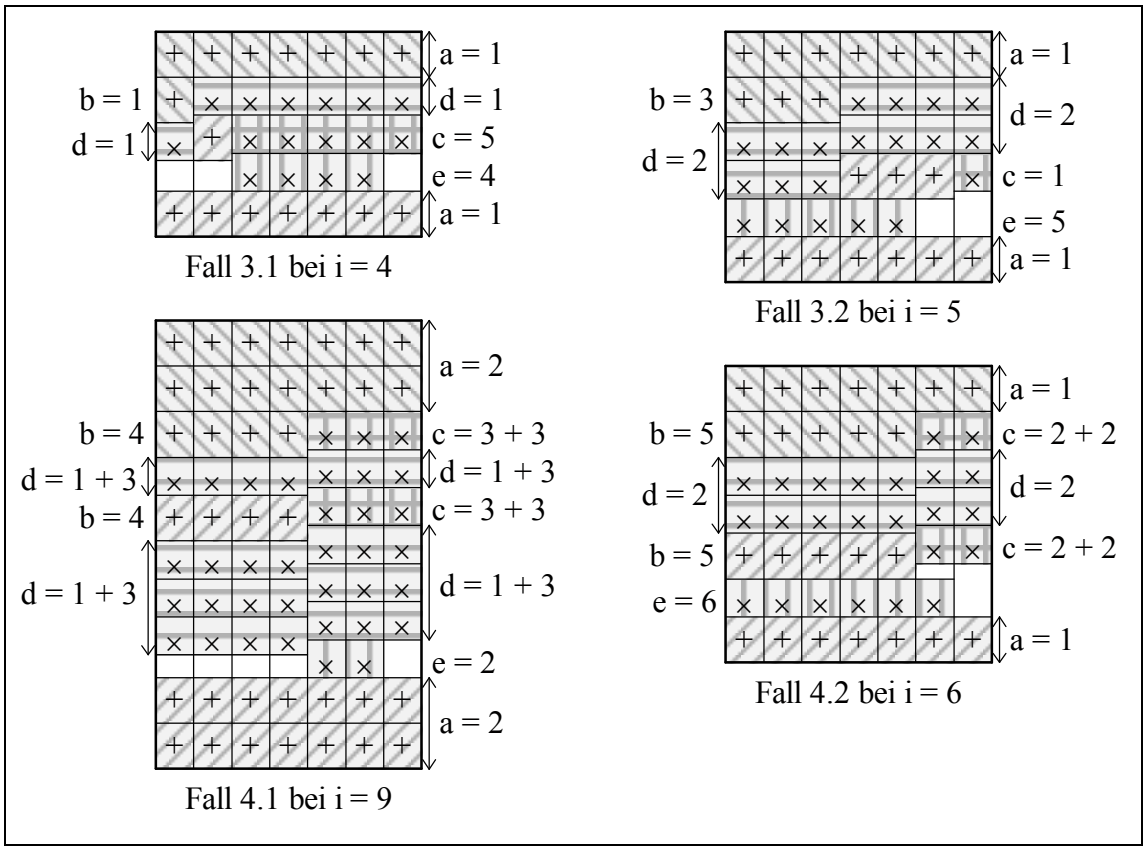


Abbildung 115 - Modullaufzeiten für FFT bei Komplexität 2 und N = 7.

Fall 3.1: Es gilt $3 \cdot b \leqslant N - 1$.

Aus der Fallvoraussetzung läßt sich ableiten:

$$N - 1 \geqslant 3 \cdot b \quad \Leftrightarrow \quad N - b - 1 \geqslant 2 \cdot b \quad \Leftrightarrow \quad \frac{N - b - 1}{2} \geqslant b \quad \Rightarrow \quad \left\lfloor \frac{N - b}{2} \right\rfloor \geqslant b.$$

Man erkennt, daß jede Addition der dritten Ebene genau 2 Multiplikationen als Vorgänger hat. Somit reichen die parallel zu den b Restadditionen der ersten Ebene ausführbaren $N - b$ Multiplikationen aus, um die b Restadditionen der dritten Ebene auszuführen. Da $a \geqslant 1$ ist, sind auch mindestens $2 \cdot N$ Multiplikationen vorhanden und gleichzeitig mit den b Restadditionen der ersten Ebene startbar. Diese $2 \cdot N$ Multiplikationen reichen aus, um parallel zu den jeweils b Additionen der ersten und dritten Ebene jeweils $N - b$ Multiplikationen auszuführen.

Fall 3.2: Es gilt $3 \cdot b > N - 1$.
Bevor in diesem Fall die b Restadditionen der dritten Ebene gestartet werden können, müssen $2 \cdot (N - b)$ Multiplikationen ausgeführt wurden sein, denn aus der Voraussetzung von Fall 3 folgt:

$$N \geqslant 2 \cdot b \quad \Leftrightarrow \quad N - b \geqslant b \quad \Leftrightarrow \quad \left\lfloor \frac{2 \cdot (N - b)}{2} \right\rfloor \geqslant b.$$

Auch diese $2 \cdot (N - b)$ Multiplikationen sind wegen $a \geqslant 1$ parallel zu den b Restadditionen der ersten Ebene ausführbar. Da die Laufzeit von 2 Multiplikationen nach Voraussetzung mindestens so hoch ist, wie die Laufzeit einer Addition, sind bis zum Ende der b Restadditionen der ersten Ebene alle Prozessoren ausgelastet. Nach dem Ende der b Restadditionen der ersten Ebene sind alle vorhandenen Multiplikationen ausführbar. Somit werden auf den b Prozessoren, die die Restadditionen der ersten Ebene ausgeführt haben, noch je 2 Multiplikationen ausgeführt. Nach dem Ende der ersten $2 \cdot (N - b)$ Multiplikationen werden auf den $N - b$ Prozessoren die b Restadditionen der dritten Ebene ausgeführt. Die verbleibenden $(N - b) - b$ Prozessoren führen wiederum je eine Multiplikation aus. Insgesamt werden also parallel zu den Restadditionen der Ebenen Eins und Drei exakt

$$2 \cdot (N - b) + 2 \cdot b + (N - b) - b = 3 \cdot N - 2 \cdot b$$

Multiplikationen ausgeführt. Da $a \geqslant 1$ ist, existieren mindestens $N + b$ Additionen in der ersten Ebene. Somit sind auch mindestens

$$2 \cdot (N + b) = 2 \cdot N - b + 3 \cdot b > 2 \cdot N - b + N - 1 \qquad \text{(nach Fallvoraussetzung)}$$
$$= 3 \cdot N - b - 1 \geqslant 3 \cdot N - 2 \cdot b$$

Multiplikationen vorhanden.
In den Fällen 3.1 und 3.2 führen $c = N - 2 \cdot b$ Prozessoren anstatt einer Restaddition eine Multiplikation aus. Somit verbleiben noch $4 \cdot i - c$ Multiplikationen. Diese belegen $d = \left\lfloor \frac{4 \cdot i - c}{N} \right\rfloor$ Mal alle Prozessoren. Anschließend verbleiben noch $e = 4 \cdot i - c - N \cdot d$ Multiplikationen. Gilt $e = 0$, so erhält man als Gesamtlaufzeit $2 \cdot a + 1 + d \cdot y$. Ist $0 < e \leqslant c$, so können die e Multiplikationen ebenfalls noch parallel zur den Restadditionen ausgeführt werden und es ergibt sich eine Gesamtlaufzeit von $2 \cdot a + (d + 2) \cdot y$. Sollte $e > c$ sein, so erhält man eine Gesamtlaufzeit von $2 \cdot a + 1 + (d + 1) \cdot y$.
Fall 4: Es ist $a \geqslant 1$ und $2 \cdot b > N$.
In diesem Fall können die Restadditionen der Ebenen 1 und 3 nicht auf verschiedenen Prozessoren ausgeführt werden. Es gibt also mindestens einen Prozessor, welcher zwei Restadditionen ausführt. Zur Berechnung der Gesamtlaufzeit wird deshalb davon ausgegangen, daß die b Restadditionen der ersten Ebene auf den gleichen b Prozessoren wie die b Restadditionen der dritten Ebene ausgeführt werden.
Fall 4.1: Es gilt $2 \cdot N - 1 \geqslant 3 \cdot b$.
Parallel zu den b Restadditionen der Ebene 1 können noch $N - b$ Multiplikationen ausgeführt werden. Anschließend führen alle Prozessoren noch eine Multiplikation aus. Somit sind $2 \cdot N - b$ Multiplikationen ausgeführt. Nach dem Ende dieser Multiplikationen können wegen

$$\left\lfloor \frac{2 \cdot N - b}{2} \right\rfloor \geqslant \frac{2 \cdot N - b - 1}{2} = \frac{(2 \cdot N - 1) - b}{2} \geqslant \frac{(3 \cdot b) - b}{2} \quad \text{(nach Fallvoraussetzung 4.1)}$$

mindestens b Restadditionen der Ebene 3 ausgeführt werden. Parallel zu diesen Restadditionen sind wiederum $N - b$ Multiplikationen ausführbar.

Fall 4.2: Es ist $2 \cdot N - 1 < 3 \cdot b$.
Im Gegensatz zu Fall 4.1 werden diesmal zwischen der Ausführung der Restadditionen der Ebenen 1 und 3 genau 2 Multiplikationen ausgeführt. Insgesamt sind also $3 \cdot N - b$ Multiplikationen beendet. Wegen

$$\left\lfloor \frac{3 \cdot N - b}{2} \right\rfloor \geqslant \frac{3 \cdot N - b - 1}{2} = \frac{2 \cdot N + (N - (b + 1))}{2} \geqslant \frac{2 \cdot N}{2} \quad \text{(b ist nach Definition echt kleiner als N)}$$

sind mindestens N Restadditionen der dritten Ebene ausführbar. Parallel zu den b Restadditionen der Ebene 3 können wiederum $N - b$ Multiplikationen ausgeführt werden.
Da es zur Berechnung der Gesamtlaufzeit egal ist, ob eine oder zwei Multiplikationen zwischen den Restadditionen der Ebenen 1 und 3 ausgeführt wurden, können die Fälle 4.1 und 4.2 nun wieder gemeinsam betrachtet werden.
Insgesamt wurden bisher maximal $4 \cdot N - 2 \cdot b$ Multiplikationen positioniert. Dies ist wegen

$$4 \cdot N - 2 \cdot b = 2 \cdot N - 2 \cdot b + 2 \cdot (N) < 2 \cdot N - 2 \cdot b + 2 \cdot (2 \cdot b) \quad \text{(nach Fallvoraussetzung 4)}$$

weniger als die mindestens vorhandenen $2 \cdot N + 2 \cdot b$ Multiplikationen.
Von den $4 \cdot i$ Multiplikationen werden $c = 2 \cdot (N - b)$ Stück parallel zu den Restadditionen ausgeführt. Die anderen Multiplikationen lasten $d = \left\lfloor \frac{4 \cdot i - c}{N} \right\rfloor$ Mal alle Prozessoren aus und es verbleiben $e = 4 \cdot i - c - N \cdot d$ Multiplikationen. Gilt $e = 0$, so erhält man als Gesamtlaufzeit $2 \cdot a + 2 + d \cdot y$. Ist $0 < e \leqslant N - b$, so können die e Multiplikationen auch noch parallel zu den Restadditionen ausgeführt werden und es ergibt sich als Gesamtlaufzeit $2 \cdot a + (d + 3) \cdot y$. Falls $e > N - b$ ist, so dominieren die Prozessoren mit den Restadditionen die Gesamtlaufzeit: $2 \cdot a + 2 + (d + 1) \cdot y$.
Die Komplexität Drei bedeutet auch bei der FFT, daß jedes Modul einer Operation im Operationsgraphen entspricht und demzufolge der Modulabhängigkeitsgraph mit dem Operationsgraphen identisch ist.

9.25.3 LR - Zerlegung mittels Gauß

Wird das Standardbeispiel mit der Komplexität Eins erzeugt, so entspricht eine LR - Zerlegung einem Modul. Wie auch bei der FFT wird zur Bestimmung der Modullaufzeiten der Operationsgraph erstellt und mit Hilfe eines Greedy - Algorithmus ein Schedule erstellt. Dazu werden die Knoten des Graphen zeilenweise von links nach rechts durchnumeriert (vgl. Abbildung 116). Der Algorithmus startet dann jeweils den Knoten mit der kleinsten Nummer so zeitig wie möglich.
Bei der Komplexität Zwei wird der Operationsgraph auf einen serien - parallelen Graphen abgebildet. Dazu bildet jede Gruppe aus Divisionen mit den direkt vor ihr ausgeführten Subtraktionen ein Modul (siehe Abbildung 117). Bei einer LR - Zerlegung der Dimension r entstehen somit r Schichten, wobei die Schicht i aus $j = r + 1 - i$ Modulen besteht. Jedes dieser Module führt dabei j Divisionen und, außer in der ersten Schicht, $j + 1$ Subtraktionen aus. Die Laufzeit der Module in der ersten Schicht ist $z \cdot \left\lceil \frac{j}{N} \right\rceil$. Dabei steht z für die Laufzeit einer Division unter der Voraussetzung, daß die Laufzeit einer Addition Eins ist.
Für die Module der Schichten 2 bis r kann die Laufzeit analog zu den Überlegungen bei der Matrixmultiplikation bestimmt werden:
Fall 1: Es ist $j + 1 \leqslant N$.
Somit können zuerst alle Subtraktionen und dann alle Divisionen parallel ausgeführt werden und es ergibt sich die optimale Laufzeit $1 + z$.

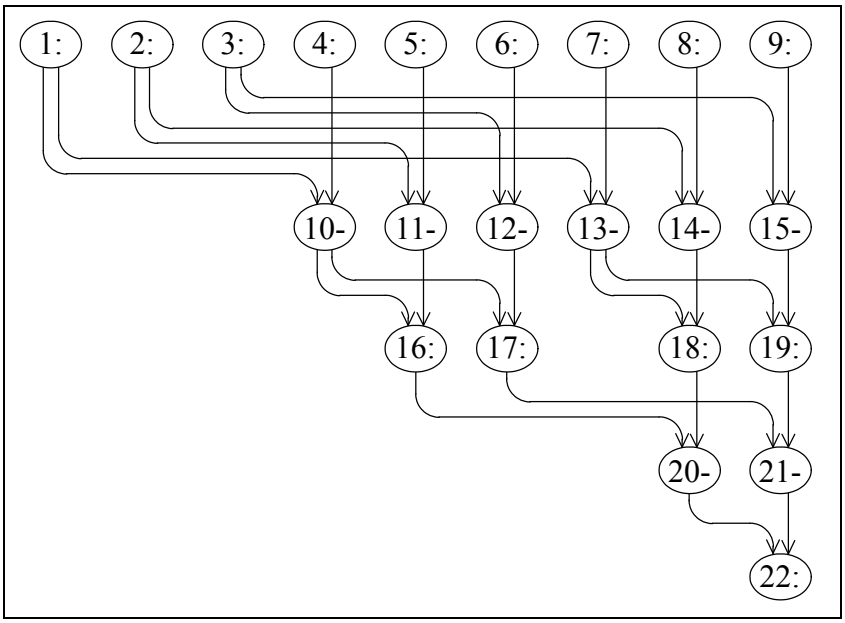


Abbildung 116 - Operationsgraph LR - Zerlegung.

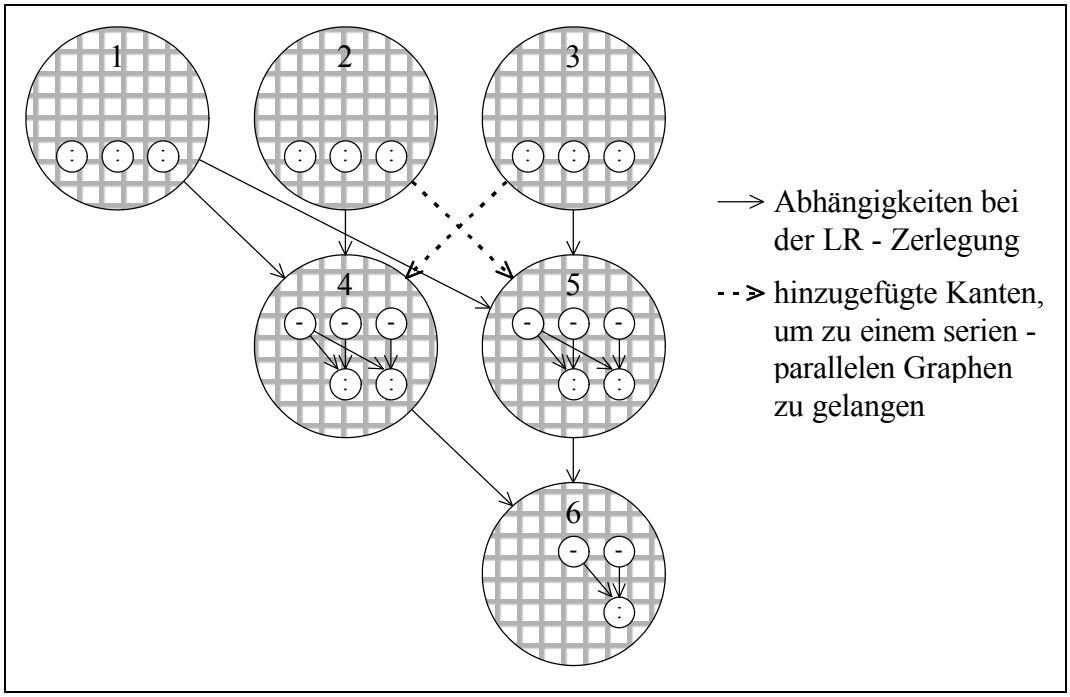


Abbildung 117 - LR - Zerlegung - Komplexität Zwei.

Fall 2: Es gilt j = N.
Da nach der letzten Subtraktion noch mindestens eine Division ausgeführt werden muß, beträgt die optimale Laufzeit 2 + z.
Fall 3: Es ist j > N.
Unter der Annahme, daß z ⩾ 1 ist, erweist es sich als günstig, mit der Positionierung der Divisionen zu beginnen. Insgesamt können $a = \left\lfloor \frac{j}{N} \right\rfloor$ Mal alle Prozessoren mit der Ausführung

einer Division ausgelastet werden. Die verbleibenden $b = j - a \cdot N$ Divisionen werden anschließend auf b verschiedenen Prozessoren ausgeführt.
Fall 3.1: Es ist $b = 0$.
Wie im Fall 2 müssen vor den Divisionen noch $j + 1 = N \cdot a + 1$ Subtraktionen ausgeführt werden. Es ergibt sich als nicht unbedingt optimale Laufzeit $z \cdot a + a + 1$.
Fall 3.2: Es gilt $b \neq 0$.
Damit die b Restdivisionen ausgeführt werden können, müssen mindestens $b + 1$ Subtraktionen beendet sein. Da $b < N$ ist, werden auf allen Prozessoren zu Beginn einmal N Subtraktionen ausgeführt. Es verbleiben also noch $c = j + 1 - N$ Subtraktionen. Von diesen können maximal $d = |z| \cdot (N - b)$ parallel zu den Restdivisionen ausgeführt werden.
Fall 3.2.1: Es gilt $c \leqslant d$.
Die Laufzeit ist somit $z \cdot (a + 1) + 1$.
Fall 3.2.2: Es ist $c > d$.
In diesem Fall verbleiben noch $c - d$ Subtraktionen. Ein Großteil von ihnen kann unter Verwendung aller Prozessoren in $e = \left\lfloor \frac{c - d}{N} \right\rfloor$ Zeiteinheiten ausgeführt werden. Es verbleiben anschließend noch $f = c - d - N \cdot e$ Subtraktionen. Gilt $f = 0$, so erhält man also Laufzeit $z \cdot (a + 1) + 1 + e$. Ist $0 < f \leqslant N - b$, so kann die Ausführung der f Subtraktionen noch parallel zu den Restdivisionen erfolgen und es ergibt sich eine Laufzeit von $z \cdot a + |z| + 2 + e$. Falls $f > N - b$ ist, so beträgt die Laufzeit $z \cdot (a + 1) + 2 + e$.

Setzt man die Komplexität auf Drei, so entspricht wieder jede Operation einem Modul.