Reduktion des Kommunikationsaufwands iterierter Runge-Kutta-Verfahren für dünnbesetzte gewöhnliche Differentialgleichungssysteme

Markus Straubinger

Bayreuth Reports on Parallel and Distributed Systems

No. 4, October 2012

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Phone: +49 921 55 7701
Fax: +49 921 55 7702
E-Mail: brpds@ai2.uni-bayreuth.de

REDUKTION DES KOMMUNIKATIONSAUFWANDES ITERIERTER RUNGE-KUTTA-VERFAHREN FÜR DÜNNBESETZTE DIFFERENTIALGLEICHUNGSSYSTEME

Bachelorarbeit eingereicht von:
Markus Straubinger
24.10.2012

Universität Bayreuth
Angewandte Informatik 2

Betreuer: Prof. Dr. Th. Rauber
Dr. Matthias Korch

Abstract

Iterated Runge-Kutta (IRK) methods are a class of solution methods for initial value problems of systems of ordinary differential equations (ODE) which offer a high level of parallelization. While implementations for dense systems of ODE have to exchange vectors of size of the ODE system regularly, specialized solvers can reduce communication costs by exchanging only the few vector elements that are actually required.

In this work, parallel implementations of IRK methods for distributed address space are considered. At first general implementations for dense ODE systems are presented. Afterwards their communication will be optimized for sparse ODE systems and problems with limited access distance. The resulting implementations are examined in terms of runtime and scalability. Therefore measurements made on different computer systems using different sparse ODE systems will be evaluated. It is found that one implementation option is particularly well suited for sparse ODE systems as well for problems with limited access distance.

Zusammenfassung

Iterierte Runge-Kutta (IRK) Verfahren sind eine Klasse von Lösungsverfahren für Anfangswertprobleme gewöhnlicher Differentialgleichungssysteme (DGL), welche ein hohes Parallelisierungspotential besitzen. Während Implementierungen für dichtbesetzte DGL regelmäßig Vektoren der Größe der DGL austauschen müssen, können spezialisierte Löser Kommunikationskosten einsparen, indem nur die wenigen tatsächlich benötigten Vektorelemente ausgetauscht werden.

In dieser Arbeit werden parallele Implementierungen von IRK-Verfahren für verteilten Adressraum betrachtet. Es werden zunächst allgemeine Implementierungen für dichtbesetzte DGL vorgestellt. Anschließend wird deren Kommunikation für dünnbesetzte DGL und Probleme mit beschränkter Zugriffsdistanz optimiert. Die entstandenen Implementierungen werden in Hinsicht auf Laufzeit und Skalierbarkeit untersucht. Dafür werden Messungen auf verschiedenen Rechnersystemen mit unterschiedlichen dünnbesetzten DGL ausgewertet. Dabei wird festgestellt, dass eine Implementierung der Kommunikation sowohl für dünnbesetzte DGL als auch für Probleme mit beschränkter Zugriffsdistanz besonders gut geeignet ist.

Inhaltsverzeichnis

1 Einleitung

1.1 Gewöhnliche Differentialgleichungssysteme

Viele naturwissenschaftliche Probleme lassen sich mit Hilfe von Differentialgleichungssystemen beschreiben. Die numerische Lösung solcher Probleme kann, vor allem wenn diese große Dimensionen annehmen, sehr rechenaufwändig sein. Von daher sind möglichst effiziente und parallele Lösungsverfahren notwendig.

In dieser Arbeit wird die Lösung von Anfangswertproblemen (AWP) gewöhnlicher Differentialgleichungssysteme (DGL) der Form

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0, \quad \mathbf{y} : \mathbb{R} \to \mathbb{R}^n, \quad \mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n, \quad t \in [t_0, t_e]$$

betrachtet. Anhand der Funktion $\mathbf{f}$, auch rechte Seite genannt, lassen sich DGL in dichtbesetzte oder dünnbesetzte Probleme unterteilen. Greifen die Komponentenfunktionen $f_j(t, \mathbf{y}(t))$ auf alle oder sehr viele Elemente des Argumentvektors $\mathbf{y}(t)$ zu, spricht man von einem dichtbesetzten System, anderenfalls von einem dünnbesetzten. Einen Spezialfall von dünnbesetzten DGL stellen Probleme mit beschränkter Zugriffsdistanz dar. Die Zugriffsdistanz $d(\mathbf{f})$ einer Funktion $\mathbf{f}$ wird definiert als der kleinste Wert b, so dass alle Komponentenfunktionen $f_j(t, \mathbf{y}(t))$ nur auf die Komponenten y_{j-b} bis y_{j+b} des Argumentvektors zugreifen. Als beschränkt wird die Zugriffsdistanz dann bezeichnet, wenn $d(\mathbf{f}) \ll n$ gilt, wobei n die Systemgröße des DGL ist.

1.2 Iterierte Runge-Kutta-Verfahren

Zur Lösung solcher DGL gibt es bereits einige Techniken, wie zum Beispiel Waveform-Relaxationsverfahren oder Runge-Kutta-Verfahren. Einen guten Überblick über verschiedene Methoden bietet [7]. Den Kern dieser Arbeit bilden iterierte Runge-Kutta (IRK)-Verfahren, welche zur Familie der Prädiktor-Korrektor-Verfahren gehören. Iterierte Runge-Kutta-Verfahren basieren auf klassischen s-stufigen, impliziten Runge-Kutta-Verfahren und gehören zur Klasse der Einschrittverfahren: in jedem Zeitschritt κ wird eine neue Approximation $\boldsymbol{\eta}_{\kappa+1}$ direkt aus der vorhergehenden Näherung $\boldsymbol{\eta}_\kappa$ berechnet. Das oftmals nichtlineare Gleichungssystem, welches aus dem impliziten Verfahren hervorgeht, wird durch einen iterativen Prozess gelöst. Als erste Näherung (Prädiktor) für diesen iterativen Prozess wird hier der "triviale" Prädiktor verwendet:

$$\mathbf{Y}_l^{(0)} = \boldsymbol{\eta}_\kappa, \quad l = 1, ..., s, \quad \boldsymbol{\eta}_0 = \mathbf{y}_0 \tag{1}$$

Der Iterationsprozess wird fortgesetzt durch eine feste Anzahl $m = ord - 1$ Korrektorschritte, wobei ord die Ordnung des Basisverfahren darstellt:

$$\mathbf{Y}_l^{(k)} = \boldsymbol{\eta}_\kappa + h_\kappa \sum_{i=1}^{s} a_{li}\, \mathbf{f}(t_\kappa + c_i h_\kappa, \mathbf{Y}_i^{(k-1)}), \quad l = 1, ..., s, \;\; k = 1, ..., m \tag{2}$$

Nach dem Iterationsprozess werden zwei neue Näherungen $\boldsymbol{\eta}_{\kappa+1}$ und $\hat{\boldsymbol{\eta}}_{\kappa+1}$ berechnet, welche sich in ihrer Ordnung unterscheiden:

$$\boldsymbol{\eta}_{\kappa+1} = \boldsymbol{\eta}_\kappa + h_\kappa \sum_{i=1}^{s} b_i\, \mathbf{f}(t_\kappa + c_i h_\kappa, \mathbf{Y}_i^{(m)}) \quad \text{und} \quad \hat{\boldsymbol{\eta}}_{\kappa+1} = \boldsymbol{\eta}_\kappa + h_\kappa \sum_{i=1}^{s} b_i\, \mathbf{f}(t_\kappa + c_i h_\kappa, \mathbf{Y}_i^{(m-1)}) \tag{3}$$

Die Koeffzientenmatrix $\mathbf{A} = (a_{li} \in \mathbb{R}^{s \times s})$, der Wichtungsvektor $\mathbf{b} = (b_i)$, der Vektor $\mathbf{c} = (c_i)$ und die Anzahl s der Stufen sind durch das verwendete Basisverfahren gegeben. h_κ ist die im Zeitschritt κ verwendete Schrittweite.

1.3 Motivation

Für Einschrittverfahren ist keine Parallelisierung bezüglich der Zeitschritte möglich, allerdings lassen sich die Korrektorschritte der IRK-Verfahren innerhalb eines Zeitschrittes sowohl bezüglich des Systems (verschiedene Komponenten des DGL werden parallel behandelt) als auch bezüglich der Methode (unabhängige Berechnungen innerhalb eines Zeitschrittes werden parallel ausgeführt) oder einer Kombination aus beidem parallelisieren.

Ist das zu lösende Probleme dichtbesetzt, muss jeder an der Rechnung beteiligte Prozessor, unabhängig von der Art der Parallelisierung, die gesamte Matrix $\mathbf{Y}^{(k-1)}$ kennen, um den Iterationsschritt aus Gleichung 2 durchführen zu können. Aus diesem Grund verwenden allgemeine Implementierungen von IRK-Verfahren für verteilten Adressraum Multibroadcastoperationen, um die einzelnen Vektoren untereinander auszutauschen. Multibroadcastoperationen wie zum Beispiel *MPI_Allgatherv* sind allerdings langsam und weisen eine schlechte Skalierbarkeit auf (vergleiche [11]).

Für dünnbesetzte Probleme reicht es aus, wenn jeder Prozessor nur die Elemente von $\mathbf{Y}^{(k-1)}$ kennt, die er tatsächlich für die Berechnung seiner Elemente der Matrix $\mathbf{Y}^{(k)}$ benötigt. Bei einer Parallelisierung ausschließlich bezüglich der Methode macht das jedoch keinen Unterschied zu einem dichtbesetzten Problem: jeder Prozessor

braucht wiederum die gesamte Matrix $\mathbf{Y}^{(k-1)}$. Bei einer Parallelisierung bezüglich des Systems hingegen kann die Anzahl der benötigten Komponenten deutlich geringer sein.

Dies kann dazu führen, dass sowohl die Anzahl als auch die Größe der zu verschickenden Nachrichten drastisch abnimmt. Ein auf dünnbesetzte DGL spezialisierter Löser kann somit Multibroadcastoperationen vermeiden und diese durch günstigere Einzeltransferoperationen ersetzen.

1.4 Zielsetzung und Gliederung

Ziel dieser Arbeit ist es, die Kommunikation von parallelen Implementierungen iterierter Runge-Kutta-Verfahren für verteilten Adressraum durch Ersetzung von teuren Multibroadcastoperationen durch günstigere Einzeltransferoperationen für dünnbesetzte DGL zu optimieren.

Hierfür werden zunächst bereits vorhandene parallele Implementierungen von IRK-Verfahren für gemeinsamen Adressraum auf MPI übertragen. Dabei wird darauf geachtet, dass sowohl die Parallelisierung an sich als auch Schleifen- und Datenstrukturen erhalten bleiben.

Daraufhin werden einige Überlegungen angestellt, auf welche Weise man die Kommunikation der neu entstandenen Implementierungen für dünnbesetzte Probleme verbessern kann. Die Ergebnisse hiervon werden auf die Implementierungen angewandt.

Danach werden die für dünnbesetzte DGL optimierten Implementierungen anhand eines dünnbesetzten Testproblems mit den dazugehörigen allgemeinen Implementierungsvarianten verglichen. Zu diesem Zweck werden die Ergebnisse von Laufzeitmessungen auf mehreren Rechnersystemen ausgewertet.

Abschließend werden die Ergebnisse dieser Arbeit noch einmal kompakt zusammengefasst und ein Ausblick auf mögliche künftige Forschungsziele gestellt.

2 Parallele Implementierungen von IRK-Verfahren

2.1 Implementierungen für gemeinsamen Adressraum

IRK-Verfahren sind schon seit einiger Zeit Forschungsgegenstand an der Universität Bayreuth. Daher lagen schon zu Beginn dieser Arbeit verschiedene Implementierungsvarianten von IRK-Verfahren für gemeinsamen Adressraum vor. Die Implementierungen differieren in den verwendeten Schleifen- und Datenstrukturen und tragen zur Unterscheidung verschiedene Bezeichner. Einen Überblick darüber kann man unter anderem in [8] und [10] gewinnen. Alle im Rahmen dieser Arbeit behandelten Implementierungsvarianten parallelisieren ausschließlich bezüglich des Systems und verteilen die Komponenten blockweise (vergleiche [9, Kapitel 3.2]).

Ein Großteil der Implementierungen stammt von der Variante mit der Bezeichnung D ab. Diese Familie speichert in jedem Iterationsschritt k aus Gleichung 2 die Matrizen $\mathbf{Y}^{(k)}$ und $\mathbf{Y}^{(k-1)}$. Alle Threads T_i teilen sich den Zugriff auf diese und berechnen in einem Korrektorschritt die ihnen von der Verteilung zugewiesenen Komponenten $first_i$ bis $last_i$. Nach einem Korrektorschritt synchronisieren die Threads mittels einer Barrier, bevor der Inhalt der Matrizen vertauscht wird. Durch die Barrier wird sichergestellt, dass im nächsten Korrektorschritt alle Threads die aktuellen Werte von $\mathbf{Y}^{(k-1)}$ kennen. Die neuen Näherungen $\boldsymbol{\eta}_{\kappa+1}$ und $\hat{\boldsymbol{\eta}}_{\kappa+1}$, die dem gemeinsamen Zugriff aller Threads unterliegen, werden ähnlich wie die Matrizen aus den Iterationsschritten komponentenweise parallel berechnet. Jeder Thread T_i berechnet aus den beiden Approximationen einen lokalen Fehler für sein Intervall $[first_i, last_i]$. Mittels einer Akkumulationsoperation *reduction_max*, welche intern auch Barriers verwendet, bestimmen die Threads den größten aufgetretenen Fehler ε_{max}. Dieser wird für eine Schrittweitenkontrolle mit der Maximumsnorm verwendet. Überschreitet ε_{max} einen vorgegebenen Maximalwert, wird der aktuelle Zeitschritt verworfen. In diesem Fall wird $\boldsymbol{\eta}_\kappa$ auf den Wert zurückgesetzt, den es vor dem gerade durchgeführten Zeitschritt hatte, eine neue Schrittweite berechnet und der Zeitschritt wiederholt. Das Zurücksetzen von $\boldsymbol{\eta}_\kappa$ geschieht wieder parallel: Jeder Thread T_i ersetzt die Komponenten in seinem Intervall $[first_i, last_i]$. Dieser Vorgang wird ebenfalls durch eine Barrier synchronisiert. Im Fall, dass der Zeitschritt akzeptiert wird, ε_{max} also unter dem vorgegebenen Maximalwert liegt, wird eine neue Schrittweite bestimmt und der nächste Zeitschritt berechnet, bis das Integrationsintervallende t_e erreicht wird.

Andere Implementierungsvarianten, welche nicht von D abstammen, wie etwa die Implementierungen mit den Bezeichnungen A und E speichern anstatt der Matrizen $\mathbf{Y}^{(k)}$ und $\mathbf{Y}^{(k-1)}$ die Ergebnisse der Funktionsauswertungen $\mathbf{F}^{(k)}$ und $\mathbf{F}^{(k-1)}$. Dies hat jedoch keinen Einfluss auf die Art der Synchronisation innerhalb der Korrektorschritte. Die Berechnung von ε_{max} und die Schrittweitenkontrolle geschieht auf die gleiche Weise wie bei Implementierung D.

Neben der Umsetzung der Berechnungsvorschrift und einer Schrittweitenkontrolle haben alle Implementierungen auch noch Funktionen zur Erhebung von statistischen Daten wie einem Zähler der Zeitschritte und Performancecounter, welche optional genutzt werden können. Ein besonderes Feature, welches für Messungen interessant ist, ist das Setzen von Abbruchkriterien. Um die Performance von Implementierungen zu messen, ist es meistens nicht nötig, eine vollständige Lösung des AWP zu berechnen. Deswegen wird die Möglichkeit geboten, die Berechnungen nach einer bestimmten Anzahl von Zeitschritten oder einer vorgegebenen Laufzeit zu beenden. Die hierfür verantwortlichen Methoden haben die Namen *simics_pth_check_and_stop* und *cws_pth_stop*. In der erstgenannten

Funktion bestimmt jeder Thread für sich, ob die vom Nutzer angegebenen Abbruchkriterien erreicht sind. Ist das der Fall, so wird eine boolesche Variable *simics_stop_threads*, die dem gemeinsamen Zugriff aller Threads unterliegt und mit 0 initialisiert wurde, auf 1 gesetzt. Unabhängig davon, ob die Abbruchkriterien erreicht sind, werden die Threads am Ende von *simics_pth_check_and_stop* mit Hilfe einer Barrier synchronisiert. In der zweiten Funktion beenden die Threads die Rechnungen, wenn *simics_stop_threads* wahr (=1) ist.

2.2 Implementierungen für verteilten Adressraum

Viele in der Praxis genutzte parallele Rechnersysteme haben einen verteilten Adressraum. Das liegt daran, dass diese im Vergleich zu Systemen mit gemeinsamen Adressraum eine einfachere Hardwarearchitektur besitzen. Dadurch lassen sich einerseits Clustersysteme aus kostengünstigen Standardkomponenten, aber andererseits auch technisch ausgefeilte parallele Rechnersysteme, wie *SuperMUC*, konstruieren. *SuperMUC* ist der derzeit leistungsfähigste deutsche Supercomputer (vergleiche [12]). Dieser befindet sich am Leibniz-Rechenzentrum in München, ist mit mehr als 155.000 Kernen ausgestattet (vergleiche [2]) und hat einen verteilten Adressraum.

Auf Rechnersystemen mit verteiltem Adressraum geschieht die Synchronisation und Kommunikation von Kontrollflüssen mittels Nachrichtenaustausches (englisch: *message passing*). Zur Realisierung des Nachrichtenaustausches wird dem Programmierer auf den meisten Parallelrechnern eine Anwendungsschnittstelle angeboten, die sich am MPI-Standard (für englisch: *message passing interface*, siehe [5]) orientiert. Da der MPI-Standard weder Protokoll noch Implementierung festlegt, wird der Funktionsumfang in Form einer Programmbibliothek zur Verfügung gestellt. Neben herstellerspezifischen Implementierungen, die überwiegend in großen Rechenzentren zum Einsatz kommen, gibt es auch einige Open-Source-Projekte zur Entwicklung von MPI-Implementierungen, wie zum Beispiel MPICH2 oder Open MPI (siehe [3] und [4]).

Möchte man die Implementierungen von IRK-Verfahren für gemeinsamen Adressraum, die im vorigen Kapitel beschrieben wurden, für Parallelrechner mit verteiltem Adressraum mit Methoden aus dem MPI-Standard umsetzen, werden an mehreren Stellen Datenaustäusche erforderlich.

In Implementierungen für gemeinsamen Adressraum, die von der Variante D abstammen, hat jeder Thread direkten Zugriff auf die Matrizen $\mathbf{Y}^{(k)}$ und $\mathbf{Y}^{(k-1)}$, die im globalen Speicher liegen. Da es im Modell eines verteilten Adressraumes keine gemeinsamen Variablen gibt, besitzt dort jeder Prozess P_i eine lokale Kopie. In einem Korrektorschritt aus Gleichung 2 berechnet Prozess P_i, analog zu Implementierungen für gemeinsamen Adressraum, die Komponenten im Intervall $[first_i, last_i]$ gemäß einer blockweisen Verteilung. Anschließend müssen die Prozesse die Zeilen von $\mathbf{Y}^{(k)}$ mittels s Multibroadcastoperationen austauschen. Der MPI-Standard sieht hierfür die Funktionen *MPI_Allgather*, im Falle von einheitlichen Nachrichtengrößen, und *MPI_Allgatherv*, im Falle von unterschiedlichen Nachrichtengrößen, vor. Die Nachrichtengröße entspricht hier der Anzahl der Komponenten, die Prozess P_i berechnet. Da diese in einer blockweisen Verteilung zwischen den Prozessen um 1 variieren kann, wird für die Implementierung die Funktion *MPI_Allgatherv* gewählt. Diese Methode sammelt die Daten, die in einer Prozessgruppe verteilt sind, und verteilt das Resultat an alle Prozesse in der Gruppe. Da die Funktion blockierend arbeitet, ist sichergestellt, dass nach deren Rückkehr alle Prozesse P_i den aktuellen Inhalt der Matrix $\mathbf{Y}^{(k)}$ kennen und diese dem Nutzer wieder zur Verfügung steht.

Wie in den Implementierungen für gemeinsamen Adressraum werden nach dem Iterationsprozess die neuen Näherungen $\boldsymbol{\eta}_{\kappa+1}$ und $\hat{\boldsymbol{\eta}}_{\kappa+1}$ komponentenweise parallel berechnet. Jeder Prozess P_i errechnet mit diesen einen lokalen Fehler für das Intervall $[first_i, last_i]$. Zur Ermittlung des größten aufgetretenen Fehlers ε_{max} ist eine Multi-Akkumulationsanweisung erforderlich. Der MPI-Standard bietet hierfür die Funktion *MPI_Allreduce*. Diese Funktion erwartet als Parameter unter anderem eine Reduktionsoperation. An dieser Stelle wird die im MPI-Standard bereits vordefinierte Methode *MPI_MAX*, die das Maximum der Daten ermittelt, verwendet. Wird der aktuelle Zeitschritt aufgrund eines zu großen Fehlers ε_{max} von der Schrittweitenkontrolle verworfen, setzen die Prozesse P_i den aktuellen Approximationsvektor auf den Wert zurück, den es vor dem Zeitschritt hatte, indem sie die Komponenten im Intervall $[first_i, last_i]$ wiederherstellen. Anschließend wird eine neue Schrittweite festgelegt und der Zeitschritt wiederholt, ohne dass ein weiterer Nachrichtenaustausch nötig ist. Wird der aktuelle Zeitschritt akzeptiert, tauschen die Prozesse die neue Approximation $\boldsymbol{\eta}_{\kappa+1}$ mittels einer Multibroadcastoperation (*MPI_Allgatherv*) aus, bestimmen eine neue Schrittweite und setzen die Berechnungen mit dem nächsten Zeitschritt fort, bis das Integrationsintervallende t_e erreicht ist.

Die Implementierung der Methoden *simics_pth_check_and_stop* und *cws_pth_stop* für verteilten Adressraum erfordert einen weiteren Datenaustausch. Jeder Prozess P_i stellt für sich fest, ob die vom Nutzer vorgegebenen Abbruchkriterien erfüllt sind und speichert das Ergebnis in einer booleschen Variable *simics_mpi_stop_condition*. Mittels einer Multi-Akkumulationsanweisung (*MPI_Allreduce*) mit einer geeigneten Reduktionsoperation, wie etwa *MPI_MIN* oder *MPI_MAX*, werden die Prozesse synchronisiert. Nach der Multi-Akkumulationsoperation steht für alle Prozesse eindeutig fest, ob die Rechnungen beendet werden sollen. Ein zusätzlicher Datenaustausch stellt einen nicht unerheblichen Mehraufwand dar, den man in der Regel vermeiden möchte. Aus diesem Grund werden die beiden Multi-Akkumulationsoperationen zur Bestimmung des Fehler ε_{max} und des Abbruchflags *simics_mpi_stop_condition* zu einer Multi-Akkumulation zusammengefasst. Hierfür wird eine Variable vom Da-

tentyp *MPI_DOUBLE_INT* angelegt, welche als erste Komponente den lokalen Fehler und als zweite Komponente *simics_mpi_stop_condition* enthält. Als Reduktionsoperation wird *MPI_MAXLOC* verwendet. Diese bewirkt, dass die Prozesse die Rechnungen dann beenden, wenn der Prozess mit dem größten lokalen Fehler die Abbruchkriterien als erfüllt erachtet. Eine genaue Beschreibung der Operation *MPI_MAXLOC* kann man in [13, Kapitel 4.1] finden.

3 Reduktion des Kommunikationsaufwandes für dünnbesetzte DGL

3.1 Vorbetrachtungen

Die im vorigen Kapitel beschriebenen Implementierungen iterierter Runge-Kutta-Verfahren für verteilten Adressraum verwenden in jedem Zeitschritt mehrere Multibroadcastoperationen um die Argumentvektoren $\mathbf{y}$ für die Funktionsauswertungen f_j auszutauschen. Multibroadcastoperationen wie *MPI_Allgatherv* sind in der Praxis sehr zeitaufwändig und skalieren schlecht (vergleiche [11]), lassen sich aber für dichtbesetzte DGL nicht vermeiden. Für dünnbesetzte DGL gilt, dass zur Auswertung der Funktionen f_j nur wenige Komponenten des Argumentvektors $\mathbf{y}$ benötigt werden. In diesem Fall kann es von Vorteil sein, wenn die Prozesse nur die tatsächlich benötigten Komponenten mittels Einzeltransferoperationen austauschen. Hierfür werden die im MPI-Standard definierten Methoden *MPI_Isend* und *MPI_Irecv* verwendet. Erstere führt eine nichtblockierende Sende- und zweitere eine nichtblockierende Empfangsoperation aus. Nichtblockierend bedeutet, dass die Sende-/Empfangsoperation gestartet wird, ohne sicherzustellen, dass die Daten nach Abschluss der Operation schon übertragen wurden. Beide Methoden erwarten als Parameter ein Objekt vom Typ *MPI_Request*, das zur Identifikation der durchgeführten Operation dient. Um das Ende der Datenübertragung abzuwarten, kann man das *MPI_Request*-Objekt an die Methode *MPI_Wait* übergeben. Diese blockiert den ausführenden Prozess bis die zugehörige Operation vollständig abgeschlossen ist.

Um die für die Funktionsauswertungen f_j benötigten Daten gezielt austauschen zu können, muss jeder Prozess feststellen, an welche anderen Prozesse er welche Daten senden muss und von welchen Prozessen er welche Daten empfangen wird. Dies soll zunächst aus einer mathematischen Perspektive betrachtet werden:

Die Elemente des Intervalls $[first_i, last_i]$ bilden die Menge $\mathcal{P}_i = \{x \in \mathbb{N}_0 \mid first_i \leq x \leq last_i\}$. Weiterhin seien $\mathcal{F}_j$ die Indexmenge der zur Auswertung von f_j benötigten Komponenten des Argumentvektors $\mathbf{y}$, so dass $\mathcal{F}_j = \{x \in \mathbb{N}_0 \mid f_j \text{ benötigt } y_x\}$ gilt, und *processes* die Anzahl der MPI-Prozesse.

Dann bestimmt der Prozess P_{me} an welche Prozesse er welche Komponenten senden muss, indem er für jeden Prozess P_i $(i \neq me)$ die Indexmenge $\mathcal{F}^{(i)} = \bigcup_{j \in \mathcal{P}_i} \mathcal{F}_j$ aller von diesem benötigten Komponenten ermittelt. Die Schnittmenge $\mathcal{S}_i = \mathcal{F}^{(i)} \cap \mathcal{P}_{me}$ beinhaltet dann die Indizes aller Komponenten, die Prozess P_{me} an Prozess P_i senden muss. In Listing 1 wird das Vorgehen in Pseudocode dargestellt.

Das Bestimmen der zu empfangenden Komponenten geschieht auf ähnliche Weise. Der Prozess P_{me} ermittelt die Indexmenge $\mathcal{F}^{(me)} = \bigcup_{j \in \mathcal{P}_{me}} \mathcal{F}_j$ der von ihm benötigten Komponenten. Anschließend wird für jeden Prozess P_i $(i \neq me)$ die Schnittmenge $\mathcal{R}_i = \mathcal{F}^{(me)} \cap \mathcal{P}_i$ gebildet, die die Indizes der von Prozess P_i zu empfangenden Komponenten enthält. Dieser Vorgang wird in Listing 2 beschrieben.

```
1 for i in processes:
2     F^(i) = ⋃_{j∈P_i} F_j
3     S_i = F^(i) ∩ P_me
```

Listing 1: Bestimmen der zu versendenden Komponenten

```
1 F^(me) = ⋃_{j∈P_me} F_j
2 for i in processes:
3     R_i = F^(me) ∩ P_i
```

Listing 2: Bestimmen der zu empfangenden Komponenten

3.2 Implementierung

Um den Kommunikationsaufwand der allgemeinen Implementierungen für verteilten Adressraum für dünnbesetzte DGL zu reduzieren, müssen einige Änderungen am Quellcode vorgenommen werden.

Jeder Prozess muss in der Lage sein, eingehende und ausgehende Nachrichten unabhängig voneinander zu verwalten. Die Anzahl der Nachrichten steht im Vorfeld nicht fest, weshalb dynamische Datenstrukturen von Vorteil sind. Aus diesem Grund verwenden die Implementierungen für dünnbesetzte DGL zwei einfach verkettete Listen *inbox* und *outbox* um zu empfangende und zu versendende Nachrichten zu speichern. Eine Nachricht besteht hierbei aus den von *MPI_Isend*/*MPI_Irecv* benötigten Parametern und einer Indexmenge, die als Feld von Ganzzahlen namens *template* abgelegt ist. Die Indexmenge entspricht $\mathcal{S}_i$ im Falle einer ausgehenden und $\mathcal{R}_i$ im Falle einer eingehenden Nachricht. Das Listing 4 zeigt den Aufbau der implementierten Struktur *Message*.

Zum Erstellen der Nachrichten werden die Indexmengen $\mathcal{F}_j$ benötigt. Weil diese vom DGL abhängen, wurde für jedes Problem die Methode *ode_get_required_indices_for_component(int j)* implementiert. Die Methode liefert die Indizes der zur Auswertung von f_j benötigten Komponenten und deren Anzahl zurück. Hierfür wurde eine Datenstruktur *Set* umgesetzt, deren Aufbau in Listing 5 zu sehen ist. Da alle in dieser Arbeit betrachteten

Implementierungen eine blockweise Verteilung verwenden, wurde zusätzlich noch die Methode *ode_get_required_indices_for_intervall(int first, int last)* realisiert. Diese gibt die Indizes aller Komponenten, die zur Auswertung von f_{first} bis f_{last} benötigt werden, und deren Anzahl als *Set* zurück. Diese Methode hat den Vorteil, dass man die Vereinigung von Mengen für spezielle Probleme vermeiden und durch performanteren Code ersetzen kann. So müssen für Probleme mit beschränkter Zugriffsdistanz nicht die einzelnen Indexmengen $\mathcal{F}_{first}$ bis $\mathcal{F}_{last}$ bestimmt und vereinigt werden. Stattdessen kann man ohne großen Rechenaufwand die Menge $\{x \in \mathbb{N}_0 \mid first - d(\mathbf{f}) \leq x \leq last + d(\mathbf{f})\}$ zurückgeben.

Um die zu versendenden Nachrichten zu erstellen, iteriert Prozess P_{me} über die Prozesse P_i und ruft für jeden die Methode *ode_get_required_indices_for_intervall* mit den entsprechenden Intervallgrenzen auf. Anschließend wird die Schnittmenge $\mathcal{S}_i$ gebildet. Ist deren Kardinalität größer als 0, so wird eine neue Nachricht angelegt und in *outbox* gespeichert.

Danach werden die zu empfangenden Nachrichten erzeugt. Der Prozess P_{me} ruft dafür die Methode *ode_get_required_indices_for_intervall* mit seinen Intervallgrenzen auf um die von ihm benötigten Indizes zu bestimmen. Im Anschluss iteriert Prozess P_{me} über die Prozesse P_i und bildet die Schnittmenge $\mathcal{R}_i$. Wenn deren Mächtigkeit größer als 0 ist, wird eine neue Nachricht erstellt und in *inbox* abgelegt.

Für diese Arbeit wird angenommen, dass alle F_j konstant sind. Dadurch reicht es aus, wenn man die Nachrichten einmal am Beginn des Programmes erstellt, da bei jedem Datenaustausch die selben Komponenten benötigt werden.

Das Erstellen der Nachrichten wird in Listing 3 noch einmal ausführlicher dargelegt:

```
Message inbox  = NULL
Message outbox = NULL

for i in processes:
  if i == me:                          # fill inbox
    Set required = ode_get_required_indices_for_intervall(first_me, last_me)
    for p in processes:
      Set r = required ∩ {first_p, ..., last_p}
      if r.size > 0:
        Message m  = new Message
        m.receiver = me
        m.sender   = p
        m.template = r.elements
        m.size     = r.size
        m.buffer   = double[m.size]
        m.next     = NULL
        inbox.addMessage(m)
  else:                                # fill outbox
    Set required = ode_get_required_indices_for_intervall(first_i, last_i
    Set s = required ∩ {first_me, ..., last_me}
    if s.size > 0:
      Message m  = new Message
      m.sender   = me
      m.receiver = i
      m.template = s.elements
      m.size     = s.size
      m.buffer   = double[m.size]
      m.next     = NULL
      outbox.addMessage(m)
```

Listing 3: Das Zusammenstellen der Nachrichten in Pseudocode

```
struct Message:
  int            sender
  int            receiver
  int            size
  int[size]      template
  double[size]   buffer
  MPI_Request    request
  Message        next
```

Listing 4: Aufbau der Struktur *Message* in Pseudocode

```
struct Set:
  int            size
  int[size]      elements
```

Listing 5: Aufbau der Struktur *Set* in Pseudocode

Sind die Nachrichten erstellt, müssen diese nur noch mittels Einzeltransferoperationen versandt werden. Hierfür sind sechs Schritte notwendig, die anstelle eines *MPI_Allgatherv*-Aufrufes ausgeführt werden:

1. Empfangsoperationen starten
2. Vorbereiten ausgehender Nachrichten
3. Sendeoperationen starten
4. Auf Abschluss der Sendeoperationen warten
5. Auf Abschluss der Empfangsoperationen warten
6. Eingegangene Nachrichten entpacken

Zuerst ruft jeder Prozess die Funktion *MPI_Irecv* für die Nachrichten in dessen *inbox* auf, um die Empfangsoperationen zu starten. Im zweiten Schritt werden für alle Nachrichten in *outbox* die Komponenten des Argumentvektors $\mathbf{y}$, deren Indizes in *template* gespeichert sind, in den Sendepuffer *buffer* kopiert. Nachdem der Sendepuffer gefüllt ist, kann die Funktion *MPI_Isend* aufgerufen werden. Anschließend wird auf das Ende der Sende- und Empfangsoperationen mit *MPI_Wait* gewartet. Hat ein Prozess alle von ihm benötigten Daten empfangen, so liegen diese noch in den Empfangspuffern *buffer* der Nachrichten in *inbox*. Diese müssen nun noch an die Stellen des Argumentvektors $\mathbf{y}$, die in *template* spezifiziert sind, kopiert werden.

Um den Quellcode möglichst übersichtlich zu halten, wurde die Ausführung der oben genannten Schritte in eine Methode namens *sparse_gather* ausgelagert. Die Methode erwartet als Parameter zwei Nachrichtenlisten (*inbox* und *outbox*) und einen Zeiger auf den Argumentvektor $\mathbf{y}$. Nachfolgendes Listing zeigt deren Aufbau in Pseudocode:

```
sparse_gather(Message in, Message out, double[] y):
  # Empfangsoperationen starten
  while(in):
    MPI_Irecv(in.buffer, in.size, MPI_DOUBLE, in.sender,
              MPI_ANY_TAG, MPI_COMM_WORLD, in.request)
    in = in.next

  # Ausgehende Nachrichten vorbereiten und versenden
  while(out):
    for i in out.size:
      out.buffer[i] = y[out.template[i]]

    MPI_Isend(out.buffer, out.size, MPI_DOUBLE, out.receiver,
              0, MPI_COMM_WORLD, out.request)

    out = out.next

  # Auf Ende der Sendeoperationen warten
  while(out):
    MPI_Status mpi_status
    MPI_Wait(out.request, mpi_status)
    out = out.next;

  # Auf Ende der Empfangsoperationen warten und
  # empfangene Komponenten in den Argumentvektor kopieren
  while(in):
    MPI_Status mpi_status
    MPI_Wait(in.request, mpi_status)

    for i in in.size:
      y[in.template[i]] = in.buffer[i]

    in = in->next;
```

Listing 6: Die Methode sparse_gather in Pseudocode

Die oben beschriebene Vorgehensweise sogt dafür, dass jeder Prozess nur die Komponenten des Argumentvektors $\mathbf{y}$ kennt, die er für die Funktionsauswertungen f tatsächlich benötigt. Das führt allerdings dazu, dass die Lösung des AWP am Ende der Berechnungen verteilt im Speicher liegt. Aus diesem Grund wird der Lösungsvektor mit einer Gatheroperation eingesammelt. Als Gatheroperation wird die im MPI-Standard vorgeschriebene Methode *MPI_Gatherv* verwendet. Diese erwartet als Parameter unter anderem den Rang des empfangenden Prozesses.

An dieser Stelle wird 0 angegeben, so dass die Lösung des AWP am Ende der Berechnungen dem ersten Prozess vollständig bekannt ist.

Implementierungen, welche die in diesem Kapitel beschriebene Art der Kommunikation verwenden, werden mit einem 'S' am Ende der ursprünglichen Bezeichnung markiert. So entspricht die Variante *DS* der Implementierung *D* in Bezug auf die verwendeten Daten- und Schleifenstrukturen. Allerdings ist die Kommunikation von *DS* für dünnbesetzte DGL optimiert.

3.3 Optimierung der Kommunikation für Probleme mit beschränkter Zugriffsdistanz

Probleme mit beschränkter Zugriffsdistanz sind ein Spezialfall von dünnbesetzten DGL, bei denen zur Auswertung von $f_j(t, \mathbf{y}(t))$ nur die Komponenten $y_{j-d(\mathbf{f})}$ bis $y_{j+d(\mathbf{f})}$ nötig sind. Dabei ist die Zugriffsdistanz $d(\mathbf{f})$ der Funktion $\mathbf{f}$ wie in Kapitel 1.1 definiert. Hat ein Problem eine beschränkte Zugriffsdistanz, so kann man diese für eine verbesserte Kommunikation ausnutzen.

Bei einer blockweisen Verteilung benötigt der Prozess P_i für die Funktionsauswertungen die Komponenten im Intervall $[first_i - d(\mathbf{f}), last_i + d(\mathbf{f})]$. Die Komponenten im Intervall $[first_i, last_i]$ sind diesem bereits bekannt, so dass nur $[first_i - d(\mathbf{f}), first_i[$ und $]last_i, last_i + d(\mathbf{f})]$ von anderen Prozessen empfangen werden müssen. Ist $d(\mathbf{f})$ kleiner als die Blockgröße der Verteilung, so befinden sich die Komponenten $[first_i - d(\mathbf{f}), first_i[$ bei Prozess P_{i-1} und die Komponenten $]last_i, last_i + d(\mathbf{f})]$ bei Prozess P_{i+1}. Das führt dazu, dass nur Datenaustäusche mit benachbarten Prozessen erforderlich sind:

Der Prozess P_i $(i \neq 0)$ empfängt die Komponenten $first_i - d(\mathbf{f})$ bis $first_i - 1$ von Prozess P_{i-1} und sendet diesem die Komponenten $first_i$ bis $first_i + d(\mathbf{f}) - 1$. Ist $i \neq processes - 1$, so empfängt P_i weiterhin die Komponenten $last_i + 1$ bis $last_i + d(\mathbf{f})$ von Prozess P_{i+1} und sendet diesem die Komponenten $last_i - d(\mathbf{f}) + 1$ bis $last_i$.

Implementierungsvarianten, deren Kommunikation für Probleme mit beschränkter Zugriffsdistanz optimiert sind, führen anstelle von *sparse_gather* eine Methode namens *neighbour_exchange* aus, welche die oben erläuterte Nachbarschaftskommunikation realisiert. Die Methode erwartet als Parameter die Zugriffsdistanz $d(\mathbf{f})$ und den Argumentvektor $\mathbf{y}$. Der Aufbau und die Funktionsweise der Methode werden in Listing 7 und Abbildung 1 verdeutlicht.

Implementierungen, die diese Art der Nachbarschaftskommunikation nutzen, werden mit einem 'N' gekennzeichnet.

Die Methode *neighbour_exchange* funktioniert nur dann, wenn die Zugriffsdistanz $d(\mathbf{f})$ kleiner als die Blockgröße der Verteilung ist. Für Fälle, in denen das nicht garantiert ist, wurden weitere Implementierungen erstellt, die ähnlich wie die *neighbour_exchange*-Varianten nur mit Nachbarschaftskommunikation arbeiten. Der Unterschied besteht darin, dass Daten so lange an einen Nachbarn weiterversendet werden, bis sie am Zielprozess angekommen sind. Diese Implementierungen basieren auf der Arbeit von Carsten Scholtes und werden mit 'X' gekennzeichnet.

```
neighbour_exchange(int df, double[] y):
  MPI_Request send_prev, recv_prev
  MPI_Request send_next, recv_next
  MPI_Status status

  if(me > 0) :
      MPI_Irecv(y[first - df], df, MPI_DOUBLE, me-1,
                  MPI_ANY_TAG, MPI_COMM_WORLD, recv_prev)
      MPI_Isend(y[first      ], df, MPI_DOUBLE, me-1,
                0              , MPI_COMM_WORLD, send_prev)

  if(me < processes -1)
    MPI_Irecv(y[last+1      ], df, MPI_DOUBLE, me+1,
              MPI_ANY_TAG, MPI_COMM_WORLD, recv_next)
    MPI_Isend(y[last+1 - df], df, MPI_DOUBLE, me+1,
              0           , MPI_COMM_WORLD, send_next)

  if (me > 0)
    MPI_Wait(recv_prev, status)
    MPI_Wait(send_prev, status)

  if (me < processes -1) :
    MPI_Wait(recv_next, status)
    MPI_Wait(send_next, status)
```

Listing 7: Nachbarschaftskommunikation für Probleme mit beschränkter Zugriffsdistanz

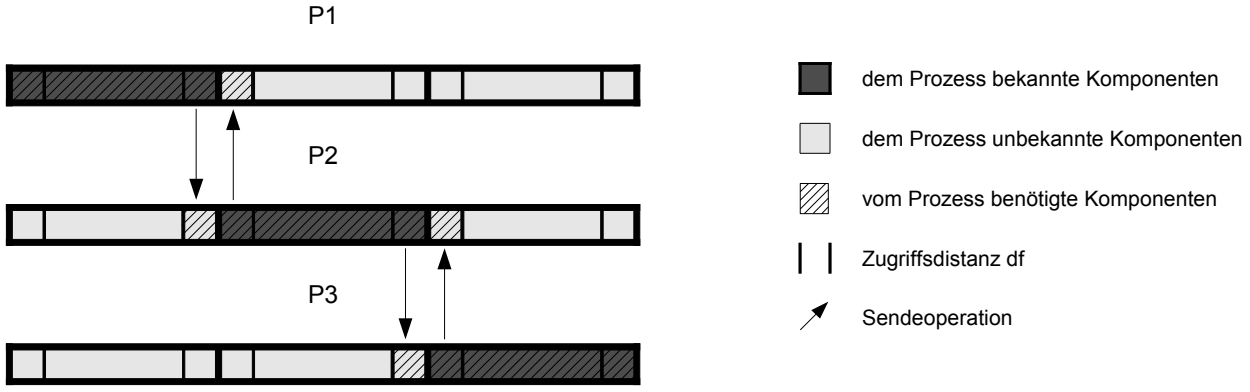


Abbildung 1: Nachbarschaftskommunikation für Probleme mit beschränkter Zugriffsdistanz

4 Experimentelle Untersuchung der Implementierungen

Um die Qualität der in dieser Arbeit vorgestellten Implementierungsvarianten zu untersuchen, wurden Laufzeitmessungen für unterschiedliche Testprobleme auf verschiedenen Rechnersystemen durchgeführt.

Die behandelten Implementierungen sind sehr zahlreich, weshalb in diesem Kapitel nur die Varianten *PipeDb2mt*, *PipeDb2mtS*, *PipeDb2mtN* und *PipeDb2mtX* betrachtet werden. Die Implementierung *PipeDb2mt* nutzt die selben Datenstrukturen wie *D*, verwendet aber im Gegensatz zu dieser eine andere Schleifenstruktur und Loop-Tiling um ein besseres Lokalitätsverhalten zu erreichen. Die übrigen Varianten basieren auf *PipeDb2mt* und machen Gebrauch von den in Kapitel 3 vorgestellten Techniken um den Kommunikationsaufwand für dünnbesetzte Probleme, bzw. Probleme mit beschränkter Zugriffsdistanz, zu minimieren.

Als Testprobleme kamen zwei Implementierungen des Brüsselatorproblems, BRUSS2D-ROW und BRUSS2D-MIX, zum Einsatz. Bei dem Brüsselatorproblem handelt es sich um eine zweidimensionale partielle Differentialgleichung, welche die chemische Reaktion zweier Substanzen in einem quadratischen Gebiet modelliert. Durch Diskretisierung mittels Linienmethode über einem $N \times N$-Gitter und den Neumann-Randbedingungen erhält man ein gewöhnliches Differentialgleichungssystem der Dimension $n = 2N^2$. Weitere Informationen zu diesem Problem sind in [9, Anhang B.1] zu finden. Die Implementierung BRUSS2D-ROW verwendet eine zeilenorientierte Linearisierung, was zu einem dünnbesetzten Problem ohne beschränkter Zugriffsdistanz führt. Im Gegensatz dazu verwendet BRUSS2D-MIX eine gemischt-zeilenorientierte Linearisierung, wodurch dieses Problem eine beschränkte Zugriffsdistanz hat.

Die DGL wurden für das Integrationsintervall $[0, 10]$ mit unterschiedlichen Systemgrößen gelöst. Dabei wurde die Fehlertoleranz, die im Programm die Bezeichnung *bf* trägt, auf 10^{-6} festgelegt. Alle anderen problemspezifischen Parameter entsprechen Defaultwerten.

Die Lösungen der AWP wurden mit zwei Verfahren, der 3-stufigen Radau IA (5) und der 5-stufigen Lobatto IIIC (8) Methode berechnet. Es ist zu erwarten, dass bei Messungen, die das Lobatto-Verfahren verwenden, größere Unterschiede in der Performance der Implementierungen festzustellen sind, da wegen der größeren Stufenzahl mehr Datenaustäusche pro Zeitschritt erfolgen.

Die Rechnersysteme haben die Bezeichnungen **Cluster**, **Hydra** und **Hydrus**. Eine ausführliche Beschreibung von diesen ist im Anhang A zu finden.

Auf jedem der Rechnersysteme wurde das Programm mit dem GNU C-Compiler *gcc* in der Version 4.7.1 kompiliert und mit der MPI-Implementierung OpenMPI in der Version 1.5.5 gelinkt.

Bei jeder Messung wurden die Kernelzeit t und die Anzahl der ausgeführten Zeitschritte erhoben. Die Kernelzeit ist die Zeit, die ein Programmlauf für die Schleife über die Zeitschritte benötigt.

Es gibt verschiedene Maße zur Bewertung paralleler Programme. In dieser Arbeit wird die Effizienz herangezogen. Die Effizienz eines parallelen Algorithmus ist definiert als

$$E_p(n) = \frac{T^*(n)}{p \cdot T_p(n)}$$

wobei $T^*(n)$ die Laufzeit des schnellsten sequentiellen Algorithmus, p die Anzahl der genutzen Prozesse und $T_p(n)$ die parallele Laufzeit ist (siehe [13, Kapitel 6]). Lässt sich ein Programm perfekt parallelisieren, so beträgt die optimale Effizienz $E_p(n) = 1$.

Es wurden Messungen für eine Vielzahl von unterschiedlichen Systemgrößen durchgeführt und ausgewertet. Dabei hat sich gezeigt, dass sich die Effizienzen der untersuchten Implementierungen für ausreichend große Systemgrößen nur wenig unterscheiden. Aus diesem Grund werden in diesem Kapitel nur Ergebnisse für die Systemgröße $n = 1,125 \cdot 10^6$ präsentiert. Weitere Systemgrößen sind auf der beiliegenden CD zu finden.

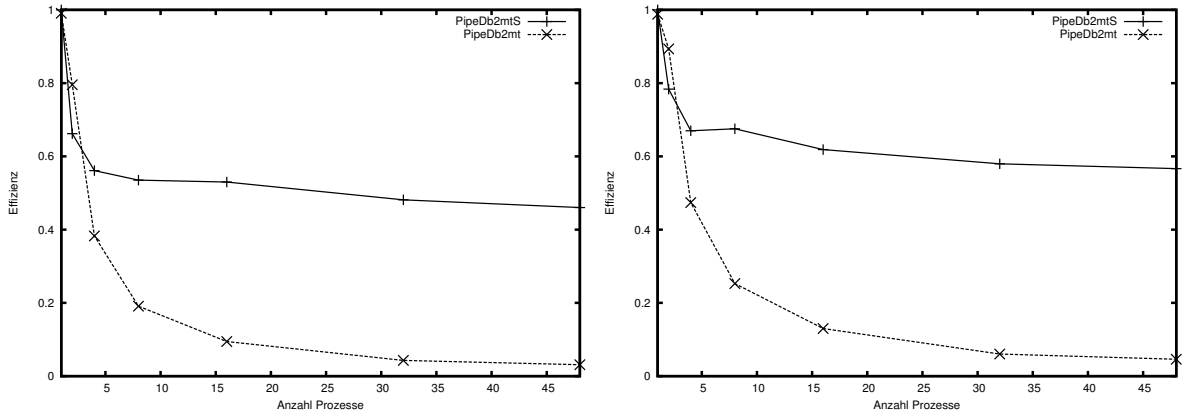
4.1 Effizienz der Implementierungen bei der Lösung dünnbesetzter Probleme

In diesem Kapitel werden die Effizienzen der Implementierungsvarianten *PipeDb2mt* und *PipeDb2mtS* bei der Lösung des Problems BRUSS2D-ROW verglichen. Dieses hat keine beschränkte Zugriffsdistanz, weshalb die restlichen Implementierungen nicht untersucht wurden. Hierbei sei erwähnt, dass die Ergebnisse für BRUSS2D-ROW nicht repräsentativ für alle dünnbesetzten Probleme sind, da diese unterschiedlich stark besetzt sein und ein anderes Zugriffsmuster der Funktionsauswertungen f_j aufweisen können.

4.1.1 Cluster

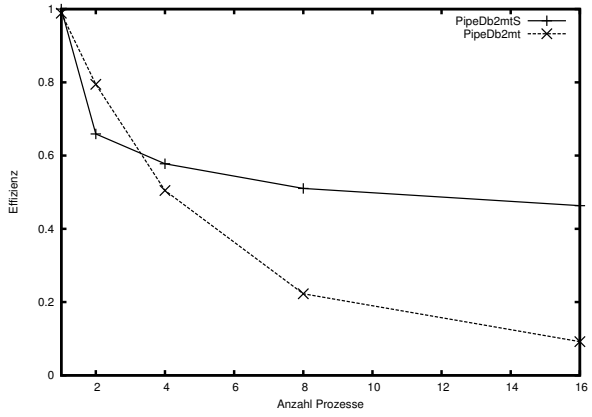
In der Abbildung 2a sind die Effizienzen der Implementierungen *PipeDb2mt* und *PipeDb2mtS* auf dem **Cluster** zu sehen, wobei die dazugehörigen Messungen mit dem Radau-Verfahren ausgeführt wurden.

Es ist gut zu erkennen, dass die Effizienz der Implementierung *PipeDb2mt* mit zunehmender Prozessanzahl etwa exponentiell abnimmt. Bei einer sequentiellen Ausführung ist *PipeDb2mt* fast so schnell wie *PipeDb2mtS*, aber schon bei 4 Prozessen beträgt die Effizienz nur noch knapp 40%. Bei 8 Prozessen ist die Effizienz dann auf ungefähr

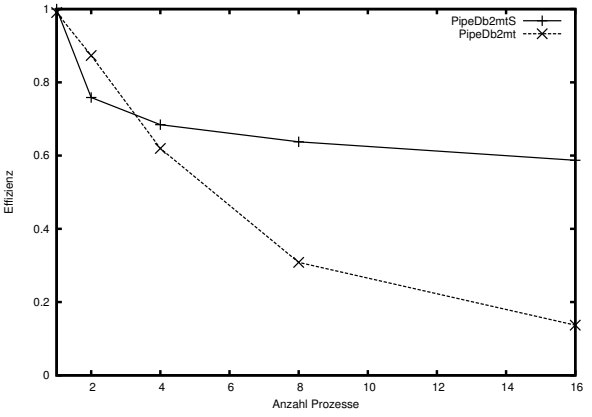


(a) Cluster - Radau IA (5)

(b) Cluster - Lobatto IIIC (8)



(c) Hydrus - Radau IA (5)



(d) Hydrus - Lobatto IIIC (8)

Abbildung 2: Effizienzen der Implementierungen *PipeDb2mt* und *PipeDb2mtS* bei der Lösung von BRUSS2D-ROW

20% gefallen und sinkt von da an stets weiter. Bei der maximalen Anzahl von 48 gestarteten Prozessen erreicht die Effizienz ihr Minimum von ca. 3%. Hier ist der Performancegewinn durch die Parallelisierung also fast nicht mehr vorhanden.

Im Gegensatz dazu fällt die Effizienz der Implementierung *PipeDb2mtS* nur einmal stark von 100% bei sequentieller Ausführung auf etwa 56% bei 4 Prozessen. Anschließend nimmt die Effizienz bei steigender Prozessanzahl nur sehr langsam ab. Selbst bei 48 Prozessen erreicht *PipeDb2mtS* noch eine Effizienz von mehr als 46%.

Die für dünnbesetzte Probleme optimierte Implementierung *PipeDb2mtS* ist für fast alle Prozessanzahlen effizienter als die Basisvariante *PipeDb2mt*. Die einzige Ausnahme bildet die Verwendung von zwei Prozessen. Hier ist *PipeDb2mt* mit einer Effizienz von 79,6% um mehr als 10% schneller als *PipeDb2mtS*. Der Grund hierfür liegt im speziellen Zugriffsmuster der Funktionsauswertungen f_j des Testproblems BRUSS2D-ROW. Zur Auswertung der Komponente j werden die Komponenten $j - N$, $j - 1$, j, $j + 1$, $j + N$ und $j + N^2$ wenn $j < N^2$, bzw. $j - N^2$ wenn $j > N^2$, benötigt. Da die Systemgröße des Brüsselatorproblems $n = 2N^2$ beträgt, liegt die Komponente mit dem Abstand N^2 von j stets in der anderen Hälfte des Argumentvektors $\mathbf{y}$ als die Komponente j. Das führt bei der Verwendung von zwei Prozessen dazu, dass diese alle Komponenten austauschen müssen. Die Implementierung der Kommunikation mit *MPI_Allgatherv* scheint also effizienter als die Realisierung mit Einzeltransferoperationen, wenn die zu übertragenden Datenmengen annähernd gleich groß sind. Diese Vermutung wird in Kapitel 4.3 noch einmal genauer untersucht.

In Abbildung 2b sieht man die Ergebnisse auf dem **Cluster** für das Lobatto-Verfahren. Qualitativ unterscheidet sich der Verlauf der Effizienz nicht wesentlich von dem Verlauf für das Radau-Verfahren. Allerdings sind die Effizienzen stets ein wenig besser, was daran liegt, dass das Lobatto-Verfahren 2 Stufen mehr als das Radau-Verfahren besitzt. Dadurch ist der Einfluss der Parallelisierung höher, was sich in besseren Effizienzen widerspiegelt. So erreicht die Implementierung *PipeDb2mt* eine Effizienz von 4,61% und die Variante *PipeDb2mtS* eine Effizienz von 56,7% bei 48 MPI-Prozessen.

4.1.2 Hydra

Für diese Messreihe ist beim Kompiliervorgang für das Rechnersystem *Hydra* leider ein Fehler unterlaufen. Die entstandenen Messergebnisse sind deshalb unter Umständen nicht aussagekräftig und werden ignoriert. Die anderen

Messreihen sind nicht von diesem Fehler betroffen.

4.1.3 Hydrus

Bei dem Rechnersystem **Hydrus** ist ein ähnliches Verhalten der Effizienz zu erkennen wie bei dem **Cluster**. Die Implementierung *PipeDb2mtS* ist für mehr als zwei verwendete Prozesse stets wesentlich effizienter als *PipeDb2mt*. Die Diagramme hierzu sind in den Abbildungen 2c und 2d zu finden. Bei der Betrachtung von diesen ist darauf zu achten, dass auf dem System **Hydrus** maximal 16 Prozesse gestartet wurden. Dadurch ist der Maßstab der Diagramme anders als bei denen vom **Cluster**, so dass die Effizienz der *MPI_Allgatherv*-Varianten höher scheint als sie ist.

4.1.4 Zwischenfazit für dünnbesetzte Probleme

Anhand der bisherigen Ergebnisse lässt sich sagen, dass die Optimierung der Kommunikation einen großen Einfluss auf die Kernel-Laufzeit von parallelen Implementierungen iterierter Runge-Kutta-Verfahren bei der Lösung von dünnbesetzten Problemen hat. Implementierungen, deren Kommunikation entsprechend angepasst ist, erreichen auch bei größeren Prozesszahlen Effizienzen von ungefähr 50%, während nicht optimierte Varianten kaum mehr einen Nutzen aus der parallelen Ausführung ziehen können.

4.2 Effizienz der Implementierungen bei der Lösung von Problemen mit beschränkter Zugriffsdistanz

Probleme mit beschränkter Zugriffsdistanz lassen sich von allen vier Implementierungen *PipeDb2mt*, *PipeDb2mtS*, *PipeDb2mtN* und *PipeDb2mtX* lösen. *PipeDb2mt* ist eine allgemeine Implementierungsvariante, die *MPI_Allgatherv* verwendet, um die Argumentvektoren $\mathbf{y}$ für die Funktionsauswertungen f_j auszutauschen. Die Implementierung *PipeDb2mtS* verwendet eine für dünnbesetzte Probleme angepasste Kommunikation, ohne dabei Annahmen über die Art der Besetzung zu machen. Die Implementierungen *PipeDb2mtN* und *PipeDb2mtX* hingegen sind speziell für Probleme mit beschränkter Zugriffsdistanz optimiert.

In diesem Kapitel werden die Effizienzen dieser vier Implementierungen bei der Lösung des Testproblems BRUSS2D-MIX verglichen. Dieses hat eine Zugriffsdistanz von $d(\mathbf{f}) = 2N$ bei einer Systemgröße von $n = 2N^2$.

4.2.1 Cluster

Die Effizienzen der Implementierungen auf dem **Cluster** werden in den Abbildungen 3a und 3b graphisch dargestellt.

Wie schon bei der Lösung von BRUSS2D-ROW weist die Implementierung *PipeDb2mt* unabhängig vom verwendeten Verfahren eine sehr schlechte Effizienz für mehr als zwei Prozesse auf.

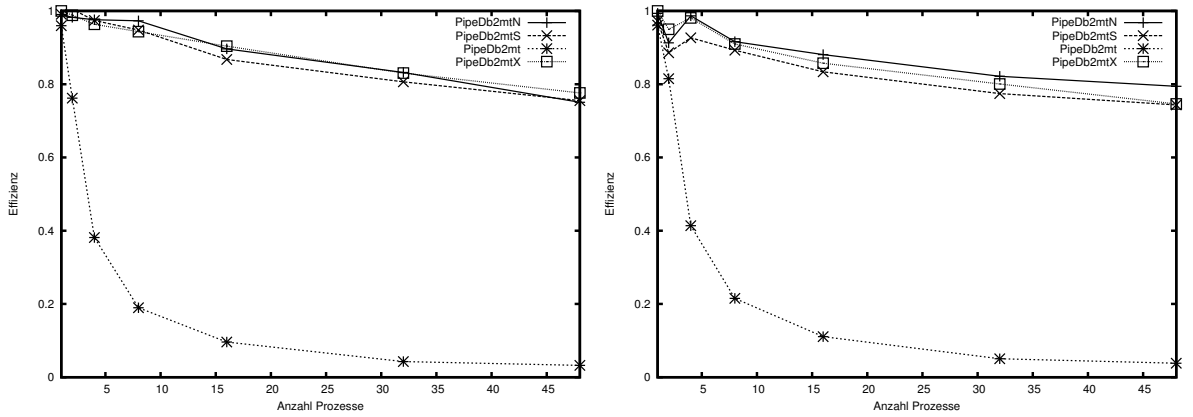
Anders als für das Testproblem BRUSS2D-ROW verhält sich jedoch die Effizienz der Implementierung *PipeDb2mtS*: Der starke Abfall der Effizienz bei der Erhöhung von einem Prozess auf vier Prozesse bleibt hier aufgrund des veränderten Zugriffsmusters der Funktionsauswertungen aus. Bei zwei Prozessen hat diese Implementierung mit dem Radau-Verfahren sogar eine Effizienz von mehr als 100%. Die Gründe für diese gute Performance sind zum einen die kleineren Datenmengen, die die Prozesse austauschen müssen, und zum anderen wahrscheinlich Lokalitätseffekte. Betrachtet man den qualitativen Verlauf der Effizienz von *PipeDb2mtS*, so kann man sagen, dass die Effizienz von knapp 100% bei sequentieller Ausführung mit zunehmender Prozessanzahl langsam abnimmt. Für 48 Prozesse erreicht die Implementierung sowohl mit dem Radau- als auch mit dem Lobatto-Verfahren eine Effizienz von ungefähr 75%.

Interessanterweise unterscheiden sich die Effizienzen der für Probleme mit beschränkter Zugriffsdistanz optimierten Implementierungen *PipeDb2mtN* und *PipeDb2mtX* kaum von der Effizienz von *PipeDb2mtS*. Mit dem Radauverfahren erreichen die beiden Varianten Effizienzen zwischen 75,1 (*PipeDb2mtN*) und 77,6% (*PipeDb2mtX*) bei 48 MPI-Prozessen, während *PipeDb2mtS* eine Effizienz von 75,5% hat. Lediglich bei der Verwendung des Lobatto-Verfahrens heben sich die Effizienzen der beiden Implementierungen leicht von *PipeDb2mtS* ab. Die Implementierung *PipeDb2mtN* erreicht hier eine Effizienz von 79,4% und *PipeDb2mtX* 74,6%. Die Effizienz von *PipeDb2mtS* beträgt 74,3%. Die Prozentangaben beziehen sich wieder auf 48 Prozesse.

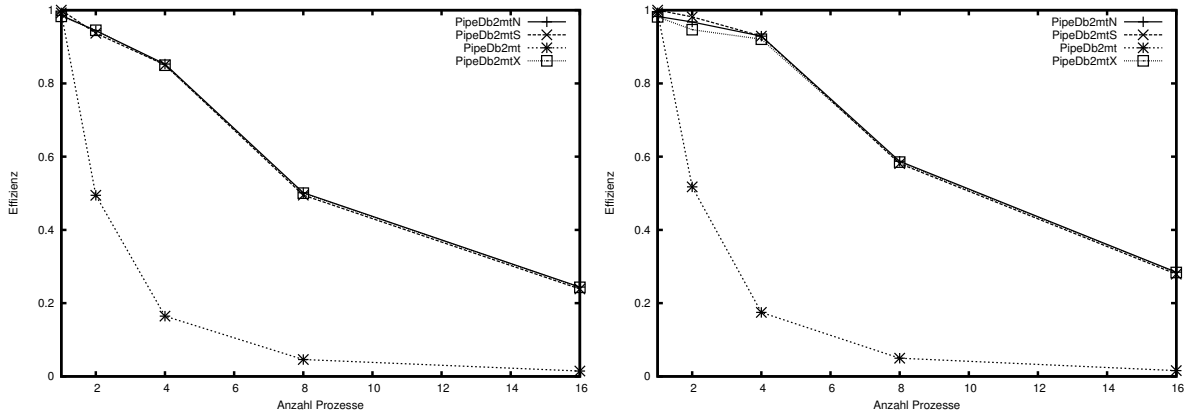
Eigentlich war zu erwarten, dass die Implementierung *PipeDb2mtN* die höchste Effizienz aufweist, da die Zugriffsdistanz kleiner als ein Block der blockweisen Verteilung ist und diese Implementierung nur die notwendige Kommunikation ohne großen Verwaltungsaufwand ausführt.

4.2.2 Hydra

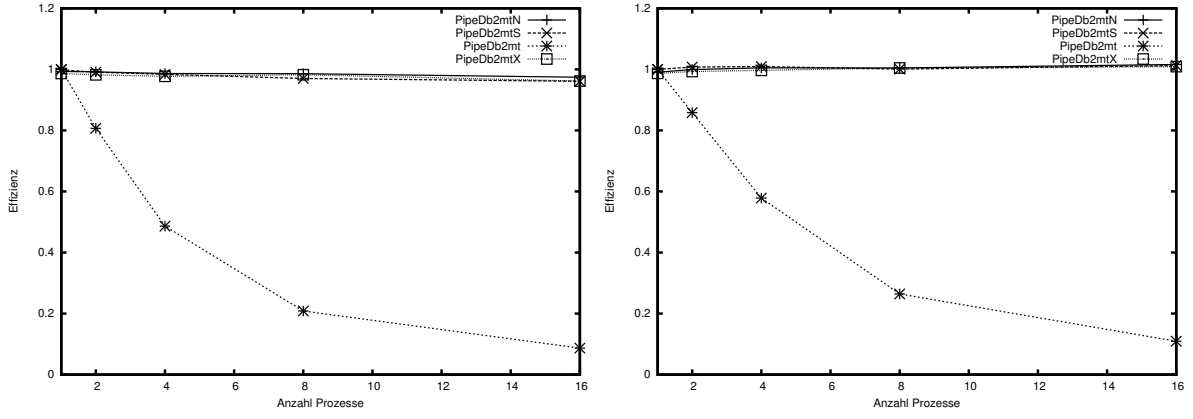
Auch bei dem Rechnersystem **Hydra** weist die Implementierung *PipeDb2mt* äußerst geringe Effizienzen auf. Die Effizienzen der anderen Implementierungen nehmen ausgehend von 100% für einen Prozess stark ab und enden bei



(a) Cluster - Radau IA (5) **(b)** Cluster - Lobatto IIIC (8)



(c) Hydra - Radau IA (5) **(d)** Hydra - Lobatto IIIC (8)



(e) Hydrus - Radau IA (5) **(f)** Hydrus - Lobatto IIIC (8)

Abbildung 3: Effizienzen der Implementierungen bei der Lösung von BRUSS2D-MIX

etwa 24% mit dem Radau-Verfahren oder 27% mit dem Lobatto-Verfahren für 16 Prozesse. Zwischen den Implementierungen unterscheiden sich die erreichten Effizienzen fast nicht. Die schlechte Performance der optimierten Implementierungen liegt wahrscheinlich an der Architektur **Hydra**. Der Memory-Controller der Intel Xeon E7330 CPU ist auf der Northbridge untergebracht, was zu schlechteren Zugriffszeiten führt als wenn dieser direkt im Prozessor integriert wäre, wie es beim **Cluster** und **Hydrus** der Fall ist.

4.2.3 Hydrus

Die Effizienz der Implementierung *PipeDb2mt* sinkt, wie es bisher immer der Fall war, sehr stark wenn mehr als zwei Prozesse für die Berechnungen genutzt werden. Die anderen Implementierungen erreichen mit dem Radau-Verfahren Effizienzen zwischen 96 und 97,5% bei der Verwendung von 16 Prozessen. Mit dem Lobatto-Verfahren werden sogar Effizienzen von fast 102% erreicht. Die ungewöhnlich hohen Effizienzen lassen sich vermutlich auf Lokalitätseffekte zurückführen. Die Effizienzen der unterschiedlichen Implementierungsvarianten *PipeDb2mtS*, *PipeDb2mtN* und *PipeDb2mtX* unterscheiden sich fast nicht.

Die Diagramme sind in den Abbildungen 3e und 3f zu finden.

4.2.4 Zwischenfazit

Es hat sich gezeigt, dass die Optimierungen der Kommunikation speziell für Probleme mit beschränkter Zugriffsdistanz keinen sichtbaren Performancegewinn gegenüber der Variante für dünnbesetzte DGL bringen. Die Implementierungen, deren Kommunikation für dünnbesetzte Probleme angepasst ist, eignen sich daher auch für Probleme mit beschränkter Zugriffsdistanz.

4.3 Effizienz der Implementierungen bei der Lösung von dichtbesetzten Problemen

Nicht für jedes Problem lässt sich zweifelsfrei feststellen, ob dieses dicht- oder dünnbesetzt ist. Der Grund dafür ist, dass schon der Begriff “dünnbesetzt” nicht eindeutig definiert ist. Im Wikipedia-Eintrag ([6]) für “Dünnbesetzte Matrix” steht beispielsweise: eine Matrix heißt dünnbesetzt, wenn “so viele Einträge aus Nullen bestehen, dass es sich lohnt, dies auszunutzen”. Um herauszufinden, wie effizient die Implementierung *PipeDb2mtS* für nicht dünnbesetzte Probleme ist, wurden Messungen mit einem modifizierten BRUSS2D-MIX durchgeführt. Bei der in diesem Kapitel verwendeten Variante des Problems wurde die Methode *ode_get_required_indices_for_intervall* so angepasst, dass alle Komponenten zurückgegeben werden. Damit verhält sich die hier verwendete Implementierung von BRUSS2D-MIX wie ein dichtbesetztes Problem. Als Vergleich dient die Implementierung *PipeDb2mt*, die in jedem Fall alle Komponenten austauscht.

4.3.1 Cluster

Die Effizienz der Implementierung *PipeDb2mt* nimmt etwa exponentiell ab. Mit dem Radau-Verfahren beträgt sie bei 2 Prozessen noch ungefähr 80%. Verdoppelt man die Anzahl der Prozesse so halbiert sich die Effizienz annäherungsweise. Bei 48 MPI-Prozessen beträgt die Effizienz von *PipeDb2mt* nur noch 3-4%, abhängig vom Verfahren.

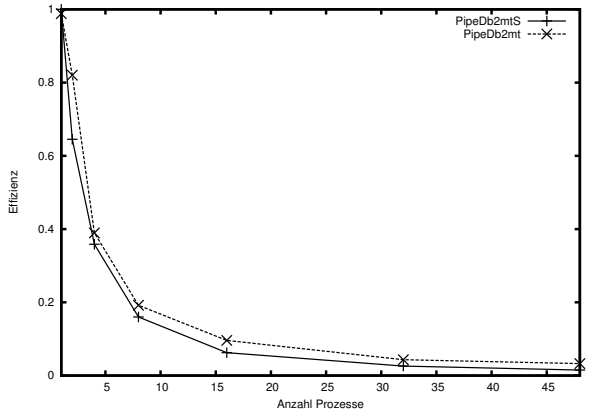
Die Effizienz der Implementierung *PipeDb2mtS* verhält sich für das dichtbesetzte Problem ähnlich wie die Effizienz der *MPI_Allgatherv*-Variante, ist aber stets ein wenig schlechter. Bei 48 Prozessen werden zum Beispiel nur 1,5 bis 1,8% erreicht (siehe Abbildungen 4a und 4b).

4.3.2 Hydra und Hydrus

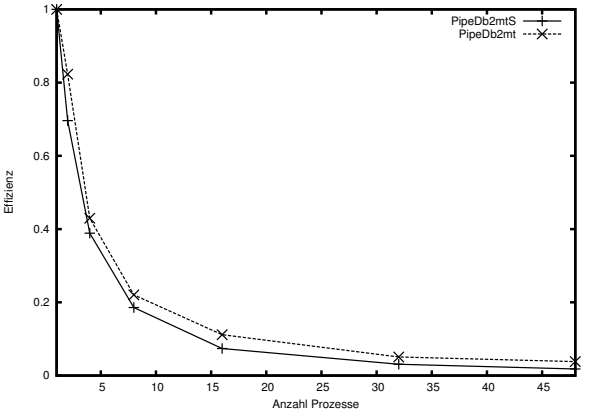
Die Ergebnisse für den **Cluster** lassen sich fast 1:1 auf die Rechnersysteme **Hydra** und **Hydrus** übertragen. Die Implementierung *PipeDb2mtS* ist stets ein wenig ineffizienter als *PipeDb2mt*.

4.3.3 Zwischenfazit

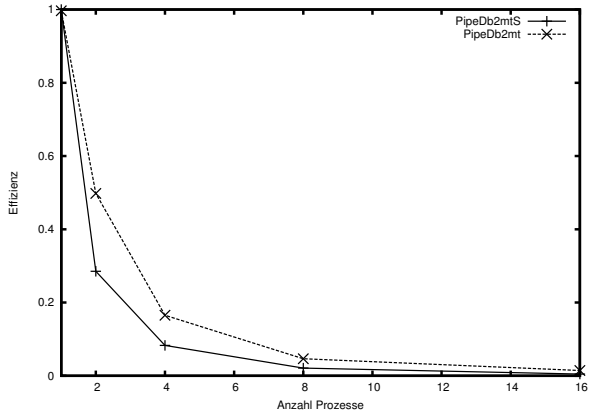
Für dichtbesetzte Probleme sind die Implementierungen, deren Kommunikation für dünnbesetzte DGL optimiert ist, nicht zu empfehlen. Sie sind sogar schlechter als die *MPI_Allgatherv*-Varianten. Je dichtbesetzter ein Problem ist, desto schlechter ist die Performance der Implementierungen für dünnbesetzte DGL. Um herauszufinden, ob diese für ein Problem geeignet sind, müssen im Zweifelsfall einige Proben durchgeführt werden. Mit Hilfe von adaptiven Techniken, die auch Forschungsgegenstand an der Universität Bayreuth sind, lässt sich unter Umständen die effizienteste Implementierung bestimmen, ohne dass viel Rechenzeit in unperformante Implementierungen investiert werden muss.



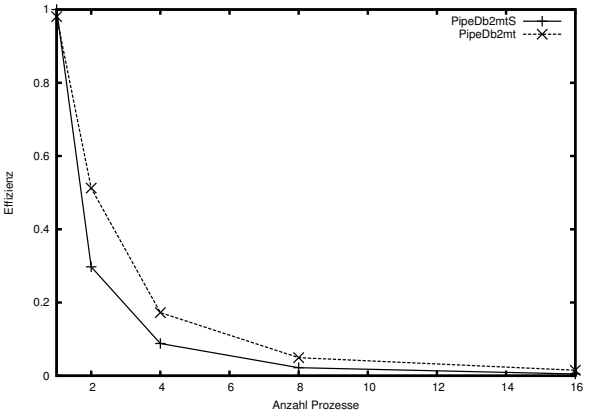
(a) Cluster - Radau IA (5)



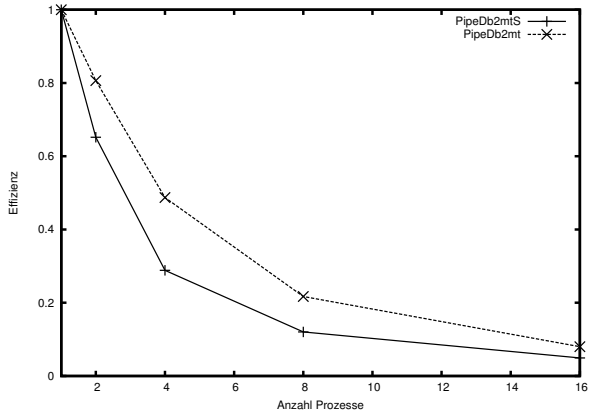
(b) Cluster - Lobatto IIIC (8)



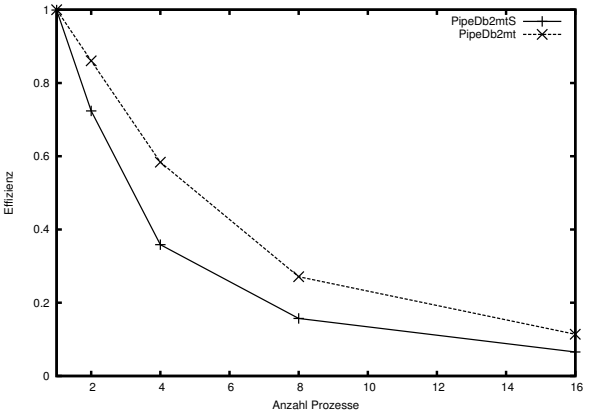
(c) Hydra - Radau IA (5)



(d) Hydra - Lobatto IIIC (8)

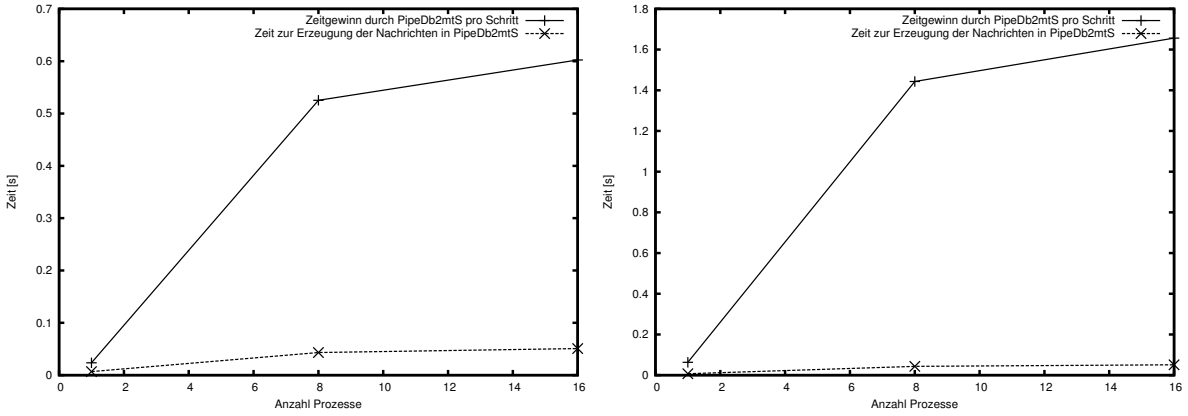


(e) Hydrus - Radau IA (5)



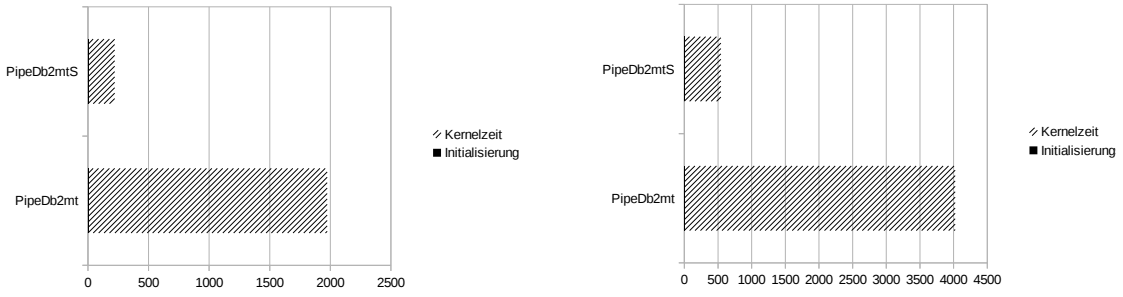
(f) Hydrus - Lobatto IIIC (8)

Abbildung 4: Effizienzen der Implementierungen *PipeDb2mt* und *PipeDb2mtS* bei der Lösung von einem dichtbesetzten DGL



(a) Radau IA (5) **(b)** Lobatto IIIC (8)

Abbildung 5: Initialisierungszeit im Vergleich zu dem Zeitgewinn durch dünnbesetzte Kommunikation



(a) Radau IA (5)

(b) Lobatto IIIC (8)

Abbildung 6: Initialisierungskosten und Gesamtlaufzeit

4.4 Einfluss der Initialisierungskosten von Implementierungen für dünnbesetzte Probleme

Bisher wurden die Effizienzen der Implementierungen für die Berechnung der Zeitschritte untersucht. Dabei konnte festgestellt werden, dass die 'S'-Implementierungen für dünnbesetzte Probleme wesentlich bessere Effizienzen erreichen. Diese Varianten erstellen vor der Ausführung der Zeitschritte die Nachrichtenlisten *inbox* und *outbox*. In diesem Kapitel soll untersucht werden, wie groß der Anteil dieser zusätzlichen Initialisierungskosten an der Gesamtausführungszeit ist.

Dazu wurden auf dem **Cluster** Messungen mit dem Testproblem BRUSS2D-MIX ausgeführt, bei denen neben der Kernel- auch die Initialisierungszeit erhoben wurde.

Um den Einfluss der zusätzlichen Initialisierungskosten unabhängig von der Länge des Integrationsintervalls untersuchen zu können, wird die Zeit zur Erzeugung der Nachrichten mit der Differenz der Zeiten , die *PipeDb2mt* und *PipeDb2mtS* durchschnittlich für die Berechnung eines Zeitschrittes benötigt haben, verglichen. Die Ergebnisse hiervon sind in Abbildung 5 zu sehen. Es ist ersichtlich, dass sich der parallele Einsatz der Implementierung *PipeDb2mtS* schon bei einem einzigen Zeitschritt lohnt, da das Erstellen der Nachrichten weniger Zeit beansprucht, als in jedem Zeitschritt an der Kommunikation gespart wird.

Die Lösung des Brüsselatorproblems unter Verwendung der in Kapitel 4 aufgeführten Parameter benötigt mehrere Hundert Zeitschritte, so dass die Initialisierungskosten kaum einen Einfluss auf die Gesamtausführungszeit haben, was nun anhand der Messungen mit 16 Prozessen gezeigt werden soll.

Die Lösung des Problems BRUSS2D-MIX mit der Implementierung *PipeDb2mt* hat mit dem Radau-Verfahren 1973,9 Sekunden benötigt. Das entspricht ungefähr 33 Minuten. Davon wurden nur 0,4 Sekunden für die Initialisierung verwendet. Die Implementierung *PipeDb2mtS* hat für das selbe Problem 219,9 Sekunden ($\approx 3,6$ Minuten) benötigt, wovon 0,5 Sekunden für die Initialisierung verwendet wurden.

Für die Lösung mit dem Lobatto-Verfahren hat die Implementierung *PipeDb2mt* 4023,8 Sekunden (1 Stunde und 7 Minuten) benötigt, wovon 0,5 Sekunden für die Initialisierung verwendet wurden. Die Implementierung *PipeDb2mtS* hat für das selbe Problem 542,0 Sekunden ($\approx$ 9 Minuten) benötigt, wovon 0,5 Sekunden für die Initialisierung verwendet wurden. Dieser Tatbestand ist in Abbildung 6 zu sehen.

5 Fazit und Ausblick

In dieser Arbeit wurde gezeigt, wie man Multibroadcastoperationen für parallele Implementierungen iterierter Runge-Kutta-Verfahren für verteilten Adressraum bei der Lösung von dünnbesetzten DGL und Problemen mit beschränkter Zugriffsdistanz durch Einzeltransferoperationen ersetzen kann. Die enstandenen Implementierungsfamilien 'S', 'N' und 'X' wurden anschließend mit Laufzeitmessungen untersucht.

Dabei hat sich herausgestellt, dass die Implementierungen der Familie 'S' sowohl für dünnbesetzte als auch für Probleme mit beschränkter Zugriffsdistanz wesentlich höhere Effizienzen aufweisen als entsprechende Varianten, die *MPI_Allgatherv* verwenden.

Je dichtbesetzter das zu lösende DGL aber ist, desto unbefriedigender wird die Performance der 'S' - Implementierungen. Um festzustellen, ob sich der Einsatz einer solchen Implementierung für ein spezielles Problem rentiert, könnte man adaptive Techniken anwenden, um wertvolle Rechenzeit zu sparen.

Der MPI-Standard bietet neben der Methode *MPI_Allgatherv* auch die Multibroadcastoperation *MPI_Allgather*, welche nur gleich große Nachrichten handhaben kann. Diese Methode ist eventuell effizienter als *MPI_Allgatherv*, so dass es sich unter Umständen lohnt, die Größe der Argumentvektoren auf das nächste Vielfache der Prozessanzahl zu erhöhen.

Am 21. September wurde der MPI-Standard 3.0 verabschiedet. In diesem wurden nichtblockierende Multibroadcastoperationen eingeführt. Mit deren Hilfe kann man unabhängige Berechnungen ausführen, während im Hintergrund die Daten übertragen werden. Die meisten Implementierungen von IRK-Verfahren werden davon keinen Vorteil haben, da die Argumentvektoren sofort nach der Multibroadcastoperation wieder benötigt werden und es dadurch keine unabhängigen Berechnungen gibt. Allerdings könnte man das etwas genauer untersuchen und dabei vielleicht doch Möglichkeiten finden, die im neuen MPI-Standard definierten Methoden gewinnbringend einzubringen.

Anhang

A Verwendete Rechnersysteme

A.1 Cluster

Der **Cluster** besteht aus 32 SMP-Knoten mit je zwei Prozessoren vom Typ AMD Opteron DP 246 mit einer Taktfrequenz von 2 GHz, deren Cacheparameter in Tabelle 1 zu finden sind. Die Knoten können über drei unterschiedliche, voneinander unabhängige Netzwerke miteiander kommunizieren. Dabei handelt es sich um ein Fast-Ethernet-Netzwerk mit 100 MBit/s, ein Gigabit-Ethernet-Netzwerk mit 1 GBit/s und ein Infiniband-Netzwerk mit 10 GBit/s (aus [9, Kapitel 3.3]). Für die Laufzeitmessungen dieser Arbeit wurde das Infiniband-Netzwerk verwendet.

Stufe	Typ	Größe	Assoziativität	Zeilengröße
1	Daten	64 KB	2-fach	64 Byte
1	Instruktionen	64 KB	2-fach	64 Byte
2	Unified	1 MB	16-fach	64 Byte

Tabelle 1: Cacheparameter von **Cluster**

A.2 Hydra

Das Rechnersystem **Hydra** besitzt 16 GB gemeinsamen Adressraum und besteht aus 4 Quad-Core Intel Xeon E7330 Prozessoren mit einer Taktfrequenz von 2.4 GHz. Die Cacheparameter stehen in Tabelle 2.

Stufe	Typ	Häufigkeit	Größe	Assoziativität	Zeilengröße
1	Daten	je Kern	32 KB	8-fach	64 Byte
1	Instruktionen	je Kern	32 KB	8-fach	64 Byte
2	Unified	zweimal	3 MB	8 oder 16-fach	64 Byte

Tabelle 2: Cacheparameter von **Hydra**

A.3 Hydrus

Hydrus hat einen gemeinsamen Adressraum von 32 GB und besteht aus 4 Quad-Core AMD Opteron 8350 Prozessoren mit einer Taktfrequenz von 2 GHz. Jeder Prozessor hat vier L1-Caches der Größe 128 KB, vier L2-Caches je 512 KB und einen 2 MB großen L3-Cache (siehe [1]).

Literatur

[1] Datenblatt des AMD Quad-Core Opteron 8350 Prozessors. http://products.amd.com/de-de/OpteronCPUDetail.aspx?id=334&f1=&f2=&f3=Ja&f4=&f5=&f6=&f7=&f8=&f9=&f10=&f11=&, 18. September 2012.

[2] Homepage Leibniz-Rechenzentrum. http://www.lrz.de/services/compute/supermuc/systemdescription/, 18. September 2012.

[3] Internetpräsenz von MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/, 18. September 2012.

[4] Internetpräsenz von Open MPI. http://www.open-mpi.org/, 18. September 2012.

[5] MPI-Forum. http://www.mpi-forum.org/, 18. September 2012.

[6] Wikipedia-Eintrag für Dünnbesetzte Matrix. http://de.wikipedia.org/wiki/DünnbesetzteMatrix, 6. Oktober 2012.

[7] K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations*. Oxford University Press, 1995.

[8] N. Kalinnik, M. Korch, and T. Rauber. An efficient time-step-based self-adaptive algorithm for predictor-corrector methods of runge-kutta type. Technical report, Journal of Computational and Applied Mathematics, 2011.

[9] M. Korch. *Effiziente Implementierung eingebetteter Runge-Kutta-Verfahren durch Ausnutzung der Speicherzugriffslokalität*. PhD thesis, Universität Bayreuth, 2006.

[10] M. Korch and T. Rauber. Locality optimized shared-memory implementations of iterated runge-kutta methods. Technical report, University of Bayreuth, Department of Computer Science, 2007.

[11] M. Korch, T. Rauber, and C. Scholtes. Scalability and locality of extrapolation methods on large parallel systems. Technical report, University of Bayreuth, Applied Computer Science 2, 2011.

[12] H. Meuer, E. Strohmaier, J. Dongarra, and H. D. Simon. Top500 supercomputer sites. http://www.top500.org, 18. September 2012.

[13] T. Rauber and G. Rünger. *Parallele und verteilte Programmierung*. Springer-Verlag, 2000.

Eigenständigkeitserklärung

Hiermit versichere ich, Markus Straubinger, dass ich diese Arbeit selbstständig verfasst und keine anderen als die von mir angegebenen Quellen und Hilfsmittel verwendet habe. Des Weiteren versichere ich, dass diese Arbeit nicht bereits zur Erlangung eines akademischen Grades eingereicht wurde.

Bayreuth, 24.10.2012

Markus Straubinger