Erweiterung eines channelbasierten Task-Laufzeitsystems um eine OpenMP-Nutzerschnittstelle

Christopher Brunner

Bayreuth Reports on Parallel and Distributed Systems

No. 16, März 2022

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

UNIVERSITÄT
BAYREUTH

Lehrstuhl für Angewandte Informatik II
Institut für Informatik

Bachelor-Arbeit

Erweiterung eines channelbasierten Task-Laufzeitsystems um eine OpenMP-Nutzerschnittstelle

Extension of a channel-based task runtime system to include an OpenMP user interface

Christopher Brunner

1. Prüfer:
Prof. Dr. Matthias Korch
2. Prüfer:
Prof. Dr. Thomas Rauber

Betreuer:
Prof. Dr. Matthias Korch

Bayreuth, der 27. Januar 2022

Zusammenfassung

In der Prozessorentwicklung der letzten 15 Jahre konnte Fortschritt aufgrund von physikalischen Limitierungen nicht mehr weiter durch ein stetiges Erhöhen der Taktrate erzielt werden. Stattdessen wurden weitere Leistungsverbesserungen vor allem durch eine größere Anzahl von Rechenkernen auf dem Prozessorchip erreicht. Damit ein einzelnes Programm von diesem Trend profitieren kann, teilt man seine auszuführenden Instruktionen in *Tasks* ein. Diese werden von einem Laufzeitsystem anschließend Threads zugeordnet, welche parallel auf den verschiedenen Prozessorkernen arbeiten. Die Aufgabe eines solchen System besteht dabei in einer möglichst performanten Aufteilung der Tasks auf die Threads, sodass die vorhandenen Rechenressourcen bestmöglich ausgenutzt werden. Ein entsprechendes Task-Laufzeitsystem *Tasking 2.0* [Pre] von A. Prell nutzt zur Verteilung sogenanntes *Work-Stealing*, bei welchem unbeschäftigte Threads die Tasks anderer Threads stehlen. Die Kommunikation zwischen den Threads geschieht dabei ausschließlich über Kommunikationskanäle (*Channels*), welche flexibel an die jeweils zugrunde liegende Architektur des Zielsystems angepasst werden können. Die zur Verwendung des Laufzeitsystems bereitgestellte projekteigene Schnittstelle definiert C-Makros und erfordert ein manuelles Auslagern von Task-Anweisungssequenzen in Funktionen. Weitere Laufzeitsysteme, die ebenso ein Modell zur Parallelisierung auf gemeinsamen Speicher bieten, setzten stattdessen auf die weit verbreitete Programmierschnittstelle *OpenMP* [Ope21], welche sich durch Compiler-Anweisungen anwenderfreundlich nutzen lässt. Im Rahmen dieser Arbeit wird deshalb die OpenMP-Nutzerschnittstelle partiell für das Task-Laufzeitsystem von A. Prell implementiert. Ein spezieller Fokus liegt dabei auf der Realisierung der grundlegenden Schnittstellenfunktionen zur Verwendung von Tasks, parallel ausführbaren Regionen sowie For-Schleifen, welche die Funktionalitäten des Laufzeitsystems abbilden. Das auf diese Weise erweiterte Laufzeitsystem vereint dadurch die Implementierungsbesonderheiten des Tasking 2.0 mit der einfachen Anwendbarkeit einer OpenMP-Nutzerschnittstelle.

Abstract

In the development of processors over the past 15 years, progress could no longer be achieved by constantly increasing the clock speed due to physical limitations. Instead, further performance improvements have primarily been obtained by increasing the number of processing cores on the processor chip. For a single program to benefit from this trend, its instructions must be divided into *tasks*. These are then assigned to threads by a runtime system, which operate in parallel on the various processor cores. The purpose of such a system is to distribute the tasks to the threads as efficiently as possible, so that the available computing resources are utilised in the best possible way. A corresponding task runtime system *Tasking 2.0* [Pre] by A. Prell uses so-called *Work-Stealing* for distribution, in which idle threads steal the tasks of other threads. The communication between the threads takes place exclusively via *Channels*, which can be adapted flexibly to the underlying architecture of the target system. The project-specific interface provided for using the runtime system defines C macros and requires manual swapping of a tasks statement-sequence into functions. Other runtime systems, which also provide a shared memory programming model for parallelisation, instead rely on the widely used programming interface *OpenMP* [Ope21], which can be used in a user-friendly way through compiler instructions. In the context of this thesis, the OpenMP user interface is therefore partially implemented for A. Prell's task runtime system. A special focus is put on the realization of the fundamental interface features for the use of tasks, parallel executable regions as well as for-loops, which represent the functionalities of the runtime system. The result is an extended runtime system that combines the special implementation properties of Tasking 2.0 with the simple applicability of an OpenMP user interface.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Da die Taktrate eines Prozessors im direkten Zusammenhang mit einem höheren Energieverbrauch sowie einer höheren Wärmeentwicklung steht, ist diese nach oben durch die maximale Kühlleistung des Systems beschränkt. Aus diesem Grund konnten Performanceverbesserungen in der Prozessorentwicklung der letzten 15 Jahre nicht mehr alleine durch ein stetiges Erhöhen der Taktrate erzielt werden. Stattdessen wurden weitere Leistungsgewinne vor allem durch eine zunehmende Parallelisierung auf Hardwareebene erreicht. So lässt sich in diesem Zeitraum ein deutlicher Anstieg an Rechenkernen auf dem Prozessorchip beobachten.

Damit ein einzelnes Programm allerdings auch von diesem Trend profitieren kann, ist es notwendig, dass seine auszuführenden Instruktionen auf mehrere Threads aufgespalten werden. Verteilt man die Threads anschließend auf die vorhandenen Kerne, so können mehrere Instruktionen des Programms parallel ausgeführt werden, wodurch sich schließlich eine kürze Programmausführzeit erzielen lässt.

Ein verbreiteter Ansatz zur Aufteilung der Instruktionen ist dabei die Untergliederung von parallel bearbeitbarem Programmcode in *Tasks*. Diese Tasks werden anschließend von einem Laufzeitsystem (LZS) auf die Threads aufgeteilt, welche die Tasks auf ihrem jeweiligen Prozessorkern bearbeiten. Ein LZS hat dabei die Aufgabe, die Verteilung möglichst gleichmäßig und performant durchzuführen, sodass die vorhandenen Rechenressourcen bestmöglich ausgenutzt werden.

Im Rahmen dieser Arbeit wird dabei auf ein bereits existentes Task-Laufzeitsystem von A. Prell (*Tasking 2.0* [Pre]) aufgebaut. Dieses nutzt zur Verteilung der Tasks sogenanntes *Work-Stealing*. Dabei verwaltet jeder Thread eine eigene private Warteschlange von Tasks, welche er bearbeitet. Sobald diese Warteschlange leer ist, versucht er durch Stehlen von anderen Threads weitere Tasks zur Ausführung zu erhalten.

Einen großen Augenmerk legt dieses LZS zudem auf eine explizite Kommunikation zwischen den Threads über Kanäle (engl.: *Channels*). Aktuell wird dazu eine Channel-Implementierung unter Ausnutzung eines gemeinsamen Adressraums bereitgestellt. Allerdings ist aufgrund der Verwendung von expliziter Kommunikation auch ein zukünftiges Anwenden des Laufzeitsystems für das Arbeiten auf Hochleistungsrechnern möglich, denen häufig verteilter Speicher zugrunde liegt. Dazu lässt

sich die Channel-Implementierung flexibel durch eine Variante austauschen, welche an die jeweilige Architektur angepasst ist.

Zur Verwendung des LZS von A. Prell wird aktuell eine projekteigene Schnittstelle zur Verfügung gestellt. Diese definiert einzufügende C-Makros und erfordert für ihre Anwendung ein manuelles Auslagern von Task-Anweisungssequenzen in Funktionen. Weitere Laufzeitsysteme, welche ein Modell zur Parallelisierung auf gemeinsamen Speicher bieten, verwenden stattdessen die Nutzerschnittstelle *OpenMP* [Ope21]. Diese ist den meisten Programmierern aufgrund ihrer weiten Verbreitung bereits vertraut und funktioniert auf Grundlage von Compiler-Anweisungen, welche von vielen bekannten C-Compilern unterstützt werden.

1.2 Zielsetzung

Das Tasking 2.0 LZS, welches im Rahmen der Dissertationsarbeit [Pre16] von A. Prell entstanden ist, soll im Zuge dieser Bachelorarbeit durch eine partielle Implementierung der OpenMP-Nutzerschnittstelle erweitert werden. Dies soll zum einen die Verwendung des Laufzeitsystems für den Anwender erleichtern und zum anderen eine bessere Vergleichbarkeit mit bereits existierenden OpenMP-Laufzeitsystemen wie dem *libgomp* [GCC] aus *GCC* oder *libomp* [LLV] aus dem *LLVM*-Projekt schaffen.

Erstrebt wird eine möglichst breite Abdeckung der Funktionalitäten des Tasking 2.0 durch die Implementationen der entsprechenden Schnittstellendefinitionen des OpenMP-Standards. Dies beinhaltet hauptsächlich die Konstrukte zur Verwendung von Tasks sowie von For-Schleifen. Gleichzeitig soll eine Erhöhung der Ausführzeiten nach Möglichkeit vermieden werden.

Zudem soll das LZS auch um grundlegende OpenMP-Konstrukte wie beispielsweise Parallelregionen erweitert werden, da diese für den Umgang mit Tasks nach OpenMP-API-Definition erforderlich sind. Auf diese Weise soll auch eine möglichst große Breite von Programmen mit dem LZS ausführbar werden, die bereits mittels OpenMP parallelisiert sind.

1.3 Aufbau

Die Arbeit gliedert sich im Allgemeinen in sechs Kapitel. Im Anschluss an die Einleitung wird in Kapitel 2 eine Einführung in die Grundlagen von Task-Laufzeitsystemen und den damit verbundenen Termini gegeben. Spezieller Fokus liegt dabei auf den im Tasking 2.0 LZS implementierten Techniken. Ebenso behandelt dieses Kapitel in seinem zweiten Teil die Nutzerschnittstelle OpenMP. Es wird geklärt, auf welche Weise aus OpenMP-Konstrukten paralleler Code erzeugt werden kann.

In Kapitel 3 wird erläutert, auf welche Weise die in Kapitel 2 beschriebenen Elemente eines Task-Laufzeitsystems im Falle des Tasking 2.0 realisiert sind.

Die Funktionalität und Implementierung der im Zuge der Arbeit eingefügten OpenMP-Konstrukte wird anschließend in Kapitel 4 erläutert. Dabei wird zu jedem OpenMP-Konstrukt erklärt, welche Funktionalitäten des Tasking 2.0 für die Implementation verwendet werden können und welche Funktionalitäten neu hinzugefügt werden müssen.

In Kapitel 5 werden Laufzeitmessungen mit dem erweiterten LZS durchgeführt und die Performance wird mit dem ursprünglichen Tasking 2.0 sowie ähnlichen OpenMP-Laufeitsystemen verglichen.

Eine Zusammenfassung der in dieser Arbeit gewonnenen Erkenntnisse wird abschließend in Kapitel 6 gegeben.

2 Grundlagen und Begriffsklärungen

In diesem Kapitel wird im ersten Teil eine Einführung in Task-Laufzeitsysteme gegeben. Dabei wird insbesondere auf die Begriffe eingegangen, mit welchen sich das Tasking 2.0 LZS klassifizieren lässt. Im zweiten Teil wird eine Einführung in die OpenMP-Nutzerschnittstelle gegeben. Es wird darin der Erzeugungsprozess von parallelem Code aus OpenMP-annotiertem Quellcode erläutert, welcher von einem OpenMP-unterstützenden Compiler durchgeführt wird.

2.1 Grundlegende Termini

Das folgenden Kapitel erklärt Begriffe, welche zur Klassifizierung des Tasking 2.0 verwendet werden. Die Beschreibung der Begriffe basiert dabei auch immer auf der zugehörigen Arbeit von A. Prell [Pre16]. Weitere Quellen werden wie üblich notiert.

2.1.1 Tasks

Unter dem Begriff *Task* versteht man im Kontext eines Task-Laufzeitsystems eine Sequenz an Anweisungen, welche parallel zum Rest des Programms ausgeführt werden kann und soll. Beispielsweise lässt sich ein asynchroner Funktionsaufruf oder auch eine unabhängige Schleifeniteration als Task verstehen. Dabei ist es in der Regel die Aufgabe des Programmierers, jene Stellen seines Codes als Tasks zu markieren. Die Bearbeitung eines Tasks findet zu dem Zeitpunkt statt, an welchem ein Thread die Befehlssequenz des Tasks auf einer Recheneinheit ausführt.

Die Abstraktion auf Tasks wird vorgenommen, da sie dem Programmierer eine einfachere Parallelisierung seines Programms ermöglicht, als dies bei der klassischen Thread-Programmierung der Fall ist. Statt den Programmablauf für jeden Thread einzeln programmieren zu müssen, genügt es bei einer Parallelisierung mit Tasks, einzelne Aufgaben zu definieren, welche anschließend von einem sogenannten *Task-Laufzeitsystem* (siehe Kap. 2.1.2) automatisiert den Threads zugewiesen werden.

2.1.2 Task-Laufzeitsysteme

Damit ein Programm zur Laufzeit Speicherplatz, den es anfordert, auch auf dem Hauptspeicher zugewiesen bekommt und damit parallel auszuführende Abschnitte des Programms auch wirklich von mehreren Threads durchgeführt werden, verwendet der Compiler ein sogenanntes *Laufzeitsystem*. Dieses kümmert sich zur Ausführzeit unter anderem um eine Ressourcenverteilung für das laufende Programm und bildet eine Schnittstelle zwischen Programmcode und Betriebssystem [al07].

Das in dieser Arbeit beschriebene Task-LZS kümmert sich dabei in erster Linie um eine möglichst ausgeglichene Zuweisung von Tasks auf Threads. Es stellt damit nur den Teil eines gesamten Laufzeitsystems dar, welches vom Compiler zur Ausführung von C-Programmen auf Unix-Betriebssystemen verwendet wird. Aus diesem Grund wird das Tasking 2.0 LZS ausschließlich im Zusammenspiel mit dem LZS des benutzten Compilers verwendet.

Ein Task-LZS wird dann aktiv, wenn aus dem parallelisiertem Programm heraus eine Funktion der Schnittstelle des Task-Laufzeitsystems aufgerufen wird. Diese Aufrufe werden entweder vom Programmierer manuell eingefügt (im Falle von Tasking 2.0) oder auf Wunsch des Programmierers vom Compiler zur Übersetzungszeit eingebaut (im Falle von OpenMP). Auf den Übersetzungsprozess wird dabei genauer in Kapitel 2.2.3 eingegangen.

2.1.3 Verteilte Warteschlangen

Innerhalb eines Task-Laufzeitsystems werden Tasks, welche bereits erstellt aber noch nicht ausgeführt wurden, in einer Warteschlange zwischengespeichert. Verwaltet jeder Thread eine eigene private Warteschlange mit Tasks spricht man auch von *verteilten Warteschlangen*. Das Gegenstück dazu wäre eine einzelne, gemeinsame Warteschlange für alle Tasks. Da ein Zugriff auf eine gemeinsame Warteschlange jedoch synchronisiert werden muss, hat sich gezeigt, dass verteilte Warteschlangen besser mit einer Erhöhung der Threadanzahl skalieren.

Es kann allerdings vorkommen, dass die private Warteschlange eines Threads bereits leer ist, während ein anderer Thread noch viele Tasks zwischenspeichert. Ohne zusätzliche Kommunikation zwischen den Threads würde es hier schnell zur einer schlechten Lastverteilung kommen. Bei Bedarf kann deshalb ein unbeschäftigter Thread mithilfe sogenannter Stehl-Anfragen (siehe Kap. 2.1.4) Tasks aus der Warteschlange eines anderen Threads erhalten.

2.1.4 Work-Stealing

Wie bereits in Kapitel 2.1.2 beschrieben, kümmert sich das Task-Laufzeitsystem um die Verteilung der Tasks auf die Threads. Realisiert wird dies bei privaten

Warteschlangen durch ein Stehlen von Arbeit (engl.: *Work-Stealing*). Dabei versenden *Dieb*-Threads (engl.: thiefs) sogenannte *Stehlanfragen* (engl.: steal-requests) an die anderen Threads (sog.: *Opfer*, engl.: victims). Diese Anfragen werden üblicherweise dann gesendet, wenn die Warteschlange eines Dieb-Threads bereits leer ist oder erwartet wird, dass diese bald leer sein wird. Empfängt ein Opfer-Thread mit gefüllter Warteschlange eine solche Anfrage, beantwortet er diese, indem er einen oder mehrere Tasks zurückschickt. Die Anzahl der Tasks hängt dabei von der gewählten Work-Stealing-Strategie ab.

Da das Stehlen eines Tasks mit zeitlichen Kosten für den Thread verbunden ist, welcher die Anfrage sendet, sowie auch für das Opfer, welcher die Anfrage mit Tasks beantwortet, ist es in einigen Situationen ratsam, mehrere Tasks auf einmal zu stehlen. Dies ist vor allem dann der Fall, wenn sich viele Tasks in der Warteschlange des Opfers befinden, welche jeweils nur eine geringe Bearbeitungszeit benötigen (sog. *leichtgewichtige* Tasks, engl.: lightweight tasks). Beim Stehlen eines einzelnen Tasks würde der Dieb dann nach dem Empfang des Tasks schon nach kurzer Zeit wieder erneut unbeschäftigt sein und eine Stehlanfrage senden. Das viele Stehlen würde zu einem großen zeitlichen Mehraufwand (engl.: Overhead) verglichen mit der Ausführzeit der eigentlichen Tasks führen. Strategien für das Stehlen mehrerer Tasks sind beispielsweise *Steal-Half* (dt.: Stehle die Hälfte), bei welcher die Hälfte aller Tasks aus der Warteschlange des Opfer-Threads gestohlen wird.

Im Gegensatz dazu kann das Stehlen eines einzelnen Tasks in Fällen ratsam sein, in welchen das Senden von Tasks zwischen zwei Threads nur mittels langsamer Kommunikationskanäle möglich ist. Hier kann ein unnötiges Senden zu vieler Tasks mehr Overhead produzieren, als durch das Verteilen für eine bessere Balancierung gewonnen werden kann.

2.1.5 Channels

Zum Senden von Stehlanfragen und Tasks zwischen den Threads können sogenannte *Channels* (dt.: Kanäle) zum Einsatz kommen. Ein Channel ist dabei die Bezeichnung für ein abstraktes Modell zum Nachrichten-Austausch zwischen den Threads. Die Kommunikation mittels Channels kann dabei asynchron stattfinden, sofern Puffer für die Zwischenspeicherung von Nachrichten vorhanden sind. Ohne Zwischenspeicherung erfolgt ein synchrones Senden und Empfangen. Die Anzahl der Sender und Empfänger, die über einen einzelnen Channel miteinander kommunizieren können, ist dabei implementierungsabhängig (siehe Kap. 3.6).

Verwendet ein LZS ausschließlich explizite Kommunikation über Channels zwischen den Threads, so ist das LZS dadurch auch einfach auf andere Architekturen portierbar. In diesen Fällen genügt es, die jeweilige Channel-Realisierung auszutauschen. Beispielsweise existieren Channels, welche den Nachrichtenaustausch über einen

gemeinsamen Speicher (engl.: shared memory) zwischen den Threads realisieren, sowie auch Implementierungen, bei welchen der Austausch auf Systemen mit ausschließlich verteiltem Speicher (engl.: distributed memory) stattfinden kann.

2.2 OpenMP

OpenMP ist eine weit verbreitete Programmierschnittstelle (engl.: Application Programming Interface, kurz: API) für das parallele Programmieren auf Systemen mit gemeinsamen Speicher. Die in diesem Kapitel präsentierten Informationen über OpenMP entspringen der Definition des OpenMP-Standards [Ope21].

2.2.1 Verwenden der Schnittstelle

Ein zu parallelisierendes C-Programm kann durch den Programmierer mit Compiler-Anweisungen (sog. *Direktiven*) und Laufzeitroutinen (engl.: runtime routines) aus der OpenMP-API annotiert werden (vlg. Codebeispiel 2.1). Eine OpenMP-Compiler-Direktive in C beginnt dabei immer mit `#pragma omp`. Sie kann zusätzlich mit Klauseln (engl.: clauses) erweitert werden, mithilfe derer sich die Funktionalität der Direktive genauer einstellen lässt. Eine Direktive inklusive ihrer Klauseln sowie der zur Direktive ggf. zugehörige Anweisungsblock, welcher direkt auf diese folgt, definieren zusammen ein *Konstrukt* (engl.: construct). Zusätzlich existieren in der OpenMP-API noch verschiedene Umgebungsvariablen, mit denen sich das Verhalten eines parallelisierten Programms steuern lässt.

Damit ein OpenMP-unterstützender Compiler den annotierten Code in parallelen Zielcode verwandeln kann (siehe dazu Kap. 2.2.3), enthält die Schnittstelle *SPMD*-Konstrukte (engl.: Single Program Multiple Data, dt.: ein Programm auf mehreren Daten), Task-Konstrukte, sowie Konstrukte zur Synchronisation, zur Arbeitsverteilung und viele weitere. Die Implementierung eines OpenMP-realisierenden Laufzeitsystems ist dabei nicht durch den OpenMP-Standard festgelegt. Es werden lediglich die Funktionalitäten der Schnittstelle festgehalten. Aus diesem Grund unterscheiden sich die Laufzeitsysteme und daraus resultierend auch die Laufzeiten eines parallelisierten Programms je nach verwendetem LZS.

2.2.2 Konstrukte für das Tasking 2.0 LZS

Betrachtet man die Schnittmenge der Funktionalitäten des Tasking 2.0 Laufzeitsystems mit den Funktionalitäten, welche in der OpenMP-API festgeschrieben sind, lässt sich in Erfahrung bringen, welche Direktiven und Laufzeitroutinen aus der Schnittstelle für das LZS implementiert werden sollten. Eine große Überschneidung gibt es demnach bei den Konstrukten für Tasks. Im Tasking 2.0 LZS existieren Realisierungen zum Erstellen, Verteilen und Bearbeiten eines Tasks durch Threads. Verglichen dazu werden in der OpenMP-API-Definition ([Ope21] Kapitel 12) verschiedene Direktiven

zur Verwaltung von Tasks aufgeführt. Auch Barrieren zur Synchronisation auf Task-Ebene existieren sowohl im LZS als auch in der API ([Ope21] Kapitel 15). Gleiches gilt für eine Parallelisierung von For-Schleifen ([Ope21] Kapitel 11).

Selbstverständlich gibt es auch einige Funktionalitäten, welche nicht in der Schnittmenge liegen. Einige OpenMP-Konstrukte, welche im Zusammenhang mit Tasks zwingend benötigt werden, müssen dem Tasking 2.0 deshalb zusätzlich im Zuge dieser Arbeit hinzugefügt werden. So zum Beispiel die Realisierungen der OpenMP-Parallelregionen, in welchen mehrere Threads parallel den gleichen Anweisungsblock bearbeiten. Nur Tasks, welche innerhalb einer solchen Parallelregion erstellt werden, werden nach dem OpenMP-Standard auch auf die verschiedenen Threads der Parallelregion aufgeteilt.

Ein großer Teil der OpenMP-API enthält allerdings auch Konstrukte, welche in keinerlei Relation mit den Funktionalitäten des Tasking 2.0 LZS stehen. Diese werden in dieser Arbeit daher nicht implementiert. Aus diesem Grund stellt die hier realisierte Erweiterung auch nur eine partielle OpenMP-Implementierung dar.

```c
#include <assert.h>
#include "omp.h"

int main(void) {
    int s, a = 4, b = 2;

    // Start of parallel region
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task \
            firstprivate(a, b) \
            shared(s)
            {
                int c = omp_get_num_threads();
                s = a + b + c;
            }
        }
        #pragma omp barrier
        assert(s == (6 + omp_get_num_threads()));

    } // End of parallel region

    return 0;
}
```

Listing 2.1.: C-Programm mit OpenMP-Konstrukten und Laufzeitfunktionsaufrufen

2.2.3 Compilieren

Ein mit OpenMP oder auch mit der API des Tasking 2.0 Laufzeitsystems annotiertes Programm wird jeweils durch einen Compiler in Maschinencode übersetzt. Das Interface des ursprünglichen Tasking 2.0 Laufzeitsystems verwendet dabei C-Makros, welche vom Präprozessor durch C-Code ersetzt werden, um die zusätzliche Logik für die Parallelisierung einzufügen (beispielsweise Funktionsaufrufe ins LZS). Der annotierte Code kann dadurch von allen C-Compilern übersetzt werden. Der Nachteil liegt allerdings darin, dass man an die Limitierungen des Präprozessors und an die vorhandenen Sprachkonstrukte der Programmiersprache C gebunden ist. So muss deshalb bei der Tasking 2.0 Schnittstelle Code, welcher innerhalb eines Tasks ausgeführt werden soll, händisch im ursprünglichen Programm in eine Funktion ausgelagert werden (vgl. Codeausschnitt 2.2). Auch lässt sich der mit Makros versehene Code nicht mehr sequentiell ohne die Verwendung des Laufzeitsystems ausführen ohne dass vorher wieder alle Änderungen am Ausgangsprogramm rückgängig gemacht werden.

```c
#include <assert.h>
#include "tasking.h"

void sum(int a, int b, int* c) {
    *c = a + b;
}

DEFINE_ASYNC(sum, (int, int, int*));

int main(int argc, char *argv[]) {

    int s, a = 4, b = 2;

    // Start of parallel region
    TASKING_INIT(&argc, &argv);

    ASYNC(sum, (a, b, &s));

    TASKING_BARRIER();

    assert(s == 6);

    // End of parallel region
    // Implicit barrier
    TASKING_EXIT();

    return 0;
}
```

Listing 2.2.: Tasking 2.0 API-Aufrufe

Bei OpenMP auf der anderen Seite erhält man durch ein einfaches Ignorieren der eingefügten OpenMP-Direktiven wieder einen sequentiell ausführbaren Code. Dies ist beispielsweise hilfreich, wenn einmal nicht die passende Hardwareunterstützung für

eine parallele Ausführung gegeben ist. Das Ignorieren ist möglich, da die Compiler-Anweisungen keinen eigentlichen Bestandteil des C-Sprachstandards darstellen. Bei einer API, welche Compileranweisungen verwendet, werden Codeveränderungen, wie z.B. Auslagerung in eine Funktion vom Compiler durchgeführt. Damit beim Übersetzen dann paralleler Code entsteht, muss ein Compiler verwendet werden, welcher die OpenMP-API und deren Direktiven unterstützt. Bekannte C-Compiler, welche dies tun, existieren unter anderem von GNU, Intel, LLVM und NVIDIA (siehe für eine vollständige Liste [Ope20]).

Der Aufbau eines solchen Compilers lässt sich nach [al07] in ein Frontend und ein Backend gliedern (vlg. Abbildung 2.1). Quellcode wird durch das Frontend in sogenannten *Zwischencode* (engl.: intermediate representation, kurz: IR) umgewandelt. Dieser ist unabhängig von der Programmiersprache des Quellcodes und lässt sich besonders gut für weitere Codeoptimierungen nutzen. Der ggf. optimierte Zwischencode wird anschließend an das Backend des Compilers übergeben, welches Zielcode in gewünschter Form ausgibt. Im Rahmen dieser Arbeit entspricht der Quellcode einem OpenMP-annotiertem C-Code und der Zielcode ist Maschinensprache für die Architektur des Rechners, welcher das Programm ausführen soll.

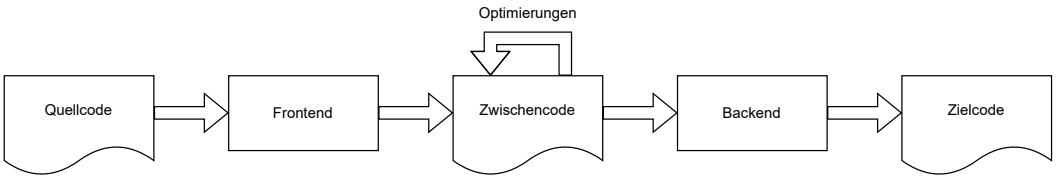

Abb. 2.1.: Compiler Phasen

Auch die OpenMP-Direktiven werden als Bestandteil des Quellcodes durch das Frontend umgewandelt. Vergleicht man nun den OpenMP-C-Code mit dem daraus resultierendem Zwischencode, so erkennt man die Umwandlungen, welche der Compiler in der ersten Phase unternimmt. Als Beispiel für einen Quellcode wird hier ein einfaches *Hallo-Welt*-Programm in C verwendet, welches durch eine OpenMP *task*-Direktive ergänzt wurde (siehe Listing 2.3). Zur Laufzeit soll hier ein Task erstellt und ausgeführt werden, welcher „Hallo Welt!“ ausgibt. Zur Umwandlung wird Clang als Compiler-Frontend verwendet, welches LLVM-Zwischencode für das Compiler-Backend LLVM produziert.

Der erstellte Zwischencode ist verkürzt in Listing 2.4 abgebildet. Es lässt sich erkennen, dass der Code, welcher als Task ausgeführt werden soll, vom Compiler in eine neue Funktion namens `.omp_task_entry.` ausgelagert wurde. In die `main`-Funktion wurden im Zuge der Übersetzung drei Funktionsaufrufe eingefügt, welche ein Laufzeitsystem aufrufen (`__kmpc_global_thread_num`, `__kmpc_omp_task_alloc`, `__kmpc_omp_task`). Mithilfe dieser Aufrufe wird die Thread-ID des aufrufenden Threads ermittelt, diese anschließend zur Allokation eines Tasks verwendet und zuletzt ein Task im LZS erstellt (siehe dazu Kap. 4.7). Der Compiler übernimmt hier

also auch jene Schritte, welche zur Verwendung des Tasking 2.0 Laufzeitsystems noch händisch zu erledigen waren.

```
#include <stdio.h>

int main(void) {
            #pragma omp task
            {
                printf("Hallo Welt!\n");
            }
    return 0;
}
```

Listing 2.3.: Annotierter Quellcode

```
...
@.str = private unnamed_addr constant [13 x i8] c"Hallo Welt!\0A\00", align 1

define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca %struct.anon, align 1
  %3 = call i32 @__kmpc_global_thread_num(%ident_t* @0)
  store i32 0, i32* %1, align 4
  %4 = call i8* @__kmpc_omp_task_alloc(%ident_t* @0, i32 %3, i32 1, i64 40, i64 0,
        i32 (i32, i8*)* bitcast (i32 (i32, %struct.kmp_task_t_with_privates*)*
        @.omp_task_entry. to i32 (i32, i8*)*))
  %5 = bitcast i8* %4 to %struct.kmp_task_t_with_privates*
  %6 = getelementptr inbounds %struct.kmp_task_t_with_privates,
        %struct.kmp_task_t_with_privates* %5, i32 0, i32 0
  %7 = call i32 @__kmpc_omp_task(%ident_t* @0, i32 %3, i8* %4)
  ret i32 0
}

...

define internal i32 @.omp_task_entry.
        (i32, %struct.kmp_task_t_with_privates* noalias) #2 {
  ...
  %20 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([13 x i8],
        [13 x i8]* @.str, i32 0, i32 0)) #3
  ret i32 0
}
...
```

Listing 2.4.: LLVM Zwischencode

In dieser Arbeit kann man sich diese Vorgehensweise eines Frontend-Compilers zu Nutze machen, indem man am Zwischencode ansetzt und die Aufrufe, welche eigentlich in das LZS von LLVM führen, an das erweiterte Tasking 2.0 LZS "umleitet". Dies kann erreicht werden, indem das Tasking 2.0 für die eingefügten Funktionen im Zwischencode eigene Implementierungen bereitstellt. Wird dann das Tasking 2.0 anstelle des von LLVM bereitgestellten Laufzeitsystems gelinkt, so werden durch die Aufrufe die Implementierungen aus dem Tasking 2.0 LZS ausgeführt.

Zur Erweiterung des Tasking 2.0 wurde sich dazu entschieden, den beschriebenen Clang/LLVM-Compiler und die entsprechenden Laufzeitaufrufe im Zwischencode zu verwenden. Im Vergleich zu anderen Compilern wie dem GCC ist die Schnittstelle hier vergleichsweise gut dokumentiert (siehe [LLV15]) und die durchgeführten Umwandlungen der Direktiven sind im Zwischencode gut erkenntlich. Auch die Arbeit von Cownie et al. [CK21a], welche ebenso ein eigenes OpenMP-LZS [CK21b] kreiert, wählte aus genannten Gründen bereits diese Vorgehensweise und dient dieser Arbeit daher an einigen Stellen als Implementierungsvorlage.

3 Implementierungsmerkmale des Tasking 2.0 Laufzeitsystems

In diesem Kapitel wird die grundlegende Implementierung des Tasking 2.0 Laufzeitsystems erklärt. Zu den in Kapitel 2.1 eingeführten Begriffen wird dabei jeweils die Realisierung innerhalb des Tasking 2.0 dargestellt. Besonderer Fokus liegt auf Eigenschaften, welche für die Realisierung der OpenMP-API in Kapitel 4 wiederverwendet werden. Die Beschreibung bezieht sich auf die Implementierung aus [Pre] und die dazugehörige Arbeit von A. Prell [Pre16].

3.1 Tasks

Mithilfe des Tasking 2.0 lässt sich ein Task realisieren, indem der Programmierer den Anweisungsblock für einen Task händisch in eine eigene Funktion auslagert. Ein Zeiger auf diese Funktion wird intern zusammen mit den Werten für die Übergabeparameter in einer C-Struktur `task` abgespeichert (siehe Listing 3.1 Elemente `fn` und `data`). Diese Struktur enthält zusätzlich noch Metadaten zum Task, wie beispielsweise einen Zeiger auf den Eltern-Task, falls ein Task geschachtelt innerhalb eines anderen Tasks erstellt wird.

Im Tasking 2.0 LZS ist es zudem möglich, neben einzelnen Codeblöcken, die als Funktion ausgelagert werden, auch die Schleifeniterationen einer For-Schleife in Tasks aufzuspalten. Dabei wird in der `task`-Struktur zu jedem Schleifen-Task zusätzlich im Element `splittable` abgespeichert, ob es sich um einen solchen handelt. Die Elemente `start`, `cur` und `end` geben an, welche Iterationen der Schleife der jeweilige Task umfasst (siehe Kap. 3.11).

```
struct task {
        struct task *parent;
        void (*fn)(void *);
        char data[TASK_DATA_SIZE];

        // Ausschließlich für Schleifen-Tasks:
        long start;
        long cur;
        long end;
        bool splittable;
        ...
};
```

Listing 3.1.: Gekürzte `task` Struktur

Tasks werden zu Beginn der Laufzeit nur vom Masterthread erstellt, wenn dieser entsprechende API-Aufrufe im parallelisiertem Programm ausführt. Ein solcher Task wird nach seiner Verteilung innerhalb des Laufzeitsystems durch einen der Threads ausgeführt. Der Anweisungsblock, welcher den Task darstellt, kann dabei selbst allerdings auch wieder eine Task-Kreierung beinhalten. Auf diese Weise können auch alle anderen Threads im LZS Tasks anfertigen. Nach seiner Erstellung wird ein Task in die private Warteschlange desjenigen Threads eingefügt, welcher die Erstellung durchgeführt hat.

3.2 Threads

Im LZS wird zwischen dem sogenannten *Masterthread* und den *Workerthreads* unterschieden. Der Masterthread ist derjenige Thread, welcher bereits zu Beginn eines parallelisierten Programms existiert und den dort sequentiellen Code ausführt. Er ruft von dort aus auch die Initialisierung sowie das Beenden des Laufzeitsystems auf. `NUM_THREADS-1`-viele Workerthreads werden im Zuge der Initialisierung vom Masterthread erstellt. Der Wert für die Anzahl aller Threads (inklusive des Masterthreads) ist dabei über die Umgebungsvariable `NUM_THREADS` einstellbar.

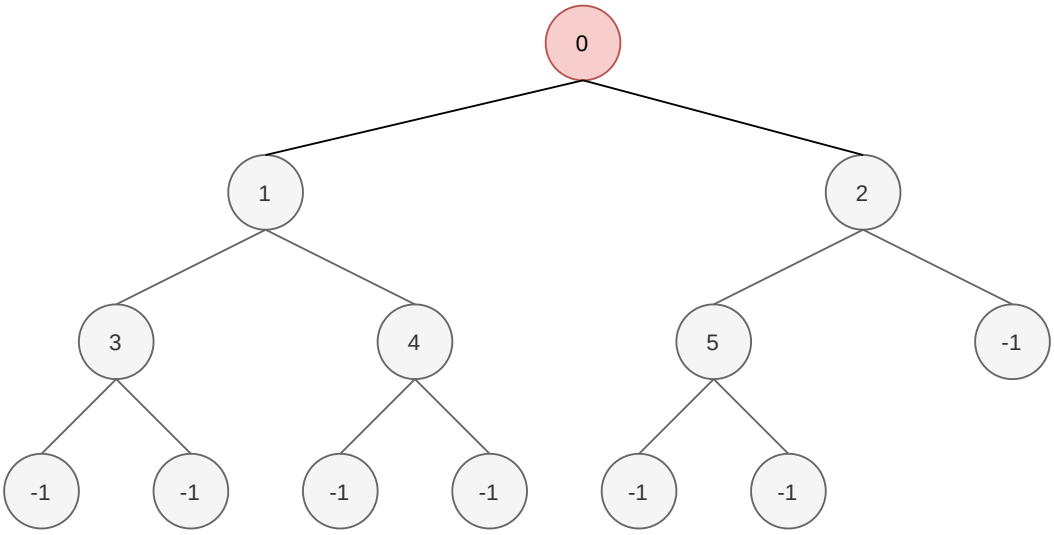

Abb. 3.1.: Thread Binärbaum bei sechs Threads

Alle Threads verfügen über eine einzigartige ID sowie über eine private Taskwarteschlange (siehe Kap. 2.1.3). Zudem besitzen sie Channels zum Empfangen von Stehlanfragen und Tasks (siehe Kap. 2.1.5). Jeder Thread ist anhand seiner ID in einem Binärbaum eingeordnet. Der Masterthread bildet den Wurzelknoten des Baums. Siehe dazu Abbildung 3.1, in welcher der Masterthread als roter Knoten und die Workerthreads als graue Knoten abgebildet sind. Die Zahl innerhalb eines Knotens entspricht der ID des Threads. Jedem Thread sind auf diese Weise zwei sog.

Kinder-Threads zugeordnet. Threads ohne Kinder besitzen im Baum zwei Blattknoten mit dem Wert -1.

3.3 Verteilte Warteschlangen

Zur Realisierung der Warteschlangen wurden im Tasking 2.0 LZS doppelt verkettete Listen gewählt, für welche ein Einfügen und Entfernen sowohl am Anfang als auch am Ende erlaubt ist. Diese sogenannten *Deques* (double-ended queues) werden im Laufe der Arbeit abweichend von der Definition als Warteschlangen bezeichnet.

Neue Tasks werden einer Warteschlange am Kopf hinzugefügt. Zur Abarbeitung von Tasks wird die Warteschlange am Kopf geleert. Auf das Ende der Datenstruktur wird zugegriffen, wenn der zugehörige Thread Stehlanfragen anderer Threads mit Tasks beantwortet (siehe dazu Kap. 3.4).

3.4 Work-Stealing

Im Tasking 2.0 LZS lässt sich zur Übersetzungszeit festlegen, zu welchem Zeitpunkt ein Thread eine Stehlanfrage versenden soll. Es besteht die Auswahl zwischen einem Stehlen, sobald die Warteschlange des Threads leer ist und einem Stehlen, sobald die Warteschlange nur noch N viele Tasks enthält. Letzteres entspricht einer sogenannten *Steal-early*-Strategie (dt.: Frühes Stehlen). Sie lässt sich mithilfe des Compiler-Option `STEAL_EARLY` aktivieren und der Wert von N lässt sich mithilfe der Option `STEAL_EARLY_THRESHOLD` festlegen.

Die Idee hinter dieser Strategie ist es, dass zwischen dem Senden einer Stehlanfrage und dem Zurückerhalten eines Tasks einige Zeit vergehen kann. Durch ein frühes Stehlen wird eine Anfrage bereits gesendet, sobald nur noch N Tasks in der Warteschlange verfügbar sind und eine leere Warteschlange daher in baldiger Zeit erwartet wird. Während der Zeit, in der die Stehlanfrage durch einen Opfer-Thread bearbeitet wird, kann der Dieb in diesem Fall noch die übrigen N Tasks ausführen und verbringt dadurch im Idealfall weniger Zeit mit Warten auf Threads.

Hat sich ein Thread nun dazu entschieden eine Stehlanfrage loszuschicken, so erstellt er dazu eine `steal_request`-Struktur (vlg. Listing 3.2) und füllt diese mit seiner eigenen Thread-ID im Element `ID`. Damit ein Opfer-Thread weiß, auf welchem Wege er Tasks zum Dieb zurücksenden soll, verweist der Dieb in `chan` noch auf einen extra dafür eingerichteten Channel. Das Element `state` enthält den aktuellen Status der Anfrage.

Mit dem Element `try` ist ein Zähler für die Anzahl an bereits durchgeführten Stehlversuchen realisiert. Um die Bedeutung dieses Zählers zu verstehen, muss man zuerst den Fall betrachten, in welchem ein Opfer-Thread eine Stehlanfrage nicht

mit einem Task beantworten kann. Dieser Fall tritt ein, wenn die Anfrage an einen Opfer-Thread gesendet wird, welcher zu diesem Zeitpunkt keine Tasks in seiner Warteschlange lagert. Der Opfer-Thread lehnt in diesem Zustand alle eintreffenden Anfragen ab. Würde der Opfer-Thread die Anfrage allerdings gleich abgelehnt an den Dieb zurücksenden, würde das Task-Bedürfnis des Diebes weiterhin bestehen bleiben. Der Dieb würde im Anschluss erneut eine Anfrage losschicken. Dieser Umweg wird im Tasking 2.0 eingespart, indem ein Opfer, welches die Anfrage nicht erfüllen kann, schon eigenständig ein neues Opfer auswählt und die Anfrage dorthin weiterleitet.

```
struct steal_request {
    Channel *chan;
    int ID;
    int try;
    unsigned int victims; // Bit-Feld
    state_t state;
    ...
};
```

Listing 3.2.: Gekürzte `steal_request` Struktur

Nun ist allerdings das Ereignis zu betrachten, in welchem keiner der Threads, welche diese Anfrage bearbeiten, in der Lage ist mit Tasks zu antworten. Hier würde es zu einem endlosen Weiterleiten der Stehlanfrage kommen. Durch den Zähler für die Anzahl an bereits durchgeführten Stehlversuchen wird dies verhindert. Sollten `MAX_STEAL_ATTEMPTS`-viele Versuche erreicht worden sein, wird die Anfrage kein weiteres Mal weiterverschickt. Die Anfrage gilt dann erst als endgültig gescheitert und wird zum Dieb zurückgesendet. Empfängt dieser eine Anfrage mit seiner eigenen ID als Ursprung, so weiß er über das Scheitern der Anfrage Bescheid. Der Wert von `MAX_STEAL_ATTEMPTS` lässt sich im LZS als Compiler-Option setzen.

Im Tasking 2.0 ist es zudem erlaubt, dass ein Thread Stehlanfragen sendet, während er noch auf die Antwort von anderen Stehlanfragen wartet. Dies kann nützlich sein, um schneller neue Tasks zu erhalten oder um mehr Tasks insgesamt zu stehlen. Die maximale Anzahl gleichzeitig aktiver Stehlanfragen wird durch die Compiler-Option `MAXSTEAL` eingestellt.

3.5 Auswahl eines Opfers

Stellt sich noch die Frage, auf welche Weise ein Opfer zum Stehlen gewählt wird. Der einfachste Ansatz wäre hierbei ein zufälliges Wählen aus allen Threads. Dabei kann es natürlich nach mehrfachem Weiterleiten einer Anfrage passieren, dass ein Opfer durch Zufall zum zweiten Mal ausgewählt wird. Der Füllstand der Warteschlange des Opfers wird sich in dieser kurzen Zeit allerdings nicht verändert haben. Zufälliges Wählen alleine stellt also kein effizientes Vorgehen dar.

Im Tasking 2.0 bedient man sich deshalb eines Bit-Feldes `victims` (vgl. Listing 3.2). Der Initialwert eines Bits für jeden Thread ist dabei Eins. Der Dieb setzt zu Beginn das ihn repräsentierende Bit auf Null. Ebenso setzt jeder Opfer-Thread, welcher die Anfrage aufgrund einer leeren Warteschlange weiterleiten muss, sein repräsentierendes Bit auf Null bevor er ein neues Opfer auswählt. Das Bit-Feld speichert also das Wissen der Anfrage darüber ab, bei welchen Thread ein Stehlen mit hoher Wahrscheinlichkeit fehlschlagen würde. Aus der Menge an potentiellen Opfern im Bit-Feld wird dann standardmäßig zufällig gewählt.

Das Tasking 2.0 bietet allerdings hierfür noch zwei Alternativen. Durch das Setzen des Compiler-Option `STEAL_LASTVICTIM` wird immer zuerst der Thread als nächstes Opfer ausgewählt, von welchem der aktuelle Thread in der Vergangenheit zuletzt erfolgreich stehlen konnte. Man erhofft sich bei diesem Opfer-Thread eine hohe Wahrscheinlichkeit, dass ein erneutes Stehlen wieder klappen wird.

Die zweite Alternative wird durch die Compiler-Option `STEAL_LASTTHIEF` aktiviert. Dabei wird als nächstes Opfer der Thread priorisiert, welcher vom aktuellen Thread als letztes erfolgreich gestohlen hat. Es wird dabei davon ausgegangen, dass dieser aufgrund des Stehlens in der Vergangenheit (vorzugsweise von mehreren Tasks) nun wieder Tasks in seiner Warteschlange lagert.

3.6 Channels

Alle aktuell im LZS enthaltenen Channel-Implementierungen legen eine Zielarchitektur zugrunde, welche über einen gemeinsamen Speicher für alle Threads verfügt. Sie unterscheiden sich durch die Anzahl von Sendern und Empfängern, welche einen Channel gemeinsam verwenden können.

- Über sog. **SPSC**-Channels (Single Producer Single Consumer, dt.: ein Erzeuger ein Verbraucher) kann nur ein Thread Nachrichten senden und nur ein Thread Nachrichten empfangen.
- Bei **MPSC**-Kanälen (Multy Producer Single Consumer, dt.: mehrere Erzeuger ein Verbraucher) können Nachrichten von mehreren Threads in den Channel eingefügt werden. Auch hier kann allerdings nur ein einziger Thread die Nachrichten empfangen.
- Die letzte hier implementierte Art ist ein **MPMC**-Channel (Multy Producer Multy Consumer, dt.: mehrere Erzeuger mehrere Verbraucher). Er bietet die Möglichkeit der Verwendung durch beliebig viele Threads sowohl beim Senden als auch beim Empfangen.

Die channelspezifischen Daten werden für alle drei Arten jeweils in der Struktur `SHM_channel` (siehe Listing 3.3) abgespeichert. Durch das Element `impl` wird zwischen SPSC, MPSC und MPMC unterschieden.

```
struct SHM_channel {
        pthread_mutex_t head_lock, tail_lock;
        int impl;
        unsigned int size;
        unsigned int itemsize;
        unsigned int head;
        unsigned int tail;
        char *buffer;
        ...
};
```

Listing 3.3.: Gekürzte `SHM_channel` Struktur

Zur Zwischenspeicherung von gesendeten Nachrichten innerhalb eines Channels wird eine FIFO-Warteschlange verwendet, welche als Array-Ringpuffer implementiert ist. Dort lassen sich N-viele Nachrichten mit maximaler Nachrichtengröße `itemsize` abspeichern. Eine Implementierung mit variabler Puffergröße ist nicht erforderlich, da die maximale Anzahl an Nachrichten, die gepuffert werden muss für jeden Channel im LZZ bekannt ist (siehe [Pre16]). Zum Zugriff auf die Warteschlange werden zwei Zähler in der Struktur abgelegt. Der Wert von `tail` notiert, an welcher Position im Ringpuffer die nächste Nachricht gespeichert werden kann. Soll hingegen eine Nachricht entnommen werden, so befindet sich diese an der Position von `head`. Der Zeiger `buffer` zeigt auf den Beginn des Puffers im Speicher. Die Größe des Puffers wird in `size` vermerkt.

Da bei den MPSC- und MPMC-Channels mehrere Threads gleichzeitig Nachrichten versenden bzw. versenden und empfangen können, muss der Zugriff auf die Datenstruktur vor Wettlaufsituationen geschützt sein. Dies wird durch eine Mutex-Variable für das Einfügen (`head_lock`) und eine für das Entfernen (`tail_lock`) von Nachrichten realisiert.

Wie bereits in Kapitel 3.4 erwähnt, werden Channels sowohl zum Versenden von Stehlanfragen, als auch zum Übermitteln von Tasks verwendet. Dazu existiert im Tasking 2.0 ein globales Array, in welchem zu jedem Thread ein Channel hinterlegt ist, durch welchen Stehlanfragen an den jeweiligen Thread versenden werden können. Da ein Thread potentiell Stehlanfragen von allen anderen Threads erhalten kann, sind die im Array abgelegten Channels vom Typ MPSC. Jeder Thread besitzt zudem ein Array mit mehreren SPSC-Channeln zum Empfangen von Tasks. Sollte ein Thread eine Stehlanfrage versenden, so referenziert er darin einen dieser SPSC-Channels. Sollte das Stehlen erfolgreich sein, erhält er über diesen Channel dann die Tasks des Opfers.

3.7 Scheduling Algorithmus

Um nun im nächsten Schritt verstehen zu können, wie die bereits vorgestellten Funktionalitäten im Tasking 2.0 LZS zusammenarbeiten, wird nun der zugrunde liegende Algorithmus zum Verteilen der Tasks im LZS genauer betrachten. Dafür wird zuerst ein grober Programmablaufplan des gesamten Laufzeitsystems von Programmstart bis -ende gezeichnet. Anschließend wird der darin enthaltene Scheduling-Algorithmus für Tasks genau erläutert.

In Figur 3.2 ist der Programmablauf des gesamten Tasking 2.0 dargestellt. Die Ausführungen des Masterthreads innerhalb des parallelisierten Programms sind in grün abgebildet. Wechselt der Masterthread durch die API-Aufrufe des Tasking 2.0 in das LZS so ist die Arbeit innerhalb des Laufzeitsystems in rot eingefärbt. Die Ausführungen der Workerthreads sind grau markiert. In dem Diagramm ist dabei exemplarisch der Ablauf eines einzelnen Workerthreads dargestellt.

Sobald in einem parallelisiertem Programm der API-Aufruf `TASKING_INIT` durch den dort einzigen Thread (den Masterthread) ausgeführt wird, wird das LZS durch diesen initialisiert. Dabei erzeugt der Masterthread gleich zu Beginn die Workerthreads. Parallel führen im Anschluss daran alle nun existierenden Threads noch weitere Initialisierungen durch, wie beispielsweise das Erstellen ihrer Channels für das Work-Stealing. Mithilfe einer Threadbarriere wird abschließend sichergestellt, dass alle Threads die aufgerufene Initialisierung beendet haben.

Die Workerthreads beginnen nach der Barriere mit dem Task-Scheduling-Algorithmus, während der Masterthread zunächst weiter das parallelisierte Programm durchläuft. Sollte er dabei auf einen der API-Aufrufe zur Task-Erstellung stoßen, führt er die dazugehörigen Funktionen im LZS aus und erstellt Tasks. Diese werden dann innerhalb des Laufzeitsystems (durch den Scheduling-Algorithmus) verteilt und abgearbeitet.

Erreicht der Masterthread stattdessen den API-Aufruf `TASKING_BARRIER`, wird eine Task-Barriere im LZS durchgeführt. Er kann das parallelisierte Programm erst wieder weiter ausführen, sobald alle vorher erstellten Tasks abgearbeitet sind.

Beim finalen API-Aufruf `TASKING_EXIT` soll das LZS beendet werden. Zu diesem Zweck sollten allerdings keine Tasks mehr existieren, weshalb der Masterthread vor dem eigentlichen Beenden des Laufzeitsystems intern noch eine Task-Barriere anstößt. Sobald diese beendet ist, werden allen Workerthreads über das Beenden benachrichtigt. Sie geben zur Beendigung nach einer weiteren Thread-Barriere ihren allokierten Speicher frei und terminieren sich anschließend selbst. Der Masterthread verlässt darauf folgend als einziger Thread das beendete LZS und bearbeitet das parallelisierte Programm noch solange weiter, bis auch dieses schlussendlich beendet ist.

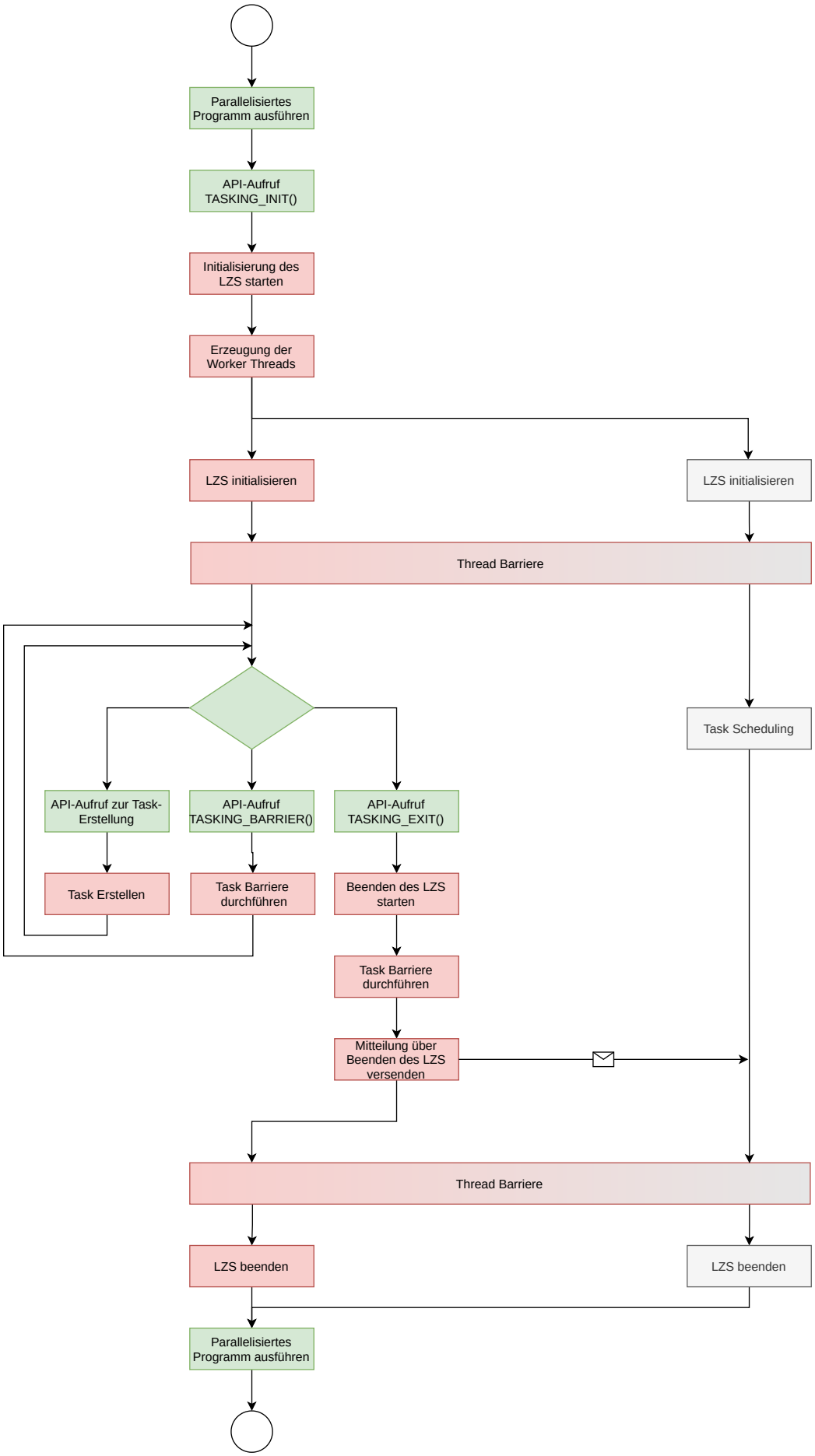

Abb. 3.2.: Allgemeiner Algorithmus des Laufzeitsystems

Im Folgenden soll nun genauer auf das eigentliche Task-Scheduling eingegangen werden, welches zwischen einem erfolgreichen Start der Laufzeit und dessen Beendigung stattfindet. In Abbildung 3.3 ist dazu in orange das Abarbeiten von Tasks aus der eigenen Warteschlange markiert. In den blau markierten Elementen wird auf das Empfangen neuer Tasks gewartet. In grau wird der Umgang mit Stehlanfragen dargestellt.

Alle Workerthreads befinden sich im Zuge ihres Scheduling-Algorithmus in einer Dauerschleife, bis sie die Benachrichtigung über das Beenden des Laufzeitsystems erhalten. Im ersten Schritt innerhalb dieser Schleife werden immer zuerst alle Tasks, welche sich in der jeweiligen privaten Warteschlange befinden, abgearbeitet. Hierzu wird je ein Task auf einmal entnommen. Bevor der Task allerdings ausgeführt wird, werden zuerst noch Stehlanfragen bearbeitet. Dafür wird überprüft, ob noch alte Stehlanfragen von Kinder-Threads abgespeichert sind und ob in dem Channel des Threads für Stehlanfragen neue Anfragen existieren. Solange noch weitere Tasks in der privaten Warteschlange vorhanden sind, werden all diese Anfragen mit den Tasks aus der Warteschlange beantwortet. Das Vorgehen bei einer leeren Warteschlange wird in Kapitel 3.8 beschrieben. Erst im Anschluss an die Behandlung der Stehlanfragen wird der zu Beginn entnommene Task vom Thread ausgeführt.

Sobald die private Taskwarteschlange leer ist, sendet der Thread eine Stehlanfrage an einen anderen Thread. Er befindet sich nun im sogenannten *idle*-Zustand (dt.: unbeschäftigt), in welchem er keine Arbeit im Sinne von Tasks durchführt. Er wartet in dieser Zeit auf eine Beantwortung seines losgeschickten Stehlversuches. In dieser Wartezeit lehnt er bei sich selbst ankommende Stehlanfragen ab. (Die Steal-Early-Strategie sowie das Senden mehrerer Stehlanfragen ist nicht in der Abbildung enthalten.)

Sollte seine Anfrage schließlich mit einem oder mehreren (Steal-Half) Tasks beantwortet worden sein, werden alle empfangenen Tasks - mit Ausnahme von einem - in die private Warteschlange eingefügt. Es wird überprüft, ob gemerkte Stehlanfragen von Kinder-Threads vorliegen. Falls ja, werden diese nach Möglichkeit mit Tasks aus der Warteschlange beantwortet. Im Anschluss wird der als einziger nicht in die Schlange eingefügte Task ausgeführt. (Im Diagramm ist das Einfügen aller Tasks mit Ausnahme von einem, durch ein Einfügen aller Tasks und anschließendem Entnehmen vereinfacht.)

Sollte in der Zwischenzeit das Beenden des Laufzeitsystems nicht durch den Masterthread initiiert worden sein, startet der Workerthread im Anschluss in die nächste Iteration der Scheduling-Schleife.

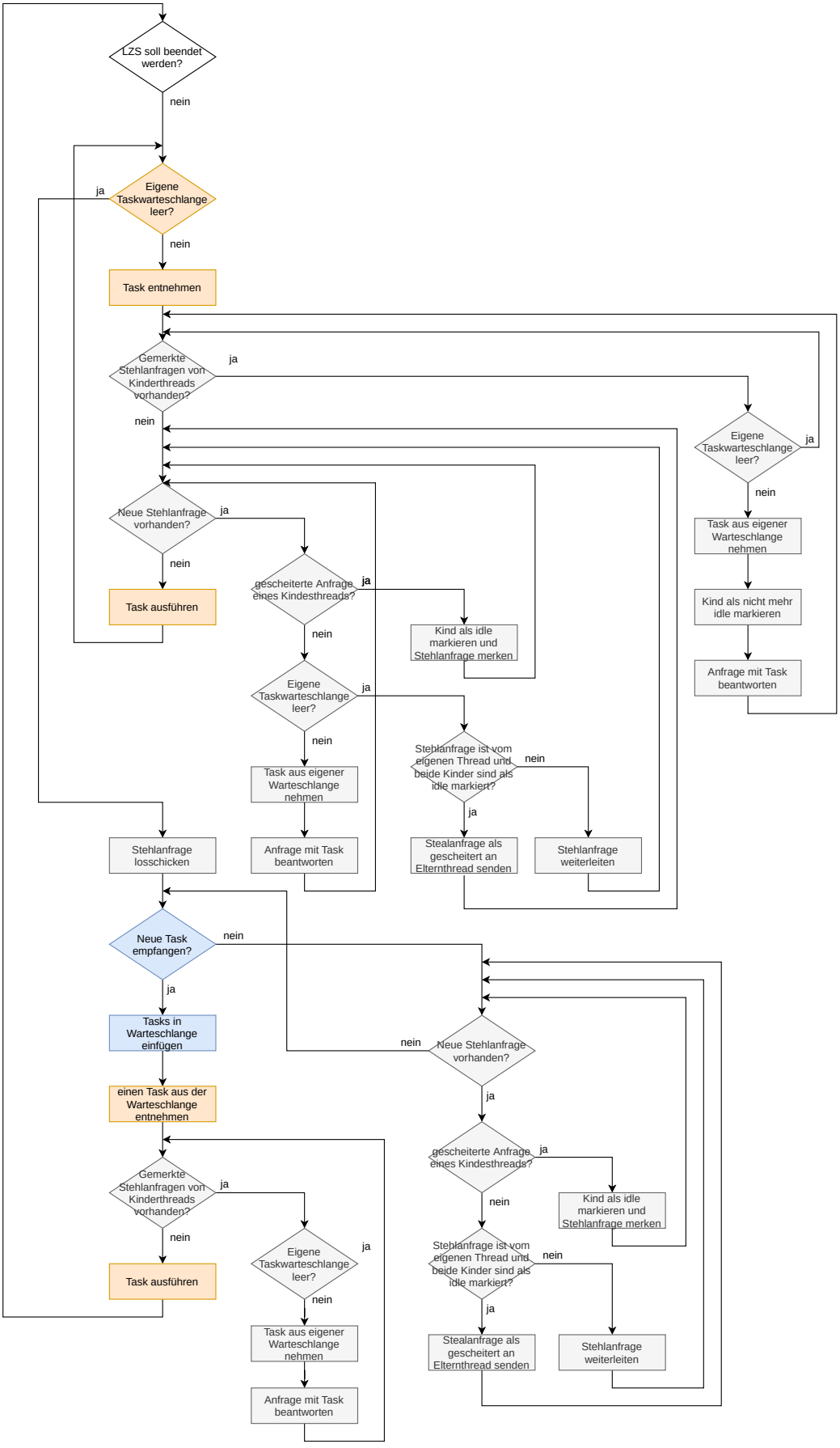

Abb. 3.3.: Scheduling-Algorithmus eines Workerthreads

3.8 Task-Barrieren

Zur Erfüllung einer Task-Barriere, muss der Masterthread in Erfahrung bringen können, ob sich noch unausgeführte Tasks im LZS befinden. Dabei liegt die Schwierigkeit darin, dass dieser nicht in die privaten Warteschlangen der Workerthreads schauen kann. Auch sieht er nicht, ob ein Thread gerade noch einen Task ausführt oder ob ein Task als Antwort auf eine Stehlanfrage gerade in einem Channel liegt. Aus diesem Grund geschieht der Informationsaustausch über den aktuellen Arbeitsstand der Threads indirekt über die Stehlanfragen.

Aus Kapitel 3.4 ist bekannt, dass eine gescheiterte Stehlanfrage zum Dieb zurückgesendet wird. Bearbeitet dieser eine solche Anfrage, so weiß er also über den fehlgeschlagenen Stehlversuch Bescheid. Nun gibt es die Möglichkeit, den Zähler der gescheiterte Anfrage zurückzusetzen und diese neu zu verschicken. Sollten im gesamten LZS allerdings wenige (/ keine) Tasks mehr existieren wird diese Anfrage mit hoher Wahrscheinlichkeit (/ sicher) erneut scheitern. Um dies zu vermeiden, zieht sich ein Thread vom Stehlen zurück, indem er keine weiteren Stehlversuche mehr unternimmt. Dies tut er genau dann, wenn er weiß, dass sich seine beiden Kinder im Thread-Baum (vlg. Abb. 3.1) bereits auch vom Stehlen zurückgezogen haben. Im Umkehrschluss bedeutet dies, dass die ersten Threads, welche sich vom Stehlen zurückziehen dürfen, Threads ohne Kinder sind.

Ein Zurückziehen vom Stehlen wird implementiert, indem ein Kinder-Thread, welcher seine eigene gescheiterte Stehlanfrage bekommt, diese als gescheitert markiert und an seinen Eltern-Thread sendet. Der Eltern-Thread speichert sich daraufhin die Information, dass sein entsprechendes Kind sich nun vom Stehlen zurückgezogen hat (idle-Status) und lagert zusätzlich die Stehlanfrage des Kindes ein. Existieren keine Tasks mehr im LZS, so findet schrittweise ein Zurückziehen beginnend von den untersten Threads im Thread-Baum bis hin zur Wurzel statt. Der Masterthread, welcher der Wurzel entspricht, kann schließlich aus einem idle-Zustand seiner beiden Kinder-Threads folgern, dass das gesamte LZS keine Tasks mehr enthält. Dadurch weiß er, wann eine Task-Barriere durchschritten werden darf.

Nachdem das gesamte LZS sich im idle-Zustand befand, kann nur der Masterthread wieder neue Tasks durch entsprechende API-Aufrufe aus dem parallelisiertem Programm erzeugen. Der Masterthread überprüft dann, ob er mit diesen Tasks die gespeicherten Anfragen seiner Kinder erfüllen kann. Wird eine solche Anfrage mit einem Task beantwortet, invalidiert er die bis dahin bei sich abgespeicherte Information über den idle-Status des Kindes. Auch alle Workerthreads überprüfen auf die selbe Weise in ihrem Scheduling-Algorithmus, ob noch gespeicherte Stehlanfragen von Kinder-Threads vorhanden sind, sobald wieder neue Tasks existieren (vgl. Abb. 3.3).

3.9 Einfügen eines Task

Nach dem Erstellen eines Tasks durch einen entsprechenden Aufruf aus der Tasking 2.0 API wird dieser in das LZS eingefügt. Dabei wird der Task immer zuerst in die private Warteschlange des jeweiligen Threads eingefügt. Da diese nun auf jeden Fall nicht leer ist, wird überprüft, ob Stehlanfragen mit den Tasks aus der Warteschlange beantwortet werden können. Dabei werden Anfragen von Kinder-Threads priorisiert behandelt. Sobald keine Tasks mehr in der privaten Warteschlange vorhanden sind, werden alle weitere Anfragen abgelehnt (bzw. an neue Opfer-Threads weiterverschickt).

3.10 Beenden des Laufzeitsystems

Das Beenden des Laufzeitsystems wird durch den API-Aufruf `TASKING_EXIT` angestoßen. Das parallelisierte Programm erwartet, dass alle Tasks, welche bis zu diesem Zeitpunkt erstellt wurden, im Anschluss an den Aufruf vollständig abgearbeitet sind. Aus diesem Grund wird intern zuallererst eine Task-Barriere durchgeführt (siehe Kap. 3.8).

Im Anschluss an die Barriere weiß der Masterthread über den idle-Zustand aller Threads Bescheid. Im nächsten Schritt muss er allerdings noch alle Workerthreads über das Beenden informieren. Zu diesem Zweck erstellt der Masterthread für jedes seiner Kinder je einen Schein-Task und versendet diese Tasks an sie. Ein solcher Schein-Task enthält keine Anweisungen aus dem parallelisiertem Programm. Stattdessen speichert er nur eine Funktion, welche ebenfalls wieder Schein-Tasks erstellt und diese an die jeweiligen Kinder-Threads weiterleitet. Mithilfe dieser Schein-Tasks wird ausschließlich die Information übermittelt, dass das LZS im beendet werden soll. Somit wird die Information im Thread-Baum von der Wurzel ausgehend bis hin zu den Blättern propagiert. Alle Workerthreads beenden daraufhin ihren Scheduling-Algorithmus und nehmen am Beenden des Laufzeitsystems teil (vlg. Fig. 3.2).

3.11 For-Schleifen

Schleifen, welche in jeder ihrer Iteration voneinander unabhängige Arbeit verrichten, sind ideal für eine Parallelisierung geeignet. Im Tasking 2.0 LZS lassen sich daher auch einfache For-Schleifen als Task repräsentieren (vgl. Kap. 3.1). Der Task speichert dabei den Start- und Endwert der Schleifenvariable. Zudem wird festgehalten, bis zu welcher Iteration der Schleife der Task bereits bearbeitet ist, sollte seine Abarbeitung bereits begonnen worden sein.

Während der Abarbeitung überprüft der ausführende Thread nach je einer Iteration, ob an ihn adressierte Stehlanfragen existieren. Ist dies der Fall und existieren zusätzlich dazu keine Tasks mehr in seiner Warteschlange, so versucht er den aktuell laufenden Schleifen-Task zu zerteilen. Falls noch mehr als eine Iteration der Schleife noch nicht bearbeitet wurde, beantwortet der Thread die Stehlanfrage mit einer Kopie des Schleifen-Tasks, mit entsprechend angepassten Start- und Endwerten für die Schleifenvariable. Der noch nicht bearbeitete Iterationsbereich der Schleife ist im Anschluss zwischen dem ursprünglichen Task und der Kopie aufgeteilt. Die Kopie wird dann als Antwort auf eine Stehlanfrage versendet.

An welcher Stelle diese Aufteilung geschieht, wird im LZS durch eine von drei verschiedenen Strategien bestimmt, welche mittels der Compiler-Option `SPLIT` festgelegt wird. Ist die Strategie `split_half` (dt.: Teile in der Hälfte) gewählt, so werden die noch nicht bearbeiteten Schleifeniterationen genau in der Hälfte geteilt, sodass im Anschluss beide Tasks in etwa die selbe Anzahl an Iterationen enthalten.

Ist `split_guided` gewählt, wird die Aufteilung in Abhängigkeit von der Anzahl Threads im LZS durchgeführt. Je mehr Threads existieren, desto weniger Iterationen erhält ein Dieb-Thread im neu erzeugten Schleifen-Task. Dies soll verhindern, dass bei mehreren Anfragen am Opfer-Thread der erste Dieb-Thread mit seiner Anfrage einen viel größeren Teil des Iterationsbereichs stiehlt als ein gleich darauf folgender zweiter Dieb-Thread.

Die dritte Strategie wird als `split_adaptive` bezeichnet und führt die Aufteilung in Abhängigkeit der Anzahl aktuell vorhandener Stehlanfragen beim Opfer-Thread durch. In der Regel entspricht die Anzahl der Threads, welche gerade beim Opfer-Thread zum Stehlen anfragen, nicht der Anzahl aller Threads im LZS. Daher erhofft man sich durch diese Strategie eine noch gleichmäßigere Aufteilung der Iterationen auf Threads.

In der Arbeit von A. Prell [Pre16] wird in Kapitel 5.2 auf die Performance in Abhängigkeit der Task-Granularität eingegangen. Die Granularität eines Tasks beschreibt dabei die Menge an Arbeit, welche innerhalb dieses Tasks zu erledigen ist. Erzeugt man in einer Schleife beispielsweise für jede Iteration einen eigenen Task, so erhält man eine große Anzahl feingranularer Tasks. Da das Erzeugen und Verwalten eines Tasks aber immer einen gewissen zeitlichen Aufwand mit sich bringt, erstellt das Tasking 2.0 zu Beginn nur einen einzigen, grobgranularen Task pro Schleife. Dieser Task wird erst dann in kleinere Tasks aufgeteilt, sobald dies auch durch Stehlanfragen gefordert wird. Auf diese Weise wird der Overhead so gering wie möglich gehalten, während trotzdem eine gute Lastbalancierung der Tasks auf die Threads erreicht wird. Dieses Vorgehen wird auch als *Lazy-Splitting* (dt: Faules Stehlen) bezeichnet.

4 Eingebaute OpenMP-Konstrukte und Laufzeitroutinen

In diesem Kapitel wird die Funktionalität der OpenMP-Konstrukte und Routinen erläutert, welche im Zuge dieser Arbeit in das Tasking 2.0 LZS eingebaut werden. Es wird erklärt, auf welche Weise das jeweilige Konstrukt dabei realisiert wurde und wie die Implementierung des ursprünglichen Tasking 2.0 LZS dafür angepasst bzw. erweitert werden musste.

Welche Teile der OpenMP-Schnittstelle dabei implementiert wurden, ist bereits in Kapitel 2.2.2 beschrieben. Ebenso wurde in Kapitel 2.2.3 erklärt, auf welche Weise der Clang-Compiler verwendet werden kann, um die OpenMP-Compiler-Direktiven in Code umzuwandeln, welcher Funktionen aus dem erweiterten LZS aufruft.

Für die Implementierung wurde sich dabei an den OpenMP-Laufzeitsystemen *libgomp* des GCC [GCC], *libomp* der LLVM [LLV] und *LOMP* von Cownie et al. [CK21b] orientiert. Die in diesem Kapitel auftretenden Definitionen von OpenMP-Direktiven und Laufzeitroutinen beziehen sich immer auf die jeweilige Spezifikation der OpenMP-API in der Version 5.2 [Ope21]. Die Beschreibung der vom Compiler eingefügten Funktionsaufrufe ist [LLV15] entnommen.

In der Tabelle in Abb. 4.1 ist eine Übersicht über die eingebauten und getesteten Konstrukte abgebildet. Alle implementierten Laufzeitroutinen sind in Abb. 4.2 zusammengefasst. Eine Aufstellung der verwendbaren Umgebungsvariablen bietet Abb. 4.3.

Konstrukt	Klauseln	Kapitel
parallel	if, num_threads, firstprivate, private, shared	4.3
master		4.5
single	nowait, firstprivate, private	4.6
task	firstprivate, private, shared	4.7
barrier		4.8.1
taskwait		4.8.2
taskgroup		4.8.3
for	schedule, firstprivate, private	4.9

Abb. 4.1.: Unterstützte OpenMP-Konstrukte

omp_get_thread_num	omp_get_num_threads
omp_get_max_threads	omp_set_num_threads
omp_get_dynamic	omp_set_dynamic
omp_get_schedule	omp_set_schedule
omp_get_wtime	omp_in_parallel

Abb. 4.2.: Unterstützte OpenMP-Laufzeitroutinen

OMP_NUM_THREADS	OMP_DYNAMIC
OMP_SCHEDULE	

Abb. 4.3.: Unterstützte OpenMP-Umgebungsvariablen

4.1 Starten des Laufzeitsystems

Das erweiterte LZS wird OpenMP-typisch nicht durch eine extra dafür vorgesehene Direktive oder Bibliotheksfunktion gestartet, sondern initiiert sich automatisch, sobald es zum ersten Mal im Ablauf eines parallelisierten Programms benötigt wird. In allen durch den Compiler eingefügten Funktionsaufrufen, welche ein gestartetes LZS für ihre korrekte Funktionalität benötigen, wird daher immer in einem ersten Schritt überprüft, ob eine Initiierung bereits stattgefunden hat. Ist dies nicht der Fall, wird zuerst der Start des Laufzeitsystems durchgeführt, bevor im Anschluss die eigentliche Funktionsimplementierung durchlaufen wird. Die vorgenommene Überprüfung vermeidet gleichzeitig auch eine Mehrfachinitiierung. Das Einfügen eines Makros durch den Benutzer, welches bisher zum Start des Tasking 2.0 benötigt wurde, entfällt nun durch diese Implementierung.

Ein Konstrukt, welches beispielsweise zwingend ein initiiertes LZS benötigt, ist das *parallel*-Konstrukt bzw. eine Parallelregion (siehe Kap. 4.3). Hier muss das LZS die Verwaltung der Threads übernehmen, welche für das Konstrukt benötigt werden. Auf der anderen Seite lässt sich beispielsweise die Funktionalität der OpenMP-Funktion `omp_get_wtime()`, welche die aktuelle Zeit in Sekunden zurückgibt, vollständig innerhalb ihres Funktionsrumpfes ohne Verwendung des Laufzeitsystems verwirklichen.

4.2 Beenden des Laufzeitsystems

Ebenso wie auch das Starten, soll auch das Beenden des Laufzeitsystems in der OpenMP-erweiterten Version des Tasking 2.0 ohne ein Zutun des Benutzers geschehen. Dies wird im Fall des Beendens durch die Funktion `atexit` aus der C-Standard-Bibliothek realisiert. Mithilfe dieser wird bereits im Initiierungsschritt des Laufzeitsystems eine eigens dafür erstellte Funktion `tasking_exit` registriert. Die Registrierung mittels `atexit` bewirkt, dass die Funktion ausgeführt wird, sobald das ausgeführte parallelisierte Programm regulär terminiert.

Innerhalb der registrierten Funktion wird dann ein kontrolliertes Aufräumen des Laufzeitsystems sichergestellt. Dazu zählt unter anderem das Einsammeln aller erstellter Threads, sowie das Freigeben von allokierten Speicherbereichen. Bei Bedarf können an dieser Stelle auch noch Statistiken zur Ausführung des Laufzeitsystems ausgegeben werden.

4.3 Parallel Konstrukt

Ein *parallel*-Konstrukt bzw. eine Parallelregion wird durch die Compiler-Direktive `pragma omp parallel` im Programmcode eingeleitet (vlg. dazu das Beispielprogramm 4.1). Auf die Direktive folgt ein Anweisungsblock, welcher nach der Definition der OpenMP-Schnittstelle von einer festgelegten Anzahl an Threads ausgeführt werden soll. Jeder Thread soll dabei exakt ein Mal den gesamten Block bearbeiten. Die Gruppe der Threads, welche eine solche Parallelregion gemeinsam ausführt wird als *Threadteam* bezeichnet. Sie lässt sich unterteilen in einen Masterthread (oder Primärthread), welcher bereits das parallelisierte Programm vor der *parallel*-Direktive ausgeführt hat, sowie beliebig viele weitere Workerthreads. Die Workerthreads arbeiten anders als der Masterthread nur innerhalb einer Parallelregion am Code des parallelisierten Programms mit.

```
int main(void) {
    int threads = 16;

    #pragma omp parallel if(threads>1) num_threads(threads)
    {
        /* Anweisungen */
    }
    return 0;
}
```

Listing 4.1.: OpenMP *parallel*-Konstrukt

Die Anzahl der Threads in einem Team wird durch Umgebungsvariabeln, Klauseln und zuvor aufgerufene Laufzeitroutinen bestimmt. Das Schema zur Berechnung ist dabei im OpenMP-Standard [Ope21] in Kapitel 10.1.1 vorgegeben. Im erweiterten Tasking 2.0 LZS, welches nur eine partielle OpenMP-Implementierung darstellt, wird die Anzahl Threads (im folgenden: `#threads`) wie folgt bestimmt:

1. Existiert zur *parallel*-Direktive eine *if*-Klausel, so wird zuerst der Ausdruck innerhalb der *if*-Klausel ausgewertet. Entspricht dieser Ausdruck Wahr, so wird die Parallelregion mit `#threads` ausgeführt (Variable wird in den nächsten Schritten gefüllt). Lässt sich der Ausdruck zu Falsch auswerten, so wird die Parallelregion auf jeden Fall sequentiell ausgeführt. Im sequentiellen Fall werden alle weiteren Schritte des Algorithmus ignoriert.

2. Falls eine *num_threads*-Klausel zur *parallel*-Direktive gehört, wird der Ausdruck innerhalb der Klammern ausgewertet und `#threads` auf den resultierenden Wert des Ausdrucks gesetzt. Springe bei vorhandener *num_threads*-Klausel weiter zu Schritt 6.

3. Falls im Programm vor der *parallel*-Direktive mindestens einmal die Laufzeitroutine `omp_set_num_threads` aufgerufen wird, so wird `#threads` auf den Wert gesetzt, welcher durch den letzten vorhergehenden Aufruf dieser Routine gesetzt wurde. Springe zu Schritt 6, wenn dieser Fall eintritt.

4. Ist die Umgebungsvariable `OMP_NUM_THREADS` gesetzt, so wird der gesetzte Wert für `#threads` verwendet. Der nächste Schritt wird ignoriert, wenn dieser Fall eintritt.

5. Da `#threads` nicht durch einen vorhergehenden Schritt gesetzt wurde, wird der Wert hier laufzeitspezifisch auf die Anzahl verfügbarer Prozessorkerne gesetzt, welche auf der ausführenden Maschine verfügbar sind.

6. Ist durch die Umgebungsvariable `OMP_DYNAMIC` oder die Routine `omp_set_dynamic` ein dynamisches Bestimmen der Threadanzahl vor der *parallel*-Direktive festgelegt worden, so setzt das LZS `#threads` auf die Anzahl der Prozessorkerne auf der ausführenden Maschine. Dieser neu bestimmte Wert darf allerdings den vorher festgelegten Wert von `#threads` nicht überschreiten.

7. Überschreitet `#threads` die im LZS festgelegte maximale Anzahl an möglichen Threads, so wird `#threads` auf die maximale Anzahl an Threads gekürzt.

Die OpenMP-Umgebungsvariable `OMP_NUM_THREADS` ersetzt dabei im Tasking 2.0 die bisherige Umgebungsvariable `NUM_THREADS`. Die im LZS neu hinzugefügte Threadteam-Datenstruktur `thread_team` ist in Listing 4.2 abgebildet. Diese speichert die Anzahl aller Threads des Teams im Element `num_threads`. Eine Threadbarriere des gesamten Teams kann mithilfe des Elements `team_barrier` durchgeführt werden. Mittels `single_count` wird die Anzahl bisher durchgeführter *single*-Direktiven mitgezählt (siehe Kap. 4.6). Der Zähler `task_count` vermerkt die Anzahl aller Tasks, welche aktuell noch unbearbeitet im Team existieren.

```
struct thread_team {
    int num_threads;
    pthread_barrier_t team_barrier;
    unsigned long single_count;
    atomic_t task_count;
}
```

Listing 4.2.: Struktur `thread_team`

Im nächsten Schritt wird nun ein Ausschnitt aus dem LLVM-Zwischencode 4.3 betrachtet, welcher mittels des Clang-Compilers aus dem Programm 4.1 erzeugt wurde.

Es lässt sich erkennen, dass der Anweisungsblock der Parallelregion aus der `main`-Funktion vom Compiler in die eigene Funktion `.omp_outlined.` ausgelagert wurde. Das Vorgehen ähnelt dabei sehr dem Vorgehen bei einem *task*-Konstrukt (siehe Kap. 4.7). In der OpenMP-API-Definition wird der ausgelagerte Anweisungsblock einer Parallelregion daher auch als impliziter (engl.: implicit) Task bezeichnet.

```
define i32 @main() #0 {
  %2 = alloca i32, align 4
  %3 = call i32 @__kmpc_global_thread_num(%ident_t* @0)
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 0, i32* %5, align 4
  store i32 16, i32* %2, align 4
  %6 = load i32, i32* %2, align 4
  call void @__kmpc_push_num_threads(%ident_t* @0, i32 %3, i32 %6)
  %8 = icmp sgt i32 %6, 1
  br i1 %8, label %9, label %10

; <label>:9:
  call void (%ident_t*, i32, void (i32*, i32*, ...)*, ...) @__kmpc_fork_call(
    %ident_t* @0, i32 0, void (i32*, i32*, ...)* bitcast (void (i32*, i32*)*
    @.omp_outlined. to void (i32*, i32*, ...)*))
  br label %11

; <label>:10:
  call void @__kmpc_serialized_parallel(%ident_t* @0, i32 %3)
  store i32 %3, i32* %4, align 4
  call void @.omp_outlined.(i32* %4, i32* %5) #1
  call void @__kmpc_end_serialized_parallel(%ident_t* @0, i32 %3)
  br label %11

; <label>:11:
  ret i32 0
}

define internal void @.omp_outlined.(i32* noalias, i32* noalias) #0 {
  ; Anweisungen
  ret void
}
```

Listing 4.3.: Gekürzter LLVM-Zwischencode für ein OpenMP *parallel*-Konstrukt
Ungekürzte Version in B.1

Man erkennt zudem, dass insgesamt fünf Funktionsaufrufe durch den Compiler eingefügt wurden, welche jeweils Funktionen aus dem LLVM-LZS libomp aufrufen sollen. Diese sind allesamt an ihrem Präfix `__kmpc_` auszumachen. Das erweiterte Tasking 2.0 LZS stellt daher auch Implementierungen dieser Funktionen bereit, welche mit den eingefügten Aufrufen beim Linken verknüpft werden (vlg. Kap. 2.2.3).

Der erste Aufruf `__kmpc_global_thread_num` gibt dabei die globale Thread-ID des aufrufenden Threads zurück. Eine exakte Definition der Funktion liefert [LLV15]. Die ID ist in allen Anwendungsfällen, welche vom erweiterten Tasking 2.0 bearbeitet

werden können, immer gleich mit der Rückgabe der Funktion `omp_get_thread_num` aus der OpenMP-API.

Die Thread-ID wird im Zwischencode anschließend der Funktion `__kmpc_push_num_threads` übergeben. Diese dient zur Realisierung der *num_threads*-Klausel aus dem ursprünglichen C-Programm. Der Wert der Variable `threads`, welche in den Klammern angegebenen ist, wird ebenfalls an den Funktionsaufruf weitergereicht. In der Funktionsimplementierung des erweiterten Tasking 2.0 wird dieser Wert für die folgende Parallelregion zwischengespeichert.

Die *if*-Klausel wird durch die Vergleichsoperation `icmp sgt` und die bedingte Sprungoperation `br` im Zwischencode implementiert. Der hier dargestellte Vergleich entspricht einer Auswertung des ursprünglichen Ausdrucks `(threads>1)`. Wird dieser Ausdruck zu Wahr ausgewertet, so bewirkt ein Sprung zu `label %9`, dass die Parallelregion parallel mit `#threads`-vielen Threads ausgeführt wird. Eine Auswertung zu Falsch endet mit einem Sprung zu `label %10` und einer sequentiellen Ausführung der Region.

Die sequentielle Ausführung wird bei der Sprungmarke 10 durch einen direkten Aufruf der ausgelagerten Funktion `.omp_outlined.` realisiert. Da hierdurch bereits alle vom OpenMP-Standard geforderten Funktionalitäten für diesen Fall implementiert sind, ist es nicht mehr notwendig in den beiden zusätzlich eingefügten Funktionsaufrufen `__kmpc_serialized_parallel` und `__kmpc_end_serialized_parallel` weitere Arbeit zu verrichten. Diese werden deshalb im erweiterten Tasking 2.0 LZS leer implementiert.

Bei der Sprungmarke 9 wird stattdessen der Fall eines parallelen Durchlaufens der Parallelregion realisiert. An dieser Stelle wurde der Aufruf `__kmpc_fork_call` vom Compiler eingefügt. Diesem wird unter anderem ein Zeiger auf die ausgelagerte Funktion übergeben. Es wird erwartet, dass dieser Aufruf innerhalb des Laufzeitsystems eine parallele Ausführung mit `#threads`-vielen Threads bewirkt. Um dieser Anforderung im erweiterten Tasking 2.0 gerecht zu werden, erstellt der Masterthread, welcher als einziger den Aufruf ausführt, ein Threadteam mit der dafür eingeführten Datenstruktur `thread_team`. Er erstellt dabei so viele Threads, sodass die nach dem obigen Algorithmus bestimmte Anzahl erreicht wird. Für jeden Thread des Teams initiiert er im Anschluss einen Task im LZS. Diese Tasks speichern dabei jeweils einen Funktionszeiger auf `.omp_outlined.` ab, sowie die Anzahl an Übergabeparametern für diese Funktion (vgl. Struktur 4.10). Eine Kopie der eigentlichen Übergabeparameter wird im Anschluss an jede Task-Struktur im Speicher abgelegt. Der Masterthread versendet je einen der Tasks an einen anderen Thread aus dem Team über dafür eingerichtete Channels. Im Anschluss daran führt er den für sich selbst erstellten Task aus.

Wie auch im OpenMP-Standard vorgegeben, wird schlussendlich zum Beenden der Parallelregion eine Task- und Threadbarriere durchgeführt. Die Implementation einer solchen Barriere ist dabei in Kapitel 4.8 beschrieben. Sie bezweckt, dass der Masterthread sicherstellen kann, dass alle Tasks der Parallelregion ausgeführt worden sind und die Parallelregion daraufhin für beendet erklärt werden kann. Das Ende der Parallelregion teilt der Masterthread den anderen Threads des Teams durch das Versenden der Nachricht `RT_THREAD_BARRIER_AND_EXIT_PARALLEL` mit (siehe Kap. 4.11). Er kehrt daraufhin aus dem Aufruf von `__kmpc_fork_call` zurück und führt den sequentiellen Teil des Programms im Anschluss an das *parallel*-Konstrukt weiter aus. Dieser Code befindet sich im LLVM-Zwischencode bei der Sprungmarke 11, zu welcher sowohl nach einem parallelem als auch nach einem sequentiellem Durchlaufen gesprungen wird.

Die erstellten Threads werden im erweiterten Tasking 2.0 nicht am Ende der Region wieder zerstört, sondern warten bis zum Beenden des Laufzeitsystems auf das Eintreten weiterer Parallelregionen. Sollte nämlich erneut eine solche betreten werden, hat dieses Vorgehen den Vorteil, dass die bereits erstellten Threads wiederverwendet werden können. Auf diese Weise kann der große Laufzeit-Overhead für das Erstellen und Beenden von Threads ab der zweiten Parallelregion vermindert werden. Nachteil hiervon ist aber, dass erstellte Threads auch während einer sequentiellen Phase des Programms Rechenzeit auf den Kernen einnehmen, da sie in einer Schleife *busy* (dt.: beschäftigt) auf die Ankunft der nächsten Region warten.

Im Vergleich zu der ursprünglichen Implementiation im Tasking 2.0, werden Threads nun also nicht mehr zwingend gleich zu Beginn des Laufzeitsystems gestartet, sondern erst dann, wenn auch eine Parallelregion eintritt und dadurch die Anzahl benötigter Threads bekannt ist. Beide Ereignisse können allerdings immer noch aufeinanderfolgend geschehen, wenn das LZS beim Betreten eines *parallel*-Konstrukts noch nicht initiiert ist. Siehe dazu Kapitel 4.1.

Aus Gründen der Komplexität unterstützt das erweiterte Tasking 2.0 LZS keine geschachtelten Parallelregionen.

4.4 Umgebungsvariablen

Wie bereits im vorausgehenden Kapitel 4.3 kennengelernt, werden zwei Umgebungsvariablen zur Festlegung der Thread-Anzahl im erweiterten LZS unterstützt. Eine weitere Variable steuert das Scheduling-Verhalten bei For-Schleifen (vgl. Kap. 3.11). Der Wert dieser Variablen wird gleich zu Beginn des Laufzeitsystems ausgelesen und abgespeichert. Spätere Werteveränderungen der Umgebungsvariablen werden gemäß des OpenMP-API-Standards nicht beachtet. Auf Unix-Betriebssystemen lassen sich Umgebungsvariablen beispielsweise durch den Befehl `export UMGEBUNGSVARIABLE=wert` setzen. Zu jeder der Variablen existieren auch Laufzeitroutinen zum Setzen und

Auslesen der im LZS abgespeicherten Werte. Folgende Liste gibt einen detaillierten Überblick über die implementierten Umgebungsvariablen:

- `OMP_NUM_THREADS`: Der Wert dieser Variablen wird zur Bestimmung der Threadanzahl in Parallelregionen verwendet (siehe Kap. 4.3). Nur echt positive Ganzzahlen sind dabei erlaubt. Mit der Laufzeitroutinen `omp_get_num_threads` lässt sich der im LZS gesetzte Wert zur Laufzeit auslesen. Mit `omp_set_num_threads` lässt sich dieser verändern.
- `OMP_DYNAMIC`: Auch diese Variable wird zur Bestimmung der Threadanzahl in Parallelregionen verwendet (siehe Kap. 4.3). Als Werte sind ausschließlich `true` oder `false` erlaubt. Mit der Laufzeitroutine `omp_get_dynamic` lässt sich der im LZS gesetzte Wert zur Laufzeit auslesen. Mit `omp_set_dynamic` lässt sich dieser verändern.
- `OMP_SCHEDULE`: Mit dieser Variable wird ein Scheduling-Verfahren für For-Schleifen festgelegt. Dieser wird dann verwendet, wenn das Scheduling-Verfahren für eine Schleife noch nicht zur Übersetzungszeit feststeht, sondern erst zur Laufzeit gewählt wird (vlg. Kapitel 3.11). Aktuell unterstützte Werte hierfür sind: `static`, `dynamic`, `guided` und `auto`. Mit `omp_get_schedule` lässt sich das gesetzte Scheduling-Verfahren zur Laufzeit auslesen. Mit `omp_set_schedule` lässt sich dieses verändern.

4.5 Master Konstrukt

Ein Anweisungsblock, welcher mit `pragma omp master` markiert ist, darf - dem OpenMP-Standard nach - nur durch den Masterthread des Teams betreten werden, welches das Konstrukt ausführt. Außerhalb einer Parallelregion wird der Anweisungsblock von dem Masterthread bearbeitet, welcher dort als einziger Thread existiert. Das Konstrukt zählt im OpenMP-5.2-Standard als veraltet. Da es allerdings auch von LLVM's libomp unterstützt wird und immer noch in vielen parallelisierten Programmen zur Anwendung kommt, ist es auch im erweiterten Tasking 2.0 implementiert.

Im Programm 4.4 ist ein Beispielcode für das Konstrukt gegeben. In realen Anwendungen würden das Konstrukt innerhalb von Parallelregionen verwendet werden. Die Umwandlung in Zwischencode ist jedoch die selbe, wie in dem weniger komplexen Beispielfall. Der aus dem Programm erzeugte Zwischencode in Listing 4.5 fragt die globale Thread-ID des ausführenden Threads wie bereits in Kapitel 4.3 mittels `__kmpc_global_thread_num` ab. Der Rückgabewert wird an den Aufruf `__kmpc_master` weitergeleitet. Dieser ist im erweiterten LZS dazu programmiert worden, den Wert `1` zurückzuliefern, falls der Thread, welcher den Funktionsaufruf tätigt,

der Masterthread ist. Bei allen anderen Threads wird stattdessen `0` zurückgegeben. Intern ist dies durch eine einfache Überprüfung der Thread-ID realisiert.

```
int main(void) {

    #pragma omp master
    {
        /* Anweisungen */
    }
    return 0;
}
```

Listing 4.4.: OpenMP *master*-Konstrukt

```
define i32 @main() #0 {
  %2 = call i32 @__kmpc_global_thread_num(%ident_t* @0)
  %3 = call i32 @__kmpc_master(%ident_t* @0, i32 %2)
  %4 = icmp ne i32 %3, 0
  br i1 %4, label %5, label %6

; <label>:5:
  ; Anweisungen
  call void @__kmpc_end_master(%ident_t* @0, i32 %2)
  br label %6

; <label>:6:
  ret i32 0
}
```

Listing 4.5.: Gekürzter LLVM-Zwischencode für ein OpenMP *master*-Konstrukt
Ungekürzte Version in B.2

Im Zwischencode wird der Rückgabewert anschließend mithilfe der Operation `icmp ne` mit `0` verglichen. Wird der Code vom Masterthread ausgeführt, schlägt der Vergleich also fehl und es wird bei Sprungmarke 5 weitergearbeitet. Der Code dort entspricht dem Anwendungsblock innerhalb des *master*-Konstrukts. Im Anschluss wird noch die Funktion `__kmpc_end_master` aufgerufen. Diese ist im LZS allerdings leer implementiert, da alle Funktionalitäten des Konstruktes bereits vollständig erfüllt wurden. Nach dem Funktionsaufruf wird schließlich der an das Konstrukt folgende Code weiter ausgeführt. Dieser befindet sich bei Sprungmarke 6.

Da bei allen anderen Threads der Vergleich mit `0` Wahr liefert, springen diese direkt zur Sprungmarke 6 und überspringen dadurch die Anweisungen innerhalb des *master*-Konstrukts.

4.6 Single Kontrukt

Ähnlich zum *master*-Konstrukt in Kapitel 4.5, bewirkt auch ein mit `pragma omp single` markierter Anweisungsblock, dass dieser nur von genau einem Thread des

Teams bearbeitet wird. Anders als beim *master*-Konstrukt, ist dieser Thread hier allerdings nicht zwingend der Masterthread sondern ein beliebiger Thread. Auch in zwei aufeinanderfolgenden Programmdurchläufen muss der Anweisungsblock nicht beide Male von dem selben Thread ausgeführt werden. Die einzige Vorgabe des OpenMP-Standards besteht darin, dass genau ein Thread aus dem ausführenden Threadteam den Codeblock innerhalb des *single*-Konstrukts bearbeitet. Das Vorgehen bei der Auswahl eines Threads ist daher laufzeitsystemspezifisch.

```
int main(void) {

    #pragma omp single
    {
        /* Anweisungen */
    }
    return 0;
}
```

Listing 4.6.: OpenMP *single*-Konstrukt

```
define i32 @main() #0 {
  %2 = call i32 @__kmpc_global_thread_num(%ident_t* @0)
  %3 = call i32 @__kmpc_single(%ident_t* @0, i32 %2)
  %4 = icmp ne i32 %3, 0
  br i1 %4, label %5, label %6

; <label>:5:
  ; Anweisungen
  call void @__kmpc_end_single(%ident_t* @0, i32 %2)
  br label %6

; <label>:6:
  call void @__kmpc_barrier(%ident_t* @1, i32 %2)
  ret i32 0
}
```

Listing 4.7.: Gekürzter LLVM-Zwischencode für ein OpenMP *single*-Konstrukt
Ungekürzte Version in B.3

Auch der produzierte Zwischencode 4.7 aus einem Beispielprogramm mit *single*-Konstrukt 4.6 ähnelt sehr dem Zwischencode eines *master*-Konstrukts (vgl. Listing 4.5). Auch hier ist der gezeigte Code nur als Beispiel zur Erklärung der Arbeit des Compilers zu verstehen, da auch das *single*-Konstrukt i.d.R. nicht außerhalb von Parallelregionen zur Anwendung kommt. Im Zwischencode wird wieder mit einer Abfrage der globalen Thread-ID gestartet, welche nun allerdings an den Aufruf `__kmpc_single` übergeben wird. Ein Rückgabewert von `1` entscheidet auch hier über die Ausführung des Anweisungsblocks innerhalb des Konstrukts. Die Anweisungen befinden sich hier bei Sprungmarke 5. Eine Rückgabe von `0` auf der anderen Seite lässt den aufrufenden Thread den genannten Anweisungsblock überspringen.

Die Funktion `__kmpc_single` ist im erweiterten LZS so implementiert, dass sie immer `1` zurückgibt, falls der Aufruf außerhalb einer Parallelregion stattfindet. Dort existiert nämlich nur ein einziger Thread. Befindet man sich stattdessen in einer Parallelregion, versucht jeder Thread des Teams der Parallelregion durch den Aufruf von `__kmpc_single` Zugang zum Anweisungsblock zu erhalten. Es ist dabei im Allgemeinen die beste Laufzeit des Programms zu erwarten, wenn der erste Thread, welcher das *single*-Konstrukt erreicht, auch den Codeblock darin ausführen darf. Dieses Vorgehen ist daher im erweiterten LZS implementiert.

Jeder Thread zählt dazu in einem privaten Zähler die *single*-Konstrukte mit, denen er begegnet. Dies passiert durch ein Inkrementieren des Zählers um `1` zu Beginn der Funktion. Im Anschluss vergleicht der Thread mittels einer atomaren `__sync_bool_compare_and_swap`-Operation den Wert der Variable `single_count` aus der Struktur des Threadteams mit seinem privaten *single*-Konstrukt-Zähler. Gelingt der Vergleich, wird `single_count` noch innerhalb der selben atomaren Operation gegen den privaten Zähler addiert mit `1` ausgetauscht. Auf diese Weise scheitert ein Vergleich bei allen nachfolgenden Threads. Der Vergleich gelingt also für ein *single*-Konstrukt immer nur demjenigen Thread, welcher die atomare Operation als Erster ausführt.

Auf die Funktion `__kmpc_end_single` entfallen auch hier keine weiteren Aufgaben mehr, sodass sie im erweiterten Tasking 2.0 LZS leer implementiert ist.

Zusätzlich fordert der Standard am Ende des *single*-Konstruktes noch eine Task- und Threadbarriere für das gesamte Team. Diese wird in der Funktion `__kmpc_barrier` im LZS implementiert. Der Compiler fügt einen Aufruf dazu bei der Sprungmarke 6 ein, sodass sie von allen Threads des Teams am Ende des Konstrukts aufgerufen wird. Da eine solche Barriere auch als eigene Direktive existiert wird hier auf die Erläuterung ihre Realisierung in Kapitel 4.8 verwiesen. Die Barriere kann durch ein Anfügen der *nowait*-Klausel an die Compiler-Direktive entfernt werden.

4.7 Task Konstrukt

Mithilfe der Direktive `pragma omp task` lässt sich durch den Nutzer ein Anweisungsblock als Task definieren. Dieser wird dann vom LZS verpackt und in eine Warteschlange eingegliedert. Wird der Task innerhalb einer Parallelregion definiert, so wird er von einen vorher nicht festgelegten Thread des zugehörigen Threadteams abgearbeitet.

Betrachtet man den LLVM-Zwischencode 4.9, welcher aus dem C-Code 4.8 erstellt wurde, erkennt man, dass der Anweisungsblock innerhalb des *task*-Konstrukts in eine neue Funktion `.omp_task_entry.` ausgelagert wurde. Ein Zeiger auf diese Funktion

wird dem eingefügten Aufruf `__kmpc_omp_task_alloc` zusammen mit der globalen Thread-ID übergeben.

```
int main(void) {

    #pragma omp task
    {
        /* Anweisungen */
    }
    return 0;
}
```

Listing 4.8.: OpenMP *task*-Konstrukt

```
define i32 @main() local_unnamed_addr #0 {
  %1 = tail call i32 @__kmpc_global_thread_num(%ident_t* nonnull @0) #2
  %2 = tail call i8* @__kmpc_omp_task_alloc(%ident_t* nonnull @0, i32 %1, i32 1,
    i64 40, i64 0, i32 (i32, i8*)* bitcast (i32 (i32,
    %struct.kmp_task_t_with_privates*)* @.omp_task_entry. to i32 (i32, i8*)*)) #2
  %3 = tail call i32 @__kmpc_omp_task(%ident_t* nonnull @0, i32 %1, i8* %2) #2
  ret i32 0
}

define internal i32 @.omp_task_entry.(i32, %struct.kmp_task_t_with_privates*
    noalias nocapture readnone) #1 {
  ; Anweisungen
  ret i32 0
}
```

Listing 4.9.: Gekürzter LLVM-Zwischencode für ein OpenMP *task*-Konstrukt
Ungekürzte Version in B.4

Die Implementierung der Funktion `__kmpc_omp_task_alloc` im erweitertem LZS bewirkt die Allokation und Erstellung eines Tasks, jedoch noch keine Verteilung und Ausführung. Intern wird in der Funktion auf dem Heap-Speicher Platz für die *task*-Struktur des Laufzeitsystems allokiert. Der Aufbau der ursprünglichen *task*-Struktur des Tasking 2.0 Laufzeitsystems (siehe Listing 3.1) wurde im Zuge dieser Arbeit angepasst und ist abgebildet in Listing 4.10.

Ein Task ist dabei aufgegliedert in die Struktur `task_metadata_t` mit Metadaten zum Task und einer Struktur `task_closure_t`, welche in ihrem Aufbau durch die OpenMP-Schnittstelle des LLVM-Compilers vorgegeben ist. In den Metadaten werden in der Erweiterung des Tasking 2.0 nun zusätzlich Informationen zur sog. Taskgruppe (engl.: taskgroup) eines Tasks abgespeichert (siehe Kap. 4.8.3). Außerdem wird mittels des Elements `fn_style` zwischen den verschiedenen Task-Typen unterschieden. Dabei differenziert man zwischen Tasks, welche durch ein *task*-Konstrukt erzeugt werden (`LLVMTaskingStyle`), Tasks einer Parallelregion (`LLVMParallelStyle` vgl. Kap. 4.3) und Tasks, welche Iterationen einer For-Schleife repräsentieren (`LLVMLoopStyle` vgl. Kap. 4.9). Ein zusätzlich eingefügter Zähler `child_tasks` hilft dabei, die Anzahl noch nicht beendeter Kinder-Tasks für das Konstrukt *taskwait* abzuspeichern

(siehe Kap. 4.8.2). In der vorgegebenen Struktur `task_closure_t` wird im Element `routine` der Zeiger auf die ausgelagerte Funktion eines Tasks abgelegt.

```
struct task {
    task_metadata_t metadata;
    task_closure_t closure;
    // Private Variablen im Anschluss an diese Struktur (optional)
    // Geteilte Variablen im Anschluss an diese Struktur (optional)

};

struct task_metadata {
    struct task *parent;
    taskgroup_t *taskgroup;
    atomic_t child_tasks;
    enum fn_style {
        LLVMTaskingStyle, LLVMParallelStyle, LLVMLoopStyle
    } fn_Calling_Style;
    ...

    // Ausschließlich für LLVMParallelStyle Tasks:
    int argc;
    void (*fn)(void *);

    // Ausschließlich für LLVMLoopStyle Tasks:
    enum loop_dt {
        LOOP4, LOOP4U, LOOP8, LOOP8U
    } loop_datatype;
    union loop_int start;
    union loop_int end;
    bool splittable;
};

struct task_closure {
    void *shareds;
    kmp_routine_entry_t routine;
    ...
};
```

Listing 4.10.: Aufbau einer *task*-Struktur

Die beschriebene Funktion `__kmpc_omp_task_alloc` gibt schließlich einen Zeiger auf den allokierten Task zurück. Falls Daten-Klauseln zur Direktive vorhanden sind, würden in einem darauf folgenden Schritt die darin verwendeten Variablen zum Task abgespeichert werden (siehe Kap. 4.10). Im Beispielprogramm ist dies nicht der Fall, weshalb gleich im nächsten Schritt der Task an den eingefügten Aufruf `__kmpc_omp_task` übergeben wird. Die Implementation dieser Funktion bewirkt, dass für den Task nun eine Verteilung und Abarbeitung durch das LZS stattfindet (siehe Kap. 3.9). Ein vorher nicht festgelegter Thread aus der zugehörigen Parallelregion wird den Task schließlich bearbeiten. Wird der Task außerhalb einer Parallelregion erstellt, wird eine sofortige sequentielle Abarbeitung durch den Masterthread angestoßen.

4.8 Barrieren

Innerhalb der OpenMP-API existieren verschiedene Möglichkeiten, auf die Beendigung von Tasks zu warten. Sie unterscheiden sich darin, auf welche Tasks gewartet wird.

4.8.1 Barrier Konstrukt

Wird eine Barriere explizit durch den Programmierer mithilfe der Direktive `pragma omp barrier` in eine Parallelregion eingefügt, so schreibt der OpenMP-Standard vor, dass alle Threads der Parallelregion diese Barriere ausführen müssen und alle innerhalb der Region erstellten Tasks vor einem Verlassen des *barrier*-Konstruktes abgearbeitet sein müssen. Die impliziten Tasks vom Typ `LLVMParallelStyle`, welche zu Beginn einer Parallelregion erstellt wurden sind hierbei von dieser Task-Barriere ausgenommen.

Der Compiler fügt für das Konstrukt an der Stelle der Direktive den Funktionsaufruf `__kmpc_barrier` ein, sowie (falls noch nicht ermittelt) einen Aufruf zur Bestimmung der globalen Thread-ID des aufrufenden Threads. Ein einfacher Beispielcode für eine Barriere ist in Listing 4.11 abgebildet. Auch wenn die Barriere im Beispiel außerhalb einer Parallelregion steht, sind die durchgeführten Umwandlungen in Zwischencode identisch. Der resultierende Zwischencode ist in Listing 4.12 dargestellt.

```
int main(void) {
    #pragma omp barrier
    return 0;
}
```

Listing 4.11.: OpenMP *barrier*-Konstrukt

```
define i32 @main() local_unnamed_addr #0 {
  %1 = tail call i32 @__kmpc_global_thread_num(%ident_t* nonnull @1) #1
  tail call void @__kmpc_barrier(%ident_t* nonnull @0, i32 %1) #1
  ret i32 0
}
```

Listing 4.12.: Gekürzter LLVM-Zwischencode für ein OpenMP *barrier*-Konstrukt
Ungekürzte Version in B.5

Diese Art der Barriere wird von allen Threads des Teams bearbeitet. Der Masterthread wartet wie schon im Tasking 2.0 LZS darauf, dass seine beiden Kinder-Threads einen *idle*-Status erreichen (vgl. Kap. 3.8). Ist dies der Fall, weiß er, dass keine weiteren Tasks mehr im LZS enthalten sind. Im erweiterten LZS informiert er die anderen Threads des Teams durch das Senden der Nachricht `RT_EXIT_TASK_BARRIER` darüber, dass die Task-Barriere verlassen werden darf.

Betrachtet man die Abarbeitung des Konstrukts hingegen aus der Sicht eines Workerthreads, so führt ein solcher einen Scheduling-Algorithmus ähnlich zu Abb. 3.3 aus. Dieser terminiert, sobald die Nachricht `RT_EXIT_TASK_BARRIER` empfangen wird.

Sowohl Worker- als auch Masterthread führen im Anschluss an die Task-Barriere noch eine Thread-Barriere durch, bevor sie schließlich aus der Funktion `__kmpc_barrier` zurückkehren.

4.8.2 Taskwait Konstrukt

Bei der Direktive `pragma omp taskwait` wird nicht auf die Beendigung aller Tasks gewartet, sondern nur auf die Fertigstellung aller Kinder-Tasks desjenigen Tasks, in wessen Anweisungsblock das *taskwait*-Konstrukt steht. Dieser Task kann dabei auch ein impliziter Task einer Parallelregion sein, wenn das Konstrukt innerhalb des Anweisungsblocks eines *parallel*-Konstruktes steht.

Um diese Vorgaben des Standards zu erfüllen, speichert im erweiterten Tasking 2.0 LZS ein Task in seinen Metadaten mittels des Zählers `child_tasks` die Anzahl seiner Kinder-Tasks (siehe Struktur 4.10). Wird während der Abarbeitung dieses Tasks ein neuer Task erstellt, wird der Zähler inkrementiert. Nach dem Beenden jedes Tasks wird zudem überprüft, ob jeweils ein Eltern-Task existiert und falls ja, wird der `child_tasks`-Zähler des Eltern-Tasks dekrementiert.

Die Listings 4.13 und 4.14 sind dabei wieder nur Beispiele, welche ausschließlich die eingefügten Aufrufe durch den Compiler zeigen sollen. Im LLVM-Zwischencode ist erkenntlich, dass neben einem Aufruf zur Ermittlung der globalen Thread-ID auch ein Aufruf `__kmpc_omp_taskwait` eingefügt wurde. Im erweiterten Tasking 2.0 LZS wird innerhalb dieser Funktion vom aufrufenden Thread eine Abwandlung des Scheduling-Algorithmus aus Abb. 3.3 durchgeführt. Dabei überprüft der Thread allerdings zusätzlich in regelmäßigen Abständen, ob der entsprechende Kinder-Task-Zähler in der Zwischenzeit den Wert `0` erreicht hat und beendet in diesem Fall den Algorithmus.

```
int main(void) {
    #pragma omp taskwait
    return 0;
}
```

Listing 4.13.: OpenMP *taskwait*-Konstrukt

Eine Dekrementierung des Zählers kann dabei durch die Ausführung der Kinder-Tasks erzielt werden, welche sich in der privaten Warteschlange des Threads befinden. Auch durch Stehlanfragen kann der Thread an Kinder-Tasks gelangen. Neben dem Thread, welcher das Konstrukt bearbeitet, können allerdings auch alle anderen

Threads des Threadteams in der Vergangenheit durch Stehlen an Kinder-Tasks gelangt sein. Auf das Scheduling der anderen Threads wird allerdings nicht eingewirkt. Befinden sich entsprechende Kinder-Tasks in stark ausgelasteten Warteschlangen von anderen Threads kann es deshalb mitunter zu großen Wartezeiten kommen.

```
define i32 @main() local_unnamed_addr #0 {
  %1 = tail call i32 @__kmpc_global_thread_num(%ident_t* nonnull @0) #1
  %2 = tail call i32 @__kmpc_omp_taskwait(%ident_t* nonnull @0, i32 %1) #1
  ret i32 0
}
```

Listing 4.14.: Gekürzter LLVM-Zwischencode für ein OpenMP *taskwait*-Konstrukt
Ungekürzte Version in B.6

Wurden alle Kinder-Tasks beendet, so kehrt der aufrufende Thread aus der Funktion `__kmpc_omp_taskwait` zurück und führt die umschließende Anweisungssequenz weiter aus. Im Sonderfall, dass die *taskwait*-Struktur außerhalb eines Tasks angewendet wird, kann die Funktion sofort beendet werden. Dies gilt auch für den Fall, in welchem das Konstrukt außerhalb einer Parallelregion steht, da hier alle Tasks sofort sequentiell bei ihrem Auftreten ausgeführt werden.

4.8.3 Taskgroup Konstrukt

Eine weitere Möglichkeit zu Synchronisation von Tasks bietet das *taskgroup*-Konstrukt. Es wird durch `pragma omp taskgroup` eingeleitet und ordnet alle Tasks, welche innerhalb des Anweisungsblocks des Konstrukts erstellt werden, einer gemeinsamen Gruppe von Tasks zu. Ebenso sind alle Kinder-tasks dieser Tasks und auch alle anderen Nachkommen wiederum der Gruppe zugehörig. Sollte das Ende des *taskgroup*-Anweisungsblocks erreicht sein, darf das Konstrukt erst wieder verlassen werden, wenn vorher alle Tasks aus der Gruppe beendet sind.

Im erweiterten LZS wird ein *taskgroup*-Konstrukt durch eine zugehörige Struktur `taskgroup` (siehe Listing 4.15) verwaltet. Diese zählt mittels eines atomaren Zählers `active_tasks` diejenigen Tasks mit, welche innerhalb der Gruppe erstellt, aber noch nicht fertiggestellt sind. Nur wenn der Wert des Zähler 0 entspricht, darf das *taskgroup*-Konstrukt durch den ausführenden Thread wieder verlassen werden. Zudem wird ein Zeiger auf eine mögliche äußere Gruppe, welche das aktuelle *taskgroup*-Konstrukt umschließt, im Element `outer` abgelegt.

```
struct taskgroup {
    taskgroup_t *outer;
    atomic_t activeTasks;
};
```

Listing 4.15.: Aufbau einer `taskgroup`-Struktur

Durch den Clang-Compiler wird aus dem Beispiel 4.16 der Zwischencode 4.17 erstellt. Er enthält im Anschluss an eine Bestimmung der globalen Thread-ID die beiden Aufrufe `kmpc_taskgroup` und `kmpc_end_taskgroup`.

```
int main(void) {
    #pragma omp taskgroup
    {
        /* Anweisungen */
    }
    return 0;
}
```

Listing 4.16.: OpenMP *taskgroup*-Konstrukt

```
define i32 @main() local_unnamed_addr #0 {
  %1 = tail call i32 @__kmpc_global_thread_num(%ident_t* nonnull @0) #1
  tail call void @__kmpc_taskgroup(%ident_t* nonnull @0, i32 %1) #1
  ; Anweisungen
  tail call void @__kmpc_end_taskgroup(%ident_t* nonnull @0, i32 %1) #1
  ret i32 0
}
```

Listing 4.17.: Gekürzter LLVM-Zwischencode für ein OpenMP *taskgroup*-Konstrukt
Ungekürzte Version in B.7

Im erweiterten LZS wird in der Funktion `kmpc_taskgroup` eine Gruppe mithilfe der `taskgroup`-Struktur erstellt und beim aufrufenden Thread abgespeichert. Nachdem dieser Thread den Anweisungsblock des *taskgroup*-Konstrukts bearbeitet hat, wartet er innerhalb der Funktion `kmpc_end_taskgroup` auf die Beendigung aller Tasks aus der Gruppe. Dazu führt er einen vergleichbaren Algorithmus aus, wie auch bei einem Warten im *taskwait*-Konstrukt (siehe Kap. 4.8.2). Anstelle eines Vergleichs des Kinder-Task-Zählers wird in diesem Algorithmus allerdings der `active_tasks`-Zähler aus der `taskgroup`-Struktur verglichen.

Für die Rückkehr aus der Funktion und dem Sonderfall außerhalb einer Parallelregion gilt dabei das Gleiche wie bereits in Kapitel 4.8.2 für das *taskwait*-Konstrukt beschrieben.

4.9 For Konstrukt

In Kapitel 3.11 wurde bereits beschrieben, auf welche Weise im Tasking 2.0 LZS Schleifeniterationen von For-Schleifen als Tasks dargestellt und bei Bedarf verteilt werden können. In der OpenMP-API ist zur Parallelisierung von For-Schleifen das Konstrukt *for* definiert. Zur Aufteilung von Schleifeniterationen auf die Threads einer umschließenden Parallelregion beschreibt der OpenMP-Standard mehrere verschiedene Vorgehensweisen, welche durch die Klausel *schedule* festgelegt werden.

Die davon im erweiterten LZS implementierten Vorgehensweisen werden hier im folgenden vorgestellt:

- ***static nochunk*** (dt.: statisch zusammenhängend): Die Iterationen werden zu etwa gleich großen Blöcken auf die Threads verteilt. Jeder Thread erhält dabei genau einen Block zusammenhängender Iterationen.
- ***static chunk*** (dt.: statisch gestückelt): Die Iterationen werden zu Blöcken von festgelegter Größe aufgeteilt. Die Blöcke werden dann reihum den Threads zugeordnet bis keine Blöcke mehr übrig sind.
- ***dynamic*** (dt.: dynamisch): Immer wenn ein Thread keine Arbeit zu verrichten hat, nimmt er sich selbstständig einen Block an Iterationen. Alle Blöcke besitzen dabei die gleiche festgelegt Größe. Die Threads weisen sich selbstständig immer einen einzelnen Block zu bis schlussendlich keine Iterationen mehr zu verteilen sind.
- ***guided*** (dt.: geführt): Auch hier holen sich die Threads wie bei *dynamic* selbstständig Blöcke mit Iterationen. Die Blockgröße richtet sich allerdings danach, wie viele Threads das Team enthält und wie viele Iterationen noch zu verteilen sind. Nach unten kann eine minimale Blockgröße festgelegt werden.
- ***auto*** (dt.: automatisch): Bei dieser Vorgehensweise wird dem jeweiligen LZS überlassen, wie die Iterationen auf Threads zu verteilen sind.
- ***runtime*** (dt.: Laufzeit): In diesem Fall wird erst zur Laufzeit des Programms entschieden, welches Vorgehen verwendet wird. Dieses wird durch die Umgebungsvariable `OMP_SCHEDULE` und die zugehörigen Laufzeitroutine `omp_set_schedule` festgelegt.

In Listing 4.18 ist ein Beispielprogramm mit einem *for*-Konstrukt dargestellt. Auch hier ist wieder anzumerken, dass das Beispielprogramm nicht einem standardmäßigen Gebrauch des Konstrukts entspricht, da es i.d.R. nur innerhalb einer Parallelregion verwendet wird. Neben der *for*-Direktive ist die Klausel *schedule* notiert. Ihr Wert im Beispielcode entspricht einem statischen, zusammenhängenden Scheduling.

```
int main(void) {

    #pragma omp for schedule(static)
    for (int i = 0; i < 42; i++) {
        /* Anweisungen */
    }
    return 0;
}
```

Listing 4.18.: OpenMP *for*-Konstrukt

Betrachtet man zu diesem Beispiel den produzierten LLVM-Bytecode in Kapitel A.1, so erkennt man, dass durch den Compiler der Aufruf `__kmpc_for_static_init_4` einfügt wurde. Dieser bekommt dabei Werte übergeben, mit welchen sich der Iterationsbereich der Schleife definieren lässt. Dazu gehören der Start- und Endwert, sowie das Inkrement der Schleifenvariablen. Innerhalb der Funktion implementiert das erweiterte Tasking 2.0 LZS das oben beschriebene statische Scheduling. Jeder Thread ruft dabei diese Funktion auf und erhält jeweils seinen eigenen Iterationsblock zurück, indem seine übergebenen Werte für die Schleifenvariable entsprechend innerhalb der Funktion überschrieben werden. Ebenso würde ein Thread im Falle eines gestückelten statischen Schedulings auf diese Weise auch die Schrittweite bis zum nächsten Iterationsblock erfahren.

Im Zwischencode wurden dabei Anweisungen eingeführt, die sicherstellen, dass ein einzelner Iterationsblock nicht über das ursprüngliche Schleifenende hinausgeht. Bei Sprungmarke 15 werden zu diesem Zweck zu große Endwerte eines einzelnen Blockes auf den Endwert der ursprünglichen Schleife limitiert.

Es wurde zudem vom Compiler eine neue Schleife implementiert, welche über die Iterationen eines einzelnen Blockes läuft. Die Schleifenbedingung wird dabei bei Sprungmarke 18 ausgewertet. Der Schleifenrumpf befindet sich bei Marke 22.

Sobald die Iterationen eines Blocks beendet sind, ist auch für den ausführenden Thread keine Arbeit mehr im *for*-Konstrukt zu erledigen und er ruft die eingefügte Funktion `__kmpc_for_static_fini` bei Sprungmarke 30 auf. Da bereits die Vorgaben des Standards mithilfe der anderen Funktionen vollständig erfüllt sind, enthält diese Funktion im erweiterten LZS eine leere Implementation. Im Anschluss daran wird noch eine Barriere im Team durchgeführt, indem jeder Thread den Aufruf `__kmpc_barrier` tätigt.

Die bisher betrachteten Funktionsaufrufe werden allerdings ausschließlich beim Vorgehen *static nochunk* und *static chunk* eingefügt. Für die anderen Verfahren ist in Kapitel A.2 exemplarisch der Zwischencode für ein dynamisches Scheduling mit Blockgröße `1` dargestellt.

Im Unterschied zum statischen Scheduling werden beim Zwischencode für dynamisches Scheduling die Blöcke für jeden Thread nicht durch ein einmaliges Aufrufen von `__kmpc_for_static_init_4` bestimmt. Stattdessen wird einmal zu Beginn `__kmpc_dispatch_init_4` aufgerufen. Im erweiterten LZS wird innerhalb dieser Funktion vom Masterthread für jeden Iterationsblock ein eigener Schleifen-Task erstellt, welcher den Iterationsbereich des Blockes in seinen Elementen abspeichert (vlg. `task_t`-Struktur 4.10). Im Anschluss daran versuchen alle Threads in einer vom Compiler eingefügten Schleife bei Sprungmarke 10 einen dieser Tasks zu erhalten, indem sie die Funktion `__kmpc_dispatch_next_4` aufrufen.

Die Implementierung dieser Funktion entspricht dabei einem klassischem Scheduling-Algorithmus ähnlich zu Abb. 3.3. Sobald sie mittels des Algorithmus einen Schleifen-Task erhalten haben, kehren sie aus der Funktion zurück und bearbeiten die Iterationen aus dem Task in einer vom Compiler dafür extra eingebauten Schleife wie bereits beim statischen Verfahren besprochen. Ein Workerthread versucht nach einer Bearbeitung eines Blockes dabei immer wieder einen weiteren Schleifen-Task zu erhalten. Er beendet den Algorithmus nur dann ohne einen Task, wenn er vom Masterthread die Nachricht `RT_EXIT_FOR_LOOP` empfängt.

Der Masterthread arbeitet währenddessen nach einem ähnlichen Prinzip wie auch die Workerthreads aber kontrolliert immer zusätzlich, ob alle Schleifen-Tasks bereits ausgeführt wurden. Tritt dieser Fall ein, informiert er alle Worker mittels der Nachricht `RT_EXIT_FOR_LOOP`. Alle Threads verlassen nach einer anschließenden gemeinsamen Barriere das Konstrukt.

Die beim dynamischen oder auch geführten Scheduling-Verfahren verwendeten Schleifen-Tasks dürfen allerdings wegen der Vorgaben aus dem OpenMP-Standard nicht zerteilt werden. Die Implementierungen zum Zerteilen von Schleifen-Tasks aus dem ursprünglichem Tasking 2.0 LZS (vgl. auch Kap. 3.11) können daher nur beim Verfahren *auto* zum Einsatz kommen, da für dieses keine Vorgaben von Seiten der OpenMP-API definiert sind. In diesem Scheduling-Verfahren erstellt der Aufruf `__kmpc_dispatch_init_4` nur einen einzigen grobgranularen Schleifen-Task im LZS, welcher alle Iterationen der For-Schleife beinhaltet. Das durchgeführte Scheduling entspricht grundsätzlich dem Scheduling bei *dynamic* und *guided*. Allerdings kann ein Thread, welcher einen Schleifen-Task besitzt, diesen zerteilen, um damit bei ihm ankommende Stehlanfragen zu beantworten. Auf diese Weise werden nur dann neue Tasks durch Zerteilen erstellt, wenn dies auch wirklich von anderen Threads gefordert wird (Lazy-Splitting-Strategie).

4.10 Daten-Klauseln

Für einige OpenMP-Konstrukte, lassen sich sog. Daten-Klauseln zusätzlich verwenden. Möchte man innerhalb eines solchen Konstrukts Variablen nutzen, welche außerhalb des Konstrukts deklariert wurden, ist dies durch eine Angabe der Variablen in solchen Klauseln möglich. Dabei unterstützt das erweiterte Tasking 2.0 LZS im Allgemeinen die Klauseln *firstprivate*, *private* und *shared* aus der OpenMP-Schnittstelle.

- ***shared*-Klausel**: Auf Variablen, die innerhalb dieser Klausel aufgelistet werden, kann von allen Threads innerhalb des zugehörigen Konstrukts gemeinsam zugegriffen werden. Es existiert dabei nur eine einzige Instanz pro Variable. Der Programmierer hat hier selbst für möglicherweise auftretende Wettlaufsituationen Sorge zu tragen.

- ***private*-Klausel**: In allen generierten Tasks existiert jeweils eine private Version zu jeder angegebenen Variable. Diese Variablen sind nicht mit Werten initialisiert.
- ***firstprivate*-Klausel**. Auch hier besitzt jeder Task eine lokale Version zu jeder Variable. Diese sind allerdings mit den Werten initialisiert, welche die Variablen außerhalb des Konstruktes besaßen.

Daten-Klauseln können bei den Konstrukten *parallel*, *task*, *single* und *for* angewendet werden. Siehe für eine exakte Aufgliederung der Klauseln auf die Konstrukte die Übersicht in Abb. 4.1. Im LZS werden die Klauseln dabei wie folgt für die verschiedenen Direktiven realisiert:

- ***parallel*-Konstrukt**: Hier werden die in den Daten-Klauseln definierten Variablen als Übergabeparameter an die ausgelagerte Funktion der Parallelregion übergeben. Bei *firstprivate*-Variablen wird dazu der Wert der Variablen als Übergabeparameter verwendet. Für *shared*-Variablen wird ein Zeiger auf diese benutzt. Variablen vom Typ *private* tauchen nicht als Übergabeparameter auf. Stattdessen wird die private Version einer solchen Variable durch vom Compiler eingefügten Code direkt innerhalb jedes impliziten Tasks erzeugt. Pro Task werden neben dem Zeiger auf die ausgelagerte Funktion nun auch noch die Übergabeparameter für die Funktion mit abgelegt. Dies geschieht dabei für jeden Task einzeln in einem vom LZS allokierten Speicherbereich, welcher direkt im Anschluss an die Task-Struktur liegt (vlg. Struktur 4.10). Das Element `argc` aus der Task-Struktur speichert dazu die Anzahl der auf diese Weise abgelegten Übergabeparameter. Bei der Ausführung eines solchen impliziten Tasks wird dann die in `fn` hinterlegte Funktion mit den dazu gespeicherten Parametern ausgeführt. Vergleiche dazu Kapitel 4.3.
- ***task*-Konstrukt**: Auch hier werden die in den Daten-Klauseln verwendeten Variablen im Anschluss an die *task*-Struktur im Speicher abgelegt. Anders als beim *parallel*-Konstrukt ist es hier allerdings nicht die Aufgabe des Laufzeitsystems die Variablen im Speicher abzulegen. Es wird ausschließlich erwartet, dass ausreichend allokierter Speicher bei der Rückgabe des Tasks aus dem Aufruf `__kmpc_omp_task_alloc` zur Verfügung gestellt wird und dass das Strukturelement `shareds` auf den Beginn des Speicherbereichs für *shared*-Variablen zeigt. Gefüllt werden die Variablenspeicher von darauf folgendem Code, welcher vom Compiler dafür in den Zwischencode eingefügt wird. In der erweiterten Tasking 2.0 Implementierung entspricht die Speicherrepräsentation eines solchen Tasks dem Aufbau in Abbildung 4.4.
- ***for*- und *single*-Konstrukt**: Da in diesen beiden Konstrukten keine ausgelagerten Funktionen vom Compiler erzeugt werden, befinden sich die Variablen außerhalb sowie innerhalb des Anweisungsblocks eines Konstruktes immer

im selben Sichtbarkeitsbereich. Aus diesem Grund müssen die Variablen nicht speziell durch das LZS behandelt werden. Der Compiler übernimmt in diesem Fall alleine die Realisierung der jeweiligen Klausel.

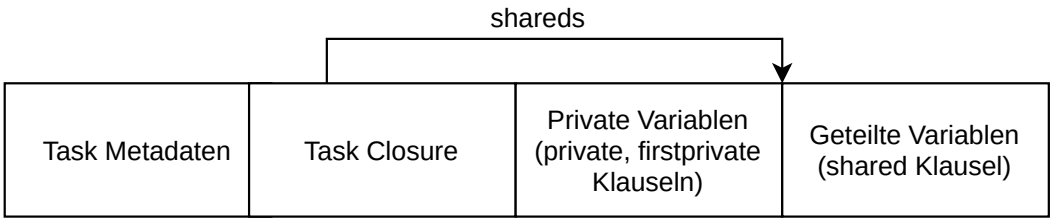

Abb. 4.4.: Task-Struktur Speicherrepräsentation

4.11 Nachrichtenaustausch

Möchte der Masterthread den Workerthreads mitteilen, dass das LZS im nächsten Schritt beendet werden soll, so geschah dies im ursprünglichen Tasking 2.0 LZS noch durch das Versenden eines Schein-Tasks, welcher ausschließlich zum Transportieren eines Funktionszeigers verwendet wurde. Ein solcher Task wurde vom Masterthread an seine beiden Kinder-Threads im Thread-Baum (siehe Abb. 3.1) geschickt. Das Ausführen eines Schein-Tasks durch einen Kinder-Thread veranlasste dieses Kind, gleichermaßen solche Schein-Tasks an die eigenen Kinder-Threads zu versenden. Auf diese Weise wurde die Nachricht im Thread-Baum nach unten propagiert.

In der Erweiterung des Laufzeitsystems wurde sich dazu entschieden, Nachrichten getrennt von Tasks zu behandeln. Dies ermöglicht es, für eine Nachricht ein weniger verschwenderisches Speicherformat zu wählen. Statt eine `task_t`-Struktur zu versenden wird nun nur noch ein Enumerationswert des Aufzählungstyps `RT_messaging_t` verschickt (siehe Listing 4.19).

```
enum RT_messaging_t {
    RT_EXIT,
    RT_THREAD_BARRIER_AND_EXIT_PARALLEL,
    RT_EXIT_TASK_BARRIER,
    RT_EXIT_FOR_LOOP
};
```

Listing 4.19.: Aufzählungstyp RT_messaging_t

Aufgrund der Leichtgewichtigkeit einer Nachricht wurde sich zudem dazu entschieden, dass der Masterthread das Versenden der Nachrichten an alle Workerthreads selbst übernimmt. Der Versand geschieht nun über eigens dafür eingefügte Channel. Diese Aufteilung hilft auch beim gezielten Auslesen von Nachrichten durch einen Thread, sodass dieser nicht länger einen Channel ausliest, in welchem potentiell

sowohl Nachrichten als auch Tasks lagern können und er stets eine anschließende Unterscheidung durchführen muss.

Nachrichten kommen im erweiterten LZS zum Einsatz, wenn der Masterthread alle erstellten Threads über die Beendigung des Laufzeitsystems informiert (`RT_EXIT`) (siehe Kap. 4.2. Der Nachrichtentyp `RT_THREAD_BARRIER_AND_EXIT_PARALLEL` wird versendet, wenn eine Parallelregion verlassen werden kann (siehe Kap. 4.3). Nach einer Task-Barriere kommen Nachrichten vom Typ `RT_EXIT_TASK_BARRIER` zum Einsatz, um den Workerthreads mitzuteilen, dass die Barriere erfolgreich durchgeführt wurde (siehe Kap. 4.8.1). Möchte der Masterthread über das Beenden eines *for*-Konstrukts informieren, versendet er `RT_EXIT_FOR_LOOP`-Nachrichten (siehe Kap. 4.9).

5 Laufzeittests und Vergleiche

In diesem Kapitel werden verschiedene Laufzeitmessungen mit dem erweitertem Tasking 2.0 LZS durchgeführt. Die Ergebnisse werden im Anschluss den Laufzeiten des ursprünglichen LZS und weiteren OpenMP-Implementierungen gegenübergestellt. Es wird dabei stets versucht zu erklären, wodurch sich die Laufzeitunterschiede begründen. Ein Punkt in einem der Diagramme entspricht hier immer dem Median von mehreren gleichen Messungen.

Im Vergleich zum Tasking 2.0 ist die maximale Anzahl an Threads im erweiterten LZS nicht mehr auf 32 sondern nun auf 127 Threads begrenzt. In beiden Tasking 2.0 Laufzeitsystemen ist jeweils die Steal-Half-Strategie zum Stehlen von Tasks gewählt. Dies soll vor allem bei vielen Tasks eine gute Verteilung innerhalb des Laufzeitsystems sicherstellen. Die folgenden Compiler-Optionen wurden, sofern nicht anders vermerkt, beim Übersetzen gewählt: `STEAL=half`, `STEAL_EARLY`, `STEAL_EARLY_THRESHOLD=0`, `SPLIT=adaptive`, `MAXSTEAL=1`

Die zusätzlich verglichenen OpenMP-Laufzeitsysteme setzen sich zusammen aus:

- **libgomp** des GCC [GCC]: Tasks werden hier in mehreren Warteschlangen parallel gehalten. In einer Warteschlange des Eltern-Task, in einer Schlange des zugehörigen Teams und in einer Schlange der Taskgroup. Wenn ein Task aus einer dieser Warteschlangen entnommen wird, muss er auch aus den anderen entfernt werden.
- **libomp** der LLVM [LLV]: Jeder Thread besitzt eine eigene private Warteschlange. Aus anderen Warteschlangen können Tasks direkt mittels Synchronisation gestohlen werden.
- **LOMP** von Cownie et al. [CK21b]: Dieses LZS wurde unter anderem zu Bildungszwecken entwickelt und entspricht einer sehr grundlegenden und einfachen Realisierung. Wie auch bei libomp verwaltet jeder Thread eine eigene Warteschlange, auf welche mittels Synchronisation zugegriffen und gestohlen wird. Dieses LZS ist vergleichsweise fehleranfällig.

Zur Ausführung der Laufzeittests wurden *AMD EPYC™ 7551* Serverprozessoren auf zwei Sockeln verwendet. Insgesamt lassen sich dadurch 64 Kerne bzw. 128 Threads verwenden.

5.1 Barriere

In Diagramm 5.1 ist die Laufzeit abgebildet, die für eine Barriere benötigt wird, welche gleich im Anschluss an die Initiierung des Laufzeitsystems durchgeführt wird. Die Laufzeitmessung beinhaltet jeweils nicht die Initiierung- sowie die Beendigungskosten eines Laufzeitsystems. Man erkennt, dass das erweiterte Tasking 2.0 LZS für die ersten 32 Threads nur einen minimalen Overhead zu den Laufzeiten des ursprünglichen Tasking 2.0 hinzufügt. Allerdings enthält die Barriere des ursprüngliche Tasking 2.0 dafür auch ausschließlich eine Task- und keine Thread-Barriere.

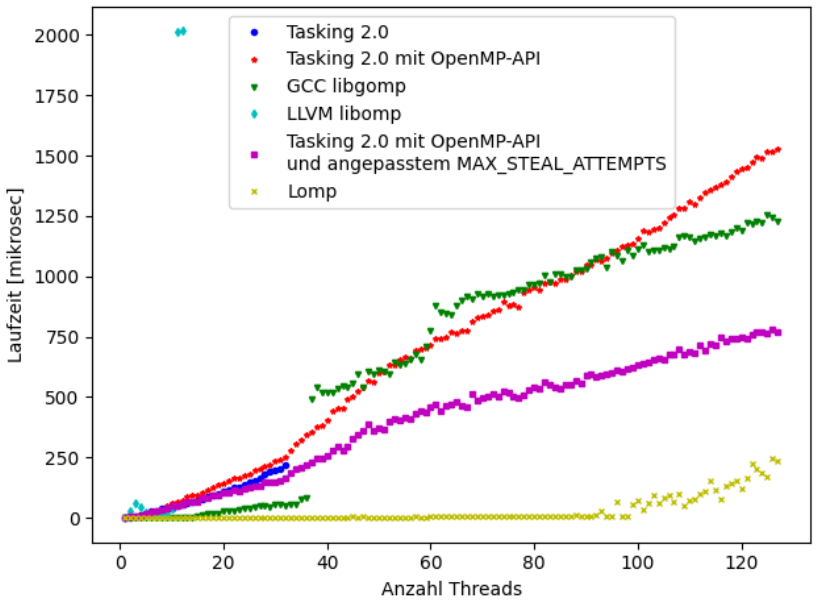

Abb. 5.1.: Zeit für die erste Barriere

Der Laufzeitanstieg verhält sich bei steigender Threadanzahl relativ linear. Dies ist damit zu begründen, dass in einer Barriere darauf gewartet wird, dass sich der Thread-Baum von unten nach oben hin als unbeschäftigt markiert (vgl. Kap. 3.8). Threads melden sich bei ihren Eltern als unbeschäftigt, wenn sie eine gescheiterte Stehlanfrage erhalten. Eine Stehlanfrage wird dabei standardmäßig nach `#threads-1`-vielen Ablehnungen als gescheitert markiert, sodass sie zum Zeitpunkt der Ablehnung von jedem anderen Thread einmal bearbeitet wurde. Setzt man diesen Wert stattdessen auf `#threads/4` wird eine Anfrage nur noch mehr von einem Viertel aller Threads bearbeitet, bevor sie als gescheitert gelten darf. Dies verkürzt die Laufzeit auf bis zur Hälfte des Ausgangswerts.

Das libgomp ist für eine Threadanzahl unter 36 schneller als das erweiterte Tasking 2.0, springt allerdings nach dieser Schranke plötzlich auf eine 6-fache Ausführzeit und ist ab dieser Threadanzahl langsamer als das erweiterte Tasking 2.0 mit angepasster Anzahl Stehlversuche. Das *libomp* liefert nur für unter 10 Threads im

Diagramms darstellbare Laufzeiten. Im Anschluss dauert eine Barriere länger als 2 Millisekunden.

Das LOMP-LZS liefert mit Abstand die beste Performance in diesem Test. Bis zu einer Threadanzahl von etwa 90 erzeugt es stets Laufzeiten unter 10 Mikrosekunden. Erst im Anschluss steigen die Werte langsam an. Begründet werden kann dies mit der dort verwendeten, sehr einfachen OpenMP-Implementierung. Ein Zähler speichert dort die Anzahl unbearbeiteter Tasks innerhalb des Teams. Die Threads dürfen bereits nach einer kurzen Überprüfung dieses Zählers die Barriere wieder verlassen.

5.2 Fibonacci

Die Berechnung der 28-ten Zahl aus der Fibonacci-Folge ist im Diagramm 5.2 dargestellt. Dieser Test zeigt dabei den Umgang des Laufzeitsystems mit der Erstellung vieler feingranularer Tasks beim Absteigen in die Rekursion und einer häufigen Synchronisierung beim Wieder-Aufsteigen. Das Starten und Beenden des jeweiligen Laufzeitsystems wird in diesem Benchmark mitgemessen.

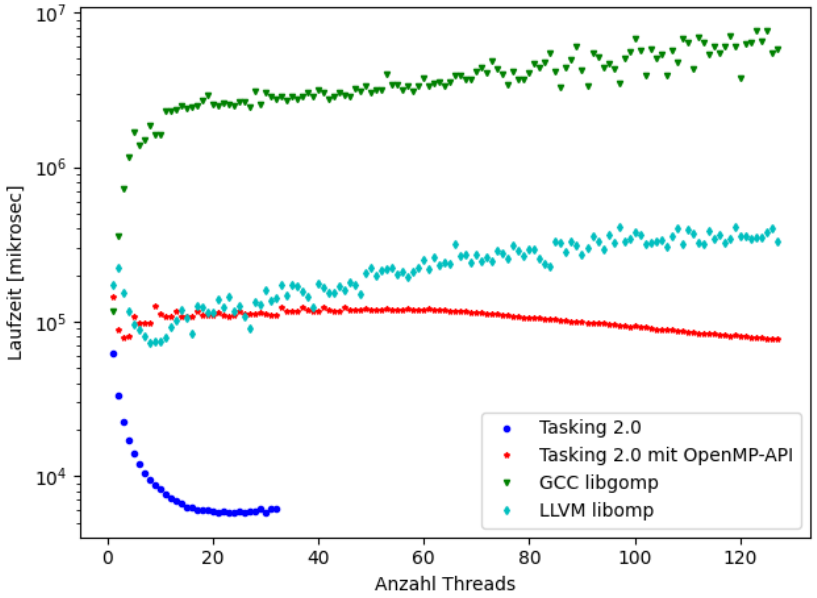

Abb. 5.2.: Laufzeit für die parallele Berechnung der 28-ten Fibonacci-Zahl

Während eine gute Verteilung der erstellten Tasks im ursprünglichen Tasking 2.0 ein Verringern der Laufzeit mit steigender Threadanzahl zur Folge hat, scheitert das erweiterte Tasking 2.0 an einer solchen. Die Kosten für die Verteilung und für die Barrieren überschattet die Vorteile, welche eine Parallelisierung der Arbeit mit sich bringt. Dies ist dabei mit der besonderen Feingranularität der Tasks zu begründen, welche hier vorliegt. Auch das libgomp und das libomp-LZS scheitern an dieser und erzielen sogar noch schlechtere Resultate.

Beim erweiterten Tasking 2.0 LZS ist festzustellen, dass die Laufzeiten ab einer Threadanzahl von etwa 60 wieder leicht sinken. Dies liegt daran, dass ab 65 Threads, ein Prozessorkern von mehreren Threads verwendet wird. Threads, welche abwechselnd auf einem gemeinsamen Kern arbeiten benötigen länger, bis sie ihre erste Stehlanfrage versenden. Dadurch werden insgesamt weniger Tasks gestohlen und weniger Overhead erzeugt.

Das LOMP terminierte bei der Berechnung der Fibonacci-Zahlen nicht.

5.3 Simple Producer-Consumer

Bei diesem Benchmark erzeugt ein einzelner Thread (in diesem Fall der Masterthread) N-viele Tasks von einer einstellbaren Granularität. Dieser Benchmark wird durchgeführt, um das Verhalten des Laufzeitsystems bei grobgranulareren Tasks zu beobachten. Im Diagramm 5.3 und 5.4 ist der Speedup für die unterschiedlichen Laufzeitsysteme abgebildet.

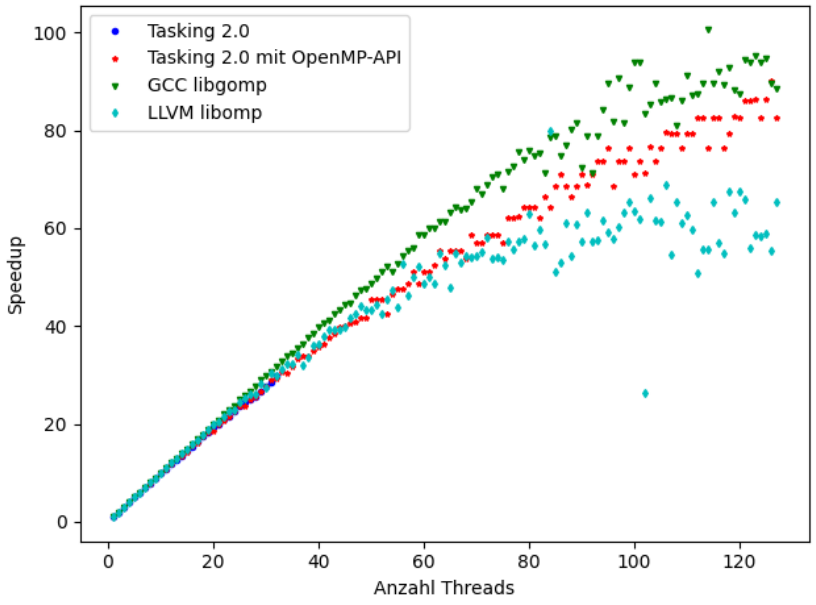

Abb. 5.3.: Speedup für die parallele Berechnung von 2000 Tasks bei einer Taskgranularität von 10ms

Für eine Granularität von 10 Millisekunden erzielen das erweiterte und das ursprüngliche Tasking 2.0 ähnliche Speedups (bis zu 90 beim erweitertem LZS). Diese ordnen sich zwischen die Speedups des libgomp und des libomp ein. Das LOMP erzielte in diesem Test ausschließlich geringe und sehr sprunghafte Speedups und ist daher nicht in der Grafik dargestellt. Ab 64 Threads steigt im Allgemeinen der Speedup nicht mehr in der gleichen Geschwindigkeit an wie vorher, da von diesem Punkt an mehrere Threads sich einen Prozessorkern teilen.

Vermindert man die Granularität auf 1 Millisekunde, so liefert das Tasking 2.0 LZS weiterhin gute Speedups, während libgomp und libomp ab etwa 64 Threads einbrechen und von dort an sogar schlechtere Speedups erzielen.

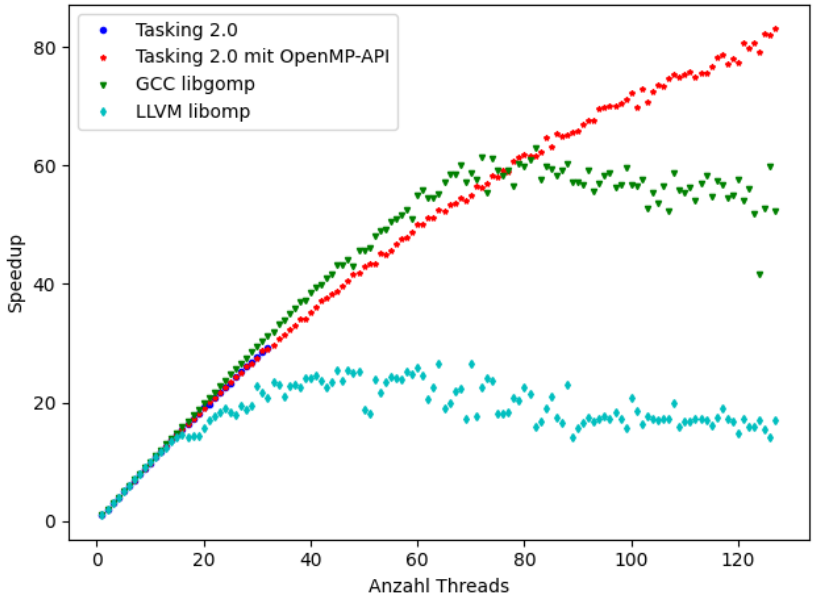

Abb. 5.4.: Speedup für die parallele Berechnung von 2000 Tasks bei einer Taskgranularität von 1ms

5.4 For-Schleifen

Das eingebaute OpenMP For-Konstrukt und die damit verbundenen verschiedenen Scheduling-Verfahren sollen in diesem Test evaluiert werden. Im Testprogramm wird dabei eine Schleife mit 2000 Iterationen ausgeführt. Diese ist mit den For-Konstrukten der jeweiligen API annotiert. Die zeitliche Dauer einer Iteration lässt sich dabei über die Granularität steuern.

Bei einem Test mit einer Granularität von 10 Millisekunden (siehe Abb. 5.5) erreicht das erweiterte Tasking 2.0 bei einem `auto`-Scheduling mit `split_half`-Strategie (vlg. Kap. 3.11) nur einen maximalen Speedup von 4. Dies ist darauf zurückzuführen, dass bei diesem Verfahren zu Beginn nur ein einziger großer Task für alle Iterationen erstellt wird, welcher erst bei Bedarf geteilt wird. Diese Lazy-Splitting-Strategie aus dem ursprünglichen Tasking 2.0 soll dabei das Erstellen von mehr Tasks, als für eine Verteilung der Iterationen notwendig sind, verhindern. Dieses Teilen kann allerdings im erweiterten Tasking 2.0 für einen Task nicht mehr stattfinden, sobald seine Ausführung bereits begonnen wurde. Der Masterthread, welcher zu Beginn den initialen Task besitzt, teilt diesen für seine beiden Kinder-Threads je einmal in der Mitte und führt danach noch 500 Iterationen aus. Dieser Task mit 500 Iterationen kann nicht mehr geteilt werden und gibt damit die Ausführzeit des

gesamten Programms vor. Dies entspricht einem Viertel der sequentiellen Ausführzeit (2000 Tasks) woraus ein Speedup von 4 resultiert. Die nicht abgebildeten Varianten split_guided und `split_adaptive` verändern nichts am bestehenden Problem sodass sie daher auch vergleichbare Werte erzielen.

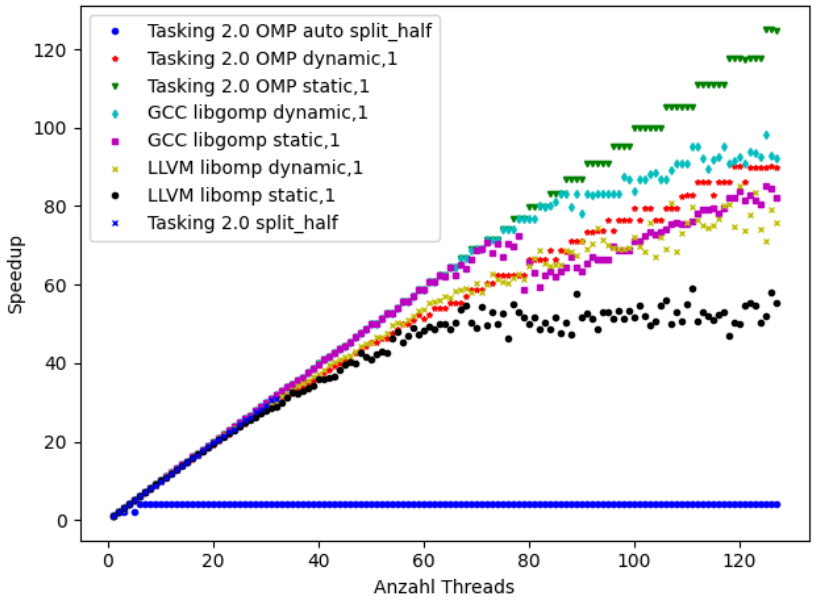

Abb. 5.5.: Speedup für die parallele Berechnung von 2000 Iterationen bei einer Granularität von 10ms

Das ursprüngliche LZS verwendet auch das gerade beschriebene Verfahren, liefert allerdings mit `split_half` (alternativ auch `split_adaptive` und `split_guided`) optimale Speedups. Dies ist damit zu begründen, dass im Vergleich zum erweiterten Tasking 2.0 hier ein Teilen nach jeder durchgeführten Iteration möglich ist. Dadurch kann es nicht passieren, dass ein Thread ein Viertel der Iterationen durchführen muss, während alle anderen Threads unbeschäftigt sind.

Ein dynamisches Scheduling mit einer Blockgröße von 1 entspricht beim erweiterten Tasking 2.0 dem selben Vorgehen wie beim entsprechenden Simple-Producer-Consumer-Test. Es werden beide Male 2000 Tasks mit einer Granularität von 10 Millisekunden erstellt. Deshalb ist auch beide Male der selbe Speedup zu beobachten. Dieser ist ähnlich zu dem Speedup des libomp für dynamisches Scheduling und etwas schlechter als der Speedup des libgomp für dynamisches Scheduling, obwohl das erweiterte Tasking 2.0 als einziges zur Realisierung von dynamischen Schleifen Tasks verwendet.

Statisches Scheduling mit einer Blockgröße von 1 erreicht einen optimalen Speedup von bis zu 125 beim erweiterten Tasking 2.0. Der stufenweise Anstieg ist damit zu begründen, dass es zu jedem Zeitpunkt mehr Tasks als Threads gibt und dadurch einige Threads mehrere Tasks bearbeiten müssen. Die Laufzeit ist dabei durch den Thread mit den meisten Tasks beschränkt. Es kommt immer erst zu einem Anstieg

des Speedups, wenn genug Threads existieren, sodass der Thread mit den meisten Tasks durch das Scheduling einen Task weniger bearbeitet. Die Speedups von libomp und libgomp für statisches Scheduling liegen beide um einiges unter den Speedups des erweiterten Tasking 2.0.

Bei dem LOMP-LZS scheiterte die Einstellung des Scheduling-Verfahrens mittels der dafür vorgesehenen Umgebungsvariable.

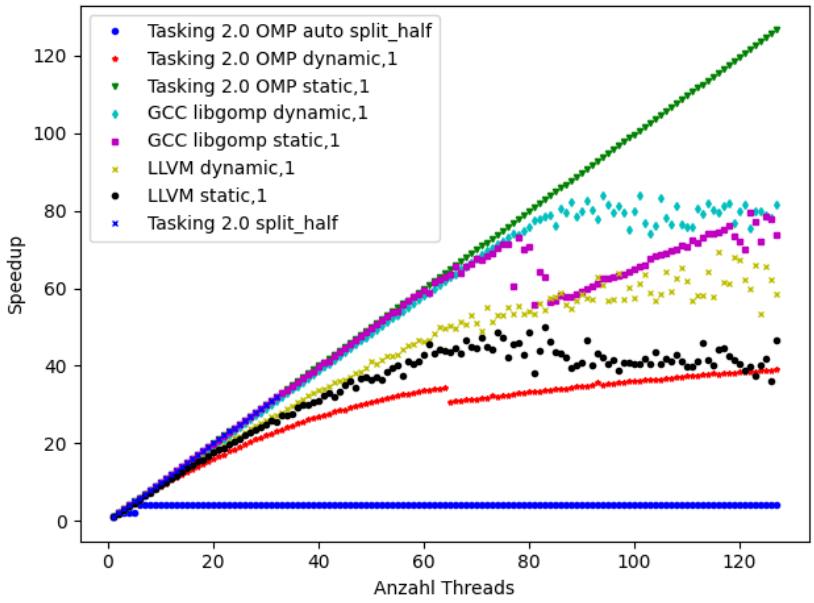

Abb. 5.6.: Speedup für die parallele Berechnung von 2000000 Iterationen bei einer Granularität von 0.01ms

Im Diagramm 5.6 sind nun die Laufzeiten für eine Schleife mit 2 Millionen Iterationen und 10 Mikrosekunden Granularität pro Iteration eingezeichnet. Dadurch ist die sequentiell auszuführende Arbeit identisch mit der aus Diagramm 5.5.

Das erweiterte Tasking 2.0 scheitert aus selbigen Problemen auch hier wieder an einem `auto`-Scheduling mit `split_half`-Strategie. Die statische Verteilung erreicht weiterhin einen optimalen Speedup, da sie intern keine Task-Erstellung vornimmt. Das dynamische Scheduling, welches auf der anderen Seite einen Tasks pro Iteration erzeugt, ist mit der hier auftretenden Menge an Tasks überfordert. Die Kosten für eine Task-Erstellung wirken sich hier stark auf den erzielbaren Speedup aus. Die Idee, Tasks für die Iterationen einer Schleife zu erstellen, funktioniert also schlecht für kleine Granularitäten.

Bei libomp und libgomp lassen sich jeweils Verringerungen bei den Speedups gegenüber den grobgranularen Iterationen nachweisen. Diese kommen bei den statischen Verfahren wahrscheinlich alleinig durch die erhöhte Anzahl von Iterationen, welche ein einzelner Thread zu bearbeiten hat. Daraus resultieren mehr Sprungoperationen,

Additionen der Schleifenvariable sowie Funktionsaufrufe pro Thread. Bei den dynamischen Verfahren wird der Aufruf zum Anfordern eines neuen Iterationsblocks durch einen Thread 1000 mal öfter durchgeführt als zuvor. Da hier keine Tasks erstellt werden, fallen die Einbußen allerdings insgesamt geringer aus als beim erweiterten Tasking 2.0.

Das ursprüngliche Tasking 2.0 erzielt aufgrund des Lazy-Splittings für die unterschiedliche Granularität identische Speedups.

6 Zusammenfassung

In dieser Arbeit wurde das Laufzeitsystem Tasking 2.0 von A. Prell um eine partielle Implementierung der OpenMP-Nutzerschnittstelle erweitert. Alle Konstrukte aus dem LZS, zu denen ein entsprechendes Gegenstück in der OpenMP-API existiert, wurden erfolgreich realisiert. Die Möglichkeiten zur Synchronisation von Tasks wurden weiter ausgebaut, ebenso wie die Anzahl der nutzbaren Scheduling-Verfahren für For-Schleifen. Dabei wurde der grundlegende Algorithmus zur Verteilung von Tasks mittels Stehlanfragen stets bewahrt.

Wo immer möglich, wurden intern Tasks zur Realisierung der eingefügten Konstrukte verwendet. So bietet das erweiterte LZS nun beispielsweise einen eigenen Ansatz für OpenMP-For-Schleifen mit dynamischen Scheduling, indem intern Iterationsblöcke als Tasks gespeichert werden. Dadurch kann bei der Zuweisung der Iterationen an die Threads auf das Task-Scheduling des Tasking 2.0 zurückgegriffen werden.

Der Laufzeitvergleich mit anderen OpenMP-Implementierungen zeigte, dass der verwendete Task-Stealing-Algorithmus mit den entsprechenden Algorithmen des GCC und des LLVM für grobgranulare Tasks und Schleifeniterationen mithalten kann oder diese teilweise sogar hinter sich lässt. Bei besonders feingranularen Tasks fällt der Overhead der Task-Erstellung allerdings stark ins Gewicht. So schneidet das erweiterte Tasking 2.0 bei Schleifen mit wenig Arbeit pro Iteration schlechter ab als seine Konkurrenten, welche keine Tasks zur Schleifenrealisierung verwenden. Hier musste leider aufgrund der Vorgaben durch den Clang/LLVM-Compiler auf das im ursprünglichen Tasking 2.0 verwendete Lazy-Splitting verzichtet werden, welches eine entsprechende Strategie für die Lösung dieses Problems darstellt.

Aufgrund der im OpenMP-Standard vorgeschriebenen Verwendung von Parallelregionen zur Verteilung von Tasks konnte außerdem keine Abwärtskompatibilität zur Schnittstelle des ursprünglichen Tasking 2.0 LZS aufrecht erhalten werden.

Literatur

[al07] Alfred V. Aho et al. *Compilers: Principles, Techniques and Tools*. 2. Aufl. Pearson, 2007 (zitiert auf den Seiten 5, 10).

[CK21a] Jim Cownie und Michael Klemm. *High Performance Parallel Runtimes*. De Gruyter, 2021 (zitiert auf Seite 12).

[LLV15] LLVM. „LLVM OpenMP Runtime Library“. In: (2015) (zitiert auf den Seiten 12, 26, 30).

[Ope21] OpenMP. „OpenMP Application Programming Interface“. In: (2021) (zitiert auf den Seiten ii, iii, 2, 7, 8, 26, 28).

[Pre16] Andreas Prell. „Embracing Explicit Communication in Work-Stealing Runtime Systems“. In: (2016) (zitiert auf den Seiten 2, 4, 13, 18, 25).

Webseiten

[CK21b] Jim Cownie und Michael Klemm. *Little OpenMP Runtime*. February 2021. URL: `https://github.com/parallel-runtimes/lomp` (besucht am 22. Jan. 2022) (zitiert auf den Seiten 12, 26, 49).

[GCC] GCC. *GNU Compiler Collection*. URL: `https://gcc.gnu.org/git/gcc.git` (besucht am 22. Jan. 2022) (zitiert auf den Seiten 2, 26, 49).

[LLV] LLVM. *The LLVM Compiler Infrastructure*. URL: `https://github.com/llvm/llvm-project` (besucht am 22. Jan. 2022) (zitiert auf den Seiten 2, 26, 49).

[Ope20] OpenMP. *OpenMP Compilers & Tools*. 2020. URL: `https://www.openmp.org/resources/openmp-compilers-tools/#compilers` (besucht am 22. Jan. 2022) (zitiert auf Seite 10).

[Pre] Andreas Prell. *Tasking 2.0*. URL: `https://github.com/aprell/tasking-2.0` (besucht am 22. Jan. 2022) (zitiert auf den Seiten ii, iii, 1, 13).

Abbildungsverzeichnis

Listing-Verzeichnis

Anhang

A

A.1 Gekürzter LLVM-Zwischencode eines For Konstrukts mit statischem Scheduling

```
define i32 @main() #0 {
  %2 = alloca i32, align 4
  %4 = alloca i32, align 4 ; lb
  %5 = alloca i32, align 4 ; ub
  %6 = alloca i32, align 4 ; Schrittweite
  %7 = alloca i32, align 4
  %8 = alloca i32, align 4
  %9 = call i32 @__kmpc_global_thread_num(%ident_t* @1)
  store i32 0, i32* %4, align 4 ; init lb
  store i32 9, i32* %5, align 4 ; init ub
  store i32 1, i32* %6, align 4 ; init Schrittweite
  store i32 0, i32* %7, align 4
  call void @__kmpc_for_static_init_4(%ident_t* @0, i32 %9, i32 34, i32* %7,
    i32* %4, i32* %5, i32* %6, i32 1, i32 1)
  %10 = load i32, i32* %5, align 4
  ; Überprüfe ob Schleifenende außerhalb des Iterationsbereich liegt
  %11 = icmp sgt i32 %10, 9
  br i1 %11, label %15, label %13

; <label>:13: Schleifenende in Ordnung
  %14 = load i32, i32* %5, align 4
  br label %15

; <label>:15: Schleifenende auf Iterationsbereich limitieren
  %16 = phi i32 [ 9, %12 ], [ %14, %13 ]
  store i32 %16, i32* %5, align 4
  %17 = load i32, i32* %4, align 4
  store i32 %17, i32* %2, align 4
  br label %18

; <label>:18: Schleifenbedingung prüfen
  %19 = load i32, i32* %2, align 4
  %20 = load i32, i32* %5, align 4
  %21 = icmp sle i32 %19, %20
  br i1 %21, label %22, label %30

; <label>:22: Schleifenrumpf
  %23 = load i32, i32* %2, align 4
  %24 = mul nsw i32 %23, 3
```

```
  %25 = add nsw i32 12, %24 ; Wert der aktuellen Schleifenvariable
  store i32 %25, i32* %8, align 4
  ; Anweisungen
  %28 = load i32, i32* %2, align 4
  %29 = add nsw i32 %28, 1 ; Schleifenvariable inkrementieren
  store i32 %29, i32* %2, align 4
  br label %18 ; springe zurück zum Schleifenkopf

; <label>:30: Schleife beendet
  call void @__kmpc_for_static_fini(%ident_t* @0, i32 %9)
  call void @__kmpc_barrier(%ident_t* @2, i32 %9)
  ret i32 0
}
```

A.2 Gekürzter LLVM-Zwischencode eines For Konstrukts mit dynamischem Scheduling

```
define i32 @main() #0 {
  %2 = alloca i32, align 4
  %4 = alloca i32, align 4 ; lb
  %5 = alloca i32, align 4 ; ub
  %6 = alloca i32, align 4 ; Schrittweite
  %7 = alloca i32, align 4
  %8 = alloca i32, align 4
  %9 = call i32 @__kmpc_global_thread_num(%ident_t* @0)
  store i32 0, i32* %4, align 4 ; init lb
  store i32 9, i32* %5, align 4 ; init ub
  store i32 1, i32* %6, align 4 ; init Schrittweite
  store i32 0, i32* %7, align 4
  call void @__kmpc_dispatch_init_4(%ident_t* @0, i32 %9, i32 35, i32 0, i32 9,
    i32 1, i32 1)
  br label %10

; <label>:10: Erhalte neuen Iterationsblock
  %11 = call i32 @__kmpc_dispatch_next_4(%ident_t* @0, i32 %9, i32* %7, i32* %4,
    i32* %5, i32* %6)
  %12 = icmp ne i32 %11, 0
  br i1 %12, label %13, label %29

; <label>:13: Neuen Iterationsblock erhalten
  %14 = load i32, i32* %4, align 4
  store i32 %14, i32* %2, align 4
  br label %15

; <label>:15: Schleifenbedingung prüfen
  %16 = load i32, i32* %2, align 4, !llvm.mem.parallel_loop_access !2
  %17 = load i32, i32* %5, align 4, !llvm.mem.parallel_loop_access !2
  %18 = icmp sle i32 %16, %17
```

```
  ; Schleifenrumpf ausführen oder nächsten Iterationsblock anfordern
  br i1 %18, label %19, label %10 ;

; <label>:19: Schleifenrumpf
  %20 = load i32, i32* %2, align 4, !llvm.mem.parallel_loop_access !2
  %21 = mul nsw i32 %20, 3
  %22 = add nsw i32 12, %21
  store i32 %22, i32* %8, align 4, !llvm.mem.parallel_loop_access !2
  ; Anweisungen
  %25 = load i32, i32* %2, align 4, !llvm.mem.parallel_loop_access !2
  %26 = add nsw i32 %25, 1
  store i32 %26, i32* %2, align 4, !llvm.mem.parallel_loop_access !2
  br label %15, !llvm.loop !2 ; springe zurück zum Schleifenkopf

; <label>:29: keine Iterationsblöcke mehr übrig
  call void @__kmpc_barrier(%ident_t* @1, i32 %9)
  ret i32 0
}
```

Digitaler Anhang

B

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen der Bachelorarbeit, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Die Arbeit wurde bisher nirgends in gleicher oder ähnlicher Form zur Erlangung eines akademischen Grades eingereicht.

Bayreuth, 27. Januar 2022

Christopher Brunner