Projekt „Parallelrechnerevaluation“

Thema	Fortran Parallelisierung Schwerpunkt OpenMP
Autor	Anna Fuchs
Datum	September 2012
In Zusammenarbeit mit	Petra Nerge Julian Kunkel Michal Kuhn Nathanael Hübbe

Inhaltsverzeichnis

1. Vorwort

Heutzutage ist Multiprocessing nicht mehr vom Markt wegzudenken. Alle Rechner sind mittlerweile mit mindestens zwei Prozessoren ausgestattet, sogar Mobiltelefone verfügen über Multicore Prozessoren. Damit tritt auch immer mehr paralleles Programmieren in den Vordergrund, um die Hardware optimal auszunutzen.
Es gibt sehr viele Konzepte des Parallelisierens – sprachenspezifisch, hardwareabhängig, an die Anwendung und nicht zuletzt die Mittel des Kunden angepasst. Der Aufwand und die Effizienz der Verfahren variieren entsprechend.

2. Einleitung

In dieser Ausarbeitung wird das Konzept OpenMP in Fortran näher untersucht. Dabei wird primär auf Schleifenparallelisierung im Hinblick auf Korrektheit und Effizienz geachtet. OpenMP genießt oft den Ruf der „hingepfuschten“ Parallelisierung. Es bedarf nicht sehr viel Aufwand (relativ zu anderen Technologien) bis zum ersten Erfolgserlebnis. Aber ebenso wie man schnell Ergebnisse sieht, kann man auch schnell viel falsch machen, oft bemerkt man das gar nicht, oder erst nachträglich. Die vermeintliche Einfachheit birgt große Gefahren. Bei bedachtem Einsatz und sorgfältiger Überlegung erreicht man mit OpenMP eine hohe Qualität und Effizienz des Programmcodes.
Die Ausarbeitung ist an eine breite Zielgruppe gerichtet, von Anfängern bis hin zu erfahrenen Programmierern. Die Grundlagen werden angerissen, geben den nötigen Impuls und zeigen die Richtung auf, in der man sich nach Eigeninteresse vertiefen kann. Der motivierte Leser wird auch aufgefangen und erfährt viele Details am Rande von OpenMP, die ein gewisses Maß an Erfahrung vorgreifen.
An kleineren, oft von der Anwendung abstrahierten Beispielen werden Verfahren der Optimierung und Parallelisierung gezeigt und analysiert. Das Ziel dabei ist weniger ein konkretes Verfahren zu etablieren, sondern viel mehr das Verständnis von Abhängigkeiten und der Umgebung zu fördern. Beim Parallelisieren einer konkreten Anwendung kommt es oft darauf an, einige Kombinationen auszuprobieren, den Einfluss der anwendungsspezifischen Faktoren zu berücksichtigen und die richtige Balance herauszufinden. Kleinere abstrahierte Beispiele können einfacher direkte Zusammenhänge zeigen und helfen beim gezielten Testen der Anwendungen.
Es wird dem Leser überlassen, letztlich die richtige Strategie für seine Anwendung auszuwählen, sicherlich ist es aber immer hilfreich die Möglichkeiten zu kennen.

Das folgende Kapitel gibt eine knappe Übersicht über Fortran. Dabei wird die Vergangenheit, die aktuelle Rolle im Hochleistungsrechnen und die Zukunft dieser Sprache beschrieben. Im Kapitel 4 folgt eine klassische Übersicht über die Architekturen. Weitere zwei Kapitel beschreiben abstrakte Unterteilung in eingebauten parallele Konzepte und externe äußere Mechanismen und Methoden zum Parallelisieren. Der Schwerpunkt liegt beim externen Parallelismus auf gemeinsamem Speicher. Hierbei wird eine Einführung in OpenMP gegeben, andere Konzepte werden kurz angerissen. Die vorgeführte Theorie soll dann praktisch umgesetzt werden, so wird im Kapitel 7 erst das Vorgehen der Analyse erklärt, und direkt darauf praktische Beispiele gezeigt. Kapitel 9 stellt eine Art herausgearbeitetes Rezept zum Vorgehen beim Parallelisieren. Der Abschluss findet im 9 Kapitel statt, gefolgt vom Anhang mit technischem Hintergrund und dem Quellenverzeichnis.

3. Fortran

3.1 Einführung

Fortran (FORmula TRANslation) ist die erste funktionsfähige höhere prozedurale Programmiersprache entwickelt von Backus bei IBM. In erster Linie wurde diese Sprache für numerische Berechnungen entwickelt und optimiert. Die Ursprünge der Sprache gehen in die 1950-er Jahre zurück, der erste Compiler wurde 1957 entwickelt. Die Sprache hat sich schnell in der Wissenschaft und im Militär verbreitet.
Heute herrscht eine weit verbreitet Meinung, Fortran wäre tot, quasi Latein unter den Programmiersprachen. Diese Meinung ist sehr irreführend. Tatsächlich ist Fortran in weiten Entwicklerkreisen nicht beliebt und weitgehend tot geglaubt. Der moderne Markt wird von neuen, jungen, objektorientierten, Web-adaptierten Sprachen beherrscht, und diese werden mit breitem Bildungsangebot gefördert. Fortran taucht nur selten in Popularitätsstatistiken der Programmiersprachen auf. Vielen Nutzern und Entwicklern bleibt die wissenschaftliche Entwicklung und die Tatsache, dass der Hochleistungsrechnen-Markt von C/C++ und Fortran dominiert wird, verborgen.
Die unübersehbaren Vorteile von Fortran liegen u.A. auch in seiner langjährigen Entwicklung. Als erste höhere Sprache, verfügt Fortran über das größte Volumen an realisierten und getesteten Programmen und Bibliotheken überwiegend im mathematischen Bereich. Viele Projekte haben mittlerweile langjährige Tradition und sind einfach zu groß, um sie in eine andere Sprache umzuschreiben.
Die Einsatzgebiete heute erstrecken sich über Klimamodellierung, Strömungsmechanik, Astronomieanwendungen und viele anderen Gebiete der Wissenschaft und des Inginuerwesens.
Entgegen den Behauptungen, Fortran würde von C/C++ abgelöst werden, hat sich die Sprache auch weiterentwickelt und an die neuen Anforderungen angepasst.

3.2 Versionen[1]

Als erste Version gilt FORTRAN I, entwickelt zusammen mit dem ersten Compiler im Jahr 1957, wobei der Standard mit 1954 datiert wird. Direkt danach im Jahr 1958 kam FORTRAN II mit der Möglichkeit einzelne Module zu kompilieren. Noch im selben Jahr hat man FORTRAN III entwickelt, aber nie veröffentlicht. In dieser Version hätte man Assembler Code direkt in das Programm einbetten können. Davon hat man sich bessere Effizienz versprochen, weil jedoch die Bequemlichkeit und Eleganz einer höheren Programmiersprache verloren ginge, hat man sich damals gegen die Neuerung entschieden. Die 1961 herausgekommene Version FORTRAN IV ist lediglich eine Überarbeitung von FORTRAN II und enthielt keine großen Innovationen. In den Jahren ist die ASA (American Standards Association) auf die Entwicklung von Fortran aufmerksam geworden und hat sich ab 1962 intensiv mit der Sprache beschäftigt. Nach wenigen Jahren brachte die ASA 1966 den ersten Hochsprachen Standard – FORTRAN 66, damals noch von IBM geführt, heraus. Elf weitere Jahre hat es gedauert, bis 1977 Fortran mit dem ISO Standard FORTRAN77 die Welt erobert hat. Die Quellen[2] geben keine eindeutige Antwort, ob die Veröffentlichungen auf das Jahr 1977 oder 1978 zurückzuführen ist. In dieser Version wurden Konstrukte implementiert, auf die heute kaum eine höhere Sprache verzichten kann – PRINT, IF-Blöcke, INCLUDE, rückwärts laufende DO-Schleifen, CHARACTER Datentyp und vieles mehr. Diese Version hatte noch eine sehr strenge Vorgabe über die Codeformattierung: 1 Zeile Bestand aus 80 Spalten, konnte somit 80 Zeichen lang sein, eine Art Lochkarten-Layout, wobei die ersten 7 Zeichen eine spezielle Bedeutung hatten. Darin mussten z.B. Kommentar- und Fortsetzungszeilen eingeleitet werden. Bis zum 72. Zeichen kam der Anweisungscode und die letzten 8 Zeichen galten als Identifikationsfeld, kenntliche Lochungen für Karten. Als Ablöse kam 1991 Fortran 90 und brachte bedeutende Veränderungen, die die Sprache nicht aussterben ließen. Die Spaltenvorgabe wurde aufgelöst, Modularisierung erlaubt und viele mächtige Features sind hinzugekommen. Zu den Neuheiten zählten Operatoren für dynamische Speicherverwaltung (ALLOCATE, DEALLOCATE, NULLIFY), neue Komponenten wie MODULE, PRIVATE, USE etc. SELECT CASE hat umständliche IF-Blöcke und GOTO Anweisungen abgelöst. Der Standard wurde global modernisiert und erlaubte nun auch bis zu 31 Zeichen lange Namen, Überladen der Operatoren, benutzerspezifische Interfaces, abstrakte Datentypen und vieles mehr. Aus dem 90-er Standard entstand eine Version für das Hochleistungsrechnen, HPF, explizit für das parallele Rechnen. HPF beinhaltete das neue interessante Konstrukt FORALL (siehe Kapitel Interner Parallelismus), das mehr Flexibilität und Parallelität ermöglichen sollte. 1997 kam Fortran 95, im Großen und Ganzen eine Korrektur des Vorgängers, jedoch mit dem

1 http://de.wikipedia.org/wiki/Fortran
2 http://www.liv.ac.uk/HPC/HTMLF90Course/HTMLF90CourseNotesnode29.html

aufgenommenen FORALL. Damit wurde ein Trend zum direkten Einbetten der parallelen Konstrukte gesetzt. Fortran 2003 setzte dann auf die Verbesserung der Objektorientierung und ermöglichte nun auch die bessere Kompatibilität mit C. Aus den beiden Versionen 95/03 entstand Co-Array Fortran, das dem Modell PGAS zugeordnet wird (siehe Kapitel Interner Parallelismus). Fortran 2008 beinhaltete diese Konzepte dann komplett, und ermöglichte somit standardisiert eingebettete Parallelisierung der Sprache.

Im Laufe der Ausarbeitung werden prinzipielle Möglichkeiten der Parallelisierung erläutert und an Beispielen veranschaulicht. Im Vordergrund steht dabei die Effizienz der parallelen Verfahren, die im Anschluss analysiert wird. Um diese analysieren und verstehen zu können, folgt eine kurze Zusammenfassung über die wichtigsten Architekturarten.

4. Architekturen

Die für das parallele Rechnen relevante Unterscheidung der Architekturen läuft auf die Klassifizierung der Rechnerarchitekturen in *shared memory* (gemeinsamer Speicher) und *distributed memory* (verteilter Speicher) hinaus. Beim gemeinsamen Speicher geht es dabei um so genannte Rechnerknoten, mehrere CPUs, die sich einen Speicher teilen und somit schnellen Zugriff darauf haben. Solche Bauteile sind in der Regel sehr teuer. Bei *distributed memory* geht es hingegen um CPUs (oder Rechner), die miteinander verbunden werden. Sie haben jeweils ihren eigenen Speicher können und meist mittels Nachrichten miteinander kommunizieren. Letzteres kostet weniger Geld, ermöglicht größeren Ausbau, ist aber entsprechend langsamer.
Im Bereich des Hochleistungsrechnens (HLR) findet man somit meist den Mittelweg, mehrere Rechnerknoten mit jeweils gemeinsamem Speicher. Jeder Prozessor hat einen eigenen Speicher, über den gemeinsamen Adressraum können die anderen aber darauf zugreifen. Das geht schneller oder langsamer, je nachdem ob es sich um die lokale oder fremde Speicheradresse handelt. Eine solche Architektur wäre NUMA (Non-Uniform Memory Access).
Reine *shared memory* Architektur wäre entsprechend UMA (Uniform Memory Access), da der Speicherzugriff im gemeinsamen Adressraum und somit im besten Fall für alle gleich ist. Eine kleine Menge solchen Speichers ist auch bekannt als Cache. Cache ermöglicht die schnellsten Speicherzugriffe und nutzt die Lokalität am besten aus. Dieser Speicher ist für gewöhnlich der teuerste, weswegen man auf verteilten Speicher nicht verzichten kann.

Der Zugriff auf fremden Speicher bringt aber oft Probleme mit sich, falls ein Prozess fremden Speicher berechtigt manipuliert, aber die Änderung nicht rechtzeitig bekannt macht. Das kann passieren, wenn die Veränderung erst im lokalen Cache abgespeichert wird. Es könnte schnell zu Inkonsistenzen führen, falls in der Zeit ein anderer Prozess die nicht aktualisierten Daten anfordert, oder schon veränderte Daten auch in seinem Cache hat. Um das zu verhindern müsste man den Speicher synchronisieren, das würde jedoch alle Vorteile durch den Overhead wieder schnell vernichten. Auch wenn solche Systeme einfacher zu bauen sind, setzt man heute überall auf Hardwareunterstützung, die die Cache-Kohärenz sicherstellt – ccNUMA (cache-coherent NUMA).

Die Kohärenz wird dabei mithilfe von *interconnects* sichergestellt, die in die CPUs eingebaut sind. Man schafft damit einen gemeinsamen virtuellen Adressraum. Die Zugriffen auf physisch fremden Speicher, Remotezugriffe genannt, bleiben dennoch langsamer.

Für die Parallelisierung folgt dabei ein Zusammenhang: um das Optimum an Performance herauszuholen, wäre es Ideal, wenn benötigte Daten im lokalen Speicher der Threads/Prozesse sind. Man muss darauf achten, dass der Thread/Prozess, der die Daten später bearbeiten soll, diese Daten auch initialisiert. So landen die Daten automatisch im lokalen Speicher des zuständigen Threads/Prozesses und minimieren dabei die teuren Zugriffe. Das Verfahren „Master initialisiert alles und verteilt später“ ist meist einfacher zu realisieren, aber viel teurer.

5. Interner Parallelismus

Die Sprachen entwickeln sich weiter und passen sich den gegebenen Anforderungen an. So gibt es viele teils erfolgreiche Versuche parallele Konstrukte in die Sprachen zu integrieren. Einige von den werden hier kurz erläutert.

5.1 PGAS

PGAS *(partitioned global address space)* - bei diesem Programmiermodell wird der Adressbereich logisch unterteilt und auf die Prozessoren aufgeteilt, wobei jeder Prozessor auf den gesamten Speicher mit ggf. geringerer Geschwindigkeit zugreifen kann. Dadurch wird ermöglicht, dass große Datenmengen effizienter unterteilt werden ohne ihre Datenbeschreibung zu verlieren. Unter anderem kann man bei geschicktem Einsatz sehr guten Lastausgleich erreichen. Etwas komplexer ist die Verwaltung von Variablen, da die Zuteilung intern geschieht. Die meist verbreiteten PGAS – Sprachen sind Unified Parallel C und Co-array Fortran, bzw. Fortran 2008.

5.2 Co-array Fortran

Die seit 2008 in den Standard aufgenommene Spracherweiterung funktioniert nach dem Prinzip der Codereplizierung, wobei alle Kopien asynchron ausgeführt werden. Jede Kopie hat ihre eigenen Datenobjekte. Die Syntax wurde entsprechend erweitert. Seit 2008 verbreitet sich das Konzept, Tendenz steigend. Führende Compiler-Hersteller (z.B. Intel, Cray) haben das Konzept mittlerweile aufgenommen, es wird sich erst zeigen, ob Co-array eine vielversprechende Zukunft hat.

5.3 SSE Intrinsic

SSE (Streaming SIMD Extension) ist eine von Intel entwickelte Befehlssatzerweiterung der X86-Architektur. Das Konzept impliziert 16 Register mit 128 Bit Breite (mittlerweile auch 256 Bit). Ein Prozessor braucht 1 Taktzyklus für eine Operation wie Addition oder Multiplikation auf SSE Registern. Damit können z.B. 4 Multiplikationen von Gleitkommazahlen in 1 Taktzyklus berechnet werden. SSE ist durch Intrinsics - in Compiler eingebaute Funktionen - zu nutzen. Das hardwarenahe Programmieren ist sehr aufwendig und weniger elegant im Vergleich zu höheren Programmiersprachen, kann aber das Optimum an Soft- und Hardware herausholen.

5.4 forall

In diesem Abschnitt geht es um ein internes Fortrankonstrukt, das mittels Vektorisierung parallelisiert werden kann.
Für die Verarbeitung und Manipulation der Feldinhalte stehen dem Nutzer die Konstrukte *where* und *forall* zur Verfügung. Das *where* Konstrukt manipuliert nach Kriterien ausgewählte Elemente. Das *forall* setzt im Prinzip das gleiche um, erlaubt elegantere Verpackung mehrerer Anweisungen in einem *forall* Statement und soll bessere parallele Abarbeitung ermöglichen, als *where*. Mit forall kann man einen Idexabhängigen oder Werteabhängigen Zugriff ermöglichen. Die *forall Statements* können besser vom Compiler parallelisiert werden, als gewöhnliche DO Schleifen. Dies gilt es zu überprüfen.

Das where spezifiziert eine Maske und funktioniert analog zum IF-Block. Es gibt ein *where-statement* und ein *where-construct*. Der Unterschied liegt in der Komplexität der Aufgabenstellung. Das *statement* muss in einer Zeile geschrieben sein (& Umbrüche erlaubt), während das *construct* einen Block enthalten kann. Die Form des *statement* sieht wie folgt aus:

where(bedingung) variable = ausdruck

Analog dazu gibt es auch einen *forall* – Einzeiler und Blockkonstrukt.

forall(index = subscript:subscript[:stride]) anweisung !für eindimensionale Arrays

forall(ind1 = subs1a:subs1b[:str1], ind2 = subs2a:subs2b[:str2] [, indn = subsna:subsnb[:strn]])
anweisung !für mehrdimensionale Arrays

Im Gegensatz zum *where* - Konstrukt wird hier auf die Elemente abhängig vom Index zugegriffen. Das *forall* - Konstrukt kann aber auch analog zum *where* abhängig von Werten auf die Elemente zugreifen und eine Maske verwenden.

forall(ind1 = s1a:s1b[:s1] [, indn = sna:snb[:sn] [, maske]) anweisung

So sieht der forall Block, analog auch mehrdimensionale Konstrukte:

```
forall( index = subscript:subscript[:stride] )
        anweisungsblock
end forall
```

6. Externer Parallelismus

Die internen Konstrukte ersetzten natürlich in keiner Weise die klassischen Werkzeuge zum parallelen Programmieren. Alle führenden Technologien unterstützen selbstverständlich Fortran und werden dahingehen weiterentwickelt. Wie auch sonst im HLR unterscheidet man zwischen verteiltem und gemeinsamem Speicher. Führende Technologien sind hier OpenMP für gemeinsamen und MPI für verteilten Speicher, einige weiteren Technologien, wie z.B. GPU-Einsatz, als „Speed-Accelerator“ werden auch angeschnitten.

6.1 OpenMP

OpenMP ist eine Programmierschnittstelle für parallele Programme innerhalb vom gemeinsamen Speicher. Hierbei werden nicht die Daten aufgeteilt, sondern die Arbeit, das Synchronisieren geschieht dabei meist implizit.
Die Entwicklung von OpenMP geht auf 1997 zurück, wobei für C/C++ und Fortran getrennt entwickelt wurde. Erst ab 2005 wurden beide Sprachen in einem Standard vereinigt. Der aktuelle Standard ist v 3.1 vom Juli 2011. OpenMP ist „de facto“ Standard für Parallelisierung auf gemeinsamem Speicher (shared memory).

Es folgt eine kurze Anleitung zu OpenMP in Fortran, Erläuterung besonderer Schwierigkeiten und Rahmenbedingungen, insbesondere für Fortran. Für weiterführende Details siehe Literaturverweis.

a. Kompilieren

Für das Kompilieren muss Folgendes eingestellt werden:

```
gfortran -fopenmp -o test test.f90        # für gfortran
ifort -openmp -o bsp bsp.f90              # für Intel Fortran Compiler
```

Das Modul **`omp_lib`** muss eingebunden werden:

```
use omp_lib  ,
```

statt der alten Version

```
!$ include 'omp_lib.h'
```

Die OpenMP Direktiven werden für Preprocessing kenntlich gemacht. So beginnt jede mit

```
!$omp
```

Sollten die OpenMP Optionen nicht eingeschaltet sein, werden alle OpenMP Direktiven aufgrund von vorangehendem Ausrufezeichen als Kommentar gewertet und beeinflussen das Programm nicht.

b. Threads erzeugen

Der parallele Block wird mit **`!$opm parallel [options]`** eingeleitet und mit **`!omp end parallel`** beendet. An dieser Stelle werden die Threads zerstört, sofern vom Compiler nicht anders optimiert. Wichtig zu beachten, das **`parallel`** und **`end parallel`** Paar muss in derselben Routine sein, auch wenn zwischendurch andere aufgerufen werden.

c. Anzahl der Threads

Mit folgender Anweisung wird die Anzahl der zu erzeugenden Threads gesetzt:

```
!$omp parallel num_threads(10)
```

Sollte man keine Anzahl angegeben haben, so hängt die Anzahl der erzeugten Threads von der Implementierung von OpenMP ab. Meist werden zwar so viele erzeugt, wie viele Prozessoren es gibt (zusätzliche Abhängigkeit von Hyperthreading-Optionen), man sollte sich jedoch nie auf die Grundeinstellungen der Bibliothekshersteller verlassen. Die Zahl der gesetzten Threads gilt nur für den kommenden parallelen Block. Geschachtelte, *nested Threads,* die ggf. von „oberen“ Threads erzeugt werden, sind **nicht** vom „oberen“ Parameter begrenzt und müssen explizit im geschachtelten parallelen Block angegeben werden.
So kann man herausfinden, wie viele Threads gerade überhaupt verfügbar sind:

```
omp_get_max_threads()
```

Dabei wird die Anzahl der tatsächlich parallel verfügbaren Threads angegeben. Man kann sein Programm aber dennoch mit einer größeren Anzahl ausführen, es kommt keine Fehlermeldung. Durch den zusätzlichen Aufwand wird es nur viel langsamer.

d. Synchronisierung

Barrier ist das einfachste Mittel um Threads zu synchronisieren. Alle Threads warten solange vor der Barriere, bis alle da sind. Unbedachte Benutzung von Barrieren kann die Performance vernichten. Die Syntax sieht wie folgt aus:

```
!$omp barrier
```

Ein paralleler Block hat am Ende immer eine impilize *barrier.* Diese kann mit explizitem *nowait* deaktiviert werden. Wichtig: am Anfang ist keine implizite *barrier.* Sollten die Threads nicht gleichzeitig gestartet worden sein, kann es zu gefährlichen *„Race Conditions“* führen (mehr dazu im Kapitel 7 „Analyse“).

Sequentielle Kapselung

In jedem Programm gibt es einen Teil, der logisch nicht parallel ausgeführt werden kann und darf. Entweder sind es algorithmische Details, oder technische, wie z.B. gleichzeitige Verwendung gemeinsamer Ressourcen.

e. single / master

Für die Kapselung des Codes, der von einem einzigen Thread ausgeführt werden soll, gibt es das

Konstrukt *single block.* Die von diesem Block umschlossene Anweisung wird von dem Thread ausgeführt, der sie als erstes erreicht. Da man in OpenMP nur eine relativ schwache Kontrolle über Threads und interne Variablen hat, gibt es hierfür das Konstrukt *master,* welches ähnlich funktioniert, der Block wird jedoch explizit vom Master ausgeführt. Sehr wichtig, beide Konstrukte implizieren am Ende jeweils eine Barriere. Eine Besonderheit von OpenMP liegt darin, dass nur Teile des Programms parallelisiert werden könnten, während der restliche Code unangetastet sequentiell bleibt und zur Zeit seiner Ausführung tatsächlich auch nur ein Thread existiert. Aus Performance Gründen ist es jedoch immer ratsam, statt zwei aufeinanderfolgenden parallelen Blöcken, die von wenigen sequentiellen Anweisungen getrennt werden, eine parallele Region mit einem *single* bzw. *master* Block einzuführen, da hier die redundante Threaderzeugung erspart bleibt.

f. critical section

Sollte es nicht möglich gewesen sein, die Parallelität zu erhalten, so kann auch eine *critical section* eingeführt werden. Der Unterschied zum *single* besteht darin, dass die *critical section* von allen Threads ausgeführt wird, jedoch nie gleichzeitig. Dieses Konstrukt muss nur sehr bedacht benutzt werden, da es quasi eine sequentielle Abarbeitung erzwingt.

g. Ressourcenverwaltung

Effizienter ist es sicherlich, immer alle Threads arbeiten zu lassen. Dabei ist es wichtig darauf zu achten, in wessen Gewalt sich die Ressourcen befinden. Alle Ressourcen sind standardmäßig gemeinsam (shared). Zur besseren Übersicht sollten sie jedoch immer explizit als solche gekennzeichnet werden. *private*, *firstprivate*, *lastprivate* geben die Möglichkeit für jeden Thread einen eigenen Datensatz zu reservieren. Beim *private* werden die Daten nicht initialisiert und existieren nur innerhalb des Blocks. Beim *firstprivate* werden die Daten mit dem letzten Wert vor dem parallelen Abschnitt initialisiert. Beim *lastprivate* wird der Wert der letzten Iteration außerhalb des Blocks herausgeschrieben. Bei beiden handelt es sich um private Daten, beides kann kombiniert werden. Der Laufindex einer Schleife ist immer intern *private*, auch wenn vom Programmierer als *shared* deklariert.

h. Schleifen

Die meiste Rechenlast findet sich fast immer bei iterativen Berechnungen. Von daher ist es meist die Hauptaufgabe, Schleifen möglichst effizient zu parallelisieren.
Fortran bietet eine Schleifenart, die DO-Schleife. Daraus lässt sich u.A. die DO WHILE Schleife ableiten, die hier nicht näher betrachtet wird. Entsprechend gibt es die DO-directive.

```
!$omp DO
    do_loop
!$omp END DO
```

Auch für diesen Block können alle Variablen als *shared*, *private* etc. deklariert werden.
Nicht jede Schleife kann in OpenMP parallelisiert werden. Schleifen, deren einzelne Iterationen voneinander abhängen, können nur bedingt bzw. gar nicht parallelisiert werden. Im allgemeinen muss eine Schleife dem *canonical shape* Konstrukt entsprechen. Das bedeutet, dass es zunächst keine Sprünge aus der Schleife geben darf, kein *return*, kein *break*, kein *exit*, kein *goto*. Außerdem ist nur + und – als In- bzw. Dekrementierung erlaubt. Als logische Operationen, die den Abbruch signalisieren sind nur < > = oder Kombinationen daraus erlaubt. Es sind keine logischen Verknüpfungen wie logisches „und“ und logisches „oder“ erlaubt. Im Anhang ist die vollständige Definition von *canonical shape*. Aus diesen Eigenschaften kann man darauf schließen, dass der Rahmen der Schleife zu ihrem Beginn feststehen muss. Es sind keine dynamischen Sprünge außerhalb der Schleife erlaubt, und auch keine dynamischen Grenzenänderungen. Der Start- und der Endwert der Schleife sind innerhalb der Schleife nicht änderbar. Man könnte auf die Idee kommen, eine Abbruchbedingung, die innerhalb der Schleife zum Austritt führen würde, durch eine *logical* (*bool*) Variablen umzuschreiben, und das Ende der Schleife logisch *.and.* damit zu verbinden, um den vorzeitigen Abbruch zu erzwingen. Dies ist jedoch nicht möglich, da kein logisches *.and.* erlaubt ist. Es ist eine relativ strenge Vorgabe, der längst nicht alle Schleifen entsprechen. Gleichzeitig ist es auch eines der Nachteile von OpenMP.

Oft findet man verschachtelte Schleifen vor. OpenMP bietet hierfür eine Option *collapse(x).* Damit werden *x* äußere Schleifen zusammengefasst und erst dann aufgeteilt. Die vermeintliche Effizienz wird an

Beispielen untersucht. In jedem Falle sollte ohne Benutzung von *collapse* immer möglichst die äußerste Schleife parallelisiert werden, sobald die Anwendung kein anderes Vorgehen impliziert.

i. Scheduling

Scheduling ist dafür verantwortlich, wie die Schleife auf die Threads verteilt wird. Mit

```
schedule(type, chunk)
```

kann man das Schema vorgeben. Dabei kann *type static, dynamic, guided, runtime und auto* sein. *chunk* ist eine Art Portionsgröße. Beim Typ *static* werden vor der Abarbeitung der Schleifen schon statisch alle Iterationen aufgeteilt. Dabei gibt es Blöcke fester Größe (chunk) und der Rest wird nach dem Prinzip Round-Robin aufgeteilt. Der letzte Block kann dabei logischerweise kleiner sein. *chunk* ist ein optionaler Parameter, bei keiner Angabe wird intern eine möglichst ausbalancierte Verteilung vorgenommen.
Bei *dynamic*, im Gegensatz zum *static*, steht vor dem Abarbeiten der Schleife nicht fest, welcher Thread welche Iterationen macht. Der Anfang funktioniert ähnlich wie *static*, erst wird jedem Threads *chunk* Iterationen zugewiesen, sobald ein Thread fertig ist, bekommt er wieder *chunk*-viele Iterationen, bis die Schleife abgearbeitet ist. Sollte kein *chunk* angegeben worden sein, wird der Bibliothek-interner Wert verwendet. Dynamic liefert eine bessere Lastverteilung vor allem dann, wenn die Rechenlast der Iterationen unterschiedlich ist.
Bei *guided* wird jedem Thread erst eine bestimmte Anzahl der Iterationen zugewiesen, dann eine kleinere, bis keine Iterationen mehr übrig sind. Die Blockgröße nimmt also ab. Dabei nimmt man irreführend an, dass *chunk* die Anfangsgröße eines Blocks ist. In Wirklichkeit wird *max(chunk, i/pn)* genommen, wobei *i* die Anzahl der verbliebenen Iterationen ist und *pn* die Anzahl der verfügbaren Threads. Die Iterationen werden insgesamt auf weniger Blöcke verteilt, als bei *guided*, die Berechnung der Blockgröße ist jedoch aufwendiger.
Die letzte Option ist *schedule(runtime)*. Hier wird die Strategie erst zur Laufzeit bestimmt. Man kann sie entweder mit

```
setenv OMP_SCHEDULE „schedule, chunk“
```

vor dem Programmstart selbst angeben, oder aber das Scheduling wird von der Bibliothek gesetzt, falls man nichts vorgibt.
Das Schedule *auto* lässt den Compiler zur Laufzeit ermittelb welche Strategie gewählt wird.
Sollte man keinerlei Angaben zum Scheduling gemacht haben, so wird der Bibliothek-interne Wert von *def-sched-var* verwendet, auf den der Programmierer keinen Einfluss mehr hat.

Es gibt keinen Test für das Scheduling. Die Einstellungen sind anwendungsabhängig und müssen oft einfach ausprobiert werden. Ist man sich der Aufteilung nicht sicher, ist es nicht verkehrt auf gute Umsetzung der Bibliothek zu hoffen, eher man mit ungeeigneten Parametern größeren Schaden anrichtet.

j. Defaults

Neben der festgelegten Rahmen gibt es im Konzept OpenMP auch vieles, was nicht eindeutig definiert ist. Hier kommt eine kleine Sammlung der im Standard nicht vorgeschriebenen „defaults“.
In erster Linie hat man bei OpenMP nur sehr schwache Kontrolle über die interne Arbeit, wenn man sie überhaupt hat (im Gegensatz zu PThreads und MPI). Daher ist es sehr wichtig zu wissen, welche Konfigurationen eingestellt sind, um zumindest bei kontrollierbaren Parametern die nötige Sicherheit zu haben.

Wie schon erwähnt, ist es nicht vorgeschrieben, mit wie vielen Threads das Programm gestartet wird, falls der Programmierer das nicht explizit angegeben hat. Auch bei *nested parallelism*, verschachteltem Parallelismus muss immer vor jedem Block explizit angegeben werden, wie viele Threads gestartet werden, wobei eine globale Begrenzung programmübergreifend nicht möglich ist. Beim *nestet parallelism* ist auch nicht definiert, auf wie vielen Ebenen verschachtelt werden kann.
Beim *scheduling* ist ebenfalls nicht definiert welchen Wert *chunk* bei *guided* Einstellungen hat, man sollte sich nicht darauf verlassen, dass es 1 ist. Das *runtime* Scheduling ist ebenfalls nicht fest definiert, man hat auch keinen Einfluss darauf, und muss sich auf die Intelligenz der Bibliotheken verlassen können. Das *Default schedulung* an sich ist ebenfalls nicht standardisiert.

Trotz der Empfehlung in dieser Ausarbeitung kein *include* '*omp_lib.h*' zu verwenden, ist es dennoch nicht standardisiert.

k. Race Conditions

OpenMP ist auf Kosten der Kontrolle über die Threads recht einfach zu benutzten. Die meisten Fehler sind dabei auf so genannte *Race Conditions* zurückzuführen, die zu falschen Ergebnissen oder ggf. zum Deadlock führen. *Race Conditions* sind Wettlaufsituation. Das gefährliche dabei ist, dass das Ergebnis davon abhängt, welcher Thread als erster die Ressource erreicht hat. Die zeitliche Abhängigkeit ist vor allem beim Fehlerbehandeln ein großes Problem, da solches Verhalten kaum reproduzierbar ist. Ein sehr gutes, einfach zu handhabendes Werkzeug zum Auffinden der *Race Conditions* ist der *Thread Checker* von Intel, der gerade auf das Problem spezialisiert ist. Bei tieferem Verständnis helfen natürlich auch „normale“ Debugger.
Einige der häufigen Fälle werden näher im Kapitel Beispiele untersucht.

l. workshare

Um *forall* Statements oder Blöcke zu parallelisieren sieht OpenMP ab der Version 2.0 und nur für Fortran die Direktive *workshare* vor. Diese Direktive wird auch nur vom GNU Compiler unterstützt, Intel geift diese Idee nicht auf. Hierbei werden nicht die Iterationen, sondern die Arbeit auf die Threads aufgeteilt, es entstehen einzelne *work units*. Jede Arbeitseinheit ist dabei genau einem Thread zugewiesen. Das *workshare* kann nicht nur speziell für *forall* benutzt werden, sondern ist für jegliche Art von Arrayzuweisungen gedacht. Die Reihenfolge der Abarbeitung darf keine Rolle spielen (z.B. zwischendurch Aufsummieren oder einfache direkte Zuweisung), sonst darf *forall* nicht parallelisiert werden.
Die Tests sind leider in dieser Ausarbeitung gescheitert, da der Compiler offensichtlich das *workshare* nicht umsetzen kann. Weil das Konstrukt nicht weit verbreitet ist, wird an dieser Stelle von der Benutzung oder zumindest der parallelen Nutzung abgeraten.

6.2 MPI

Message Passing Interface (MPI) ist ein Standard für paralleles Programmieren mittels Nachrichtenaustausch. Das Konzept wird in erster Linie für Architekturen mit verteiltem Speicher oder mit verteilten Nodes mit gemeinsamem Speicher verwendet.

MPI ist die dominante Vorgehensweise des parallelen Rechnens weltweit, wird von Fortran und C/C++ unterstützt. Interessant hierbei wäre die Gegenüberstellung von Kommunikationsarten:

- *blockierend point-to-point vs. nichtblockierend point-to-point*
- *kollektiv vs. One-sided*

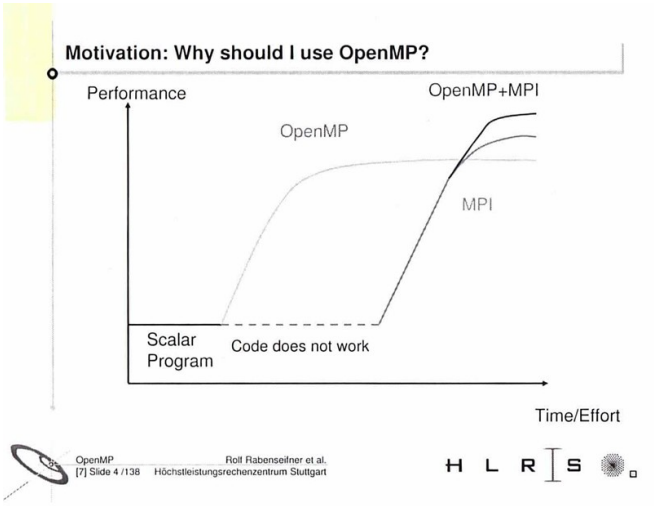
und Herstellern:

- *Intel Compiler vs. GNU Compiler*
- *mpich Bibliothek vs. openmpi Bibliothek*

6.3 Hybrid

Hybrides Programmieren meint einen Mix aus verschiedenen Konzepten, hier ein Mix aus OpenMP und MPI. Auf den ersten Blick mag es logisch sein, auf OpenMP verzichten zu wollen, weil MPI den gemeinsamen Speicher gut beherrscht. Man wird allerdings schnell merken, dass MPI einen enorm höheren Aufwand an Arbeit verlangt, im Gegensatz zu OpenMP. Aber nicht nur „Sparfüchse“ sind an hybrider Programmierung interessiert. Ein solcher Mix kann oft deutliche Effizienzvorteile schaffen. Man könnte nun annehmen, das Erfolgsrezept wäre, so viele MPI Prozesse zu starten, wieviele Nodes man verfügbar hat, und auf diesen jeweils mit maximal möglicher Threadzahl zu arbeiten. Man wird aber

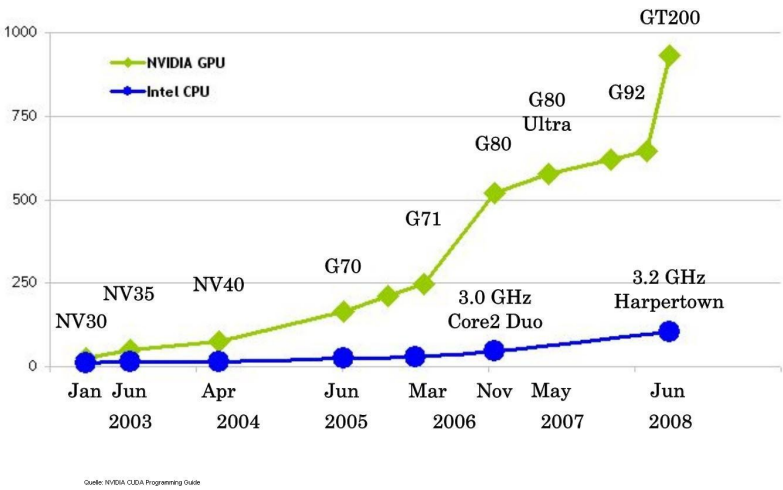
schnell herausfinden, dass immer eine Anwendungs- und architekturspezifische Balance hergestellt werden sollte.
Die folgende Grafik ist nicht als allgemeingültig anzusehen, zeigt aber die zusammenfassende Tendenz. Die hybride Programmierung kann bei bedachtem Einsatz mehr Gewinn bringen, als reines MPI. Es lohnt sich genauer zu untersuchen, wie man die Balance zwischen MPI und OpenMP wählen sollte.



6.4 GPU – Programmierung: CUDA/OpenCL

Die auf einer Grafikkarte befindliche GPU (*Graphics Processing Unit)* wird oft als eine Art Beschleuniger für parallel abgearbeitete Programme benutzt. Die GPU ist vom internen Aufbau ungeschlagen in der Anzahl der Operationen, die pro Sekunde berechnet werden können. Während eine CPU einige Dutzend Threads effizient einsetzen kann, sind es bei einer GPU gleich mehrere Tausend.

Das führende Konzept für GPU Programmierung ist zur Zeit CUDA (Compute Unified Device Architecture), eine von Nvidia entwickelte Technik. CUDA setzt natürlich entsprechend voraus, dass man Nvidia Hardware hat. Alternativ dazu existiert das Konzept OpenCL (Open Computing Language), eine Technik, die Flexibilität und Hardwareunabhängigkeit weitgehend versucht zu realisieren. Es soll für GPUs diverser Hersteller (Nvidia, AMD, VIA, etc.) verwendet werden können. Beide Konzepte basieren auf C, wobei es CUDA auf Wrapper aufbauend für andere Sprachen gibt. OpenCL hat gegenüber C/C++ gewisse Einschränkungen (keine Rekursion, keine variable Array-Größe, keine Pointer auf Funktionen).



NVIDIA CUDA Programming Guide

Der Quellcode für Programmierung der Kernel wird von einem speziellen Compiler in virtuelle Maschinensprache übersetzt, der Rest des Codes wird wie gewohnt von einem C/C++ Compiler erledigt.

Virtueller Maschinencode wird zur Laufzeit in Code für die konkrete GPU übersetzt. Hierfür sind weitere Spracherweiterungen und Bibliotheken nötig (z.B. Brook+).

Die Grafik zeigt die Entwicklung von Nvidia GPU im Vergleich zu einer Intel CPU. Auf der y-Achse ist Peak in GFLOPS/s abgebildet. Die GPU ist extrem schnell, aber auch sehr teuer. Viele führenden Höchstleistungsrechner setzen auf GPUs. Die Benutzbarkeit ist jedoch umstritten. Nicht sehr viele Anwendungen können aufgrund des hohen Programmieraufwandes eine GPU benutzten. Ein Rechner mit einem Prozessorpaar, einer CPU und einer GPU, kann somit nicht gut ausgelastet werden.

7. Analyse

Die im nächsten Kapitel folgenden Beispiele werden getestet und analysiert. Dabei wird es in erster Linie um den Zeitbedarf gehen.
Die verwendete Hardware:

Cluster[3] aus 10 Knoten, jeder Knoten besitzt folgende Ausstattung:
2 Prozessoren (Intel Xeon Westmere 5650 @ 2.67GHz) mit jeweils 6 cores
12 GByte DDR3 / PC1333 Hauptspeicher (System taktet mit 1333 MHz)
2 x Gigabit-Ethernet

Compiler:
GNU Fortran (Ubuntu 4.4.3-4ubuntu5.1) 4.4.3 Compiler
Optimierungen:
O3

Die Programme werden mit O3 kompiliert, es erfolgt keine weitere Optimierung, sofern nicht anders vermerkt. In speziellen Fällen kann das O3 den Zeitgewinn verstecken. Das Kompilieren mit -fopenmp schaltet intern einige üblichen Compileroptimierungen aus, so dass beim direkten Vergleich OpenMP auf den ersten Blick keinen Gewinn bringt. Der Gewinn des Parallelisierens kann gleich dem Verlust durch entfallene Optimierung sein. Besonders wichtig kann es beim Vergleich MPI vs. Hybrid (MPI+OpenMP) sein. Auf einigen CRAY-Maschinen sind Optimierungen, die durch OpenMP nicht mehr möglich sind, grundsätzlich abgeschaltet, auf den meisten anderen jedoch nicht. Die folgenden Testfälle bleiben davon unberührt. Grundsätzlich sollte man nur optimierte Programme auf den Gewinn durch Parallelität testen.
In den Beispielen wird möglichst auf Seiteneffekte, die z.B. durch Systemaufrufe wie *write* o.ä. hervorgerufen werden, verzichtet.

Es werden immer 3 gleiche Tests durchgeführt und deren Mittelwert gebildet, soweit einer der Werte nicht signifikant abweicht.
Die Zeitmessungen sollten nie auf dem Masterknoten durchgeführt werden, da die Aktivität des Knotens Ergebnisse schnell verfälschen kann. Wegen technischer Defekte war es jedoch nicht möglich andere Knoten zu benutzen.
Hauptsächlich handelt es sich um einzelne Schleifen, die sich an die realen Anwendungen anlehnen. Um aussagekräftige Ergebnisse zu bekommen, wird eine Hilfsschleife um die Berechnung gebaut, die lediglich eine wiederholte Ausführung der Berechnungen bewirkt. Wenn es sich um eine solche Schleife handelt, wird dies entsprechend vermerkt. Die Hilfsschleife wird nicht parallelisiert, denn sie hängt nicht mit der Berechnung zusammen und führt somit zu *Race Conditions*. Der Nachteil eines solchen Konstrukts ist die redundante Erzeugung und Zerstörung von Threads in jeder Iteration der Hilfsschleife.

3 (http://wr.informatik.uni-hamburg.de/teaching/ressourcen/cluster)

```
1  program race
2     !NICHT NACHMACHEN
3     USE omp_lib
4     implicit none
5     INTEGER , PARAMETER :: second = 1000
6     INTEGER , PARAMETER :: third = 1000
7     INTEGER , PARAMETER :: first = 1000
8     INTEGER :: one, two, three
9     INTEGER , DIMENSION (1:second, 1:third) :: matrix = 10
10
11    !$omp parallel num_threads(12)
12    !$omp do
13           DO one = 1, first
14              DO three= 1, third
15                 DO two = 1, second
16                     matrix(two, three) = matrix(two, three)       &
17                                          + two + three
18                 END DO
19              END DO
20           END DO
21    !$omp end do
22    !$omp end parallel
23    write(*,*) matrix(30,30)
24
25 END program race
```

Hinweis für die Zukunft: nicht bei der Deklaration initialisieren !

!Hilfsschleife

	falsch	richtig (Hilfsschleife außerhalb des parallelen Blocks)
Ergebnis:	immer unterschiedlich	60010

Der Overhead für die Threaderzegung bei 1000 äußeren Iterationen liegt unter 1 Sekunde und wird im folgenden vernachlässigt.

Die Laufzeit kann mit verschiedenen Funktionen gemessen werden. OpenMP bietet eine Funktion zum messen der Zeit pro Thread.

```
real :: start, end_t
start = omp_get_wtime()
end = omp_get_wtime()
write(*,*) end_t - start
```

Man kann auch die gesamte CPU Zeit messen:

```
real :: start, end_t
call cpu_time(start)
call cpu_time(end_t)
write(*,*) end_t - start
```

Um die gesamte Ausführungszeit zu messen, kann man das Programm auch mit time ./run ausführen. Letztes wurde in dieser Ausarbeitung verwendet.

8. Beispiele

8.1 Array-Initialisierung

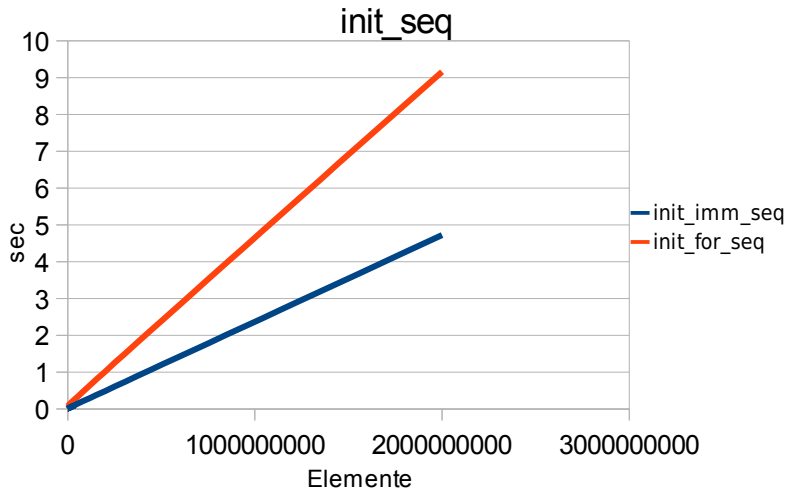
Ein Array kann in Fortran auf zwei unterschiedliche Weisen initialisiert werden. Das „altmodische“ Verfahren in einer DO-Schleife, und das neue Konzept der einfachen Zuweisung.

```
1  program init_imm_seq
2     implicit none
3
4     INTEGER, DIMENSION(2000000000) :: a
5     a = 0
6
7  END program  init_imm_seq
```

ist äquivalent zu

```
1  program init_for_seq
2     implicit none
3
4     INTEGER, DIMENSION(2000000000) :: a
5     INTEGER :: i
6
7     DO i = 1, 2000000000
8        a(i) = 0
9     END DO
10
11 END program init_for_seq
12
```

Die folgende Grafik zeigt den Zeitverbrauch, abhängig von der Größe des Arrays. Kleine Datenmengen zeigen einen wenig beachtlichen Unterschied auf, also treiben wir es auf die noch mögliche Spitze (2Mrd. Elemente). Der Einzeiler zeigt sich als verdienter Sieger mit 4,72 sec gegen 9,16 sec. Man bedenke, dass die Kompilierzeit für 2Mrd. Elemente großes Array ca. eine halbe Minute dauert. Nur der Masterknoten verfügte über genügend Speicher, um ein solches Array überhaupt deklarieren zu können.



Elemente	init_imm_seq	init_for_seq
200000	0,003	0,004
2000000	0,01	0,02
20000000	0,06	0,11
200000000	0,49	1,02
2000000000	4,72	9,16

Das interne Initialisieren ist also nicht zwingend das Verstecken der gewöhnlichen DO-Schleife, es gibt einen offensichtlichen Performance-Unterschied. Einen Einzeiler kann man jedoch nicht parallelisieren, im Gegensatz zu einer gewöhnlichen Schleife, die in *canonical shape* vorliegt. Bei absoluten Zeiten von wenigen Sekunden sollte man sich wenig Gedanken um MPI machen, denn eine Array-Initialisierung wird kaum auf tausenden Prozessoren ausgeführt werden. Mit OpenMP könnte man jedoch grundsätzlich untersuchen, ob und ab welcher Threadzahl eine parallelisierte Schleife schneller ist, als der Einzeiler. Es

lohnt sich, wenn überhaupt für den letzten Messwert. Leider lässt aber die Implementierung von OpenMP nicht so viel Speicher allokieren, wie die Arrays benötigen. Das verhindert schon alleine der Compiler. Kompiliert man eine der sequentiellen Versionen mit dem Flag *-fopenmp*, so lässt sich diese nicht mehr ausführen, ungeachtet dessen, ob omp Direktiven im Programm vorkommen, oder nicht. Erst für eine viel niedrigere Anzahl der Elemente (ca. 20K) kann man die Initialisirungsschleife parallelisieren. In diesem Bereich sind die absoluten Zeiten aber so niedrig, dass sich die Mühe nicht lohnt.

Fazit: der Einzeiler als Initialisierung lohnt sich. Er sieht eleganter aus, ist effizienter, spart eine Laufindexvaribale und verleitet den Laien nicht dazu, die Schleife zu parallelisieren und auf oben genannte Probleme zu stoßen. Es ist hinnehmbar, dass OpenMP solche Einschränkungen hat. Im Normalfall der Anwendungen kommen so große Arrays nicht vor. Es wäre jedoch gut zu wissen, wo man diese Konfigurationen nachschlagen und ggf. verändern kann.

8.2 Race conditions

a. Abhängige Iterationen

Um mit OpenMP eine Schleife zu parallelisieren, müssen u.A. die Iterationen unabhängig voneinander sein. Eine Abhängigkeit wäre z.B. das Zugreifen der n-ten Iteration auf die in der vorigen Iteration berechneten Werte. Die OpenMP Bibliothek teilt die gegebene Schleife intern auf die verfügbaren Threads auf. Es ist nicht automatisch gewährleistet, dass der Wert der logisch gesehen früheren Iteration tatsächlich früher berechnet wird, als der Wert der späteren Iteration. Es kann passieren, dass alte Werte oder sogar noch gar nicht berechnete Werte benutzt werden.
Ein solches Programm kann wie folgt aussehen:

```
1   program fix_me
2      USE omp_lib
3      implicit none
4      ! NICHT NACHMACHEN
5      INTEGER  , PARAMETER :: m_end = 10000000
6      INTEGER :: in, it
7      INTEGER :: it_stop = 1000
8      INTEGER , DIMENSION (1:m_end) :: matrix_a = 10
9      INTEGER , DIMENSION (1:m_end) :: matrix_b = 0
10
11     DO it = 1, it_stop                                    !Hilfsschleife
12     !$omp parallel num_threads(2)
13     !$omp do
14        DO in = 2, m_end
15           matrix_a(in) = 2 * in * (in - 1)
16           matrix_b(in) = matrix_a(in) - matrix_a(in-1)
17        END DO
18     !$omp end do
19     !$omp end parallel
20     END DO
21
22  END program fix_me
23
```

Der Thread Checker würde nun ein *Write → Read data-race* aufzeigen, und zwar steht der Lesezugriff in der Zeile 16 im Konflikt mit dem Schreibzugriff der vorigen Iteration in der Zeile 15.

```
15              matrix_a(in) = 2 * in * (in - 1)
16              matrix_b(in) = matrix_a(in) – matrix_a(in-1)
```

Eine Lösung wäre, die beiden Zeilen in jeweils einer eigenen Schleife zu berechnen, doch dabei würde der Cachegewinn verloren gehen. Eine solche *Race condition* tritt nur an den Übergängen auf, wo die Grenzen der Threadgebiete sind. Ansonsten werden die benötigten Werte von demselben Thread berechnet und befinden sich in seinem Cache. Die beiden Schleifen könnte man natürlich dafür getrennt parallelisieren.

Eine elegantere Lösung wäre den benötigten Wert nicht auszulesen, sondern redundant berechnen:

```
15              matrix_a(in) = 2 * in * (in - 1)
16              matrix_b(in) = matrix_a(in) - 2 * (in -1) *(in - 2)
```

In diesem Fall sind die Iterationen nun unabhängig, natürlich auf Kosten der redundanten Berechnung. Folgend wird die Performance der beiden Lösungen untersucht.

Die sequentielle Ausführung der eleganteren Lösung ist mit *16,5 sec* nahezu identisch mit der Zeitmessung des ursprünglichen Programms. Es bringt hier keine Performancenachteile mit sich, den Wert redundant zu berechnen. Die weniger elegante Lösung mit zwei Schleifen braucht in sequentieller Ausführung *18 sec*, und ist auch parallelisiert maximal 1 Sekunde langsamer. Die äußere Hilfsschleife kann dabei außerhalb des kompletten Blocks sein (*fixed_non_elegant_1*), oder aber sie wird auch doppelt aufgeführt (*fixed_non_elegant_2*). Im letzteren Fall riskiert man noch mehr Overhead durch dopplet so viele Kosten für Erzeugung und Zerstörung der Threads. Wie man hier sieht liegt dieser Overhead bei *<0,5 sec* für jeweils 1000 Iterationen.

fix_me (16,5sec)	fixed_elegant (sec)	fixed_non_elegant_1 (sec)	fixed_non_elegant_2 (sec)
seq	17,0	17,9	18,3
omp(2)	9,1	9,4	9,6
omp(4)	6,9	7,4	7,9
omp(12)	5,3	6,5	6,9

Der Schlüssel zum Erfolg liegt darin, dass sich die redundante Berechnung nicht spürbar auswirkt. Es ist somit gleich teuer auf den Speicher zuzugreifen oder die Berechnung zu machen. Wie sieht es aber aus, wenn man teurere Rechenoperationen hat? Es folgt ein Test mit *sinus*:

```
1   program fix_me_real
2      implicit none
3      !NICHT NACHMACHEN
4      INTEGER  , PARAMETER :: m_end = 100000
5      INTEGER :: in, it
6      INTEGER :: it_stop = 1000
7      REAL , DIMENSION (1:m_end) :: matrix_a = 10.0
8      REAL , DIMENSION (1:m_end) :: matrix_b = 0.0
9
10     DO it = 1, it_stop
11       DO in = 2, m_end
12         matrix_a(in) = sin(REAL(in)) * 2.0
13           matrix_b(in) = matrix_a(in) - matrix_a(in-1)
14        END DO
15     END DO
16
17  END program fix_me_real
```

Zum Parallelisieren werden ebenfalls beide Lösungen getestet. Hier sieht die „weniger elegante“ Lösung besser aus. Die Zeit im Vergleich zum sequentiellen ursprünglichen Programm ist etwa gleich. Wie erwartet ist jedoch der Wert der Lösung mit der redundanten Berechnung doppelt so hoch. Man sollte bei einer solchen Schleife abwägen, wie teuer die Berechnung gegenüber dem Speicherzugriff (austesten oder Zugriffszeiten nachlesen) ist und entsprechend eine der Varianten auswählen. Dabei ist es wichtig welcher Art die Speicherzugriffe sind. Die getesteten Programme liefen auf einem Knoten mit gemeinsamem lokalen Speicher, die Zugriffe sind günstiger. Wären die Daten im Speicher eines anderen Knoten, müsste man untersuchen, ob die redundante Berechnung günstiger ist. Die *fixed_elegant* Variante kann man aber sicherlich noch mit Cacheoptimierung verbessern.
Auffällig ist, dass die Threaderzeugung und Zerstörung die Zeiten kaum beeinflussen. Bei der maximalen Zahl von 12 Threads ist der Overhead bei 1000 Iterationen nahezu vernachlässigbar.

fix_me_real (41,8)	fixed_elegant (sec)	fixed_non_el_1 (sec)	fixed_non_el_2 (sec)
seq	81,9	41,5	41,2
omp(2)	43,4	21,7	21,9
omp(4)	21,7	11,2	11,6
omp(12)	8	4,3	4,4

Die Sinusberechnung ist viel teurer als eine Multiplikation. Die Werte in den beiden Tabellen kann man jedoch nicht direkt vergleichen, um abzuschätzen, wie viel teurer die Berechnung ist. Einerseits musste die Arraygröße bei der komplexeren Berechnung von *10Mio.* auf *100.000* reduziert werden, um Zeiten im Bereich von einer Minute zu halten. Die Zeit für *fix_me_int* bei *100.000* Elementen großem Array beträgt nur *0,2 sec*. Der Unterschied liegt somit bei einem Faktor von ca. 200. Auch hier darf man keinen direkten Vergleich machen, da zum einen die Zeit von *0,2 sec* zu gering ist, um Messfehler, Seiteneffekte oder ggf. Aktivität auf dem Knoten zu tolerieren, zum anderen geht es um den Typ des Arrays. Bei der simpleren Berechnung handelte es sich um zwei Integer Arrays, alle Operationen wurden mit Integer Werten durchgeführt. Deklariert und initialisiert man die beiden Arrays von *fix_me_int* als *REAL*, macht die Berechnung aber weiter mit einem Integer 2, so steigt die Zeit von *16,2 sec* auf *19,2 sec*. Rechnet man dann auch typkonform mit einem *REAL 2.0*, so steigt die Zeit bei derselben Berechnung mit Muliplikation von *16,2 sec* auf *21,5 sec*. Die Rechnung mit Integer oder Real unterscheidet sich bishin zur Hardwarebene. Registernutzung und Auswertung werden hier aber nicht thematisiert.
Man muss bedenken, dass Sinus von den mathematischen Funktionen nicht die teuerste ist, Wurzel und Divisionen würden ggf. einen noch größeren Unterschied bewirken.

Die Auflösung der *Race conditions* durch die zweite Schleife sieht wie folgt aus:

```
1    program fixed_real
2       USE omp_lib
3       implicit none
4
5       INTEGER , PARAMETER :: m_end = 100000
6       INTEGER :: in, it_begin
7       INTEGER :: it_stop = 1000
8       REAL , DIMENSION (1:m_end) :: matrix_a = 10.0
9       REAL , DIMENSION (1:m_end) :: matrix_b = 0.0
10
11      DO it_begin = 1, it_stop                                !Hilfsschleife
12      !$omp parallel num_threads(12)
13      !$omp do
14         DO in = 2, m_end
15            matrix_a(in) = sin(REAL(in)) * 2.0
16         END DO
17      !$omp end do
18      !     !$omp barrier
19      !$omp do
20         DO in = 2, m_end
21         matrix_b(in) = matrix_a(in) - matrix_a(in-1)
22         END DO
23      !$omp end do
24      !$omp end parallel
25      END DO
26
27   END program fixed_real
28
```

b. barrier

In dem oberen Beispiel mit 2 inneren Schleifen ist die *barrier* auskommentiert. Am Ende des parallelen Blocks (*end do*) ist eine implizite *barrier* eingebaut, eine explizite Barriere erübrigt sich somit. Gäbe es an dieser Stelle keine Synchronisierung, hätte man das Problem der *Race conditions* nicht gelöst.
OpenMP bietet zum Auflösen der impliziten *barrier* eine Option *nowait an*. Das kann durchaus nützlich sein, denn bei ungleicher Lastverteilung in der Schleife werden Threads, die mit ihrer Arbeit fertig sind, durch die *barrier* aufgehalten. Oft ist es beabsichtigt, denn das Aufhalten bedeutet gleichzeitig die Sicherstellung der vollständigen Abarbeitung der Schleife, bevor der weitere Code ausgeführt wird. Zwei **direkt** aufeinander folgende Schleifen müssen bei Abhängigkeiten durch eine *barrier*, ob implizite oder explizite, getrennt werden. Kommt dazwischen jedoch eine oder mehrere Anweisungen, die unabhängig von der kompletten Abarbeitung der vorhergehenden Schleife sind, so kann man die erste Schleife mit *nowait* versehen. Es lohnt sich vor allem, wenn die Anweisungen zwischen den Schleifen von einem Thread ausgeführt werden sollen. Hier kann man einen *single* Block einführen, der von dem Thread ausgeführt wird, der mit der ersten Schleife am schnellsten fertig war. Vorausgesetzt wird, dass auch der *single* Block nicht von der ersten Schleife abhängig ist.

```
1  program fixed_nowait
2     USE omp_lib
3     implicit none
4
5     INTEGER , PARAMETER :: m_end = 1000000
6     INTEGER :: i1, i2, i3, i4, i5, i6, tid
7     INTEGER :: it_stop = 1000
8     REAL , DIMENSION (1:m_end) :: matrix_a = 10.0
9     REAL , DIMENSION (1:m_end) :: matrix_b = 0.0
10    REAL , DIMENSION (1:10000, 1:10000)  :: dummy = 0.0
11
12    !$omp parallel num_threads(6), private (tid)
13    tid = OMP_GET_THREAD_NUM()
14
15    !$omp do
16    DO i1 = 1, it_stop                                    !Erste Schleife
17       DO i2 = 2, m_end-it_stop
18          matrix_a(i1+i2) = sqrt(REAL(i2)) * 2.0
19
20       IF (tid > 2) THEN                                  !Lastungleichheit (extra eingebaut
21          DO i3 = 1, 50                                   zum Testen), die erste 3 Threads
22             matrix_b(i1) = sqrt(REAL(tid+i3))            arbeiten das nicht ab
23          END DO
24       END IF
25       END DO
26    END DO
27    !$omp end do nowait
28
29     write (*,*) 'check1'                                 !Die 3 Threads, die den IF Block
30                                                         ignorieren, können wegen dem
31    !$omp single                                         nowait  direkt weitermachen. Einer
32    write (*,*) 'single'                                 von den geht in den single Block,
33    DO i6=1, 10                                          der Rest wartet an der impliziten
34       DO i4=1, 10000                                    barrier vom single. Auch wenn nur
35          DO i5 = 1, 10000                               ein Thread den Block abarbeitet,
36             dummy(i4,i5) =  i4 * i5                     die Barrier gilt für alle. Der
37          END DO                                         dummy Aufruf ist nur konstruiert
38       END DO                                            um zeitlichen Bedarf des Blocks zu
39    END DO                                               imitieren, auf die Schleifen soll
40    write (*,*) 'end_single'                             nicht eingegangen werden. Es
41    !$omp end single !nowait                             könnte auch nur eine zeitintensive
42                                                         Anweisung drin stehen.
43    write (*,*) 'check 2'
44
45    !$omp do                                             Wenn der single Block und die
46    DO i1 = 1, it_stop                                   erste Schleife abgearbeitet sind,
47       matrix_b(i1) = matrix_a(i1) + sin(REAL(i1))       können alle die zweite Schleife in
48    END DO                                               Angriff nehmen.
49    !$omp end do
50
51    !$omp end parallel
52
53    write(*,*) matrix_b(200)
54    write(*,*) dummy(10000,1000)
55 END program fixed_nowait
```

Es gibt 3 Möglichkeiten, wie das Programm richtig abgearbeitet werden kann. Man kann alle *barriers* unberührt lassen. Man kann die erste *barrier* auflösen (wie im Beispiel), die zweite muss aber zwingend bleiben. Man kann aber auch die erste *barrier* erhalten und die zweite auflösen. Eine der *barrier*s muss auf alle Fälle vorhanden sein, welche oder sogar ob beide, ist nur eine Frage der Performance.

```
!nowait !nowait     83,3 sec
!nowait  nowait     82,3 sec
 nowait !nowait     58,6 sec
```

Man sieht einen erheblichen Unterschied. Lässt man die erste *barrier* drin, so werden aufgrund der Lastungleichheit in der ersten Schleife die ersten 3 Threads viel schneller fertig und müssen an der *barrier* warten. Lässt man auch die zweite *barrier* drin, so müssen nach der kompletten Abarbeitung der ersten Schleife alle Threads auf den einen warten, der den *single* Block (ca. *13 sec*) abarbeitet. Die zweite Schleife nimmt offensichtlich wenig Zeit in Anspruch und bewirkt keinen großen Unterschied. Einen erheblichen Unterschied macht es aber, wenn die erste Barriere aufgelöst wird und der *single* Block parallel zum IF-Block der ersten Schleife abgearbeitet werden kann.

c. private

Ein beliebter Fehler, der zu *Race conditions* führt, ist die fehlende Deklarierung der Variablen in einem parallelen Block als *private* Variablen. Welche privat sind, und welche *shared* hängt natürlich von der Anwendung ab.
Folgendes Beispiel zeigt, dass *private* allein nicht immer die richtige Lösung des Problems ist.

```
1   program no_private
2
3      USE omp_lib
4      implicit none
5
6      INTEGER, DIMENSION(10000000) :: arr = 1
7      INTEGER :: i, j, sum = 0
8
9
10     !$omp parallel num_threads(100)
11     !$omp do
12     DO i = 1, 100000
13        sum = sum + arr(i)
14     END DO
15     !$omp end do
16
17     !$omp end no_parallel
18
19     write(*,*) sum
20
21  END program no_private
22
```

Schon bei mehrmaligem Ausführen merkt man, dass unterschiedliche Ergebnisse rauskommen, 42000, 58000, 62000 etc. Sequentiell, wie auch erwartet, wäre die richtige Lösung: 100000. Alle Threads teilen sich eine Variable, das Lesen und Schreiben steht somit jedes mal im Konflikt, daher auch falsche, vor allem laufzeitabhängige Ergebnisse. Die intuitive Lösung wäre daher die Summe zu „privatisieren“, so wären die *Race conditions* beseitigt:

```
1   program private
2
3      USE omp_lib
4      implicit none
5
6      INTEGER, DIMENSION(100000) :: arr = 1
7      INTEGER :: i, sum = 0
8
9      !$omp parallel num_threads(100),    &
10     firstprivate (sum)
11
12     !$omp do
13     DO i = 1, 100000
14        sum = sum + arr(i)
15     END DO
16     !$omp end do
17     write(*,*) sum
18     !$omp end parallel
19
20  END program private
```

```
!hier sum „privatisieren“, i ist default
private
! WICHTIG: sum muss im parallelen Block
initialisiert werden, oder außerhalb, jedoch
mit firstprivate(sum)
```

Mit der neueren Version vom *Thread Checker* gibt es nun keine Warnungen mehr. Die ältere gibt überraschenderweise einen Fehler aus, der darauf hinweist, *sum* dürfe nicht privat sein, da seriell initialisiert sein sollte. Das wurde behoben.

Dennoch kommt ein unerwartetes Ergebnis. Mit *private* hat man das Problem der *Race conditions* gelöst, jedoch den Zweck der Berechnung verfälscht. Man bekommt lediglich die Teilsummen der jeweiligen Threads, nicht die Gesamtsumme. Es gibt hierzu zwei Lösungen. Man beachte, die folgenden Lösungen sind je um eine äußere Schleife erweitert, zum Zweck der aussagekräftigen Zeitmessung. Diese Schleife hat inhaltlich keinen Zusammenhang mit dem Problem.

```
program private_all

   USE omp_lib
   implicit none

   INTEGER, DIMENSION(10000000) :: arr = 1
   INTEGER :: i, j, sum
   INTEGER :: allsum = 0
   INTEGER :: sum = 0

   DO j = 1, 10000      !Hilfsschleife

   !$omp parallel num_threads(10),  &
   firstprivate (sum)
   !$omp do
      DO i = 1, 10000000
         sum = sum + arr(i)
      END DO
   !$omp end do
   !$omp critical
   allsum = allsum + sum
   !$omp end critical
   !$omp end parallel

   END DO               !Hilfsschleife

   write (*,*) allsum

END program private_all
```

```
program private_red

   USE omp_lib
   implicit none

   INTEGER, DIMENSION(10000000) :: arr = 1
   INTEGER :: i, j, sum = 0

   DO j =1, 10000      !Hilfsschleife

   !$omp parallel num_threads(10),     &
   reduction(+:sum)
   !$omp do
      DO i = 1, 10000000
         sum = sum + arr(i)
      END DO
   !$omp end do
   !$omp end parallel

   END DO               !Hilfsschleife

   write(*,*) sum

END program private_red
```

Beide Versionen liefern das „korrekte“ Ergebnis 1215752192. Man erwartet zwar 1x10^11, der Integer läuft jedoch über. Die äußere Hilfsschleife sprengt den Rahmen des 32-bit Integers. Wichtig bei diesem Test ist jedoch, dass das Ergebnis nicht vom sequentiellen abweicht, auch wenn die eigentliche Summe falsch ist.

threads_num	private_all (sec)	private_red (sec)
1	52,3	51,8
2 6 12	27,3 15,5 13,0	26,8 16,2 13,8

Beide Lösungen verhalten sich nahezu gleich. Interessant wäre jedoch einen Test mit einer sehr hohen Anzahl an Threads durchzuführen und zu untersuchen, ob die *critical section* sich negativ auswirken würde.

8.3 Verschachtelte Schleifen

a. Optimierung

Bevor man eine (mehrmals) verschachtelte Schleife parallelisiert, sollte man unbedingt die Speicherzugriffe optimieren. Hierbei ist es sehr wichtig zu wissen, dass in Fortran ein Array spaltenweise abgespeichert wird (im Gegensatz zu C/C++). Die Reihenfolge der Indizes einer verschachtelten Schleife muss umgekehrt zur Reihenfolge der Dimensionen des Arrays sein. Einfaches Beispiel:

```
 . . .

INTEGER , DIMENSION (1:one, 1:two, 1:three) :: dummy

DO i = 1, three
   DO j = 1, two
      DO h = 1, three
         [Anweisungen]
      END DO
   END DO
```

```
!Umgekehrte
Indiziesreihenfolge
```

```
END DO
. . .
```

An einem Beispiel aus dem realen Projekt „Harmonische Analyse der Gezeiten“ wird eine solche Optimierung erklärt und vorgenommen.

```
   ...
1  !--- Determine sum of cosine and sine constituents
2          DO t = 1, sizeTimeDim                          !hier : 8760
3             DO k = 1, numConstituents                   !hier : 33
4                DO d = 1, SIZE( inData , 3 )             !hier : 1
5                   DO j = 1, SIZE( inData , 2 )          !hier : 220
6                      DO i = 1, SIZE( inData , 1 )       !hier : 256
7                         IF ( inData(i,j,d,t) > lbtreshhold ) THEN
8
9                                 ycos_ysin(i,j,d,k)%co = ycos_ysin(i,j,d,k)%co  &
10                                                      + inData(i,j,d,t)     &
11                                                      * cos_sin(k,t)%co
12
13                                ycos_ysin(i,j,d,k)%si = ycos_ysin(i,j,d,k)%si  &
14                                                      + inData(i,j,d,t)     &
15                                                      * cos_sin(k,t)%si
16                        END IF
17                   END DO
18                END DO
19             END DO
20          END DO
   ...
```

Man sollte grundsätzlich nie eine Schleife einführen, die nur einen Durchlauf macht. In dieser Anwendung ist das *d* jedoch immer eingabeabhängig und nicht immer 1.
Ein IF-Block innerhalb der Schleife sollte näher untersucht werden. Die Abfrage im IF Block hängt von 4 Indizes der 5-fach verschachtelten Schleife ab. Sollte die Bedingung nicht erfüllt sein, wird keine Anweisung ausgeführt. Man kann sich alle äußeren *k* Durchläufe sparen, indem man die k-Schleife in den IF-Block verschiebt. Da die Bedingung sowieso nicht an *k* geknüpft ist, wird diese Schleife auch nur dann durchlaufen, wenn die Bedingung erfüllt ist. Im gegebenen Anwendungsfall wird der Block in ca. 15% von der Gesamtzahl der Iterationen ausgeführt.
Wenn man die Optimierung ohne weiteres vornimmt, wird man schnell feststellen, dass die Zeiten sich vervielfachen. Original wurde eine Zeit von etwa *36 sec* für diese Schleife gemessen. Verschiebt man das *k*, liegt man bei über *90 sec*. Man hat sich zwar die überflüssigen Iterationen eingespart, jedoch die Reihenfolge der Indizes durcheinander gebracht. Das *ycos_ysin* muss in der Reihenfolge *k , d , j , i* durchlaufen werden. Das verschieben von *k* ergibt eine Reihenfolge *d , j , i , k* . Obwohl es sich in dem Fall um den lokalen Speicher gehandelt hatte, erweisen sich die Speicherzugriffe als maßgebliche Komponente.
Eine Schleife im existierenden größeren Programm zu optimieren kann sehr schwer werden. Die Lösung in diesem Fall wäre das Umdeklarieren des *ycos_ysin* Arrays zu *ysin_ycos (k, i , j , d)* . Eine Umdeklarierung hat die Konsequenz der Umschreibung des ganzen Programms. Man sollte sich seiner theoretischen Überlegungen sicher sein, bevor man solch umfangreiche Veränderungen vornimmt. Man könnte es vorher testen, indem man die Zeilen und Spalten beim Abspeichern mittels des Compiler Flags

`-floop-interchange`

vertauscht. Somit wäre das *k* die erste Dimension und es ließe sich testen, ob die Optimierung tatsächlich fruchtbar ist. Der Compiler muss aber entsprechend konfiguriert sein **`(--with-ppl and --with-cloog)`** und diese Funktionalität unterstützen. Die Schleife kann auch separat mit den gemessenen Werten simuliert werden, hier wäre die Umschreibung nicht aufwendig.
Auf der angesprochenen Architektur hat sich die Zeit nach der Optimierung halbiert. Auf einer anderen Architektur brachte die Optimierung einen Gewinn von Faktor 6.

Wenn es um das Parallelisieren verschachtelter Schleifen geht, sollte die oben erklärte Speicheroptimierung immer vorgenommen werden, ansonsten wird das Programm langsamer. Ein Beispiel hierfür:

```
1   program wrong
2      ! NICHT NACHMACHEN
3      USE omp_lib
4      implicit none
5
6      INTEGER , PARAMETER :: out_end = 3600
7      INTEGER , PARAMETER :: in_end  = 1800
8      INTEGER :: faktor = 2
9      INTEGER :: out, in, it_begin, it_stop = 100
10     INTEGER , DIMENSION (1:out_end, 1:in_end) :: matrix = 10
11
12     DO it_begin = 1, it_stop                                  !Hilfsschleife
13
14     !$omp parallel num_threads(4)
15     !$omp do
16
17        DO out = 1, out_end                                    !Falsche Reihenfolge
18           DO in = 1, in_end
19              matrix(out, in) = matrix(out,in) * faktor
20           END DO
21        END DO
22     !$omp end do
23     !$omp end parallel
24
25     END DO
26
27  END program wrong
28
```

Es ergeben sich folgende Zeiten:

thr	Falsche Reihenfolge (sec)	Richtige Reihenfolge (sec)
1	14,5	0,7
2	14,6	0,4
4	15,7	0,3

Es lässt sich ein gewaltiger Unterschied zwischen den Umsetzungen feststellen. Während es in der falschen Reihenfolge zu kleineren Verlusten oder zumindest keinem Zeitgewinn bei Verdopplung der Threadzahl führt, gibt es in der richtigen Reihenfolge eine normale Skalierung (hier jedoch aufgrund der absoluten niedrigen Zeiten nicht gezeigt).

b. collapse

Bei mehrfach verschachtelten Schleifen stellt sich die Frage, ob es effizienter wäre, eine kürzere Schleife nach außen zu verschieben, oder eine längere. In unserem Anwendungsbeispiel liegen folgende Daten vor:
t = 8760
d = 1
k = 33
j = 220
i = 256

Da *d* nicht konstant in jeder Anwendung ist, kann es nicht eliminiert werden. Man könnte jedoch darüber nachdenken, *t* und *d* zu vertauschen, sofern es keine inhaltlichen Einwände gibt. Es folgt ein ausgiebiger Test zum Verhalten der Zeiten abhängig von der äußeren Schleifengröße.

Das Testprogramm lehnt sich an das gegebene Beispiel an, die Werte variieren.
Es wird die Funktionalität der OpenMP Option *collapse(x)* untersucht. Die Idee dabei ist, die angegebene Anzahl *x* der Schleifen zusammenzufassen und diese auf die Threads aufzuteilen, während im Normalfall nur die Zahl der äußeren Iterationen aufgeteilt wird. Bedeutend wird diese Option vor allem dann, wenn die Zahl der äußeren Iterationen etwa <= der Zahl der Threads ist.

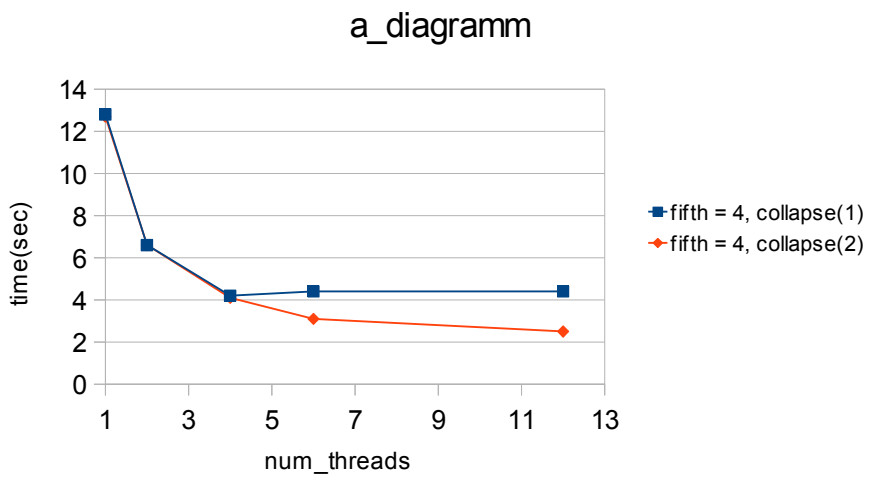
```
1   program test_collapse
2
3      USE omp_lib
4      implicit none
5
6      INTEGER , PARAMETER :: second = 21
7      INTEGER , PARAMETER :: third = 500
8      INTEGER , PARAMETER :: fourth = 2100
9      INTEGER , PARAMETER :: fifth = 4
10
11     INTEGER :: two, three, four, five
12     REAL , DIMENSION (1:second, 1:third, 1:fourth, 1:fifth) :: matrix = 10.0
13
14     !$omp parallel num_threads(1)
15     !$omp do collapse(1)
16              DO five = 1, fifth
17                 DO four = 1, fourth
18                    DO three= 1, third
19                       DO two = 1, second
20                             matrix(two, three, four, five) = matrix(two, three, four, five) &
21                                                              * sin(REAL(two*two))
22                       END DO
23                    END DO
24                 END DO
25              END DO
26
27     !$omp end do
28     !$omp end parallel
29     write(*,*) matrix(2,2,2,2)
30
31  END program test_collapse
32
33
```

Folgende Konfigurationen werden getestet:

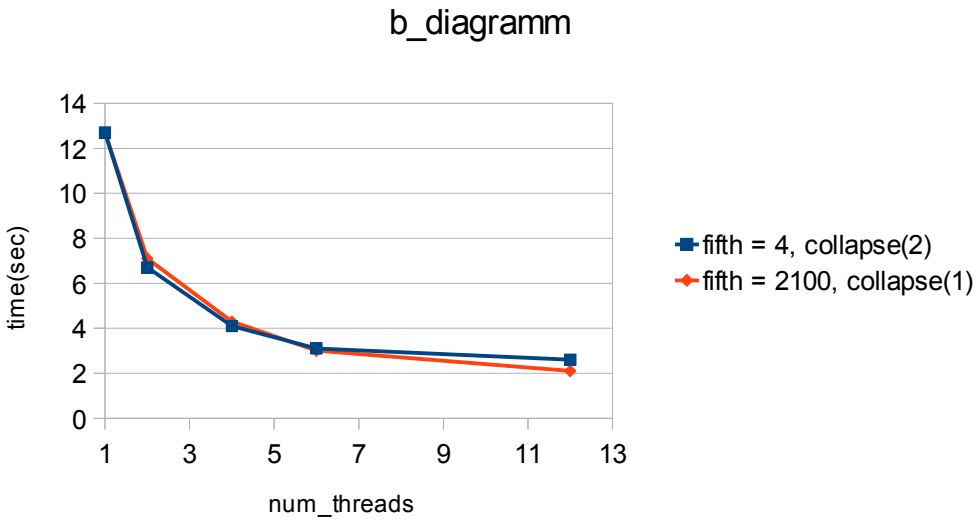
Parameter: collapse(x) num_threads	fifth = 4 fourth = 2100 third = 500 second = 21 (sec)	fifth = 4 fourth = 500 third = 2100 second = 21 (sec)	fifth = 500 fourth = 4 third = 2100 second = 21 (sec)	fifth = 2100 fourth = 4 third = 500 second = 21 (sec)	fifth = 2100 fourth = 500 third = 4 second = 21 (sec)
collapse(1)					
1	12,8	12,8	12,8	12,7	12,9
2	6,6	6,7	6,9	7,3	7,1
4	4,2	4,2	4,1	4,3	4,3
6	4,4	4,5	3,1	3,1	3,0
12	4,4	4,4	2,6	2,1	2,1
collapse(2)					
1	12,7	12,8	12,8	12,7	12,8
2	6,6	6,7	6,9	7,2	7,2
4	4,1	4,1	4,0	4,3	4,3
6	3,0	3,1	3,1	3,0	3,1
12	2,5	2,6	2,3	2,2	2,1
collapse(3)					
1	12,7	12,7	12,8	12,9	12,8
2	6,5	6,4	6,3	6,8	7,1
4	4,1	4,1	4,1	4,1	4,2
6	3,1	3,1	3,0	3,1	3,1
12	2,5	2,5	2,3	2,2	2,1

Erste Beobachtung: auf die sequentielle Ausführungszeit wirken sich weder die Reihenfolge der Werte, noch das *collapse* aus. Beim Ausführen des Programms mit nur einem Thread gibt es keine spürbaren Abweichungen von der komplett OpenMP freien Version.

Beide Fälle mit 4 Iterationen außen (fifth=4) und ohne *collapse* skalieren wie erwartet nur bis 4 Threads. Alle anderen Threads, falls vorhanden, sind nicht beschäftigt, die Leistung bleibt konstant auf dem Level der 4 Threads mit einer geringen Verschlechterung.
collapse bewirkt eine erwartete Skalierung auf 12 Threads. Dabei ist weder ein messbarer Unterschied zwischen den Konfigurationen noch zwischen *collapse(2)* und *collapse(3)* festzustellen.



Trotz der Leistungssteigerung dank dem *collapse*, sind die Zeiten höher, als mit der Konfiguration fifth = 2100 (siehe b_diagramm). Hier ist kein *collapse* nötig, denn es gibt eine ausreichende Anzahl an Iterationen, die auf die Threads aufgeteilt werden können, entsprechend auch die beste Zeit von *2,1 sec* bei 12 Threads. Merkbar ist jedoch der Overhead bei 2 Threads im vergleich zu fifth = 4. 4 Iterationen sind offensichtlich schneller auf 2 (bzw. 4) Threads aufzuteilen, als 2100. Die „blaue“ Konfuguration zeigt somit, dass das collapse offensichtlich nicht die Iterationszahlen vor dem Aufteile zusammenlegt und als eine Einheit betrachtet. In diesem Fall sollte es mit 2000 „äußeren“ Iterationen keinen merkbaren Unterschied zu der „roten“ Konfiguraion geben. Es wird aber offensichlich die ehemals äußere Iterationszahl (4 bei blau) in erster Linie aufgeteilt, was bei 2 weniger Prozessen sicher schneller geht, als 2100 Iterationen. Das zeigt, dass man zwar in der äußeren Schleife ausreichend viele Iterationen haben sollte, jedoch nicht zu viele, so dass sich kein Aufteilungsaufwand bemerkbar macht.



Der Unterschied sieht nicht groß aus, was daran liegt, dass absolute Zeiten nicht sehr hoch sind und die Anzahl der Threads nur bis 12 getestet wurde. Bei längeren Zeiten und höherer Zahl der Threads würde sich das mehr auswirken.

Fazit:
Ist man sich sicher, dass in der äußere Schleife eine ausreichende Zahl der Iterationen vorkommt (*num_it >> num_threads*), so sollte man das *collapse* auslassen. In den oben genannten Fällen sieht es so aus, als könnte das *collapse* nie schaden. Wenn es sich jedoch um fünfstellige Anzahl der Iterationen handelt, die aus mehreren Schleifen „kollabiert“ werden, so kann das Aufteilungsoverhead teils um den Faktor 2 die Berechnung abbremsen. Liegt jedoch die Statistik so, dass die äußere Schleife *num_it <= num_threads* vorweist, ist *collapse* nötig. Sofern man Einfluss darauf nehmen kann, sollte man eine ausgewogene

Anzahl der Iterationen nach außen legen.

Hinweis: wenn die Zeit unerwartet rasant ansteigt, sobald man Gebrauch von collapse macht, ist es ein Hinweis darauf, dass sie Speicherzugriffe nicht optimal sind.

9. Vorgehen

Es gibt kein Kochbuchrezept, wie man ein Programm parallelisiert, es kristallisiert sich aber ein Plan raus.

Vom konkreten Problem zur effizienten Lösung

1. Verteilter oder gemeinsamer Speicher ?
    1a. Konkrete Architektur (ggf. Sockets, Nodes etc)
    1b. Hardwareunterstützung
    1c. Umfang der Skalierbarkeit (100 oder 10000 cores ?)
2. Analyse der sequentiellen Umsetzung
    2.1 Profiler, Speicheranalyse, Algorithmen-analyse
3. Optimierung der sequentiellen Umsetzung
    3.1 Cache Optimierung
    3.2 Speicherzugriffe
    3.3 Umstrukturierung
4. Entwurf auf Papier (domain decomposition, critical section)
5. Kommunikation schätzen für verteilten Speicher / Kritische Bereiche für gemeinsamen Speicher bestimmen
6. Abschätzung der Vorteile → das Prinzip alleine sollte effizient sein, sequentielle Messungen
7. Implementieren → Standard beachten !
8. Testen / Profilen / Analysieren / Debuggen
9. Optimierung/Skalierbarkeit:
    9.1 Maß der Skalierbarkeit bestimmen, wann bricht Speed-up ein, Maßnahmen ergreifen
    9.2 Lastausgleich
    9.3 Compiler-Optimierung
    9.4 Portabilität ermitteln

Oft stellt sich die Vorgehensweise beim Optimieren und Parallelisieren währen der Arbeit heraus. In dieser Ausarbeitung wurden jedoch grundsätzliche Strategien untersucht, die man sich je nach Anwendung zu Nutze machen kann.

10. Zusammenfassung

Die Arbeit ist auf theoretischen Überlegungen aufgebaut, die praktisch überprüft wurden. Allem voran ist eine Einführung in die verschwendete Sprache (Fortran), die gebräuchlichen Architekturen und entsprechende Parallelisierungsarten gegeben worden. Es wurde grundsätzlich in 2 Arten unterteilt: dem internen Parallelismus, der synonym für alle in den Sprachen oder dem Aufbau verankerten Mechanismen steht, und dem externen, der die Struktur der jeweiligen Sprache nicht direkt beeinflusst, sondern externe Bibliotheken benötigt. Der Schwerpunkt lag dabei beim Parallelisieren auf gemeinsamem Speicher mit OpenMP. Hierzu findet man im Kapitel 6 eine ausführliche Beschreibung der Funktionen und Optionen. Man neigt oft dazu, den Aufwand mit OpenMP zu unterschätzen. Diese Arbeit zeigt sehr viele nicht intuitive und nicht triviale Facetten, wie z.B. ausführliche Analyse des Verhaltens von *collapse*. Sehr wichtig ist die selten zu findende Zusammenstellung der nicht standardisierten Einstellungen in OpenMP. Ein hohes Risiko stellen dabei die herstellerdefinierten Parameter dar - z.B. die Anzahl der zu erzeugenden Threads, wenn explizit keine angegeben wurden.
Alle theoretisch behandelten Aspekte wurden praktisch umgesetzt und analysiert. Im Kapitel 7 (Analyse) ist die verwendete Umgebung dokumentiert. Die konstruierten Testfälle reichen vom bloßen Initialisieren auf die gewöhnliche Art (mittels Schleife) und die Fortran-spezifische Art (direkte Zuweisung), bis hin zu komplexen Kombinationen mehrerer Konstrukte. Im Vordergrund stand der Zeitbedarf der jeweiligen Berechnung, aber immer mit Berücksichtigung der Speicherzugriffe. Die Testfälle waren meist künstlich generiert, motiviert von realen numerischen Anwendungen und bezogen sich hauptsächlich auf Schleifenkonstrukte. Der Vorteil dieser Strategie liegt im Abstrahieren der Schleifenkonstrukte von der gegebenen Anwendung, die mit vielen Faktoren die Laufzeit beeinflussen kann. Es ging um die technische Umsetzung der theoretischen Ideen ohne anwendungsspezifische Seiteneffekte. Es hat sich klar herausgestellt, dass es nicht den einen Weg, wie man parallelisiert, geben kann. Die optimale Strategie hängt, auch wenn von der Anwendung losgelöst, dennoch primär von Eingangsparametern und der gegebenen Architektur ab. Alleine die Speicherkapazität und die Compilerversionen lassen kein „Rezept“ zu, wie sich bei der Arrayinitialisierung oder beim *forall*-Test gezeigt hat.
Beim Umsetzen der Testfälle sind viele unerwartete Fehler aufgetreten, die sehr dem Verständnis der Funktionsweise von OpenMP beigetragen haben. Auch wenn es kein universelles Vorgehen geben kann, so ist doch viel klarer geworden, worauf bei der Untersuchung primär geachtet werden muss – Speicherzugriffe, sequentielle Abschnitte, Flexibilität der Eingangparameter, etc.
Die ermittelten Ergebnisse können jederzeit als Grundlage für Schleifenparallelisierungen vieler Anwendungen genutzt werden.

Einige sehr interessante Fragen sind noch offen geblieben, wie z.B. die Zusammenarbeit von OpenMP und MPI. Welche Maßnahmen muss man hier treffen? Wie beeinflusst MPI das analysierte Verhalten von OpenMP und umgekehrt ? Es gibt sicher noch mehr Theorien, die die Praxis noch belegen oder widerlegen muss.

11. Anhang

Canonical shape (C/C++ Standard, gilt analog auch für Fortran)

The **for** directive places restrictions on the structure of the corresponding **for** loop. Specifically, the corresponding **for** loop must have *canonical shape*:

for (*init-expr*; *var* *logical-op* *b*; *incr-expr*)	
init-expr	One of the following: *var* = *lb* *integer-type var* = *lb*
incr-expr	One of the following: ++*var* *var*++ --*var* *var*-- *var* += *incr* *var* -= *incr* *var* = *var* + *incr* *var* = *incr* + *var* *var* = *var* - *incr*
var	A signed integer variable. If this variable would otherwise be shared, it is implicitly made private for the duration of the **for**. This variable must not be modified within the body of the **for** statement. Unless the variable is specified **lastprivate**, its value after the loop is indeterminate.
logical-op	One of the following: < <= > >=
lb, *b*, and *incr*	Loop invariant integer expressions. There is no synchronization during the evaluation of these expressions. Thus, any evaluated side effects produce indeterminate results.

Note that the canonical form allows the number of loop iterations to be computed on entry to the loop. This computation is performed with values in the type of *var*, after integral promotions. In particular, if value of *b* - *lb* + *incr* cannot be represented in that type, the result is indeterminate. Further, if *logical-op* is < or <= then *incr-expr* must cause *var* to increase on each iteration of the loop. If *logical-op* is > or >= then *incr-expr* must cause *var* to decrease on each iteration of the loop.

Weitere Funktionen, Optionen und Direktiven von OpenMP werden im Tutorial erklärt:
https://computing.llnl.gov/tutorials/openMP/

12. Quellen

Literatur

1. *Stephen J. Chapman*, „Fortran 95/2003 for Scientists and Engineers“, 2008
2. *Heiko Bauke, Stephan Mertens*, „Cluster Computing“, 2005
3. *Rolf Rabenseifner*, „Parallel Programming Workshop“, 2012

Web

1. http://www.intuit.ru/department/se/openmp/2/4.html
2. https://computing.llnl.gov/tutorials/openMP/əhttp://
3. www.math.uni-leipzig.de/~hellmund/Vorlesung/scr4.pdf
4. http://publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp?topic=%2Fcom.ibm.xlf101l.doc%2Fxlflr%2Fforcons.htm
5. http://www.openmp.org/mp-documents/cspec20.pdf
6. http://msdn.microsoft.com/de-de/library/x721b5yk%28v=vs.80%29.aspx
7. http://www2.fz-juelich.de/jsc/files/docs/vortraege/fortran_all.pdf
8. http://www.intuit.ru/department/se/openmp/2/4.html