Herausforderungen von Tausendundeinem Kern

Alexander Münch
Friedrich-Alexander-Universität Erlangen-Nürnberg
Alexander.Muench@informatik.stud.uni-erlangen.de

ZUSAMMENFASSUNG

Glaubt man den Voraussagen einiger Experten, so werden innerhalb von zehn Jahren bis zu 1000 Prozessorkerne in einer CPU untergebracht sein. Dieser Artikel wird die Herausforderungen bei der Hard- und Softwareentwicklung solcher Systeme erläutern, sowie die derzeit diskutierten Lösungsansätze vorstellen.

1. EINLEITUNG

Vor einigen Jahren hatten unsere PCs noch Prozessoren mit Taktraten im Megahertz-Bereich. Heute ist in jedem Multimedia-PC, wie man ihn im Supermarkt zu kaufen bekommt, mindestens ein Dual-Core-Prozessor mit 2 GHz eingebaut.

Statt auf weitere Taktsteigerung wird zukünftig auf mehrere Kerne gesetzt und so sind heute auch schon Quad-Core-Prozessoren üblich, d.h. eine CPU mit vier Kernen. Und die Entwicklung schreitet rasend voran. In naher Zukunft können wir uns darauf einstellen, dass auch im Privatbereich Prozessoren mit 8, 16, 32, 64 Kernen an der Tagesordnung sein werden.

Anders als bei Geschwindigkeitssteigerung durch Erhöhung der Taktfrequenz treten bei Geschwindigkeitssteigerung durch Parallelität besondere Probleme auf, die sowohl in der Hardware- als auch in der Softwareentwicklung gelöst werden müssen.

Diese Arbeit soll einen Einblick geben, welche Probleme auftreten und Lösungsansätze aufzeigen.

1.1 Warum Parallelität?

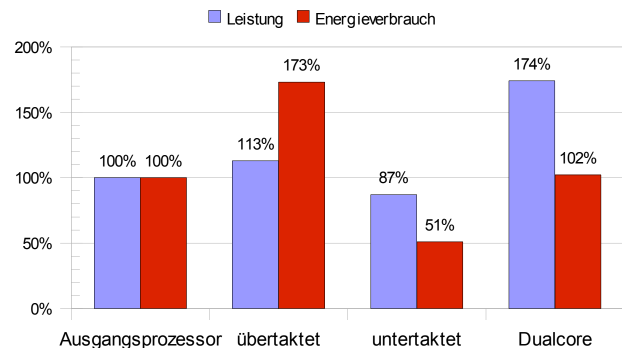
Zuerst möchte ich die Frage beantworten, wieso man überhaupt in der Prozessorenentwicklung den Weg geht, mehrere Kerne zu verbauen, anstatt wie üblich den Takt weiter zu erhöhen. Hierzu erörtere ich das Beispiel von James Reinders, Direktor der Developer Products Division von Intel, welches er in einem Vortrag zum Thema Parallelität [1] gegeben hat:

Angenommen, wir haben einen Prozessor, der eine Leistungseinheit stellt und dafür eine Energieeinheit Strom verbraucht. Diese fiktiven Größen sollen einen Vergleich von Kosten und Nutzen ermöglichen.

Nun übertakten wir den Prozessor, sodass er 1,13 Leistungseinheiten produziert, aber nun werden 1,73 Energieeinheiten benötigt. Ein zu hoher Preis für die gewonnene Performance. Der Stromverbrauch steigt exponentiell gegenüber der Leistung.

In einem zweiten Versuch untertakten wir den Prozessor, sodass er nun mehr nur noch 0,87 Leistungseinheiten produziert, aber auch nur noch 0,51 Energieeinheiten verbraucht. Wir haben jetzt zwar Leistung verloren, aber im Verhältnis mehr Energie eingespart.

Kombiniert man nun zwei dieser untertakteten Prozessoren, erhält man einen Dual-Core-Prozessor, der gegenüber dem Ausgangsprozessor nur 2% mehr Energie verbraucht, aber 74% mehr Leistung liefert. Die nachfolgende Grafik visualisiert diese Überlegungen.



Natürlich wird diese theoretische Leistung nicht immer erreicht, da man beide Kerne auslasten muss, was nicht immer möglich ist. Zudem kommt zusätzlicher Overhead durch Synchronisationsmaßnahmen hinzu.

1.2 Probleme durch die Parallelität

Der Zugriff auf gemeinsame Daten mit mehreren Kernen ist nicht trivial. Im folgenden möchte ich das Philosophenproblem vorstellen, was *das* Beispiel ist, um die Problematik beim Zugriff auf gemeinsame Betriebsmittel zu veranschaulichen:

1.2.1 Das Philosophenproblem

Fünf Philosophen sitzen um einen runden Tisch. Jeder sitzt vor einem Teller Spaghetti und zwischen den Tellern befindet sich jeweils eine Gabel. Ein Philosoph kann entweder über philosophische Probleme nachdenken oder essen.

Will ein Philosoph essen, so nimmt er zuerst die Gabel zu seiner Linken auf, dann die Gabel zu seiner Rechten und isst anschließend. Wenn er satt ist, legt er beide Gabeln wieder zurück und beginnt wieder nachzudenken. Dieses System funktioniert ganz gut, wenn immer nur ein oder zwei Philosophen essen wollen.

Was aber, wenn alle fünf Philosophen gleichzeitig essen wollen? Jeder nimmt die Gabel zu seiner Linken auf, kann aber nicht essen, weil sein rechter Nachbar ihm die andere Gabel weggenommen hat. Natürlich legt niemand seine Gabel wieder zurück, weil sie noch auf die rechte Gabel warten.

Alle fünf Philosophen verhungern.

1.2.2 *Bereit-Liste in einem Betriebssystem*

Konkret auf unser Anwendungsgebiet abgebildet entspricht ein Philosoph einem Prozessorkern und eine Gabel stellt ein gemeinsam genutztes Betriebsmittel dar.

In einem Betriebssystem könnte dieses Betriebsmittel die Bereit-Liste sein, die der Scheduler verwaltet. Dort werden alle Fäden gelistet, die lauffähig sind und darauf warten, dass der Scheduler ihnen die CPU, d.h. einen der Prozessorkerne, zuteilt und sie ausgeführt werden. Es werden Fäden von der Liste entfernt, wenn sie sich blockieren oder beendet werden und neue Fäden hinzufügt, wenn sie aufgeweckt oder erstmalig erzeugt worden sind. Jeder Kern greift potentiell zeitgleich mit einem anderen Kern auf diese Liste zu und es muss durch geeignete Synchronisation sichergestellt werden, dass die Datenstruktur nicht korrumpieren wird.

Die Vorgänge, das Einhängen und Austragen von Elementen in diese Liste, werden sehr häufig durchgeführt – Short-Term-Scheduling siedelt sich im Bereich von Mikrosekunden bis Millisekunden an – und das Synchronisieren ebendieser führt zu einem erheblichen Overhead.

2. THEORETISCHE ÜBERLEGUNGEN

Im folgenden möchte ich kurz ein wenig in die Theorie gehen und grob die Frage anreißen, ob Parallelität theoretisch überhaupt zum Erfolg führen kann.

2.1 Amdahlsches Gesetz

Gene Amdahl hat 1967 eine Aussage über die Beschleunigung von Prozessen bei der Verwendung mehrerer Recheneinheiten beschrieben, die heute als *Amdahlsches Gesetz* bekannt ist:

$$S=\frac{1}{(1-P)+\frac{P}{N}}$$

Die mögliche Beschleunigung S eines Programms, das aus einem Anteil P an parallelem und einem Anteil (1-P) an seriellem Code besteht, kann bei N Prozessoren durch obige Formel ermittelt werden.
Einfacher ausgedrückt besagt die Formel, dass die Beschleunigung, die man mit mehreren Prozessoren erwirken kann, durch den Anteil an seriellem Code begrenzt wird.

Verwendet man also einen Prozessor mit 16 Kernen und führt damit ein Programm aus, welcher nur zu 35% aus Code besteht, der parallel laufen kann, so ergibt sich keine Geschwindigkeitssteigerung auf 1600%, sondern es sind lediglich 148,8%.

Abbildung 1 zeigt, wie sich die mögliche Beschleunigung gegenüber der Anzahl an Prozessorkernen bei Programmen mit unterschiedlichem Anteil an parallelem Code verhält.

An Hand der Grafik wird deutlich, dass man auch bei vielen Kernen aus Programmen mit übermäßig seriellem Code keine signifikante Performance-Steigerung erreichen kann. Soll das heißen, die Parallelität ist eine Sackgasse?

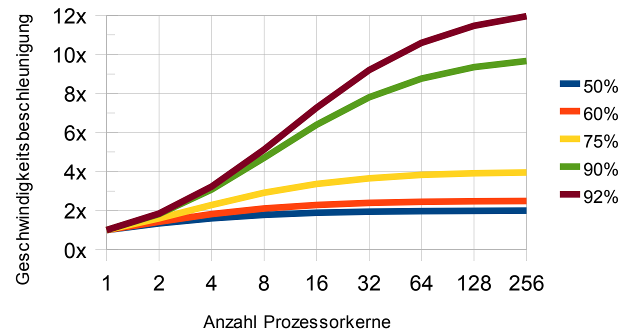


Abbildung 1: Amdahlsches Gesetz

2.2 Gustafsons Gesetz

Nein. John Gustafson zeigte 1988 den Ausweg, indem er die Sache etwas anders betrachtete. Während Ambahl davon ausgeht, dass die Datenmenge, die bearbeitet werden soll konstant ist und man die Ausführungszeit minimieren möchte, so geht Gustafson von konstantem Zeitaufwand aus und maximiert dabei die Datenmenge.

$$S=N-(1-P)\cdot(N-1)$$

Wieder gibt S die mögliche Beschleunigung eines Programms mit Anteil P an parallelisierbarem Code mit N Prozessoren an.

In der heutigen Zeit müssen immer mehr und mehr Daten analysiert und durchgerechnet werden, sodass Gustafsons Überlegung sicherlich besser passt.

Setzen wir als Beispiel einmal Zahlenwerte in die Formel ein, so zeigt sich schnell, dass sich mit Gustafsons Formel erhebliche Performance-Verbesserungen ergeben:

Für ein System mit 128 Prozessoren prognostiziert Amdahl, dass ein Programm mit 60% parallel ausführbarem Code mit einer Geschwindigkeit von 247% läuft, während Gustafson uns 7720% verspricht.

Seitens der Theorie steht der Verwendung von Mehrkernprozessoren also nichts im Wege.

3. HARDWAREENTWICKLUNG

3.1 Symmetrische vs. asymmetrische Mehrkernprozessoren

Wie in Abschnitt 1.1 am Beispiel aufgezeigt, erhält man einen Mehrkernprozessor, wenn man mehrere Kerne niedrigere Taktfrequenz auf einen Mikrochip setzt. So kann mehr Leistung bieten, als ein Einkernprozessor, der die höhere Taktrate besitzt.

Sind alle Kerne auf einem Chip vom selben Typ, so spricht man von einem *symmetrischen Mehrkernprozessor*. Verbaut man hingegen verschiedene Kerne, so nennt man das einen *asymmetrischen Mehrkernprozessor*.

Eine wichtige Größe beim Bau von Prozessoren ist der Platzbedarf. Man kann nicht beliebig viele Kerne auf einem Chip unterbringen, da man auf die benötigte Fläche und die Wärmeentwicklung achten muss. Ein Prozessor, der nur mit Flüssig-Stickstoff-Kühlung arbeiten kann, wird sich im Privathaushalt nicht durchsetzen.

Je nach Anwendungszweck kann es also sinnvoller sein, statt vier relativ großen, leistungsfähigen, lieber nur zwei Kerne einzusetzen, aber dafür zusätzlich noch eine größere Anzahl kleinere Kerne zu verbauen. Zum Beispiel komplexe Multimedia-Berechnungen können von einem großen Prozessor mit einem komplexen Instruktionssatz ausgeführt werden, während für die meisten Anwendungen grundliegende Arithmetik der kleineren Kerne vollkommen ausreicht.

Aktuell sind auf dem Markt hauptsächlich symmetrische Multi-Core-Prozessoren zu finden. Ein bekannter Vertreter bei den asymmetrischen Vertretern ist der Cell-Prozessor [2], der aus 8 kleineren Recheneinheiten (SPE = Synergistic Processing Element) und einem PPE (= PowerPC Processing Element) besteht und in der PlayStation3 zur Anwendung kommt.

3.2 Transaktionaler Speicher

Ein Problem, welches bei der Verwendung von Parallelität auftritt, sind die konkurrierenden Speicherzugriffe. Ein Prozessorhersteller muss dafür Sorge tragen, dass eine TAS- (test-and-set) oder CAS- (compare-and-swap) Operation wirklich atomar abläuft und nicht von einem anderen Kern gestört wird.

Dasselbe Problem ist von Datenbanken bekannt, wo mehrere Operationen auf denselben Daten ausgeführt werden müssen. Hier hat man sich mit Transaktionen beholfen. Semantisch bedeutet das, dass eine Folge von Änderungsoperationen grundsätzlich atomar ausgeführt wird, sodass gleichzeitig schreibende Prozesse nicht stören. Kein lesender Prozess darf Daten lesen, wenn die zugehörige Schreiboperation noch nicht beendet wurde.

Diese Idee hat man nun aufgegriffen und versucht, sie auf den Hauptspeicher zu übertragen. So sollen die Änderungen eines Kerns erst von den anderen Kernen sichtbar sein, wenn die Transaktion beendet ist. Der Speicher bietet Schnittstellen an, um "begin transaction" und "commit transaction" zu signalisieren. Gibt es beim Ausführen der Transaktion einen Konflikt, so werden alle Änderungen, die zu diesem Zeitpunkt schon gemacht wurden, rückgängig gemacht ("rollback") und man kann es erneut versuchen.

Vorteil dieses Verfahrens ist es, dass nicht mehr ganze Aufführungsstränge durch gegenseitigen Ausschluss blockiert werden müssen, sondern die Hardware entscheidet, welche Speicherzugriffe parallel ausgeführt werden können und welche nicht.

4. SOFTWAREENTWICKLUNG

In meinen Augen wichtiger als die Hardwareentwicklung ist die Softwareentwicklung. Parallele Hardware ist schon seit einigen Jahren für jedermann verfügbar, nur es fehlt noch die entsprechende Software, die auch ihre Vorteile aus der Hardware zieht.

4.1 Parallelisierbare Probleme

Eine wichtige Voraussetzung dafür, mit paralleler Hardware einen Performance-Gewinn zu erhalten, ist, dass das Problem, welches in ein Programm umgesetzt werden soll, überhaupt parallelisierbar ist. Nicht jedes Stück Code kann parallel ausgeführt werden. In Abschnitt 2 habe ich aufgezeigt, dass der Anteil an parallelem Code sehr wichtig für die Überlegungen ist, wie viel Performance-Steigerung man erhalten kann.

Interaktive Prozesse, die hauptsächlich auf Benutzereingaben warten, sind nicht besonders parallelisierbar. Parallele Algorithmen, wie zum Beispiel Filteroperationen auf Bild- und Videodaten, hingegen können hochgradig parallelisierbar sein und bei entsprechender Programmierung auch signifikant schneller ausgeführt werden. Ansatzpunkte sind vor allem Schleifen, die iterativ Datenstrukturen abarbeiten.

Im folgenden stelle ich mehrere Verfahren vor, um eben diese Parallelität in der Software umzusetzen: MPI basiert auf Nachrichtenaustausch, während OpenMP und TBB auf Basis von Schleifen im Programmcode parallele Abschnitte erzeugen, die von mehreren Kernen parallel ausgeführt werden können.

4.2 Parallelität über das Betriebssystem

Moderne Betriebssysteme nutzen völlig selbstständig die vorhandene Hardware aus und machen sich auch die Vorteile mehrerer Kerne zu Nutze. Auf einem Dual-Core-System würde ein Windows-Betriebssystem z.B. auch bis zu zwei Fäden völlig parallel ausführen können. Warum also spezielle Software entwickeln, wenn doch das Betriebssystem schon alles erledigt?

Der Grund ist einfach: Nicht immer sind genügend Prozesse laufbereit, um alle Prozessorkerne auszulasten. Vor allem dann nicht, wenn man bedenkt, dass die Anzahl der Kerne in Zukunft rasant zunehmen wird. Die meiste Zeit hat der Prozessor nichts zu tun, weil die Anwendungen alle auf Benutzereingaben warten oder mit I/O beschäftigt sind. Doch dann, wenn es darauf ankommt, eine Anwendung einen CPU-Burst hat, hilft es auch nicht, mehrere Kerne zu haben, denn ein Prozess kann immer nur von einer CPU ausgeführt werden. D.h. ein Kern läuft auf 100% Auslastung und die restlichen Kerne führen nur den Idle-Faden aus.

4.3 Message Passing Interface (MPI)

Hat man erstmal mehrere Prozesse, die parallel ablaufen, muss man sich um die Synchronisation kümmern. An bestimmten Punkten müssen die Ergebnisse von bestimmten Prozessen vorliegen, um sie weiter zu verarbeiten.

MPI [3] stellt einen Standard dar, um den Nachrichtenaustausch zwischen Prozessen zu ermöglichen. Hierbei wird kein konkretes Protokoll festgelegt, sondern lediglich eine Programmierschnittstelle und ihre Semantik. Bei MPI gibt es einen Hauptprozess, der eine besondere Stellung einnimmt und mit speziellen Funktionen andere Prozesse mit Daten versorgen und die Ergebnisse wieder einsammeln kann. Implementierungen für MPI gibt es z.B. für C++ (MPICH2 [4]) und Python (pyMPI [5]). Einen Ausschnitt über die definierten Funktionen zeigt die nachfolgende Tabelle:

Tabelle 1: MPI-Funktionen (Ausschnitt)

Funktion	Semantik
MPI_Send(), MPI_Recv()	Senden und Empfangen einer Nachricht. Diese Funktionen sind asynchron und blockierend.
MPI_Isend(), MPI_Irecv()	Senden und Empfangen einer Nachricht. Diese Funktionen sind nicht-blockierend.

MPI_Test()	Status einer nicht-blockierenden Operation abfragen.
MPI_Bcast()	Sendet eine Nachricht an alle Prozesse.
MPI_Gather()	Der Hauptprozess "sammelt" die Ergebnisse aller anderen Prozesse ein.
MPI_Allgather()	Jeder Prozess sendet seine Ergebnisse an alle Fäden. Am Ende haben alle Prozesse dieselben Daten.
MPI_Reduce()	Sonderform von MPI_Gather(). Auf die Ergebnisse wird zusätzlich noch eine Reduktionsfunktion angewandt.
MPI_Scatter()	Der Hauptprozess sendet an alle anderen Prozesse eine Nachricht.

4.3.1 Fazit zu MPI

Der MPI-Standard ist darauf ausgerichtet, Prozesse zu synchronisieren, die keinen Zugriff auf gemeinsamen Speicher haben. Üblicherweise findet man MPI meist auf Supercomputern eingesetzt, d.h. viele Computer, die auch räumlich voneinander getrennt sein können. Dadurch, dass jeder Prozess Zugriff auf exklusiven Speicher hat, erhält man eine Performance-Steigerung.

Es gibt aber keine Einschränkung, sodass MPI auch auf einem Rechensystem mit gemeinsamen Speicher eingesetzt werden kann. Allerdings rentiert sich das wegen dem zusätzlichen Overhead meistens nicht.

4.4 Open Multi-Processing (OpenMP)

Eine weitere Möglichkeit, parallele Programme zu schreiben, ist der OpenMP-Standard. Wie MPI ist auch OpenMP eine Programmierschnittstelle, allerdings fungiert OpenMP nicht auf Prozess-, sondern auf Faden- oder sogar Schleifenebene. OpenMP ist für C/ C++- und Fortran-Compiler festgelegt.

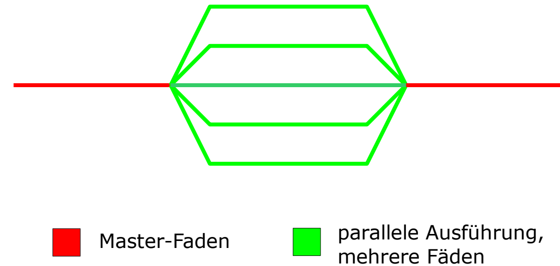
OpenMP definiert u.a. Compilerdirektiven, die bei der Übersetzung des Programms vom Compiler ausgewertet werden, sodass dieser das Programm für eine parallele Ausführung vorbereitet. Dies bedeutet, dass man als Programmierer den Vorteil hat, sich nicht explizit um die Parallelität kümmern zu müssen. Man legt lediglich fest, welche Teile im Code potentiell mehrfädig ausgeführt werden können, welche Variablen von mehreren Fäden gleichzeitig benutzt werden können und welche nicht. Den Rest erledigt der Compiler.

Darüber hinaus definiert OpenMP noch Bibliotheksfunktionen und Umgebungsvariablen, auf die ich hier nicht weiter eingehen möchte.

4.4.1 Fäden, das Team und parallele Abschnitte

Zu Beginn der Ausführung gibt es nur einen Faden, den initialen Faden [6]. Beim Eintritt in eine parallele Region erstellt dieser Faden mittels entsprechender fork()-Aufrufe neue Fäden. Er selbst bleibt der Master-Faden dieses *Teams*. Innerhalb eines solchen Abschnitts sieht die Spezifikation vor, dass auch einzelne Fäden dieses Teams sich beliebig weiter aufspalten können ("nested parallelism"). Am Ende einer parallelen Region warten alle Fäden des Teams aufeinander. Danach läuft nur der Master-Faden im seriellen Abschnitt weiter fort.

Die nachfolgende Abbildung verdeutlicht den Ablauf:



Zur Laufzeit wird bestimmt, wie viele Prozessoren tatsächlich vorhanden sind und die parallelen Abschnitte auf die entsprechende Anzahl von Fäden verteilt. Es gibt auch eine entsprechende Bibliotheksfunktion, um die Anzahl der Fäden explizit zu setzen. Die Einteilung der Fäden auf die Prozessoren übernimmt dabei das Betriebssystem.

Programme, die mit Hilfe von OpenMP geschrieben worden sind, können (bis auf Ausnahmen) auch mit Compilern übersetzt werden, die den Standard nicht beherrschen. Hier werden die OpenMP-Direktiven einfach ignoriert und das Programm läuft seriell ab.

Im Folgenden zeige ich die Verwendung von OpenMP mit C und dem GCC-Compiler. Im GCC ist der OpenMP-Standard seit Version 4.2 integriert [7] und kann einfach mittels des Flags `-fopenmp` aktiviert werden. Im Programmcode selber muss zuerst die Headerdatei "omp.h" inkludiert werden. Die eigentlichen Anweisungen für die Parallelausführung werden mittels der Compilerdirektive:

```
#pragma omp
```

eingeleitet. Danach folgen ein Schlüsselwort und ggf. weitere Parameter. Das wichtigste Konstrukt in OpenMP ist das parallel-Konstrukt.

```
#pragma omp parallel
{
  /* paralleler Code-Bereich */
}
```

Damit kennzeichnet man für den Compiler Codebereiche, die parallel ausgeführt werden können.

4.4.2 Parallele for-Schleifen

Speziell für for-Schleifen, gibt es das for-Konstrukt, dem man weitere Parameter anhängen kann. Statt

```
#pragma omp parallel
{
  #pragma omp for
  for(/* ... */)
    /* ... */
}
```

kann man auch kurz

```
#pragma omp parallel for
for(/* ... */)
  /* ... */
```

schreiben.

Das folgende Code-Beispiel zeigt die Verwendung von OpenMP zur parallelen Ausführung einer 3x3-Matrizenmultiplikation. Bei solch einer Multiplikation kann jeder Wert in der Ergebnismatrix separat, d.h. parallel, durch Bildung des Skalarprodukts ausgerechnet werden.

```
#include <omp.h>

int main(int argc, char** argv)
{
  double a[3][3], b[3][3], c[3][3];
  int i, j, k;
  double sum;

  #pragma omp parallel for private(sum)
  for(i = 0; i < 3; i++)
    for(j = 0; j < 3; j++) {
      sum = 0;
      for(k = 0; k < 3; k++)
        sum += a[i][k] * b[k][j];
      c[i][j] = sum;
    }
}
```

`a` und `b` beinhalten die Eingabewerte. Am Ende des Programms enthält `c` die Ergebnismatrix `a*b`.

Die Matrix wird hierbei in drei Fäden berechnet. Jeder Faden berechnet eine Zeile der Ergebnismatrix. Da jeder Faden Zugriff auf alle Daten hat, muss man in einer Parallel-Region explizit angeben, wenn eine Variable private sein soll, d.h. nicht von anderen Fäden gelesen und verändert werden soll. Es wird daraufhin für jeden Faden eine Kopie der Variablen angelegt. Die Iterator-Variablen werden automatisch als private deklariert.

In unserem Beispiel muss die Variable `sum` geschützt werden, da drei Fäden gleichzeitig ein Skalarprodukt berechnen und natürlich niemand das Ergebnis des anderen beeinflussen darf. Am Ende der äußeren Schleife warten alle Fäden aufeinander. Danach kann mit den Werten in `c` im Master-Faden weitergerechnet werden.

4.4.2.1 for-Schleife mit Reduktion

Häufig werden in einer for-Schleife Ergebnisse einer jeden Iteration zusammengerechnet. Bei paralleler Ausführung kann dies gefährlich werden, da mehrere Fäden auf die gleiche Speicherzelle zugreifen, um ihre Ergebnisse z.B. aufzuaddieren. Im folgenden Beispiel soll die Summe der Quadratzahlen von 1 bis 100 berechnet werden.

```
long sum = 0;
for(int i = 1; i <= 100; i++)
  sum += i * i;
```

OpenMP bietet hierfür die Reduktion an. Es wird an die Compilerdirektive der Parameter `reduction` angehängt. In Klammern schreibt man, mit einem Doppelpunkt getrennt, zuerst die Operation, die auf der gemeinsamen Variable ausgeführt werden soll und danach die Variable selber.

```
long sum = 0;
#pragma omp parallel for reduction(+:sum)
for(int i = 1; i <= 100; i++)
  sum += i * i;
```

Mit diesem Wissen legt der Compiler für jeden Faden eine private Variable an, die in "ihren" Bereich des Intervalls von 1 bis 100 die Quadratsumme hält. An der impliziten Barriere – hierzu mehr in Abschnitt 4.4.4 –, wenn alle Fäden ihre Teilsumme berechnet haben, werden die Summen zusammenaddiert.

4.4.3 Sections

Hat man mehrere Arbeitsschritte, die parallel ausgeführt werden können, aber keine for-Schleife darstellen, kann man Sections benutzen:

```
#pragma omp Sections
{
  #pragma omp Section
  do_this();

  #pragma omp Section
  and_do_that_in_parallel();
}
```

Es können beliebig viele Sections definiert werden. Für jede Section wird dann ein Faden aus dem Team zugeteilt, der parallel Code ausführt.

Analog zum for-Konstrukt gibt es hierfür auch eine Kurzform.

4.4.4 Synchronisation durch Barriere, Single- und Master-Konstrukt

In parallelen Abschnitten gelten die üblichen Gesetzmäßigkeiten von Nebenläufigkeit: Jede Zeile kann potentiell mit jeder anderen Zeile parallel ausgeführt werden. An bestimmten Stellen sind allerdings Synchronisationspunkte notwendig, weil z.B. ein Faden Daten eines anderen Fäden benötigt und die Ausführung erst weitergehen kann, wenn diese Daten vorliegen.

OpenMP definiert hierfür das barrier-Konstrukt

```
#pragma omp barrier
```

An dieser Zeile warten alle Fäden eines Teams solange, bis jeder einzelne Faden seine Ausführung bis dorthin gebracht hat.

Die bereits aufgeführten for- und sections-Konstrukte haben eine implizite Barriere am Ende, so dass man sicher sein kann, dass kein Faden schneller aus einer for-Schleife oder einem sections-Abschnitt "herauslaufen" kann als ein anderer.

Ist es erforderlich, dass ein Codeabschnitt genau einfach ausgeführt wird, so verwendet man das single-Konstrukt

```
#pragma omp single
do_something_only_once();
```

Ähnlich dem sections-Konstrukt wird nur ein Faden des Teams den Code im single-Konstrukt ausführen. Alle anderen Fäden des Teams warten an der impliziten Barriere, bis der obige Code durchgelaufen ist. Es sei angemerkt, dass beim single-Konstrukt nicht notwendigerweise der Master-Faden den Abschnitt ausführt.

Um nur den Master-Faden eine Arbeit erledigen zu lassen, stellt OpenMP das master-Konstrukt, das analog verwendet wird:

```
#pragma omp Master
only_Master_runs_this_call();
```

Anders als bei den bisher vorgestellten Konstrukten gibt es hier keine implizite Barriere. Alle anderen Fäden laufen also einfach durch und ignorieren den Codeblock, den nur der Master-Faden ausführt.

4.4.5 Fazit zu OpenMP

OpenMP stellt noch viele weitere Konstrukte, Umgebungsvariablen und Bibliotheksfunktionen bereit. Es wäre allerdings zu umfangreich, um an dieser Stelle weiter ins Detail zu gehen. Im folgenden ziehe ich ein kurzes Fazit zu OpenMP.

Mit Open Multi-Processing kann man C- und Fortran-Programme relativ einfach parallelisieren. Der Code bleibt übersichtlich und kann ganz bequem zu Debug-Zwecken auch auf seriellen Ablauf "umgeschaltet" werden, indem man mittels API-Aufruf die Anzahl der Fäden limitiert oder den Compiler anweist, die #pragma omp-Direktiven einfach zu ignorieren.

Nachteil ist allerdings, dass man einen Compiler benötigt, der diesen Standard unterstützt. Ein weiterer Kritikpunkt an OpenMP ist, dass man sich als Programmierer sehr genau mit dem Parallelismus auseinander setzen muss und in den entsprechenden Direktiven den Zugriff auf gemeinsame Daten einschränken muss. Hier ist eine große Fehlerquelle, da Deadlocks durch Barrieren entstehen können, wenn ein Faden einfach nicht fertig wird und somit das Programm steht.

Das Problem mit den gemeinsamen Daten könnte durch eine zusätzliche Abstraktionsebene gelöst werden.

4.5 Threading Building Blocks (TBB)

Threading Building Blocks, im Folgenden kurz TBB, ist ein von Intel entwickeltes freies Framework, um von parallelen Abschnitten in Programmen zu abstrahieren [14]. TBB definiert eine Reihe von Klassen, die dann zum Beispiel zur parallelen Ausführung von for-Schleifen verwendet werden können.

TBB ist sehr an die STL [15] angelehnt: Es gibt Templates, um abstrakte Container anzulegen, Algorithmen und Funktionsobjekte – auch Funktor genannt.

Ich möchte zuerst kurz einige der Basisklassen von TBB vorstellen.

Damit überhaupt mit TBB gearbeitet werden kann, muss ein Objekt der Klasse `Task_Scheduler_init` instanziert werden. Der Task-Scheduler kümmert sich um das Mapping von Tasks und Fäden, d.h. er ordnet den Aufgaben Fäden zu, die die Aufgaben dann abarbeiten.

4.5.1 Ranges

Ein *Range* stellt einen Wertebereich dar, mathematisch mit einem Zahlenintervall zu vergleichen, nur dass ein Range abstrakt ist und somit mit beliebigen Datentypen verwendet werden kann. Mit dessen Hilfe können später Schleifen und Abläufe parallelisiert werden. Hierzu sollte der Range teilbar sein. Teilbarkeit bedeutet, dass wenn angefordert, der Bereich sich selber in zwei – nicht notwendigerweise gleichgroße – Teile zerlegen kann.

Diese Aufgabe, die Zerlegung des Ranges, übernimmt ein besonderer Konstruktor – im folgenden Partitionskonstruktor genannt. Er ist ein Kopierkonstruktor, der zusätzlich noch einen Parameter vom Typ `split` übergeben bekommt. Das Schlüsselwort `split` wird von TBB definiert und dient nur dazu, um den Partitionskonstruktor von anderen Konstruktoren zu unterscheiden. Der Parameter wird deshalb niemals ausgewertet. Statt eine Kopie des übergebenen Ranges zu erstellen, wird nur ein Teilbereich übernommen. Zusätzlich wird der Wertebereich des übergebenen Ranges verkleinert, so dass die beiden Ranges am Ende disjunkt sind.

Wenn ein Bereich für eine parallele Ausführung, z.B. eine for-Schleife verwendet wird, so ruft der Task-Scheduler von TBB, der für die Koordinierung der Fäden zuständig ist, zuerst die Methode `is_divisible()` auf, um herauszufinden, ob der Bereich weiter aufgeteilt werden kann und falls ja, den Partitionskonstruktor auf, um den Bereich auf die Fäden aufzuteilen.

An dieser Stelle möchte ich ein Beispiel für einen Range vorstellen:

```
struct MyIntRange
{
  int from, to;

  bool empty() const {
    return (from == to);
  }

  int size() const {
    return to - from;
  }

  int begin() const {
    return from;
  }

  int end() const {
    return to;
  }

  bool is_divisible() const {
    return (size() > 10);
  }

  MyIntRange(int x1, int x2) {
    from = x1;
    to = x2;
  }
```

```
    MyIntRange(MyIntRange& r, split) {
      int middle = (r.from + r.to) / 2;
      from = r.from;
      to = middle;
      r.from = middle;
    }
  };
```

`MyIntRange` verwaltet das halboffene Integer-Intervall von `from` bis `to`. In unserem Beispiel soll ein Bereich nur dann weiter zerlegt werden, wenn er größer als 10 ist. Der Partitionskonstruktor zerlegt ein Intervall "in der Mitte" in zwei gleichgroße Teile.

4.5.1.1 Vordefinierte Range-Templates

TBB liefert bereits eine Reihe von Templates mit, um Ranges zu erstellen.

```
template<typename Value>
class blocked_range;
```

verwaltet einen eindimensionalen Bereich,

```
template<typename RowValue,
         typename ColValue>
class blocked_range2d;
```

einen zweidimensionalen Bereich und

```
template<typename PageValue,
         typename RowValue,
         typename ColValue>
class blocked_range3d;
```

einen dreidimensionalen Bereich.

Dadurch wird das Schreiben von parallelem Code sehr vereinfacht, weil man sofort auf diese vordefinierten Templates zurückgreifen kann.

In den Konstruktoren kann per Parameter festgelegt werden, wie klein der Bereich maximal werden darf. Das entspricht dem Wert 10 aus dem vorherigen Beispiel. Wird der Wert zu klein gewählt, so entsteht durch zu viele Aufteilungen zu viel Overhead und der Performance-Gewinn durch Parallelität geht verloren.

4.5.2 Parallele for-Schleife

Eine serielle for-Schleife sieht zum Beispiel wie folgt aus. Ein Integer-Array der Größe 100 wird iteriert und der Wert jedes Elements im Array dabei verdoppelt.

```
int data[100];
for(int i = 0; i < 100; i++)
  data[i] *= 2;
```

Anders als beim Einsatz von OpenMP muss der Code für die Verwendung von TBB entsprechend in Klassen gekapselt sein. Die Form ist in Anlehnung an den Funktor der STL [15] gehalten und ermöglicht somit generisches Programmieren. TBB erwartet lediglich einen `operator()` in der Klasse, der den Rumpf der for-Schleife enthält.

```
class DoubleInts
{
  int* const numbers;

public:
  DoubleInts(int* array) : numbers(array)
  {}

  void operator()
       (const blocked_range<int>& r) const
  {
    for(int i = r.begin(); i != r.end();
        i++)
      numbers[i] *= 2;
  }
};
```

Der Konstruktor von `DoubleInts` erhält einen Zeiger auf das Array, sodass der Operator später auf die Daten zugreifen kann. Die for-Schleife selber steckt im `operator()`, der einen Bereich als Parameter erwartet. Da der Bereich später von TBB aufgeteilt und von verschiedenen Fäden bearbeitet wird, finden wir nur die generischen Funktionen `begin()` und `end()` als Intervall-Grenzen. Der eigentliche Aufruf sieht so aus:

```
parallel_for(blocked_range<int>(0, 100),
             DoubleInts(data),
             auto_partitioner());
```

Der erste Parameter gibt den Bereich an, der iteriert werden soll. Das Template `blocked_range` erstellt uns ein halboffenes Integer-Intervall zwischen 0 und 100. Als zweiten Parameter folgt unser Objekt, das die for-Schleife enthält. Der dritte Parameter legt fest, wie TBB die Aufteilung auf die einzelnen Fäden durchführen soll. Mit dem Aufruf von `auto_partitioner()` wird automatisch ermittelt, wie weit der Bereich partitioniert werden soll.

4.5.2.1 Parallele Reduktion

TBB stellt wie OpenMP auch die Möglichkeit, for-Schleifen, die Ergebnisse innerhalb der Schleife zusammenrechnen, zu optimieren. Wir verwenden wieder das Beispiel aus Abschnitt 4.3.3.1 und wollen die Summe der Quadratzahlen von 1 bis 100 berechnen.

Durch TBBs generischen Ansatz kann man hier allerdings nicht wie bei OpenMP einfach nur einen zusätzlichen Parameter mit angeben, sondern muss bereits den ganzen Code etwas anders schreiben.

Die Schwierigkeit bei dieser Art von Code ist, dass bei der Parallelisierung die Variable, die die Summe hält, von mehreren Fäden gleichzeitig manipuliert wird. Bei OpenMP erkannte dies der Compiler zum Übersetzungszeitpunkt und konnte entsprechende Kopien der Variablen für jeden einzelnen Faden anlegen und diese an der impliziten Barriere am Ende der Schleife zusammenaddieren.

In TBB muss dieser Schritt von Hand gemacht werden. Neben dem Operator müssen bei einer Reduktion zusätzlich ein Partitionskonstruktor und eine `join()`-Methode definiert werden, die die Teilergebnisse zweier Fäden zusammenführt. Der Partitionskonstruktor muss im Beispiel keine besonderen Arbeiten

erledigen, sondern, genau wie der normale Konstruktor, nur die Summenvariablen mit 0 initialisieren.

```
class SumOfSquares
{
public:
  long sum;

  SumOfSquares() : sum(0) {}
  SumOfSquares(SumOfSquares& x, split) :
    sum(0) {}

  void operator()
       (const blocked_Range<int>& r) {
    for(int i = r.begin(); i != r.end();
        i++)
      sum += i * i;
  }

  void join(const SumOfSquares& y) {
    sum += y.sum;
  }
};
```

Jedes Objekt von `SumOfSquares` hält sich eine Variable `sum`, die eine Teilsumme enthält. Beim Instanzieren im Hauptprogramm bzw. wenn TBB den Partitionskonstruktor aufruft, wird die Summenvariable auf 0 gesetzt. Die `join()`-Methode führt zwei Teilergebnisse zusammen; in diesem Fall werden die beiden Summen einfach zusammenaddiert.

Der Rumpf der for-Schleife ist wieder im `operator()` zu finden. Im Vergleich zum parallel_for aus Abschnitt 4.4.2 ist dieser Operator allerdings nicht mehr `const`, da er schreibend auf die Variable `sum` zugreift:

```
SumOfSquares sos;
parallel_reduce(blocked_Range<int>(0, 101),
                sos,
                auto_partitioner());
cout << sos.sum;
```

Im Hauptprogramm wird eine Instanz `sos` unserer Klasse `SumOfSquares` angelegt. Mit dem Aufruf von `parallel_reduce()` wird die Schleife ausgeführt. Danach steht uns das Ergebnis im Objekt `sos` zur Verfügung.

4.5.3 Weitere Funktionalität in TBB

TBB ist sehr umfangreich. Um den Funktionsumfang in etwa abschätzen zu können, möchte ich kurz einen Überblick über die weiteren Möglichkeiten bei der Verwendung von TBB geben.

Das atomic-Template gibt die Möglichkeit, bestimmte Operationen atomar auszuführen. U.a. gibt es eine `compare_and_swap()`-Methode. Verschiedene Mutexe können einfach durch Instanzieren einer vordefinierten Klasse erzeugt werden.

TBB unterstützt das Konzept des Pipelining. Hierbei wird eine Menge von Datenpaketen durch mehrere Arbeitsschritte hindurch bearbeitet. Die Arbeitsschritte werden TBB als Filter bekannt gemacht. Man fügt die Filter einer Pipeline hinzu und füttert diese mit den Datenpaketen. TBB kümmert sich nun um die Abarbeitung und verwendet falls möglich Parallelität, um Code in allen Filtern gleichzeitig auszuführen. Dies ist dann der Fall, wenn die Ausführung eines Filters schneller geht, als die der darauf folgenden.

In TBB gibt es vordefinierte generische Container wie Queue, Vector und HashMap. All diese Klassen sind Faden-sicher und können in parallelen Programmen verwendet werden. TBB liefert des Weiteren eine komplette Speicherverwaltung mit, die sich darum kümmert, den Speicher effektiv zu allozieren und wieder freizugeben und vermeidet, dass zwei Fäden sich gegenseitig behindern.

Allem zu Grunde liegend ist der bereits zu Anfang des Kapitels erwähnte Task-Scheduler. Ein Task ist einfach nur eine Folge von Code, der ausgeführt werden muss. Der Task-Scheduler kümmert sich um die Einteilung der Tasks auf die Fäden. Verwendet man beispielsweise einen `parallel_for()`- oder `parallel_reduce()`-Aufruf, so wird intern ein Task angelegt, der für den Rumpf der Schleife zuständig ist. Tasks können auch selbst angelegt werden. TBB liefert eine Reihe von Möglichkeiten, die Ausführung der Tasks zu steuern..

4.5.4 Fazit zu TBB

TBB bietet sehr viele Funktionen, sodass man sich als Programmierer mehr auf das Wesentliche konzentrieren kann. Viele Aufgabenstellungen, wie zum Beispiel Schleifen, kann man mit vordefinierten Mitteln aus TBB lösen.

Trotz der doch relativ vielen Codezeilen, die gegenüber der seriellen Programmierung hinzukommen, ist ein Programm, das TBB nutzt, noch gut lesbar. Bei der Parallelisierung von Schleifen, Arbeitsschritten oder ganzen Tasks wird der Code in einer Klasse gekapselt und TBB übergeben.

Durch die Abstraktion werden die Gültigkeitsbereiche der Variablen separiert. Private Daten können so nicht unbeabsichtigt von anderen Fäden verändert werden.

5. FAZIT

In den ersten beiden Abschnitten haben wir gesehen, dass mit Parallelität trotz gesenktem Energieverbrauch große Performance-Gewinne möglich sind. Im Hauptteil meiner Arbeit habe ich drei verschiedene Möglichkeiten aufgezeigt, parallele Programme mittels Softwaremitteln zu schaffen.

Message Passing Interface ist besonders für den Einsatz bei größeren Problemen prädestiniert. Dadurch, dass sich die Prozesse durch einen Nachrichtenaustausch "absprechen" müssen, darf diese Kommunikation nicht zu viel Zeit benötigen. Deshalb sollte MPI nur bei solchen Problemen verwendet werden, bei denen das Verhältnis Rechenzeit zu Zeit für Nachrichtenaustausch sehr groß ist. Sind zu viele Nachrichten notwendig, wäre dieser Overhead zu groß, um das Problem effizient zu bearbeiten. Vorteil von MPI ist, dass der Nachrichtenaustausch zum Beispiel auch über TCP/IP erfolgen kann und somit ein Problem auch von einem ganzen Rechnerverbund bearbeitet werden kann.

OpenMP ist mehr dafür geeignet, kleinere Software zu schreiben. Dadurch, dass OpenMP direkt im Compiler integriert ist, kann jeder Programmierer mit wenigen Codezeilen seine Anwendungen parallelisieren. Es ist wichtig, dass nicht nur große Anwendungen "auf den Parallelbetrieb umgestellt" werden, sondern im Laufe der Zeit auch die kleineren Tools, die vor allem im Internet verbreitet

werden, sich die parallele Hardware zu Nutze machen, die immer mehr im Privathaushalt Einzug halten wird.

In TBB sehe ich großes Potential, auch größere Software professionell zu parallelisieren. TBB wird für eine bestimmte Plattform kompiliert und kann so noch spezifischer für die darunter liegende Hardware optimieren. Der objektorientierte Ansatz von TBB dürfte hier auch gelegen kommen, um selbst bei einer enormen Vielzahl von Variablen den Überblick zu behalten. Die Anlehnung an die STL macht den Code leichter wartbar, da die Konzepte der STL einen gewissen Bekanntheitsgrad haben.

Das waren jetzt drei unterschiedliche Ansätze, Programme parallel in Software umzusetzen. Je nach Problemstellung kann die Entscheidung, welcher Ansatz zu wählen ist, anders ausfallen.

Alle diese Variante sind völlig unabhängig von der Prozessoranzahl. MPI kann mittels entsprechender Funktionsaufrufe einfach mit allen verfügbaren Prozesse Nachrichtenaustausch betreiben. OpenMP und TBB – mittels Task-Scheduler – stellen die Anzahl der Prozessoren zu Laufzeit selber fest. Das bedeutet, dass so geschriebene Programm skalierbar sind und auf einem System mit 4 Kernen ebenso läuft wie auf einem System mit 128 Kernen.

Es darf nur nicht vergessen werden, dass die Probleme auch wirklich parallelisierbar sein müssen, um sie dann entsprechend in Software umzusetzen. Hat man ein Problem, welches sich nicht parallelisieren lässt und seriell ausgeführt werden muss, so ist es irrelevant, wie viele Prozessorkerne man in seinem System zur Verfügung hat.

6. REFERENZEN

[1] Vortrag von Intel-Direktor James Reinders
http://www.zdnet.de/specials/whiteboard-series/0,39038336,39157958,00.htm

[2] Wikipedia, Cell (Prozessor)
http://de.wikipedia.org/wiki/Cell_(Prozessor)

[3] MPI Standard
http://www.mpi-forum.org/

[4] MPICH2
http://www.mcs.anl.gov/research/projects/mpich2/

[5] pyMPI
http://pympi.sourceforge.net/

[6] OpenMP Application Program Interface
Version 3.0, Mai 2008, Seite 12
http://www.OpenMP.org/mp-documents/spec30.pdf

[7] GOMP - An OpenMP implementation for GCC
http://gcc.gnu.org/projects/gomp/

[8] OpenMP Fortran Application Program Interface
Version 1.0, Oktober 1997
http://www.OpenMP.org/mp-documents/fspec10.pdf

[9] OpenMP C and C++ Application Program Interface
Version 1.0, Oktober 1998
http://www.OpenMP.org/mp-documents/cspec10.pdf

[10] OpenMP Fortran Application Program Interface
Version 2.0, November 2000
http://www.OpenMP.org/mp-documents/fspec20.pdf

[11] OpenMP C and C++ Application Program Interface
Version 2.0, März 2002
http://www.OpenMP.org/mp-documents/cspec20.pdf

[12] OpenMP Application Program Interface
Version 2.5, Mai 2005
http://www.OpenMP.org/mp-documents/spec25.pdf

[13] OpenMP Application Program Interface
Version 3.0, Mai 2008
http://www.OpenMP.org/mp-documents/spec30.pdf

[14] Threading Building Blocks
http://www.Threadingbuildingblocks.org/

[15] STL - The Standard Template Library
Function Objects
http://www.sgi.com/tech/stl/functors.html