Multi- und Many-Core

Schriftliche Ausarbeitung zum Seminar

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von: Benjamin Warnke
E-Mail-Adresse: 4bwarnke@informatik.uni-hamburg.de
Matrikelnummer: 6676867
Studiengang: Bachelor Informatik

Betreuer: Michael Kuhn

Hamburg, den 01.03.2017

Abstrakt

Die folgende Ausarbeitung beschreibt die Notwendigkeit von Multi- und Manycore-Prozessoren sowie die sich daraus ergebenden Herausforderungen für die Programmierung. Es werden verschiedene Arten der Parallelisierung vorgestellt mit ihren Vor- und Nachteilen sowie ihre Verwendung mit Hilfe von Beispielen. Außerdem erfolgt abschließend ein Vergleich der Verfahren anhand der Messdaten aus einem mit verschiedenen Verfahren berechneten Programm.

Inhaltsverzeichnis

1 Einleitung

Grundsätzlich besteht immer der Wunsch, den Rechendurchsatz der eigenen Programme zu maximieren. In der Abbildung 1.1 kann man sehen, dass im Verlauf der Zeit hierbei verschiedene Komponenten im Fokus waren.

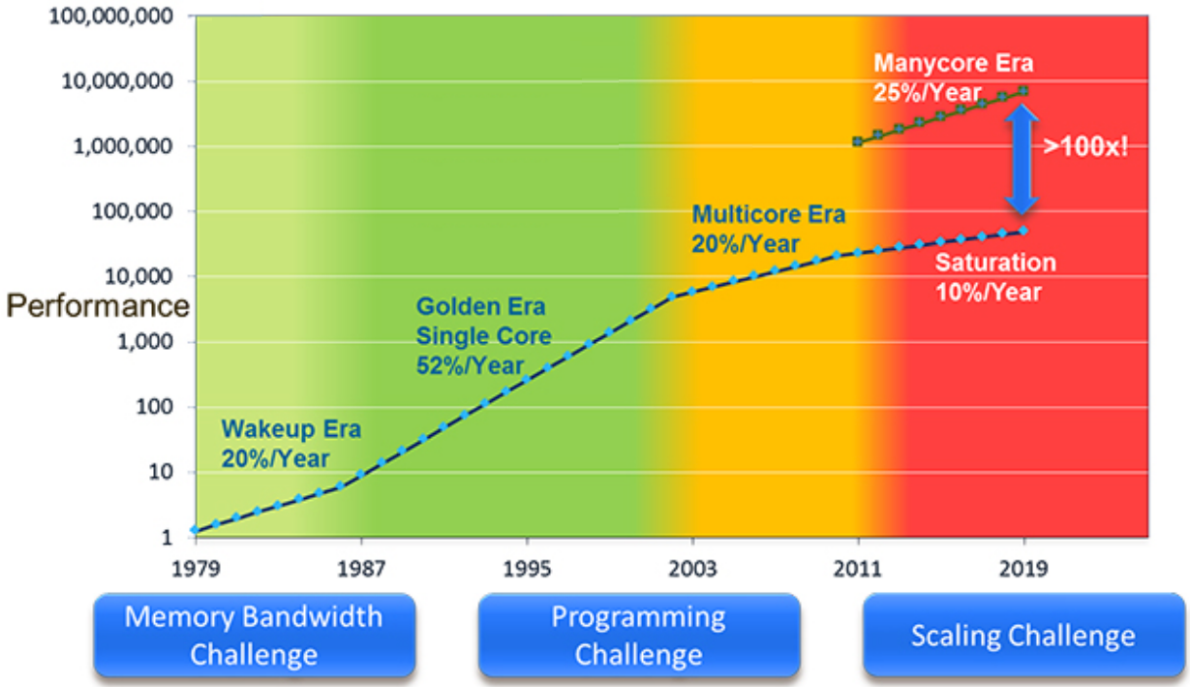

Abbildung 1.1: Herausforderungen im Verlaufe der Zeit verändert nach Quelle [1]

In den 1980ern bestand die größte Herausforderung darin, die benötigte Speicherbandbreite der eigenen Programme zu minimieren, um das Maximale aus der verfügbaren Hardware herauszuholen.

In der Zeit zwischen 1987 und 2003 wurden die Single-Core-Prozessoren immer schneller. Auch ohne zusätzliche Optimierung seiner Programme konnte man im Lauf des Jahres durch die nächste Prozessorgeneration die Programmlaufzeit verkürzen.

2003 wurden Multicore-Prozessoren der neue Standard. Um die Programmlaufzeit weiter zu verkürzen, wurde es jetzt erforderlich, die eigenen Programme zu parallelisieren. Die Anzahl der jeweils verfügbaren Kerne nahm im Verlauf der nächsten 8 Jahre langsam zu, so dass es notwendig wurde, die zu berechnenden Aufgaben in immer mehr unabhängige Segmente zu zerlegen.

Seit 2011 erlauben Beschleunigerkarten wie z.B. Grafikkarten das freie Programmieren, so dass beliebige Programme auf die große Anzahl der Kerne zugreifen können. Für die effektive Nutzung ist allerdings eine große Umstrukturierung der Programme erforderlich. Die maximal erreichbare Geschwindigkeitssteigerung ist jetzt nur noch durch die verfügbare Hardware begrenzt und damit durch die verfügbaren finanziellen Ressourcen.

1.1 Warum Multi- und Manycore

Warum wurden Multi- und Manycore Architekturen erforderlich? Immer anspruchsvollere Programme benötigten immer mehr Rechenleistung in immer kürzerer Zeit.

Um die Rechenleistung zu erhöhen, war es lange Zeit möglich, den Takt der Hardware zu steigern. Dazu mussten die Transistoren auf der untersten Ebene schneller schalten, damit der Prozessor schneller rechnen kann. Die schnellere Umschaltung der Transistoren erforderte eine Erhöhung der Spannung quadratisch zur Umschaltgeschwindigkeitssteigerung.

Wenn man die Spannung erhöht, dann fließt mehr Energie durch die Hardware. Die gesamte Energie wird in Wärme umgewandelt. Die im Prozessor entstehende Hitze steigt quadratisch zur eingesetzten Spannung - somit insgesamt mit der 4ten Potenz zur Frequenz. Damit die Hardware keine Schäden erleidet und schneller läuft, darf eine bestimmte Betriebstemperatur nicht überschritten werden. Die Kühlung konnte die zunehmende Wärmeproduktion technisch nicht mehr ableiten, so dass die Steigerung der Taktung nicht weiter möglich war. Da die Hardware aber im Laufe der Zeit immer kleiner geworden ist, kann man auch mehrere Prozessoren im gleichen Volumen unterbringen. Die gleichzeitige Berechnung verschiedener Aufgaben ermöglicht es, den Takt des einzelnen Prozessor-Kerns zu senken und in Summe trotzdem mehr Berechnungen pro Zeiteinheit durchzuführen. Die Senkung des Takts wiederum erlaubt eine Senkung der Spannung, wodurch sich die benötigte Energie pro Berechnung reduziert und die Hardware effizienter wird.

1.2 Probleme von Mehrkernsystemen

Die Verkürzung der Rechenzeit beruht auf der gleichzeitigen Ausführung mehrerer Programmteile auf verschiedenen Prozessorkernen. Der Takt des einzelnen Kerns ist niedriger. Daher muss das Programm parallelisiert werden, um die verfügbare Leistung nutzen zu können. Dies führt zu verschiedenen Problemen.

- **Speicherbandbreite** Da sich alle Prozessor-Kerne den gleichen Speicher teilen, muss der Speicher nun mehrere Prozessoren mit Daten versorgen. Wenn das Programm viele Rechnungen pro Datensatz im Prozessor durchführen muss, dann ist das kein Problem. Sobald der Prozessor die Ergebnisse schnell

berechnen kann, da ein Programm pro Datensatz nur wenige Rechnungen durchführt, werden häufige Speicherzugriffe erforderlich und dies überschreitet die Kapazität der Speicherbandbreite. Somit wird die Speicherbandbreite zu der Geschwindigkeit-begrenzenden Komponente.

- **Debuggen** Da mehrere Programmteile auf verschiedene Prozessoren verteilt werden, ergibt sich bei auftretenden Fehlern das Problem, dass der Zustand des Programms nicht sofort eindeutig ersichtlich ist. Mehrere Prozessor-Kerne können an verschiedenen Stellen im Programm sein. Dies erschwert erstens die Darstellung, wo ein Thread gerade ist, zum anderen kann eine Variable nun mehrere verschiedene Instanzen zur gleichen Zeit haben. Aus diesem Grund sind für die Fehlersuche komplizierte Debugger erforderlich.

- **Algorithmus muss parallelisierbar sein** Für die Verteilung der Aufgaben auf mehrere Prozessorkerne ist es erforderlich, die Algorithmen zu parallelisieren. Es ergibt sich das Problem, dass nicht jeder Algorithmus parallelisierbar ist. Andererseits kann auch ein grundsätzlich parallelisierbarer Algorithmus nicht geeignet sein, da durch die Parallelisierung für die Aufteilung und Zusammensetzung des Ergebnisses ein Mehraufwand an Rechenzeit benötigt wird.

- **Nichtdeterminismus** Aus technischen Gründen können die verschiedenen Rechenkerne nicht exakt im gleichen Takt rechnen. Außerdem sind die Programme in mehreren Zuständen zur gleichen Zeit. Aus diesen Gründen kann bei mehrfachem Start des Programms in jeder Ausführung eine unterschiedliche Reihenfolge der Berechnung auftreten. Somit ist nicht mehr vorhersagbar, in welcher Reihenfolge Teil-Ergebnisse berechnet werden. Dies wiederum kann zu unterschiedlichen Ergebnissen führen.

- **Overhead** Zur Koordinierung der parallel laufenden Programme, sind weitere Programmteile erforderlich, um die Aufgaben zu verteilen und später wieder zusammenzufassen. Das Starten eines parallelen Programms dauert somit immer etwas länger, als das Starten eines sequentiellen Programms.

- **Synchronisierung** Wenn mehrere Teil-Ergebnisse gleichzeitig berechnet werden, dann muss das Endergebnis anschließend zusammengefasst werden. Um mehrere Ergebnisse zusammenzufassen, werden häufig Sperrverfahren benutzt. Wenn mehrere Threads auf eine gemeinsame Variable schreiben können, lösen sie direkt vor Benutzung der gemeinsamen Variablen eine Sperrung aus. Dies kann einen Deadlock zur Folge haben, was bedeutet, dass verschiedene Threads sich gegenseitig blockieren und das Programm anhält, weil jeder Thread auf einen anderen wartet.

1.3 Multi- und Manycore

Multi- und Manycore-Komponenten werden heutzutage in Computer-Hardware verwendet. Zur Programmierung werden verschiedene Bibliotheken eingesetzt.

- Systemprozessor (2 – 16 Kerne)
  Der Systemprozessor hat mehrere Kerne und fällt somit unter den Begriff **Multicore**. Einige Bibliotheken, die zur Parallelisierung verwendet werden können, sind:
  - POSIX-Threads
  - OpenMP
  - TBB
  - MPI
  - OpenACC
- GPU (> 1000 Kerne)
  Grafikkarten sind für die Berechnung von Bildern optimiert. Bei der Berechnung eines Bildes wird die gleiche Formel auf viele Daten angewendet. Da dieser Vorgang sehr gut parallelisierbar ist, haben Grafikkarten besonders viele Rechenkerne und gehören somit zu den **Manycore**-Prozessoren. Auf Grafikkarten kann man Programme unter anderem mit den folgenden Bibliotheken parallelisieren:
  - CUDA
  - OpenCL
  - OpenGL (für Grafikberechnungen)
  - OpenACC
- Xeon Phi (~ 70 Kerne) → Manycore
  Xeon Phi's sind eine Mischung von Prozessoren und Grafikkarten. In der Abbildung 1.2 ist die Idee dieser Kombination dargestellt. Da eine Xeon Phi relativ viele Kerne hat, ist der Takt niedriger als bei Prozessoren mit wenigen Kernen. Für die effiziente Nutzung der möglichen Leistungsfähigkeit einer Xeon Phi ist eine starke Parallelisierung des Programms erforderlich. Wenn die Programme zusätzlich noch vektorisierbar sind, dann kann eine Xeon Phi noch mehr Berechnungen in kürzerer Zeit durchführen. Ursprünglich gab es Xeon Phi Beschleuniger-Karten nur in Form von Co-Prozessoren, die mithilfe von PCI-Express angeschlossen wurden. Mittlerweile gibt es auch gesockelte Varianten der Xeon Phi, wodurch auch der Systemprozessor stark parallele Codes ausführen kann.

Die Xeon-Phi in der PCI-Express Variante kann auf folgende Arten genutzt werden:

- **Aufteilung der Rechenlast mit dem Systemprozessor** Die Xeon Phi kann sich mit dem Systemprozessor die auszuführenden Programme teilen. Wenn das Betriebssystem entsprechend konfiguriert ist, dann werden die Kerne der Xeon Phi als zusätzliche Prozessorkerne erkannt. Danach können Programme beliebig auf der Xeon Phi und dem Systemprozessor laufen.
- **Eigenständiger Rechnerknoten** In diesem Modus läuft auf der Xeon Phi ein komplett unabhängiges Betriebssystem. Dies hat den Vorteil, dass keine Daten zwischen dem Hauptspeicher und der Xeon Phi kopiert werden müssen. Zusätzlich kann die Xeon Phi in diesem Modus eine eigene IP-Adresse bekommen, wodurch schon innerhalb von einem Rechenknoten mehrere Unterknoten realisiert werden können. Da diese Unterknoten einen gemeinsamen PCI-Bus haben, ist die Kommunikation zwischen mehreren Xeon Phi's in einem Knoten deutlich schneller.
- **Verwendung als GPU** Eine Xeon Phi kann genau wie eine Grafikkarte zur Berechnung von Bildern verwendet werden. In diesem Modus wird die gesteigerte freie Programmierbarkeit nicht ausgenutzt, wodurch das Potential der Xeon Phi nicht genutzt wird. Da die Xeon Phi im Vergleich zu richtigen Grafikkarten sehr wenig Kerne hat, ist allerdings das Preis-Leistungsverhältnis relativ schlecht.

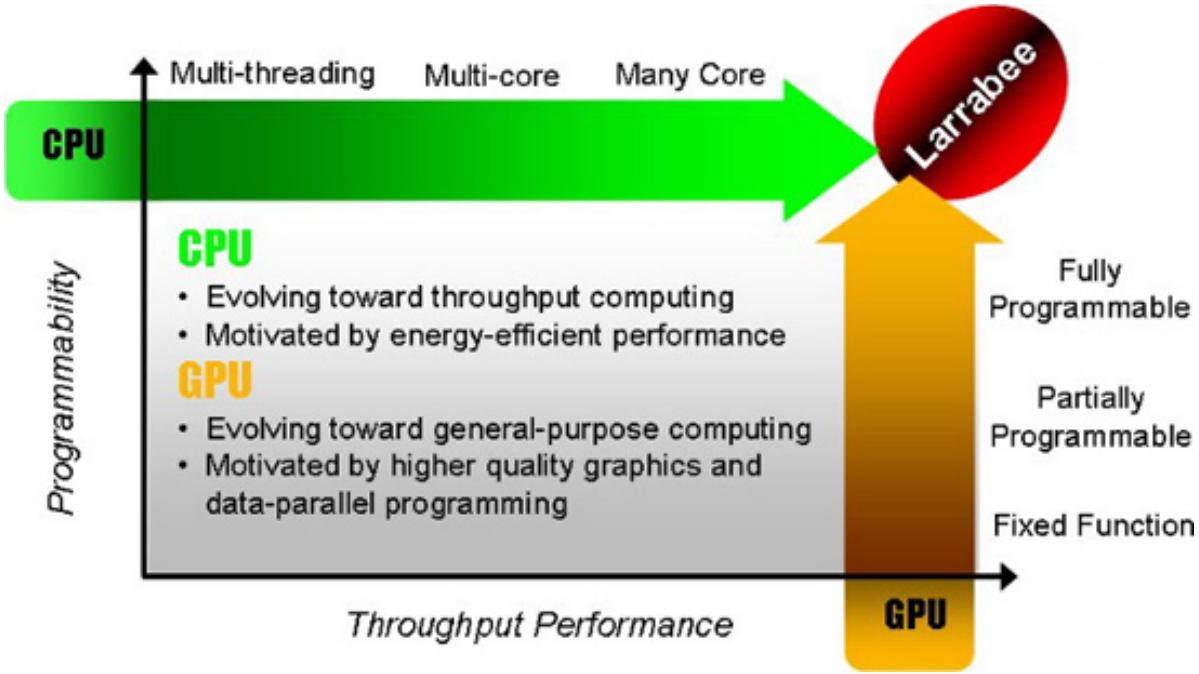

Abbildung 1.2: Xeon Phi [6]

1.4 Thread vs Prozess

Bei der Verwendung von Multi- oder Manycore-Prozessoren oder Clustern werden zur Parallelisierung Prozesse und Threads benötigt. Beide werden vom Betriebssystem verwaltet.
Jedes Programm benötigt mindestens einen **Prozess**, um die Rechnungen durchführen zu können.

- **Speicher** Jeder Prozess hat einen eigenen Adressraum im Speicher. Auf der einen Seite erschwert dies die Kommunikation zwischen Prozessen. Auf der anderen Seite können Prozesse auf verschiedenen Knoten ausgeführt werden, wodurch sich der insgesamt verfügbare Speicher erhöht wie auch die insgesamt verfügbare Speicherbandbreite.

- **Kommunikation** Bei Parallelisierung auf Prozess-Ebene muss die Datenaufteilung sowie die resultierende Kommunikation dazu explizit kodiert werden, da die aktuellen Compiler hierbei nicht helfen können. Für Kommunikation zwischen Prozessen kann man entweder Nachrichten über das Netzwerk versenden oder gemeinsame Dateien benutzen. Wenn 2 Prozesse auf dem gleichen Knoten ausgeführt werden, kann das Betriebssystem beiden Prozessen einen gemeinsamen Speicherbereich zuweisen. Dies beschleunigt die Kommunikation.

- **Fehlerfall** Das Abstürzen eines Prozesses hat zunächst keinen unmittelbaren Einfluss auf andere Prozesse. Benötigt der andere Prozess allerdings die Kommunikation mit dem abgestürzten Prozess, kommt er meistens auch durch den fehlenden Kommunikationspartner zum Absturz. Ein auf Prozessebene parallelisiertes Programm kann auf beliebig vielen Knoten laufen. Es gibt nur wenige Debugger, die bei einer derartig komplexen Fehlersuche helfen können. Aus diesem Grund ist der Anschaffungspreis dafür auch sehr hoch.

Ein Prozess kann aus mehreren **Threads** bestehen, die sich die Ressourcen des Prozesses teilen.

- **Speicher** Die Kommunikation zwischen Threads ist schnell, da sich alle Threads eines Prozesses einen gemeinsamen Speicher teilen. Dies ermöglicht die gemeinsame Verwendung von Daten und erübrigt das Kopieren. Die Teilung des Arbeitsspeichers aller Threads wirkt sich aber auch nachteilig aus, da auch die Speicherbandweite unter den Threads aufgeteilt wird.

- **Kommunikation** Compiler können bei Threads die Parallelisierung stark unterstützen, da keine größeren Daten kopiert werden müssen. Dies reduziert den Arbeitsaufwand für den Programmierer.

- **Fehlerfall** Im Fehlerfall wirkt sich die Parallelisierung mit Hilfe von Threads nachteilig aus, da der Absturz eines Threads unmittelbar auch die anderen Threads dieses Prozesses beendet.

2 Thread-Parallelisierung

2.1 TBB

Thread Building Blocks ist eine C++ template Bibliothek, die dem Programmierer verschiedene Funktionen wie auch Datentypen zur Parallelisierung bereitstellt.

Stärken:

- **Datentypen** TBB beinhaltet Datentypen, auf die parallel zugegriffen werden kann z.B. "vector", "queue", map". Bei schon bestehendem Code bietet dies den Vorteil, dass dieser mit nur kleinen Änderungen parallelisiert werden kann.
- **Work-stealing** Da TBB work-stealing implementiert, ist diese Bibliothek besonders bei nicht balancierten Schleifen im Vorteil, da alle Kerne solange wie möglich an dem Problem mitrechnen können.
- **Open Source** TBB ist Open Source, wodurch es möglich ist, den Code so umzuschreiben und zu erweitern, wie es für das eigene Projekt am besten ist.
- **Optimierung** TBB liegt komplett als Source-Code vor. Dadurch hat der Compiler viele Freiheiten, um das Programm zusätzlich zu optimieren.
- **Austauschbarkeit von Funktionen** Dadurch, dass TBB als Source-Code vorliegt, ist der Austausch von Funktionen möglich. Dadurch ist TBB zur Laufzeit anpassbar.
- **Code Lesbarkeit** Templates ermöglichen eine gute Code Lesbarkeit.

Schwächen:

- **Sprache** Ein Nachteil von TBB besteht darin, dass es nur C++ unterstützt.
- **Verbreitungsgrad** Da TBB nicht so weit verbreitet ist wie z.B. OpenMP, ist es nicht auf jedem Cluster vorinstalliert. Auf dem in dieser Arbeit verwendeten Test-Cluster läuft TBB vergleichsweise langsam.

```
#include "tbb/tbb.h"
void ParallelLoop(float a[]) {
    tbb::parallel_for(0, a.length, [&](int i) {
        DoSomething(i,a[i]);
    });
}
```

Listing 2.1: TBB Parallele Schleife

```
std::queue<T> MySerialQueue;
T item;
void SerialPop(){
    if(!MySerialQueue.empty() ) {
        item = MySerialQueue.front();
        MySerialQueue.pop_front();
        DoSomething(item);
    }
}
```

Listing 2.2: Standard Queue

```
#include "tbb/tbb.h"
tbb::concurrent_queue<T> MyParallelQueue;
T item;
void ParallelPop(){
    if(MyParallelQueue.try_pop(item) ) {
        DoSomething(item);
    }
}
```

Listing 2.3: TBB Thread sichere Queue

2.2 OpenMP

OpenMP unterstützt die Thread-Parallelisierung durch Compiler Anweisungen. Stärken:

- **Verbreitungsgrad** OpenMP ist relativ weit verbreitet und wird von den meisten Compilern direkt unterstützt.
- **Parallelisierung** OpenMP ermöglicht durch Schreiben von Anweisungen im Code für den Compiler, dass dieser die Parallelisierung ausführt. Dadurch ist es möglich, das Programm fortlaufend in kleinen Schritten zunehmend zu parallelisieren. Dies erleichtert gegebenenfalls auch die Fehlersuche.
- **Vektorisierung** Bei der Vektorisierung werden größere Datenmengen von einem Prozessorkern gleichzeitig berechnet. Da der Compiler die Parallelisierung übernimmt, wird die Vektorisierung nicht nachteilig beeinflusst.

Schwächen:

- **Datentypen** Ein Nachteil von OpenMP ist es, dass keine parallelen Datentypen zur Verfügung gestellt werden. Wenn man mit mehreren Threads auf das gleiche Objekt zugreifen möchte, dann muss man durch den eigenen Code sicherstellen, dass keine miteinander in Konflikt stehenden Schreibzugriffe auftreten.
- **Anwendbarkeit** OpenMP besteht lediglich aus Compiler-Anweisungen. Voraussetzung für den Einsatz von OpenMP ist es daher, dass der Compiler OpenMP unterstützt. Wenn dies nicht der Fall ist, werden die Compiler-Anweisungen ignoriert und das Programm wird übersetzt und anschließend sequentiell ausgeführt. Insgesamt ergibt sich dann zwar das gleiche Ergebnis, benötigt aber mehr Zeit.
- **Fehlergefahr** OpenMP sieht standardmäßig alle Variablen als shared an. Aus diesem Grund muss man bei der Programmierung private Variablen explizit als solche markieren. Sonst ist es leicht möglich, Fehler zu erzeugen. Bleiben Variablen irrtümlich public, können mehrere Threads auf den gleichen Variablen schreiben, so dass die Möglichkeit besteht, dass ein undefiniertes falsches Ergebnis auftritt.

```
1 void ParallelLoop(float a[]) {
2 #pragma omp parallel for
3     for(int i = 0; i < a.length; i++){
4         DoSomething(i,a[i]);
5     }
6 }
```

Listing 2.4: OpenMP Parallele Schleife

3 Prozess-Parallelisierung - MPI

Die Parallelisierung von Prozessen wird derzeit standardmäßig mit Hilfe von Message Passing Interface durchgeführt, da es keine Alternativen gibt. MPI ist nur ein Interface, so dass viele verschiedene Implementierungen möglich sind. Die verschiedenen Implementierungen sind weitgehend austauschbar. Wird ein Umzug auf ein anderes Cluster erforderlich, muss das Programm lediglich neu compiliert werden. Es gibt Implementierungen von MPI in verschiedenen Sprachen z.B. C, C++, Fortran, Java und Python. Jede Bibliothek, die MPI implementiert, erfordert eine explizite Kodierung für Datenaufteilung sowie die Kommunikation. Die Compilierung von MPI-Programmen erfordert Wrapper-Compiler, die benötigte Bibliotheken automatisch linken können.

```
1  #include <mpi.h>
2  int world_size;// Anzahl der Prozesse
3  int world_rank;// Nummer dieses Prozesses
4  int main(int argc, char** argv) {
5      MPI_Init(&argc,&argv);
6      MPI_Comm_size(MPI_COMM_WORLD, &world_size);
7      MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
8      doSomething();
9      MPI_Finalize();
10 }
11 void doSomething() {
12     printf("Hello World from Rank %d \n", world_rank);
13     int number; //jeder Prozess hat eine eigene Instanz
14     if (world_rank == 0) {
15         number = rand(); //ein Prozess generiert eine
               ↪ Zufallszahl
16         MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
17     } else {
18         MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
               ↪ MPI_STATUS_IGNORE);
19         //der andere Prozess schreibt die Zahl in die Ausgabe
20         printf("received %d \n", number);
21     }
22 }
```

Listing 3.1: MPI Hello World

4 Kombination Threads und Prozesse

Parallelisierung ist sowohl auf der Ebene von Prozessen als auch von Threads möglich. Eine Kombination der Parallelisierung auf beiden Ebenen gleichzeitig ist möglich. Dadurch ist eine Maximierung der Performance möglich, allerdings erhöht es auch das Auftreten von Fehlern.
Vorteile:

- **Performance** Der wichtigste Vorteil der kombinierten Parallelisierung von Prozessen und Threads liegt in der verbesserten Performance. Die Möglichkeiten der Hardware können bis zum Maximum genutzt werden.
- **Effizienz** Innerhalb von einem Knoten kann der Arbeitsspeicher benutzt werden, um das Versenden von Nachrichten zu vermeiden. In Clustern ermöglicht die Parallelisierung auf Prozessebene die Nutzung mehrerer Knoten. Dies steigert die Leistungsfähigkeit des Programms und verkürzt die Laufzeit.
- **Speicher** Die Speicherkapazität und -geschwindigkeit erhöht sich durch Verwendung mehrerer Knoten.

Nachteile:

- **Kommunikation** Der Arbeitsaufwand für den Programmierer erhöht sich, da die Kommunikation und Datenaufteilung explizit kodiert werden muss.
- **Fehlerquellen** Durch die erhöhte Programmkomplexität erhöht sich die Möglichkeit für fehlerhafte Codes und damit die Wahrscheinlichkeit für das Auftreten von Fehlern.
- **Debugging** Die Fehlersuche ist enorm erschwert, weil Fehler auf verschiedenen Ebenen auftreten können wie auch in der Kombination.

5 Vergleich

Zum Vergleich verschiedener Parallelisierungsverfahren bezüglich des Speedup wurden konkrete Berechnungen beispielhaft durchgeführt. Das Ergebnis der Messungen wurde in der Abbildung 5.1 dargestellt. Dazu wurde das Partdiff-Programm aus der Vorlesung "Hochleistungsrechnen"verwendet. Das Jacobi-Verfahren wurde hierzu mit 1024 Interlines und 800 Iterationen durchgeführt. Der Speicherbedarf liegt bei etwa 1 Gigabyte. Das verwendete Cluster hat pro Knoten 2 Sockel mit jeweils 6 Kernen. Durch Hyperthreading können bis zu 24 Threads gleichzeitig ausgeführt werden.

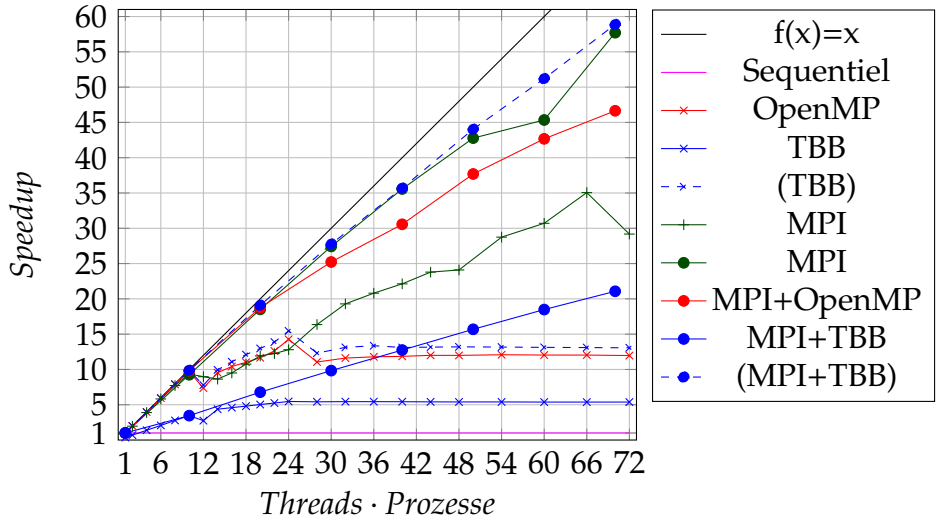

Abbildung 5.1: Skalierung des Programms

Zur Berechnung des Speedup wurde die sequentielle Programm-Variante als Basiswert gesetzt.

Die Datenpunkte auf den durchgezogenen Linien entsprechen den tatsächlich gemessenen Werten.

Für MPI und OpenMP zeigen sich wie erwartet Steigerungen des Speedup.

Für TBB hingegen ist bereits bei Ausführung mit einem Thread die Laufzeit 3x so groß wie bei dem einfach sequentiellen Programm.

Um die Korrektheit der Berechnung zu prüfen, wurde dasselbe Programm auf anderen Computern - allerdings nur mit einem Knoten - getestet, mit dem Ergebnis, dass OpenMP und TBB etwa gleich schnell sind. Mehrere weitere Knoten standen nicht zur alternativen Überprüfung zur Verfügung.

Daraus ist zu schließen, dass die Parallelisierung der Programme nicht der einzige Faktor für Erzielung des Speedup sein kann. Weitere Faktoren könnten sein, dass aufgrund von Inkompatibilitäten eine verlängerte Laufzeit entsteht oder dass keine ausreichende Optimierung für TBB auf dem Test-Cluster vorhanden ist.

Um eine Vergleichbarkeit herzustellen, wurden dann entsprechend des erniedrigten Basiswerts die Messergebnisse multipliziert und als gestrichelte Linie dargestellt. Danach ergibt sich eine Kurve, die den erwarteten Werten entspricht.

Um zu zeigen, dass sich der Speedup bei nur teilweiser Nutzung der Knoten verbessert, wurden zusätzliche Messungen durchgeführt.

Die Nutzung von bis zu 24 Kernen pro Knoten wurden im Diagramm mit Kreuzen dargestellt.

Die Nutzung von 10 Kernen pro Knoten wurden mit ausgefüllten Kreisen dargestellt.

Es zeigt sich, dass sich der Speedup im Verhältnis zu den eingesetzten Kernen vergrößert.

Außerdem lässt sich folgern, dass die Energieeffizienz insgesamt besser ist, da bei gleicher Kernanzahl die Laufzeit kürzer ist. Bei mehr als 10 Threads auf einem Knoten steigt der Speedup deutlich langsamer an als vorher. Dies könnte verschiedene Gründe haben. Zum einen ist es möglich, dass die verfügbare Speicherbandbreite ausgeschöpft ist. Andererseits ist beim Hyperthreading die Hardware nicht komplett doppelt vorhanden, wodurch eine doppelte Geschwindigkeit mittels Hyperthreading gar nicht möglich ist. Der Leistungseinbruch tritt schon bei 12 Threads ein. Dies könnte daran liegen, dass das Betriebssystem diesen einen Kern nutzt, um alle anderen Programme auszuführen. Sobald alle Kerne von dem einem Programm benutzt werden, ist dies nicht mehr möglich.

6 Zusammenfassung

Moderne Anwendungen erfordern immer mehr Rechenleistung. Ohne parallele Architekturen wären diese Anwendungen nicht berechenbar, da es aufgrund der physikalischen Grundlagen nicht möglich ist, unbegrenzt schnelle Prozessoren zu konstruieren. Die einzige Möglichkeit, die Leistung deutlich weiter zu steigern, besteht in der Parallelisierung auch wenn dies die Programmierung erschwert. Um die Parallelisierung zu vereinheitlichen und um Code wiederzuverwenden, wurden verschiedene Bibliotheken entwickelt. In diesem Bericht wurden die Bibliotheken OpenMP, TBB und MPI vorgestellt. In Hochleistungsanwendungen wird häufig Thread basierende Parallelisierung mit Parallelisierung auf Prozessebene kombiniert, um die Vorteile zu vereinen.

7 Literaturverzeichnis

[1] "Adapteva turns to Kickstarter to fund massively parallel processor." [Online]. Available: https://www.extremetech.com/electronics/137085-adapteva-turns-to-kickstarter-to-fund-massively-parallel-processor

[2] "Die Evolution von Multi-Core- zu Many-Core-Prozessoren." [Online]. Available: ftp://ftp.ni.com/pub/branches/germany/2015/artikel/06-juni/04_Der_Kern_der_Materie_Tom_Bradicich_Invision_3_2015.pdf

[3] T. I. Trofimowa, *Physik*. Springer-Verlag, 2013, available: https://books.google.de/books?id=4grKBgAAQBAJ&pg=PA134&lpg=PA134&dq=Joule-Lenz-Gesetz&source=bl&ots=P9-NBYNudv&sig=3fDts6q46ORwjF6jOFkjw2B4b2k&hl=de&sa=X&ved=0ahUKEwjk0bSuhbbSAhXFHpoKHUcmD5YQ6AEIQTAF#v=onepage&q=Joule-Lenz-Gesetz&f=false.

[4] "Ask James Reinders: Multicore vs. Manycore." [Online]. Available: https://goparallel.sourceforge.net/ask-james-reinders-multicore-vs-manycore/

[5] "Xeon Phi." [Online]. Available: http://www.intel.de/content/www/de/de/processors/xeon/xeon-phi-detail.html

[6] "The Future of Larrabee: The Many Core Era." [Online]. Available: http://www.anandtech.com/show/2580/14

[7] "Manycore processor." [Online]. Available: https://en.wikipedia.org/wiki/Manycore_processor

[8] "What is the difference between many core and multi core?" [Online]. Available: http://electronics.stackexchange.com/questions/37982/what-is-the-difference-between-many-core-and-multi-core

[9] "Intel® Threading Building Blocks (Intel®TBB) Developer Guide." [Online]. Available: https://software.intel.com/en-us/tbb-user-guide

[10] "Generic Parallel Algorithms." [Online]. Available: https://www.threadingbuildingblocks.org/tutorial-intel-tbb-generic-parallel-algorithms

[11] "Concurrent Containers." [Online]. Available: https://www.threadingbuildingblocks.org/tutorial-intel-tbb-concurrent-containers

[12] "Message Passing Interface." [Online]. Available: https://de.wikipedia.org/wiki/Message_Passing_Interface

[13] “MPI Hello World.” [Online]. Available: http://mpitutorial.com/tutorials/mpi-hello-world/