Parallele Programmierung

rumif110
Hochschule für Technik Stuttgart

31. Dezember 2007

Inhaltsverzeichnis

1 Einleitung

Die meisten Benutzer moderner grafischer Benutzungsoberflächen und Anwendungsprogrammen kommen heutzutage täglich mit paralleler Programmierung in Berührung auch wenn sie sich dessen möglicherweise nicht bewusst sind. Ein Anwendungsbeispiel, das vielen Benutzern vermutlich aus dem Alltag her bekannt sein dürfte, ist ein Textverarbeitungsprogramm wie Microsoft Word oder OpenOffice.org Writer. Die Rechtschreibkorrektur, die scheinbar während des Tippens ihre Arbeit verrichtet und den Benutzer auf diverse Rechtschreibfehler aufmerksam macht, ist ein Beispiel für parallele Programmierung in Alltag. Ein anderes, sicherlich bekanntes Beispiel dürfte der Web-Browser sein, der mehrere verschiedene Animationen wie zum Beispiel animierte GIF-Dateien (Graphics Interchange Format, dt. *Grafikaustausch-Format*) oder eingebettete Flash-Filme darstellen kann, während die Benutzungsoberfläche weiterhin für weitere Benutzereingaben wie Scrollen ansprechbar bleibt. Ohnehin wäre eine moderne grafische Arbeitsoberfläche wie sie auf verschiedenen Betriebssystemen anzutreffen ist, ohne parallele Programmierung nur schwer denkbar. Hinzukommen die Errungenschaften von Betriebssystemen selbst, die es möglich gemacht haben, transparent für den Benutzer mehrere Programme parallel ausführen zu können. Rein sequentielle Programmierung, bei denen jeweils eine Instruktion nach der anderen abgearbeitet wird, wäre für die genannten Anwendungsfälle unzureichend bis impraktikabel. Nur wenige Benutzer würden es als zumutbar auffassen, dass der Rechner scheinbar einfriert und nicht mehr auf Benutzereingaben reagiert während er den verfassten Text auf Rechtschreibfehler überprüft. Gegenüber rein sequentieller Programmierung kommen auf den Programmierer aber neue Aspekte hinzu, auf die er sein Augenmerk legen muss will er weiterhin stabile und leistungsfähige Applikationen schreiben. Die parallele Programmierung hat sich dennoch trotz erhöhtem Aufwand im Vergleich mit der sequentiellen Programmierung in vielen Fällen als unverzichtbar herausgestellt und ist heute ein fester Bestandteil der Informatik.

2 Sequentielle Programmierung

2.1 Charakteristika

Die sequentielle Programmierung ist das konzeptionelle Gegenstück zur parallelen Programmierung. In der sequentiellen Programmierung wird, wie der Name schon andeutet, alles in einer bestimmten Reihenfolge bearbeitet. Ein gegebenes Problem wird als eine Sequenz von Einzelschritten interpretiert (siehe Abb. 1). Zu jedem Zeitpunkt wird zudem jeweils nur eine Instruktion auf dem Hauptprozessor ausgeführt (unäre Instruktion). Sequentielle Programmierung wird zudem auch beeinflusst durch die Hardware auf der sie läuft. Auf einem Hardwaresystem, auf dem sequentielle Programmierung zum Einsatz kommt, befindet sich ausschließlich ein Hauptprozessor mit einem Rechenwerk. Moderne Prozessoren

[0] Adobe Flash ist eine Technologie zum Erstellen multimedialer Inhalte u.a. im Internet

mit mehreren genutzten Rechenkernen benügen nicht der Definition von sequentieller Programmierung. Dies ist leicht nachzuvollziehen, da ansonsten die Bedingung der unären Instruktion nicht mehr erfüllt wäre, die aussagt, dass jederzeit nur ein Rechenschritt durchgeführt wird. Diese Einschränkung gilt ebenfalls für vernetzte Rechner. Damit ein Programm als sequentiell programmiert betrachtet werden kann, darf kein anderer Rechner an der Lösung der Problemstellung beteiligt sein. Zu beachten sei noch, dass jedes parallele Programm in ein sequentielles Program überführt werden kann. Dabei werden die ursprünglich parallelen Programmteile nacheinander ausgeführt. Damit wird z.B. auf Geratten mit einem Prozesser und einem Rechenkern dem Benutzer Parallelität vorgetäuscht. Mehrere Programme werden sequentiell ausgeführt und teilen sich den Prozessor. Das Betriebssystem teilt jedem Programm eine kurzen Zeitraum an Rechenzeit zu (auch Zeitscheibe genannt). Durch schnelle Umschaltvorgänge zwischen den einzelnen Programmen ensteht der Eindruck von Parallelität. Umgekehrt kann man jedoch nicht jedes sequentielle Programm parallelisieren.

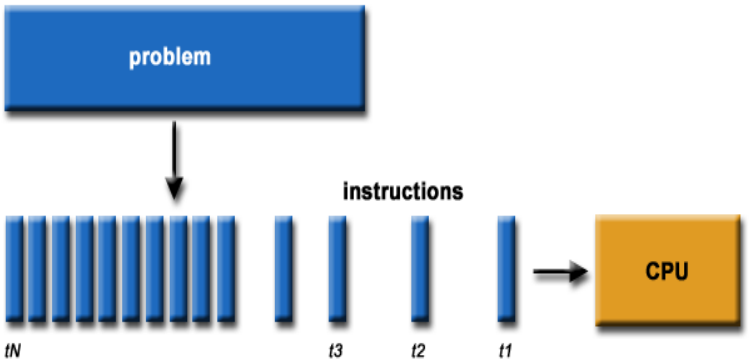

Abbildung 1: Sequentielle Bearbeitung eines Problems (Quelle: https://computing.llnl.gov/tutorials/parallel_comp/images/serialProblem.gif)

2.2 Physikalische Einschränkungen

Durch die beschriebenen Eigenschaften sequentieller Programmierung lassen sich Programme, die zur Menge der sequentiell programmierten Applikationen gehören, hauptsächlich nur über Fortschritte in der Prozessortechnik beschleunigen. Doch diesem Fortschritt ist durch physikalische Schranken Grenzen gesetzt:

- Die Lichtgeschwindigkeit von 30 cm pro Nanosekunde ist die absolute Obergrenze für die Übertragungsgeschwindigkeit von Daten auf seriellen Rechnern. Die Übertragungsgeschwindigkeit von Daten über ein Material wie das in der Netzwerktechnik häufig eingesetzte Kupfer beträgt gar nur maximal 9 cm pro Nanosekunde.

- Die Übertragungsstrecken auf Hauptprozessoren werden mit jeder Chipgeneration mittels Verkleinerung der Chipfläche immer kürzer. Doch auch dieser Verbesserungsprozess wird im Laufe der Zeit sein Ende finden, wenn die einzelnen *CPU* (Central Processing Unit, dt. *Hauptprozessor*) Bestandteile die Größe von Molekülen oder gar Atomen erreichen.

2.3 Logische und wirtschaftliche Einschränkungen

- Viele Problemstellungen in der Realität sind parallel lösbar. Als Folge von streng sequentiellen Schritten ist das Lösen der parallelisierbaren Problemstellung ineffizient. Man betrachte zum Beispiel die Berechnung von optimierten Fahrtrouten. Jede mögliche Route einzeln und für sich zu berechnen würde mehr Zeit in Anspruch zu nehmen als mehrere Routen parallel zu berechnen und anschließend die Ergebnisse zu vergleichen.

- Es existieren Anwendungsanforderungen speziell auf der Ebene der grafischen Benutzungsoberflächen, bei denen sequentielle Programmierung nur unzureichend ist. In der Einleitung wurden schon als Beispiele Web-Browser und Textverarbeitungsprogramme als typische Fälle genannt, bei denen sequentielle Programmierung zu eventuell nur sehr umständlich benutzbaren Applikationen führen würde. Die Pseudo-Parallelität durch das Zeitscheibenverfahren versagt, wenn jedem Programm nicht mehr genug Zeit zugeteilt werden kann und es für den Benutzer merkliche Verzögerungen auftreten.

- Die sequentielle Programmierung unterliegt auch ökonomischen Zwängen. Die in 2.2 beschriebene Miniaturisierung der Chipflächen kommt den physikalischen Begrenzungen immer näher. Dementsprechend steigen der Aufwand und die Kosten um die nächste Miniaturisierungsstufe zu erreichen. Dies schlägt sich demzufolge in Form höherer Preise auch auf Nutzerseite zu Buche. Aus diesem Grunde kann man bei parallelisierbaren Problemstellungen auch eine größere Anzahl von günstigen Rechnern auf das Problem ansetzen. Dieses kann unter Umständen die gleiche oder sogar eine bessere Leistung erzielen im Vergleich zu einem Ein-Prozessor-System.

3 Parallele Programmierung

3.1 Charakteristika

Die parallele Programmierung kommt dort zum Einsatz, wo sich die sequentielle Programmierung als unzureichend im Sinne der Effizienz oder Anwendbarkeit erwiesen hat. Parallele Programmierung beschreibt mit Hilfe einer Programmiersprache den Vorgang eine Problemstellung in mehrere diskrete Teile aufzubrechen. Diese Einzelteile werden dann wiederum wie in der sequentiellen Programmierung als Sequenz von Einzelschritten beschrieben, die man anschliessend parallel auf mehreren Hauptprozessoren ausführen kann (siehe Abb. 2).

Die Einzelergebnisse der parallelen Berechnungen bilden zusammengenommen eine Lösung.

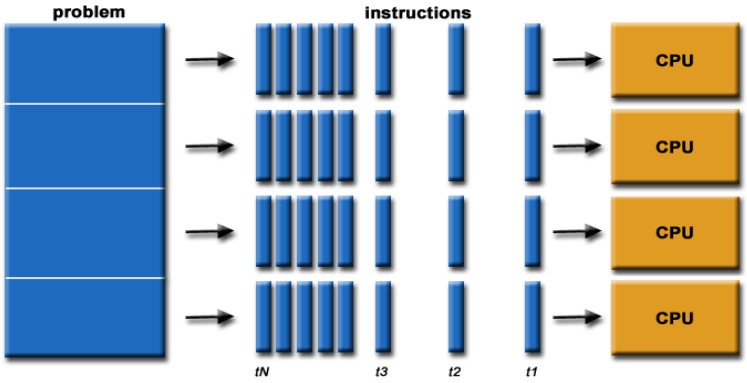

Abbildung 2: Parallele Bearbeitung eines Problems (Quelle: https://computing.llnl.gov/tutorials/parallel_comp/images/parallelProblem.gif)

3.2 Threads

Eine gängige Methode Parallelität in einem Programm zu erreichen sind *Threads* (dt. *Fäden*). Ein Thread ist eine Folge von Befehlen, die ein Hauptprozessor schrittweise bearbeiten muss. Ein Programm, in dem nur ein Thread läuft, kann auch als Prozess bezeichnet werden, wobei dies abhängig vom verwendeten Betriebssystem ist. Ein Prozess ist auch die Bezeichnung für einen gemeinsamen Kontext, den sich mehrere Threads teilen. Threads existieren sowohl auf Programmebene als auch auf Betriebssystemebene. Wenn Programme ihre Threads selbst verwalten so kann man Wissen über die Beschaffenheit der Applikation in der Programmierung von Threads mit einfließen lassen. Dies kann zu einer besseren Leistung des Programmes führen. Auf der Ebene des Betriebssystems verwaltet das jeweilige Betriebsystem die einzelnen Threads innerhalb eines Prozesses wodurch für den Entwickler der Implementierungsaufwand geringer ausfällt.
Threads gelten im Allgemeinen als leichtgewichtig. Damit wird üblicherweise ausgedrückt, dass Threads im Vergleich zu Prozessen günstiger (bezüglich Prozessorzyklen) erzeugt und wieder zerstört werden können. Zudem teilen sich Threads Daten wie statische Variablen (siehe Abb. 3). Echtes *Multithreading* (dt. *Mehrfädigkeit*) ist aber nur auf einem Rechner mit mehreren CPUs oder Prozessorkernen möglich.

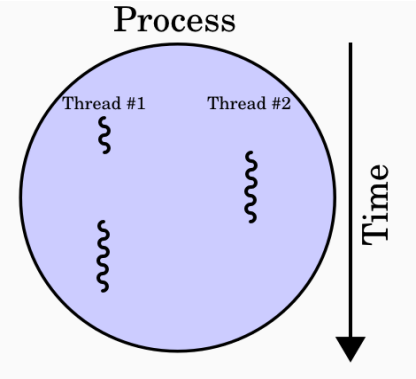

Abbildung 3: Mehrere Threads in einem Prozesskontext (Quelle: Unbekannt)

3.3 Verteilte Prozesse

Das Arbeiten mit Threads findet lokal auf einem Rechner statt. Falls man aber mehrere Rechner zum parallelen Lösen eines Problems benutzen will, stößt man mit Multithreading an die Grenzen dieser Technologie. Threads können nämlich nicht beliebig auf vernetzen Rechnern verteilt werden. Die Erweiterbarkeit einer einzigen physischen Recheneinheit ist aber begrenzt. Ist man an diese Leistungsgrenze gestoßen und ist aber trotzdem auf mehr Rechenleistung angewiesen so muss man verteilte Prozesse als Programmiermodell ins Auge fassen. Bei verteilten Prozessen arbeitet auf verschiedenen Knoten mindestens ein Prozess an der Berechnung einer Teilaufgabe. Dabei schicken sich die einzelnen Prozesse Nachrichten über ein Netzwerk. Diese Nachrichten können Synchronisationsinformationen, neue Berechnungsaufträge oder Ergebnisse enthalten. Üblicherweise ist einer der Prozesse ein Koordinator, der als Startpunkt der parallelen Berechnung fungiert. Der Koordinator steuert die restlichen Knoten und die nimmt Ergebnisse entgegen. Die Kommunikation zwischen den Knoten geschieht dabei aus der Sicht des einzelnen Prozesses transparent. Ein Prozess ruft eine Methode aus einer Bibliothek zum Versenden oder Empfangen von Daten bzw. Nachrichten auf. Diese Bibliothek implementiert die Datentransformationen und Verbindungsverwaltung, die für eine netzgestützte Zusammenarbeit von mehreren Rechnern notwendig sind.
Es gibt zwei unterschiedliche Programmierparadigmen. Das erste ist der Nachrichtenaustausch (engl. *Message passing*). Dabei arbeitet jeder Prozess, der an der verteilten Berechnung beteiligt ist, nur mit seinem lokalen Speicher. Wenn mehrere Prozesse miteinander kommunizieren wollen, müssen sie sich des schon beschriebenen Senden und Empfangen von Nachrichten bedienen (siehe Abb. 4). Dabei ist zu beachten, dass dies eine kooperierende Form der Zusammenarbeit zwischen Prozessen darstellt. Zu jedem Aufruf zum Senden einer Nachricht muss eine entsprechende Funktion zum Emfang existieren und aufgerufen werden. Für diese Art der Kooperation ist aber nicht zwingend notwendig, dass die

Prozesse sich auf verschiedenen Rechnern befinden. Die Prozesse können alle auch auf einem einzigen Rechner ausgeführt werden.

Abbildung 4: Nachrichtenbasierte Interprozesskommunikation (Quelle: https://computing.llnl.gov/tutorials/parallel_comp/images/msg_pass_model.gif)

Ein weiteres Programmierparadigma wird durch das *parallele Datenmodell* (engl. *Data Parallel Model*) beschrieben. Hierbei existiert eine zentrale Datenstruktur, auf die alle Prozesse zugreifen können. Diese Datenstruktur enthält die für die Problemlösung benötigten Daten. Der Hauptanteil der parallelen Berechnung findet auf diesen Daten statt (siehe Abb. 5). Die Datenstruktur ist üblicherweise entweder als ein Array oder Würfel organisiert. Obwohl alle Prozesse sich diese Datenstruktur teilen, arbeitet jeder Prozess alleine auf einem ihm zugewiesenen Teil davon. Eine gemeinsame Datenstruktur setzt das Konzept eines für alle Prozesse erreichbaren gemeinsamen Speichers voraus. Sind die Prozesse aber auf mehreren physikalisch verschiedenen Rechnern verteilt, so muss der Hauptspeicher abstrahiert werden. Es wird eine Abstraktionsebene über dem lokalen Speicher eingefügt. Alle Prozesse greifen auf einen logischen gemeinsamen Speicher zu, der physikalisch aus den verschiedenen lokalen Speichern besteht. Dies ist jedoch für die einzelnen Prozesse transparent.

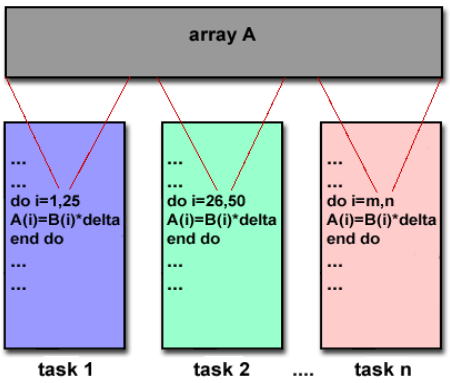

Abbildung 5: Mehrere Prozesse arbeiten auf einer Datenstruktur (Quelle: https://computing.llnl.gov/tutorials/parallel_comp/images/data_parallel_model.gif)

3.4 Unterstützende Frameworks

Die Entwicklung paralleler Algorithmen wird meistens manuell vorgenommen[1]. Dabei wird der Quellcode im Einzelnen durchgearbeitet und auf potentiell parallelisierbare Stellen überprüft. Somit ist der Grad der erreichten Parallelisierung stark abhängig vom Verständnis des Problemes bzw. des dazugehörigen Programmes und der Erfahrung und Kenntniss des Programmierers. Manuelle Parallelisierung birgt dabei die Gefahr, dass die Entwicklung sehr zeitintensiv ist und eine iterative Vorgehensweisen verlangt. Zudem ist es nicht auszuschließen, dass der Code sehr komplex und dadurch fehleranfällig bei Änderungen und generell schwer wartbar ist.
Um diesen Probleme zu mildern wurden Möglichkeiten entwickelt, mit denen man anhand von bekannten Mustern im Quellcode die Parallelisierung automatisieren kann. Dazu gehören insbesondere Compiler, die fähig sind existierenden sequentiellen Code zu parallelisieren. Dabei unterscheidet man zwischen einer voll automatischen Parallelisierung und einer durch den Entwickler gesteuerten Parallelisierung. Bei der ersten Möglichkeit wird der Quellcode durch den Compiler mit Hilfe von Mustererkennung günstige Stellen für die Parallelisierung untersucht. Anschließend wird versucht algorithmisch abzuwägen, ob eine Parallelisierung tatsächlich zu einem Leistungszuwachs führt. Insbesondere Schleifen eignen sich häufig zur Parallelisierung. Die zweite Möglichkeit sieht vor, dass der Programmierer dem Compiler durch Compilerdirektiven mitteilt, welche Stellen parallelisierbar sind und Hinweise auf die beste Art der Parallelisierung mitzugeben. Diese Vorgehensweise schließt die automatische Parallelisierung nicht aus, sondern versteht sich als zusätzliche Ergänzung.
Eines dieser Frameworks ist eine API (Application Programming Interaface, dt: Programmierschnittstelle) namens OpenMP (Open Multiprocessing, dt. *offene Multiprozessierung*). Mit OpenMP werden Programmblöcke nicht explizit parallelisiert, sondern mit Übersetzungsdirektiven als potentiell parallelisierbar markiert. Die eigentliche Parallelisierung wird dann von der API-Implementierung während des Kompiliervorgangs vorgenommen. Eine weitere unterstützende API ist MPI (Message Passing Interface). Diese API unterstützt auf verschiedenen Rechnern verteilte Prozessen mit Kommunikations- und Synchronisationsdiensten, deren Verwendung man jedoch explizit programmieren muss.

4 Herausforderungen der parallelen Programmierung

Will man bei einem Programm Parallelität erreichen, so muss man beim Entwurf des parallelen Algorithmus zusätzliche Aspekte beachten. Diese umfassen sowohl das tiefere Verständnis der zu lösenden Problemstellung als auch technische

[1] Frameworks zur Automatisierung sind erst seit kurzer Zeit für den produktiven Einsatz bereit. OpenMP (Beschreibung siehe weiterer Text) wurde für Fortran und C/C++ erst im Jahr 1997 eingeführt. Implementierungen wie in der GNU Compiler Collection sind erst ab Version 4.2 (erschienen am 15.05.07) verfügbar

Rahmenbedingungen (z.B. ob Rechner mit mehreren Prozesoren oder verteilte Rechner zum Einsatz kommen).

4.1 Grenzen der unterstützenden Frameworks

Obwohl eine automatische Parallelisierung von bereits vorhandenem sequentiellem Code in Fällen von Zeit- und Wissensmangel eine akzeptable Alternative zur manuellen Überarbeitung darstellt, so sollte man doch mögliche Probleme nicht außer Acht lassen. Die heuristische Vorgehensweise bei der automatischen Parallelisierung kann auch zu fehlerhaften Ergebnissen führen. Dies äußert sich eventuell in Leistungseinbußen anstatt Zugewinn und kann sogar zu fehlerhaftem Code führen. Zudem kann eine Heuristik nicht alle Eigenheiten eines Problems erkennen, weshalb nicht gewährleistet werden kann, dass optimal parallelisiert wird. Bei komplexem Code kann die automatische Parellelisierung zudem komplett fehlschlagen, da die eingesetzten heuristischen Verfahren parallelisierbare Stellen im Quellcode aufgrund der Komplexität nicht erkennen.

4.2 Partitionierung

Ist eine Problemstellung definiert, so muss diese am Anfang daraufhin untersucht werden, ob und wie dieses Problem so aufgeteilt werden kann, dass es parallelisierbar wird. Eine Aufteilung in diskrete Segmente ist die Grundvoraussetzung, um einen parallelen Algorithmus zu entwickeln. Dabei gibt es im Wesentlichen zwei Ansätze: Die Aufteilung basierend auf den zu bearbeitenden Daten, als auch eine Aufteilung basierend auf den zu erfüllenden Arbeitsschritten.
Die erste Variante wird *Domain Decomposition* genannt. Es ist der Versuch, die Parallelität über die Daten des zu bearbeitenden Problems zu erreichen. Dazu bricht man die Daten in kleinere, voneinander unabhängige Pakete auf (siehe Abb. 6). Diese können dann parallel bearbeitet und ausgewertet werden.

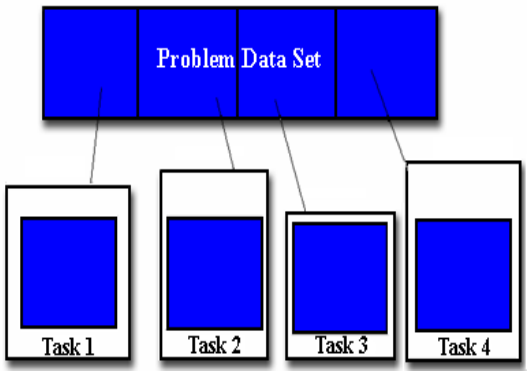

Abbildung 6: Domain Decomposition (Quelle: https://computing.llnl.gov/tutorials/parallel_comp/images/domain_decomp.gif)

Der zweite Ansatz einen parallelen Algorithmus zu entwickeln lautet *Functional Decomposition* (dt. *funktionale Zerlegung*). Bei diesem Ansatz wird das Augenmerk weniger auf die Daten der Problemstellung sondern mehr auf die Schritte zur Berechnung gelegt. Jeder Prozess bzw. Thread arbeitet dann einen Teil dieser Instruktionen ab (siehe Abb. 7).

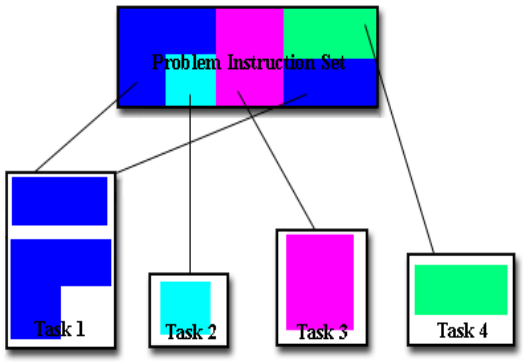

Abbildung 7: Functional Decomposition (Quelle: https://computing.llnl.gov/tutorials/parallel_comp/images/functional_decomp.gif)

Bei beiden Verfahren muss jedoch die Problemlösung parallelisier sein. Bestehen zu starke Abhängigkeiten zwischen den einzelnen Daten oder zwischen den einzelnen Rechenschritten, so sind diese Verfahren nicht anwendbar.

4.3 Wettlaufsituationen

Ein weiterer Umstand beim Parallelisieren, den ein Entwickler beachten muss, sind sogenannte Wettlaufsituationen (engl.: Race Condition). Hierbei ist die Lösung einer parallelen Berechnung abhängig von der Reihenfolge einzelner Anweisungen in Threads oder verteilten Prozessen. Bei vom Betriebssystem gesteuerten Threads ist es aus der Sicht des Programmierers nicht möglich im Voraus zu wissen, welche Threads in welcher Reihenfolge aktiviert werden. Beim Vorhandensein von schreibenden Operationen auf den von Threads gemeinsam genutzten Daten kann es dementsprechend zu verschiedenen Ergebnissen kommen, abhängig davon in welcher Reihenfolge Threads lesen und schreiben. Dieses Problem ist ebenfalls bei verteilten Prozessen mit nachrichtenbasierter Kommunikation möglich. Die Zeit zwischen Versand und Empfang einer Nachricht ist bedingt durch das Betriebssystem und physikalische Eigenschaften des Netzwerkes. Im ersten Fall kann es z.B. zu einer Verzögerung des Versands bzw. des Empfangs kommen, wenn anderen Prozessen bereits die Netzwerkschnittstelle zugewiesen wurde. Im zweiten Fall sind kurzzeitige Störungen bzw. Überlastungen des Netzes möglich, die die Weiterleitung einer Nachricht verlangsamen oder verhindern. In beiden Fällen sind Synchronisationsmechanismen anzuwenden, die die Festlegung einer logischen Reihenfolge in der verteilten Berechnung

ermöglichen.

4.4 Verklemmung

Threads und Prozesse können sich gegenseitig blockieren. Diese Situation wird als Verklemmung bezeichnet (engl. *Deadlock*). Sichern sich zwei unterschiedliche Threads bzw. Prozesse exklusiv den Zugriff auf voneinander verschiedenen Ressourcen zu, entsteht die Verklemmung dann, wenn beide Threads jeweils die Ressource des anderen benötigen, um ihre eigenen Berechnungen fortzusetzen. Im Kontext eines Threads ist diese Verklemmung nicht ersichtlich, da der Grund für die Verzögerung nicht bekannt. So könnte der andere Thread, auf den gewartet wird, immer noch im arbeitendem Zustand sein und keine Verklemmung vorliegen. Es gibt jedoch verschiedene Möglichkeiten eine Verklemmung zu erkennen. So kann z.B. durch Unterbrechungen eine Untersuchung der Threads auf Verklemmungen angestossen werden. Dabei werden durch Algorithmen wie die Breitensuche ein Graph erzeugt, der die gegenseitigen Abhängigkeiten zwischen einzelnen Threads abbildet. Dieser Graph wird dann auf potentielle Verklemmungen untersucht.

5 Zusammenfassung

Wie gezeigt wurde, bietet parallele Programmierung die Möglichkeit weit effizienter Probleme der realen Welt zu lösen, als dies mit sequentieller Programmierung auf Einprozessorsystemen möglich wäre. Dies ist jedoch unter bestimmten Umständen mit einem komplexeren Programmier- und Hardwaremodell verbunden. Frameworks sind in der Lage Parallelisierung mit Heuristiken ohne oder nur mit geringem Eingreifen des Entwicklers vorzunehmen. Aufgrund der Möglichkeit, dass diese Heuristiken ineffizienten oder gar fehlerhaften Code kreieren, ist weiterhin eine Überprüfung auf Geschwindigkeit und Funktionalität erforderlich. Im Falle einer manuellen Parallelisierung liegt es am Entwickler und dessen Verständnis der Problemstellung diese Möglichkeiten zur Parallelisierung zu untersuchen. Falls jedoch Umstände wie großer Codeumfang oder komplexe Aufgabenstellung vorliegen, gestaltet sich dies langwierig und fehleranfällig. Zudem bestehen durch die Parallelität die Möglichkeit Wettlaufsituationen und Verklemmungen ins Programm einzuführen. Die Vermeidung bzw. Erkennung solcher ist nur mit zusätzlichen Algorithmen möglich. Dies wiederum erhöht den Aufwand bei der Entwicklung. Es wurde dennoch gezeigt, dass parallele Programmierung nicht nur längerfristig unvermeidbar, sondern für viele Problemstellungen das bessere Werkzeug zur Lösungsfindung ist. Die Grundlagen der parallen Programmierung sollten jedem Programmierer geläufig sein um leistungsfähige Programme zu erstellen.

Literatur

[1] Blaise Barney, *Introduction to Parallel Computing*
http://www.llnl.gov/computing/tutorials/parallel_comp/
Abgerufen am 15.10.2007

[2] Hans Joachim Pflug, *Skript zum Kurs Paralleles Programmieren fur Mathematisch-Technische Assistenten*
http://www.rz.rwth-aachen.de/mata/downloads/paralleles_programmieren/ppj_skript.pdf
Abgerufen am 15.10.2007

[3] Peter S. Pacheco, *A User's Guide to MPI*
ftp://math.usfca.edu/pub/MPI/mpi.guide.ps.Z
Abgerufen am 15.10.2007