Parallele Programmierung

rumif110
Hochschule für Technik Stuttgart

Inhaltsverzeichnis

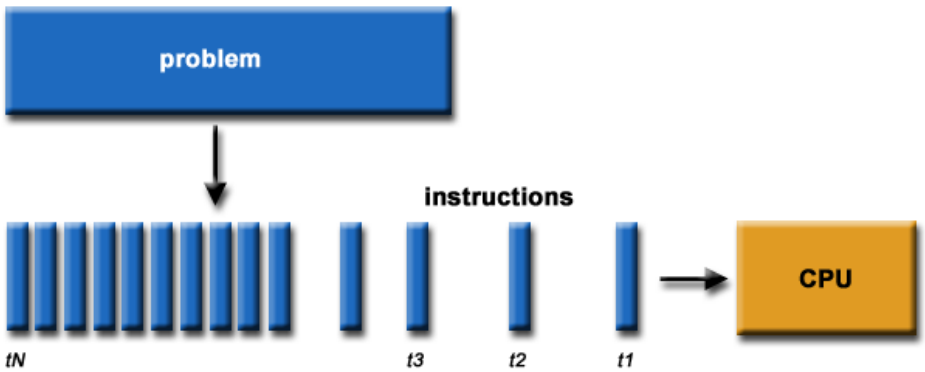
1. Einleitung

Die meisten heutigen Benutzer moderner grafischer Benutzeroberflächen und Anwendungsprogrammen kommen heutzutage täglich mit paralleler Programmierung in Berührung auch wenn sie sich dessen möglicherweise nicht bewusst sind. Ein Anwendungsbeispiel, dass 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 oder eigebettete Flash-Filme darstellen kann während die Benutzeroberfläche weiterhin responsiv gegenüber weiteren Benutzereingaben wie Scrollen bleibt. Ohnehin wäre eine moderne grafische Arbeitsoberfläche wie sie auf verschiendenen 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 inprakikabel. Nur wenige Benutzer würden erdulden, dass der Computer 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 performante Applikationen schreiben.

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 erledigt. Ein gegebenes Problem wird als eine Folge von Einzelschritten interpretiert. Zu jedem Zeitpunkt wird zudem jeweils nur eine Instruktion auf dem Hauptprozessor ausgeführt. Sequentielle Programmierung wird zudem auch bedingt durch die Hardware auf der sie läuft. Auf einem Hardwaresystem, auf dem sequentielle Programmierung zum Einsatz kommt, befindet sich auch nur ein Hauptprozessor. Dies ist leich nachzuvollziehen, da ansonsten die Bedingung der unären Instruktion nicht mehr erfüllt wäre. Diese Einschränkung gilt auch für vernetzte Computer. Damit ein Programm als sequentiell programmiert betrachtet werden kann, darf kein anderer Computer an der Lösung der Problemstellung beteiligt sein.



2.2. Physikalische Einschränkungen

Durch die beschriebenen Eingenschaften 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 Computern. Die Übertragungsgeschwindigkeit von Daten über Materiale 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 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 nehme 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 dann die Ergebnisse zu vergleichen.

- Es existieren Anwendungsanforderungen speziell auf der Ebene der grafischen Benutzeroberflächen, 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.

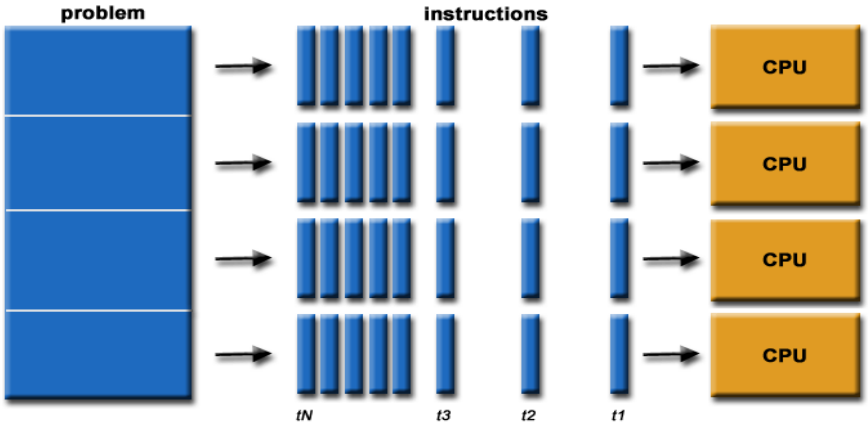
- Die sequentielle Programmierung unterliegt auch ökonomischen Zwängen. Die in 2.2 beschriebene Miniaturisierung der Chipflächen kommt den physikalen Begrenzungen immer näher. Dementsprechend steigt der Aufwand und die Kosten um die nächste Miniaturisierungsstufe zu erreichen. Dies schlägt sich demzufolge auch auf Nutzerseite zu Buche. Aus diesem Grunde kann man bei parallelisierbaren Problemstellungen auch eine größere Anzahl von günstigen Computern auf das Problem ansetzen. Dieses kann unter Umständen die gleiche oder gar 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 wiederrum dann wie in der sequentiellen Programmierung als Folge von Einzelschritten beschrieben, die man dann parallel auf mehreren Hauptprozessoren

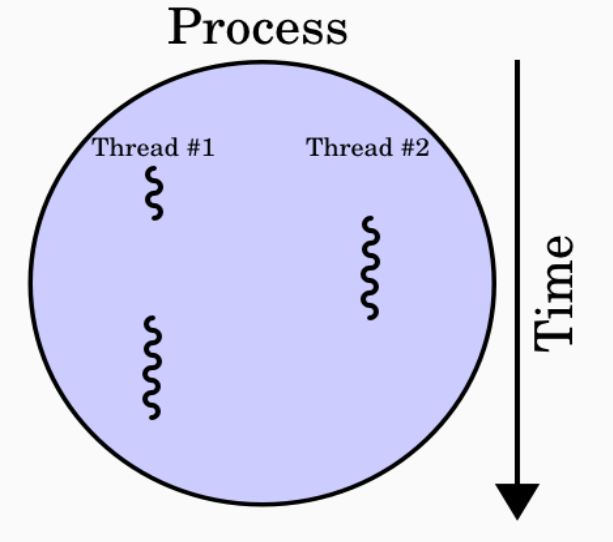
ausführen kann. Deren Einzelergebnisse zusammen erlauben eine Aussage zur gegebenen Problemstellung zu treffen.



3.2. Threads

Eine gängige Methode Parallelität in einem Programm erreichen sind die 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. Hierbei ist jedoch zu beachten, dass der Begriff des Prozesses auch Verwendung als Container für eine Vielzahl von Threads, die in einem Programm erzeugt werden verwendet werden, benutzt wird. 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 the Threadprogrammierung mit einfließen lassen. Dies kann dann zu einer besseren Leistung des Programmes führen. Auf Betriebsystemebene kümmert sich das jeweilige Betriebsystem um die einzelnen Threads innerhalb eines Prozesses was dem Entwickler Implementierungsarbeit abnimmt.

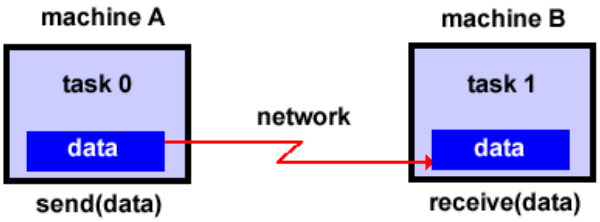
Threads gelten im Allgemeinen als „leichtgewichtig“. Damit wird üblicherweise zum Ausdruck gebracht, dass Threads im Vergleich zu Prozessen günstiger (im Sinne von Prozessorzyklen) erzeugt und wieder zerstört werden können. Zudem teilen sich Threads Daten wie statische Variablen. Echtes Multi-threading ist aber nur auf einem Rechner mit mehreren CPUs oder Prozessorkernen möglich.



3.3 Verteilte Prozesse

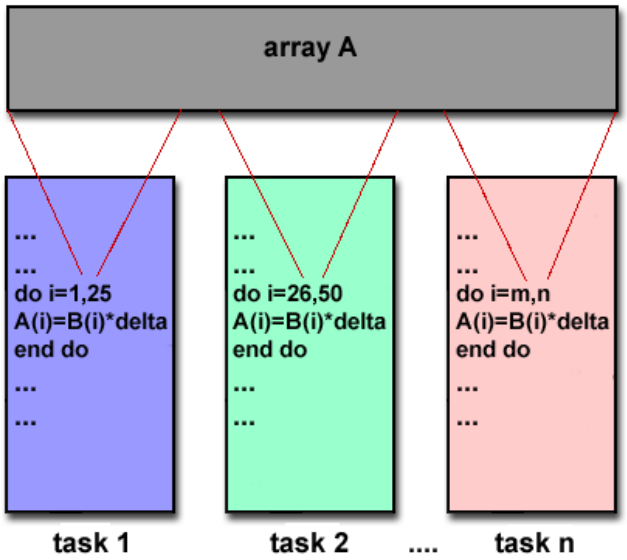
Das Arbeiten mit Threads findet lokal auf einem Computer statt. Falls man aber mehrere Rechner zum parallelen Lösen eines Problems benutzen will stösst man mit Multi-threading an die Grenzen dieser Technologie. Threads können nämlich nicht beliebig auf vernetzen Computern verteilt werden. Die Erweiterbarkeit einer einzigen physischen Recheneinheit ist aber begrenzt. Ist man an diese Leistungsgrenze gestossen und ist aber trotztdem auf mehr Rechenleistung angewiesen so muss man verteilte Prozesse als Programmiermodell ins Auge fassen. Bei verteilten Prozesson arbeiten 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 und die restlichen Knoten steuert und die Ergebnisse entgegennimmt. Die Kommunikation zwischen den Knoten geschieht dabei aus der Sicht des einzelnen Prozesses transparent. Ein Prozess ruft eine Bibliotheksmethode zum Versenden oder Empfangen von Daten bzw. Nachrichten auf. Diese Bibliothek implementiert die nötigen Datentransformationen und Verbindungsverwaltung, die für eine netzgestützte Zusammenarbeit von mehreren Computern benötigt werden.

Dabei gibt es zwei zu unterscheidende Programmierparadigmen. Das erste ist das „Message Passing“. Dabei arbeitet jeder Prozess, der an der verteilten beteiligt ist, nur mit seinem lokalen Speicher. Wenn mehrere Prozesse miteinander kommunizieren wollen müssen sie sich dem schon beschriebenen Senden und Empfangen von Nachrichten bedienen. Dabei ist zu beachten, dass dies eine kooperierende Form der Zusammenarbeit zwischen Prozessen darstellt. Zu jedem Aufruf zum Senden einer Nachricht muss eine entsprechend Funktion zum Emfang existieren. Für diese Art der Kooperation ist aber nicht zwingend notwendig, dass die Prozesse sich auf verschiedenen Computern befinden. Die Prozesse können sich alle auf einem Computer befinden.



Das zweite

Programmierparadigma wird durch das Data Parallel Model beschrieben. Hierbei existiert eine zentrale Datenstruktur auf die alle Prozesse zugreifen können. Diese Datenstruktur enthält die für die Problemlösung benötigten Daten und der Hauptanteil der parallelen Berechnung findet auf diesen Daten statt. Die Datenstruktur ist üblicherweise entweder als ein Array oder Würfel organisiert. Obwohl alle Prozesse ich diese Datenstruktur teilen arbeitet jeder Prozess alleine auf einem ihm zugewiesen Teil dieser. 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 Abstraktion 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.



3.4 Unterstützende Frameworks

Die Entwicklung paralleler Algorithmen wird meistens manuell vorgenommen. Dabei wird der Quellcode im Einzelnen durchgearbeitet und auf potentiell parallelisierbare Stellen überprüft. Somit ist der Grad der erreichten Parallelisierung stark abhänig 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 nur durch iterative Vorgehensweisen erreicht werden kann. Zudem ist es nicht auszuschließen, dass der Code sehr komplex und dadurch fehleranfällig bei Änderungen und generell schwer wartbar ist.

Um diesen Probleme abzumildern wurden Möglichkeiten entwickelt, mit denen man anhand von bekannten Mustern im Quellcode die Parellelisierung automatisieren kann. Dazu gehören insbesondere Compiler, die fähig sind existenten 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 durch Mustererkennung auf Kandidaten für die Parallelisierung untersucht. Dann 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 zu geben. Diese Vorgehensweise schließt die automatische Parallelisierung nicht aus, sondern kann mit dieser zusätzlich ergänzt werden.

4. Herausforderungen der parellen Programmierung

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

4.1 Grenzen der unterstützenden Frameworks

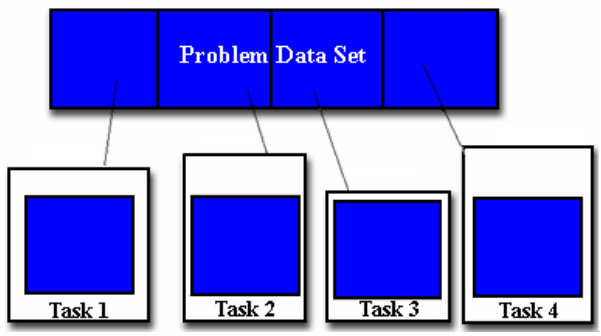
Obwohl eine automatische Parallelisierung von bereits vorhandenem sequentiellem Code in Fällen von Zeit-, und Wissensmangel eine akzeptable Alternative zur manuellen Überarbeitung darstellt, so sollte man doch mögliche Probleme nicht ausser Acht lassen.

Die heurestische Vorgehensweise bei der automatischen Parallelisierung kann auch zu fehlerhaften Ergebnissen führen. Dies äussert sich eventuell in Leistungseinbußen anstatt Zugewinn und kann sogar zu fehlerhaften Code führen. Zudem kann eine Heurestik nicht alle Eigenheiten eines Problems erkennen womit nicht gewährleistet werden kann, dass die optimale Parallelisierung erreicht wird. Bei zu komplexem Code kann die automatische Parellelisierung zudem komplett nicht zum Einsatz kommen, da die eingesetzten heurestischen Verfahren aufgrund der Komplexität parallelisierbare Stellen im Quellcode nicht erkennen.

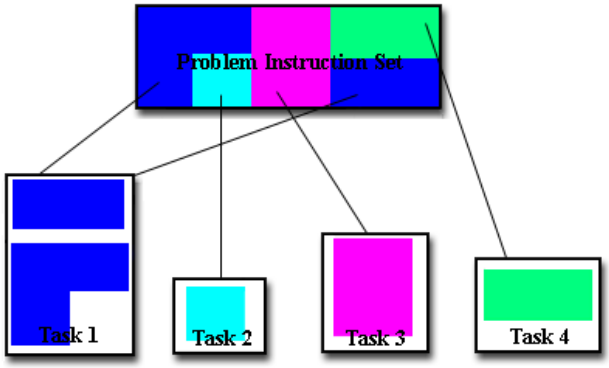
4.2 Partitionierung

Ist eine Problemstellung definiert so muss diese am Anfang darauf untersucht werden, ob und wie dieses Problem so aufgeteilt werden kann. Eine Aufteilung in diskrete Stücke ist die Grundvorausetzung, 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 nennt sich *Domain Decomposition*. Hierbei wird versucht die Parallelität über die Daten des zu bearbeitenden Problemes zu erreichen. Die Daten werden dazu in kleinere Pakete aufgebrochen. Diese können dann parallel bearbeitet und ausgewertet werden.



Der zweite Ansatz einen parallelen Algorithmus zu entwickeln lautet *Functional Decomposition.* Bei diesem Ansatz wird wird weniger auf die Daten der gestellten Problemstellung und mehr auf die Schritte zur Berechnung dieses gelegt. Jeder Prozess bzw. Thread arbeitet dann einen Teil dieser Instruktionen ab.



4.3 Wettlaufsituation

Ein weiterer Umstand den ein Entwickler beachten muss sind die sogenannten Wettlaufsituationen (engl.: Race Condition). Der Begriff beschreibt den Umstand wenn die Lösung einer parallelen Berechnung abhängig von der Reihenfolge einzelner Anweisungen in Threads bzw. verteilten Prozessen ist. Bei betriebssystemgesteuerten Threads ist es aus der Sicht des Programmieres nicht möglich im Vorraus zu wissen, welche Threads in welcher Reihenfolge aktiviert werden. Beim Vorhandensein von schreibenden Operationen auf den von Threads geteilten Daten kann es dementsprechend zu verschiedenen Ergebnissen kommen, je nachdem 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 Eingenschaften des Netzwerkes. Im ersten Fall kann es z.B. zu einer Verzögerung des Versands bzw. des Empfangs kommen, wenn andere Prozesse bereits die Netzwerkschnittstelle zugewiesen bekommen haben. Im zweiten Fall sind kurzzeitige Störungen bzw. Überlastungen des Netzes denkbar, die eine Weiterleigung einer Nachricht unmöglich machen oder verlangsamen. In beiden Fällen müssen Synchronisationsmechanismen zum Einsatz kommen, die eine logische Reihenfolge in der verteilten Berechnung ermöglichen.

4.4 Deadlocks

Threads und Prozesse können sich gegenseitig blockieren. Dies passiert dann, wenn jeweils ein Thread A bzw. Prozess eine Ressource X für sich in Anspruch genommen hat und somit diese für andere Threads nicht zugänglich ist. Gleichzeitig reserviert ein Thread B die Ressource Y. Der Deadlock entsteht dann, wenn beide Threads jeweils die Ressource des anderen Threads benötigen um ihre Berechnungen fortzusetzen. Für einen Thread ist dieser Deadlock nicht ersichtlich, da er den Grund für die Verzögerung nicht kennt. Es kann aber durch Timeouts und durch Algorithmen wie die Breitensuche Abhängigkeitsbeziehungen abgebildet werden. Diese werden dann auf potentielle Deadlocks untersucht.

Fazit

Wie gezeigt wurde, bietet parallele Programmierung die Möglichkeit weit effizienter Probleme der realen Welt zu lösen, als es dies mit sequentieller Programmierung auf Einprozessorsystem möglich wäre. Dies ist jedoch unter bestimmten Umständen mit einem komplexeren Programmiermodell verbunden. Frameworks sind in der Lage Parallelisierung mit Heurestiken ohne oder nur mit geringem Eingreifen des Entwicklers vorzunehmen. Aufgrund der Möglichkeit, dass diese Heurestiken aber 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 Verständniss der Problemstellung diese Möglichkeiten zur Parallelisierung zu untersuchen. Dies ist jedoch unter Umständen wie sehr großem Codeumfang oder komplexer Aufgabenstellung langwierig und fehleranfällig. Zudem bestehen durch die Parallelität die Möglichkeit Wettlaufsituationen und Deadlocks ins Programm einzuführen. Die Vermeidung bzw. Erkennung solcher ist nur mit zusätzlichen Algorithmen möglich. Dies wiederum erhöht den Aufwand bei der Entwicklung.