Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Der Bedarf nach immer höheren Rechenleistungen in kommerziellen und wissenschaftlichen Anwendungen hat in den letzten Jahren zu einer immer stärkeren Verbreitung von parallelen Systemen geführt. Gerade auch im Bereich des wissenschaftlichen Hochleistungsrechnens verdrängen derartige MIMD-Parallelrechner immer mehr die bisher eingesetzten dedizierten Hochleistungsrechner, wie Vektor- oder Feldrechner. Als Grund dafür kann zum einen der günstigere Preis dieser Systeme, aber auch die einfachere Verfügbarkeit üblicher Software gesehen werden, da diese Parallelrechner meist aus handelsüblichen Prozessoren aufgebaut werden.

Ein analoger Wandel mußte entsprechend auch in der Entwicklung der von diesen Systemen verarbeiteten Software erfolgen. Anders als in früheren Systemen, die Vektor- und Matrixoperationen, wie sie bei wissenschaftlichen Berechnungen fast zwangsläufig auftreten, direkt verarbeiten konnten, mußte jetzt eine effiziente Parallelisierung dieser Aufgaben gefunden werden.

1.1 Problemstellung

Bei der Parallelisierung rechenintensiver Anwendungen stellte sich schon bald heraus, daß für diese Aufteilung das klassische Prozeßkonzept, wie es aus UNIX bekannt ist nicht ausreichte. Zu kostenintensiv waren Prozeßumschaltung und Koordination, als daß damit eine feinere Parallelisierung möglich gewesen wäre.

Als erster Versuch, diese Kosten zu reduzieren, wurde das Konzept des Prozesses als Abstraktion sowohl für die Ablaufumgebung, als auch für die tatsächlich darin erfolgende Berechnung zugunsten zweier getrennter Konzepte aufgegeben. Das Betriebssystem MACH ([Bla90b], [Bla90a]) unterscheidet hier zum Beispiel den Begriff der *Task* für die Ablaufumgebung, sowie den der *Threads* zur Modellierung der innerhalb einer Task enthaltenen Aktivitätsträger. In einer Task sind mehrere Threads lauffähig, die sich deren Adreßraum und Verwaltungsstrukturen teilen. Taskinterne Threadumschaltungen können aufgrund wegfallender Adreßraumumschaltungen weitaus effizienter implementiert werden.

Obwohl dadurch die Kosten für Threadwechsel um eine Größenordnung gesenkt werden konnten, stellten sie für feingranulare Parallelisierung noch keine geeignete Grundlage dar. Erst die Verlagerung der Threadverwaltung aus dem Kern heraus auf die Benutzerebene konnte hinreichend effiziente Mechanismen zur Verfügung stellen, die einer feineren Parallelisierung nicht mehr im Wege stehen. Die eigentliche Threadverwaltung wird dabei von

Threadbibliotheken übernommen, die zu Anwendungen gebunden werden und diesen geeignete Funktionen zur Verfügung stellen. Im Gegensatz zu den Mechanismen von Betriebssystemen kann dabei unter Umständen unter mehreren Bibliotheken ausgewählt werden, von denen jede auf spezielle Anforderungen hin optimiert wurde. So kann etwa in vielen Fällen auf das Konzept von Prioritäten zwischen Threads verzichtet werden, wodurch sich die Threadverwaltung stark vereinfacht und damit auch in ihrer Effizienz steigt.

Die Entwicklung dieser Bibliotheken zur Verwaltung von Threads auf Benutzerebene orientierte sich an den Strukturen, wie man sie bisher in Betriebssystemkernen zur Verwaltung von Prozessen eingesetzt hatte. Gerade bei modernen Architekturen mit schnellen RISC-Prozessoren und mehrstufigen Speicherhierarchien, wie man sie in NUMA-Architekturen [1] antrifft hat sich aber gezeigt, daß die Überlegungen, die zur Entwicklung dieser Strukturen geführt haben, auf diese Architekturen nur noch bedingt anwendbar sind. So stand bei älteren Systemen ein möglichst einfacher und effektiver Lastausgleich im Vordergrund, um größtmögliche Auslastung der Maschinen zu erreichen. Migrationen von Berechnungen und den dafür notwendigen Daten wurden selbst unter Zuhilfenahme aufwendiger Kopierarbeiten in Kauf genommen, um alle Prozessoren zu beschäftigen.

Trotz des Einsatzes hardwareunterstützter Kommunikationsmechanismen, wie z.B. in der Convex SPP, herrscht in modernen Systemen ein Ungleichgewicht zwischen den Kommunikations- und den eigentlich notwendigen Rechenkosten. Werden Prozesse oder Threads aufgrund von Lastausgleichsüberlegungen verlagert, so kann die dazu notwendige Verlagerung der Daten den gewünschten Effekt zunichte machen. Statt einer Beschleunigung der Berechnung durch die Ausnutzung freier Rechenkapazität kann eine Verlangsamung, teilweise auch ein weiter ansteigendes Lastungleichgewicht beobachtet werden. Messungen durch Markatos ([Mar93]) haben z.B. gezeigt, daß Anwendungen, die Datenlokalität ignorieren nur sehr wenige Prozessoren effektiv ausnutzen können. Die Affinität von Prozessen oder Threads zu Prozessoren muß also als Kriterium bei Schedulingentscheidungen beachtet werden.

1.2 Verwandte Arbeiten

Die Notwendigkeit, Lokalität in Schedulingentscheidungen und Überlegungen für den Lastausgleich einfließen zu lassen, hat in den vergangenen Jahren zu einer Reihe von Veröffentlichungen geführt. Viele davon beschäftigen sich aber hauptsächlich mit dem Problem in welcher Weise die vorhandenen Prozessoren für die Berechnung der Anwendungen verwendet werden sollen. Lokalitätsbetrachtungen beeinflussen hier die Entscheidung für statisches oder dynamisches *Spacesharing*, *Timesharing* oder andere Aufteilungsverfahren.

Vaswani und Zahorjan bescheinigen in [VZ91] Affinitätsbetrachtungen einen nur geringen Einfluß auf die Berechnungsdauer verschiedener Testanwendungen. Sie verwenden aber als Grundlage für ihre Messungen ein *Spacesharing* Verfahren, das bereits eine sehr hohe Cacheausnutzung im Vergleich zu anderen Verfahren garantiert. Zusätzliche Affinitätsbetrachtungen werden erst bei einer – nur selten auftretenden – Änderung der Prozessorzuteilung verwendet. Ausgehend von nur geringen Verlusten durch das Ignorieren von Cachezuständen auf den damaligen UMA-Architekturen ergibt sich damit auch nur ein geringer Gewinn durch die erfolgten Affinitätsbetrachtungen. Jedoch stellen die Autoren durch Annahmen über die Weiterentwicklung von Mikroprozessoren die wachsende Bedeutung dieser Überlegungen heraus.

[1] **N**on **U**niform **M**emory **A**ccess

Auch Andrew Tucker kommt in [Tuc93] und [TTG95] zu dem Resultat, daß Affinity Scheduling nur Verbesserungen von wenigen Prozent gegenüber bisherigen Methoden erreicht. Er benutzte aber für seine Messungen eine UMA-Architektur mit nur wenigen Prozessoren. Trotz des geringen Gewinns schließt er aber, daß eine Integration von Lokalitätsbetrachtungen gerechtfertigt ist, da nur geringe Modifikationen im Betriebssystem z.B. an der Berechnung von Prozeßprioritäten notwendig sind. Seine Überlegungen über die Grundlagen der durch Mißachtung von Lokalität entstehenden Kosten geben einen Hinweis darauf, daß deren Einfluß bei feingranulareren Parallelisierungen zunimmt.

Wie wichtig die Beachtung von Datenlokalität sein kann stellt Markatos unter anderem in [Mar93] und [ML93] anhand von Messungen auf Architekturen unterschiedlicher Generationen dar. Er kommt zu dem Schluß, daß neue Überlegungen und Programmiermodelle notwendig sein werden, um nicht unverhältnismäßig hohe Effizienzeinbußen in Kauf nehmen zu müssen. Daß auf Lokalitätsbetrachtungen basierende Entscheidungen auch auf Kosten des Lastausgleichs Vorteile bringen, belegen Markatos und LeBlanc in [ML91].

Squillante und Lazowska verwenden in ihrer Untersuchung über den Einfluß von Lokalität beim Scheduling ein einfaches Warteschlangenmodell, parametrisiert mit den Kosten für das erneute Laden von Daten in Caches([SL89]). Als Ergebnis halten sie fest, daß bereits sehr wenig Lokalitätsinformation ausreicht, um durch geeignete Mechanismen den Durchsatz erheblich zu steigern.

Den eher umgekehrten Weg gehen Hamidzadeh und Lilja in [HL94]. Sie opfern eine Prozessor, der sich nur noch mit der Berechnung optimaler Schedules beschäftigt und den anderen Prozessoren Threads zuordnet. Obwohl dieser Ansatz noch bessere Ergebnisse zeigt, als die der oben erwähnten Untersuchungen, muß seine allgemeine Anwendbarkeit angezweifelt werden. Nicht nur die in kleineren Systemen nicht zu vertretende Aufgabe eines Prozessors für reine Verwaltungsaufgaben, sondern vor allem der für feingranulare Prozesse immens ansteigende Rechen- und Kommunikationsaufwand dürften die Verwendbarkeit dieser Methode auf rechenintensive wissenschaftliche Anwendungen beschränken, bei denen der Verlust eines Prozessors im Vergleich zur verbleibenden Zahl kaum ins Gewicht fällt.

Einen Schritt weg von Betriebssystemmechanismen hin zu Threads auf Benutzerebene machen Fowler und Kontothanassis in ihrem Artikel über *Object-Affinity Scheduling* ([FL92]). Neben Überlegungen zu Affinity Scheduling verwenden sie sogenannte *Continuations*, um Affinität von einem Thread auf einen Nachfolgerthread zu übertragen.

1.3 Übersicht über die vorliegende Arbeit

Nach einer Motivation für den Einsatz von Threadbibliotheken und einer Einführung in ihre Grundlagen, werden in Kapitel 2, anhand eines Beispiels für eine moderne NUMA-Architektur, Gründe für die Notwendigkeit der Beachtung von Datenlokalität veranschaulicht. Kapitel 3 beschäftigt sich mit einer Art der Threadverwaltung, die man in der Literatur unter der Bezeichnung *Memory Conscious Scheduling* findet und die Datenlokalität ausnutzt, um die Anzahl an Cachemisses bzw. deren mittlere Latenzzeit zu verringern. Neben Hinweisen zur Gewinnung der notwendigen Information werden hier hauptsächlich die dazu notwendigen Datenstrukturen und Algorithmen vorgestellt. Zusätzlich dazu werden in Kapitel 4 weitere Überlegungen angestellt, inwieweit Informationen über den Cachezustand einzelner Threads gewonnen und nutzbringend für Schedulingentscheidungen herangezogen werden können. Nach einer Betrachtung von Algorithmen für die bei Nebenläufigkeit unabdingbaren

Koordinierungsmechanismen, die darauf zielen, die entstehenden Kommunikationskosten zu reduzieren wird in Kapitel 6 eine Implementierung einer Threadbibliothek vorgestellt. Diese verwendet einige der in den vorangehenden Kapiteln besprochenen Mechanismen, weshalb auch vergleichende Tests für diese Mechanismen auch erst nach Vorstellung der Bibliothek erfolgen. Kapitel 7 faßt die Erkenntnisse der vorangegangenen Kapitel zusammen und gibt einen Ausblick auf mögliche Erweiterungen, die sich vor allem im Rahmen erweiterter Hardwareunterstützung neuerer Rechnergenerationen ergeben. Abgeschlossen wird die Arbeit durch eine kurze Zusammenstellung aller wichtigen Befehle der entwickelten Threadbibliothek.

Kapitel 2

Threadbibliotheken und Lokalität

Threadbibliotheken setzen sich insbesonders bei der feingranularen Parallelisierung von Anwendung immer mehr durch. Dieses Kapitel geht auf die Gründe dieser wachsenden Bedeutung und den prinzipiellen Aufbau derartiger Bibliotheken ein. Nach der Vorstellung eines modernen Parallelrechners mit NUMA-Architektur wird anschließend auf die Notwendigkeit eingegangen, Lokalitätsinformation für die Entscheidungen bei der Threadverwaltung heranzuziehen.

2.1 Begriffsbestimmung

Die Begriffe leicht- oder schwergewichtige Prozesse, Threads usw. werden in der Literatur immer wieder mit unterschiedlichen Bedeutungen belegt. Deshalb soll hier eine, für diese Arbeit verbindliche Benennung vorangestellt werden:

- **Anwendungen** sind diejenigen Einheiten, die sich dem Benutzer präsentieren. Ihr interner Aufbau kann aus einem oder mehreren Prozessen oder Threads bestehen.

- **Tasks** stellen Ablaufumgebungen dar, die aus einem Adreßraum, Zugriffsrechten, Dateideskriptoren usw. bestehen.

- **Kernelthreads** repräsentieren die eigentlichen Aktivitätsträger, die innerhalb einer Task arbeiten. Sie teilen sich alle durch eine Task zur Verfügung gestellten Resourcen. Das Konzept der Kernelthreads kennen Betriebssysteme z.B. unter den Bezeichnungen Threads (MACH) oder leichtgewichtige Prozesse (SunOS 5.4).

- **Prozesse** stehen für Tasks mit einem einzigen Kernelthread. Damit entsprechen sie den von UNIX bekannten Prozessen.

- **Threads** schließlich sind diejenigen Aktivitätsträger, die im Benutzeradreßraum durch Bibliotheken verwaltet werden. Sie werden in der Literatur auch als Userlevel Threads oder aber ebenfalls als leichtgewichtige Prozesse bezeichnet.

2.2 Gründe für die Bedeutung von Threadbibliotheken

2.2.1 Das Threadprogrammiermodell

Einer der wohl wichtigsten Gründe für die Entwicklung von Threadbibliotheken war der Wunsch feingranulare Parallelität in Programmen einfach und effizient ausdrücken zu können. Ähnlich der Entdeckung, daß sich viele Probleme durch Zerlegung in Objekte in einer weitaus verständlicheren Form strukturieren lassen als etwa durch prozedurale Techniken, hat man entdeckt, daß viele Anwendungen eine inhärente Parallelität aufweisen, die sich in ganz natürlicher Weise durch eine Menge nebenläufiger Aktivitäten modellieren läßt.

War Parallelität auf der Ebene von Prozessen eher dazu geeignet Programme so in Teile zu zerlegen, daß sie einfach auf die darunterliegende, parallele Hardware abgebildet werden konnten, so stellten erst Kernelthreads, wie sie etwa durch MACH angeboten werden, eine Möglichkeit zur Verfügung, Programme feiner zu parallelisieren. Die Forderung nach immer feinerer Parallelität, wie sie z.B. in objektorientierten Systemen mit aktiven Objekten, bei asynchronen Methodenaufrufen oder in eventbasierten Systemen auftreten, konnte aber erst befriedigt werden, nachdem man die Verwaltung der Aktivitätsträger aus dem Betriebssystemkern heraus in den Benutzeradreßraum verlagerte. Da damit Einsprünge in den Betriebssystemkern wegfielen, konnten die hauptsächlichen Operationen einer Threadverwaltung wie Erzeugen und Vernichten von Threads, bzw. die Threadumschaltung weiter optimiert werden. Effiziente Implementierungen erreichen Threadwechselzeiten, die etwa eine Größenordnung über den Zeiten für Prozeduraufrufe liegen.

Da Threads einer Anwendung, ähnlich wie Kernelthreads, innerhalb einer gemeinsamen Task laufen, können sie auf alle Daten direkt zugreifen. Die Notwendigkeit adreßraumüberbrückender Kommunikationsmechanismen, wie z.B. das aus UNIX-Systemen bekannte Shared-Memory zwischen Prozessen, entfällt und dadurch auch die damit verbundene aufwendige und fehleranfällige Programmierung. Neben dem Speicher teilen sich Threads damit auch alle anderen Ressourcen, die eine Task zur Verfügung stellt, wie z.B. Deskriptoren geöffneter Dateien.

Ebenso wie die Threadverwaltung profitierten auch die, bei nebenläufigen Systemen obligatorischen Koordinierungsmechanismen von der Auslagerung auf Benutzerebene und dem gemeinsamen Adreßraum. Anstelle teurer Systemeinsprünge werden hier billige Funktionsaufrufe verwendet – vorausgesetzt die verwendete Architektur kennt einen atomaren `test_and_set` Befehl, mit dem sich im Benutzermodus Koordinierungsmechanismen realisieren lassen.

Neben der Parallelisierung durch den Programmierer in Form expliziter Generierung von Threads, spielt immer mehr die automatische Parallelisierung durch Compiler eine Rolle. Vor allem die Aufteilung von Schleifenkonstrukten auf einzelne Prozessoren bietet hier eine einfache Möglichkeit der Automatisierung. Üblicherweise erzeugen derartige parallelisierende Compiler je einen Kernelthread für jeden vorhandenen Prozessor. Bei der Parallelisierung von Schleifen werden die einzelnen Schleifendurchläufe aufgeteilt und in Form entsprechender Arbeitsaufträge auf diese Kernelthreads verteilt. Eine Vielzahl von Algorithmen wurde entwickelt, um etwaigen Lastungleichgewichten entgegenzuwirken (siehe auch [Mar93]). Threadbibliotheken bieten jetzt die Gelegenheit für jeden Schleifendurchlauf einen eigenen Thread zu erzeugen. Dadurch wird Bearbeitung und Lastausgleich durch die Threadverwaltung übernommen, ein getrenntes Konzept für Schleifenparallelisierungen entfällt. Untersuchungen in [Mar93] zeigen, daß vor allem unter Berücksichtigung von Lokalitätsinformation damit weitaus bessere Ergebnisse erreicht werden können als mit bisher entwickelten Algorithmen.

2.2.2 Vorteile bei der Benutzung auf Parallelrechnern

Nicht nur für die Realisierung von Anwendungen, sondern auch für die Verwendung spezieller Mechanismen zur Verwaltung von Kernelthreads im Betriebssystemkern können Threadpakete von Vorteil sein.

Bisher wurden Parallelrechner hauptsächlich für sehr rechenintensive wissenschaftliche Aufgaben eingesetzt. In diesem Einsatzgebiet hat es sich eingebürgert, die Maschinen einer Anwendung exklusiv zur Verfügung zu stellen, um Verluste durch unnötige Umschaltungen zwischen Anwendungen zu vermeiden. Batchsysteme sorgen dafür, daß nach Abarbeitung der aktuellen Anwendung die nächste gestartet wird. In letzter Zeit haben Parallelrechner aber auch Verwendung in interaktiven Systemen gefunden. Batchsysteme können selbstverständlich die dafür notwendigen Antwortzeiten nicht liefern. Meist wurde deshalb ein Zeitscheibenverfahren verwendet, um Anwendungen Prozessoren zuzuteilen. Diese Zeitscheibenverfahren besitzen aber Nachteile, die sie für Anwendungen, die tatsächlich die parallele Rechenleistung mehrerer Prozessoren benötigen kaum geeignet erscheinen lassen:

1. Die Berechnungen werden nach kurzer Zeit abgebrochen und die Prozessoren anderen Anwendungen zugeordnet. Die Daten, die die unterbrochene Anwendung in die Prozessorcaches geladen hatte, gehen dabei regelmäßig verloren und müssen unter großem Zeitaufwand während der nächsten Zeitscheibe neu vom Speicher geholt werden.

2. Wird ein Kernelthread während eines koordinierten Zugriffs, d.h. zum Beispiel mit einem gesetzten Mutexlock verdrängt da seine Zeitscheibe abgelaufen ist, so werden dadurch unter Umständen andere Threads blockiert. Je nach Implementierung warten diese aktiv auf die Freigabe des Locks oder geben den Rest ihrer Zeitscheibe auf.

3. Wird nur ein Teil der Kernelthreads einer Anwendung berechnet, so kann es aufgrund des Kommunikationsverhaltens dieser Threads zu einem vorzeitigen Abbrechen ihrer Berechnung kommen. Threads blockieren sich und geben damit ihre Zeitscheibe auf, da ihre Kommunikationspartner nicht zur Verfügung stehen. Dadurch entsteht ein steigender Aufwand in Form von überflüssigen Threadwechseln.

Alle angesprochenen Probleme lassen sich nur durch Veränderungen und Ergänzungen zum einfachen Timeslicing beseitigen, bzw. mildern, wie z.B. durch Verlängerung der effektiven Zeitscheiben (zu 1.), Kommunikation zwischen der Anwendung und dem Betriebssystemkern zur Vermeidung von Verdrängungen innerhalb kritischer Abschnitte (zu 2.) und Co- oder Gangscheduling für die gemeinsame Berechnung kommunizierender Threads.

Eine weitaus bessere Ausnutzung von Prozessorcaches und TLB-Einträgen erreicht aber ein anderer Ansatz. Beim Spacesharing werden die vorhandenen Prozessoren fest auf die im System laufenden Anwendungen aufgeteilt ([MVZ93]), d.h. es treten keine unnötigen Threadwechsel auf, die Caches, TLB-Einträge o.ä. zerstören. Zwei Arten dieser Aufteilung werden dabei unterschieden:

1. Static Spacesharing: Eine Anwendung meldet beim Start die Anzahl gewünschter Prozessoren beim System und bekommt entsprechend Prozessoren zugeteilt. Diese Anzahl bleibt während der Laufzeit der Anwendung konstant. Treffen neue Anwendungen im System ein, so müssen diese entweder warten bis hinreichend viele Prozessoren zur Verfügung stehen oder es werden laufenden Anwendungen Prozessoren entzogen, wobei

die Kernelthreads der Anwendung dann in geeigneter Weise von den restlichen Prozessoren bearbeitet werden müssen. Im allgemeinen wird hierzu wieder ein Zeitscheibenverfahren verwendet, mit den schon erwähnten Nachteilen.

2. Dynamic Spacesharing: Hier werden Anwendungen je nach Last Prozessoren zugeteilt, bzw. wieder entzogen. Die Anwendung wird dafür im allgemeinen durch den Betriebssystemkern über derartige Entscheidungen informiert und muß darauf reagieren. Umgekehrt kann eine Anwendung dem Kern mitteilen, daß sie bereit wäre Prozessoren freizugeben, bzw. mehr Prozessoren benötigen würde. Im Gegensatz zum Static Spacesharing wird hier von der Anwendung die Anzahl verwendeter Kernelthreads verändert. Die verbleibenden Kernelthreads einer Anwendung bleiben dadurch weiterhin an ihren Prozessor gebunden. Threadwechsel und die damit verbundenen Verluste werden dadurch vermieden.

Sowohl Simulationen als auch Messungen an erfolgten Implementierungen haben gezeigt, daß Dynamic Spacesharing den anderen Methoden überlegen ist. Voraussetzung für die Anwendung dieser Methode sind aber Anwendungen, die darauf vorbereitet sind mit einer sich ändernden Anzahl an Prozessoren zurecht zu kommen und mit dem Betriebssystemkern über geeignete Wege zu kommunizieren.

Als besonders geeignet für die Implementierung dieser dynamischen Methode hat sich ein zweistufiges Schedulingkonzept erwiesen. Anwendungen bekommen Kernelthreads vom Betriebssystemkern als virtuelle Prozessoren zur Verfügung gestellt. Diese dienen dann dazu, die Threads der Anwendung zu bearbeiten. Dieses Konzept bietet mehrere Vorteile:

- Die Anwendung wird unabhängig von der Anzahl virtueller Prozessoren, d.h. Dynamic Spacesharing kann einfacher realisiert werden. Die Threads der Anwendung können jederzeit auf einer sich verändernden Prozessorzahl berechnet werden. Damit wird die Anwendung aber auch unabhängig von der tatsächlichen Anzahl im System vorhandener Prozessoren und damit flexibler beim Wechsel der Architektur.

- Aufgrund der zu erwartenden kürzeren Lebensdauer von Threads ergeben sich weitaus häufiger Gelegenheiten einer Anwendung sicher virtuelle Prozessoren zu entziehen. So kann etwa bei der Terminierung eines Threads davon ausgegangen werden, daß dieser keine Mutexlocks mehr gesetzt hat. Ein Entziehen des virtuellen Prozessors kann also an diesem Punkt nicht zu einem Verdrängen eines Threads innerhalb eines kritischen Abschnittes, mit den oben beschriebenen Nachteilen, führen.

- Durch die feinere Parallelisierung verringert sich das Problem eines durch Prozessorentzug entstehenden Lastungleichgewichts. Beispielhaft sei hier eine Anwendung genannt, die für die Verwendung eines einstufigen Schedulingmechanismus auf einem Rechner mit 10 Prozessoren in 10 Teile aufgeteilt wurde. Stehen dem Programm aber nur 9 Prozessoren tatsächlich zur Verfügung, so erhöht sich die effektive Rechenzeit des Programmes um 100%. Zerlegt man dasselbe Problem in einem zweistufigen Schedulingkonzept z.B. in 50 Threads, so benötigen die 9 virtuellen Prozessoren nur um 20% mehr Zeit als 10 Prozessoren.

Abbildung 2.1 zeigt diese zweischichtige Aufteilung. Die Prozessoren (1) der zugrundeliegenden Hardware werden in Form eines Spacesharing-Verfahrens durch den Kernelscheduler (2) mehreren Kernelthreads (3) zugeordnet. Das Laufzeitsystem einer Threadbibliothek (4) benutzt diese virtuellen Prozessoren um die Threads (5) der eigentlichen Anwendung in geeigneter Weise zu bearbeiten. Neben diesen *multithreaded* Anwendungen können aber weiterhin jederzeit klassische Prozesse existieren, wie etwa Anwendung 3.

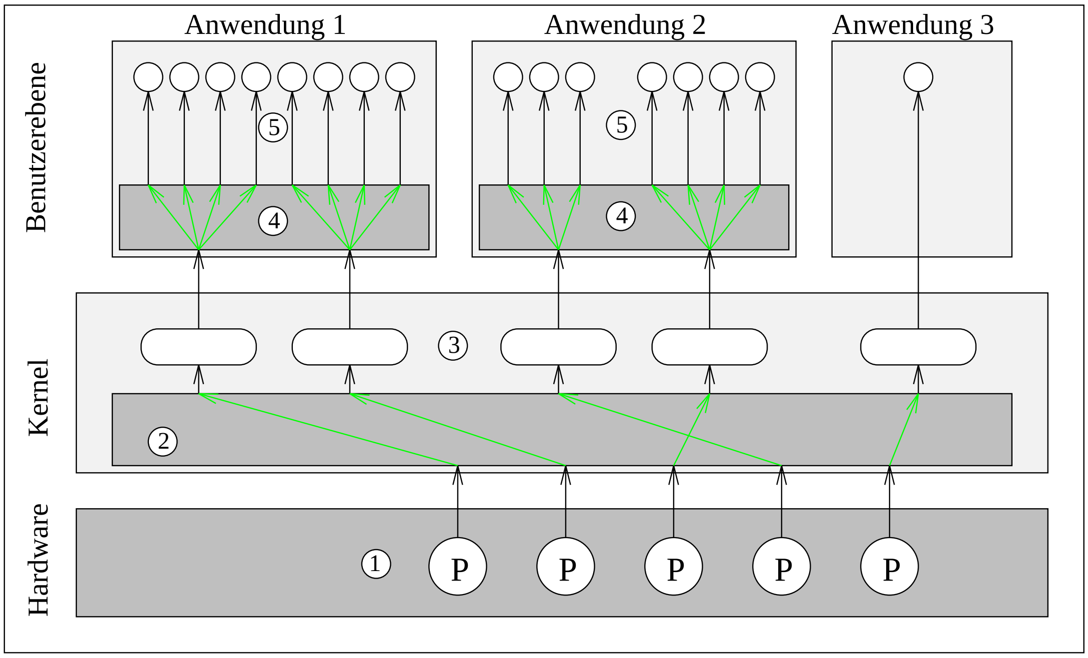


Abbildung 2.1: Schema eines zweistufigen Schedulingkonzepts

Verschiedene Veröffentlichungen beschäftigten sich mit den Möglichkeiten für die Implementierung einer derartigen Strategie. Tucker beschreibt in [Tuc93] eine Realisierung mittels einiger neuer Systemfunktionen, mit denen die Kommunikation zwischen der Anwendung und dem Betriebssystemkern erfolgt. Sein *Process Control Approach* erlaubt dynamische Prozessorallokation unter Berücksichtigung der Affinität, die Threads zu Prozessoren aufgebaut haben. Sein Hauptaugenmerk liegt dabei auf einer möglichst guten Auslastung der vorhandenen Prozessoren.

Alternativ dazu kann die Zuteilung von Prozessoren auch durch einen Benutzerprozeß – einen sogenannten *Processor Server* – erfolgen ([Bla90a], [Bel95a]). Damit bestünde auch die Gelegenheit, mehrere verschiedene Strategien – für verschiedene Klassen von Anwendungen – zur Verfügung zu stellen. So finden bei dem in [Bel95a] vorgeschlagenen Server Lokalitätsüberlegungen, wie sie vor allem bei der Verwendung von NUMA-Architekturen notwendig sind, stärkere Beachtung

2.3 Typische Aufgaben und Strukturen von Threadbibliotheken

Vergleicht man den Funktionsumfang typischer Threadbibliotheken, wie z.B. den **Solaris-Threads** aus SunOS 5.4, den **CThreads** für Mach ([DC88]) oder den **PThreads** ([Inc94], [Hau95]) entsprechend dem Standard von *POSIX 1003.4c*, so findet man in allen folgende grundlegenden Mechanismen:

- Funktionen zur Kontrolle und Konfiguration interner Mechanismen der Bibliothek. Dazu gehört im allgemeinen auch eine Funktion zum Start und zur Initialisierung des Pakets.
- Funktionen zur Verwaltung von Threads:
  - Erzeugen neuer Threads
  - Suspendierung von Threads
  - Abgeben des Prozessors zugunsten eines anderen Threads
  - Terminieren des aufrufenden Threads
  - Warten auf die Terminierung eines Threads
- Koordinierungsmechanismen: Als typische Koordinierungsmechanismen enthalten Bibliotheken meist Mutexlocks und Bedingungsvariablen. Teilweise werden aber auch weitere Mechanismen, wie Semaphoren, Barrieren oder Leser/Schreiber-Locks angeboten.

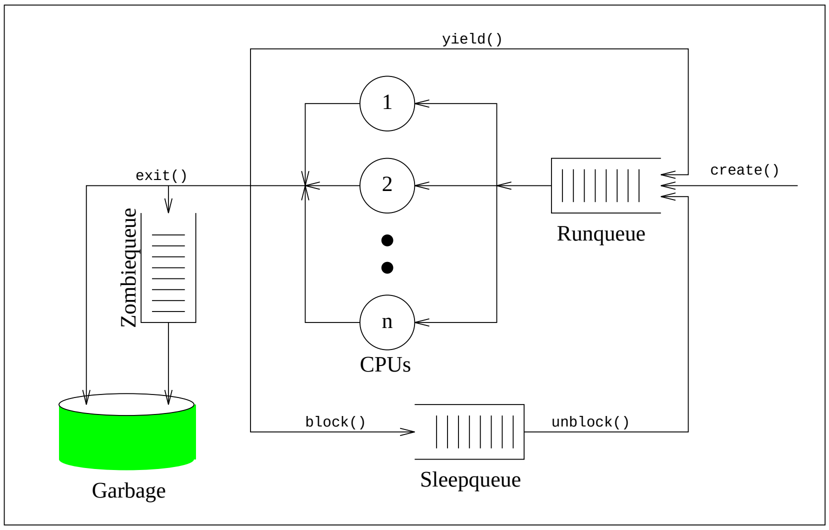


Abbildung 2.2: Struktur einer einfachen Threadbibliothek

Abbildung 2.2 zeigt den typischen Aufbau einer einfachen Threadbibliothek. Mittels der Bibliotheksfunktion `create()` erzeugte neue Threads werden in eine zentrale Runqueue übernommen. Diese Warteschlange enthält alle lauffähigen Threads und garantiert damit daß kein Prozessor unbeschäftigt wartet, solange noch lauffähige Threads existieren. Die vorhandenen CPUs, bzw. virtuellen Prozessoren entnehmen im gegenseitigen Ausschluß Threads aus dieser Runqueue und führen sie aus, bis eine der folgenden drei Fälle eintritt:

1. Der Thread gibt von sich aus den Prozessor frei und ordnet sich wieder in die Runqueue ein (`yield()`).

2. Der Thread blockiert sich an einem Koordinierungsmechanismus (`block()`) und wird in eine zentrale Sleepqueue eingetragen. Dort verbleibt er, bis er durch einen anderen Thread wieder deblockiert wird (`unblock()`).

3. Der Thread beendet sich mittels `exit()`. Abhängig davon, ob es sich um einen freien (*detached*) Thread handelt oder nicht, wandert er entweder in eine sogenannte Zombiequeue oder wird sofort gelöscht.

2.4 Aufbau eines aktuellen NUMA-Rechners

Um zu verstehen, warum ein Threadpaket mit dem eben vorgestellten, einfachen Aufbau kaum geeignet ist, um auf modernen NUMA-Architekturen bei feingranularer Parallelisierung den Rechner voll auszunutzen, muß man sich die Eigenschaften derartiger Rechner ansehen.

Bei dem in den folgenden Kapiteln, für eine prototypische Implementierung eines Threadpakets und die damit erfolgten Messungen, verwendeten Parallelrechner handelt es sich um eine Convex SPP 1000. Abbildung 2.3 zeigt den Aufbau dieses Rechners.

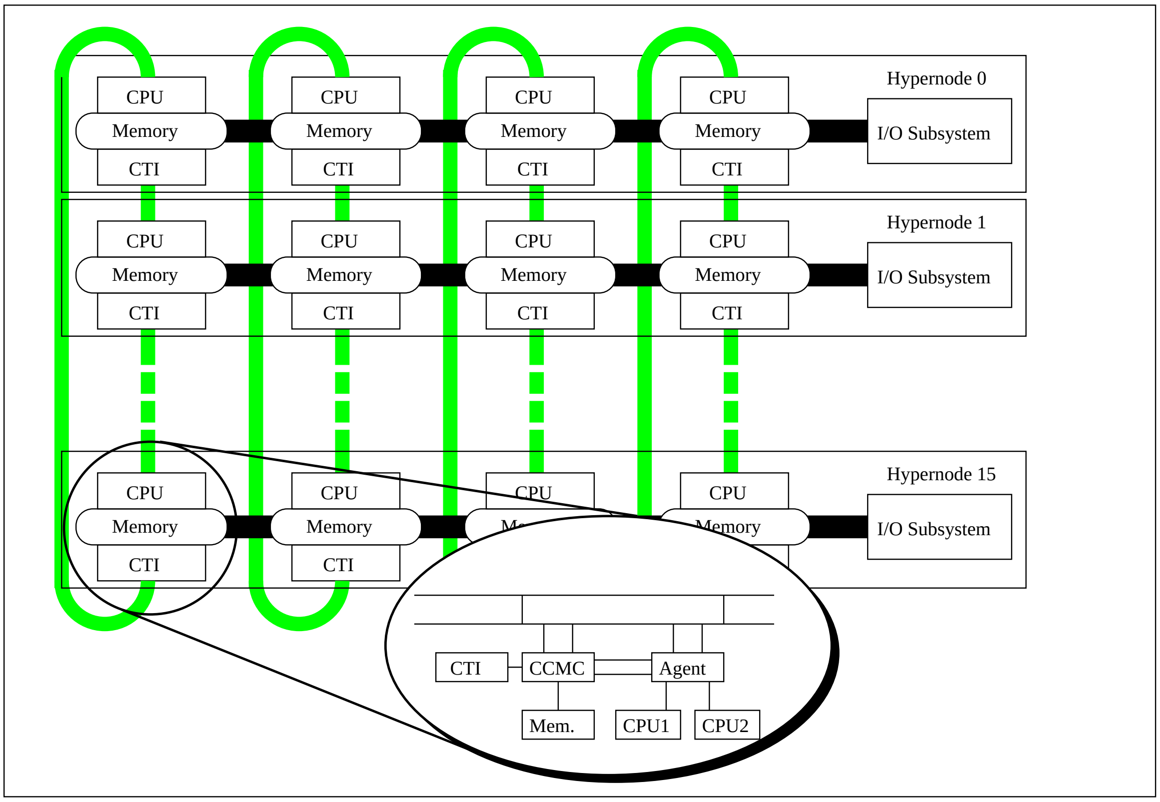
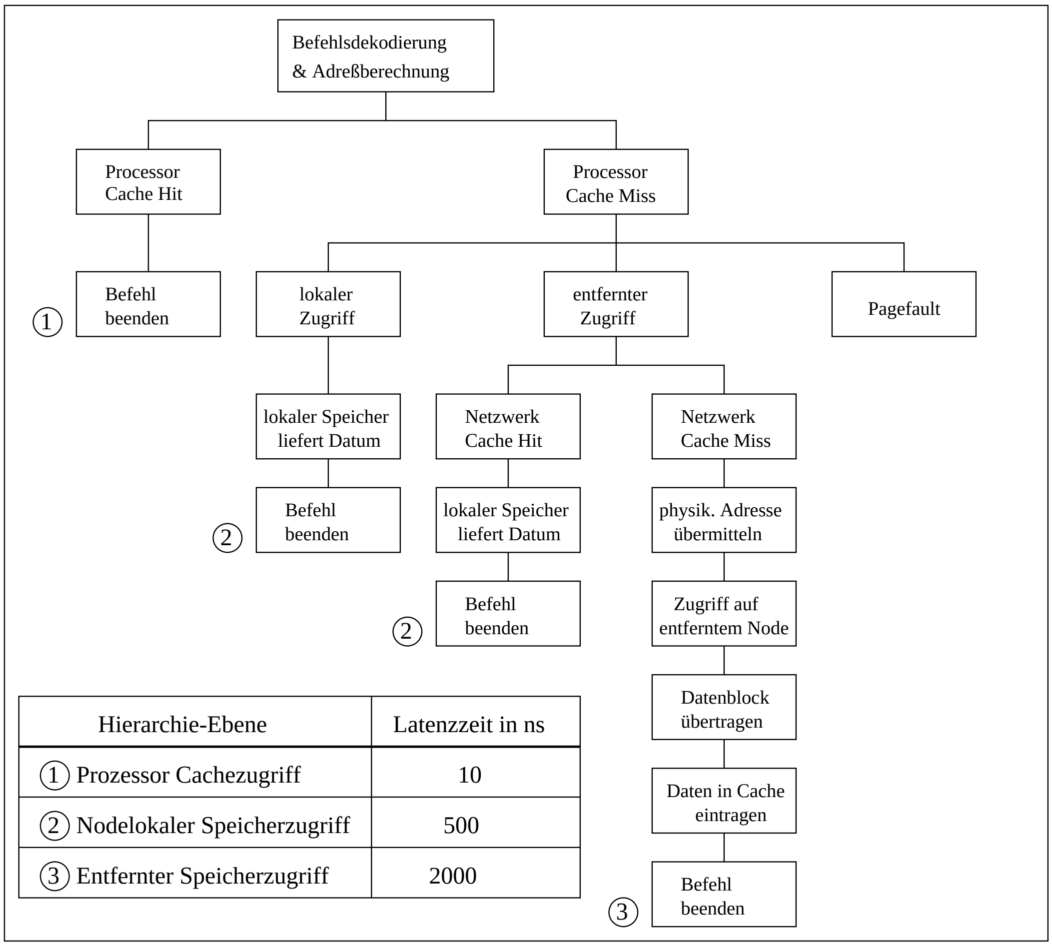


Abbildung 2.3: Der Aufbau des Convex SPP Multiprozessors

Sogenannte Hypernodes bilden die Basiseinheiten einer Convex SPP. Dabei handelt es sich im Prinzip um symmetrische Multiprozessorsysteme bestehend aus je vier CPU-Blöcken. Jeder CPU-Block wiederum besteht aus zwei HP PA-RISC Prozessoren mit jeweils 1 MByte virtuell adressiertem Daten- und Instruktionscache, bis zu 512 MByte lokalem Speicher, einer Schnittstelle zu CPU-Blöcken anderer Hypernodes, dem CTI[1] und einem Zugang zu einem Kreuzschienenverteiler. Über diesen 5x5 Kreuzschienenverteiler kann jeder CPU-Block auf den Speicher der anderen CPU-Blöcke seines Hypernodes und das I/O-Interface zugreifen. Der Zugriff erfolgt dabei nichtblockierend, mit geringer Latenzzeit und hoher Bandbreite. Speicherzugriffe auf den gemeinsamen Speicher innerhalb eines Hypernodes erfolgen immer mit derselben Zugriffszeit, unabhängig davon, ob das lokale Speichermodul oder ein fremdes angewählt wird.

Die Hypernodes das Rechners werden über vier CTI-Ringe verbunden. Das CTI stellt eine Erweiterung des Schnittstellenstandards SCI (Scalable Coherent Interface) nach IEEE P1212 dar. Mit einer Nettorate von ca. 600 MByte/s wird über jeden dieser Ringe der Zugriff auf den Speicher anderer Hypernodes realisiert. Um die mittlere Zugriffszeit darauf zu verringern wird ein Teil des lokalen Speichers eines jeden Hypernodes als CTI-Cache(Netzwerkcache) verwendet.



Hierarchie-Ebene	Latenzzeit in ns
1 Prozessor Cachezugriff	10
2 Nodelokaler Speicherzugriff	500
3 Entfernter Speicherzugriff	2000

Abbildung 2.4: Zugriffspfade auf die Speicherhierarchie

[1] **C**onvex **T**orodial **I**nterface

Der so entstandene Rechner läßt sich in mehrere logische Subkomplexe zerlegen. Jeder Subkomplex stellt einen Parallelrechner mit globalem, gemeinsamen Speicher dar, auf den über 48 Bit Adressen zugegriffen wird. Abbildung 2.4 zeigt die verschiedenen Möglichkeiten beim Zugriff auf diesen Speicher, je nachdem welche Ebene der Speicherhierarchie durch einen Befehl angesprochen wird. Die Latenzzeiten unterscheiden sich dabei erheblich, von 10 ns für einen Cachehit im Prozessorcache, über 500ns für einen lokalen Speicherzugriff, bis hin zu 2000 ns falls das Datum von einem anderen Hypernode geholt werden muß. Aufgrund der verschiedenen Caches – Prozessorcache und Netzwerkcache – gelten diese Speicherzugriffszeiten jeweils nur für das erste Datum innerhalb einer Cachezeile, deren Länge beim Prozessorcache 32 Byte und beim Netzwerkcache 64 Byte beträgt.

Vergleicht man diese Zeiten mit der Zykluszeit des verwendeten Prozessors von 10 ns, so erkennt man den Einfluß, den die Beachtung der Datenlokalität bei einer derartigen Architektur besitzt. Um die Rechenleistung der Prozessoren effektiv nutzbar zu machen, müssen Algorithmen, und damit auch Threadbibliotheken versuchen möglichst hohen Nutzen aus den verschiedenen Caches und der Kenntnis der Speicherhierarchie zu ziehen. Eine Threadbibliothek mit einem Aufbau wie in Abbildung 2.2 gezeigt, kann diese Aufgabe kaum erfüllen:

- Bereits der konkurrierende Zugriff auf die zentralen Strukturen sorgt dafür, daß regelmäßig Cachezeilen invalidiert und somit später wieder angefordert werden müssen. Davon betroffen ist nicht nur der Prozessor, der den Cachemiss ausgelöst hat, sondern auch alle anderen Prozessoren, die auf den exklusiven Zugriff auf diese Struktur warten.

- Zwar sorgt die gemeinsame Runqueue für vollständigen Lastausgleich, jedoch auch dafür, daß Threads nach einer erfolgten Deblockierung zu einem anderen Prozessor migrieren können. Sie geben dabei alle in den Cache ihres vorherigen Prozessors geladenen Daten auf und müssen diese neu in den aktuellen Cache laden. Da Migrationen auch über Hypernodegrenzen hinweg möglich sind, betrifft dieses Neuladen nicht nur die Prozessorcaches, sondern ggf. auch die Netzwerkcaches.

- Ohne die ergänzende Einführung zusätzlicher Mechanismen, wie z.B. Prioritäten, besteht keine Möglichkeit zu spezifizieren auf welchem Hypernode ein Thread bearbeitet werden soll, um ihn damit näher an seine Daten zu bringen.

Ein Threadpaket für einen Parallelrechner, insbesonders für einen NUMA-Rechner muß damit neben den vorher bereits erwähnten Aufgaben zusätzlich noch Lokalitätsinformation sammeln und so für die Verteilung und Berechnung ausnutzen, daß die Lokalität möglichst gut erhalten bleibt. Dabei kann einerseits die Lokalität der Daten auf einem Hypernode ausgenutzt werden, um die mittleren Latenzzeiten beim Speicherzugriff zu verringern und andererseits die Lokalität bzgl. eines Caches zur Reduzierung auftretender Cachemisses.

Kapitel 3

Memory Conscious Scheduling

Während bei UMA-Architekturen auf den gemeinsamen Speicher von jedem Prozessor aus mit denselben Kosten zugegriffen werden konnte, so gilt dies für NUMA-Architekturen nicht mehr. Die Maschinen sind hierarchisch strukturiert und damit ergeben sich zwischen verschiedenen Teilen dieser Hierarchie unterschiedliche Zugriffszeiten.

Betrachtet man die Speicherhierarchie einer Convex SPP, so kann man für den Hauptspeicher zwei verschiedene Stufen erkennen:

1. Lokaler Speicher auf dem Hypernode [1]: Greift ein Prozessor auf eine Speicheradresse zu, die nicht in seinem Datencache liegt und holt er eine Cachezeile von 32 Byte vom lokalen Hauptspeicher, so dauert es ca. 500 ns, bis er auf das erste Speicherwort dieser Cachezeile zugreifen kann.

2. Speicher auf einem anderen Hypernode: Liegt ein Speicherwort weder im Prozessorcache noch im lokalen Speicher oder im Netzwerkcache, so werden die Daten für eine Netzwerk-Cachezeile von einem anderen Hypernode über das CTI geholt. Trotz einer vollständigen Implementierung in Hardware besitzt die Übertragung einer Cachezeile von 64 Byte eine Latenzzeit von 2000 ns. Erst dann kann der Prozessor wieder auf das erste Speicherwort zugreifen.

Um den Einfluß dieser im Vergleich zur Prozessorzykluszeit von 10 ns sehr großen Latenzzeiten zu verringern, wurden mehrere Caches in das System eingebaut. Neben den festen Daten- und Instruktionscaches kann ein Teil des lokalen Speichers eines Hypernodes als Netzwerkcache konfiguriert werden. Zugriffe auf dieses Netzwerkcache zeigen damit dieselbe Latenzzeit wie Zugriffe auf den restlichen lokalen Speicher von ca. 500 ns. Obwohl die Caches die mittlere Zykluszeit im Zugriff auf die verschiedenen Speicherklassen reduzieren, bleibt trotzdem ein erheblicher Unterschied zwischen lokalem und entferntem Speicher bestehen.

Dominierender Faktor beim Zugriff auf den Hauptspeicher bleibt trotz Einsatz von Caches die hohe Latenzzeit, bis die ersten Daten nach einem Cachemiss verfügbar sind. Verschiedene Konzepte wurden entwickelt, um Prozessoren während dieser Latenzzeit sinnvoll zu beschäftigen. So existieren Architekturen, deren Prozessoren nicht nur einen sogenannten *Instruction Stream* bearbeiten können, sondern mehrere. Beispiele dafür sind die Architekturen HEP, MASE, Horizon oder Tera([AAC+92]). Treten in diesen Systemen Verzögerungen auf, wie

[1] Im weiteren wird die Bezeichnung Hypernode allgemein für Substrukturen in NUMA-Architekturen beibehalten, soll also nicht nur auf die Convex SPP beschränkt sein.

etwa beim Zugriff auf den langsameren Hauptspeicher, dann können sie intern auf einen anderen Instruction Stream, mit anderen Worten einen anderen Thread umschalten und damit quasi ohne Verzögerung weiterrechnen. Zusätzlich unterstützt die Tera dann auch noch bis zu acht ausstehende Speicheroperationen, d.h. diese Speicherzugriffe können gestartet werden, ohne die Berechnung zu blockieren. Entsprechende Compiler vorausgesetzt können diese Operationen dazu dienen höhere Latenzzeiten zu tolerieren, ohne daß ein Threadwechsel notwendig wird.

Moderne, konventionelle Prozessoren kennen ein ähnliches Konzept. Nichtblockierende *Prefetch-* und *Poststore-*Operationen erlauben es, parallel zur Berechnung Daten vom oder zum Hauptspeicher zu transportieren. Wie bei den obigen Prozessoren ist man auch hier wieder darauf angewiesen, daß die Befehle einige Zeit vor dem eigentlichen Zugriff initiiert werden, damit die Hardware die Gelegenheit hat, den Transfer abzuschließen. Nur dann kann ein Prozessor bei einem Lesezugriff die Daten aus dem Cache erhalten.

Die Eigenschaften moderner NUMA-Architekturen zeigen die Notwendigkeit für ein neues Programmiermodell. Das bei UMA-Architekturen verwendete *Shared Memory* Modell unter Verwendung von Prozessen kann die Rechenleistung neuerer Rechner nicht mehr ausnutzen, da ein Großteil der Zeit für Speicherzugriffe verwendet würde. Das Threadprogrammiermodell hingegen kann helfen die Probleme zu umgehen, bzw. zu mindern. Die zur Verwaltung der Threads eingesetzten Methoden werden dabei in der Literatur häufig als *Memory Conscious Scheduling* (MCS) bezeichnet.

3.1 Effizienzsteigerung durch Threadbibliotheken

Die Parallelisierung einer Anwendung mittels Threads bietet im Vergleich zur klassischen Parallelisierung mittels Prozessen oder Kernelthreads völlig neue Möglichkeiten, Kenntnisse über die Lokalität von Daten für die Verteilung der Berechnung auszunützen.

Bei grobgranularer Parallelisierung konnte oftmals keine eindeutige Entscheidung für einen Berechnungsort gefunden werden, da die notwendigen Daten etwa gleichmäßig auf verschiedene Knoten verteilt waren. Durch Threads wird es nun möglich die Berechnung so fein aufzuteilen, daß sie am Ort ihrer hauptsächlich benutzten Daten erfolgen kann.

Da Threads nur relativ kleine Zustandsinformation in Form von Threadkontrollblöcken und kleinen Stacks benötigen, sind sie mit weitaus geringeren Kosten zwischen Hypernodes migrierbar. Sie können damit bei einer Veränderung ihres Zugriffsverhaltens einfach an den Ort ihrer Berechnung gebracht werden. Bisher wurde durch Betriebssystemmechanismen und/oder Hardware versucht, Daten mit Hilfe von Caches näher an den Ort der Berechnung zu bringen. Durch Einführung von Threads kann nun der umgekehrte Weg beschritten werden: ein Thread, d.h. die eigentliche Berechnung wandert an den Ort der Daten, bzw. wird direkt dort erzeugt.

Notwendig für dieses Konzept ist eine zweite Eigenschaft, die Threads – zumindest auf einer Convex SPP – von Prozessen und Kernelthreads unterscheidet. Der komplette Zustand von Threads besteht nur aus Daten, die im Benutzeradreßraum liegen. Da diese Daten aber für jeden Prozessor des Systems erreichbar sind, können Threads beliebig zwischen Prozessoren eines Hypernodes, aber auch verschiedener Hypernodes wandern. Im Gegensatz dazu liegen bei Kernelthreads oder Prozessen Verwaltungsstrukturen auch im Adreßraum des Betriebssystems. Sind diese Strukturen nicht allgemein zugreifbar, dann wird dadurch auch die Migrationsfähigkeit eingeschränkt.

Eine zusätzliche Erhöhung der effektiv nutzbaren Rechenleistung kann durch eine bessere Cachenutzung entstehen. Obwohl längerlaufende Prozesse grobgranularer Parallelisierungen im allgemeinen besseren Nutzen aus Caches ziehen sollten, da ihre Daten nicht regelmäßig durch andere Threads verdrängt werden, kann bei einzelnen Anwendungen der umgekehrte Effekt eintreten. Im Gegensatz zu den grobgranularen Prozessen kann durch Threads eine feingranulare Parallelisierung erfolgen, bei der jeder Thread nur wenig Daten im Cache benötigt und dadurch auch nur wenig Daten anderer Threads verdrängt. Dadurch kann die Summe aller Cachemisses im Vergleich zur grobgranularen Parallelisierung abnehmen, da dort unter Umständen jeder Prozeß alle Daten anderer Prozesse aus dem Cache verdrängt.

3.1.1 Threadverteilung anhand von Lokalitätsinformation

Um einen Nutzen aus der Kenntnis über den Ort einzelner Speicherbereiche ziehen zu können, muß der Benutzer einer Threadbibliothek, sei es Programmierer oder ein parallelisierender Compiler, Threads virtuellen Prozessoren oder Hypernodes zuordnen können. Inwieweit diese Verteilung für das Threadpaket bindend ist, kann z.B. durch eine Bewertung festgelegt werden – von freier Migrierbarkeit bis hin zu fest zu beachtenden Bindungen.

Durch diese Zuordnung von Threads zu Prozessoren oder Hypernodes wird zweierlei erreicht. Einerseits wird dafür gesorgt, daß Hauptspeicherzugriffe vor allem auf den lokalen Speicher des Hypernodes, nicht aber auf andere Hypernodes erfolgen. Die Zeit, bis die Arbeitsmenge eines Threads in den Cache geladen ist reduziert sich dadurch, da jeder Cachemiss weniger Kosten aufwirft. Andererseits macht sich ein Vorteil bei Threads bemerkbar, die dieselben Speicherbereiche für ihre Berechnung benötigen. Durch eine Bindung an ein und denselben Prozessor steigt die Wahrscheinlichkeit, daß sie bei ihrem Start bereits Daten im Cache vorfinden, die einer ihrer Vorgängerthreads dort hinterlassen hat. Somit verringert sich für einen neuen Thread die Anzahl der notwendigen Cachemisses zum Aufbau seiner Arbeitsmenge im Cache.

Für eine solche Verteilung benötigt der Benutzer des Pakets Information über die Struktur der verwendeten Hardware, wie z.B. die Anzahl von Prozessoren und Hypernodes und Kenntnisse darüber, wo vom Betriebssystem angeforderter Speicherplatz tatsächlich liegt. So kennt etwa die Convex SPP unter anderem die folgenden Speicherklassen:

- NODE_PRIVATE: Dieser Speicher liegt immer auf dem aktuellen Hypernode und kann nur von Prozessoren dieses Hypernodes angesprochen werden.
- NEAR_SHARED: Auch dieser Speicher liegt immer auf dem aktuellen Hypernode, jedoch kann er auch von allen anderen Hypernodes angesprochen werden.
- FAR_SHARED: Diese Speicherklasse steht ebenfalls dem Zugriff über Hypernodegrenzen offen, wobei aber der physikalisch benutzte Speicher in Form einzelner Seiten auf alle Hypernodes des Subkomplex verteilt wird.
- BLOCK_SHARED: Wie bei FAR_SHARED, wird hier der Speicher auf alle Hypernodes verteilt, jedoch nicht in Form einzelner Seiten, sondern durch eine Aufteilung in gleichgroße Stücke für jeden Hypernode.

Die Umsetzung der Daten über Speicherklasse und Speicheradresse in den eigentlichen Ort und die damit entstehenden Erkenntnisse über den besten Berechnungsort können entweder während der Programmentwicklung durch Programmierer bzw. Compiler, oder aber auch während der Laufzeit durch das Laufzeitsystem erfolgen.

Statische Threadzuordnung zur Übersetzungszeit

Kennt ein Programmierer oder Compiler die Speicherbelegungsstrategie des Betriebssystems dann kann er zumindest für statische Daten deren tatsächlichen Ort voraussagen. Auch für dynamisch angeforderte Daten kann eine solche Vorhersage möglich sein, wenn z.B. der Ort des anfordernden Threads bekannt ist. Die eigentliche Berechnung wird nun so auf Threads aufgeteilt und auf die einzelnen Prozessoren verteilt, daß jeder Thread den größten Teil seiner Daten lokal vorfindet.

Besonders bei der automatischen Parallelisierung von Schleifenkonstrukten, die auf großen globalen Datenmengen arbeiten, hat sich diese Art der statischen, da direkt in das Programm codierten Zuordnung bewährt ([Mar93]). Während hier die Abbildung durch den Compiler erfolgt, muß sie bei manueller Parallelisierung vom Programmierer übernommen werden. Die Komplexität dieser Umsetzung läßt sich z.B. beim Einsatz von FAR_SHARED Memory und einer unter Umständen nicht trivialen Abbildung logischer Nodestrukturen auf die darunterliegende Hardware erahnen. Darüberhinaus wäre diese Berechnung der Datenlokalität abhängig von der verwendeten Hardware, wie etwa der Anzahl der Hypernodes.

Dynamische Threadzuordnung zur Laufzeit

Anstatt die Zuordnung von Threads zu Prozessoren oder Hypernodes während der Programmierphase vorzunehmen, kann diese auch zur Laufzeit durch die Threadbibliothek erfolgen. Grundlage dieser Verteilung ist wieder die Kenntnis über die von jedem Thread verwendeten Daten und deren Ort. Soweit der Ort nicht in einfacher Weise direkt, z.B. aus der Adresse eines Speicherbereichs, berechnet werden kann, muß das Betriebssystem dafür geeignete Funktionen bereitstellen. Die Information über die eigentlich verwendeten Speicherbereiche muß jedoch weiterhin vom Benutzer zur Verfügung gestellt werden.

Handelt es sich um bereits bestehende Datenbereiche, so kann diese Information beim Erzeugen eines Threads mitgegeben werden. Das Laufzeitsystem nutzt dann diese Information, um den besten Prozessor für diesen neuen Thread auszuwählen. Da das Laufzeitsystem zu diesem Zeitpunkt auch Kenntnisse über die verwendete Hardware hat, kann es auf Unterschiede zwischen Rechnern, wie etwa in der Anzahl verwendeter Hypernodes reagieren. Vorteilhaft ist diese Zuordnung vor dem Start vor allem bei Threads, die während einer einmaligen, relativ kurzen Berechnungsphase hauptsächlich auf diesen Speicherbereich zugreifen.

Sobald die Information über Speicherbereiche nicht mehr beim Start eines Threads übergeben werden kann, weil sie z.B. erst durch den Thread selbst berechnet wird, muß der Thread unabhängig davon gestartet werden. Sobald er dann seine Speicherbereiche dem Laufzeitsystem mitteilt, wird dieses darüber entscheiden, ob der Thread migriert werden soll. Durch diese Migration ergeben sich Kosten, die durch den Gewinn lokaleren Datenzugriffs aufgehoben werden müssen. Außerdem ermöglicht es diese Methode automatische Verlagerungen von Threads vorzunehmen, sobald sich ihr Zugriffsverhalten ändert. Dieses Konzept kommt somit hauptsächlich für Threads in Betracht, die viel Speicher referenzieren, während Threads mit wenig Speicherzugriffen, insbesonders solche, die innerhalb eines Berechnungsabschnittes komplett berechnet werden kaum von einer Migration profitieren sollten.

Eine weitere Automatisierung, wie die automatische Ermittlung der von einem Thread referenzierten Speicherbereiche dürfte wohl an dem dafür notwendigen Aufwand scheitern. Zwar wäre eine Auswertung von TLB-Einträgen ([SW95]) bei jedem Threadwechsel möglich – vorausgesetzt diese sind im Benutzermodus lesbar – um Änderungen festzustellen, jedoch würde diese Auswertung die Kosten für jeden Threadwechsel vervielfachen. Um derartige Informationen überhaupt nutzen zu können, müßten sie im Benutzermodus unter sehr geringem Zeitaufwand lesbar sein. Darüberhinaus müßte die Information entsprechend der Granularität der Threads feiner sein, als sie von heutigen Maschinen zur Verfügung gestellt wird ([Bel95b]).

3.1.2 Prefetching von Datenbereichen

Wie bereits vorher erwähnt, kann Prefetching – das nichtblockierende Laden von Daten in Caches – zur Verringerung des Einflusses der Latenzzeiten beim Zugriff auf den Hauptspeicher eingesetzt werden. Als Grundlage dafür dienen Befehle neuerer Prozessorgenerationen, die es ermöglichen Daten parallel zum eigentlichen Berechnungsvorgang zu laden. Die dazu notwendigen Kenntnisse über Lage und Länge benutzter Speicherbereiche ergibt sich durch die für die Threadzuordnung angegebenen Informationen. Dem Threadpaket wird es dadurch möglich, bereits vor dem Wechsel auf einen neuen Thread Daten in die Caches zu laden. Doch nicht nur der Zugriff auf die angegebenen Datenbereiche, sondern auch auf die intern verwendeten Strukturen, wie Threadstrukturen, Stack usw. kann damit beschleunigt werden.

Zu den mittels Prefetching angesprochenen Caches zählen dabei nicht nur die Prozessorcaches, sondern z.B. auch die Netzwerkcaches in der Convex SPP. Von der Kontrolle der Cachekohärenz hängt es ab, ob dieses Prefetching in Netzwerkcaches Einfluß auf die Prozessorcaches hat. Je nach Implementierung kann sonst die Anwendung dieses Prefetchings – wie auf der SPP 1000 – alle betreffenden Cachezeilen in den Prozessorcaches des Hypernodes invalidieren. Besser als ein getrenntes Konzept für die jeweiligen Hierarchiestufen wäre also ein einheitlicher Prefetchbefehl, der Prefetching über die gesamte Speicherhierarchie auslöst, wofür aber Unterstützung des Prozessors für derartige Hierarchien notwendig wäre. Als Beispiel einer Architektur, die Prefetching und Poststoring über die gesamte Speicherhierarchie beherrscht kann etwa die KSR1 von Kendall Square Research angesehen werden ([Hau95]).

3.1.3 Bindung von Threads untereinander

Neben dem Verhalten eines Threads spielt bei der Entscheidung für die Threadverteilung auf Hypernodes auch das Kommunikationsverhalten von Threads eine entscheidende Rolle. Basiert die Kommunikation nicht auf Nachrichten, sondern wird über den gemeinsamen Speicher abgewickelt, so sollten kommunizierende Threads auf demselben Hypernode berechnet werde. Selbst bei der Verwendung nur weniger Speicherzellen, z.B. für ein Mutexlock, kann sonst die Zykluszeit für einen Kommunikationsvorgang erheblich ansteigen.

Der Grund liegt im andauernden schreibenden Zugriff auf den Kommunikationsspeicher, der zu einer regelmäßigen Invalidierung der Caches führt. Die Anzahl der Cachemisses in den Datencaches der kommunizierenden Prozessoren kann dabei nicht verringert werden, wohl aber die in den Netzwerkcaches, die bei Kommunikation über Nodegrenzen entstehen, wodurch die mittlere Latenzzeit beim Zugriff auf diesen gemeinsam benutzten Speicher sinkt.

Die Bindung zwischen einzelnen Threads kann wieder sowohl statisch, als auch dynamisch erfolgen. Statische Bindung erfolgt, wie im vorherigen Punkt, durch die Zuweisung der Threads zu gemeinsamen Hypernodes ggf. sogar gemeinsamen virtuellen Prozessoren. Sollte ein Thread aber, etwa aufgrund notwendigen Lastausgleichs auf einen anderen Hypernode migriert worden sein, so geht diese statische Bindung verloren. Eine dynamische Bindung zwischen Threads kann dieses Problem beseitigen. Statt zu bestimmen, wo Threads bearbeitet werden sollen, werden hier direkt Beziehungen zwischen Threads hergestellt, d.h. von der Migration eines Threads wird auch der andere beeinflußt.

Ein Threadpaket hat zwei Möglichkeiten, diese Bindung zu realisieren. Entweder gibt der Benutzer durch eine Funktion an, mit welchen anderen Threads ein Thread kommuniziert und das Laufzeitsystem sorgt für eine automatische Migration oder aber der Benutzer löst diese Migration durch den Aufruf einer Funktion selbst aus. Solange bei der automatischen Verlagerung von Threads keine Rücksicht auf die Nebeneffekte, wie z.B. den Verlust anderweitiger Lokalität genommen werden kann, stellt die explizite Aufforderung durch den Benutzer wohl die bessere Lösung dar.

Obwohl dieses Konzept der Bindung von Threads untereinander dem des Coschedulings ähnelt, lassen sich Überlegungen aus diesem Gebiet nur teilweise dafür verwenden. So existieren dort z.B. Ansätze, das Kommunikationsverhalten von Prozessen automatisch zu ermitteln und für das Scheduling zu verwenden ([SW95]). Neben nachrichtenbasierten Systemen werden dort auch Shared-Memory Systeme betrachtet. Als Mittel, gemeinsame Speichernutzung für die Kommunikation zu finden, werden die TLB-Einträge der einzelnen Prozessoren ausgewertet. In Anbetracht der geringen Anzahl an Prozessen bei einer grobgranularen Parallelisierung und den damit selteneren Prozeßwechseln kann der dadurch entstehende Aufwand gerechtfertigt sein. Bei threadbasierten Systemen bereitet dabei aber einerseits die hohe Anzahl zu vergleichender Threads, als auch die, im Vergleich zu den von Threads benutzten Kommunikationsspeichergrößen, grobe Einteilung von TLB-Einträgen Probleme.

Einfachere Informationsquellen als TLB-Einträge stellen die Koordinierungsmechanismen eines Threadpakets dar. Mutexlocks zum Beispiel, wie sie für den exklusiven Zugriff u.a. auf Kommunikationsspeicher Verwendung finden, können helfen Gemeinsamkeiten zwischen Threads festzustellen. Speichert jedes Lock eine Referenz auf den Thread, der das Lock hält, bzw. zuletzt hielt, so kann diese Referenz ausgenutzt werden, um den anfragenden Thread oder den Kommunikationspartner ggf. auf denselben Hypernode zu migrieren.

3.1.4 Ausnutzung von Cacheaffinität

Während seiner Berechnung lädt ein Thread eine Menge von ihm benötigter Daten in den Datencache seines Prozessors. Die Anzahl der Cachezeilen, die von einem Thread im Datencache belegt werden stellt ein Maß für die Affinität dieses Threads zu diesem Prozessor dar.

Threads, die nach einer Blockierung oder der freiwilligen Abgabe ihres Prozessors wieder mit ihrer Berechnung fortfahren können, sollten bevorzugt demjenigen Prozessor zugeteilt werden auf dem sie vorher berechnet wurden. Dadurch können sie Reste des verbliebenen Cachezustandes nutzen und somit die mittlere Anzahl von Cachemisses reduzieren.

Ein Thread, der z.B. aufgrund von Lastausgleich zwischen verschiedenen Prozessoren migriert wurde hat mit großer Wahrscheinlichkeit Affinität zu all diesen Prozessorcaches. Kennt man die unterschiedlichen Affinitätswerte, so kann eine Entscheidung getroffen werden, ob der Thread auf seinem neuen Prozessor verbleiben oder nach einer Blockierung wieder auf einen der vorherigen zurückwandern soll. Untersuchungen in [SL89] haben gezeigt, daß es genügen würde Affinitätswerte für zwischen zwei und vier Prozessoren zu speichern. Eine Erhöhung dieser Anzahl würde kaum weitere Verbesserungen bringen, andererseits aber den Aufwand für die Methode stark erhöhen.

Den größten Nutzen aus dieser Art der Bindung haben Threads, die sich während ihrer Berechnung häufig blockieren. Aber nicht nur die Zahl erfolgter Blockierungen hat Einfluß auf den möglichen Nutzen. Wie Tucker in [Tuc93] zeigt, hat daneben auch die Länge eines Berechnungsabschnittes eine entscheidenden Einfluß. Je länger die Berechnung dauert, desto geringer fallen die Cachemisses ins Gewicht, die beim Start des Threads für das Laden seiner Arbeitsmenge ins Cache notwendig sind. Da aber maximal diese Anzahl an Cachemisses eingespart werden kann, nicht aber diejenigen für transiente Zugriffe, die während der Berechnung auftreten, ist das Potential für Verbesserungen darauf beschränkt.

3.2 Realisierung von MCS in Threadbibliotheken

Wie angedeutet, muß eine Threadverwaltung für Memory Conscious Scheduling Mechanismen anbieten, um Threads virtuellen Prozessoren oder Hypernodes zuordnen zu können und diese Wahl zu erhalten. Zwei Verfahren, um diese Zuordnung in Threadbibliotheken zu realisieren werden im Anschluß beschrieben, gefolgt von der Betrachtung weiterer für eine effiziente Implementierung notwendiger Details.

3.2.1 Prioritätsbasierte Zuordnungsverfahren

Prioritätsbasierte Verfahren entstanden aus der Überlegung heraus, MCS für die Prozeßverwaltung in einem konventionellen Betriebssystem einzusetzen, ohne allzu umfangreiche Änderungen an den bestehenden Strukturen und Algorithmen vornehmen zu müssen.

Vor allem im Bereich kleinerer Parallelrechnersysteme herrschen dort zentrale Strukturen, wie gemeinsame Run- und Sleepqueues vor. Ohne weitere Vorkehrungen würden Prozesse also in quasi zufälliger Reihenfolge von freien Prozessoren aus der Runqueue entnommen. Weder ein ursprüngliche Zuordnung, noch ein Erhalt einer Zuordnung wären somit möglich.

Einfache Threadpakete haben diese Strukturen übernommen, da sie einfach zu verwalten sind und vollständigen Lastausgleich garantieren. Während innerhalb der Prozeßverwaltung des Betriebssystems Prioritäten unter anderem für Fairness bei der Verteilung von Rechenleistung auf Prozesse sorgen, kann auf diese Aufgabe in Threadverwaltungen meist verzichtet werden – Prioritäten können damit andere Aufgaben übernehmen. An die Stelle fester Prioritäten treten Funktionen, die die Bindung eines Threads zu einem bestimmten virtuellen Prozessor oder Hypernode ausdrücken. Je enger die Bindung, desto höher die Priorität des Threads bzgl. dieses Prozessors.

Dieses Verfahren ermöglicht eine sehr feinstufige Spezifikation, wie stark eine Bindung sein soll und auf welchen Bereich – Prozessor oder Hypernode – sich diese Bindung bezieht. Der Benutzer gibt für jeden Thread an, wo er berechnet werden soll und wie stark diese Bindung sein soll. Abbildung 3.1 zeigt verschiedene Ausprägungen dieser Spezifikation bei der Anwendung auf einer Convex SPP mit 4 Hypernodes und 32 Prozessoren

- Thread 1 und 2 besitzen hohe Prioriät für einen speziellen Prozessor. Eine Migration dieses Threads wird damit soweit wie möglich unterdrückt. Diese Möglichkeit dient vor allem dazu, mehrere kommunizierende Threads fest Prozessoren eines Hypernodes zuweisen zu können.

- Thread 3 besitzt gleichhohe Priorität auf allen Prozessoren innerhalb eines Hypernodes. Er kann damit zwischen verschiedenen Prozessoren eines Hypernodes migrieren, wobei die Lokalität zum Hauptspeicher des Hypernodes erhalten bleibt, jedoch innerhalb des Hypernodes Lastausgleich möglich wird.

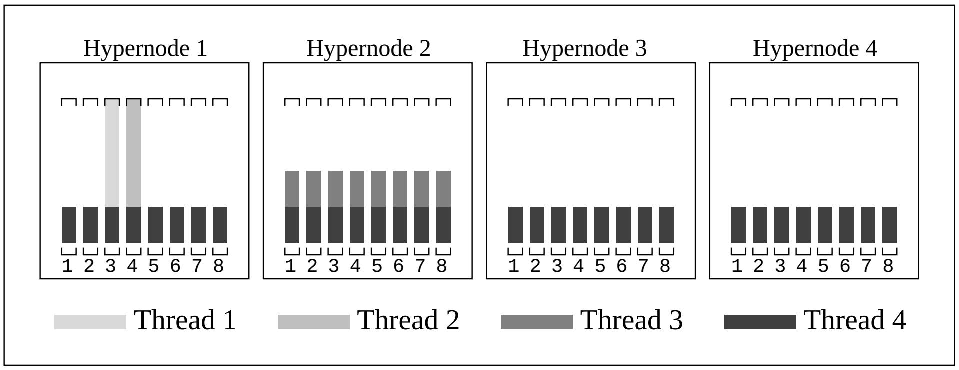


Abbildung 3.1: MCS durch Bindung mittels Prioriäten

- Thread 4 hat auf allen Hypernodes und Prozessoren dieselbe Priorität und darf deshalb beliebig innerhalb des Systems wandern. Diese Möglichkeit findet vor allem bei Threads Anwendung, die nur sehr wenig Daten benötigen, so daß ihre Migration einen nur geringen Verlust im Vergleich zum durch Lastausgleich erzielten Gewinn bedeutet.

Auch die Beachtung der Cacheaffinität von Threads läßt sich durch Prioritäten realisieren. So zeigt Tucker in [Tuc93], daß bereits eine geringe Erhöhung der Priorität eines Threads für seinen letzten Prozessor die Cacheausnutzung erheblich steigern kann. Durchsucht dabei ein Prozessor die Runqueue, so wird die Priorität aller Prozesse, die zuletzt auf diesem Prozessor bearbeitet wurden künstlich erhöht, um die Wahrscheinlichkeit für die Migration eines anderen Prozesses zu verringern. Eine zusätzliche Anhebung erfährt der letzte Prozeß dieses Prozessors, um ihn wenn möglich weiter zu bearbeiten und damit seine effektive Zeitscheibe zu verlängern. Beide Manipulationen des bestehenden Schedulingalgorithmus führen zu einer Bindung von Prozessen an Prozessoren und einer daraus resultierenden besseren Cacheausnutzung.

Durch die einfache Realisierbarkeit bietet sich diese Möglichkeit insbesonders für die ergänzende Affinitätsbetrachtung in Betriebssystemkernen mit Prioritätswarteschlangen an. Bei hohen Prozessorzahlen und in Threadbibliotheken können diese Verwaltungsmechanismen dagegen nur beschränkt eingesetzt werden, da sich zwei Probleme bei derartigen Anwendungen verschärfen.

Zum einen können die Prioritäten für alle Prozessoren unterschiedlich sein. Einfache Implementierungen für die Warteschlangen genügen deshalb nicht mehr. Aufwendige Strukturen bzw. Verkettungen werden erforderlich um einen effizienten Zugriff auf den höchstprioren Thread für jeden virtuellen Prozessor sicherzustellen. Ohne solche Strukturen sind die Prozessoren darauf angewiesen, aus der Menge verfügbarer Threads den für sie höchstprioren herauszusuchen – ein Vorgehen, das im Vergleich zum anschließenden Threadwechsel zu aufwendig ist.

Ein weiteres Problem entsteht dadurch, daß – wie bei vielen zentralen Ressourcen – die Runqueue einen Engpaß im System darstellt. In UMA-Systemen mit einer nur geringen Zahl an Prozessoren oder beim Scheduling von Prozessen in relativ großen Zeitabständen mag sich der Einfluß wenig bemerkbar machen. In Systemen mit vielen Prozessoren, insbesonders großen NUMA-Systemen wird eine zentrale Struktur schnell zum Flaschenhals. Man kann zeigen, daß bei einer genügend hohen Anzahl von Prozessoren ein Sättigungszustand im System eintritt, bei dem jede Hinzunahme weiterer Prozessoren keine Erhöhung der effektiven Rechenleistung mehr bewirkt.

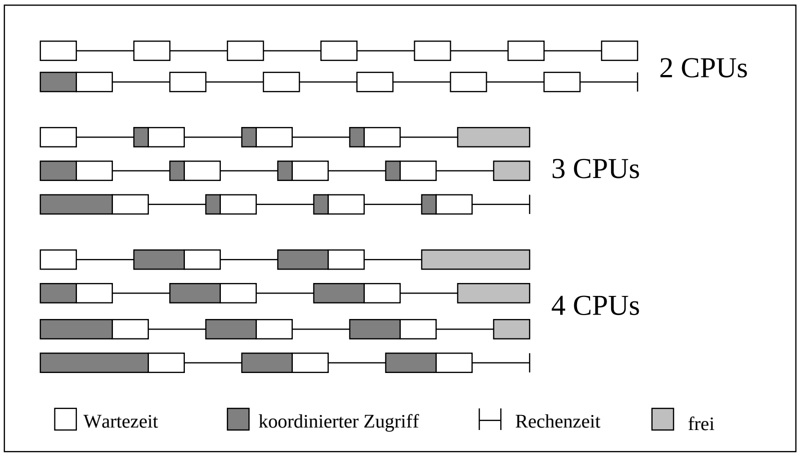


Abbildung 3.2: Wartezeiten beim Zugriff auf eine zentrale Struktur

Abbildung 3.2 zeigt diesen Effekt an einem einfachen Beispiel. Mehrere quasi identische Threads werden einer zentralen Struktur entnommen und anschließend von den Prozessoren des Systems berechnet. Während bei dem in der Abbildung gezeigten Verhältnis von Rechenzeit zur Zeit für den exklusiven Zugriff, beim Übergang von zwei auf drei Prozessoren noch eine Verkürzung der Rechenzeit eintritt, bleibt sie bei einer Erhöhung von drei auf vier Prozessoren konstant. Unter Vernachlässigung von Nebeneffekten, wie Bussättigung usw. und der Beschränkung auf Threads mit gleicher Rechenzeit, wie sie etwa bei Schleifenparallelisierungen auftreten, kann der erreichbare Speedup einer Anwendung grob abgeschätzt werden. Dabei werden die nachfolgenden Abkürzungen verwendet:

- T_k: Zeit für den koordinierten Zugriff auf die zentrale Struktur
- T_r: Rechenzeit zwischen zwei Zugriffen auf die Struktur
- N : Anzahl der Prozessoren im System,

Die Abbildung läßt bereits erkennen, daß neben den anfänglichen Wartezeiten beim Start der Anwendung keine weiteren Verzögerungen auftreten, solange gilt

$$T_r \geq T_k(N-1) \tag{3.1}$$

In diesem Fall ergibt sich für $N * n$ Threads eine Gesamtrechenzeit bis zum Ende des letzten Threads von

$$T_{G_{\geq}} = n(T_r + T_k) + (N-1)T_k \tag{3.2}$$

Sobald die Rechenzeit der einzelnen Threads unter die vorher angegebenen Schranke sinkt, müssen die Prozessoren bei jedem koordinierten Zugriff auf die Struktur warten. Die Gesamtrechenzeit wächst um diese Wartezeit auf

$$\begin{aligned} T_{G_{<}} &= n(T_r + T_k) + (N-1)T_k + (n-1)((N-1)T_k - T_r) \\ &= T_r + n * N * T_k \end{aligned} \tag{3.3}$$

Im Vergleich zur Rechenzeit auf einem Prozessor von

$$T_1 = nN(T_r + T_k) \tag{3.4}$$

sind die nachfolgenden Speedupwerte erreichbar, wobei die Abkürzung $V = T_k/(T_r + T_k)$ verwendet wurde. Für kleine Prozessorzahlen gilt

$$S_{\geq} = \frac{N}{1 + \frac{N-1}{n} * V} \tag{3.5}$$

Sobald aber die Prozessorzahl soweit erhöht wird, daß $T_r < (N-1) * T_k$ gilt, tritt der besagte Sättigungszustand ein. Für den Speedup ergibt sich dann:

$$S_{<} = \frac{N}{\frac{1-V}{n} + N * V} \tag{3.6}$$

Für große n hängt der erreichbare Speedup im Sättigungsfall nur von dem Verhältnis V aber nicht mehr von der Prozessorzahl ab.

Damit scheiden zentrale Strukturen für Threadbibliotheken auf NUMA-Architekturen aus, da Threadbibliotheken kurzlaufende Threads unterstützen sollen und NUMA-Architekturen sowohl hohe Zugriffszeiten auf zentrale und damit meist entfernte Strukturen als auch hohe Prozessorzahlen aufweisen.

3.2.2 Warteschlangenbasierte Zuordnungsverfahren

Um die oben beschriebenen Probleme zu umgeben, bleibt nur eine Verteilung der Threadverwaltungsstrukturen. In einigen Betriebssystemen existieren bereits hierarchische Schedulerstrukturen mit einer globalen Runqueue und lokalen Runqueues pro Prozessor (siehe etwa [BB95]). Prozesse werden in diesen Systemen der globalen Warteschlange entnommen und kehren nach einer Blockierung in die lokale Runqueue zurück. Eine Zuweisungsmöglichkeit von Prozessen an lokale Queues existiert aber im allgemeinen nicht.

Eine Threadbibliothek kann in seinen Strukturen der Hierarchie der Maschine angepaßt werden. Abbildung 3.3 zeigt diese Anpassung der Warteschlangenhierarchie an eine Convex SPP. Für jede Hierarchieebene existieren eigene Warteschlangen.

Prozessorqueues sorgen dafür, daß Threads an einen Prozessor gebunden werden können. Da keine Priorisierung mehr notwendig ist, genügen hier einfache Implementierungen, wie lineare Listen. Neben dem dadurch entstehenden Vorteil eines schnelleren Zugriffs als auf aufwendigere Strukturen entfallen hier die vorher notwendigen Lockingaufrufe, solange kein Prozessor auf die lokale Runqueue eines fremden Prozessors zugreifen darf. Darüberhinaus garantieren die privaten Strukturen maximale Parallelität.

Sollte ein Prozessor keinen lauffähigen Thread mehr besitzen, steigt er in die nächste, darunterliegende Hierarchiestufe ab. In der Abbildung wird er somit als nächstes die lokale Warteschlange des Hypernodes untersuchen, in die Threads eingetragen werden, die man an den Hypernode, nicht aber an einen speziellen Prozessor binden wollte. Erst wenn auch die Warteschlange des Hypernodes leer sein sollte, wird diejenige des Subkomplexe untersucht. Diese enthält die Threads, für die keine Bindung spezifiziert wurde, die also beliebig innerhalb des Systems wandern können.

Vergleicht man die dadurch entstandene Einteilung der Threads in Klassen verschiedener Bindung mit der in Abbildung 3.1, so erkennt man wieder die drei dort gezeigten Klassen. Im

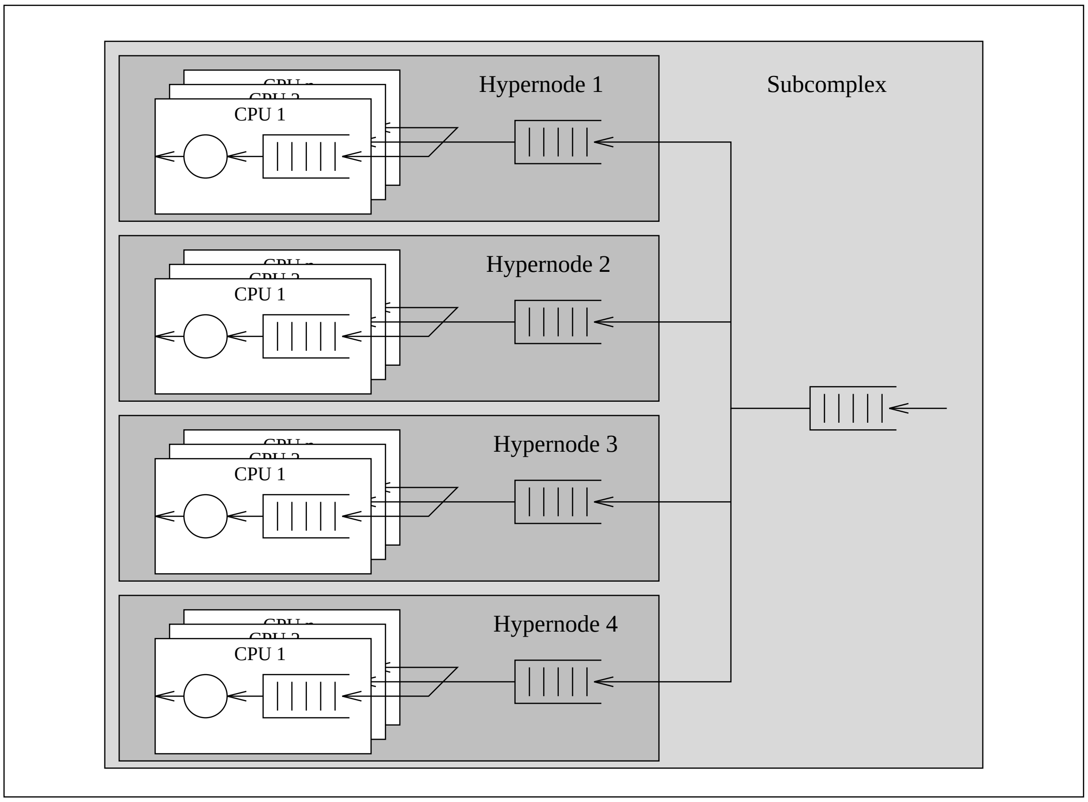


Abbildung 3.3: Hierarchische Warteschlangenstrukturen

Gegensatz zur Bindung durch Prioritäten existiert aber bei den Verfahren auf der Basis von Warteschlangen in dieser einfachen Implementierung keine feinere Spezifikationsmöglichkeit.

Um die Affinität eines Threads zu den Caches seines Prozessors aufrecht zu erhalten, wird ein Thread, wenn er seinen Prozessor abgibt oder nach einer Blockierung wieder deblockiert wird, in die Runqueue seines vorherigen Prozessors eingetragen. Somit ergibt sich ein Verhalten ähnlich dem von Tucker in [Tuc93] vorgeschlagenen Prioritätsmodell, jedoch ohne die Option einer Erweiterung für Affinitätsbewertungen bzgl. mehrerer Prozessoren. Im nächsten Kapitel wird aber gezeigt, wie die lokalen Warteschlangen zu einer erheblichen Verfeinerung der Affinitätsbetrachtungen verwendet werden können.

3.2.3 Verzögerte Threaderzeugung

Threadbibliotheken stellen sehr leichtgewichtige Threads zur Verfügung, die zu jedem Zeitpunkt der Berechnung erzeugt werden können. Solange neu erzeugte Threads nicht zwingend für den Fortgang des Berechnungvorgangs erforderlich sind und deshalb sofort gestartet werden müssen, kann dieser Start verzögert werden. Anstatt einen kompletten Thread, mitsamt des dafür notwendigen Kontextes aufzubauen, wird nur eine Struktur mit den für den Start des Threads notwendigen Daten gefüllt. Diese Verzögerung des Threadstarts ergibt verschiedene Vorteile:

- Für den Parametersatz wird bedeutend weniger Speicher benötigt, als für einen kompletten Thread, bestehend aus Threadstruktur, Stack usw.

- Durch die Verringerung der Anzahl anzufordernder Strukturen sinkt die Zeit für die Threaderzeugung.

- Soll der neu erzeugte Thread einem Prozessor auf einem anderen Hypernode zugewiesen werden, so sollte der Zielprozessor die Strukturen erzeugen. Zumindest auf einer Convex SPP kann ein Prozessor nicht direkt lokalen Speicher eines anderen Prozessors anfordern. Stattdessen müßte er eigenen Speicher verwenden, der für den anderen Prozessor dann als entfernter Speicher hohe Kosten verursachen würde.

- Durch die Unterteilung in startfähige und lauffähige Threads ergibt sich eine eindeutige Trennung zwischen Threads mit verschwindend geringer Affinität zu irgendeinem Prozessor und solchen mit zu beachtender Affinität. Beim Lastausgleich können dann zuerst startfähige Threads migiert werden, bei denen kaum Verlust von Cacheinhalten zu erwarten ist. Erst wenn keine startfähigen Threads mehr existieren muß ein – meist aufwendigerer – Algorithmus entscheiden, welcher der lauffähigen Threads für eine Migration herangezogen wird.

Erst in dem Moment seines tatsächlichen Starts wird aus dem Parametersatz für einen Thread ein vollständiger Thread instanziiert. Die dafür benötigten Strukturen, wie z.B. der Threadstack können dabei von einem vorher terminierten Thread übernommen werden. Dadurch erbt der neue Thread die verbliebene Affinität dieses Stacks zum Datencache seines Prozessors. Darüberhinaus entfällt wieder die Zeit für die Anforderung des Speichers. Falls der Thread, der den Threadwechsel auslöst gerade terminiert, kann im Idealfall der neue Thread direkt den Stack des alten Threads übernehmen. Diese, auch als sogenannte *Continuation Passing* bekannte Technik ergibt, je nach eingesetzter Prozessortechnologie eine weitere Beschleunigung des Threadwechsels, da kein Wechseln des Stacksegments mehr erforderlich wird([FL92]). Auch wenn dieser direkte Wechsel nicht unterstützt wird, so sollte bei jedem Prozessor mindestens der letzte frei gewordene Stack gehalten werden, damit bei einem Threadwechsel auf diesen umgeschaltet werden kann.

3.2.4 Lokale Freilisten für häufig benötigte Strukturen

Das Beispiel der Wiederverwendung von Stacks zeigt bereits, daß es vorteilhaft sein kann, häufig benötigte Strukturen nicht sofort freizugeben, sondern für eine Wiederverwendung zwischenzuspeichern.

Als einfachste Lösung bieten sich hier zentrale Puffer an, aus denen Strukturen entnommen werden können, bzw. wieder zurückfließen. Solange diese Puffer noch Elemente enthalten, brauchen keine Speicheranforderungen an das Betriebssystem gestellt werden. Jedoch gelten auch hier wieder die bei der zentralen Threadverwaltung getroffenen Aussagen, nachdem diese Puffer schnell wieder zu einem Flaschenhals werden. Außerdem geht dabei die Information über eine verbleibende Affinität zum vorherigen Prozessor verloren.

Freilisten für Speicherbereiche sollten also wieder lokal angelegt werden, um einerseits den parallelen Zugriff und andererseits den Erhalt von Lokalitätsinformation sicherzustellen. Als Nachteil lokaler Puffer stellt sich aber die notwendige Balancierung heraus. Vor allem bei Programmen, die nach dem sogenannten *Fork-Join*-Modell arbeiten, bei denen also meist ein Thread viele andere Threads erzeugt und anschließend darauf wartet, bis sie durch die vorhandenen Prozessoren abgearbeitet wurden, kommt es zwangsläufig zu Ungleichgewichten zwischen den Puffern. Der Prozessor des erzeugenden Threads muß andauernd Strukturen anfordern, während die Prozessoren, die die Threads dann bearbeiten freiwerdende Strukturen zwar in ihre Freilisten eintragen aber nicht mehr weiter verwenden. Als Lösung kann hier wieder ein zentraler Puffer eingeführt werden, in den überzählige Strukturen abfließen. Um dadurch nicht wieder einen Engpaß zu erzeugen, werden nicht einzelne Strukturen aus den lokalen Puffern in den zentralen übernommen, sondern immer gleich mehrere. Als einfache, aber effektive Lösung hat sich dabei folgender Algorithmus bewährt ([ALL89]):

- Jeder lokale Puffer kann eine gewisse, maximale Anzahl M an Strukturen speichern.
- Überschreitet die Anzahl der Strukturen diesen Maximalwert, so werden $M/2$ Elemente in den globalen Pool übernommen.
- Enthält der lokale Puffer bei einer Anfrage keine Elemente mehr, so werden ebenfalls wieder $M/2$ Elemente aus dem globalen Puffer bzw. falls dieser leer sein sollte, vom Betriebssystem angefordert.

Der Parameter M kann dazu dienen einen Kompromiß zu finden zwischen größeren Puffern mit dem Vorteil lokalen Zugriffs auf Kosten höheren Speicherverbrauchs und kleineren, ausgeglicheneren Puffern aber höheren Kosten aufgrund häufigerer Balancierung.

Puffer zur Verwaltung freier Strukturen müssen aber nicht zwangsläufig zu höherem Speicherverbrauch führen. Gerade wenn in Systemen für bestimmte Strukturen eine feste Ausrichtung im Speicher gefordert wird, können sich Puffer vorteilhaft auswirken. Bei den in einer Convex SPP verwendeten HP PA-RISC Prozessor müssen z.B. alle Strukturen auf die mit einem atomaren `load-and-clear` Befehl zugegriffen werden soll an 16 Byte Grenzen ausgerichtet sein. Aber auch für andere Strukturen kann eine Ausrichtung an bestimmten Speichergrenzen sinnvoll sein. So kann die richtige Platzierung einer Struktur dafür sorgen, daß sie auf eine Cachezeile abgebildet wird, anstatt auf zwei aufeinander folgende. Beim Zugriff auf die Struktur halbiert sich damit die Anzahl auftretender Cachemisses.

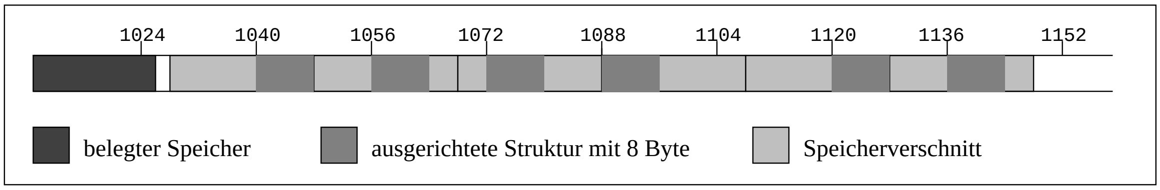


Abbildung 3.4: Speicherverschnitt bei Einzelanforderung ausgerichteter Strukturen

Da diese Ausrichtung aber vom System nicht direkt unterstützt wird, bleibt nur die Möglichkeit jede derartige Struktur künstlich zu verlängern und anschließend von Hand auszurichtet. Abbildung 3.4 zeigt die Aufteilung des Speichers, wenn mehrere acht Byte lange Strukturen im Speicher in dieser Art ausgerichtet werden[2]. Wird hingegen für einen Puffer ein größerer Speicherblock angefordert, ausgerichtet und anschließend in passende Stücke geteilt, so reduziert sich der Verschnitt.

[2] Angenommen wird 32Bit Alignment durch das Betriebssystem

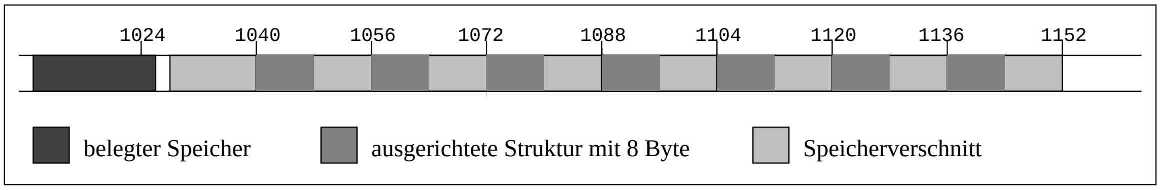


Abbildung 3.5: Speicherverschnitt bei der Pufferverwaltung ausgerichteter Strukturen

Abbildung 3.5 zeigt die resultierende Speicherbelegung. Abgesehen von der ersten Ausrichtung entsteht für jede Struktur nur noch die Differenz zum nächsten Vielfachen von 16 Byte als Verschnitt. Bei einer Strukturgröße von 16 Byte entsteht also kein Verschnitt mehr, während für dieselbe Struktur bei Einzelanforderung 28 Byte verwendet werden müßten.

3.2.5 Lastausgleich

Die Algorithmen und Datenstrukturen in klassischen Betriebssystemen und Threadbibliotheken mit ihrer zentralen Runqueue waren darauf ausgerichtet, für einen möglichst gleichmäßigen Lastausgleich zu sorgen. Solange noch lauffähige Prozesse existierten, waren alle Prozessoren beschäftigt. Untersuchungen von Markatos und LeBlanc ([Mar93], [ML93], [ML91]) haben gezeigt, daß diese Strategie auf modernen Architekturen, insbesonders NUMA-Architekturen zu dramatischen Leistungseinbußen führen kann. Trotz der Forderung Threads möglichst nicht zu migrieren, kann aber trotzdem nicht vollständig auf Lastausgleich verzichtet werden. Sobald ein Prozessor keinen lokalen Thread mehr vorfindet, muß er sich auf die Suche nach einem anderen, geeigneten Thread machen, wobei verschiedene Kriterien diese Suche beeinflussen.

Zeit für die Suche eines geeigneten Threads

Als ein entscheidender Faktor bei der Suche nach einem zu migrierenden Thread spielt die dafür notwendige Zeit eine wichtige Rolle. Der ideale Lastausgleichsalgorithmus sollte zwar unter allen möglichen Threads des gesamten Systems denjenigen aussuchen, der ein angegebenes Optimalitätskriterium maximiert, jedoch dürfte dieses Ziel kaum realisierbar sein. Besonders auf größeren Systemen könnte der Aufwand für die Suche den Gewinn des Lastausgleichs aufzehren. Darüberhinaus könnte sich bis zum Ende der Suche die Situation schon vollständig geändert haben.

Umfang der Suche

Besonders bei der vorher beschriebenen Threadverwaltung mit lokalen Runqueues pro Prozessor, muß der suchende Prozessor auf diese Runqueues zugreifen können. Das dafür notwendige Setzen von Mutex-Locks für den gegenseitigen Ausschluß, sowie die dabei entstehenden Cachemisses führen zu einer Störung des Berechnungsvorgangs auf den anderen Prozessoren. Je mehr Prozessoren frei werden und innerhalb des Systems nach Threads suchen, desto höher wird diese Belastung und die dadurch entstehende Verzögerung.

Darüberhinaus besteht die Gefahr, daß mehrere suchende Prozessoren zu demselben Ergebnis kommen, d.h. entweder versuchen sie alle denselben Thread zu migrieren, so daß alle bis auf einen Prozessor neu mit ihrer Suche beginnen müssen oder sie entfernen alle von derselben Warteschlange und erzeugen damit unter Umständen ein weiteres Lastungleichgewicht. Läßt man den Lastausgleich also über die gesamte Maschine zu, so müssen die einzelnen Entscheidungen serialisiert werden. Um dem entgegenzuwirken, kann man den Radius der Suche einschränken. Jeder Prozessor sucht nur auf einem Teil der Maschine nach einem neuen Thread. Mehrere Prozessoren können somit unbeeinflußt voneinander Entscheidungen treffen und die effektiv daraus resultierende Störung arbeitender Prozessoren wird vermindert.

Nutzen aus dem Lastausgleich

Jede Migration eines Threads erzeugt Kosten, die anschließend ausgeglichen werden müssen. Besonders auf NUMA-Architekturen kann sonst die Migration eines Threads zu einem noch größerem Lastungleichgewicht führen. Abbildung 3.6 zeigt ein Beispiel mit zwei Prozessoren.

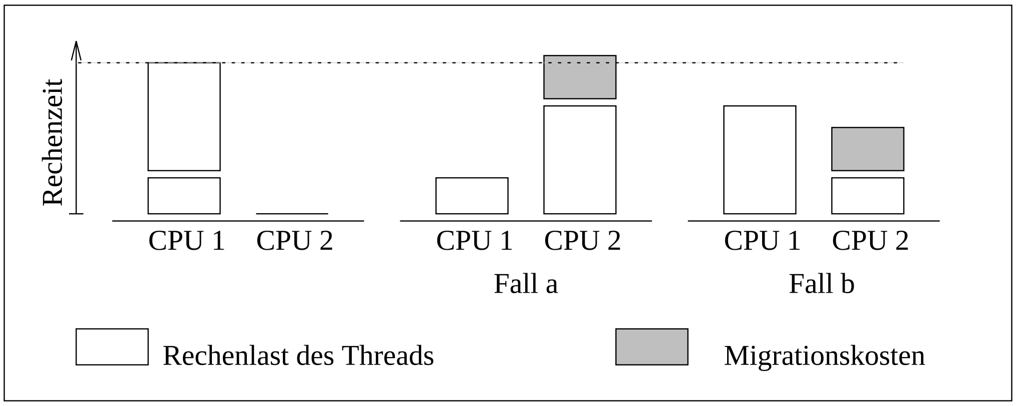


Abbildung 3.6: Verbesserung durch Lastausgleich

Prozessor 1 besitzt noch zwei lauffähige Threads, während Prozessor 2 sich auf die Suche nach einem neuen Thread begibt. In Fall *a* wählt er den oberen der beiden Threads. Durch die Kosten, die bei der Migration – etwa durch den Verlust des Cacheinhaltes des verschobenen Threads – entstehen, ergibt sich aber jetzt trotz Einsatz von zwei Prozessoren eine höhere Rechenzeit als bei alleiniger Bearbeitung durch Prozessor 1. Erst bei der Migration des anderen Threads im Fall *b* entsteht unter der Annahme derselben Kosten eine tatsächliche Verbesserung.

Das Beispiel zeigt bereits, daß beim Lastausgleich mehrere Faktoren eine Rolle spielen. Stellt sich das Problem bei zwei Prozessoren noch recht einfach dar, da hier klar ist von welchem Prozessor Threads entfernt werden müssen, um die Last zu verringern, so steigt die Komplexität mit der Prozessorzahl stark an. Bei mehreren Prozessoren kann es von Vorteil sein, einen Thread zu migrieren, bei dem weniger Gewinn zu erwarten ist, um dadurch die Voraussetzungen für eine Migration mit höherem Gewinn zu schaffen. Verzichtet man auf diese Art der Optimierung, so kann man sich für eine Reduzierung der Rechenzeit durch Lastausgleich auch einfacher Heuristiken bedienen.

Voraussetzung für die Abschätzung des Gewinns durch Lastausgleich sind ungefähre Kenntnisse über die durch eine Thread T_i hervorgerufene Last $L(T_i)$ und die bei einer Migration zu erwartenden Kosten $P(T_i)$ (**P**enalty). Schätzwerte für die Last können z.B. durch die bisherige, mittlere Laufzeit des Threads gewonnen werden, wohingegen für die Migrationskosten etwa Informationen über die Cacheaffinität des Threads verwendet werden können. Als Kandidaten für eine Migration kommen nur diejenigen Threads j aus einer Menge M verfügbarer Threads eines anderen Prozessors in Frage, für die eine tatsächliche Verbesserung entsteht, d.h. für die gilt:

$$\sum_{T_i \in M} L(T_i) > max(\sum_{\substack{T_i \in M \\ i \neq j}} L(T_i), L(T_j) + P(T_j)) \tag{3.7}$$

Aus diesen kann nun weiter ausgewählt werden, wobei man verschiedene Ziele verfolgen kann:

- *Abbau von Lastspitzen:* Wählt man unter den Threads des Prozessors mit der höchsten Last denjenigen mit der größten Verbesserung aus, so verringert man dadurch diese Lastspitze, d.h. die Maximallast hat sich verringert, die Lastverhältnisse sind ausgeglichener. Besonders am Ende einer Berechnung, wenn keine neuen Threads mehr erzeugt werden, sorgt diese Strategie dafür, daß möglichst lange alle Prozessoren beschäftigt werden können und sich damit die Gesamtrechenzeit bis zum Ende des letzten Threads verringert.

- *Größter Gewinn:* Anstatt nur die Threads des Prozessors mit der höchsten Last zu untersuchen, sucht man hier unter einer Menge von Prozessoren – etwa auf dem lokalen Hypernode – denjenigen Thread, bei dem die größte Verbesserung erzielt werden kann. Diese Strategie bietet sich vor allem dann an, wenn ein Ende der Berechnung noch nicht in Sicht ist, d.h. die Länge einer Warteschlange noch kein Maß für die zu erwartende Gesamtrechenzeit darstellt.

Durch die Auswahl der zu untersuchenden Prozessoren, bzw. Threads und eine maschinen- und threadspezifische Berechnung der Migrationskosten kann diese Strategie an verschiedene Optimalitätskriterien angepaßt werden.

Kapitel 4

Ergänzende Affinitätsbetrachtungen

Alle bisher vorgestellten Konzepte des Memory Conscious Scheduling waren darauf bedacht, die Antwort darauf zu geben, wo ein Thread berechnet werden soll. Threads, deren Berechnung bereits begonnen hat sammeln wie gesehen Daten in den lokalen Caches an. Um diese Daten möglichst effektiv zu nutzen, stellt sich neben dem Ort der Berechnung auch noch die Frage, wann ein Thread zu berechnen sei, mit anderen Worten die Reihenfolge in denen die einzelnen Threads eines Prozessors bearbeitet werden sollen.

Dieses Kapitel präsentiert Methoden, mit deren Hilfe der Cachezustand eines Threads approximiert und für die Entscheidung über die Berechnungsreihenfolge verwendet werden kann. Dazu dient neben heuristischen Überlegungen auch ein einfaches mathematisches Modell. Außerdem werden mögliche Datenstrukturen auf ihre Eignung für diese Aufgabe hin untersucht.

4.1 Abhängigkeit der Rechenzeit vom Cachezustand

Jeder durch einen Prozessor ausgelöste Cachemiss bedeutet eine unnötige Verzögerung für den Berechnungsvorgang. Besonders auf NUMA-Architekturen wie der Convex SPP schlagen die hohen Latenzzeiten für den Zugriff auf die verschiedenen Ebenen der Speicherhierarchie stark zu Buche. Um also den größtmöglichen Nutzen aus der Maschine ziehen zu können, muß die Anzahl der Cachemisses auf die absolut notwendigen reduziert werden.

Man kann dabei zwei Klassen von Cachemisses unterscheiden. Auf der einen Seite stehen diejenigen, die beim Start eines Threads entstehen. In diesem Moment wird eine Grundmenge an Daten, die der Thread benötigt in die Caches geladen. Dazu zählen neben der Threadstruktur und dem Stack auch Datenbereiche mit denen der Thread arbeitet. Kann man dafür sorgen, daß diese Arbeitsmenge beim Start bereits im Cache vorliegt, bzw. dort noch vom letzten Berechnungsabschnitt steht, so reduziert sich die Anzahl der Cachemisses und damit die effektive Berechnungsdauer dieses Berechnungsabschnittes. Je kürzer dieser Abschnitt dauert, desto stärker wird der Einfluß dieser Cachemisses und umso größer damit der Gewinn, falls Daten bereits vorliegen. Da aber die Threadverwaltung auf Benutzerebene gerade kurze Threadlaufzeiten unterstützen sollte, steigt damit die Notwendigkeit derartiger Betrachtungen.

Neben den Cachemisses zum Laden der Arbeitsmenge beim Threadstart werden auch während der Berechnung des Threads Cachemisses ausgelöst. Diese resultieren aus Invalidierungen bestehender Cachezeilen oder Veränderungen im Zugriffsverhalten, d.h. einer Änderung der Arbeitsmenge. Sie können zwar durch die Berechnungsreihenfolge nicht vermindert werden, jedoch bilden sie während der Berechnung ein Abbild der Arbeitsmenge im Cache, die in einem folgenden Berechnungsabschnitt wieder weiterverwendet werden kann.

Für den Verlust an Cacheaffinität, d.h. der Anzahl der von einem Thread belegten Cachezeilen gibt es somit zwei hauptsächliche Gründe:

1. Während ein Thread nicht bearbeitet wird, werden einzelne Cachezeilen durch die Abbildung der Arbeitsmengen anderer Threads verdrängt. Je mehr Threads zwischendurch bearbeitet und je mehr Cachemisses dabei ausgelöst wurden, desto geringer wird der beim Neustart eines Threads verbliebene Rest seiner Arbeitsmenge sein.

2. Zu jeder Zeit, in der Berechnungsphase ebenso wie während einer Blockierung, können durch Schreibzugriff anderer Prozessoren Cachezeilen invalidiert werden und damit verloren gehen.

Interessant wird die Betrachtung der Cacheaffinität immer dann, wenn ein Thread häufig blockiert und deblockiert wird. Nach seiner Deblockierung muß aufgrund der verbliebenen Affinität eine Entscheidung darüber getroffen werden, wann er wieder gestartet werden soll. Je mehr seiner Arbeitsmenge noch im Cache verblieben ist, desto geringer sollte die Anzahl der Cachemisses beim erneuten Start dieses Threads sein. Aufgabe der Threadverwaltung muß also die Approximation des Cachezustandes und die Umsetzung in eine geeignete Priorität sein.

Alternativ zu den Überlegungen der Arbeitsmenge und den daraus resultierenden Cachemisses kann man aus dem Verhalten eines Threads in seinem letzten Berechnungsabschnitt auf sein weiteres Verhalten schließen. Hier wird die Rate ausgelöster Cachemisses als charakteristische Eigenschaft eines Threads angesehen und nicht durch den Einfluß anderer Threads begründet([Bel95a]).

4.2 Affinitätsmaße

Im Gegensatz zur Lokalitätsinformation für MCS, die entweder zur Übersetzungszeit oder während des Programmlaufs ermittelt und ausgewertet werden kann, gibt es für die Messung des Cachezustandes eines Threads keine direkte Zugangsmöglichkeit. Man ist dadurch auf Schätzungen und Heuristiken angewiesen, um entweder den Cachezustand oder direkt die sich daraus ergebende Ordnung auf den Threads annähern zu können.

Zwei verschiedene Arten an Affinitätsmaßen lassen sich dabei unterscheiden:

1. Statische Maße: Bei diesen Maßen wird bei der Blockierung oder Deblockierung ein Wert für die Affinität bzw. die Priorität ermittelt. Dieser Wert bleibt konstant bis der Thread weiter bearbeitet wird.

2. Dynamische, zeitabhängige Werte: Diese Werte altern während der Wartezeit des Threads im Zustand lauffähig oder blockiert. Dabei kann die Alterung entweder durch eine tatsächlich zu erfolgende Änderung des Wertes in bestimmten Abständen erfolgen oder aber durch eine andere Bewertung dieses Wertes. Letztere Methode hat vor allem Vorteile bei einer Alterung, die die Ordnung erhält, d.h. falls für die Priorität P zweier

Threads T_i und T_j zum Zeitpunkt t gilt: $P_i(t) < P_j(t)$, dann auch für alle $t + \Delta t$. In diesem Fall genügt die Auswertung der Alterung beim Einordnen eines Threads in die Runqueue.

4.2.1 Zeitbasierte Maße

Als einfachste Möglichkeit Affinität zu approximieren, bietet sich die Auswertung von Zeitinformationen an.

Virtuelle Zeit

Wie oben erwähnt, verdrängen alle während der Blockierung eines Threads auf seinem vorherigen Prozessor berechneten, anderen Threads Teile seines Cachezustandes. Je mehr Threads zwischendurch berechnet werden, desto weiter sinkt seine Affinität. Besonders bei gleichartigen Threads wie sie etwa bei der Parallelisierung von Schleifenkonstrukten entstehen, bei denen alle Threads in etwa dieselben Eigenschaften aufweisen, kann davon ausgegangen werden, daß die inverse Anzahl zwischenzeitlich berechneter Threads eine gute Approximation für das Verhalten der Cacheaffinität liefert.

Wird ein Thread deblockiert, so wird diese Zahl ermittelt und mit denen in der Runqueue verglichen. Am einfachsten wird dafür eine Art virtuelle Zeit verwendet. Bei jeder Schedulingentscheidung durch einen Prozessor wird sein Zeitzähler inkrementiert. Ein Thread, der sich blockiert, speichert diesen Wert und verwendet ihn bei seiner Deblockierung als Priorität. Ein höherer Wert steht für einen Thread während dessen bisherigen Wartezeit weniger andere Threads berechnet wurden. Es handelt sich also hier um ein dynamisches Maß, bei dem sich die Bewertung des Prioritätswertes ändert und während der Alterung die Ordnung der Threads innerhalb der Runqueue erhalten bleibt. Da darüberhinaus dieses Maß minimalen Soft- und Hardwareaufwand erfordert kann es auch bei einfachen Systemen verwendet werden, die keine weitere Hardwareunterstützung zur Verfügung stellen.

Verwendung der Systemzeit

Statt der virtuellen Zeit kann jederzeit auch eine von der Hardware zur Verfügung gestellte, hochauflösende Systemzeit verwendet werden. Diese bietet aber an sich keine weiteren Vorteile, da sich keine andere Ordnung ergibt als bei der Auswertung der virtuellen Zeit. Vorteilhaft kann eine globale Systemzeit dann sein, wenn Threads für die Migration gesucht werden. Teilen sich alle Prozessoren eine gemeinsame Zeit, so kann ohne Probleme erkannt werden, welcher Thread die längste Zeit nicht bearbeitet wurde. Bei ihm sollte dann der Affinitätswert am stärksten gesunken und damit die Migration mit den wenigsten Verlusten möglich sein.

Beide bisherigen Methoden waren gut für Threads geeignet, die etwa dieselben Eigenschaften aufweisen. Andererseits nehmen beide Methoden aber auch keine Rücksicht auf die Eigenschaften der Threads, wie Rechenzeit, Zugriffsverhalten o.ä. Eine erweiterte Verwendung der Systemzeit kann darin bestehen, sie wie in Betriebssystemen dazu zu benützen, die Rechenzeit eines Threads zu bestimmen. Besonders bei Threads, die sehr lange Zeit rechnen kann davon ausgegangen werden, daß sie dabei ihre Arbeitsmenge komplett in den Cache laden. Sie sollten also bevorzugt weiterberechnet werden. Es zeigt sich aber, daß die Rechenzeit alleine als statisches Affinitätsmaß nicht genügt, da nicht berücksichtigt wird, wielange der Thread blockiert war. Ein geeigneter Alterungsalgorithmus muß dafür sorgen, daß die

Wartezeit der Threads berücksichtigt wird, d.h. die Priorität eines Threads T_i mit Rechenzeit $L(T_i)$ berechnet sich zum Zeitpunkt t aus einer Funktion $P_i(t) = f(L(T_i), t)$.

Verwendung findet diese Strategie auch in Betriebssystemen, wie etwa der Silicon Graphics IRIX. Dort wird ein linearer Zusammenhang zwischen Cacheaffinität und Prozeßlaufzeit angenommen. Sobald die Laufzeit eines Prozesses eine minimale Schranke überschreitet, wird diese Laufzeit als Affinitätsmaß verwendet. Umgekehrt wird gemessen, wielange ein Prozessor seit der Blockierung eines Prozesses andere Prozesse bearbeitet hat. Übersteigt diese Zeit eine bestimmte obere Grenze, so wird die Bindung des Prozesses zu seinem Prozessor aufgegeben([BB95]).

Die einfachste Möglichkeit, Alterung zu verwenden besteht darin, die Rechenzeit des Threads mit der inversen Blockierungszeit zu gewichten, d.h. für einen Thread i mit einem Startzeitpunkt $Start_i$ und einem Blockierungszeitpunkt $Stopp_i$ errechnet sich die Priorität zum Zeitpunkt t zu

$$P_i(t) = \frac{Start_i - Stopp_i}{t - Stopp_i} \tag{4.1}$$

Abbildung 4.1 zeigt den Verlauf der Prioritäten zweier Threads T_i und T_j, wobei Thread T_i 64 Zeiteinheiten rechnete und direkt anschließend Thread T_j für 16 Zeiteinheiten. Bereits dieses einfache Beispiel verdeutlicht aber ein Problem, daß durch diese Alterungsfunktion entsteht. Die Hyperbeln der Prioritäten der beiden Threads schneiden sich zum Zeitpunkt

$$t_s = \frac{Stopp_j(Stopp_i - Start_i) - Stopp_i(Stopp_j - Start_j)}{(Stopp_i - Start_i) - (Stopp_j - Start_j)} \tag{4.2}$$

Da dieser Schnittpunkt aber wie im Beispiel zeitlich auch nach $Stopp_j$ liegen kann, besteht die Gefahr, daß sich die Ordnung innerhalb der Warteschlange ändert.

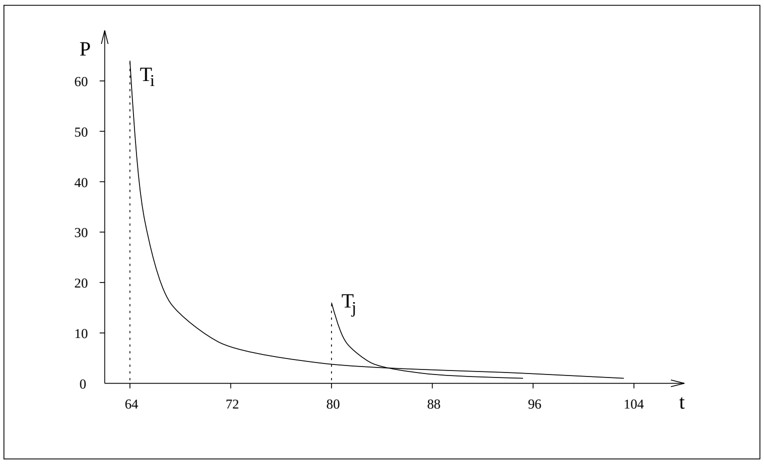


Abbildung 4.1: Hyperbolische Alterungsfunktion

Ausgeschlossen ist dieser Effekt bei einer anderen Art von Alterungsfunktionen, nämlich bei der exponentiellen Gewichtung der Rechenzeit mit der Blockierungszeit in der Form

$$P_i(t) = \frac{Start_i - Stopp_i}{K^{a(t-Stopp_i)}} \tag{4.3}$$

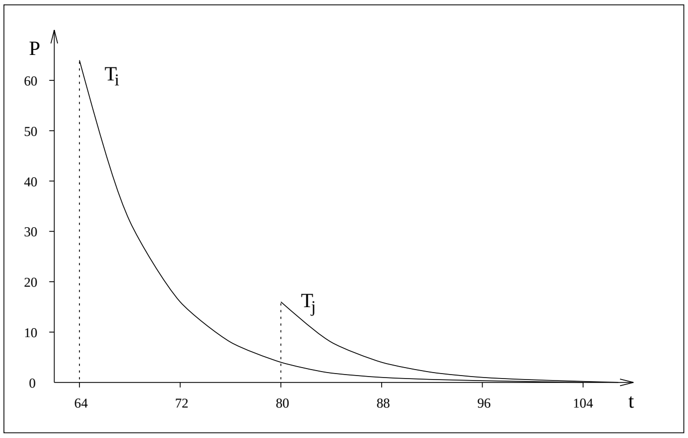


Abbildung 4.2: Exponentielle Alterungsfunktion

Die Parameter $K > 1$ und $a > 0$ lassen dabei verschiedene Ausprägungen des Alterungsvorgangs zu. Abbildung 4.2 zeigt die Kurven des vorherigen Beispiels mit zwei Threads und den Parametern $K = 2$ und $a = 0.25$. Bei dieser Art der Prioritätsberechnung genügt es, die Relation zwischen zwei Threads für einen Zeitpunkt zu bestimmen, wie etwa für $t = Stopp_j$. Sie bleibt dann für alle späteren Zeitpunkte erhalten, da sich die Kurven der Prioritätsfunktionen der beiden Threads für endliche Zeiten t nicht schneiden:

$$
\begin{aligned}
P_i(t) < P_j(t) \quad &\leftrightarrow \quad \frac{Start_i - Stopp_i}{K^{a(t-Stopp_i)}} < \frac{Start_j - Stopp_j}{K^{a(t-Stopp_j)}} \\
&\leftrightarrow \quad \frac{Start_i - Stopp_i}{K^{a(-Stopp_i)}} < \frac{Start_j - Stopp_j}{K^{a(-Stopp_j)}} \\
&\leftrightarrow \quad \frac{Start_i - Stopp_i}{K^{a(Stopp_j-Stopp_i)}} < Start_j - Stopp_j \\
&\leftrightarrow \quad P_i(Stopp_j) < P_j(Stopp_j) \qquad (4.4)
\end{aligned}
$$

Beim Eintragen eines neuen Threads muß für alle Threads, mit denen der einzutragende Thread verglichen werden soll die Priorität neu berechnet werden. Dadurch läßt sich erkennen, daß eine Struktur für die Runqueue verwendet werden muß, die eine möglichst kleine Anzahl an Vergleichen ermöglicht. Diese Aussage gilt im übrigen für alle dynamischen Maße mit der Eigenschaft, daß sie trotz Alterung die Ordnung erhalten und damit mit einer einfachen Prioritätswarteschlange realisiert werden können.

4.2.2 Cachemiss-basierte Affinitätsmaße

Um genauere Aussagen über die Cachenutzung von Threads treffen zu können, werden immer mehr moderne Systeme mit *Performance Monitoring* Funktionen ausgerüstet, die die Messung von Cachemiss-Zahlen und Latenzzeiten ermöglichen. Abhängig von den verwendeten Prozessoren wird diese Aufgabe entweder direkt von den Prozessoren übernommen, wie etwa bei der HPPA 8000 CPU oder der SGI R10000 CPU, gemischt mit Hilfe zusätzlicher Hardwareunterstützung wie bei der HPPA 7200 in der Convex SPP 1200 oder allein durch externe Hardware wie bei der HPPA 7100 in der Convex SPP 1000. Trotzdem erlauben alle bisherigen Systeme keine direkte Messung der Affinität eines Threads zu einem Prozessor.

Direkte Verwendung von Cachemiss-Zahlen

Geht man davon aus, daß das Cacheverhalten eines Threads innerhalb seines letzten Berechnungsabschnitts eine gute Prognose für sein zukünftiges Verhalten darstellt, dann kann man Cachemiss-Zahlen direkt als Affinitätsmaß verwenden. Eine Überlegung geht davon aus, daß ein Thread, der im letzten Abschnitt wenige Cachemisses erzeugte auch im nächsten Abschnitt wenig Cachemisses erzeugen wird, da er den Großteil seiner Arbeitsmenge im Cache vorliegen haben muß. Statt der sofortigen Verwendung der Cachemisses kann auch über ein geeignetes Verfahren ein Mittelwert der letzten Abschnitte berechnet werden. Dadurch kann diese Strategie das Verhalten des Threads in den letzten Abschnitten besser berücksichtigen.

Daß es sich hier um eine zweifelhafte Heuristik handelt wird klar, wenn man sich vor Augen hält, daß dadurch neugestartete Threads eindeutig benachteiligt werden. Diese werden beim Start eine große Zahl an Cachemisses erfahren wenn sie ihre Arbeitsmenge aufbauen müssen. Daraus resultiert dann eine geringe Priorität, da der Schedulingalgorithmus mit einer hohen Anzahl an Cachemisses im nächsten Schedulingabschnitt rechnet. Bis sie wieder zum Zuge kommen wurde dann aber der Großteil ihrer Daten verdrängt, so daß sie wieder viele Cachemisses auslösen. Somit behält der Algorithmus zwar recht, indem er dieses Verhalten ja prognostizierte, leider aber auch auslöste.

Besser scheint damit der umgekehrte Schluß, aus einer hohen Anzahl an Cachemisses darauf zu schließen, daß ein Thread damit einen Großteil seiner Daten in den Cache geladen hat, die es zu erhalten gilt. Jedoch schlägt der umgekehrte Schluß, aus einer geringen Anzahl von Cachemisses eine geringe Affinität zu folgern fehl. Zwar könnte der Thread tatsächlich nur wenig Daten in den Cache geladen haben, jedoch könnte der Effekt auch darauf beruhen, daß er bereits einen Großteil dort vorgefunden hat.

Statt aufgrund der Cachemisses des letzten Schedulingabschnitts auf den Cachezustand zu schließen bietet es sich also an, auch die Geschichte des Threads miteinzubeziehen. Anstelle also die Cachemisses direkt zu verwenden, kann also etwa die Summe aller oder der letzten n Abschnitte verwendet werden, wobei ggf. eine gewichtete Mittelung durchgeführt werden kann um den Verlauf der Priorität zu glätten.

4.2.3 Mathematisches Affinitätsmodell

Alle bisherigen Strategien beruhten auf intuitiven Annahmen über die Affinität eines Threads zu seinem Prozessor und in welcher Weise diese Affinität in eine Priorität umgesetzt werden kann. Keine dieser Methoden bezieht den Einfluß der Cachegröße und des Cachemiss-Verhaltens anderer Threads in die Affinitätsberechnung ein. Auch in die Berechnung der Prioritäten gehen keine Betrachtungen ein, die einen Schluß auf die insgesamt zu erwartende Anzahl an Cachemisses liefern könnten.

Bereits in einer Arbeit von 1987 stellten Thiebaut und Stone ein Cachemodell auf, daß es ermöglicht, den sogenannten *Reload Transient* von Prozessen zu berechnen ([TS87]). Als Reload Transient bezeichnen sie die Zeit, die ein Prozeß beim Beginn eines neuen Schedulingabschnittes benötigt, um seine Arbeitsmenge wieder in den Cache zu laden, nachdem andere Prozesse Teile davon aus dem Cache verdrängt haben. Sie bezeichnen diese Arbeitsmenge im Cache als *Footprint* und berechnen aus ihr und unter der Annahme, daß alle Zugriffe auf den Speicher und damit auf den Cache uniform verteilt und voneinander unabhängig sind, die Anzahl der Cachemisses beim Neustart, wenn zwischendurch ein anderer Prozeß mit bekanntem Footprint gelaufen ist. Durch Programmtraces zeigen sie die Übereinstimmung ihres analytischen Modells mit gemessenen Werten. Obwohl die Voraussetzungen – uniforme Verteilung und Unabhängigkeit der Zugriffe – stark vereinfachte Annahmen darstellen, konnten sie eine gute Modellierung der tatsächlichen Vorgänge feststellen.

Problematisch bei der Anwendung ihres Modells, wie es z.B. durch Squillante und Lazowska in [SL89] vorgeschlagen wird, ist die meist fehlende Kenntnis über den Footprint eines Prozesses bzw. Threads. Aufbauend auf den Überlegungen der genannten Arbeiten entstand das nachfolgend beschriebene Modell, das aus der Anzahl erfolgter Cachemisses Rückschlüsse auf die Affinität, also die tatsächlich gültige Anzahl an Cachezeilen eines Threads erlaubt. Dabei wird sowohl die Steigerung der Affinität während der Berechnungsphase eines Threads, als auch die Verminderung während einer Wartezeit miteinberechnet, so daß zu jedem Zeitpunkt eine Aussage über diesen Wert getroffen werden kann. Aufbauend auf diesem Modell wird anschließend eine einfache Strategie zur Umsetzung in Threadprioritäten vorgestellt.

Aufbau der Affinität während der Berechnungszeit

Das vorgestellte Modell geht von denselben Annahmen aus wie das Modell von Thiebaut und Stone, d.h. die Zugriffe sind uniform verteilt und voneinander unabhängig. Darüberhinaus wurde nur der Cachetyp, wie er bei der Convex SPP Verwendung findet – ein *virtually indexed, direct mapped Cache* – untersucht.

Löst ein Thread einen Cachemiss aus, so kann dabei zweierlei auftreten:

1. Der Cachemiss verdrängt bei der Einlagerung der geforderten Daten eine Cachezeile eines anderen Threads, bzw. eine noch nicht weiter verwendete Zeile. Damit nimmt der Cachezustand des laufenden Threads um eine Cachezeile zu.

2. Der Cachemiss verdrängt eine Cachezeile des laufenden Threads und ersetzt sie durch eine andere – der Cachezustand des Threads bleibt also konstant.

Unter den vorher getroffenen Annahmen über Speicherzugriffe und Kenntnis sowohl der Cachegröße N als auch des Cachezustands C des laufenden Threads, kann man die Wahrscheinlichkeiten für diese Ereignisse angeben. Mit einer Wahrscheinlichkeit von C/N wird eine eigene Cachezeile verdrängt; mit der Wahrscheinlichkeit von $(N-C)/N$ die eines anderen Threads. Das Verhalten des Cachezustandes eines Threads läßt sich damit durch die in Abbildung 4.3 gezeigte Markoffkette mit einem absorbierenden Zustand modellieren. Diese

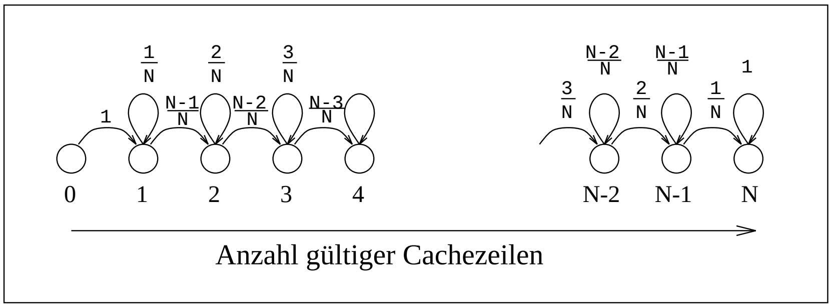


Abbildung 4.3: Markoffkette für die Zunahme des Cachezustandes

Markoffkette läßt sich repräsentieren durch ihre Übergangsmatrix M:

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & & 0 \\ 0 & \frac{1}{N} & \frac{N-1}{N} & 0 & \cdots & & 0 \\ 0 & 0 & \frac{2}{N} & \frac{N-2}{N} & \cdots & & 0 \\ \vdots & & & & \ddots & & \vdots \\ & & & & & \frac{N-1}{N} & \frac{1}{N} \\ 0 & & & & \cdots & 0 & 1 \end{pmatrix} \tag{4.5}$$

Jedes Element $m_{i,j}$ dieser Matrix gibt die Wahrscheinlichkeit an, daß sich bei einem Cachemiss die Anzahl der Cachezeilen des Threads von i auf j ändert. Die Potenzen M^n dieser Matrix ergeben damit die Wahrscheinlichkeiten für einen Übergang von Zustand i zum Zustand j nach n Cachemisses. Für die einzelnen Elemente $m_{i,j}^n$ der Matrix der Mehrschrittübergangswahrscheinlichkeiten ergibt sich folgender Ausdruck:

$$m_{i,j}^n = \begin{cases} 0 & : \ j < i \\ \frac{(N-i)!(-1)^{j+i}}{(N-j)!(j-i)!N^n} \sum_{k=0}^{j-i} \binom{j-i}{k} (k+i)^n (-1)^k & : \ i \le j \le i+n \\ 0 & : \ i+n < j \end{cases} \tag{4.6}$$

Mit Hilfe dieser Wahrscheinlichkeiten läßt sich nun der Erwartungswert für die Anzahl an Cachezeilen ausrechnen, wenn von einer bekannten Anzahl i gestartet wird und n Cachemisses ausgelöst werden:

$$\begin{aligned} E_i^n &= \sum_{j=0}^{N} j m_{i,j}^n \\ &= \frac{(N-i)!(-1)^i}{N^n} \sum_{j=i}^{i+n} j \frac{(-1)^j}{(N-j)!(j-i)!} \sum_{k=0}^{j-i} \binom{j-i}{k} (k+i)^n (-1)^k \\ &\vdots \\ &= N - (N-i) \left(\frac{N-1}{N} \right)^n \end{aligned} \tag{4.7}$$

Abbau von Cachezustand durch andere Threads

Sobald ein Thread in einer Warteschlange steht und andere Threads durch seinen vorherigen Prozessor berechnet werden und dabei Cachemisses auslösen, verdrängen sie dadurch ggf. auch Cachezeilen des wartenden Threads. Analog dem obigen Vorgehen kann auch hier wieder eine Markoffkette zur Modellierung dieses Verhaltens herangezogen werden.

Die Ähnlichkeit dieser Markoffkette mit der vorherigen setzt sich auch in der Matrix der Übergangswahrscheinlichkeiten und derer Potenzen fort. Die Wahrscheinlichkeit $m_{i,j}^n$, daß nach n Cachemisses durch andere Threads noch j Cachezeilen von ursprünglich i übrig geblieben sind, ergibt sich zu:

$$m_{i,j}^n = \begin{cases} 0 & : \ j < i-n \\ \binom{i}{j} \sum_{k=j}^{i} \binom{i-j}{k-j} \left(\frac{N-k}{N} \right)^n (-1)^{k+j} & : \ i-n \le j \le i \\ 0 & : \ i < j \end{cases} \tag{4.8}$$

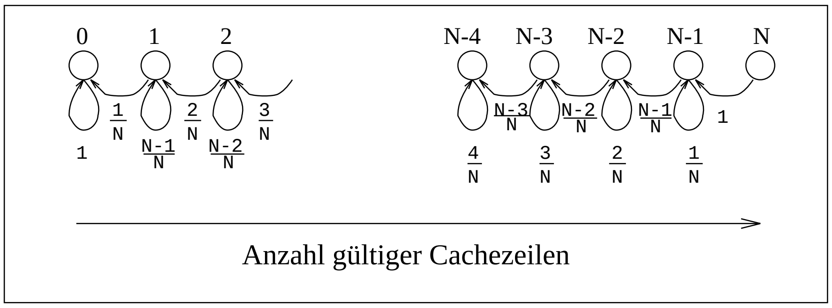


Abbildung 4.4: Markoffkette für die Verdrängung von Cachezeilen

Daraus kann nun der Erwartungswert für die nach n Cachemisses noch verbliebene Anzahl an Cachezeilen errechnet werden, wenn man bei einem bekannten Cachezustand von i Cachezeilen startete:

$$E_i^n = i\left(\frac{N-1}{N}\right)^n \tag{4.9}$$

Zusammensetzen beider Wachstumsfunktionen

Voraussetzung für die Verwendung der oben berechneten Wachstums- bzw. Alterungsfunktionen war die Kenntnis über die Anzahl gültiger Cachezeilen eines Threads. Diese Zahl ist zwar nicht bekannt, jedoch kann sie durch sukzessive Anwendung der beiden Formeln angenähert werden, indem man dafür jeweils, den für den letzten Berechnungsabschnitt ermittelten Erwartungswert verwendet.

Geht man davon aus, daß ein Thread bei seinem Start keine Daten im Cache vorfindet, kann man ausgehend von diesem leeren Cachezustand den Erwartungswert nach dem ersten Schedulingabschnitt aus den aufgetretenen Cachemisses berechnen. Setzt man diesen Wert für i in die Alterungsfunktion für den Cachezustand ein, so kann zu jedem Zeitpunkt eine Aussage über den noch verbliebenen Rest gemacht werden. Beim nächsten Schedulingabschnitt wird dieser Rest dann wieder als Startwert für die Wachstumsfunktion eingesetzt.

Nimmt man nun analog der Arbeit von Thiebaut und Stone ([TS87]) an, daß ein Thread beim Beginn eines neuen Schedulingabschnittes seine Arbeitsmenge im Cache wieder vervollständigt, so läßt sich dieser *Reload Transient* durch die beiden Formeln abschätzen. Besaß ein Thread i am Ende des letzten Berechnungsabschnitts einen Erwartungswert von E_i Cachezeilen und wurden durch andere Threads vom Beginn seiner Wartezeit bis zum Zeitpunkt t, an dem sein nächster Berechnungsabschnitt beginnt, insgesamt $n_i(t)$ Cachemisses ausgelöst, so sind nach der Beziehung 4.9 noch etwa

$$E_i * \left(\frac{N-1}{N}\right)^{n_i(t)} \tag{4.10}$$

Cachezeilen vorhanden. Diese werden nun vom Thread wieder auf den ursprünglichen Wert E_i vervollständigt. Durch Einsetzen des eben ermittelten Rests in die Beziehung 4.7 und anschließendes Umformen nach der Anzahl notwendiger Cachemisses erhält man für den Reload

Transient $R(i, n_i(t))$ (mit der Konstante $K = (N-1)/N$):

$$R(i, n_i(t)) = log_K \frac{N - E_i}{N - K^{n_i(t)} E_i} \tag{4.11}$$

Reload Transient als inverse Priorität

Die einfachste Möglichkeit, den Reload Transient für die Prioritätsberechnung einzusetzen besteht darin, demjenigen Thread die höchste Priorität zu geben, der den kleinsten Reload Transient aufweist. Dadurch werden diejenigen Threads bevorzugt, die noch möglichst viel ihrer Arbeitsmenge im Cache vorliegen haben und damit ohne große Verzögerung durch weitere Cachemisses losrechnen können. Darüberhinaus erzeugt diese Strategie den kleinstmöglichen Einfluß auf die Cachezustände der übrigen Threads. Für zwei Threads i und j gilt also zum Zeitpunkt t:

$$P_i(t) > P_j(t) \leftrightarrow R(i, n_i(t)) < R(j, n_j(t)) \tag{4.12}$$

Auch bei diesem Prioritätsmaß handelt es sich wieder um ein zeitabhängiges, dynamisches Maß. Ähnlich den exponentiellen Alterungsfunktionen bei den zeitbasierten Strategien läßt sich hier wieder zeigen, daß die Ordnung innerhalb der Runqueue erhalten bleibt, obwohl sich die Prioritäten dynamisch ändern. Das bedeutet aber auch, daß der Thread mit minimalem Reload Transient nicht bei jeder Schedulingentscheidung gesucht werden muß. Stattdessen reicht es den Reload Transient zu berechnen, den ein eben deblockierter Thread erfahren würde, wenn er sofort wieder gestartet werden würde. Anhand dieses Wertes wird er in die Runqueue eingetragen. Ausschlaggebend ist also nicht der Reload Transient, der sich beim Start seines nächsten Abschnittes tatsächlich ergibt, sondern nur die Relation zu den anderen Threads der Runqueue.

Bei der Berechnung des Reload Transients wird man aus Effizienzgründen auf die genaue Auswertung der oben beschriebenen Beziehungen verzichten. Stattdessen reicht es, eine hinreichend genaue Tabelle zu verwenden, aus der die Prioritäten in Abhängigkeit von E_i und $n_i(t)$ entnommen werden können.

4.3 Datenstrukturen zur Unterstützung von Affinitätsbetrachtungen

Die hier vorgestellten Datenstrukturen stellen eine Ergänzung zu den im vorherigen Kapitel vorgestellten Grobstrukturen für MCS dar. Aufgrund des klareren Aufbaus und der unumstrittenen Vorteile wird hier nur noch auf den verteilten Ansatz mit lokalen Warteschlangen, statt auf zentrale Strukturen aufgesetzt.

In den Überlegungen zu MCS wurde festgestellt, daß lokale Runqueues helfen, die Lokalität der Berechnung zu erhalten und damit auch die Cacheausnutzung zu verbessern. Für die Auswertung von Affinitätsinformationen hat sich gezeigt, daß die Threads eines Prozessors anhand einer Priorität zu ordnen sind. Die lokale Runqueue muß also eine derartige Ordnung unterstützen. Darüberhinaus werden an die Warteschlange noch weitere Anforderungen gestellt.

- Sowohl das Einfügen eines Threads mit den dafür notwendigen Vergleichen und den evtl. dabei zu erfolgenden Berechnungen wie sie für dynamische Strategien auftreten, als auch das Suchen von Threads sollte möglichst effizient erfolgen. Die Struktur sollte also möglichst wenige Vergleiche erfordern.

- Das Finden und Entfernen des Threads höchster Priorität sollte schnell möglich sein.
- Für den Lastausgleich sollten ggf. auch Zugriffe auf das Element niedrigster Priorität möglich sein, ohne die gesamte Struktur durchsuchen zu müssen.
- Für die Flexibilität bzgl. verschiedener Affinitätsmaße wäre eine einfache Abbildung dieser Maße bzw. der daraus resultierenden Prioritäten auf die Struktur wünschenswert.
- Elemente derselben Priorität sollten ggf. in einer deterministischen Reihenfolge abgearbeitet werden, z.B. FIFO oder LIFO.
- Besonders in Systemen, bei denen sich die Anzahl von virtuellen Prozessoren während der Berechnung dynamisch ändern kann, sollte eine einfache Aufteilung einer Runqueue auf mehrere Prozessoren, bzw. ein einfaches Verschmelzen verschiedener Runqueues ohne großen Aufwand möglich sein.

Aufgrund ihres Einsatzes in vielen Algorithmen wurden im Laufe der Jahre viele Datenstrukturen und Algorithmen für Prioritätswarteschlangen entwickelt[Knu73], [OW93], [Bro77]). Sie lassen sich dabei grob in die folgenden Kategorien einteilen:

4.3.1 Listenbasierte Strukturen

Listenbasierte Strukturen bieten trotz der bekannten Nachteile, wie linearem Aufwand entweder beim Eintragen oder beim Suchen einige Vorteile, die sie für die Verwendung in Prioritätswarteschlangen interessant machen. Neben ihrem geringen Implementierungsaufwand und relativ kleinem Ressourcenverbrauch ist es vor allem die Einfachheit und Schnelligkeit der einzelnen Operationen, die sie bei kurzen Warteschlangen bis zu etwa 5 Elementen (siehe [Bro77]) anderen Strukturen überlegen sein lassen. Um sich diesen Vorteil auch bei größeren Anzahlen zu erhalten, kann man lineare Listen – wie in Betriebssystemen wie UNIX geschehen – in eine Reihe einzelner Listen aufteilen, die je einen Teil des Prioritätswertebereichs abdecken und damit von geringerer mittlerer Länge sind. Konzentrieren sich aber Threads mit vergleichbarer Priorität auf einen dieser Teile, so bringt die Aufteilung keine Vorteile mehr. Um diesem Problem aus dem Weg zu gehen, könnte eine dynamische Anpassung der Prioritätsgrenzen für eine gleichmäßigere Verteilung sorgen, wobei dann z.B. mittels binärer Suche die für einen Wert zuständige Warteschlange gefunden werden kann.

Somit bieten sich lineare Listen vor allem für Systeme an, bei denen mit einer geringen Anzahl lauffähiger Threads gerechnet werden kann. Sonst kann die Einführung einer Unterteilung in mehrere Warteschlangen einerseits zu einer geringeren Länge führen und außerdem zusätzlich noch Prioritätsmodelle mit wenigen Prioritäten, wie etwa durch die Prioritätsberechnung mittels Tabellen, unterstützen. Diese Prioritäten lassen sich ggf. direkt auf einzelne Warteschlangenteile abbilden.

4.3.2 Skiplisten

Im Vergleich zu geordneten Datenfeldern, in denen durch binäre Suche die Anzahl an Vergleichen auf $O(logN)$ reduziert werden kann, gibt es bei einfachen linearen Listen keine derartige Möglichkeit. Eine Erweiterung dieser Listen um Verkettungen, die nicht nur zum nächsten Element führen, sondern auch zu weiter entfernten Elementen, die sogenannten Skiplisten, lassen aber ein vergleichbares Zugriffsverhalten zu.

Skiplisten leiden aber unter dem Nachteil ihrer starren Struktur. Sobald durch eine Einfüge- oder Löschoperation in diese Struktur eingegriffen werden soll, muß sie vollständig neu erzeugt werden. Sie eignen sich damit hauptsächlich für statische Aufgaben, in denen sich die Datenmenge nicht verändert, sondern hauptsächlich durchsucht wird. Weniger anfällig für Änderungsoperationen sind die randomisierten Skiplisten. Bei ihnen wird die starre Festlegung für die Verkettung aufgeweicht. Erkauft wird dieser Vorteil dadurch, daß die logarithmischen Zugriffskosten nur noch im Mittel erreicht werden. Darüberhinaus benötigen Skiplisten im Vergleich zu normalen linearen Listen viel Speicher zur Verwaltung der notwendigen Zeiger. Gerade dieser Punkt stellt bei der Verwendung für Runqueues einen großen Nachteil dar, da mehr Speicher pro Strukturen mit großer Wahrscheinlichkeit auch zu mehr Cachemisses führt und damit die eigentliche Operation verlangsamt.

4.3.3 Baumbasierte Strukturen

Baumbasierte Strukturen bieten bei geringem Speicherbedarf im allgemeinen logarithmischen Aufwand bei den Operationen Einfügen, Suchen und Ersetzen von Elementen. Je nach Typ der Struktur wird dieser Aufwand entweder in jedem Fall oder nur im Mittel erreicht.

Einfache Binärbäume können den Aufwand nur im Mittel garantieren. Im schlimmsten Fall entarten sie zu einer linearen Liste und teilen sich deren Nachteile. Da aber spezielle Verwendungseigenschaften, wie etwa der ausschließliche Zugriff auf das größte oder kleinste Element bekannt sind, lassen sich zumindest die Entfernungsoperationen einfach optimieren.

Neben den einfachen Binärbäumen existieren viele Baumvarianten, die alle versuchen, die Höhendifferenz zwischen den Blättern klein zu halten. Beispiele dafür sind etwa AVL-Bäume, Bruderbäume oder balancierte Binärbäume. Zusätzlich dazu reduziert eine höhere Ordnung des Baums, wie etwa bei B-Bäumen, die Gesamthöhe des Baums.

Meist weisen diese Strukturen Eigenschaften auf, die über das hier benötigte Maß hinausgehen. Beispielsweise reduziert ein B-Baum im allgemeinen die Häufigkeit, mit der auf Hintergrundspeicher zugegriffen werden muß, indem immer mehr als ein Vergleichswert geholt wird. Je nach Komplexität der Vergleichsoperation, die wie gezeigt unter Umständen eine Berechnung erfordert, kann die dafür notwendige Zeit höher sein als bei einem Binärbaum mit nur einem Vergleich pro Knoten. Auch auf die Eigenschaft, Teile eines Baumes unabhängig vom Rest manipulieren zu können wird man im allgemeinen verzichten. Der Grund liegt dabei in dem dazu notwendigen, hohen Koordinierungsaufwand. Für den konkurrierenden Zugriff wären viele einzelne Koordinierungsaufrufe notwendig, die auf einer NUMA-Architektur mehr Zeit in Anspruch nehmen können, als der Rest der eigentlichen Operation. So kommt etwa auch Anderson et.al. in [ALL89] zu dem Schluß, daß ein Lock pro Struktur zu bevorzugen sei.

4.3.4 Heapbasierte Strukturen

Im Gegensatz zu Baumstrukturen garantieren heapbasierte Strukturen den Zugriff auf das Element höchster Priorität in konstanter Zeit. Das Entfernen dieses Elements kann dann in logarithmischer Zeit erfolgen. Bei der Suche nach dem Element niedrigster Priorität bleibt jedoch keine andere Möglichkeit, als die gesamte Struktur zu untersuchen. Damit scheiden heapbasierte Strukturen in Fällen, in denen dieser Zugriff benötigt wird üblicherweise aus.

Implementiert werden Heapstrukturen meist ebenfalls als Bäume oder Wälder, d.h. als verkettete Strukturen, um dem Nachteil fixer Größe der klassischen Implementierung als Feld zu entgehen. Typische Vertreter heapbasierter Strukturen sind Linksbäume, Binomialheaps und Fibonacciheaps. Eine Untersuchung verschiedener Prioritätswarteschlangen hat gezeigt, daß heapbasierte Strukturen allen anderen im Mittel überlegen sind, solange kein Zugriff auf ein anderes als das Elemente höchster Priorität notwendig ist ([Bro77]).

Kapitel 5

Koordinierungsmechanismen in Threadbibliotheken

Effiziente Koordinierungsmechanismen stellen neben der Threadverwaltung den zweiten, wichtigen Teil einer Threadbibliothek dar. Auch hier gilt wieder, daß die durch das Betriebssystem zur Verfügung gestellten Mechanismen für die Anwendung in feingranular parallelisierten Programmen zu viel Zeit benötigen, da mit jedem Betriebssystemeinsprung auch der Adreßraum gewechselt werden muß. Dementsprechend muß wieder ein Ersatz dafür im Benutzermodus zur Verfügung gestellt werden. Im allgemeinen sollte aber durch eine geeignete Schnittstelle dafür gesorgt werden, daß Schedulingentscheidungen des Betriebssystems Rücksicht auf die Koordinierungsmechanismen der Threadbibliothek nehmen. Die Verdrängung nur eines Kernelthreads, d.h. eines virtuellen Prozessors kann sonst die gesamte Anwendung lahmlegen. Für die Kommunikation mit dem Betriebssystemkern wurden in der Literatur verschiedene Verfahren vorgestellt, wie gemeinsame Speicherbereiche zwischen Betriebssystemkern und Benutzeradreßraum, Signale, durch die das Betriebssystem eine nahende Verdrängung mitteilen kann und andere.

Neben dem Problem eine geeignete Schnittstelle zwischen Betriebssystem und Threadbibliothek herzustellen, ergeben sich aus der Forderung nach geringen Kosten der Mechanismen und guter Skalierbarkeit, insbesonders auch auf NUMA-Architekturen mit hohen Prozessorzahlen, neue Kriterien für Implementierungen. Besonders auch der konkurrierende Zugriff auf den Speicherbereich von Koordinierungsmechanismen erzeugt auf diesen Architekturen Kosten, die nur durch aufwendigere Verfahren reduziert werden können. Ähnlich wie bei der Threadverwaltung stellt sich hier heraus, daß die bisher in Betriebssystemen verwendeten Datenstrukturen und Algorithmen dafür nur bedingt geeignet sind.

5.1 Wartemechanismen

Allen Koordinierungsmechanismen ist gemein, daß Threads unter gewissen Bedingungen – an einem gesetzten Lock, an einer Semaphore, o.ä. – warten müssen. Die Implementierung dieses Wartens spielt also eine zentrale Rolle in Koordinierungsmechanismen. Für sie existieren prinzipiell zwei Möglichkeiten: aktives Warten und Blockieren des Threads, d.h. Umschalten auf einen anderen Thread. In der Praxis sind sowohl diese Extreme, als auch verschiedene Zwischenstufen anzutreffen.

Always Spin

Bei dieser Methode wird aktiv auf die Erfüllung einer Bedingung, etwa dem Freiwerden eines Mutexlocks gewartet, d.h. die Bedingung wird andauernd überprüft. Dadurch entsteht nur eine minimale Verzögerung zwischen dem Eintreten dieses Ereignisses und dem tatsächlichen Weiterrechnen des bis dahin wartenden Threads. Außerdem bleibt, abgesehen von Invalidierungen durch andere Prozessoren, der Cachezustand des wartenden Threads erhalten, während dieser bei einer Blockierung zumindest teilweise zerstört werden kann. Die Strategie weist jedoch einige schwerwiegende Nachteile auf:

- Während des aktiven Wartens geht wertvolle Rechenleistung verloren.
- Sollte die Bedingung nur durch einen Thread erfüllt werden können, der nicht bearbeitet wird oder sollte er mit seinem virtuellen Prozessor verdrängt worden sein, so warten alle anderen Threads die gesamte Zeit, bis dieser Thread wieder weiterbearbeitet wird.
- Sobald mehr Threads als virtuelle Prozessoren existieren kann aktives Warten zu einer Verklemmung führen, wenn alle Prozessoren von wartenden Threads belegt sind. Andere lauffähige Threads, die die Bedingung ggf. erfüllen könnten, werden deshalb nicht mehr bearbeitet.

Neben diesen offensichtlichen Nachteilen entsteht durch das aktive Warten je nach Architektur eine nicht unwesentliche Busbelastung, die sich störend auf die Berechnung der anderen Prozessoren auswirkt und damit die Wartezeit künstlich verlängern kann.

Always Block

Diese Strategie stellt das entgegengesetzte Extrem zu Always Spin dar. Sobald ein Thread feststellt, daß er warten müßte, gibt er seinen Prozessor frei und trägt sich in eine Warteschlange ein, aus der er dann, nach Eintreten des geforderten Ereignisses, wieder entnommen und in die Runqueue seines Prozessors eingetragen wird. Obwohl in der Zwischenzeit ein anderer Thread berechnet werden kann und somit keine Verluste durch aktives Warten entstehen, erzeugt der zweimalige Threadwechsel des betroffenen Threads Kosten. Darüberhinaus kann die etwaige Verdrängung von Cachezeilen des blockierten Threads zu zusätzlichen Kosten führen.

Bei allen blockierenden Mechanismen stellt sich die Frage, wie bei der Erfüllung der Bedingung, z.B. dem Freigeben eines Locks verfahren werden soll. Einerseits können ein oder mehrere wartende Threads deblockiert werden und diese versuchen anschließend konkurrierend das Lock zu setzen, oder aber das Lock wird für einen einzigen deblockierten Thread reserviert. Die erste Version besitzt selbst unter der Annahme, daß nur ein Thread deblockiert wird den Nachteil, daß das Lock bis zum erneuten Berechnen des Threads bereits wieder gesetzt sein kann. Der Thread wird sich also erneut blockieren, wodurch wiederum die Kosten für den Threadwechsel entstehen. In manchen Systemen wird trotzdem komplett auf den Einsatz einer zusätzlichen Warteschlange verzichtet und der Thread sofort wieder in die Runqueue eingetragen. Er überprüft damit die Bedingung in jedem Berechnungsabschnitt erneut. Diese Technik wird in der Literatur auch als *Switch-Spinning* bezeichnet.

Die zweite Variante, das Lock für den deblockierten Thread zu reservieren vermeidet zwar die unnötigen Threadwechsel dieses Threads, erhöht aber dabei die Zeit in der kein anderer Thread auf dieses Lock zugreifen kann. Besonders in Systemen, in denen nicht für eine sofortige Berechnung des deblockierten Threads garantiert werden kann, scheidet diese Möglichkeit damit aus.

Spin-Blocking

Diese Strategie vereinigt die Vorteile des aktiven Wartens mit denen der sofortigen Blockierung. Dabei wartet ein Thread zuerst eine gewisse Zeit aktiv. Sollte nach Ablauf dieser Zeitspanne die Bedingung noch nicht erfüllt worden sein, dann wird der Thread blockiert und in eine Warteschlange eingetragen. Üblicherweise wird die Zeitspanne in der Größenordnung der Zeit für einen Threadwechsel gewählt.

Kennt man die mittlere Zeit, die ein Lock gesetzt bleibt, so kann dieser zweistufige Mechanismus weiter verfeinert werden. Sobald die mittlere Zeit den Aufwand für zwei Threadwechsel, also den Aufwand einer Blockierung überschreitet wird sofort blockiert. Im anderen Fall wird aktiv gewartet, wobei auch hier wieder ein Zeitlimit existiert, nachdem trotzdem blockiert wird. Die mittlere Wartezeit kann dabei aus den letzten Zugriffen auf das Lock angenähert werden. Insgesamt ergibt sich damit die nachfolgend gezeigte Grundstruktur[1]. Dabei wird die Konvention verwendet, daß ein freies Lock durch einen Wert ungleich Null, ein gesetztes Lock durch den Wert Null repräsentiert wird. Diese Werte ergeben sich durch die in vielen Systemen verwendeten, atomaren `test_and_set`- bzw. `load_and_clear`-Befehle.

```
void aquire_lock(lock_t *lock)
{
   int count;

   while(! TestAndSet(lock->addr))
   {
      if(lock->MeanTime > Threshold)
         block();
      else
      {
         count = 0;
         while((count < MaxSpins) && !TestAndSet(lock->addr))
            count++;
         if( count == MaxSpins)
            block();
         else
            break;
      }
   }
   lock->starttime = current_time();
   return;
}
```

[1] Bei diesem, wie auch bei den nachfolgenden Beispielen wurde auf den Einsatz einer Metasprache verzichtet. Stattdessen wurden sie in der allgemein bekannten Programmiersprache C formuliert

```
void release_lock(lock_t *lock)
{
   lock->MeanTime = evaluate_meantime(lock->starttime, current_time());
   *lock->addr    = 1
}
```

Obwohl dieser Algorithmus im schlechtesten Fall genauso hohe Kosten verursacht als Spin-Blocking mit einer festen Zeitspanne, kann bei Locks mit höherer Wartezeit auf das aktive Warten verzichtet und damit Rechenleistung anderen Thread zur Verfügung gestellt werden ([And90]).

5.2 Spinlocks als Implementierungsgrundlage blockierender Mechanismen

Bei allen blockierenden Mechanismen werden Threads im allgemeinen in Warteschlangen eingetragen. Um hier korrektes Verhalten zu garantieren, müssen die folgenden zwei Schritte atomar ausgeführt werden können:

1. Die Bedingung, die eventuell zur Blockierung führt, muß getestet werden.
2. Der Thread wird als blockiert gekennzeichnet, bzw. wird in eine Warteschlange eingetragen, falls die Bedingung nicht erfüllt war.

Solange diese beiden Schritte nicht direkt durch einen atomaren Befehl des Prozessors ausgeführt werden können, bleibt zu deren Sicherung nur die Verwendung von Spinlocks. Die dabei entstehenden Nachteile wiegen dabei wegen des meist nur kurzen kritischen Abschnitts wenig.

Als Grundlage von Spinlocks dienen die auf fast allen Prozessoren verfügbaren, atomaren Befehle wie *ldcws* (load and clear word short) der HP PA-RISC Prozessoren, *ldstub* (load and store unsigned byte) der SPARC-Prozessoren oder *xchg* (exchange) auf Intel Prozessoren. Alle Befehle haben gemeinsam, daß sie atomar eine Speicherzelle auslesen und verändern, ohne daß während ihrer Durchführung ein weiterer Zugriff erlaubt wäre. Liegt dieses Datum im Cache, bzw. hat man wie auf UMA-Architekturen einen relativ schnellen Zugriff auf den Hauptspeicher, so macht sich diese exklusive Belegung nur wenig bemerkbar. Schwerwiegender wird der Einfluß auf NUMA-Architekturen, wie der Convex SPP mit ihren hohen Zugriffszeiten und dem Einsatz von vielen Prozessoren aus mehreren Gründen:

- Durch den exklusiven Zugriff auf die Speicherzelle werden alle Zugriffe von Prozessoren serialisiert, d.h. hat ein Prozessor das Lock gesetzt und alle anderen anfragenden Prozessoren konkurrieren um den exklusiven Zugriff, so muß unter Umständen die Freigabe des Locks warten, bis alle bisher schon gestarteten atomaren Zugriffe seriell abgearbeitet wurden. Dadurch ergibt sich eine künstliche Verlängerung der Zeit für den kritischen Abschnitt.
- Alle vorher genannten Befehle haben gemeinsam, daß sie die Speicherzelle verändern. Damit erfahren alle Prozessoren, die auf das Datum zugreifen wollen einen Cachemiss. Bei der Betrachtung der Latenzzeiten von 2000 ns für entfernten Speicher auf der Convex

SPP wird klar, welche zusätzliche Verzögerung dadurch entsteht. Zusätzlich zur reinen Wartezeit während des Cachemisses trägt aber auch der meist hohe Aufwand zur Erhaltung der Cachekohärenz zu einer steigenden Busbelastung und dadurch zu einer eventuellen weiteren Verlängerung des kritischen Abschnittes bei.

Da die genannten Probleme bereits bei einer relativ kleinen Prozessorzahl zu nicht mehr tolerierbaren Leistungseinbußen führen, wurden im Laufe der Jahre verschiedene Methoden entwickelt, um das reine Spinning zu verbessern ([ALL89], [And90], [KLMO91], [MCS90]).

In der anschließenden Beschreibung einiger dieser Methoden wird der atomare Befehl *fetch_and_clear* verwendet, der den Inhalt der angegebenen Speicherzelle ausliest, auf Null setzt und den alten Inhalt als Ergebnis zurückliefert. Ein freies Lock wird dabei durch den Wert Eins, ein gesetztes Lock durch den Wert Null repräsentiert.

Einfaches Spinning

Beim einfachen Spinning, bei dem die oben genannten Probleme auftreten, wird mit Hilfe des atomaren Befehls andauernd versucht, das Lock zu setzen:

```
/* Lock setzen */
while(fetch_and_clear(lock_position)==0);

/* kritischer Abschnitt */

/* Lock freigeben */
*lock_position = 1;
```

Spin on Read, Snooping Locks

Bei dieser Variante wird auf den andauernden Einsatz des atomaren Befehls verzichtet. Stattdessen wird mit einem normalen Lesebefehl geprüft, ob das Lock freigegeben wurde; erst dann erfolgt der Setzversuch. Durch den Einsatz eines Lesebefehls fällt – zumindest auf Rechnern mit Datencaches – die Busbelastung weg, da die Lesebefehle ausschließlich durch die Caches befriedigt werden. Außerdem sollte das Lock jederzeit durch einen Schreibzugriff freigegeben werden können, ohne daß auf den Abschluß anstehender atomarer Befehle gewartet werden muß.

```
/* Lock setzen */
while((*lock_position==0) || (fetch_and_clear(lock_position)==0));

/* kritischer Abschnitt */

/* Lock freigeben */
*lock_position = 1;
```

Trotzdem besitzt auch dieser Algorithmus einen Nachteil, der ihn für die Verwendung bei hohen Prozessorzahlen ausscheiden läßt. Der schreibende Zugriff beim Freigeben des Locks sorgt für eine Invalidierung der zugehörigen Cachezeilen bei allen anderen Prozessoren. Beim lesenden Zugriff in der Schleife wird also bei jedem dieser Prozessoren ein Cachemiss ausgelöst. Aufgrund der hohen Latenzzeiten werden alle wartenden Prozessoren quasi synchron feststellen, daß das Lock freigegeben wurde und versuchen es mit dem `fetch_and_clear`-Befehl zu setzen, bevor alle bis auf einen wieder in ihre Leseschleife zurückkehren können. Einerseits wird also möglicherweise wieder die Freigabe des Locks durch die anstehenden atomaren Befehle verzögert, andererseits führt jeder dieser Befehle zu weiteren Cachemisses bei den Prozessoren, die bereits wieder zum lesenden Testen zurückgekehrt sind. Insgesamt kann durch diesen Algorithmus eine Welle von $O(n^2)$ Cachemisses ausgelöst werden, wenn n Prozessoren um das Lock konkurrieren.

Backoff Spinning

Die Nachteile des vorherigen Algorithmus wurden durch die Synchronisation ausgelöst, den die Cachemisses in der Leseschleife verursachen. Beim Backoff Spinning wird versucht, diese Synchronisation zu verhindern, bzw. deren Wahrscheinlichkeit zu verringern. Das Verfahren verwendet dazu eine Verzögerung zwischen den einzelnen Leseversuchen. Dadurch steigt die Wahrscheinlichkeit, daß das Lock bereits wieder gesetzt wurde, bis ein Thread einen erneuten Leseversuch unternimmt. Andererseits nimmt aber auch die Wahrscheinlichkeit zu, daß ein Thread das Lock setzen kann, wenn er es beim Lesezugriff frei aufgefunden hat.

```
/* Lock setzen */
while(fetch_and_clear(lock_position) == 0)
{
   while(1)
   {
      while(*lock_pos == 0);
      delay(waiting_period());
      if(*lock_pos != 0)
         break;
   }
}

/* kritischer Abschnitt */

/* Lock freigeben */
*lock_position = 1;
```

Die eingesetzten Verzögerungszeiten können entweder zufällige Werte annehmen oder nach festen Regeln berechnet werden. Als besonders geeignet hat sich dabei ein exponentielles Ansteigen dieser Zeiten herausgestellt. Durch den starken Anstieg der Zeiten werden die einzelnen Anfragen zeitlich schnell voneinander getrennt und damit eine Synchronisation wirkungsvoll verhindert. Um nicht zu große Verzögerungen entstehen zu lassen sollte aber bei dieser Strategie eine obere Schranke gesetzt werden. In Untersuchungen (siehe etwa [And90]) hat die gezeigte Verbindung von Lesezugriffen mit exponentiellem Backoff eine sehr gute Skalierbarkeit, auch für hohe Prozessorzahlen gezeigt. Als Nachteil dieser Methode kann aber angesehen werden, daß sie neu angekommene Threads bevorzugt, da diese eine noch kurze Wartezeit aufweisen, während länger wartende Threads immer weniger Gelegenheit bekommen, das Lock zu setzen.

Ticket Locks

Besonders auf NUMA-Architekturen, bei denen jeder Verlust einer Cachezeile unter Umständen einen zeitaufwendigen Zugriff auf entfernten Speicher bedeuten kann, stellen unnötige Versuche ein Lock mittels eines atomaren Befehls zu setzen ein Problem dar. Im Gegensatz zu allen bisherigen Varianten garantieren Ticket Locks, daß nur genau ein derartiger Befehl notwendig ist. Der Algorithmus ähnelt dabei dem bei Ämtern verwendeten Verfahren, bei denen jeder Besucher eine Nummer erhält, die anschließend irgendwann einmal aufgerufen wird.

Das Lock besteht hier aus zwei Speicherzellen, die möglichst auf zwei getrennte Cachezeilen abgebildet werden sollten. Die erste Speicherzelle enthält immer die nächste freie Nummer (`new_ticket`), die andere hingegen die Nummer, die gerade in Besitz des Locks ist (`served`). Ein Thread, der das Lock setzen will zieht eine Nummer, indem er mittels eines atomaren `fetch_and_inc`-Befehls die erste Speicherzelle erhöht und anschließend wartet, bis `served` dieser Nummer entspricht.

```
/* Lock setzen */
my_ticket = fetch_and_inc(new_ticket);
while(*served != my_ticket);

/* kritischer Abschnitt */

/* Lock freigeben */
(*served)++;
```

Ähnlich wie bei den vorherigen Algorithmen kann die Anzahl an Cachemisses, die in der Warteschleife auftreten dadurch verringert werden, daß nicht andauernd abgefragt wird, sondern zwischen jedem Test eine gewisse Zeit gewartet wird. Statt einer exponentiellen Zeit wird hier ein Vielfaches der Differenz zwischen `my_ticket` und `served` verwendet.

Queued Locks

Alle bisher gezeigten Varianten hatten immer eine Speicherzelle gemeinsam, an der sie alle warteten. Das bedeutet aber, daß die Freigabe des Locks bei jedem der wartenden Prozessoren einen Cachemiss auslöst. Die Erhöhung der für eine Anwendung zur Verfügung gestellten Prozessoren erhöht also mindestens linear die Anzahl an Cachemisses und verringert damit den erreichbaren Speedup.

Um diesem Verhalten aus dem Weg zu gehen, muß auf die gemeinsame Speicherzelle verzichtet werden. Stattdessen erhält jeder Prozessor, bzw. jeder Thread eine eigene Speicherzelle, an der er wartet. Wie die Bezeichnung dieser Variante schon andeutet, werden die einzelnen Anforderungen hier in einer Warteschlange verwaltet. Für die Manipulation dieser Warteschlange muß aber dann die Hardware geeignete atomare Befehle zur Verfügung stellen. Die nachfolgende Implementierung aus [MCS90] verwendet dafür die Befehle `fetch_and_store` und `compare_and_store`:

fetch_and_store(p,i) ersetzt den Wert der durch p bezeichneten Speicherzelle durch i und liefert den alten Wert zurück.

compare_and_store(p,c,i) vergleicht den Wert der durch p bezeichneten Speicherzelle mit c. Bei Übereinstimmung wird der Wert durch i ersetzt und ein Wert ungleich Null zurückgeliefert. Sonst ändert sich an dem Wert der Speicherzelle nichts und der Befehl kehrt mit dem Ergebnis Null zurück.

Jede Anforderung des Locks besteht jetzt aus einer eigenen Struktur (I), aufgebaut aus einem Zeiger auf die nächste dieser Anforderungen in der Warteschlange und einer privaten Variable, an der der Prozessor anschließend warten kann. Das Lock besteht nur noch aus einem Zeiger (Lock_ptr) auf derartige Strukturen. Der besseren Übersicht wegen wurden hier die beiden Operation zum Setzen und Freigeben des Locks als Funktionen realisiert:

```
/* Struktur der Lockanforderungen */
struct
{
  int            wait;
  struct node_t *next;
}node_t;

/* Funktion zum Anfordern des Locks */
void aquire_lock(struct node_t *Lock_ptr, struct node_t *I)
{
  node_t *old;

  I->next = NULL;
  old = fetch_and_store(Lock_ptr,I);
  if(old != NULL)                /* die Warteschlange war nicht leer     */
  {
    I->wait   = 1;
    old->next = I;               /* I in Warteschlange eintragen         */
    while(I->wait);              /* warten, bis Lock freigegeben wird    */
  }
}

/* Funktion zum Freigeben des Locks */
void release_lock(struct node_t *Lock_ptr, struct node_t *I)
{
  if(I->next == NULL)           /* es gibt noch keinen Nachfolger       */
  {
    if(compare_and_store(Lock_ptr, I, NULL)) /* Struktur austragen      */
      return;
    else                        /* es wartet bereits ein anderer Thread */
      while(I->next == NULL);   /* warten, bis er sich eingetragen hat  */
  }
  I->next->wait = 0;            /* Lock freigeben                       */
}
```

5.3 Datenstrukturen für blockierte Threads

Um blockierte Threads beim Eintreffen des erwarteten Ereignisses wieder deblockieren zu können, müssen sie in einer geeigneten Weise mit dem Koordinierungsmechanismus verbunden sein. Ähnlich wie bei der Verwaltung lauffähiger Threads gelten für die dabei verwendeten Warteschlangenstrukturen Forderungen nach schnellem Zugriff, hohem Parallelitätsgrad und Berücksichtigung der Lokalität der dabei benötigten Daten.

Zentrale Sleepqueue

In klassischen Betriebssystemen wie etwa UNIX wird eine zentrale Sleepqueue verwendet, in die blockierte Prozesse eingetragen werden. Jeder der eingetragenen Prozesse erhält dabei einen für den Mechanismus charakteristischen Wert, wie etwa die Adresse des für den Mechanismus verwendeten Speicherbereichs. Anhand dieses Wertes kann bei der gewünschten Deblockierung eines Prozesses die Sleepqueue durchsucht werden. Um einerseits die Suche zu beschleunigen und andererseits bei einem Mehrrechnersystem einen parallelen Zugriff auf diese Struktur zu ermöglichen, hat man auch hier meist mehrere Warteschlangen anstatt einer verwendet, die jeweils durch ein eigenes Lock geschützt werden. Die Abbildung des bei der Blockierung angegebenen Wertes auf eine der Warteschlangen wird mittels einer geeigneten Hashfunktion vorgenommen. Abbildung 5.1 zeigt schematisch den Aufbau eines derartigen Systems mit zentralen Warteschlangen für die Run- und Sleepqueue.

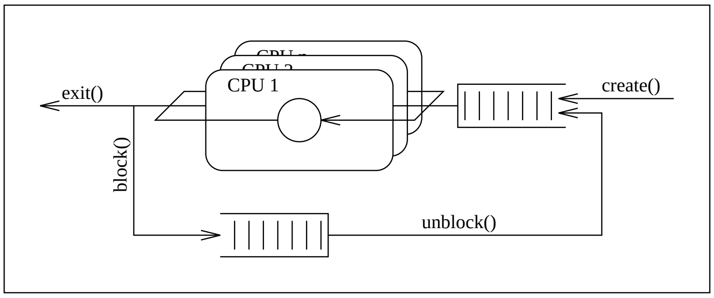


Abbildung 5.1: Verwendung von zentralen Sleep- und Runqueues

Dem Vorteil ihrer einfachen Implementierbarkeit stehen die Probleme beim Einsatz in Parallelrechnern, insbesonders mit NUMA-Architektur gegenüber. Selbst bei Verwendung einer aufgeteilten Sleepqueue kann diese Struktur zu einem Flaschenhals werden. Vor allem wenn beim Benützen der Struktur Zugriffe auf entfernten Speicher notwendig werden, verschlimmert sich dieser Effekt, da jeder Zugriff länger dauert.

Lokale Sleepqueues

Statt einer zentralen Struktur könnte man nun wie bei den Runqueues lokale Warteschlangen einführen – entweder pro Hypernode oder pro Prozessor. Dadurch würde einerseits der maximale Parallelitätsgrad steigen, da jetzt mehrere Prozessoren gleichzeitig auf diesen Strukturen arbeiten können und andererseits lägen die Strukturen immer im lokalen Speicher. Abbildung 5.2 zeigt Sleepqueues lokal zu jedem Prozessor.

Problematisch wird dieses Verfahren, wenn sich Threads verschiedener Prozessoren oder Hypernodes an einem Mechanismus blockieren. Bei einer Deblockierung müssen alle beteiligten Sleepqueues untersucht werden. Zwar können Hinweise auf die zu untersuchenden Warteschlangen beim Koordinierungsmechanismus, etwa in Form von gesetzten Bits hinterlegt

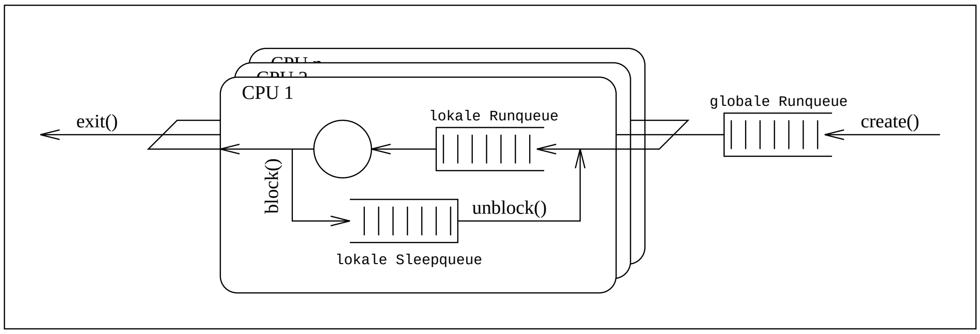


Abbildung 5.2: Lokale Sleepqueues für jeden Prozessor

werden, was bleibt ist aber der hohe Suchaufwand. Darüberhinaus geht damit auch wieder die Lokalität des Zugriffs verloren, da unter Umständen auch Warteschlangen entfernter Hypernodes in die Suche miteinbezogen werden müssen.

Sleepqueues pro Koordinierungsmechanismus

Komplett verzichten kann man auf die Suche nach zu deblockierenden Threads, wenn man die Warteschlange direkt an den Mechanismus bindet. Jeder Mechanismus weist eine private Sleepqueue auf, in die alle an ihm blockierten Threads eingetragen werden. Wird ein Thread deblockiert, so wird er daraus entfernt und wieder in die zentrale bzw. eine lokale Runqueue eingefügt. Vor allem auf Systemen wie der Convex SPP, mit ihren Netzwerkcaches hat diese Methode den Vorteil, daß der Mechanismus mitsamt seiner Warteschlange quasi auf den Hypernode migriert, auf dem er verwendet wird. Solange dann nur dort darauf zugegriffen wird, erscheint der Mechanismus lokal zu liegen. Dieses Verhalten wäre mit einer zentralen Sleepqueue nicht möglich, da hier immer alle Prozessoren darauf zugreifen und die entsprechenden Strukturen andauernd migrieren würden.

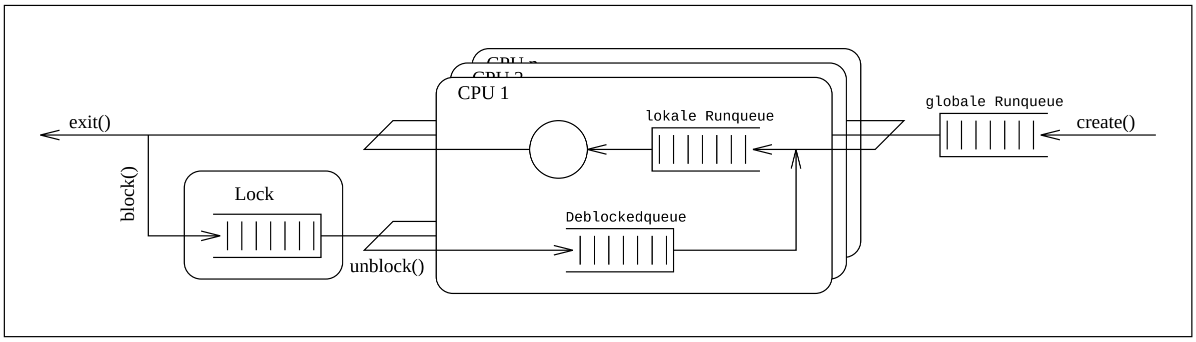


Abbildung 5.3: Lokale Sleepqueues pro Koordinierungsmechanismus

Vor allem in Systemen mit lokalen Runqueues ergibt sich bei der Deblockierung noch eine weitere Verbesserungsmöglichkeit. Werden Affinitätsbetrachtungen für Schedulingentscheidungen verwendet, so weist die Runqueue jedes einzelnen Prozessors eine relativ aufwendige Struktur auf. Greift ein deblockierender Prozessor im gegenseitigen Ausschluß darauf zu, so können sehr hohe Zugriffszeiten entstehen, die den anderen Prozessor bei seiner Arbeit behindern. Statt nun die Threads direkt in die Runqueue einzutragen, können diese vom deblockierenden Prozessor auch in eine einfachere, lokale Struktur – einen Stack oder eine lineare Liste – beim Zielprozessor eingetragen werden. Daraus werden sie dann vom Zielprozessor entnommen und anhand ihrer Affinität bzw. Priorität in die lokale Runqueue übernommen. Dadurch läßt sich einerseits der Deblockierungsvorgang vollständig von anderen Schedulingentscheidungen entkoppeln und andererseits wird die Warteschlange des Koordinierungsmechanismus schneller wieder zugänglich, da die Deblockierung weniger Zeit kostet. Die Struktur, die sich daraus ergibt ist schematisch in Abbildung 5.3 zu sehen.

Kapitel 6

Die MThreads Threadbibliothek

Im Rahmen des ELiTE-Projekts[1] entstand eine Threadbibliothek in der mehrere der Konzepte der letzten Kapitel realisiert wurden. Nach einem Überblick über den internen Aufbau der Threadbibliothek wird anhand einiger Testapplikationen der Einfluß der vorgestellten Strategien für MCS und zusätzlicher Affinitätsbetrachtungen aufgezeigt.

6.1 Interne Struktur der Threadbibliothek

Die MThreads Bibliothek wurden speziell für den Einsatz auf NUMA-Architekturen mit einer großen Anzahl an Prozessoren entwickelt. Als Zielarchitektur stand dabei die anfangs beschriebene Convex SPP mit mehreren Subkomplexen von bis zu 32 Prozessoren am Regionalen Rechenzentrum Erlangen zur Verfügung.

Die eigentliche Threadumschaltung innerhalb der Bibliothek basiert auf den *Quickthreads* ([Kep93]) von David Keppel. Dieses Paket stellt grundlegende Mechanismen zur Verfügung um Threads zu erzeugen, zwischen Threads umzuschalten und wieder zu terminieren, wobei diese ausschließlich durch ihren Stackpointer repräsentiert werden. Das Quickthreads-Paket wurde im Rahmen einer Studienarbeit von Uwe Reeder ([Red95]) auf die Convex SPP portiert.

Abbildung 6.1 zeigt die interne Struktur der darauf aufbauenden Threadbibliothek. Sie orientiert sich in ihrem Aufbau stark an den Hierarchiestufen der verwendeten Convex SPP. Für jede Stufe, vom Subkomplex über die Nodes zu den einzelnen Prozessoren existieren eigene Strukturen. Diese Strukturen dienen eigentlich nur dazu die Komponenten des Systems zu verbinden, d.h. die eigentlichen Algorithmen sind nicht von der tatsächlichen Struktur des Rechners abhängig, sondern benutzen die Informationen in den logischen Strukturen der Hierarchiestufen für ihre Entscheidungen, wobei mindestens eine Struktur pro Hierarchiestufe tatsächlich existieren muß.

6.1.1 Die Threadverwaltungsstrukturen

Die virtuellen Prozessoren

Das Threadpaket erzeugt beim Start durch die Funktion `mthr_startup()` einen Kernelthread für jeden Prozessor des Subkomplexes, solange keine weiteren Einschränkungen durch den Benutzer vorgenommen wurden. Jeder dieser Kernelthreads – auch derjenige des ursprünglich

[1] **E**rlangen **Li**ghtweight **T**hread **E**nvironment

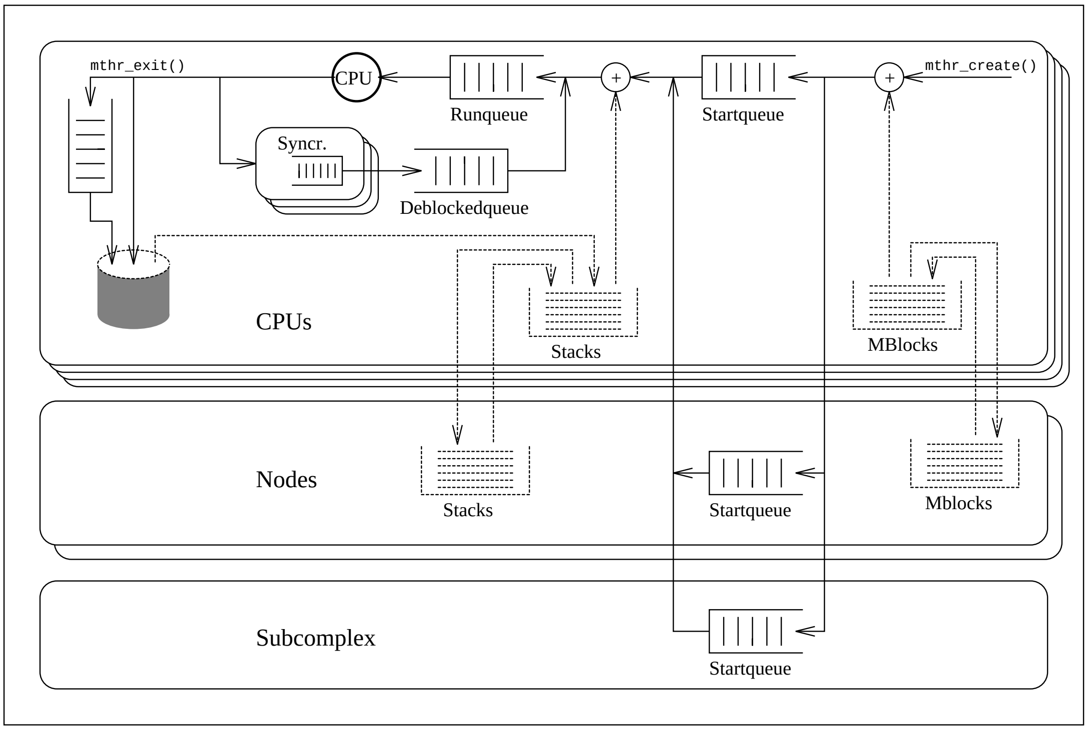


Abbildung 6.1: Struktur der MThreads-Bibliothek

startenden UNIX-Prozesses – wird durch eine **vcpu_t** Struktur repräsentiert. Sie beinhaltet hauptsächlich Referenzen auf prozessorlokale Strukturen, wie die lokale Start-, Run- und Deblockedqueue und lokale Freilisten für häufig benötigte Strukturen wie Stacks, Threadstrukturen usw.

Sobald sich ein Thread blockieren oder terminieren will, sucht der Prozessor nach einem geeigneten Nachfolgerthread. Dabei entnimmt er wenn möglich einen lauffähigen Thread aus seiner lokalen Runqueue. Sollte der Prozessor keinen lauffähigen Thread finden, so entnimmt er einen startfähigen Thread aus seiner lokalen Startqueue, stattet ihn mit einem Stack aus und läßt ihn durch das Quickthreads-Paket initialisieren. Anschließend wird auf diesen Thread umgeschaltet. Sollte auch die lokale Startqueue des Prozessors leer sein, so wird zuerst die Startqueue des Nodes, anschließend die des Subkomplexes und zuletzt die Startqueues der anderen Prozessoren des Hypernodes untersucht. Erst wenn auch diese Suche fehlschlägt, wird auf einen privaten Idlethread umgeschaltet, der für den dann notwendigen Lastausgleich sorgt.

Die Nodestruktur

Ebenso wie für jeden Prozessor im System existiert für jeden Node eine eigene Struktur. Auch sie enthält wieder eine Startqueue für neu erzeugte Threads, die nun nur einem Node, nicht einem speziellen Prozessor zugeordnet wurden und Freilisten für verschiedene Strukturen.

Die Nodestruktur dient aber hauptsächlich der Verwaltung der CPU-Strukturen dieses Nodes. Sie enthält sowohl eine Liste aller dieser `vcpu_t` Strukturen, sowie einen Zähler und eine Warteschlange für blockierte Kernelthreads. Sobald sich beim Lastausgleich durch den Idlethread eines Prozessors kein neuer Thread mehr findet, wird der zugehörige Kernelthread blockiert, wobei wieder ein zweistufiges Spin-Blocking Verfahren verwendet wird. Sollte sich der letzte Kernelthread des Nodes blockieren, wird dies mittels eines nodeinternen Zählers festgestellt und an die Subkomplexstruktur weitergeleitet.

Die Subkomplexstruktur

Analog den Nodestrukturen enthält die Subkomplexstruktur eine Liste aller Nodes und einen Zähler für vollständig blockierte Nodes. Sollte sich der letzte Node als blockiert melden, wird die Anwendung beendet, da dann kein lauf- bzw. startfähiger Thread mehr existiert. Darüberhinaus findet sich auch hier eine Startqueue – jetzt für Threads, die an keinen Node oder Prozessor gebunden, sondern als allgemein migrierbar gekennzeichnet wurden.

Die lokalen Runqueues

Jeder Prozessor besitzt eine lokale Runqueue mit lauffähigen Threads. Aufgrund der Anforderungen an Runqueues, wie schneller Zugriff, gute Abbildung unterschiedlicher Affinitätsmaße und leichte Teilungs- und Verschmelzungseigenschaften wurde ein Binärbaum als Grundstruktur gewählt. Dieser besitzt aber nicht nur wie üblich einen Zeiger auf die Wurzel, sondern auch noch Zeiger auf das minimale und maximale Element. Da nur auf diese beiden Elemente direkt zugegriffen werden muß, konnten die Algorithmen für das Entfernen von Elementen stark vereinfacht werden.

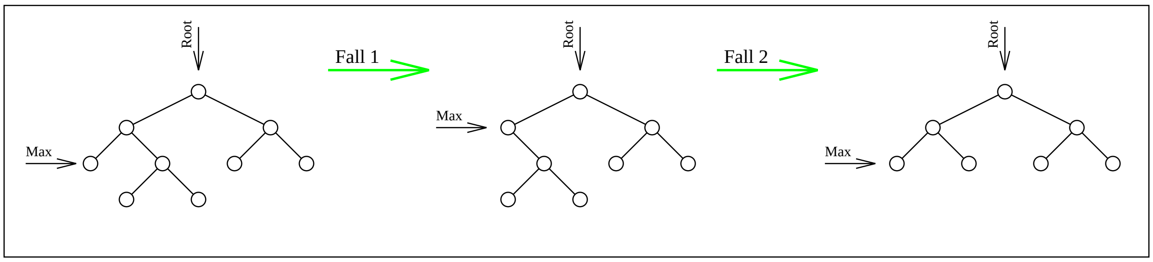


Abbildung 6.2: Zugriff auf das höchstpriore Element der Runqueue

Soll etwa das maximale Element entfernt werden und handelt es sich um ein Blatt (siehe Abbildung 6.2, Fall 1), dann wird es entfernt und der Verweis auf das maximale Element auf den Vorgängerknoten gesetzt. Sollte es sich bei dem maximalen Element um einen Knoten handeln, so kann es an maximal einem nichtleeren Ast einen Teilbaum besitzen, durch dessen Wurzel das es ersetzt wird. Anschließend wird wieder solange nach unten gelaufen, bis wieder das maximale Blatt erreicht ist (Abbildung 6.2, Fall 2). Man erkennt, daß selbst bei einem entarteten Baum im Mittel nur wenige Operationen ausgeführt werden müssen. Selbst wenn die entstandene Liste komplett durchlaufen werden muß, um wie in Fall 2 das maximale Blatt zu finden, so können doch anschließend alle weiteren Entfernungen in konstanter Zeit erfolgen.

Die Startqueues

Zusätzlich zu den sonst in Betriebssystemen oder Threadbibliotheken vorzufindenden Prozeß- bzw. Threadzuständen, wie lauffähig, blockiert, laufend usw. existiert hier ein weiterer Zustand für Threads. Neu erzeugte Threads werden im Zustand `STARTABLE` in eine Startqueue eingetragen. Startfähige Threads stellen ein, aus einer Threadstruktur bestehendes Gerüst dar, das die gesamte zur tatsächlichen Instanziierung notwendige Information enthält.

Bei der Erzeugung eines neuen Threads durch den Aufruf `mthr_create()` kann der Benutzer spezifizieren, wo der Thread berechnet werden soll. Zur Auswahl steht dabei eine Bindung an einen bestimmten virtuellen Prozessor, einen bestimmten Node oder die freie Migrierbarkeit über den gesamten Subkomplex. Verzichtet der Benutzer auf eine spezielle Zuordnung, so wird der neue Thread standardmäßig in die Startqueue des erzeugenden Prozessors eingetragen. Realisiert wird diese Bindung durch eine Hierarchie an Startqueues, die von Prozessoren zum Finden eines Nachfolgerthreads für einen terminierenden oder blockierenden Thread abgesucht wird. Neben dieser statischen Zuordnung können startfähige Threads mit dem Befehl `mthr_prefetch_thread()` zur Laufzeit in die Startqueue des aufrufenden Prozessors migriert werden.

Threads werden erst dann aus einer Startqueue entnommen, wenn kein lauffähiger Thread mehr in der lokalen Runqueue existiert. Damit wird sichergestellt, daß immer nur die absolut notwendige Zahl an Threads gleichzeitig bearbeitet wird und damit weniger Beeinflussungen untereinander in den Prozessorcaches entstehen. Außerdem steigt bei dieser Strategie die Wahrscheinlichkeit, daß beim Start eines Threads ein nicht mehr benutzter Stack in einer lokalen Freiliste gefunden werden kann. Gerade auf Systemen wie der Convex kann dieses Wiederbenutzung von Stacks die notwendigen Kosten stark reduzieren. Dazu zählt nicht nur die Zeit zum Anfordern des Speichers mittels einer Bibliotheksfunktion, sondern vor allem die Zeit die das darunterliegende MACH-Betriebssystem zum Anlegen des Speichers beim ersten Zugriff benötigt. Abgesehen von den gesparten Kosten für das Anlegen eines neuen Speicherbereichs kann bei der Wiederbenutzung von Stacks auch noch Nutzen aus einer unter Umständen verbliebenen Abbildung dieses Speicherbereichs in den Cache des zugreifenden Prozessors gezogen werden.

Freilisten

Jeder Prozessor besitzt Freilisten für häufig verwendete Strukturen. Dazu zählen neben Stacks vor allem Threadstrukturen und Speicherbereiche für Koordinierungsmechanismen. Neben dem Vorteil entfallender Speicheranforderungen und einer besseren Cacheausnutzung besteht ein Hauptvorteil auch in der durch die Freilisten garantierten Ausrichtung dieser Strukturen im Speicher. So sind etwa alle Elemente der Freiliste für Threadstrukturen an 64 Bytegrenzen ausgerichtet um eine Abbildung der Threadstrukturen auf eine CTI-Cachezeile sicherzustellen.

Strukturen werden innerhalb der Freilisten in LIFO-Ordnung verwaltet, um immer diejenigen Strukturen zuerst zu verwenden, die noch die höchste Affinität zum Prozessorcache aufweisen. Sollte eine der Listen für kleine Strukturen überlaufen, so wird die Hälfte ihrer Elemente in die globalere Freiliste des Nodes übernommen. Entsprechend wird aus dieser globaleren Liste eben diese Anzahl an Elementen angefordert, sollte die lokale Freiliste leer sein. Im Gegensatz dazu wird bei den Freilisten für Stacks immer nur ein Element an den globalen Pool abgegeben, bzw. von dort angefordert.

6.1.2 Die Koordinierungsmechanismen

Die Threadbibliothek bietet zur Koordinierung Mutexlocks, Semaphoren, Barrieren und Bedingungsvariablen an. Alle dieser Mechanismen bauen intern auf einer speziellen Form von Bedingungsvariablen auf, die eine private Warteschlange für blockierte Threads verwaltet. Bei der Blockierung an dieser internen Bedingungsvariable wird ein zweistufiges Spin-Blocking Verfahren verwendet, wobei beim Blockierungsaufruf die vermutete Wartezeit angegeben werden kann. Aufgrund dieser Information wird entschieden, ob sich aktives Warten lohnt, oder ob sofort blockiert werden soll. Wird etwa ein Mutexlock erfolgreich gesetzt, so liest es eine hochauflösende Systemuhr aus und speichert diesen Wert. Bei der Freigabe des Locks kann damit die Zeit ermittelt werden, die das Lock gesetzt war. Mittels einer einfachen Alterungsfunktion wird aus dem bisher erwarteten Wert für die Dauer und dem eben ermittelten ein neuer Wert berechnet, der für die Entscheidung zwischen aktivem Warten oder Blockieren herangezogen werden kann.

Werden Threads deblockiert, so trägt sie der deblockierende Thread in die Deblockedqueues der Prozessoren ein, von denen sie vorher berechnet wurden. Bei den Deblockedqueues handelt es sich um einfache lineare Listen, die eine schnelle Einfügeoperation zulassen. Sobald ein Prozessor nach einem neuen Thread sucht, entnimmt er alle Threads aus seiner Deblockedqueue, berechnet aus deren Affinitätsinformation ihre Priorität und trägt sie in seine lokale Runqueue ein.

Für die notwendige Atomarität bei der Manipulation der einzelnen Warteschlangen und Strukturen sorgen Spinlocks, die aus Effizienzgründen in Maschinensprache implementiert wurden, da sie die am häufigsten verwendeten Funktionen der Threadbibliothek darstellen. Um eine hinreichend gute Skalierbarkeit sicherzustellen wurden dabei Snooping Locks mit exponentiellem Backoff und oberer Grenze für die Wartezeit zwischen den einzelnen Lesezugriffen verwendet.

6.1.3 Lastausgleich zwischen den Prozessoren

Sollte ein Prozessor bei der Suche nach einem startfähigen Thread innerhalb seines Hypernodes nicht fündig werden, so schaltet er auf einen privaten Idlethread um, der dann den Lastausgleich übernimmt. Um nicht bei einem Mangel an Threads, etwa am Ende eines Programmlaufes, die restlichen Berechnungen durch eine Vielzahl an suchenden Prozessoren zu belasten, wird maximal einem Prozessor pro Hypernode Lastausgleich erlaubt. Alle anderen müssen gegebenenfalls warten.

Der Lastausgleichsalgorithmus arbeitet auf zwei Ebenen. Zuerst versucht er die startfähigen Threads umzuschichten, da diese im Vergleich zu lauffähigen Threads noch kaum Affinität zu einem Prozessor aufweisen. Dazu untersucht er die Startqueues aller Prozessoren, entnimmt jeder nichtleeren Warteschlange einen Thread und überträgt ihn in die Startqueue der Nodestruktur dieses Prozessors. Anschließend entnimmt er aus den Startqueues jeder Nodestruktur je einen Thread und übergibt ihn der Startqueue des Subkomplexes. Dadurch wird sichergestellt, daß sich der Lastausgleich für alle Prozessoren etwa gleich auswirkt und trotzdem anschließend mehr als ein Thread zur Verfügung steht, um nicht sofort wieder Anlaß für eine erneute Suche zu haben. Die durch den Benutzer getroffene Zuordnung von Threads zu Prozessoren oder Nodes wird außerdem durch dieses Verfahren immer nur in geringem Umfang gestört.

Sollte kein startfähiger Thread mehr auffindbar sein, d.h. die Startqueue des Subkomplex nach dieser Suche immer noch keinen Thread enthalten, so geht der Algorithmus in die zweite Phase über und sucht einen lauffähigen Thread. Dazu untersucht er, beginnend bei seinem eigenen Node alle nichtleeren Runqueues. Von den Threads niedrigster Priorität in diesen Warteschlangen sucht er sich denjenigen aus, von dem er die größte Verbesserung erwartet. Dabei verwendet der Prozessor Information über die zu erwartende Rechenzeit aller Threads innerhalb der jeweiligen Runqueue, die erwartete Rechenzeit des Threads niedrigster Priorität und dessen Affinität. Migriert werden nur Threads, bei denen die Summe aus Rechenzeit und Zeit für die Cachemisses zum Aufbau der Arbeitsmenge des Threads auf dem neuen Prozessor kleiner ist als die Rechenzeit aller Threads auf dem alten Prozessor, inklusive des untersuchten.

Erst wenn auf dem lokalen Node kein lauffähiger Thread gefunden, bzw. bei keinem Thread eine Verbesserung erwartet werden kann, wiederholt der Prozessor den Vorgang auf einem anderen Node, wobei dann auch die Berechnung der Verluste durch die Cacheladezeiten entsprechend angepaßt erfolgt. Auch damit werden wieder lokale Migrationen, d.h. innerhalb des Nodes bevorzugt, um damit Nutzen aus der Lokalität der Daten im Hauptspeicher des Nodes bzw. der Affinität des Threads zum Netzwerkcache zu ziehen.

Sollte kein lauffähiger Thread gefunden werden können, so wird der virtuelle Prozessor in eine Warteschlange beim Node eingetragen und der zugehörige Kernelthread blockiert.

Ermittlung und Auswertung der Affinität

Um die Untersuchung verschiedener Strategien bei der Ermittlung der Affinität von Threads zu Prozessoren und deren Umsetzung in eine Priorität zu unterstützen, wurden diese Aufgaben in Form eines eigenständigen Moduls modelliert, das innerhalb der Bibliothek jederzeit durch ein anderes ersetzt werden kann.

Das Modul besteht aus drei Funktionen, von denen zwei der Ermittlung der Affinität dienen, während die dritte die Umsetzung in eine Priorität bzw. den Vergleich der sich ergebenden Prioritäten zweier Threads realisiert.

aff_thread_stopped() wird durch die Bibliothek aufgerufen wenn sich ein Thread blockiert oder terminiert, um die Affinitätsmessung für diesen Thread zu beenden.

aff_thread_started() wird im Anschluß daran für den Nachfolgerthread aufgerufen, um für diesen die Affinitätsmessung zu starten.

aff_greater() legt die Ordnung innerhalb der Runqueue fest, d.h. wenn beim Eintragen eines neuen Threads in die Runqueue ein Vergleich zwischen zwei Threads erfolgt, wird diese Vergleichsfunktion herangezogen. Werden innerhalb dieser Funktionen die Prioritäten der Threads berechnet, dann können damit auch die vorgestellten dynamischen Maße realisiert werden.

Für die Messung der Cachemisses stellt die Convex SPP Zähler zur Verfügung, mit denen man entweder die Cachemisses im Datencache bzw. im Netzwerkcache oder deren Summe messen kann. Eine getrennte Messung hingegen ist bei dem eingesetzten Modell SPP 1000 noch nicht möglich. Auf der Basis dieser Meßmöglichkeiten wurden folgende Strategien implementiert und getestet:

VTime: Diese Strategie verwendete eine virtuelle Zeit, indem bei jedem Aufruf der Funktion `aff_thread_stopped()` ein Zähler beim Prozessor inkrementiert und sein Wert als Priorität des Threads interpretiert wird.

CMisses: Hier werden die innerhalb des Berechnungsabschnittes aufgetretenen Cachemisses gezählt und als Priorität verwendet, d.h. es wird davon ausgegangen, daß ein Thread mit vielen Cachemisses einen großen Teil seiner Arbeitsmenge in den Cache geladen hat und deshalb vorrangig zu berechnen sei.

CmSum: Anstatt der Cachemisses nur eines Berechnungsabschnittes wird hier die Summe aller bisherigen Abschnitte verwendet, um damit die Berechnungsgeschichte des Threads mit in Prioritätsentscheidung einfließen zu lassen.

Reload: Bei dieser Strategie erhält der Thread die höchste Priorität, der nach dem vorgestellten mathematischen Modell die wenigsten Cachemisses bei seinem Start erwarten läßt. Dazu wird beim Starten und Stoppen eines Threads jeweils der Erwartungswert für seine Arbeitsmenge berechnet und beim Aufruf von `aff_greater()` in die zu erwartenden Reloadzeiten umgesetzt, wobei diese Werte einer Tabelle entnommen werden.

6.2 Resultate verschiedener Beispielanwendungen

Um den Einfluß von MCS und den verschiedenen Strategien für Affinitätsbetrachtungen sichtbar zu machen, wurden zwei numerische Verfahren in einfacher Weise mittels der Threadbibliothek parallelisiert. Neben diesen beiden Anwendungen wurden außerdem verschiedene synthetische Testprogramme auf einem aus 4 Hypernodes, d.h. 32 Prozessoren bestehenden Subkomplex der Convex SPP des Regionalen Rechenzentrums in Erlangen getestet.

Einige Eigenschaften des verwendeten Rechners haben sich dabei als hinderlich beim Einsatz von MCS, bzw. allgemein bei der Verteilung der Berechnung herausgestellt. Zum einen ist es nicht möglich die Lokalität angeforderten Speichers festzustellen. Speicher, den man mittels der Bibliotheksfunktion `malloc()` anfordert, wird standardmäßig als *FAR_SHARED* Memory in Seiten von 4 KByte auf die Hypernodes des gesamten Subkomplexes verteilt. Daraus ergeben sich mehrere Nachteile:

- Da der Ort des Speichers nicht direkt ermittelt werden kann, wird eine Zuordnung von Threads zu bestimmten Hypernodes anhand von Lokalitätsinformation erschwert, wenn nicht unmöglich. Ausgenützt werden kann nur die Lokalität zu den Abbildungen dieses Speichers in die Netzwerkcaches.

- Ähnlich der Verteilung der Daten müßte auch eine Verteilung der Berechnung auf alle Hypernodes des Systems erfolgen, unabhängig davon, daß unter Umständen nur ein Teil der Prozessoren verwendet werden soll. Dadurch entstünde dann aber wieder ein bedeutend höherer Kommunikationsaufwand.

- Die Netzwerkcaches sind physikalisch adressiert. Sollten beim Anfordern eines großen Speicherbereichs alle Hypernodes noch wenig belastet sein, so ist die Wahrscheinlichkeit groß, daß die jeweilige Beiträge der Hypernodes auf dieselben Bereiche im Netzwerkcache abgebildet werden. Eine Anwendung, die den gesamten Speicherbereich benötigt wird sich aufgrund dieses Aliasing andauernd die eben eingelagerten Cachezeilen wieder verdrängen und damit sehr hohe Cachemisszahlen erzeugen.

Eine anderer Effekt beruht auf einer Eigenschaft des zugrundeliegenden Betriebssystems MACH. Fordert eine Bibliotheksfunktion Speicher vom Betriebssystem an, so wird dieser nicht tatsächlich sofort angelegt, sondern es werden nur die notwendigen Informationen im

Betriebssystem hinterlegt. Erst wenn der erste Prozessor auf eine Seite zugreift, die bis zu dem Zeitpunkt noch nicht benutzt wurde, wird sie tatsächlich angelegt. Startet man nun eine Anwendung, bei der eine hohe Zahl an Prozessoren gleichzeitig diese Aktion auslösen, so müssen sie sich alle innerhalb des Betriebssystems koordinieren. Als Resultat entstehen sehr hohe Systemzeiten, die bei den Beispielanwendungen die Größenordnung der reinen Rechenzeiten erreichten.

6.2.1 Vorstellung der Beispielapplikationen

Zwei der verwendeten Beispielanwendungen stammen aus dem Bereich der numerischen Mathematik. Anwendungen aus der numerischen Mathematik benötigen meist sehr hohe Rechenleistungen und stellen damit typische Beispiele für wissenschaftliches Hochleistungsrechnen dar. Bei der dritten Anwendung handelt es sich um ein synthetisches Testprogramm, mit dem die Effizienz der verwendeten Koordinierungsmechanismen getestet wurde ([Bel95b]).

Bei der Parallelisierung der nachfolgend vorgestellten, numerischen Anwendungen wurde auf die Verwendung spezieller Optimierungen verzichtet, sondern der zugrundeliegende Algorithmus in einfacher Weise parallelisiert. Ziel der Beispiele war nicht einen optimalen Algorithmus für die Parallelisierung auf einer NUMA-Architektur zu entwickeln, sondern den Einfluß der Mechanismen der Threadbibliothek auf ein, mit Hilfe von Threads parallelisiertes Programm zu zeigen.

LR-Faktorisierung

Bei der ersten Anwendung handelt es sich um die LR-Faktorisierung einer regulären Matrix mittels Gauss-Elimination. Die Matrix A wird durch dieses Verfahren in eine rechte obere Dreiecksmatrix R und eine linke untere Dreiecksmatrix L in der Art zerlegt, daß gilt $A = L * R$. Dieses direkte Verfahren stellt damit eine Grundlage etwa zur Invertierung von Matrizen und Lösung linearer Gleichungssysteme dar, wobei im allgemeinen Fall noch durch geeignete Maßnahmen für hinreichende numerische Stabilität gesorgt werden muß.

Parallelisiert wurde das Verfahren, indem für jede Zeile der 1024x1024 Elemente umfassenden Matrix A ein Thread erzeugt wurde, der für die Berechnung seiner Zeile verantwortlich ist. Um sicherzustellen, daß die für die Berechnung seiner Zeile notwendigen Daten der darüberliegenden Zeilen fertig berechnet vorliegen, wurden geeignete Koordinierungsmaßnahmen ergriffen.

Durch diese Koordinierung kann es zu mehrfachen Blockierungen während der Berechnungsphase eines Threads kommen. Damit sind die Voraussetzung für den gewinnbringenden Einsatz von Affinitätsbetrachtungen und MCS geschaffen. Untersucht man aber das Zugriffsverhalten der Threads vor und nach einer Blockierung, so stellt man fest, daß er nachher auf andere Bereiche zugreift, da er immer genau vor dem Zugriff auf die nächste der darüberliegenden Matrixzeilen blockiert wird. Wenn der Thread also Nutzen aus seiner Affinität zieht, dann nur aufgrund seiner privaten Daten wie seinem Stack und seiner eigenen Matrixzeile, auf die er weiterhin zugreift.

Als problematisch haben sich die oben beschriebenen Eigenschaften der verwendeten Maschine erwiesen. Da vor allem die Threads für die letzten Zeilen fast die gesamte Matrix für ihre Berechnung benötigen macht sich bei ihnen das Aliasing innerhalb der Netzwerkcaches stark bemerkbar. Um diesem Effekt aus dem Weg zu gehen, wurde die Matrix als *NEAR_SHARED* angefordert, d.h. sie liegt komplett im Speicher eines Hypernodes und belegt damit keine physikalischen Adressen, die sich im Netzwerkcache überlagern können. Damit wurde aber auch die Möglichkeit aufgegeben, Threads am Ort ihrer Daten zu starten. Stattdessen wurden die Threads gleichmäßig auf die Hypernodes verteilt. Das Problem der hohen Systemzeiten konnte hingegen nur unzureichend beseitigt werden, indem vor dem Start der eigentlichen Berechnung pro Hypernode ein Thread alle Seiten des verwendeten Speichers anfaßt. Dadurch daß nur ein Thread pro Hypernode das Anlegen interner Strukturen im Betriebssystemkern auslöst, konnte die Systemzeit aber zumindest um eine Größenordnung gesenkt werden.

Jacobi-Iteration

Die zweite Anwendung stellt ein iteratives Lösungsverfahren für lineare Gleichungssysteme dar, wie sie vor allem bei der Diskretisierung partieller Differentialgleichungen entstehen. Es handelt sich dabei um das sogenannte Gesamtschritt- oder Jacobiverfahren, bei dem iterativ eine Lösung des linearen Gleichungssystems $A * x = b$ ermittelt wird. Dazu wird die Matrix A folgendermaßen zerlegt:

$$A = D - L - U \tag{6.1}$$

Darin stellt D die Matrix der Diagonalelemente dar, während die Summe $-(L + U)$ den restlichen Teil der Matrix enthält. Ausgehend von einem Startwert für die Lösung x^0 wird die Lösung iterativ durch die folgende Gleichung ermittelt

$$x^{n+1} = D^{-1}(L + U)x^n + D^{-1}b \tag{6.2}$$

Auf die Voraussetzungen, die die Matrix A aufweisen muß, damit diese Iteration tatsächlich konvergiert soll hier nicht eingegangen, sondern auf die einschlägige Literatur verwiesen werden (siehe etwa [Sch93]).

Wie bei der Lösung diskretisierter, partieller Differentialgleichungen üblich wurde auch hier auf ein explizites Aufstellen der Matrix A verzichtet. Stattdessen wird das Verfahren direkt auf dem diskretisierten Gebiet ausgeführt, das in obiger Gleichung als Vektor b enthalten war. Das Gebiet wurde dabei wieder in Streifen unterteilt, die einzelnen Threads zugeordnet wurden. Nach jedem Iterationsschritt synchronisieren sich alle Threads mittels einer Barriere und es wird aufgrund einer berechneten Fehlergröße entschieden, ob eine weitere Iteration angestoßen werden soll.

Da ein Thread in jeder Iteration, abgesehen von den Randwerten seines Streifens nur die Daten seines letzten Iterationsschrittes verwendet, bietet diese Anwendung gute Voraussetzungen um Nutzen aus MCS und Affinitätsbetrachtungen zu ziehen. Auch spielen hier die negativen Eigenschaften der verwendeten Architektur nur eine untergeordnete Rolle. So ergeben sich nur geringe Systemzeiten, da alle Threads ein anderes Zugriffsverhalten zeigen und damit weniger Synchronisationsaufwand innerhalb des Betriebssystems notwendig ist. Andererseits greifen sie aber auch immer nur auf einen kleinen Teil des Speichers zu, so daß auch das Aliasing im Netzwerkcache kaum Einfluß auf die Berechnung zeigt.

Koordinierung durch Mutexlocks

In dieser Testanwendung wurden eine große Anzahl an Threads erzeugt, die sich paarweise mittels eines Mutexlocks synchronisieren, wobei regelmäßig Threadwechsel ausgelöst werden. Zwischen den Lockingaufrufen werden keine Berechnungen oder Speicherzugriff durchgeführt.

Damit beschränkt sich die Arbeitsmenge der Threads nur auf ihre Threaddatenstrukturen und ihren Stack. Der erreichbare Durchsatz wird also hauptsächlich davon bestimmt, ob diese Strukturen in den einzelnen Caches vorliegen und wie schnell die Synchronisation über die Mutexlocks erfolgt. Durch die hohe Anzahl an Threadwechsel und den damit notwendigen Schedulingentscheidungen kann, unabhängig von etwaigen Abhängigkeiten der Berechnung einzelner Threads, wie sie bei den numerischen Anwendungen auftreten, der Einfluß der Threadverwaltung untersucht werden.

6.2.2 Einfluß von MCS auf den erreichbaren Speedup

Um den Einfluß von MCS auf den erreichbaren Speedup der Anwendungen testen zu können, wurde eine zentrale Version der Bibliothek erstellt, die nur eine gemeinsame Run- und Startqueue aufweist. Auch die lokalen Freilisten wurden auf ihre Minimalgröße reduziert und die Strukturen stattdessen ausschließlich aus entsprechend vergrößerten globalen Freilisten des Nodes entnommen. Demgegenüber steht die in Abbildung 6.1 gezeigte Struktur mit lokalen Warteschlangen und Freilisten, aber ebenfalls noch ohne ergänzende Betrachtung von Affinitätsinformationen.

Beurteilung der Meßergebnisse

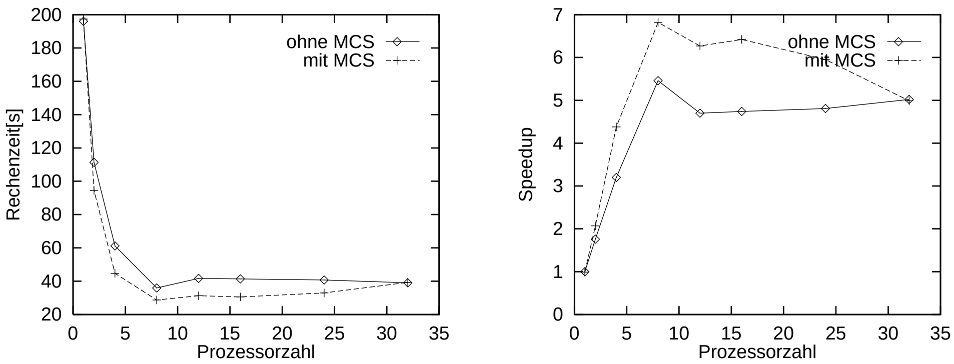


Abbildung 6.3: Das Jacobiverfahren mit und ohne MCS

Abbildung 6.3 zeigt die gemessenen Rechenzeiten und Speedupwerte des Jacobiverfahrens auf einem Gebiet mit 512x512 Punkten und 1000 Iterationen. Man erkennt deutlich den Einfluß der hohen Latenzzeiten beim Zugriff auf entfernten Speicher. Sobald mehr als 8 Prozessoren verwendet werden, erfolgt die Kommunikation an den Rändern verschiedener Streifen über Nodegrenzen hinweg. Im Vergleich zur Rechenzeit von nur wenigen Hundert Mikrosekunden pro Streifen in jeder Iteration macht sich diese Kommunikation stark bemerkbar. Für jeden der 128 in der Parallelisierung eingesetzten Threads entstehen pro Iteration 256 Cachemisses im Datencache, bzw. 128 Cachemisses im Netzwerkcache, falls die Kommunikation über Nodegrenzen erfolgt. Diese Kosten in Verbindung mit den Verlusten durch die notwendigen Threadwechsel erklären den relativ geringen erreichbaren Speedup.

Ebenso deutlich ist aber auch der Unterschied zwischen der zentralen und der dezentralen Version erkennbar. Zwar gleichen sich die Formen der Speedupkurven, da diese charakteristisch für die gewählte Parallelisierung sind, jedoch verläuft die Kurve für die zentrale Version des Programms aufgrund der weitaus größeren Anzahl an Cachemisses flacher. Bei steigender Prozessorzahl macht sich die Kommunikation über Nodegrenzen als dominanter Faktor bemerkbar. Da sich diese Kosten nicht reduzieren lassen, gleichen sich die beiden Speedupkurven langsam einander an, obwohl die zentrale Version bei 32 Prozessoren etwa 60 Millionen Cachemisses mehr erzeugt, als die dezentrale.

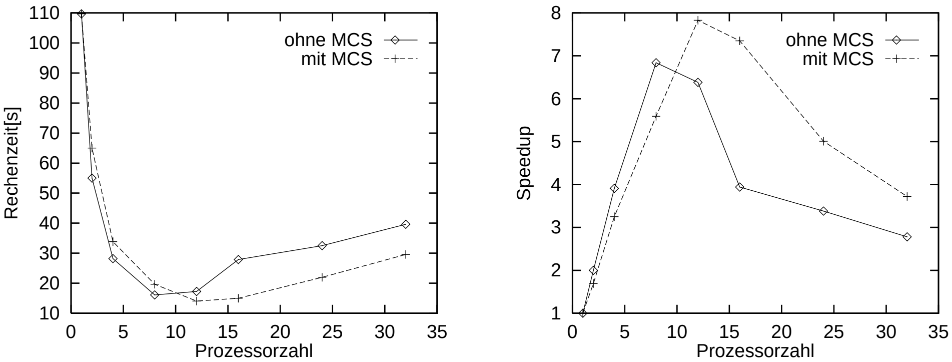


Abbildung 6.4: Die LR-Faktorisierung mit und ohne MCS

Noch stärker als beim Jacobiverfahren macht sich die Kommunikation bei der LR-Zerlegung mittels der Gauss-Elimination auf der verwendeten 1024x1024 Matrix bemerkbar. Zwar kann hier MCS ein Steigerung des Speedups über Hypernodegrenzen hinaus bis zu 12 Prozessoren erreichen, jedoch fällt die Speedupkurve anschließend sehr schnell.

Durch die großen Unterschiede in der Laufzeit der einzelnen Threads entsteht bei der Gauss-Elimination ein erhebliches Lastungleichgewicht. Durch den besseren Lastausgleich zeigt deshalb die zentrale Version innerhalb eines Hypernodes bessere Ergebnisse. Jedoch kann sie keinen Nutzen aus den Prozessoren mehrerer Hypernodes ziehen, da durch die entstehenden Migrationskosten der Gewinn durch den Lastausgleich aufgehoben wird. Durch die Bindung an Prozessoren bei der dezentralen Version werden diese Kosten reduziert, so daß auch mehr als ein Hypernode ausgenutzt werden kann. Der maximale Speedup wird hier bei 12 Prozessoren erreicht und bleibt auch im Anschluß deutlich höher als bei der zentralen Version ohne MCS, die bei 32 Prozessoren wiederum etwa 40 Millionen Cachemisses mehr erzeugt, als die dezentrale Version.

Abbildung 6.5 zeigt die maximal erreichte Anzahl an Synchronisationsoperationen, die durch die paarweise Synchronisation von 4096 Threads mit jeweils 8 KByte Stack gemessen wurden. Bei einem Prozessor werden sowohl von der zentralen, als auch von der dezentralen Version etwa 100000 Synchronisationen und damit Threadwechsel pro Sekunde erreicht. Steigt die Anzahl der Prozessoren, so fällt diese Zahl bei der zentralen Version aber sofort stark ab. Neben dem entstehenden Engpaß beim Zugriff auf die zentralen Strukturen wird diese Abnahme vor allem durch das komplette Ignorieren von Cacheaffinität hervorgerufen. Die Threads migrieren unkontrolliert zwischen den verwendeten Prozessoren und verbringen dabei den Großteil der Zeit damit, ihre Arbeitsmengen in die Prozessorcaches zu laden.

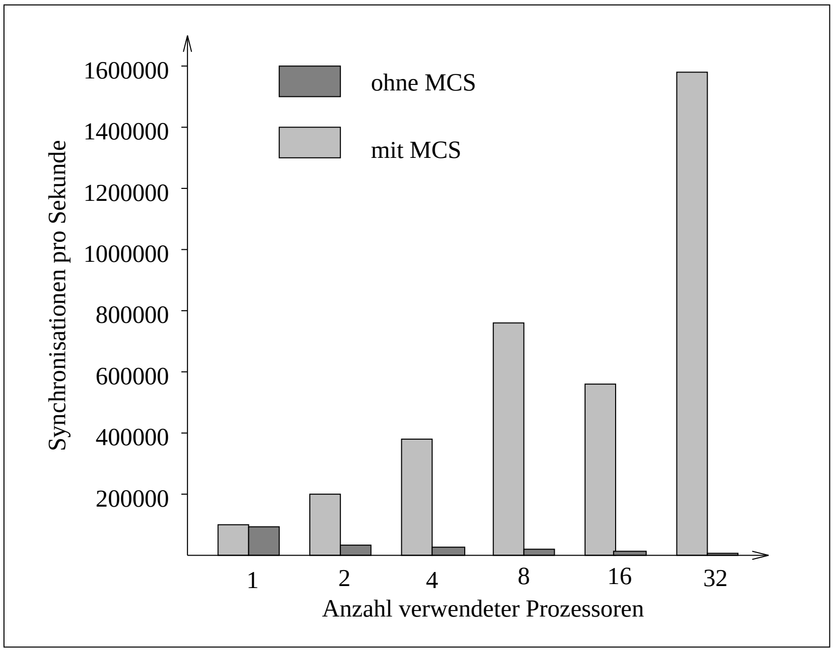


Abbildung 6.5: Synchronisation durch Mutexlocks

Die dezentrale Version skaliert hingegen bis zu 8 Prozessoren fast linear. Jeder Prozessor leistet etwa 100000 Synchronisationen pro Sekunde. Sobald die Lockingoperationen aber Hypernodegrenzen überschreiten bricht die erreichbare Zahl auf ca. 40000 Operationen pro Sekunde ein. Steigt die Anzahl an Prozessoren weiter, nehmen die notwendigen Cachemisses zum Laden der Arbeitsmengen weiter ab, bis diese, bei 32 Prozessoren fast vollständig in den Caches der Prozessoren verbleiben. Dadurch steigt der Durchsatz wieder auf ca. 50000 Operationen pro Sekunde an. Trotz der hohen Kommunikationskosten skaliert die Anwendung aber jenseits dieses Einbruchs wieder mit der steigenden Prozessorzahl.

Schlußfolgerung für die Verwendung von MCS

Die Meßergebnisse machen deutlich, daß die Anwendung von MCS in Form dezentraler Strukturen durchwegs positive Ergebnisse zeigt. Für fast jede Prozessorzahl wurde in den Anwendungen eine zumindest gleichwertige Auslastung der Maschine, erkennbar an den meist höheren Speedupzahlen erreicht. Spätestens aber bei der Überschreitung von Hypernodegrenzen mit den dann anfallenden hohen Latenzzeiten beim Zugriff auf entfernten Speicher überwiegen die Vorteile von MCS die Nachteile, die durch die aufwendigeren Algorithmen bei der Threadverwaltung erzeugt werden. Im Fall der LR-Faktorisierung gelang es zum Beispiel den sogenannten *Operating Point*, also den Punkt höchsten Speedups zu höheren Prozessorzahlen und damit mehreren Hypernodes zu verlagern([Tuc93]).

Für Anwendungen, die ähnlich der synthetischen Testanwendung nur geringe Berechnungen ausführen und deren Durchsatz hauptsächlich von der Zeit zum Laden ihrer Arbeitsmenge dominiert wird, ist MCS mit dezentralen Strukturen bereits bei kleinen Prozessorzahlen unabdingbar. Die Verbesserungen sind dabei einerseits durch den wegfallenden Engpaß der zentralen Datenstrukturen, andererseits aber vor allem auch durch eine besser Beachtung der Affinität der einzelnen Threads zu ihren Prozessoren zurückzuführen. Die Tatsache, daß es sich bei den Hypernodes der Convex SPP um symmetrische Multiprozessoren mit UMA-Charakteristik handelt, zeigt, daß die dezentralen Verfahren nicht nur auf Hochleistungsrechner mit NUMA-Architektur anwendbar sind, sondern auch auf kleineren Parallelrechnern.

Auf Architekturen mit besseren Möglichkeiten die Lokalität von Daten zu bestimmen, sollte sich der Einfluß von MCS sogar noch weiter erhöhen lassen, da damit nicht nur die Anzahl an Cachemisses reduziert werden kann, sondern auch deren mittlere Latenzzeit.

6.2.3 Einfluß verschiedener Affinitätsmodelle

Um den Einfluß der verschiedenen implementierten Affinitätsmodelle beurteilen zu können, wurden wieder die beiden numerischen Anwendungen verwendet. Beide wurden mit den entsprechenden Bibliotheken übersetzt und mehrfach bei verschiedenen Prozessorzahlen getestet. Da dabei die gemessenen Zeitunterschiede relativ klein sind, machen sich Einflüsse des Betriebssystems, etwa bei der Zeit zum Generieren der Kernelthreads, hier stärker bemerkbar und verfälschen die Meßergebnisse. Die eindeutige Entscheidung für oder gegen ein Modell kann also nicht immer getroffen werden.

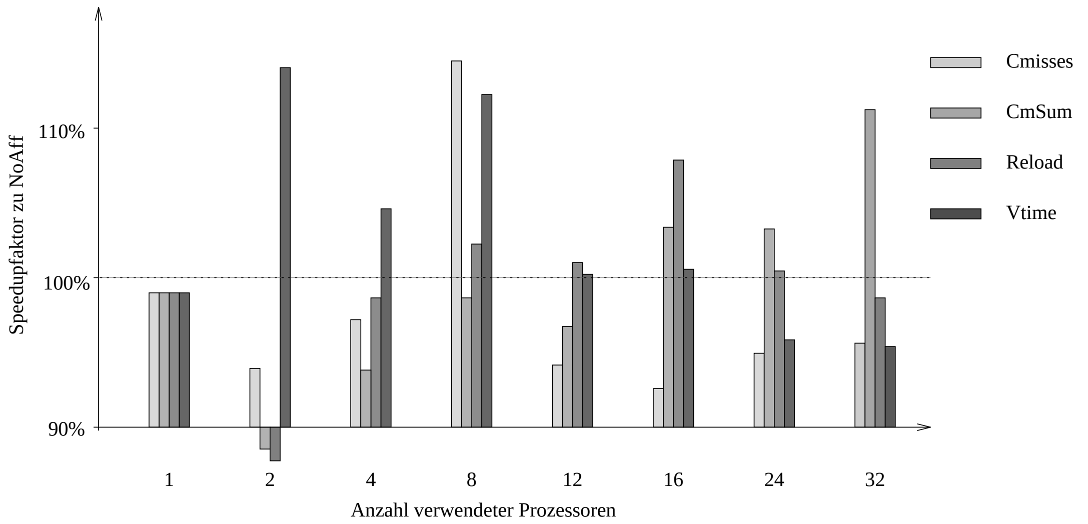


Abbildung 6.6: Einfluß von Affinitätsbetrachtung auf die LR-Faktorisierung

Abbildung 6.6 zeigt den Einfluß, den die zusätzlichen Affinitätsbetrachtungen auf den Speedup der LR-Faktorisierung hatten. Alle Werte wurden dabei auf den Speedup bezogen, der bei einfachem MCS ohne Affinitätsbetrachtungen gemessen wurde. Bei der Interpretation der Werte muß darauf geachtet werden, daß bei der gewählten Parallelisierung eine Abhängigkeit zwischen den Threads existiert. Die Entscheidung über eine gewisse Reihenfolge beeinflußt damit nicht nur die auftretenden Cachemisses, sondern unter Umständen auch die Anzahl notwendiger Threadwechsel. Dadurch entstehen Effekte, wie etwa bei zwei Prozessoren, wo bei **Reload**, trotz einer Verringerung der Cachemisses um 8 Millionen, die Rechenzeit steigt.

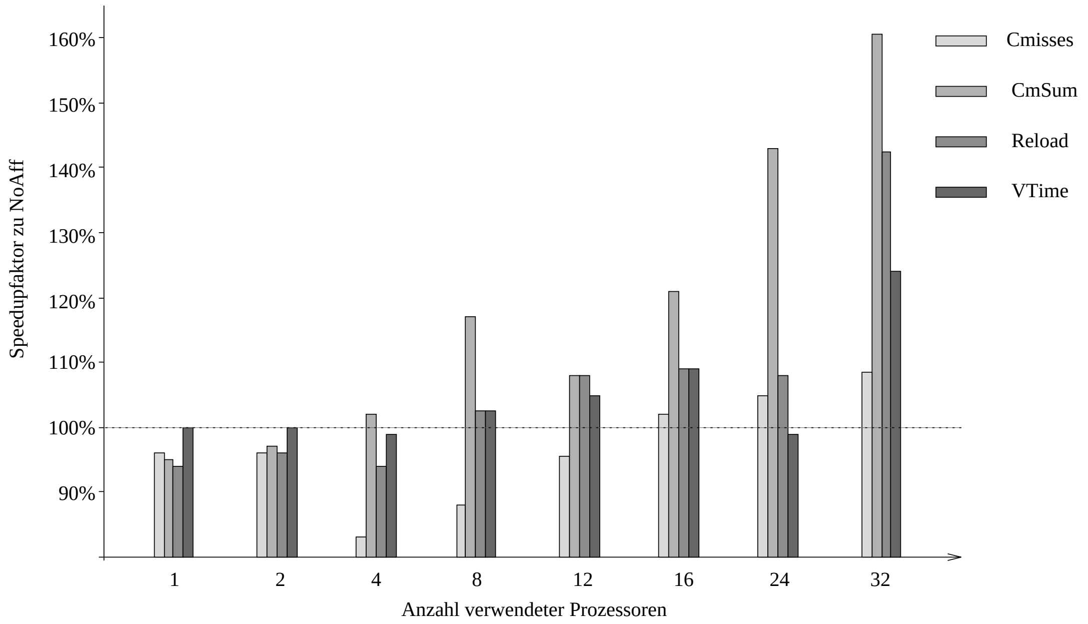


Abbildung 6.7: Einfluß von Affinitätsbetrachtungen auf das Jacobiverfahren

Abbildung 6.7 zeigt dieselbe Auswertung für das Jacobiverfahren. Im Vergleich zur LR-Faktorisierung, bei der sich die Arbeitsmenge der einzelnen Threads andauernd verändert, bleibt diese bei der Jacobi-Iteration konstant. Affinitätsbetrachtungen können damit viel besser aus dem bisherigen Verhalten des Threads Schlüsse auf sein weiteres Verhalten ziehen und damit durch eine entsprechende Berechnungsreihenfolge Verbesserungen erzielen. Entsprechend deutlicher fällt das Ergebnis der Tests aus. Obwohl die Methoden, die eine Messung der Cachemisses benötigen mehr Aufwand erzeugen und deshalb bei geringen Prozessorzahlen teilweise schlechter abschneiden, steigt bei fast allen Methoden der Speedup bei zunehmender Prozessorzahl über den Speedup ohne Affinitätsbetrachtungen. Da außerdem bei der Jacobi-Iteration keine weitere Abhängigkeiten zwischen den Threads zu einer Verlangsamung der Berechnung führt, treten hier auch keine unerwarteten Einbrüche bei den einzelnen Methoden auf. Insgesamt kann man dabei die einzelnen Methoden folgendermaßen beurteilen:

Cmisses: Diese Methode approximiert das Cacheverhalten der Threads schlecht, da aus dem letzten Lauf eines Threads nur ungenügend auf seinen tatsächlichen Cachezustand geschlossen werden kann. Nur selten erzeugt dieses Verfahren, wie etwa bei der LR-Faktorisierung mit 8 Prozessoren, eine Berechnungsreihenfolge, die eine Beschleunigung ergibt. Die Verbesserung im Verhalten bei steigender Prozessorzahlen in der Jacobi-Iteration kann darauf zurückgeführt werden, daß immer weniger Threads pro Prozessor zur Verfügung stehen. Damit haben alle Verfahren aber nur noch einen geringeren Einfluß auf die Berechnungsreihenfolge.

CmSum: Im Gegensatz zu **Cmisses** ergibt diese Methode eine bessere Modellierung des Cachezustandes der Threads. Sowohl bei der LR-Faktorisierung, als auch bei der Jacobi-Iteration wurden mit dieser Methode die besten Meßwerte erzielt. Gegenüber einem maximalen Speedup von 5.5 bei der zentralen bzw. 7 bei der dezentralen Threadverwaltung beim Einsatz von 8 Prozessoren steigert dieses Affinitätsmodell den Speedup bei 8 Prozessoren auf 7.96 und auf maximal 8.5 beim Einsatz von 24 Prozessoren.

Reload: Die Zuhilfenahme der mathematischen Überlegungen zur Modellierung des Reload Transients ergab neben der Methode **CmSum** die besten, reproduzierbaren Ergebnisse. Bei kleinen Prozessorzahlen wird dieser Gewinn bei beiden Anwendungen durch den hohen Aufwand bei der dynamischen Berechnung der Prioritäten aufgebraucht. Erst bei höheren Prozessorzahlen überwiegen die Verbesserungen die Kosten. Auf der anderen Seite nehmen bei einer Erhöhung der Prozessorzahl aber auch die Cachemisses im Netzwerkcache zu. Da diese sich, aufgrund fehlender Meßmöglichkeiten aber nicht getrennt modellieren ließen, nimmt die Treffsicherheit des Algorithmus im Vergleich zu den anderen langsam wieder ab.

VTime: Die Einfachheit dieser Methode macht sich vor allem bei geringen Prozessorzahlen bemerkbar, wo sie den aufwendigeren Methoden überlegen ist. Sie erreicht bei hohen Prozessorzahlen zwar nicht die Leistung dieser Algorithmen, erzeugt aber zumindest in vielen Fällen eine Verbesserung gegenüber einfachem MCS. Die Tatsache, daß sie keine Hardwareanforderungen stellt und bei geringer Prozessorzahl, d.h. hohen Threadzahlen gute Ergebnisse liefert macht diese Methode besonders beim Einsatz auf kleineren, einfachen Parallelrechnern interessant. Aber sogar auf Monoprozessoren kann diese Strategien eine bessere Cacheausnutzung ergeben. So dauerte die Jacobi-Iteration bei ihrer Berechnung mit nur einem Thread auf einem Prozessor, also der klassischen Monoprozessorlösung, um etwa 25% länger, als bei der Parallelisierung mit 128 Threads und der Anwendung von **VTime**, trotz des für die Threadverwaltung notwendigen Overheads.

Schlußfolgerung für zusätzliche Affinitätsbetrachtungen

Spätenstens beim Überschreiten von Hypernodegrenzen, d.h. beim verstärkten Auftreten von Zugriffen auf entfernten Speicher lohnt eine erweiterte Untersuchung der Cacheaffinität. Die Kosten für die notwendigen Messungen und Berechnungen werden durch eine Verringerung der Cachemisses aufgewogen. Solange keine Hardwareunterstützung angeboten wird, kann zumindest durch **VTime** eine Verbesserung erreicht werden. Bessere Werte lassen sich aber durch genauere Untersuchungen des Cachemiss-Verhaltens der einzelnen Threads erzielen, wobei aber für eine effiziente Implementierung der einzelnen Methoden Anforderungen gestellt werden müssen, denen zumindest die Convex SPP 1000 momentan noch nicht entspricht:

- Eine getrennte Messung von Cachemisses beim Zugriff auf lokalen und entfernten Speicher muß möglich sein.

- Auf alle Funktionen des Performancemonitorings muß mit einer Latenzzeit vergleichbar einem Cachehit zugegriffen werden können, um die Kosten für die Messung zu reduzieren. So erzeugt das Auslesen der verschiedenen Speicherzellen in der momentanen Implementierung einen Zeitaufwand, der den eines Threadwechsels des darunterliegenden Quickthreads-Pakets übersteigt. Dadurch sinkt die Einsetzbarkeit dieser Methoden im Bereich feinster Parallelisierungen, die eigentlich am meisten aus diesen Überlegungen profitieren könnten.

- Um etwa beim Lastausgleich die verbliebene Affinität eines Threads zu seiner bisherigen CPU abschätzen zu können, wäre der Zugriff auf die Cachemisszähler fremder Prozessoren wünschenswert.

Unter Einbeziehung all dieser Verbesserungen sollte eine weitere Reduzierung der Cachemisszahlen durch die aufwendigeren Modelle erreichbar sein.

Kapitel 7

Zusammenfassung und Ausblick

Die Arbeit hat gezeigt, daß Threadbibliotheken nicht nur ein probates Mittel zur feingranularen Parallelisierung von Programmen darstellen, sondern darüberhinaus ideale Voraussetzungen für das Einbeziehen von Lokalitätsinformation bieten.

Zwei verschiedene Techniken wurden vorgestellt, die zusammen eine optimale Ausnutzung der Datenlokalität im Hauptspeicher und in den verschiedenen Caches sicherstellen. Memory Conscius Scheduling dient dabei dazu, Threads unter Beachtung ihrer hauptsächlich benutzten Daten an Prozessoren zu binden, um damit einen weitestgehend lokalen Speicherzugriff zu garantieren. Dadurch, daß die Threads auch in hintereinander folgenden Berechnungsabschnitten weitestgehend auf diesem Prozessor berechnet werden, steigt zudem die Cacheausnutzung der Prozessorcaches. Bei der Analyse zweier möglicher Implementierungsarten wurde festgestellt, daß ein zentraler Ansatz den Anforderungen, wie sie vor allem auf modernen NUMA-Architekturen mit vielen Prozessoren gestellt werden, nicht genügen kann. Stattdessen hat sich ein verteilter Ansatz als geeignet herauskristallisiert, da er eine einfache Implementierung mit einer weitaus besseren Skalierbarkeit verbindet. Lokale Warteschlangen und Freilisten sorgen für eine einfache Abbildung von Lokalitätsinformation auf eine Zuordnung von Threads zu Prozessoren und außerdem für den Erhalt der dadurch festgelegten Lokalität.

Zusätzliche Affinitätsbetrachtungen helfen anschließend eine Entscheidung über die Reihenfolge der Threads in den lokalen Runqueues der Prozessoren zu finden. Auch hier war das Ziel eine weitere Reduktion der auftretenden Cachemisses erreichen. Neben einfachen heuristischen Ansätzen, wie sie häufig in der Literatur zu finden sind, wurde auch ein mathematisches Modell präsentiert, daß die Anzahl an Cachemisses beim erneuten Start eines Threads abschätzen hilft.

Neben der eigentlichen Threadverwaltung wurde auch auf die Probleme bei der Implementierung von Koordinierungsmechanismen eingegangen. Auch diese dürfen auf einer NUMA-Architektur die Lokalität ihrer Daten und den Zugriff darauf nicht ignorieren. Neben Techniken, die die Anzahl auftretender Cachemisses reduzieren, wie etwa Backoff Spinning, wurde unter anderem auch auf die Frage eingegangen, wie blockierende Mechanismen im Rahmen des vorher entwickelten Schedulingkonzepts eingebaut werden können. Neben der Entscheidung für zweistufige Blockierungsmechanismen stellte sich hier die Frage, wo blockierte Threads gespeichert werden sollten. Auch hierfür wurden ein tragfähiges Konzept vorgestellt, das anstelle einer zentralen Warteschlange eine Verteilung der Datenstrukturen vorsieht und damit eine Verbesserung der Lokalität zu den einzelnen Prozessoren erreicht.

Um die getroffenen Aussagen überprüfen zu können, wurde im Rahmen dieser Arbeit eine Threadbibliothek erstellt, die besonders auf die Belange einer NUMA-Architektur wie der Convex SPP 1000 eingeht. Neben einem vollständig verteilten Schedulingkonzept verwendet die Bibliothek die von der Maschine angebotenen Hardwaremechanismen um Cachemisszahlen zu ermitteln und für die Realisierung einiger angesprochener Affinitätsmodelle zu verwenden. Die erfolgten Tests zeigten dabei sowohl die gute Skalierbarkeit der eingesetzten Koordinierungsmechanismen, als auch den positiven Einfluß von Memory Conscious Scheduling und Affinitätsmodellen auf die Rechenzeit von Anwendungen.

Die Tests zeigten aber auch die Grenzen der eingesetzten Modelle, die sowohl durch die Beschränkungen der Hardware, als auch durch den weitgehenden Einsatz heuristischer Ansätze begründet werden kann. Bessere Meßmöglichkeiten, wie die getrennte Messung von Cachemisses beim Zugriff auf lokalen und entfernten Speicher könnten hier als Grundlage für den gewinnbringenden Einsatz genauerer mathematischer Modelle dienen. Darüberhinaus wäre unter Zuhilfenahme dieser Modelle eine mathematische Beantwortung der Frage interessant, in welcher Reihenfolge eine Menge von Threads berechnet werden muß, um die wenigsten Cachemisses auszulösen. Bei allen Ansätzen darf dabei aber nicht aus den Augen verloren werden, daß die dazu notwendigen Berechnungen durch die entstehenden Gewinne aufgewogen werden müssen. Die Beispielanwendungen haben hier etwa gezeigt, daß aufwendigere Mechanismen erst dann eine Verbesserung erreichten, wenn die Berechnung mit den hohen Latenzzeiten beim Zugriff auf den Speicher anderer Hypernodes konfrontiert wurde.

Neben der Verfeinerung der theoretischen Überlegungen und ihrer Realisierung in der Threadbibliothek, fehlt momentan noch die komplette Anbindung der Bibliothek an das Betriebssystem. Das Konzept der *Sleeping Threads* ([Kop95]), die im Rahmen einer Zusammenarbeit des Lehrstuhls IV für Betriebssysteme mit der Firma Convex in deren Betriebssystem implementiert wurde, könnte hier als Grundlage dienen. Dadurch wäre es möglich die Anzahl an Kernelthreads, die der Bibliothek zur Verfügung stehen konstant zu halten, selbst wenn sich Kernelthreads im Betriebssystemkern blockieren.

Anhang A

Befehlsumfang der Mthreads-Bibliothek

Die Mthreads-Bibliothek gliedert sich, wie in Kapitel 2.3 angedeutet, in drei Funktionsgruppen:

1. Funktionen zur Konfiguration und Initialisierung der Bibliothek
2. Funktionen zur Threadverwaltung
3. verschiedene Koordinierungsmechanismen

Die Funktionalität und Aufrufsyntax der einzelnen Funktionen soll im nachfolgenden kurz vorgestellt werden. Für eine genauere Zusammenstellung aller Optionen und deren Bedeutung sei auf die zu der Bibliothek verfügbaren Manuals verwiesen, die sowohl in nroff-, als auch in HTML-Format existieren.

A.1 Funktionen zur Verwaltung der Bibliothek

mthr_config, mthr_config_v

```
int  mthr_config(int option, int value);
void mthr_config_v(int *argc, char *argv[]);
```

Wird die Threadverwaltung der Bibliothek gestartet, so werden Standardwerte für verschiedene Strukturen, wie etwa die Länge der Freilisten verwendet. Diese Werte sichern ein korrektes Verhalten der Verwaltungsfunktionen der Bibliothek. Abhängig von der Anwendung, die mit Hilfe der Bibliothek parallelisiert wird, kann es aber andere Einstellungen geben, die ein besseres Laufzeitverhalten erzeugen. Dem Benutzer stehen für derartige Eingriffe zwei verschiedene Funktionen zur Verfügung:

- `mthr_config()` ändert den Wert einer internen Einstellung. Dieser Aufruf dient im allgemeinen dazu, neue Standardwerte für eine Anwendung fest in das Programm zu codieren.
- `mthr_config_v()` kann dazu verwendet werden, um Kommandozeilenparameter an die Bibliothek weiterzugeben und damit mehrere Einstellungen gleichzeitig zu ändern. Vor

einer Auswertung der Kommandozeile durch die eigentliche Anwendung wird der Parametervektor an die Bibliothek übergeben. Sie wertet die Parameter aus, die ihr bekannt sind und entfernt sie aus dem Vektor, so daß die Anwendung nachher unabhängig von den tatsächlichen Optionen der Bibliothek ihre Parameter auswerten kann. Der Zähler für die Anzahl gültiger Parameter `argc` wird dabei entsprechend angepaßt.

Das maschinenabhängige Headerfile `mthr_config.h` enthält die für eine Architektur möglichen Werte für `option`. So kennt etwa die Version für die Convex SPP die folgenden Optionen:

CPU_MBLOCKPOOL_SIZE (-m_c_mp) gibt die Größe der lokalen Freilisten für kleine Strukturen bei jeder CPU an. Die Freilisten der Nodes besitzen keine obere Grenze und sind deshalb nicht konfigurierbar.

CPU_STACKPOOL_SIZE (-m_c_sp) gibt an, wieviele Stacks in der lokalen Freiliste einer CPU maximal enthalten sein dürfen, bevor ein Stack an die Freiliste der Node übergeben wird.

NODE_STACKPOOL_SIZE (-m_n_sp) spezifiziert die maximale Größe der Freiliste für Stacks innerhalb jeder Node.

MIN_STACKSIZE (-m_min_ss) setzt ein unteres Minimum für die Größe jedes Stacks.

DEF_STACKSIZE (-m_def_ss) legt die Standardgröße für Stacks fest. Diese wird verwendet, wenn der Benutzer bei der Erzeugung eines Threads keinen anderen Wert angibt.

SPINS_BEFORE_BLOCK (-m_sbb) gibt die Anzahl von aktiven Wartezyklen in den zweistufigen Koordinierungsmechanismen an.

KTHR_SPINS_BEFORE_BLOCK (-m_k_sbb) legt die Wartezyklen vor einer Blockierung eines Kernelthreads fest.

NO_OF_NODES (-m_nodes) dient einer Einschränkung der Anzahl verwendeter Nodes. Standardmäßig wird beim Start der Bibliothek sonst der gesamte Subkomplex verwendet.

NO_OF_CPUS (-m_cpus) schränkt die Anzahl zu verwendender Prozessoren ein, wobei die Bibliothek versucht, die generierten Kernelthreads auf eine möglichst kleine Anzahl an Nodes zu verteilen, um die Kommunikationskosten zu reduzieren.

In Klammern angegeben stehen die entsprechenden Optionen bei der Verwendung des Befehls `mthr_config_v()`. Nach je einer dieser Optionen folgt der entsprechende Wert, d.h. ein Aufruf eines Programms kann etwa folgendermaßen aussehen:

```
prog_name -m_sbb 100 -m_cpus 4 eigene_Optionen
```

mthr_startup

```
void mthr_startup(void);
```

Im Anschluß an eine etwaige Konfiguration kann die Threadverwaltung eines Programms durch den Aufruf der Funktion `mthr_startup()` gestartet werden. Dabei werden alle internen Strukturen erzeugt und, wenn nicht durch den Benutzer anders konfiguriert, für jeden

Prozessor des Subkomplexes ein Kernelthread gestartet. Der Kernelthread, der die Funktion aufruft wird dabei von der Threadverwaltung übernommen.

Vor dem Aufruf dieser Funktion darf kein Thread und kein Koordinierungsmechanismus erzeugt werden, da die dafür notwendigen Strukturen noch nicht existieren!

mthr_set_shutdownhandler

```
shutdown_handler_t mthr_set_shutdownhandler(shutdown_handler_t new_handler);
```

Programme, die mit der Mthreads-Bibliothek parallelisiert wurden, laufen bis keine lauffähigen oder startfähigen Threads mehr existieren, ein Thread die Funktion `exit()` aufruft oder der ursprüngliche Kernelthread, mit dem das Programm startete, die Funktion `main()` beendet.

In allen dieser Fälle wird eine zentrale Funktion, der sogenannte Shutdownhandler, aufgerufen. Diese Funktion ermöglicht es, vor dem tatsächlichen Beenden der Anwendung noch spezielle Aktionen anzustoßen, wie etwa das Sammeln von Profiling-Daten. Der Benutzer kann zu diesem Zweck durch `mthr_set_shutdownhandler` eine Funktion angeben, die dann von der Bibliothek aufgerufen wird. Als Rückgabewert liefert der Aufruf eine Referenz auf den bisher gesetzten Shutdownhandler.

A.2 Funktionen zur Threadverwaltung

mthr_create

```
    mthread_p mthr_create(unsigned stacksize, unsigned mode,
                          start_func ufct, void *arg);
```

Dieser Aufruf erzeugt einen neuen Thread und trägt ihn in eine Startqueue ein. Wird der Thread anschließend von einem Prozessor gestartet, so erhält er einen Stack mit der durch `stacksize` angegebenen Größe bzw. der Standardgröße, falls als Wert Null angegeben wurde. Anschließend wird von diesem Thread die Startfunktion `ufct` mit dem Parameter `arg` aufgerufen. Sollte diese Funktion zurückkehren, so wird der Thread durch die Bibliothek beendet.

Welche Art von Thread dabei erzeugt und in welche Startqueue er eingetragen werden soll, kann durch den Parameter `mode` festgelegt werden. Dabei stehen folgende Optionen zur Auswahl:

DETACHED: Threads dieses Typs werden bei ihrer Terminierung vollständig freigegeben. Ohne Anwendung dieser Option bleibt bei der Terminierung die Threadstruktur übrig, in der der Exitwert des Threads gespeichert wird. Ein anderer Thread kann dann auf das Ende dieses Threads warten und den Rückgabewert entnehmen.

CPU_LOCAL: Der Thread wird in die Startqueue des aufrufenden Prozessors eingetragen.

NAMED_CPU: Der Thread wird in die Startqueue der angegebenen CPU eingetragen, wobei die CPUs von 0 bis $n-1$ durchnumeriert sind. Die tatsächliche Anzahl von CPUs läßt sich durch den Befehl `mthr_no_cpus()` ermitteln.

NODE_LOCAL: Der Thread wird in die Startqueue des lokalen Node eingetragen.

SC_GLOBAL: Der Thread wird in die globale Startqueue des Subkomplexes eingetragen.

Die Funktion liefert eine Referenz auf den neu erzeugten Thread zurück, bzw. NULL im Fehlerfall.

merror

```
void merror(char *msg);
```

Tritt bei einer Funktion der Threadbibliothek ein Fehler auf, so enthält die globale Variable `merrno` eine Fehlernummer. Statt diese Nummer direkt auszuwerten, kann mittels `merror()` eine Fehlermeldung auf den Standardfehlerkanal ausgegeben werden. Wird vom Benutzer in `msg` eine eigene Meldung mitgeliefert, so wird diese gefolgt von der internen Fehlermeldung ausgegeben.

mthr_exit

```
void mthr_exit(int exitvalue);
```

Durch diesen Aufruf wird ein Thread terminiert, d.h. er kommt nicht mehr aus diesem Aufruf zurück. Handelt es sich um einen Thread vom Typ **DETACHED**, so wird er komplett freigegeben. Sonst geht der Thread in einen Zombiezustand über, aus dem er nur durch einen anderen Thread mittels `mthr_join()` befreit werden kann. Dabei hat der anderere Thread die Möglichkeit, den Exitwert des terminierten Threads abzufragen.

Sollte ein Thread nicht ordnungsgemäß durch diese Funktion beendet werden, sondern aus seiner Startfunktion zurückkehren, so wird er intern mit dem Exitwert Null terminiert. Eine Sonderstellung nimmt dabei der Thread ein, der nach dem Aufruf von `mthr_startup()` die Funktion `main()` weiterbearbeitet. Beendet er `main()` nicht mit `mthr_exit()`, sondern verläßt sie mit `return` , so wird die gesamte Anwendung dadurch terminiert!

mthr_join

```
mthread_p mthr_join(mthread_p tp, int *exitloc);
```

Diese Funktion blockiert den aufrufenden Thread solange, bis der durch `tp` angegebene Thread terminiert wurde, vorausgesetzt es handelt sich bei `tp` nicht um einen Thread vom Typ **DETACHED**. Der terminierte Thread wird durch `mthr_join()` aus seinem Zombiezustand befreit und vollständig freigegeben, wobei sein Exitwert vorher an der durch `exitloc` spezifizierten Speicherzelle hinterlegt wird.

mthr_no_cpus

```
int mthr_no_cpus(void);
```

Dieser Aufruf liefert die Anzahl tatsächlich von der Bibliothek verwendeter Prozessoren zurück.

mthr_prefetch_thread

```
void mthr_prefetch_thread(mthread_p tp);
```

Sollte der durch `tp` referenzierte Thread noch im Zustand **STARTABLE** sein, so wird er durch diesen Aufruf in die lokale Startqueue des aufrufenden Prozessors migriert.

mthr_self

```
mthread_p mthr_self(void);
```

Dieser Aufruf liefert eine Referenz auf den aufrufenden Thread zurück, wie sie etwa benötigt wird, damit sich der Thread selbst mittels `mthr_suspend()` suspendieren kann.

mthr_suspend, mthr_resume

```
int  mthr_suspend(mthread_p tp, int wait);
void mthr_resume(mthread_p tp);
```

Durch die Funktion `mthr_suspend()` kann ein Thread suspendiert werden, d.h. er wird bis zu einem nachfolgenden Aufruf von `mthr_resume()` nicht weiter berechnet. In der momentanen Implementierung wird ein Thread auf einer fremden CPU durch `mthr_suspend()` nicht verdrängt. Erst wenn er von sich aus den Prozessor aufgibt, etwa weil er sich blockiert, wird die Suspendierung tatsächlich wirksam. Durch den Parameter `wait` kann der aufrufende Thread angeben, ob er auf diese erfolgreiche Suspendierung des Threads warten oder sofort fortfahren will.

Mehrere Suspendierungsaufrufe an einem Thread können nur durch dieselbe Anzahl an Resumeaufrufen aufgehoben werden. Umgekehrt kann ein Resumeaufruf vor einer Suspendierung dazu führen, daß diese Suspendierung nicht wirksam wird.

mthr_yield

```
void mthr_yield(void);
```

Durch diesen Aufruf kann ein Thread seinen Prozessor freiwillig abgeben, d.h. er wird gestoppt und als lauffähiger Thread wieder in die Runqueue eingetragen.

A.3 Koordinierungsmechanismen

Alle Koordinierungsmechanismen der Bibliothek müssen dynamisch angefordert werden, da statisches Anlegen durch den Compiler an der notwendigen Ausrichtung im Speicher scheitert. Für jeden Mechanismus existiert aus diesem Grund ein Aufrufe der Form `new_...` zum Anfordern und `delete_...` zum Freigeben des Mechanismus. Die zurückgelieferten Referenzen müssen bei jedem Koordinierungsaufruf des Mechanismus mitangegeben werden.

A.3.1 Bedingungsvariablen

```
cond_p new_cond(void);
void   delete_cond(cond_p cvp);
cond_p cond_init(cond_p cvp);
int    cond_empty(cond_p cvp);
void   cond_wait(cond_p cvp, mutex_p mp);
void   cond_signal(cond_p cvp);
void   cond_broadcast(cond_p cvp);
```

Bedingungsvariablen werden im allgemeinen verwendet, um – meist im gegenseitigen Ausschluß – eine Bedingung zu testen und sich zu blockieren, falls die Bedingung nicht erfüllt war. Ändert ein Thread den Wert der Bedingung, so kann er einen oder alle wartenden Threads deblockieren, die dann wiederum die Bedingung prüfen können.

`new_cond()` erzeugt eine neue Bedingungsvariable, initialisiert sie mittels `cond_init()` und liefert eine Referenz darauf zurück. Freigegeben wird sie nach ihrem Gebrauch mit `delete_cond()`.

Entscheidet sich ein Thread nach der Überprüfung der Bedingung, daß er sich blockieren will, so ruft er `cond_wait()` auf. Dem Aufruf wird dabei zusätzlich zur Referenz auf die Bedingungsvariable noch eine Referenz auf das Mutexlock mitgegeben, daß für den gegenseitigen Ausschluß beim Überprüfen der Bedingung gesorgt hat. Dieses Mutexlock wird vor der tatsächlichen Blockierung freigegeben und nach der erfolgten Deblockierung, vor der Rückkehr aus der Funktion wieder gesetzt, um die Bedingung sofort wieder im gegenseitigen Ausschluß testen zu können.

Ändert ein Thread die Bedingung, so kann er mit `cond_signal()` einen oder durch `cond_broadcast()` alle wartenden Threads wieder aufwecken. Die Funktion `cond_empty()` kann dabei dem Test dienen, ob momentan Threads an der Bedingungsvariable blockiert sind.

A.3.2 Mutexlocks

```
mutex_p new_mutex(void);
void    delete_mutex(mutex_p mp);
void    mutex_lock(mutex_p mp);
void    mutex_unlock(mutex_p mp);
```

Mutexlocks dienen dazu den gegenseitigen Ausschluß innerhalb eines kritischen Abschnittes zu garantieren. Dabei wird dieser Abschnitt in `mutex_lock()` und `mutex_unlock()` eingeschlossen.

Auch hier werden die Mutexlocks wieder durch einen Aufruf `new_mutex()` angefordert und durch `delete_mutex()` wieder freigegeben.

A.3.3 Semaphoren

```
sem_p new_sema(int n);
void  delete_sema(sem_p sp);
sem_p sema_init(sem_p sp, int n);
void  sema_wait(sem_p sp);
void  sema_post(sem_p sp);
int   sema_trywait(sem_p sp);
```

Bei diesem Mechanismus handelt es sich um einfache zählende Semaphoren. Im Gegensatz zu den bisherigen Mechanismen erwartet hier die Funktion zum Anfordern des Mechanismus `new_semaphore()` einen Wert. Die Semaphore wird bei ihrer Erzeugung mit diesem Wert initialisiert, indem intern die Funktion `sema_init()` aufgerufen wird. Freigegeben wird die Semaphore wie üblich mit dem Aufruf `delete_sema()`.

Die Funktion `sema_wait()` versucht den Wert der Semaphore zu dekrementieren. Hatte sie aber vorher bereits den Wert Null, so wird der aufrufende Thread solange blockiert, bis ein anderer Thread die Semaphore mittels `sema_post()` inkrementiert.

Im Gegensatz zu `sema_wait()` wird der Thread bei `sema_trywait()` nicht blockiert falls die Semaphore bereits den Wert Null aufweist, sondern kehrt mit einem Fehlerwert zurück.

Barrieren

```
barr_p new_barrier(int no_of_threads);
void   delete_barrier(barr_p bp);
barr_p barrier_init(barr_p bp, int no_of_threads);
void   barrier_check(barr_p bp);
void   barrier_checkin(barr_p bp, int no);
void   barrier_checkout(barr_p bp, int no);
```

Barrieren dienen dazu, nach einer parallelen Berechnungsphase alle Threads zu synchronisieren. Dazu wird beim Anfordern einer neuen Barriere die Anzahl der Threads angegeben, die sich daran synchronisieren wollen. Intern wird für die notwendige Initialisierung wieder die Funktion `barrier_init` verwendet.

Anschließend bestehen zwei verschiedene Verwendungsmöglichkeiten, die aber nicht gleichzeitig für eine Barriere verwendet werden dürfen:

- Bei Verwendung des Befehls `barrier_check()` werden alle aufrufenden Threads solange blockiert, bis die bei der Anforderung des Mechanismus angegebene Anzahl an Threads den Befehl aufgerufen hat. Dann werden alle blockierten Threads deblockiert.

- Bei der Verwendung der Befehle `barrier_checkin()` und `barrier_checkout()` erfolgt die eigentliche Synchronisation in zwei Stufen. Alle Threads, die diese Funktionen aufrufen müssen eine eindeutige Nummer mitliefern. Der Thread mit der Nummer Null wird als Masterthread bezeichnet. Am Ende einer parallelen Phase verwenden alle Threads den Aufruf `barrier_checkout()`, wobei sich der Masterthread solange blockiert, bis alle Slaves die Funktion aufgerufen haben. Vor dem Beginn der nächsten parallelen Phase warten dann alle Slaves am Aufruf `barrier_checkin()`, bis der Masterthread angekommen ist. Durch diese Aufrufreihenfolge wird eine vollständige Synchronisation der Threads erreicht.

Beide Verwendungsmöglichkeiten haben die Eigenschaft, daß die Barriere nach einer erfolgreichen Synchronisation automatisch für den nächsten Durchlauf initialisiert wird, so daß kein expliziter Aufruf von `barrier_init()` notwendig ist.

Literaturverzeichnis

[AAC+92] Gail Alverson, Robert Alverson, David Callahan, et al: Exploiting heterogeneous parallelism on a multithreaded multiprocessor. *6th ACM International Conference on Supercomputing*, Seite 188–197, Juli 1992.

[ALL89] T.E. Anderson, E.D. Lazowska, H.M. Levy: The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12), Dezember 1989.

[And90] T.E. Anderson: The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Januar 1990.

[BB95] James M. Barton, Nawaf Bitar: A scalable multi-discipline, multiple-processor scheduling framework for IRIX. In Dror G. Feitelson and Larry Rudolph: *Job Scheduling Strategies for Parallel Processing*, Vol. 949, Seite 45–69. Springer, 1995.

[Bel95a] Frank Bellosa: Memory conscious scheduling and processor allocation on NUMA architectures. Technical report, IMMD IV, Friedrich-Alexander Universität Erlangen-Nürnberg, Juni 1995.

[Bel95b] Frank Bellosa: The performance implications of locality information usage in shared-memory multiprocessors. Technical report, IMMD IV, Friedrich-Alexander Universität Erlangen-Nürnberg, 1995.

[Bla90a] D.L. Black: *Scheduling and Resource Management Techniques for Multiprocessors*. PhD thesis, Carnegie-Mellon University, July 1990.

[Bla90b] D.L. Black: Scheduling support for concurrency and parallelism in the MACH operating system. Technical report, Carnegie Mellon University, Pittsburgh, PA 15213, 1990.

[Bro77] Mark Robbin Brown: *The Analysis of a Practical and Nearly Optimal Priority Queue*. PhD thesis, Stanford University, Februar 1977.

[DC88] R. Draves, E. Cooper: C Threads. Technical report, Carnegie Mellon University, Juni 1988.

[FL92] R.J. Fowler, L.I. Kontothanassis: Improving processor and cache locality in fine-grained parallel computations using object-affinity scheduling and continuation passing. Technical Report 411, University of Rochester, Rochester, New York, Juni 1992.

[Hau95] Andrea Hauth: Vergleichende Untersuchung von Threadbibliotheken auf ihre Eignung für numerische und systemnahe Anwendungen. Studienarbeit, IMMD IV, Friedrich-Alexander Universität Erlangen-Nürnberg, April 1995.

[HL94] Babak Hamidzadeh, David J. Lilja: Self-adjusting scheduling: An on-line optimization technique for locality management and load balancing. In *International Conference on Parallel Processing*, Vol. II, Seite 39–46, 1994.

[Inc94] Sun Microsystems Inc.: Pthreads and Solaris threads. A comparison of two user-level threads APIs. Technical report, SunSoft, Mai 1994.

[Kep93] D. Keppel: Tools and techniques for building fast portable threads packages. Technical report, University of Washington, Mai 1993.

[KLMO91] A.R. Karlin, Kai Li, M.S. Manasse, S. Owicki: Empirical studies of competitive spinning for a shared-memory multiprocessor. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Seite 41–55, Oktober 1991.

[Knu73] D.E. Knuth: *The Art of Computer Programming*, Vol. 3. Addison-Wesley, 1973.

[Kop95] Christoph Koppe: Sleeping threads: A kernel mechanism for support of efficient user level threads. *International Conference of Parallel and Distributed Computing and Systems*, 1995.

[Mar93] Evangelos Markatos: *Scheduling for Locality in Shared-Memory Multiprocessors.* PhD thesis, University of Rochester, Rochester, New York, 1993.

[MCS90] J.M. Mellor-Crummey, M.L. Scott: Algorithms for scalable synchronization on shared-memory multiprocessors. Technical Report 342, University of Rochester, Department of Computer Science, 1990.

[ML91] E.P. Markatos, T.J. LeBlanc: Load balancing vs. locality management in shared-memory multiprocessors. Technical Report 399, University of Rochester, Computer Science Department, Rochester New York 14627, Oktober 1991.

[ML93] Evangelos Markatos, Thomas J. LeBlanc: Locality-based scheduling for shared-memory multiprocessors. Technical report, Computer Science Department, Rochester , New York, 1993.

[MVZ93] C. McCann, R. Vaswani, J. Zahorjan: A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):147–178, Mai 1993.

[OW93] T. Ottmann, P. Widmayer: *Algorithmen und Datenstrukturen.* BI Wissenschaftsverlag, Reihe Informatik, 1993.

[Red95] U. Reder: Implementierung eines effizienten Prozeßumschalters auf Benutzerebene. Studienarbeit, IMMD IV, Friedrich Alexander Universität Erlangen-Nürnberg, 1995.

[Sch93] H.R. Schwarz: *Numerische Mathematik.* Teubner Verlag, Stuttgart, 1993.

[SL89] M. Squillante, E.D. Lazowska: Using processor-cache affinity information in shared-memory multiprocessor scheduling. Technical report, Department of Computer Science, University of Washington, 1989.

[SW95] Patrick G. Sobalvarro, William E. Weihl: Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In Dror G. Feitelson and Larry Rudolph: *Job Scheduling Strategies for Parallel Processing*, Seite 107–126. Springer, 1995.

[TS87] D. Thiebaut, H.S. Stone: Footprints in the cache. *ACM Transactions on Computer Systems*, 5:305–329, 1987.

[TTG95] J Torrellas, A. Tucker, A. Gupta: Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, Februar 1995.

[Tuc93] A. Tucker: *Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors.* Dissertation, Stanford University, 1993.

[VZ91] R. Vaswani, J. Zahorjan: The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. *Proceedings of the 13th Symposium on Operating Systems Principles*, Seite 26–40, Oktober 1991.