Diplomarbeit
'Träge Release-Konsistenz'

Peter Weisse
weisse@informatik.hu-berlin.de

Humboldt Universität Berlin
Fachbereich Informatik, Lehrstuhl für
Softwaretechnik und Theorie der Programmierung II

Betreuer: Dr. Frank Müller

17. Februar 2002

Für Lisa und Conny

Eigenständigkeitserklärung

Hiermit erkläre ich, daß die vorliegende Arbeit 'Träge Release-Konsistenz' von mir selbst und ohne jede unerlaubte Hilfe nur unter Verwendung der angegebenen Quellen angefertigt wurde.

Berlin, den 17. Februar 2002 Peter Weisse

Einverständniserklärung

Hiermit erkläre ich mein Einverständnis, daß diese Arbeit in der Bibliothek des Institutes für Informatik der Humboldt-Universität zu Berlin öffentlich ausgelegt wird.

Berlin, den 17. Februar 2002 Peter Weisse

Thesen

- Der Ablauf eines Programmes unter Verwendung des MRMW-Protokolles ist oft schneller, als wenn ein MRSW-Protokoll benutzt wird.
- Der resultierende Geschwindigkeitsgewinn wird durch eine Einschränkung des Programmierers erkauft.
- Ein *gleichzeitiger* schreibender Zugriff auf eine identische Speicherstelle von zwei verschiedenen Threads kann es nicht geben.
- Es besteht ein erhöhter Implementationsaufwand.
- Es besteht extreme Problemabhängigkeit.
- Bei Variation kleinster problemunabhängiger Programmablaufparameter wie z.B. der Speicherseitengröße ändert sich die Performance drastisch.
- Es besteht ein Zusammenhang zwische der Speicherseitengröße und maximaler Größe zu versendender Nachrichten.
- Es kann keine befriedigende Lösung zur Speicherfreigabe bereits erzeugter Diff-Zeichenketten angegeben werden.
- Die Implementation eines bestehenden Problemes kann so geändert werden, dass sich die Performance des Programmablaufes unter Verwendung des MRSW-Protokolles an die des MRMW-Protokolles annähert.

Zusammenfassung

Ein verteiltes System ist eine Menge voneinander unabhängiger Computer, die dem Benutzer des Systems den Eindruck vermitteln, es handle sich um einen einzigen Computer [30].

Ein verteiltes System, welches einen einzigen virtuellen Adreßraum benutzt, wird als DSM-System (*Distributed Shared Memory, verteilter gemeinsam genutzter Speicher*) bezeichnet[5, 6, 7, 27, 28]. Jeder beteiligte Prozeß nimmt an, über den gesamten Speicher allein zu verfügen. Greift er auf eine Speicherseite zu, die er real nicht besitzt, erfolgt ein PAGEFAULT-Fehler. Dieses Verhalten ist ähnlich der Auslagerung von realem Arbeitsspeicher auf eine Festplatte durch ein modernes Betriebssystem. Durch den PAGEFAULT-Fehler wird die benötigte Speicherseite von seinem momentanen Besitzer angefordert und übertragen. Der Programmablauf des ursprünglichen Prozesses wird an der unterbrochenen Position fortgesetzt, ohne dass der Prozeß von der Unterbrechung erfahren hätte.

Die Performance einer DSM-Struktur ist stark abhängig von der Nachrichtenmenge, die über das Netzwerk übertragen werden muss. Eine Strategie, den Netzverkehr zu vermindern, ist die Replikation gemeinsam genutzter Variablen auf mehreren Maschinen. Dadurch wird das Problem der gemeinsamen Nutzung von Variablen darauf reduziert, wie verschiedene Kopien einer Menge von Datenstrukturen konsistent gehalten werden können. Dazu werden verschiedene Konsistenzmodelle definiert.

Gegenstand dieser Arbeit ist eine Erweiterung der Funktionalität des bestehenden DSM-Systems [27, 28] um das MRMW-Protokoll (*Multi-Reader Multi-Writer, viele Leser, viele Schreiber*). Dies wird durch die Benutzung des trägen Release-Konsistenzmodelles erreicht. Es werden der gegenseitige Ausschluß[29] und Barrieren mit Hilfe des MRMW-Protokolles[3] implementiert.

Eine erfolgte Änderung am bestehenden DSM-System ist die Erzeugung eines Moduls, welches Zeitstempel verwalten kann. Zusätzlich wurden die Module, die die Kopien der Speicherseiten halten und die Veränderungen dieser Seiten kodieren und verschicken, den neuen erweiterten Erfordernissen angepaßt.

Ein Teil dieser Aufgaben wurden im Rahmen der gleichnamigen Studienarbeit zu diesem Thema erledigt. Zusätzlich zur Implementation wurden

umfangreiche Vergleichsmessungen durchgeführt, in denen die Laufzeiteffizienz des MRMW-Protokolles mit dem bereits bestehenden MRSW-Protokoll verglichen wurde.

Nicht behandelt wurden alle Aspekte einer zufriedenstellenden Speicherfreigabe der erzeugten Diffstrings. Eine Zeichenkette, die die Veränderungen eines Prozesses an einer Speicherseite beinhaltet, kann erst dealloziiert werden, wenn sie von allen beteiligten Prozessen behandelt und in die jeweiligen Kopien der Speicherseiten eingefügt wurde. Nur nach erfolgreichem Abschluß einer Barriere, an der alle Prozesse beteiligt waren, können alle jemals erzeugten Diffstrings ersatzlos zerstört werden.

Inhaltsverzeichnis

1 Einführung

Die nebenläufige und parallele Programmierung eröffnet neue Möglichkeiten, Algorithmen zu entwerfen und reale Abläufe zu modellieren.

Ein verteiltes System ist eine Menge voneinander unabhängiger Computer, die dem Benutzer des Systems den Eindruck vermitteln, es handle sich um einen einzigen Computer [30, 31]. Die Grundlage dieser Systeme sind Multicomputer bzw. Multiprozessoren. Entgegengesetzt zu Multiprozessoren sind Multicomputer einfacher aufzubauen und schwieriger zu programmieren. Li (1986)[25] und Li und Hudak (1989)[24] definierten ein verteiltes DSM-System auf der Basis von Multicomputern, welches einfach aufzubauen und zu programmieren war.

Ein verteiltes System, dass einen einzigen virtuellen Adreßraum benutzt, wird als DSM-System (*Distributed Shared Memory, verteilter gemeinsam genutzter Speicher*) bezeichnet [27, 28, 30, 31]. Eine Menge von Workstations, die über ein LAN verbunden sind, benutzen einen virtuellen gemeinsamen Adressraum. Der Zugriff auf eine lokal vorhandene Seite erfolgt hardwaremäßig, mit voller Geschwindigkeit. Der Versuch, auf eine Seite einer anderen Maschine zuzugreifen, bewirkt einen Hardware-Seitenfehler, wodurch ein Trap in das Betriebssystem erzeugt wird. Das Betriebssystem sendet daraufhin eine Nachricht an die entfernt gelegene Maschine, die die benötigte Seite sucht und sie an den anfordernden Prozessor sendet. Die zuerst fehlgeschlagene Anweisung wird wiederholt und kann jetzt abgeschlossen werden.

Die Performance einer DSM-Struktur ist stark abhängig von der Nachrichtenmenge, die über das Netzwerk übertragen werden muss. Eine Strategie, den Netzverkehr zu vermindern, ist die Replikation gemeinsam genutzter Variablen auf mehreren Maschinen. Dadurch wird das Problem der gemeinsamen Nutzung von Variablen darauf reduziert, wie verschiedene Kopien einer Menge von Datenstrukturen bzw. Seiten konsistent gehalten werden können. Dazu werden verschiedene Konsistenzmodelle definiert.

Gegenstand dieser Arbeit ist eine Erweiterung der Funktionalität des bestehenden DSM-Systems [27, 28] um das MRMW-Protokoll (*Multi-Reader Multi-Writer, viele Leser, viele Schreiber*). Dies wird durch die Benutzung des trägen Release-Konsistenzmodelles erreicht. Es werden der gegenseitige Ausschluß[29] und Barrieren mit Hilfe des MRMW-Protokolles[3] implementiert.

Diese Arbeit ist in folgende Abschnitte gegliedert. Der Abschnitt 2 auf Seite 7 erläutert die zentralen Begriffe dieser Arbeit und der verwendeten Semantik. Das Kapitel 3 auf Seite 9 geht allgemein auf Konsistenzmodelle und im Besonderen auf das implementierte Konsistenzmodell näher ein. Im Kapitel 4 auf Seite 21 wird über die Implementation des gegenseitigen Ausschlusses und im Abschnitt 5 auf Seite 29 über die Barrieren referiert. Abschnitt 6 auf Seite 35 zeigt die Veränderungen auf, die am bestehenden DSM-System nötig waren. Das zentrale Kapitel 7 auf Seite 55 vergleicht die Effizienz des Programmablaufes bezüglich des MRMW- und MRSW-Protokolles. Zum Schluß wird im Kapitel 8 auf Seite 87 eine Zusammenfassung dieser Arbeit und ein Ausblick auf zukünftige Aufgabenstellungen umrissen.

Nicht behandelt werden alle Aspekte einer zufriedenstellenden Speicherfreigabe. Im Besonderen besteht das Problem der Dealloziierung der erzeugten Diffs. Unter der Voraussetzung, dass alle Prozesse an einer Barriere teilnehmen, könnte dort die gesamte Garbage Collection durchgeführt werden. Nur in diesem Fall ist sichergestellt, dass alle beteiligten Prozesse die lokale Veränderung einer Speicherseite durch einen beliebigen anderen Prozess bereits in ihre Speicherseiten integriert haben. Im jetzigen DSM-System kann nur durch Segmentierung des Programmcodes mit Hilfe von Barrieren einer Verschwendung von Speicherplatz und steigendem Dekodierungsaufwand vorgebeugt werden. Eine automatische Speicherfreigabe von Diffstrings wäre eine Problemstellung, die in weiteren Arbeiten untersucht werden könnte.

2 Terminologie

In der Informatik ist die Verwendung von Begriffen aus dem Englischen sehr verbreitet. Diese Studienarbeit einer deutschen Universität versucht, so viele Begriffe wie möglich aus dem Deutschen zu verwenden. Problematisch sind Verben, deren Konjugation im Deutschen nicht festgeschrieben ist.

Die Begriffe, zu denen es keine prägnante deutsche Entsprechung gibt, wurden englisch belassen. Dazu gehören Prozess, Thread, Mutex.

Prozesse stehen für UNIX-Prozesse. Ein Prozess ist dabei die spezifische Laufzeitumgebung eines Programmes. Ein Prozess hat einen eigenen Kontrollfluß und Speicher. Er wird über eine Kombination aus Hostnamen und Portadresse identifiziert. Prozesse werden vom System geschedult, d.h sie erhalten CPU-Zeit nach einem vom System vorgegebenen Plan. Wenn ein Prozess die CPU wieder abgeben muß, dann findet ein Kontextwechsel statt. Hierbei müssen alle prozessrelevanten Informationen gespeichert werden.

Threads sind nebenläufige Einheiten, die gemeinsamen Speicher, aber getrennte Kontrollflüsse haben. Bei der betrachteten Bibliotheksimplementierung der POSIX-Threads sind Threads immer einem Prozess zugeordnet, soweit es sich um ein UNIX-ähnliches System handelt. Threads sind im Gegensatz zu Prozessen ,,leichtgewichtiger", d.h die Thread-relevanten Informationen, die für die Ausführung bzw. Unterbrechung wichtig sind, haben einen kleineren Umfang. Da Threads einem Prozess zugeordnet sind, muß man bei einem Thread-Kontextwechsel keine prozessspezifischen Daten speichern. Wenn der zugeordnete Prozess die CPU abgeben muß, dann werden vom System alle prozessspezifischen Daten gerettet. Threads können darüber hinaus Multiprozessoren mit gemeinsamen Speicher ausnutzen, um die Programmausführung zu beschleunigen.

Als Mutex wird hier das Objekt bezeichnet, dass zur Realisierung eines gegenseitigen Ausschlusses (*mutual exclusion*)[29] benutzt wird. Durch ein mutex lock kann dieser Mutex (für kurze Abschnitte) von einem Thread angefordert werden und mittels mutex unlock wieder abgegeben werden.

3 Grundlagen

Die Möglichkeiten zur Synchronisation von Prozessen in verteilten Systemen sind vielfältig. Sie sind in jedem Fall komplizierter als in zentralen Systemen, da verteilte Systeme verteilte Algorithmen verwenden. Es ist auf keinen Fall wünschenswert, allgemeine oder spezielle Informationen an zentraler Stelle zu halten. Verteilte Algorithmen besitzen im allgemeinen folgende Eigenschaften[30, 31, 6, 7]:

- Alle relevanten Informationen sind auf mehrere Maschinen verteilt.
- Prozesse treffen Entscheidungen nur aufgrund lokaler Informationen.
- Ein einziger Ausfallpunkt im System sollte vermieden werden.
- Es existiert keine gemeinsame Uhr oder andere präzise globale Zeit.

Es wird zwischen absoluter und logischer Zeitsynchronisation unterschieden.

Absolute Zeitsynchronisation ist die Realisierung einer globalen Uhrzeit. Besonders für Echtzeitsysteme existiert eine große Anzahl von Algorithmen, die physikalische Uhren auf verteilten Systemen implementieren. Dies ist in verteilten Systemen aber nur mit großem Aufwand realisierbar.

Eine logische Synchronisation von Uhren ist möglich, wenn das Fehlen der Synchronisation nicht auffällt [30, 31]. Das folgende Unterkapitel 3.1 auf Seite 11 beschreibt ausführlich die Grundlagen der logischen Synchronisation.

Die Anzahl der zu verschickenden Nachrichten beeinflußt die Performance des Programms wesentlich. Durch die Einführung verschiedener Konsistenzmodelle kann die Menge von Nachrichten vermindert werden. Schwächere Konsistenzmodelle bedingen zusätzliche Einschränkungen, die der Programmierer zu beachten hat.

Abbildung 1 zeigt die verschiedenen Typen des Zugriffs auf gemeinsame Variablen in verteilten Systemen.

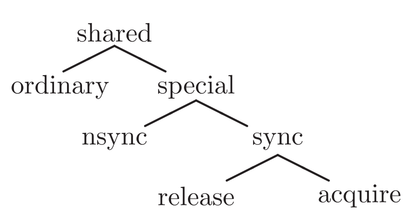

Abbildung 1: Zugriffstypen auf den Speicher bei verteilten Anwendungen

3.1 Synchronisation

Ein nichttriviales Problem ist die Synchronisation von Prozessen in verteilten Systemen.

In einem Computer mit einem Prozessor kann die interne Uhr als absolute Zeitsynchronisation benutzt werden. Dabei ist es unwichtig, ob diese Uhr von der realen Zeit abweicht. Wenn alle Prozesse auf der Maschine dieselbe Uhr verwenden, sind sie doch intern konsistent.

Sobald jedoch mehrere CPUs mit jeweils einer eigenen Uhr eingesetzt werden, verändert sich diese Situation. Es kann nicht garantiert werden, dass alle Uhren mit genau derselben Frequenz laufen. Diese *Uhrabweichung* bedingt unterschiedliche Auslesungen der Werte und somit die Ursache für Synchronisationsfehler.

In [20, 21, 22] und [23] stellte Lamport einen klassischen Algorithmus vor, der zeigte, dass die Uhrsynchronisation möglich ist. Er legte dar, dass die Uhrsynchronisation nicht absolut sein muß. Wenn zwei Prozesse miteinander interagieren, dann ist es nicht notwendig, dass ihre Uhren synchronisiert werden, weil das Fehlen der Synchronisation nicht auffällt und somit keine Probleme verursacht. Darüber hinaus hat er gezeigt, dass es normalerweise nicht darauf ankommt, dass alle Prozesse sich genau auf eine Zeit einigen, sondern dass sie sich auf die Reihenfolge einigen, in der die Ereignisse auftreten. Für eine bestimmte Klasse von Algorithmen ist es die interne Konsistenz, auf die es ankommt, nicht, ob sie der realen Zeit entsprechen. Für diese Algorithmen werden die Uhren auch als *logische* Uhren bezeichnet.

Um logische Uhren zu beschreiben, definierte Lamport[20] eine Relation *passiert_vor*. Der Ausdruck $a \rightarrow b$ wird gelesen als *passiert_vor* und bedeutet, dass alle Prozesse übereinstimmen, dass zuerst Ereignis a passiert und danach Ereignis b.

- Wenn a und b Ereignisse im selben Prozess sind und a vor b geschieht, dann gilt $a \rightarrow b$.
- Wenn Ereigniss a das Senden einer Nachricht von einem Prozess ist und Ereignis b das Empfangen dieser Nachricht in einem anderen Prozess, dann trifft ebenfalls $a \rightarrow b$ zu. Eine Nachricht kann nicht empfangen werden, bevor sie gesendet wurde, weil es eine endliche Zeit dauert, bis die Nachricht angekommen ist.
- Die Transitivität dieser Relation gilt: wenn $a \rightarrow b$ und $b \rightarrow c$, dann $a \rightarrow c$.

Wenn zwei Ereignisse x und y in verschiedenen Prozessen auftreten, die keine Nachrichten austauschen (nicht einmal indirekt über Dritte), dann trifft $x \rightarrow y$ nicht zu, aber auch nicht $y \rightarrow x$. Diese Ergebnisse heißen *nebenläufig*, was bedeutet, dass man nichts darüber aussagen kann, welches dieser Ereignisse zuerst passiert ist.

Lamports Algorithmus folgt direkt aus der *passiert_vor*-Relation. Er gibt eine Methode an, die für jedes Ereignis a einen Zeitwert $C(a)$ definiert, über den sich alle Prozesse einig sind. Diese Zeitwerte müssen die Eigenschaft besitzen, dass, wenn $a \rightarrow b$ gilt, auch $C(a) < C(b)$ gilt. Mit anderen Worten, wenn a vor b geschieht, muß auch $C(a) < C(b)$ sein. Wenn a das Senden einer Nachricht von einem Prozess beschreibt und b den Empfang dieser Nachricht durch einen anderen Prozess, dann müssen $C(a)$ und $C(b)$ so angegeben werden, dass jeder die Werte von $C(a)$ und $C(b)$ mit $C(a) < C(b)$ akzeptiert. Darüber hinaus muß die Uhrzeit C immer ansteigen.

Wenn eine Nachricht ankommt und die Uhr des Empfängers einen Wert zeigt, der vor der Zeit liegt, zu der die Nachricht abgeschickt wurde, stellt der Empfänger seine Uhr nach vorne, damit sie einen Tick mehr als die Sendezeit anzeigt.

Mit dieser Methode existiert die Möglichkeit, allen Ereignissen in einem verteilten System eine Zeit zuzuordnen, die den folgenden Bedingungen genügt:

- Wenn a in demselben Prozess vor b passiert, dann gilt $C(a) < C(b)$.
- Wenn a und b das Senden und Empfangen einer Nachricht darstellen, dann gilt $C(a) < C(b)$.
- Für alle Ereignisse a und b gilt $C(a) \neq C(b)$.

Die logischen Uhren wurden für das Modell der trägen Release-Konsistenz im bestehenden DSM-System implementiert (siehe Kapitel 6.1 auf Seite 37).

3.2 Konsistenz

Ein Konsistenzmodell ist ein Vertrag zwischen Software und Speicher[2]. Wenn die Software bereit ist gewissen Regeln zu gehorchen, verspricht der Speicher korrekt zu arbeiten. Verletzt die Software diese Regeln, gilt der Vertrag nicht mehr und die Korrektheit der Speicheroperationen ist nicht länger sichergestellt. Es gibt ein großes Spektrum an möglichen Verträgen, von solchen, die der Software nur kleine Einschränkungen auferlegen, bis hin zu solchen, die eine normale Programmierung unmöglich machen. Die Software mit den geringeren Einschränkungen bieten keine so gute Performance wie diejenigen mit den größeren Forderungen.

Folgende Konsistenzmodelle werden unterschieden: [30, 31, 2, 6, 7, 8]

- Strenge Konsistenz
- Sequentielle Konsistenz [20, 21, 22, 23]
- Kausale Konsistenz [14]
- PRAM-Konsistenz [26]
- Prozessor-Konsistenz [13]
- Schwache Konsistenz [11, 10]
- Release-Konsistenz [12, 17, 16, 19, 18]
- Eintritts-Konsistenz [9, 15, 1]
- Bereichs-Konsistenz [15]

In dieser Arbeit wird auf die Release-Konsistenz näher eingegangen. Für alle anderen Konsistenzmodelle wird auf die im Quellenverzeichnis genannte Literatur verwiesen.

3.3 Release-Konsistenz

Im Release-Konsistenzmodell erfolgt eine wesentliche Unterscheidung zwischen Ein- und Austritt in den kritischen Bereich. *Acquire*-Zugriffe teilen dem Speichersystem mit, dass ein kritischer Abschnitt betreten werden soll. *Release*-Zugriffe besagen, dass ein kritischer Bereich gerade verlassen wurde. In jedem Falle ist der Programmierer dafür verantwortlich, den expliziten Code in sein Programm aufzunehmen, der angibt, wann sie ausgeführt werden sollen.

Bei der Release-Konsistenz ist es außerdem möglich, *Barrieren* statt kritischer Bereiche zu verwenden. Eine Barriere ist ein Synchronisierungs-Mechanismus, der einen Prozess daran hindert, Phase *n+1* eines Programmes zu beginnen, bis alle Prozesse Phase *n* beendet haben. Wenn ein Prozess an eine Barriere kommt, dann muß er warten, bis alle anderen Prozesse auch dort angekommen sind. Wenn der letzte Prozess ankommt, dann werden alle gemeinsam genutzten Variablen synchronisiert, und alle Prozesse werden fortgesetzt. Die Ankunft an der Grenze bedeutet *Release*, ihr Verlassen bedeutet *Acquire*.

Neben diesen Synchronisierungs-Zugriffen ist auch das Lesen und Schreiben gemeinsam genutzter Variablen möglich. Acquire und Release müssen nicht auf den gesamten Speicher angewendet werden. Statt dessen können sie auch eingesetzt werden, um nur bestimmte gemeinsam genutzte Variablen zu bewachen. In diesem Falle werden nur diese Variablen konsistent gehalten. Die gemeinsam genutzten Variablen, die konsistent gehalten werden, nennt man auch *bewacht*.

Ein verteilter gemeinsam genutzter Speicher ist dann Release-konsistent, wenn er folgenden Regeln gehorcht:

- Bevor ein Zugriff (Read/Write) auf eine gemeinsam benutzte Variable ausgeführt wird, müssen alle zuvor abgesetzten Acquires erfolgreich abgeschlossen sein.

- Bevor ein Release ausgeführt werden darf, müssen alle vorherigen Read- und Writevorgänge des Prozesses abgeschlossen sein.

- Die Acquire- und Release-Zugriffe müssen dem Modell der sequentiellen Konsistenz folgen [17, 16, 19, 18]. In [30, 31] wird die sequentielle Konsistenz abgeschwächt und nur eine Prozessor-Konsistenz gefordert.

In Abbildung 2 wird eine gültige Ereignisabfolge für die Release-Konsistenz dargestellt. Prozess P1 setzt ein Acquire ab, verändert eine gemeinsam genutzte Variable zweimal und führt dann ein Release aus. Prozess P2 setzt ein Acquire ab und liest x. Dabei ist sichergestellt, dass der Wert von x zurückgegeben wird, den es zum Zeitpunkt des letzten Release hatte, nämlich 2. Wenn das Acquire vor P1's Release erfolgt, dann wird es verzögert, bis das Release erfolgt ist. Wenn P3 kein Acquire absetzt, bevor es eine gemeinsam genutzte Variable liest, ist der Speicher nicht verpflichtet, den aktuellen Wert von x zurückzugeben, so daß die Rückgabe von 1 erlaubt ist.

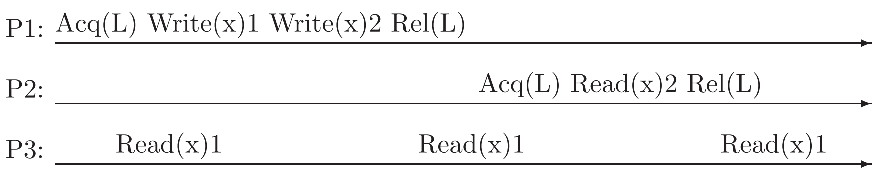

Abbildung 2: Eine gültige Ereignisabfolge für die Release-Konsistenz

Die Einschränkung der Software unter Verwendung aller schwachen Konsistenzen besteht in der Verpflichtung der verschiedenen Prozesse, zwischen zwei Synchronisationspunkten stets nur auf disjunkte Daten schreibend zuzugreifen. Darauf hat der Programmierer zu achten.

Es wird zwischen der eifrigen Release-Konsistenz (ERC, *Eager Release Consisteny*) und der trägen Release-Konsistenz (LRC, *Lazy Release Consisteny*) unterschieden.

Eifrige Release-Konsistenz

Der Vertrag zwischen der Software und dem Speicher besagt, dass, wenn die Software ein Acquire absetzt, der Speicher sicherstellt, dass alle lokalen Kopien der bewachten Variablen aktualisiert werden, so dass sie gegebenenfalls mit den entfernt gelegenen Variablen konsistent sind. Wenn ein Release erfolgt, dann werden die bewachten Variablen, die verändert wurden, den anderen Maschinen mitgeteilt. Ein Acquire garantiert nicht, dass alle lokalen Veränderungen unmittelbar allen anderen Maschinen mitgeteilt werden. Analog müssen bei einem Release nicht unbedingt die Veränderungen auf anderen Maschinen zur Kenntnis genommen werden.

Träge Release-Konsistenz

Das Modell der trägen Release-Konsistenz besagt, dass lokale Modifikationen einer Seite zwischen Acquire und Release nicht bei der Releaseoperation global propagiert werden. Dies erspart unnötigen Kommunikationsaufwand, da nicht alle Prozesse, die eine Kopie der Seite besitzen, diese Modifikationen auch verwenden wollen. Statt dessen versucht der Prozess bei einem Acquire, den aktuellen Wert der Variablen von den Maschinen, auf denen er sich befindet, zu erhalten. Um festzustellen, welche Variablen übertragen werden müssen, wird ein Zeitstempel-Protokoll benutzt. Bei einem Acquire werden alle *vorherigen* Veränderungen der Seite erfragt. Der Term *vorherig* ist im Sinne der *Happened-before-once*-Relation zu interpretieren.

Die Happened-before-once Relation

Sie beschreibt die partielle Ordnung, beschrieben mit $\xrightarrow{hb1}$, mit der Synchronisationsoperationen, wie Acquire, Release und Speicherzugriffe geordnet sind:

- Wenn a1 und a2 Acquire, Release oder Speicherzugriffe auf dem gleichen Prozessor sind, und a1 geschieht vor a2 in der Programmreihenfolge, dann ist a1 $\xrightarrow{hb1}$a2.
- Wenn a1 ein Release auf Prozessor p1 und a2 das korrespondierende Acquire auf Prozessor p2, dann ist a1 $\xrightarrow{hb1}$a2. Für einen Lock korrespondiert ein Acquire mit einem Release, wenn zwischen ihnen kein Acquire bzw. Release existiert. Für eine Barriere korrespondiert ein Acquire mit einem Release, wenn das Acquire und das Release der gleichen Barriereninstanz zugeordnet sind.
- Wenn a1 $\xrightarrow{hb1}$a2 und a2 $\xrightarrow{hb1}$a3 , dann a1 $\xrightarrow{hb1}$a3.

3.4 Eifrige vs. Träge Release-Konsistenz

Im direkten Vergleich wird die höhere Effizienz der trägen Release-Konsistenz gegenüber der eifrigen Release-Konsistenz deutlich.

Tabelle 1 stellt die Unterschiede der beiden Protokolle in knapper Form dar.

	ERC	**LRC**
Datenübergabe	bei Release	bei Acquire
Kommunikation	mit n-1 Prozessen	mit 1 Prozess
Effizienz	immer Verbreitung	nur wenn nötig

Tabelle 1: Vergleich ERC und LRC

Die Datenübergabe erfolgt bei der eifrigen Release-Konsistenz durch die Verbreitung der eigenen Änderungen bei Ausführung einer Release-Operation. Dabei wird mit allen Prozessen kommuniziert, die eine Kopie der Speicherseite besitzen. Es existiert keine Unterscheidung, ob die aktualisierten Kopien jemals benötigt werden.

Mit dem Protokoll der trägen Release-Konsistenz werden bei Aufruf der Acquire-Operation alle Änderungen bis zu diesem Zeitpunkt beschafft. Es wird nur mit dem einen Prozess, der das letzte Release durchführte, kommuniziert. Dadurch wird implizit die Effizienz erhöht, da nur bei beabsichtigter Benutzung der betreffenden Speicherseite über vergangene Modifikationen informiert wird.

Es sei dabei auf die zu beachtenden Einschränkungen des Programmierers in Kapitel 3.3 auf Seite 17 verwiesen.

Gravierende Unterschiede zwischen ERC und LRC werden besonders in dem folgenden Beispiel einer schlecht programmierten Schleife mit einem Acquire-Datenzugriff-Release-Tupel im Schleifenrumpf deutlich (siehe Abbildung 3). Viele Prozesse schreiben wiederholt unabhängig voneinander auf getrennte Speicherbereiche. Solange die Prozesse sich in dieser Schleife befinden, wäre kein Informationsaustausch untereinander nötig.

Bei Existenz des ERC-Modells müssen bei jedem Release die modifizierten Variablen an die anderen Prozessoren weitergegeben werden. Dies verursacht unnötige Bandbreitenverschwendung.

```
P[i]: i = 1..n
    Solange ( Bedingungsvariable )  {
      Acquire(L)
      x[i] = ...;    // Write(x[i])
      Release(L)
    }
```

Abbildung 3: Eifrige vs. Träge Release-Konsistenz: Beispiel

Mit Hilfe des LRC-Modells wird bei Release nichts ausgeführt. Bei dem folgenden Acquire wird festgestellt, dass die aktuellen Daten schon vorhanden sind und es werden keine Nachrichten erzeugt. Wiederholte Acquire-Release-Paare des gleichen Prozessors sind bei fehlender Konkurrenz von außen kostenlos.[6, 7]

4 Gegenseitiger Ausschluß

Gegenseitiger Ausschluß[29] ist ein wichtiges Konzept, um den Zugriff auf gemeinsam genutzte Ressourcen in verteilten Systemen zu schützen. Eine Anzahl von Prozessen synchronisiert ihren Zugriff auf gemeinsame Daten durch den Eintritt in eine kritische Sektion. Die verteilten Algorithmen müssen, um korrekt zu sein, den gegenseitigen Ausschluß (mutual exclusion) garantieren, frei von Verklemmungen sein (deadlock free) und das Verhungern einzelner Prozesse ausschließen (absence of starvation).

Nachteile strengerer Konsistenzmodelle

In verteilten Systemen sind sie in der Hauptsache über die Performance des Programmes zu spüren. Ein wichtiger Faktor bezüglich der Geschwindigkeit ist die Anzahl verschickter Nachrichten. Im Vergleich zum hardwaremäßigen direktem Zugriff ist das Senden und Empfangen von Nachrichten bzw. Signalen erheblich langsamer. Die durch die Definition neuer Konsistenzmodelle eingesparten Nachrichten erreichen den größten Geschwindigkeitsgewinn.

Ein Problem der sequentiellen Konsistenz ist in Abbildung 4 ersichtlich.

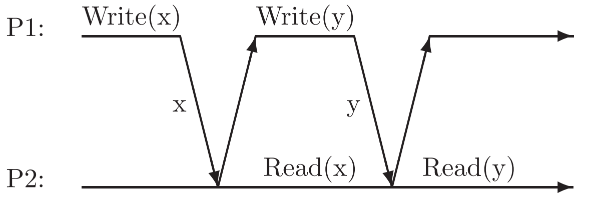

Abbildung 4: Performance-Problem: Sequentielle Konsistenz

Jeder Schreibzugriff auf den verteilten gemeinsam genutzten Speicher ist *sofort* allen anderen Prozessen sichtbar. Durch den Versand zu vieler, eigentlich unnötiger Nachrichten und der daraus folgenden Latenzzeit ergibt sich ein hoher Zeitverbrauch. In diesem Beispiel liest der Prozess P2 diese Variablen, aber P2 wäre auch von den Veränderungen informiert worden, wenn er keinerlei Read-Operationen auf den Variablen durchgeführt hätte.

Ein wichtiger Design-Aspekt unter der Nutzung strengerer Konsistenzmodelle liegt in der Wahl der Speicherabschnittsgröße, die eine Seite darstellt. Die Verwendung von größeren Einheiten für DSM hat Vor- und Nachteile.

Der größte Vorteil ist, dass es nicht sehr viel länger dauert, größere Abschnitte zu übertragen - mehr Zeit wird benötigt für die Einrichtung einer Netzwerkübertragung. Die Übertragung von Daten in großen Einheiten reduziert die Anzahl der Transfervorgänge. Diese Eigenschaft ist wesentlich, weil viele Programme ein lokales Zugriffsmuster aufweisen, d.h., wenn ein Programm auf ein Wort einer Seite zugegriffen hat, dann ist es wahrscheinlich, daß es in allernächster Zukunft auf weitere Worte derselben Seite zugreifen wird.

Andererseits führt eine zu große effektive Seitengröße zu einem neuen Problem, der sogenannten *falschen gemeinsamen Nutzung*[4], wie in Abbildung 5 dargestellt.

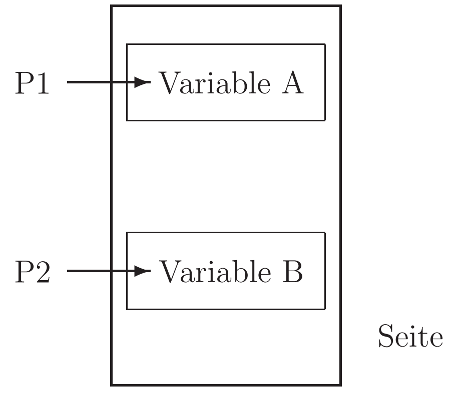

Abbildung 5: Performance-Problem: Falsche gemeinsame Nutzung

Es existiert eine Seite, die zwei nicht voneinander abhängige gemeinsam genutzte Variablen A und B enthält. Prozess P1 führt häufig Zugriffe auf A durch, wobei er die Speicherstelle liest und beschreibt. Analog verwendet Prozess P2 die Variable B. Unter diesen Umständen wird die Seite, die die beiden Variablen enthält, ständig zwischen den beiden Maschinen hin und her geschoben.

Das Problem dabei ist, dass die Variablen zwar nicht voneinander abhängig sind, aber weil sie sich zufällig auf derselben Seite befinden, erhält ein Prozeß, der eine davon anfordert, auch die andere. Je größer die effektiven Seiten sind, desto öfter tritt die falsche gemeinsame Nutzung auf, und je kleiner die effektiven Seiten sind, desto weniger tritt sie auf. Bei normalen virtuellen Speichersystemen gibt es nichts, was mit diesem Problem vergleichbar wäre.

Intelligente Compiler erkennen das Problem und platzieren die Variablen so im Adreßraum, dass sie die falsche gemeinsame Nutzung verhindern und die Performance verbessern können. Wenn die falsche gemeinsame Nutzung jedoch darin besteht, dass Prozess P1 ein Element aus einem Array und Prozess P2 ein anderes Element aus demselben Array benutzt, dann kann selbst ein intelligenter Compiler wenig gegen dieses Problem unternehmen.

Die Implementierung des gegenseitigen Auschlusses unter Benutzung des trägen Release-Konsistenzmodelles ist Grundlage dieser Arbeit. Das Ziel ist die Verringerung der Kommunikation der Prozesse untereinander unter Beibehaltung des verteilten Speichermodelles. Unter Verwendung des Release-Konsistenzmodelles ist die Wahl der Granularität von Speicherabschnitten gegenstandslos. Zugriffe auf eine Speicherseite beinhalten Locks auf einen Teildatenbereich dieser Seite. Damit ist das MRMW-Protokoll[3] eigentlich erweitertes MRSW-Protokoll, mit dem der Programmierer darauf zu achten hat, nicht gleichzeitig zwei Teildatenbereiche einer Seite zu beschreiben. Näheres zu diesen Einschränkungen ist in Kapitel 3.3 auf Seite 17 zu finden.

Die Modellierung des gegenseitigen Ausschlusses

Mit Hilfe zweier Arten von Synchronisierungs-Variablen bzw. -Operationen wird eine effizientere Implementation des gegenseitigen Ausschlusses ermöglicht. *Acquire*-Zugriffe teilen dem Speichersystem mit, dass ein kritischer Abschnitt betreten werden soll. *Release*-Zugriffe besagen, dass ein kritischer Bereich gerade verlassen wurde.

Ein Acquire beinhaltet einen Lock auf einen Teildatenbereich dieser Speicherseite. Beim Release wird dieser wieder freigegeben.

Ein Prozess P1 hat ein Acquire auf die Seite L mit einem Lock auf den Teildatenbereich x durchgeführt und ist noch nicht zum Release gekommen. Wenn jetzt ein zweiter Prozess P2 ein Acquire auf die gleiche Seite und auf den gleichen Teildatenbereich ausführt, bleibt er solange stehen, bis P1 das Release beendet hat. Danach erfragt P2 die Modifikationen der Seite von P1.

Bei folgender Situation (Abbildung 6) wird das Acquire von P2 solange verzögert, bis das Release von P1 durchgeführt wurde:

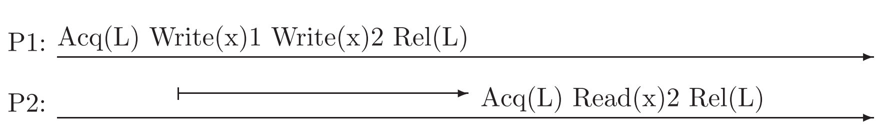

Abbildung 6: Acquire-Release-Modellierung eines geg. Ausschlusses

In der Realität werden sogar nur die Veränderungen des Teildatenbereichs x ausgetauscht.

Das MRMW-Protokoll

Das prinzipielle Vorgehen des MRMW-Protokolls erfolgt in der gleichen Reihenfolge.

- Jede Seite erhält ein Attribut, mit dem sie den Zustand UNMAPPED, READONLY , READWRITE oder INVALID annehmen kann.
- Zum Startzeitpunkt sind alle Seiten READONLY.
- Bei einem Schreibzugriff auf die Variable x (Write(x)) erfolgt eine Zugriffsverletzung, die DSM-Software erstellt eine Kopie der Seite und markiert diese als READWRITE (siehe Abbildung 7).

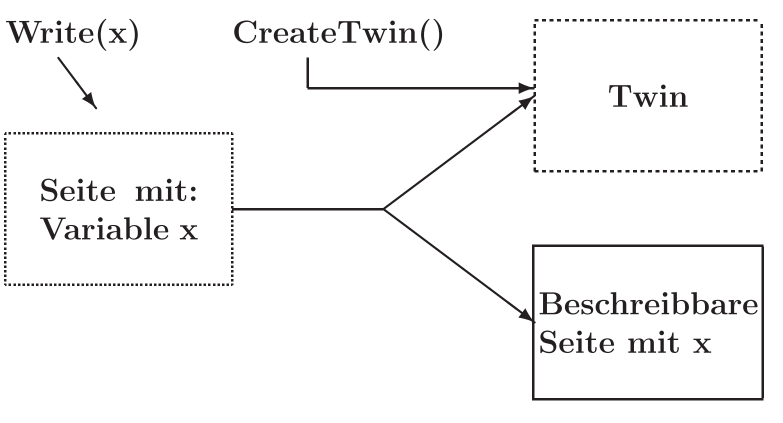

Abbildung 7: Erstellung eines Zwillings

Ein durchgezogener Rand einer Speicherseite der Abbildung 7 bedeutet, dass die Seite bescheibbar zur Verfügung steht. Gepunktete Seiten sind READONLY bzw. eigene Zwillinge, also nicht beschreibbare Seiten. Die Funktion *CreateTwin()* erstellt den Zwilling einer Seite und speichert ihn ab.

Alle folgenden Schreibzugriffe können ohne Eingriff der DSM-Software durchgeführt werden. Der Zugriff auf die jetzt lokal vorhandene Seite erfolgt hardwaremäßig, mit voller Geschwindigkeit.

Wie diese lokalen Veränderungen auf die Kopien der anderen Prozessoren übertragen werden, ist von der Verwendung des implementierten Release-Konsistenzmodelles abhängig.

Innerhalb der Release-Operation wird der Diff erstellt, der die Unterschiede zwischen Seite und Twin in komprimierter Form enthält. Dieser Vorgang ist in Abbildung 8 dargestellt.

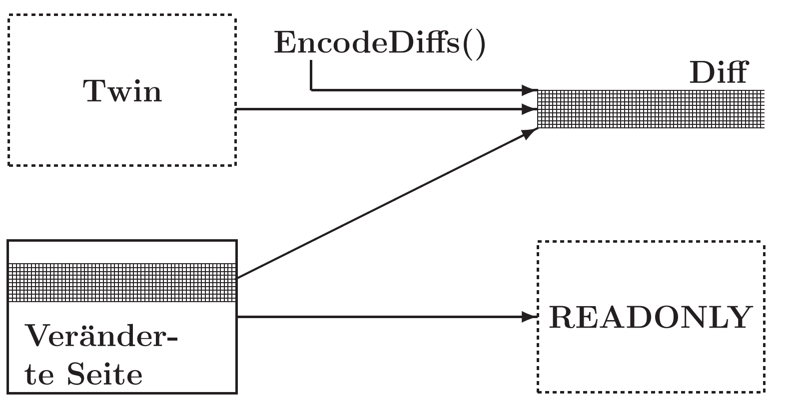

Abbildung 8: Erstellung eines Diffs

Diffs werden generiert, wenn

- Ein Prozessor die Modifikationen einer Seite erfragt
- Eine Schreibkopie von einem anderen Prozessor für eine Seite angefordert wird, die das Attribut READWRITE besitzt, d.h. sie ist modifiziert worden.
- Bei Entritt eines Prozesses in eine Barriere

Eifrige Invalidierung

Modifikationen des gemeinsamen Speichers werden sofort zum Zeitpunkt der Durchführung der Synchronisierungsoperation Release global sichtbar gemacht (siehe Abbildung 9). Dies geschieht unabhängig davon, ob die anderen Prozesse die Veränderungen bestimmter Speicherseiten überhaupt zur

Kenntnis nehmen wollen. Es wird versucht, alle existierenden Kopien einer modifizierten Seite zu invalidieren.

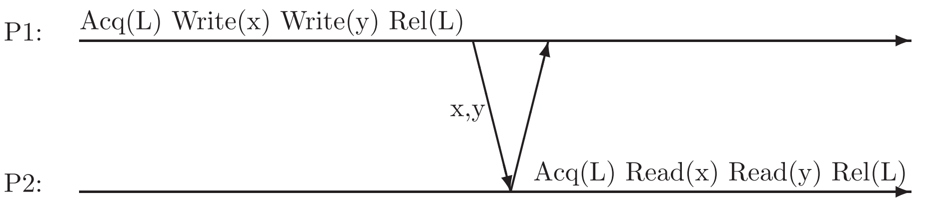

Abbildung 9: Eifrige Invalidierung

Besitzt die modifizierte Seite das Attribut

- READONLY, invalidieren
- READWRITE, die Kopie hängt eine Diff-Beschreibung seiner Modifikationen an die Antwortnachricht und invalidiert seine Kopie. Der empfangene Prozess fügt diese Diff-Beschreibungen in seine Kopie ein [16, 19].

Zur Identifizierung, wer eine Invalidierungsnachricht erhalten soll, wird eine ungefähre Copyset benutzt. Eine Copyset ist eine Menge, die beschreibt, welche Prozesse eine Kopie der Seite besitzen.

Träge Invalidierung

Modifikationen des gemeinsamen Speichers werden zum Zeitpunkt des *nachfolgenden* Acquire sichtbar gemacht. Der Prozess, der das Acquire durchführt, informiert sich über die *vorherigen* Modifikationen dieser Seite. Die Bedeutung der Terme *nachfolgenden* und *vorherig* enthält die *Happened-before-once*-Relation (siehe Kapitel 3.2 auf Seite 13).

Die Ausführung eines jeden Prozesses ist unterteilt in *Intervalle*. Das Intervall beschreibt die globale Zeit im Sinne der *passiert_vor*-Relation. Bei jedem Acquire bzw. Release beginnt ein neues Intervall. Intervalle von verschiedenen Prozessen sind partiell sortiert:

- Intervalle auf einem einzelnen Prozessor sind durch die Programmreihenfolge geordnet.

- Ein Intervall auf Prozessor p passiert vor einem Intervall auf Prozessor q, wenn das Acquire von q korrespondiert mit dem Release des Prozessors p.

Eine einzelner *Timestamp* besteht aus einer Prozessidentifikation und einem Intervall. Die Zeitmarke beschreibt einen Prozess auf einem über den Hostnamen identifizierbaren Rechner eindeutig. Der Prozess ist dort unter einem definierten Port zu erreichen. Seine logische Uhrzeit wird durch eine Intervallnummer bestimmt.

Die partielle Ordnung der Intervalle kann durch einen *Timestamp-Vektor* in jedem Intervall repräsentiert werden. Zur Konstruktion eines Timestamp-Vektors sind im Kapitel 6.1 auf Seite 37 ausführliche Informationen zu finden. Ein Timestamp-Vektor existiert einmal für jeden beteiligten Prozess. Er verwaltet die eigene Zeitmarke und enthält einen Eintrag für jeden Prozessor. Der Eintrag für Prozessor p im Intervall i ist gleich i. Der Eintrag für Prozessor $q \neq p$ beschreibt das größte Intervall des Prozessors q, dass vor dem jetzigen Intervall des Prozessors p, entsprechend der partiellen Ordnung, liegt. Dieser q-te Eintrag j ist so zu interpretieren, dass j das jüngste Intervall ist, in dem p ein Update von q erhalten hat. Ein Prozessor generiert bei einem Acquire einen neuen Timestamp-Vektor , indem er das paarweise Maximum seines vorherigen Timestamp-Vektors und dem Release-Timestamp-Vektor des Fremdprozessors ermittelt. Danach sendet er q seinen Timestamp-Vektor , damit er von q alle Modifikationen, die er noch nicht kennt, die aber q bekannt sind, erhält.

Implementation

Der Schreibzugriff auf eine invalidierte Seite verursacht eine Zugriffsverletzung. Der Prozessor p ermittelt den letzten Releaser dieses Teildatenbereichs. Existiert kein solcher Prozess (bei Programmstart) bzw. ist der Prozessor selbst der letzte Releaser, bekommt der fragende Prozess die Schreibrechte und kann mit seiner Programmabarbeitung fortfahren.

Ist ein fremder Prozessor q als der letzte Releaser erkannt, wird der gesamte eigene Timestamp-Vektor zu diesem Prozeß q geschickt. Dies geschieht *piggybacked* auf dem Rücken der Nachricht, die das Schreibtoken[29] auf dieser Seite verlangt. Mit Übersendung des Tokens schickt q alle Diffs, die Prozess p laut seinem Timestamp-Vektor unbekannt sind (siehe Abbildung 10). Eine Zeitmarke ist unbekannt, wenn sich im eigenen Timestamp-Vektor keine Zeitmarke mit gleichem Hostnamen und gleichem Port befindet bzw. bei Übereinstimmung die Intervallnummer niedriger als die der fremden Zeitmarke ist. Der Prozessor *p* erhält das Token und fügt alle ihm unbekannten Diffs jüngerer Intervalle ein. Damit ist seine Speicherseite bezüglich dieses Teildatenbereichs aktualisiert und konsistent. Es wird ersichtlich, dass die Diffanforderung, die in Abbildung 10 als Pfeil dargestellt ist, aus zwei Nachrichten besteht.

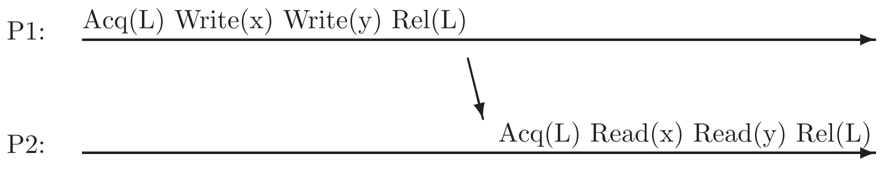

Abbildung 10: Träge Invalidierung

5 Barrieren

Eine Barriere kann mit Hilfe der Acquire-Release-Intervalle modelliert werden.

Barrieren sind Synchronisierungs-Mechanismen, der einen Prozess daran hindern, Phase *n+1* eines Programmes zu beginnen, bis alle Prozesse Phase *n* beendet haben. Wenn ein Prozess an eine Barriere kommt, dann muß er warten, bis alle anderen Prozesse auch dort angekommen sind. Wenn der letzte Prozess die Barriere erreicht, werden alle gemeinsam genutzten Variablen synchronisiert. Erst danach werden alle Prozesse fortgesetzt. Die Ankunft an der Grenze bedeutet *Release*, ihr Verlassen bedeutet *Acquire*.

Das Modell einer Barriere

In Abbildung 11 wird ein einziger zentralisierter *Barrieren-Manager* benutzt, der alle Ankunfts- und Abfahrtsmessages sammelt.

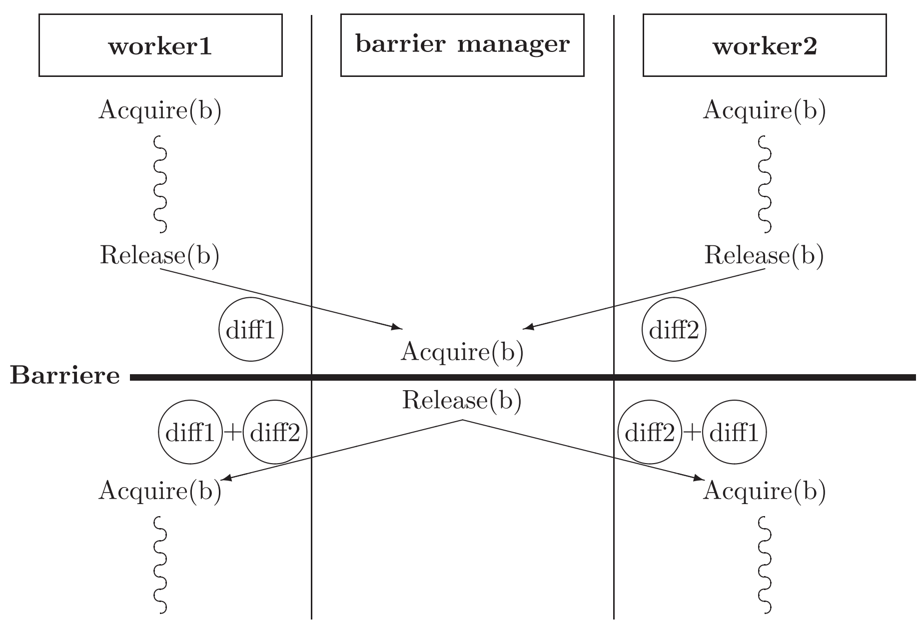

Abbildung 11: Acquire-Release-Modellierung einer Barriere

Alle Prozesse führten initial ein Acquire auf die Barriere b durch. Das implizite Lock auf die Barriere ist als Multi-Writer-Lock definiert. Bei Ankunft an die Barriere wird das Release(b) aufgerufen. Nach dem letzten Prozess setzt der Barrieren-Manager ein Acquire(b) ab. Dadurch erfragt er von allen Worker-Prozessen die Teildatenbereiche, die diese im letzten Intervall verändert haben und generiert einen Satz aktueller Seiten. Nach seinem Release(b) erfragen alle Worker-Prozesse (durch ein eigenes Acquire(b)) diese Modifikationen. Nach Verlassen der Barriere besitzen alle Prozesse einen konsistenten Speicher, wie er mit dem Protokoll der sequentiellen Konsistenz aussehen würde. Die Timestamp-Vektoren werden ebenso aktualisiert.

Die Implementation einer Barriere

Im Gegensatz zu diesem theoretischen Modell wird im bestehenden DSM-System[27, 28] eine verteilte baumstrukturierte Barriere verwendet[6, 7] (siehe Abbildung 12).

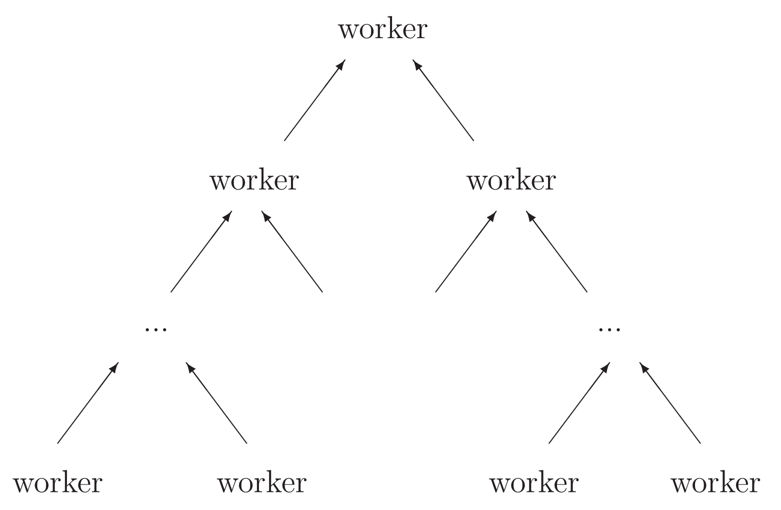

Abbildung 12: Eine baumstrukturierte Barriere

Sie vermeidet den Flaschenhals der Kommunikation mit einem zentralisierten Barrierenmanager. In einer Baumbarriere wird die Arbeit des Managers auf alle Knoten verteilt. Alle Worker-Threads organisieren sich in einem Baum wie in Abbildung 12 gezeigt. Der erste an der Barriere ankommende Knoten bildet die Wurzel des Baumes. Alle weiteren Knoten, die die Barriere erreichen, sortieren sich als logische Kindknoten innerhalb des binären Baumes an.

Es werden zwei Arten von Nachrichten unterschieden. Betreten die Knoten die Barriere, senden sie ein Signal an ihren Vaterknoten, also im Baum empor. Erhalten sie ein Antwortsignal von ihrem Vaterknoten, verlassen sie die Barriere und arbeiten weiter.

Der Ablauf in einer Baumbarriere

Bevor ein Knoten seine Ankunftsnachricht an seinen Elternknoten sendet, wartet er erst auf das Ankunftssignal seiner beiden Kindknoten(Abbildung 13). Da dieser auch erst auf das Eintreffen seiner Kinder wartet, startet erst der zuletzt ankommende Knoten die Nachrichtenkette im Baum. Alle im Baum befindlichen Knoten senden das Signal an ihren jeweiligen Vaterknoten. Wenn die Wurzel des Baumes die Signale seiner beiden Kindknoten erhält und selbst an der Barierre ist, sind alle Prozesse an der Barriere beteiligt(Abbildung 14).

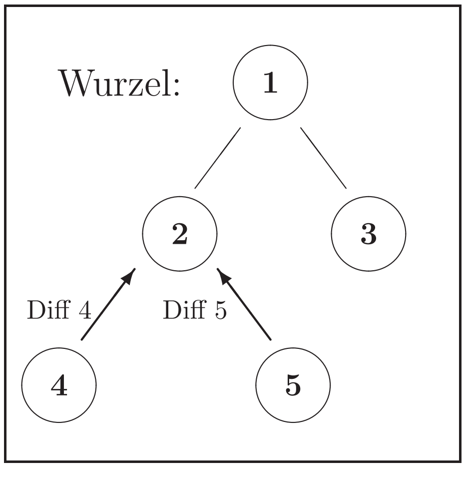

Abbildung 13: Ankunft an der Barriere

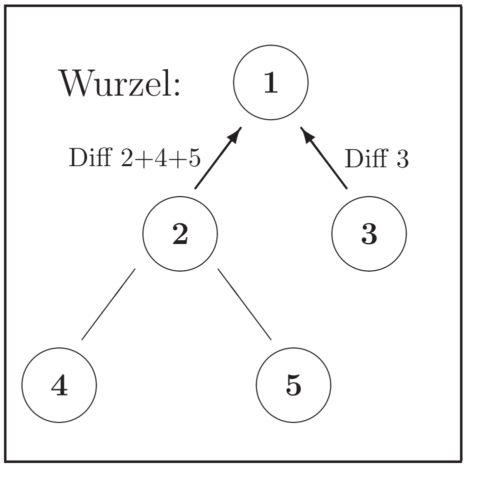

Abbildung 14: Weiterleitung der Diffs

Sie schickt daraufhin das Signal für das Weiterarbeiten an ihre beiden Kindknoten und verläßt die Barriere. Diese schicken die Abfahrts-Nachricht weiter zu ihren Kindern, bis alle Prozesse die Barriere verlassen haben (siehe Abbildung 15). In Abhängigkeit von der verwendeten Hardware kann dazu auch ein eventuell effizienterer *Broadcast* verwendet werden.

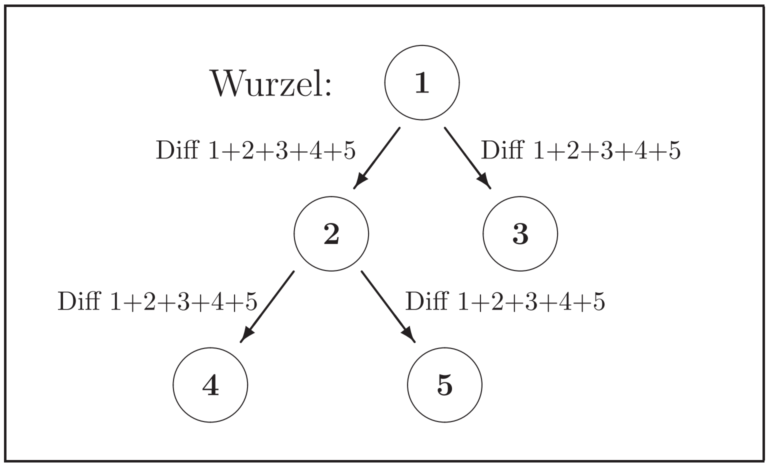

Abbildung 15: Abfahrts-Nachricht aus der Barriere

Die Effizienz einer Baumbarriere

Die Tiefe des Baumes beträgt $\lceil log_2 n \rceil$. Er ist deshalb für große n sehr effizient.

Die Größe einer Nachricht beeinflußt nicht die Performance des verteilten DSM-Systems. Die zentrale Idee bei der Implementation der trägen Release-Konsistenzmodelles ist es, die nötigen Aktualisierungsnachrichten an die Ankunfts- bzw. Abfahrts-Nachrichten anzuhängen. Damit könnte der gemeinsame verteilte Speicher ohne Performanceverlust konsistent gemacht werden. Unter der Voraussetzung, dass alle Knoten an einer Baumbarriere teilnehmen, wären nach der Barriere alle Speicherseiten aktualisiert.

Tatsächlich sind die Benutzung aller Nachrichten in beide Richtungen nötig. Jeder ankommende Knoten packt die von ihm erzeugten Veränderungen aller seiner Intervalle in einen Diff-String. Damit ist die Eigenschaft der Prozessor-Konsistenz zugesichert. Im Anhang der Ankunftsnachricht schickt er seine Diffs zu seinem Vaterknoten im Baum. Dieser ergänzt den Anhang um die von ihm generierten Diffs, ohne den Inhalt der empfangenen Diff-Zeichenkette zu dekodieren.

Nach Erhalt der Ankunftssignale besitzt die Wurzel des gesamten Baumes die Diffs aller an der Barriere beteiligten Knoten. Sie aktualisiert ihre eigenen Speicherseiten und hängt den großen Diff-String an die Abfahrts-Nachricht an. Jeder Knoten im binären Baum führt schrittweise in der Reihenfolge der Intervalle jedes Prozesses die Speicherveränderungen aus, damit die verlangte Prozessor-Konsistenz eingehalten wird. Nachdem alle Knoten die Barriere verlassen haben, ist der Speicher konsistent.

Um Zählerüberläufe zu vermeiden, werden nach der Barriere alle Intervalle neu initialisiert. Ebenso können die erzeugten Twins und Diffs aller Prozesse dealloziiert werden. Dies ist nur unter der Bedingungung möglich, dass alle Prozesse an der Barierre teilnehmen.

6 Veränderungen am DSM-System

Für die Implementation des MRMW-Protokolles wurden die Module *Timestamp* und *Release* erzeugt und die Module *Diff*, *Twin*, *Pagefunc*, *Pagetab* und *Barrier* verändert. An dieser Stelle werden die erzeugten bzw. veränderten Module näher vorgestellt, zu denen umfangreichere Programmierarbeiten nötig waren.

Im Verlauf der Software-Entwicklung stellte sich das Problem der internen Kodierung der Zeitmarken und Diffstrings. Eine Darstellung in Strukturform (*struct*) würde einen aufwandslosen Zugriff auf die Memberdaten garantieren. Damit wäre z.B. ein schneller Intervallvergleich zwischen zwei Zeitmarken möglich. Bestimmte Einträge in Difflisten könnten effizienter bearbeitet werden. Demgegenüber steht die Bedingung, dass Diffstrings und Zeitmarken sehr oft zu anderen Prozessen verschickt werden müssen. Dazu ist bei einer Darstellung in Strukturform eine Kodierungsfunktion nötig, die die gesamte Struktur zum Senden in Stringform ver- bzw. bei Erhalt einer empfangenen Nachricht entpackt.

Durch die eindeutige Ineffizienz dieser Funktion wurde die Darstellung in Strukturform verworfen und alle Zeitmarken und Diffstrings in Stringform kodiert. Der Versand- und Empfangsaufwand reduzierte sich gegen Null. Der relativ aufwändige Zugriff wurde mit den beiden effizient programmierten Konvertierungsfunktionen *dsm_Char2LongInt()* und *dsm_LongInt2char()* in erträglichen Maßen gehalten. Im direkten Vergleich zwischen beiden Darstellungsformen zeigte sich diese Kodierung als gleichwertig effizient und programmiertechnisch einfacher.

6.1 Das Modul Timestamp

Lamport stellte schon 1979 den klassischen Algorithmus vor, der zeigte, dass die Uhrsynchronisation in verteilten Systemen möglich ist. Er legte dar, dass die Uhrsynchronisation nicht absolut sein muß, sondern die Beachtung einer Ereignisreihenfolge als Synchronisationsalgorithmus ausreichend sein kann. Wenn zwei Prozesse miteinander interagieren, dann ist es nicht notwendig, dass ihre Uhren synchronisiert werden, weil das Fehlen der Synchronisation nicht auffällt und somit keine Probleme verursacht. Die Uhren werden dann als *logische* Uhren bezeichnet.

Um logische Uhren zu beschreiben, definierte Lamport[20, 21, 22, 23] eine Relation *passiert_vor*. Lamports Algorithmus folgt direkt aus der *passiert_vor*-Relation. Er gibt eine Methode an, allen Ereignissen in einem verteilten System eine Zeit zuzuordnen, die bestimmten Bedingungen genügt. Mehr zu diesem Thema findet sich in Kapitel 3.1 auf Seite 11.

Das Modul Timestamp implementiert die logischen Uhren für das Modell der trägen Release-Konsistenz im bestehenden DSM-System.

Eine einzelne Zeitmarke

Ein einzelner Timestamp beschreibt einen Prozess auf einem über den Hostnamen identifizierbaren Rechner eindeutig. Der Prozess ist dort unter einem definierten Port zu erreichen. Seine logische Uhrzeit wird durch eine Intervallnummer bestimmt.

Das Memberdatum Intervall beschreibt die globale Zeit im Sinne der *passiert_vor*-Relation. Beim Senden einer Nachricht wird das eigene Intervall inkrementiert.

In den folgenden Abschnitten werden die Begriffe Zeitmarke und Timestamp synonym verwendet werden.

Abbildung 16 zeigt den internen Aufbau einer Zeitmarke .

TS-Länge	Host	Port	Intervall

Abbildung 16: Struktur eines Timestamps

In Abbildung 17 wird beispielhaft der Timestamp des Prozesses aufgezeigt, der auf Rechner *localhost* unter Port *1234* erreichbar ist und sich im momentanen Intervall *5* befindet.

21	'localhost'	1234	5

Abbildung 17: Beispiel einer Zeitmarke

Der Speicherplatzbedarf des Hostnamens ist seiner Größe angepasst. Für die Dezimalzahlen wird ein Platz von vier Bytes reserviert. Sie werden mit Hilfe der beiden effizienten Konvertierungsfunktionen *dsm_Char2LongInt()* und *dsm_LongInt2char()* erzeugt bzw. ausgelesen. Zu diesem Thema wird auf Kapitel 6 auf Seite 35 verwiesen.

Ein Timestamp-Vektor

Ein Timestamp-Vektor dient der Implementation einer logischen Uhr in einem verteilten System. Er existiert einmal für jeden beteiligten Prozess. Er verwaltet die eigene Zeitmarke und alle empfangenen Timestamps der anderen Prozesse. Ein fremder Timestamp wird dann in den eigenen Timestamp-Vektor aufgenommen, wenn die eigenen Kopien der Speicherseiten mit den Speicherveränderungen, die der fremde Prozess geschickt hat, aktualisiert wurden. Im globalen Timestamp-Vektor ist also die Information über die erfolgte Prozesssynchronisation direkt erkennbar.

Abbildung 18 zeigt den internen Aufbau eines Timestamp-Vektors .

Länge	#TS	TS-Länge	Host	Port	Intervall	...

Abbildung 18: Struktur eines Timestamp-Vektors

Der erste Eintrag definiert die Länge des gesamten Timestamp-Vektors in Zeichen. Die zweite Dezimalzahl beschreibt die Anzahl von angehängten kodierten Zeitmarken . Die erste Zeitmarke definiert den eigenen globalen

Timestamp . Alle folgenden Einträge (symbolisiert mit ...) lassen die vergangene Kommunikation mit fremden Prozessen erkennen.

Im Beispiel der Abbildung 19 wird der Timestamp-Vektor des Prozesses beschrieben, der auf Rechner *localhost* unter Port *1234* erreichbar ist, sich im momentanen Intervall *5* befindet und insgesamt *3* Timestamps besitzt.

76	3	21	'localhost'	1234	5	17	'worf'	6666	5	...

Abbildung 19: Beispiel eines Timestamp-Vektors

Ausser der eigenen Zeitmarke sind noch noch zwei Timestamps fremder Prozesse kodiert. Die jeweilige Intervallnummer des Fremdprozesses zeigt, dass dem Besitzer dieses Timestamp-Vektors alle Speicheränderungen von *worf:6666* bis einschließlich Intervall *5* bekannt sind, d.h. dessen Speicherschreibvorgänge bis zu diesem Intervall in den eigenen Kopien aktualisiert wurden.

Zwei Zeitmarken des gleichen Rechners (Hostnamens) und unterschiedlichen Ports sind voneinander verschieden. Dies trifft zu, falls sich mehrere Threads auf einem Computer befinden und unter jeweils anderen Portadressen erreichbar sind. Im logischen Sinne sind sie als verschiedene Knoten eines verteilten Systems zu interpretieren.

Die Funktion 'void timestamp_vector_init()'

Diese Funktion alloziiert einen Timestamp-Vektor , der mit initialen Werten belegt ist (siehe Abbildung 20). Die erste Zeitmarke beinhaltet die Kodierung des Besitzerprozesses: den eigenen Hostnamen und Port, sowie das Intervall Eins. Alle anderen Prozesse sind zu diesem Zeitpunkt noch nicht bekannt.

21	'localhost'	1234	1

Abbildung 20: Initialer Zustand eines Timestamp-Vektors

Zum Aufbau eines Timestamp-Vektors sei auf das Kapitel 6.1 auf Seite 38 verwiesen. Die Kodierung der eigenen Eigenschaften ist relativ aufwändig. Demgegenüber wird diese Funktion nur einmal von jedem beteiligten Prozess aufgerufen. Die Funktion *void timestamp_vector_init()* muß nicht zwangsläufig aufgerufen werden, die meisten der Funktionen dieses Moduls testen vor ihrer Ausführung die Existenz des globalen Timestamp-Vektors und erzeugen ihn bei Bedarf.

Die Funktion 'void timestamp_vector_destroy()'

Diese Prozedur zerstört den eigenen globalen Timestamp-Vektor . Diese Routine ist für eine zufriedenstellende Speicherfreigabe am Ende des Programmes explizit aufzurufen.

Die Funktion 'char* timestamp_vector_get()'

Die Funktionen *char* timestamp_vector_get()* und *char* timestamp_get()* geben den eigenen Timestamp-Vektor bzw. Timestamp an den Aufrufer zurück. Sie testen, ob der Timestamp-Vektor , der Besitzer dieser Zeitmarke ist, alloziiert wurde. Im Falle der Nichtexistenz wird er durch den Aufruf von *timestamp_vector_init()* neu erzeugt. Die zurückgegebene Ergebnis-Zeitmarke ist ein direkter Pointer auf den eigenen globalen Timestamp-Vektor . Dies ermöglicht eine effiziente Ausführung der Funktionen. Der Ergebnispointer darf also weder vorher alloziiert noch nach Aufruf der Funktionen dealloziiert werden. Die Funktion *char* timestamp_vector_length()* gibt bei Bedarf die Stringlänge des Timestamp-Vektors zurück.

Die Funktion 'void timestamp_new_interval()'

Die Routine inkrementiert die Intervallnummer des eigenen Prozesses um Eins. Dies geschieht im Falle einer Acquire- bzw. Releaseoperation. Auch hier erfolgt ein Existenztest und eine eventuelle Initialisierung des Timestamp-Vektors .

Die Funktion 'int timestamp_know(char* stamp, char *tsv)'

Ihr wird eine Zeitmarke und ein Timestamp-Vektor übergeben. Es werden alle Timestamps des übergebenen Vektors mit der Zeitmarke verglichen. Eine Zeitmarke ist bekannt, wenn sich im Vektor ein Timestamp mit gleichem

Hostnamen, gleicher Portnummer und geringerem Intervall befindet. In diesem Fall ist der Rückgabewert Eins, sonst Null.

Die Funktion 'int timestamp_add (timestamp_t *stamp)'

Die Funktion fügt eine Zeitmarke in den globalen Timestamp-Vektor ein. Es sind zwei mögliche Fälle zu unterscheiden. Wenn der Timestamp *bekannt* ist (s.o.), so wird die Funktion mit dem Rückgabewert Null beendet. Nur unbekannte Zeitmarken werden in den Timestamp-Vektor aufgenommen. Der Returnwert von Eins zeigt diese Eigenschaft auf (in diesem Falle benötigt der Prozess die noch unbekannten Diffs).

DEBUG-Funktionalität

Für DEBUG-Sitzungen gedacht sind folgende Funktionen, die TRACE-Ausschriften ihrer Objekte aufzeigen: *int dump_timestamp(char* stamp)* und *void dump_timestamp_vector(char* titel, char* _tsv)*.

6.2 Das Modul Diff

In dem Modell der Release-Konsistenz werden die Synchronisationsoperationen Acquire und Release unterschieden. Zusätzlich wird die Lebensdauer eines Prozesses in Intervalle unterteilt, die bei der Ausführung der Release-Operation inkrementiert werden.

Ein Schreibzugriff auf eine Datenstruktur einer Seite, die für die träge Release-Konsistenz markiert ist, löst die Ausführung der Acquire-Funktion auf der Speicherseite aus. Dabei wird diese Seite kopiert und als Twin abgespeichert (siehe Kapitel 6.3, Seite 49). Danach kann auf die Datenstrukturen dieser Seite beliebig zugegriffen werden. Innerhalb der Release-Operation wird ein Diff erstellt, der die Unterschiede zwischen Seite und Twin in komprimierter Form enthält. Alle Diffs der Seiten werden in einer globalen Liste gespeichert, da nur sequentielle Zugriffe darauf nötig sind. Zu der Struktur, in dem diese Diffs vorliegen, wird auf Kapitel 6 auf Seite 35 verwiesen.

Der Term *Diff* wird in diesem Dokument in mehreren Zusammenhängen verwendet. Ein Diff definiert den Unterschied zwischen einer Speicherseite und seinem erstellten Twin. Aber auch die entstandene Zeichenkette, die sämtliche Differenzen einer Speicherseite in kodierter Form enthält, wird als Diff bezeichnet. Semantisch sind beide Interpretationen äquivalent.

Als *Diff-String* wird die Zeichenkette bezeichnet, die einen oder mehrere Diffs inklusive einer Zeitmarke enthält. Sie kann zu anderen Prozessen verschickt werden.

Ein einzelner Seitendiff

Der Diff einer einzelnen Seite wird bei Anforderung eines Acquires von einem anderen Prozess erstellt. Er wird in Abbildung 21 dargestellt.

PageIndex	#Diffs	Off/Len	Diff	...	Off/Len	Diff

Abbildung 21: Struktur eines Seiten-Diffs

Das Attribut *Pageindex* definiert die Nummer der ursprünglichen Seite. Dann werden die Anzahl der in diesem String kodierten Diffs festgelegt (*#Diffs*). Alle folgenden Einträge wiederholen sich um diese Menge. Die nächsten beiden Attribute (*Off/Len, Offset/Länge*) beschreiben die Anfangsposition des Diffs auf der Seite und seine Länge. Um Übertragungsmenge zu sparen, sind sie in ein Zahlenfeld kodiert. Dies beschränkt ihre Größe auf jeweils 32767. Indirekt wird damit die Seite auf diese Größe beschränkt, da das Attribut Offset keinen größeren Wert annehmen kann. Der eigentliche *Diff* beinhaltet die Veränderungen der Speicherseite an der Position *Off* mit der Länge *len*. Im Beispiel der Abbildung 22 wird der Diff der Seite *4* beschrieben, der *3* Diffs kodiert.

4	3	24/4	'abcd'	48/4	'wxyz'	96/8	'12345678'

Abbildung 22: Beispiel eines Seiten-Diffs

Der erste Diff von dieser Seite wurde an der Position *21* mit der Länge *4* erzeugt. Die Werte, die sich vorher an dieser Stelle befanden, wurden mit den Werten *'abcd'* überschrieben. In diesem Beispiel werden nur Zeichenketten zur besseren Darstellung eines Diffs benutzt. Die anderen beiden Diffs beschreiben die Offsets *122* und *200*. Die Offsets und Längen aller Diffs sind Vielfache von Vier. Dies ist durch die Kodierungsfunktionen bedingt, die möglichst effizient arbeitet.

Ein Diff-String

Diese Zeichenkette beinhaltet alle Seitendiffs, die ein Prozess in einem definierten Intervall erzeugt hat. Er besteht aus einer Kette von Einzeldiffs.

Die Abbildung 23 zeigt den Aufbau eines solchen Diff-Strings.

Stringlänge	Zeitmarke	#Diffs	Diff1	...	Diffn

Abbildung 23: Struktur eines Diffs

Der erste Eintrag *Stringlänge* definiert die Gesamtlänge der Zeichenkette. Die Zuordnung des Diff-Strings zu einem Prozess und einer Intervallnummer erfolgt durch die *Zeitmarke* . Der Timestamp besitzt die Attribute HOST, PORT und INTERVALL. Der Parameter *#Diffs* legt die Anzahl der folgenden Seitendiffs fest, die nach obigen Schema aufgebaut sind.

Die Abbildung 24 beschreibt den Diff der Länge *28*, der von dem Prozess erstellt wurde, der auf dem Computer *'worf'* unter der Portadresse *1234* erreichbar ist.

28	'worf'	1234	5	1	4	1	128/4	'wxyz'

Abbildung 24: Beispiel eines Diffs

Er befand sich zum Zeitpunkt der Erzeugung dieser Zeichenkette im Intervall *5*. Es ist *1* einzelner Diff kodiert (s.o.). Von Seite *4* wurde *1* Zeichenkette von Position *128* der Länge *4* mit dem Inhalt *'wxyz'* erzeugt.

Die Funktion 'void dsm_InitDiffs()'

Diese Routine initialisiert alle Listenstrukturen. Es existieren zwei verschiedene globale Listen für jeden Thread: *current_diff_list* und *old_diff_list*.

In der Liste *current_diff_list* werden alle Differenzen abgespeichert, die der Thread selbst erzeugt hat. Der kodierte Timestamp in ihnen entspricht der thread-eigenen Zeitmarke mit geringerer Intervallnummer. Diese Diffs werden bei einem Mutex und innerhalb einer Barriere in die zu versendende Nachricht kodiert.

In die Liste *old_diff_list* werden alle Diffs abgespeichert, die der Thread von anderen Prozessen innerhalb eingegangener Nachricht empfangen und dekodiert hat. Diese Diffs werden bei einem Mutex in die zu versendende Nachricht kodiert. Sie dürfen aus Effizienzgründen nicht bei einer Barriere verwendet werden. Mit der Teilnahme von n Prozessen an einer Barriere teilen alle Threads ihre eigenen Diffs den anderen teilnehmenden Prozessen ihre eigenen Differenzen mit. Nach Austritt aus der Barriere besitzten n Prozesse alle Informationen der anderen Barrierenmitglieder. Es ist nicht nötig, die Zeichenketten der Liste *old_diff_list* zu übertragen.

Die Funktion 'int dsm_CreateDiff(long index, char **diff, int *len)'

Die Routine bekommt eine Seitennummer *index* als Parameter. Sie alloziiert eine Ergebniszeichenkette *diff*, vergleicht die Seite mit ihrem Zwilling und packt die Unterschiede nach obigen Format in die Zeichenkette. Nach Ausführung dieser Operationen definiert der Parameter *difflen* die Länge von *diff*. Diese Funktion wird im DSM-System von dem Protokoll der *Bereichs-Konsistenz* direkt benutzt.

Die Funktion 'char *dsm_ApplyDiff(long index, char *diff)'

Diese Operation besitzt die inverse Funktionalität der obigen Routine. Sie bekommt eine Seitennummer *index* und eine Zeichenkette *diff* als Parameter. Sie wendet den Diff auf die entsprechende Seite an, d.h. sie aktualisiert ihre Seitenkopie mit den Veränderungen des fremden Prozesses. Diese Funktion wird im DSM-System von dem Protokoll der *Bereichs-Konsistenz* direkt benutzt.

Die Funktion 'int dsm_CreateAllDiffs()'

Sie durchsucht alle von dem eigenen Prozess erzeugten Twins (siehe Kapitel 6.3, Seite 49). Für jede veränderte Seite (d.h. es existiert ein Unterschied zwischen einer Seite und ihrem Twin) wird ein Diff einer einzelnen Seite nach obigem Format erzeugt. Anschließend werden alle Einzeldiffs zu einem Diff-String zusammengepackt und mit einem Header versehen (*Timestamp*, *#Pages*, s.o.). Der resultierende Diff-String wird in der eigenen globalen Liste abgespeichert. Diese Funktion wird i.a. am Ende eines Intervalles aufgerufen.

Die Funktion 'int dsm_encode_diffs(char **diff, char* tsv_string)'

Die Routine erhält als Parameter eine Zeichenkette *diff* und einen Timestamp-Vektor *tsv_string*. Sie alloziiert und belegt einen Diff-String, der alle dem Timestamp-Vektor *tsv_string* unbekannten Diffs enthält. Zu der Definition eines *unbekannten Diffs* sei auf das Kapitel 6.1 auf der Seite 37 verwiesen. Ist beim Aufruf dieser Funktion der Parameter *tsv_string* gleich NULL, werden alle besitzenden Diffs in den Ergebnisstring kodiert. Sollte die übergebene Zeichenkette *diff* nicht leer sein, so wird angenommen, dass er bereits kodierte Diffs enthält. Er wird in diesem Falle realloziiert und nur die eigenen (von diesem Prozess erzeugten) Diffs hinten in die Zeichenkette kodiert. Diese beiden Eigenschaften sind bei einer Barriere nötig.

Die Funktion 'void dsm_decode_and_add_diffs(char *diffstream)'

Die Parameterzeichenkette *diffstream* kodiert eine Diff-Struktur. Die Funktion ermittelt den jeweiligen Timestamp und testet, ob ihr dieser Diff *unbekannt* (s.o.) ist. Trifft diese Aussage zu, werden alle zugehörigen Einzeldiffs dekodiert, die zugehörigen Speicherseiten aktualisiert und der Diff-String in der globalen Liste abgespeichert.

Die Funktion *void dsm_destroy_DiffList()* Diese Operation zerstört alle bestehenden Diffs und die Difflisten *current_diff_list*, *old_diff_list* und *barrier_diff_list*. Diese Routine ist für eine zufriedenstellende Speicherfreigabe am Ende des Programmes explizit aufzurufen.

DEBUG-Funktionalität

Für DEBUG-Sitzungen gedacht sind folgende Funktionen, die TRACE-Ausschriften ihrer besitzenden Objekte in Textform aufzeigt: *void dump_DiffList(char* titel)* und *void dump_diffstring(char* titel, char *string)*.

6.3 Das Modul Twin

Die Funktion 'int dsm_CreateTwin(long index)'

Der Parameter *index* definiert den Index einer Speicherseite, über den mittels *dsm_GetPage(index)* ein direkter Zugriff möglich ist. Die Twins werden mit Hilfe von Listen verwaltet, da nur ein sequentieller Zugriff benötigt wird. Die entsprechende Seite wird kopiert und die Kopie gesichert. Die Ausführung dieser Funktion setzt voraus, dass die entsprechende Seite bereits gelockt ist.

Die Funktion 'void dsm_DeleteTwin(long index)'

Die Routine zerstört den über den Parameter *index* definierten Zwilling.

DEBUG-Funktionalität

Für DEBUG-Sitzungen gedacht sind folgende Funktionen, die TRACE-Ausschriften ihrer besitzenden Strukturen aufzeigt: *void dsm_DumpPage(char* titel, long index)* und *void dsm_DumpTwin(char* titel, long index)*.

6.4 Das Modul Release

Das Modul Release stellt die Schnittstelle zwischen DSM-System und dem trägen Release-Konsistenzmodell. Sie bietet alle Funktionen, die von der DSM-Software benötigt werden, um das MRMW-Protokoll nutzen zu können.

Alle Operationen dieses Moduls sind von einem lokalen Mutex umhüllt. Dieser verhindert die mißbräuchliche zeitgleiche Nutzung globaler Listenstrukturen bzw. Funktionen zweier Threads auf einem gemeinsamen Knoten.

Die Funktion 'void dsm_ReleaseInit(long FirstReleasePage)'

Diese Operation initialisiert die eigenen Listenstrukturen und den lokalen Mutex. Sie ist zwingend aufzurufen. Sie führt die initialen Funktionsaufrufe der Module *Timestamp* und *Diff* auf.

Die Funktion 'void dsm_SaveTwin(long TwinIndex)'

Die Routine speichert den Zwilling einer Seite in der eigenen globalen Liste ab. Der Twin wird solange in der Liste aufgehoben, bis das Programm beendet bzw. ein Diff von diesem Twin und der zugehörigen Speicherseite durchgeführt wird. Dann wird der Twin ersatzlos zerstört.

Die Funktion 'void dsm_CloseInterval()'

Sie wird bei Ende eines Intervalles aufgerufen. Der Intervallzähler wird inkrementiert. Zusätzlich werden alle Diffs des Intervalles erzeugt und abgespeichert.

Ein Intervall wird beendet bei:

- Diffanforderung eines anderen Prozesses
- Diffempfang eines anderen Prozesses
- Eintritt in eine Barriere

Die Funktion 'int dsm_SendInterval(char **diff, char* tsv_string)'

Die Prozedur ruft direkt die Funktion *int dsm_encode_diffs(char **diff, char* tsv_string)* des Moduls *Diff* auf. Dazu sei auf das Kapitel 6.2 auf Seite 43 verwiesen. Es werden alle dem Parameter *tsv_string* unbekannten Diffs in einen Diff-String *diff* kodiert.

Die Funktion 'void dsm_ReceiveInterval(char *diff)'

Die Parameterzeichenkette *diff* definiert einen solchen Diff-String. Das eigene Intervall wird mittels der Funktion *void dsm_CloseInterval()* beendet (Intervallerhöhung und Differstellung). Danach werden die im Parameter kodierten Diffs entpackt und die entsprechenden Speicherseiten aktualisiert. Dies entspricht dem direkten Aufruf der Funktion *void dsm_decode_and_add_diffs(char *diffstream)* im Modul *Diff*.

6.5 Die Aufrufhierarchie

Die zentrale Rolle des Modules *release* als Schnittstelle zur DSM-Software ist in Abbildung 25 deutlich erkennbar. Die Richtung der Pfeile entspricht den *#include*-Anweisungen im Quelltext.

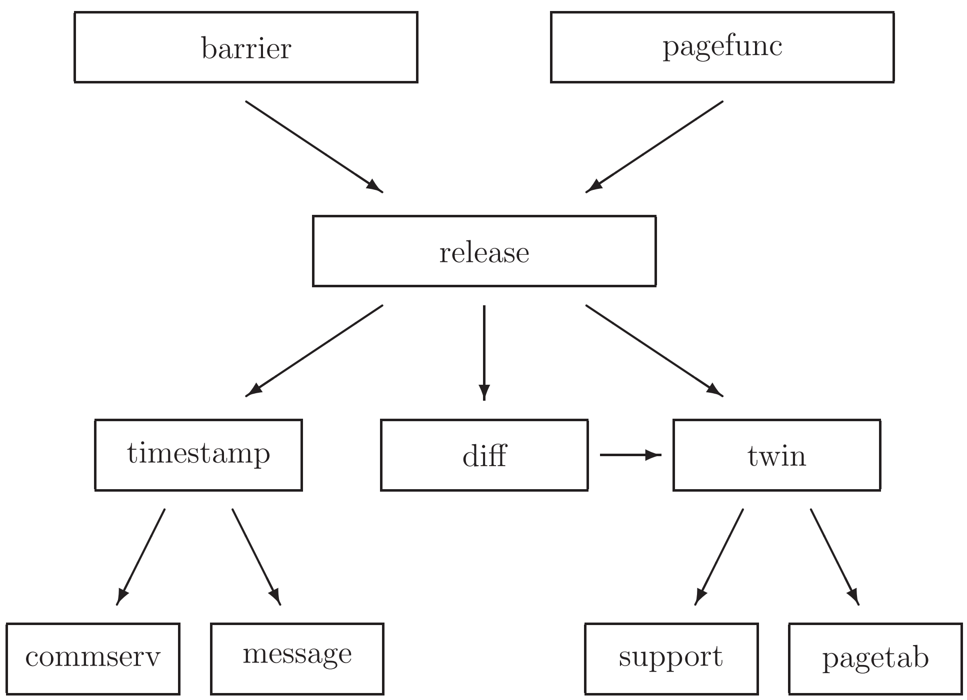

Abbildung 25: Die Aufrufhierarchie der Module

Das Modul *commserv* wird für die Funktionen *char* dsm_selfhost()* und *int dsm_selfport()* benötigt. Sie liefern den Namen des Computers, auf dem sich der ausführende Prozess befindet und die Portadresse, unter der der Prozess erreichbar ist. Beide Angaben dienen der Identifizierung eines Prozesses. Mehr Angaben dazu sind unter Kapitel 6.1 auf Seite 37.

Im Modul *message* sind die beiden Funktionen *dsm_Char2LongInt()* und *dsm_LongInt2char()* definiert. Sie werden für die Konvertierungen von Dezimalzahlen in Zeichenketten und umgekehrt benutzt. Zu diesem Thema sind im Kapitel 6 auf Seite 35 mehr Informationen.

Das Modul *support* unterstützt die umfangreiche Funktionalität zum Abspeichern beliebiger Datenstrukturen in sortierten Listen. Mit Hilfe dieses

Moduls werden alle *Diffs* und *Twins* abgespeichert.

Die Speicherseiten des DSM-Systems werden im Modul *pagetab* verwaltet. Innerhalb der Datenstruktur befindet sich ein Pointer, der für die Aufnahme des Zwillings einer Seite reserviert ist. Der direkte Zugriff auf die Seiten (Status, Access, Lock) wird über verschiedenste Funktionen unterstützt. Ob und welche Speicherseite über das träge Konsistenzmodell verwaltet wird, definieren *int dsm_isRelease(void)* und *int dsm_isDSMRelease(char *addr)*.

7 Performancetests

Um das neue MRMW-Protokoll mit dem trägen Release-Konsistenzmodell einschätzen zu können, werden einige Performancetests durchgeführt. Als direktem Vergleichsprotokoll steht im aktuellen DSM-System das MRSW-Protokoll zur Verfügung. Grundsätzlich wurden Funktionalitätstests durchgeführt, d.h. alle Berechnungen der verschiedenen Protokolle mussten zu identischen Resultaten führen. Zuerst wurde ein spezielles Testprogramm entworfen, welches besonders die Eigenheiten des trägen Release-Konsistenzmodelles nutzt. Es wird vermutet, daß sich mit ihm deutliche Geschwindigkeitsunterschiede zeigen lassen. Desweiteren wurde von den SPLASH-2-Benchmarks das Programme FFT (Fast Fourier Transformation) benutzt, um einen plattformübergreifenden Vergleich zu ermöglichen, da FFT ein repräsentatives Programm für parallele Systeme darstellt.

Die Performancetests wurden auf einem Cluster folgender PC's durchgeführt:

- Hauptrechner : 2 x Pentium III 500 MHz, 1024 MB
- snow0 - snow15: Athlon 550 Mhz, 256 MB
- snow16, snow20: 2 x Celeron 466 MHz, 128 MB
- snow17 - snow23: Celeron 466 MHz, 128 MB

Diese PC's sind über Fast Ethernet und Myrinet (LANai Version 4.03) miteinander vernetzt. Es wurde die Kommunikationsarchitektur TCP verwendet.

Die Performancetests wurden stets am Wochenende und bevorzugt nachts durchgeführt. Die Auslastung der zu dieser Zeit unbenutzten Rechner lag nahe null Prozent. Alle Versuche liefen mit ausreichenden Wiederholungen, um eventuelle Performanceschwankungen zu eliminieren. Aufgrund ihrer identischen Hardwareausstattung wurden für die endgültigen Tests ausschließlich die Rechner snow0 bis snow15 verwendet.

7.1 Das Testprogramm *test_release*

Das Ziel diese Testreihe war die Erschaffung eines Programmes, welches die MRMW-Eigenschaften des trägen Konsistenzmodelles am besten nutzen kann. Da die Matritzenmultiplikation über lineare Gleichungssysteme ableitbar ist, wurde eine Matrixmultiplikation zweier Matritzen ausgewählt. Zwei Matritzen werden miteinander multipliziert, indem die jeweiligen Zeilen- und Spaltenvektoren skalar miteinander multipliziert werden:

$$c_{ij} = b_{i1} * a_{1j} + b_{i2} * a_{2j} + ... + b_{im} * a_{mj} = \sum_{v=1}^{m} b_{iv} * a_{vj} \tag{1}$$

Matrix B ist von der Dimension (k,m) und Matrix A vom Grad (m,n). Die Produktmatrix C besitzt die Dimension (k,n) und ihre Elemente sind gegeben durch

$$c_{ij} = \sum_{v=1}^{m} b_{iv} * a_{vj} \tag{2}$$

Ersichtlich ist, daß eine Matrixmultiplikation nur dann möglich ist, wenn eine Matrix genausoviele Spalten besitzt, wie die Andere Zeilen hat. Aus Gründen der Einfachheit werden hier nur quadratische Matritzen verwendet. Besonders wichtig für unsere Tests bei der Matrixmultiplikation ist der nur lesende Zugriff auf die Faktoren der Multiplikation und der ausschließlichen Durchführung von Schreiboperationen auf der Produktmatrix. Dadurch wird eine strikte Zugriffstrennung zwischen den Produkten und der Zielmatrix erreicht.

Eine variable Anzahl von Threads führt eine Matrixmultiplikation zweier Matritzen flexibler Größe aus. Für eine möglichst optimale Verteilung der Zeilen auf die einzelnen Threads wird folgende Berechnung durchgeführt. Einem einzelnen Thread ist die eigene ID, die Gesamtanzahl aller Threads und die Matrixgröße bekannt. Eine Division der Zeilenzahl durch die Anzahl der Threads definiert die Position und Anzahl zu bearbeitender Zeilen bzw. Spalten.

Die Faktorenmatritzen werden pseudozufällig gefüllt. Dabei wird die C-Funktion rand() aus der stdlib.h benutzt. Ohne Initialisierung erzeugt sie in jedem Fall dieselben pseudozufälligen Zahlwerte. Dadurch können beliebige Matritzen dynamisch gefüllt werden, die aber bei gleicher Matrixgröße stets identische Werte besitzen.

Die zu lesenden Faktorenmatritzen werden im gemeinsamen verteilten DSM-Speicher angelegt. Alle Threads erhalten eine Kopie für den nur lesenden Zugriff und können echt nebenläufig damit arbeiten. Der Aufwand für den Zugriff auf diese Matritzen ist im MRMW- und MRSW-Protokoll stets identisch und fließt somit in die Performancetests nicht ein.

Die Position der Zielmatrix definiert die Benutzung des zu verwendenden Protokolles. Wird sie als RELEASE-Variable deklariert (innerhalb des Wirkungsbereiches des Makros *DSM_RELEASE_BEGIN*), wird eine Speicherseite mit dem Typ RELEASE erzeugt. Für sie gelten alle Regeln des trägen Release-Konsistenzmodelles. Ohne diese RELEASE-Deklaration ist das MRSW-Protokoll für die Produktmatrix definiert.

Durchschnittlich wurden für eine ermittelte Zahl ca. 300 bis 500 Testläufe durchgeführt. Die gemessenen Zeiten wurden in einer Datei abgespeichert. Von diesen Zeiten wurden der Median und der Mittelwert errechnet. Der Vorteil des Medians bei der Beurteilung der Ergebnisse ist, daß vereinzelte Kommunikationsstörungen nicht komplett in die Auswertung einfließen. Bei einem deutlichen Unterschied zwischen Median und Mittelwert wurde die Anzahl der Testläufe drastisch erhöht, bis sich beide Werte auf dem gleichen Niveau befanden.

Grundsätzlich ist es bei Performancetests nicht sinnvoll mehr Threads zu erzeugen, als Prozessoren im System vorhanden sind. In diesem Beispiel sollten keine verschiedene Typen von Prozessoren verwendet werden. In dem vorhandenen PC-Cluster wurden für das Testprogramm *test_release* nur die 16 AMD-Prozessoren mit exakt gleicher Hardwareausstattung verwendet. Die Anzahl der zur Verfügung stehenden Prozessoren definieren die kleinste Länge l einer Zeile bzw. Spalte der Matrix. Die daraus resultierende minimale Matrixgröße ist die Dimension (lxl).

Die Inputparameter des Testprogrammes sind die zu benutzende Matrixgröße und die Anzahl zu verwendender Threads. Alternativ zur Menge der Threads kann eine Liste von Rechnernamen übergeben werden, auf denen die Berechnung ablaufen soll. Die Threads werden in neuen Prozessen auf den Rechnern gestartet, wobei pro erzeugtem Thread ein neuer Prozeß angelegt wird. Sie führen lesende und schreibende Zugriffe auf gemeinsame Variablen aus. Diese Zugriffe erfordern den Austausch von Nachrichten. Die Terminierung basiert ebenfalls auf der Basis der Kommunikation.

Das vollständige Programm ist im Anhang A auf Seite 97 enthalten. Nach Definition der maximal möglichen Matrixgröße und Threadanzahl (Zeilen 21 und 22) werden die beiden Faktormatritzen *matrix1* und *matrix2* im gemeinsamen verteilten Speicher des DSM-Systems deklariert (Zeilen 29 und 30). Die Produktmatrix *matrix3* wird bei der Nutzung des MRSW-Protokolles durch das Makro *DSM_NEWPAGE* auf einer neuen Speicherseite deklariert. Damit wird sichergestellt, daß nur eine Speicherseite für diese Matrix verwendet wird, um das *false sharing* zu reduzieren. Bei der Verwendung des trägen Release-Konsistenzmodelles unter dem MRMW-Protokoll deklariert das Makro *DSM_RELEASE_BEGIN* die Produktmatrix zugehörig zu einer Speicherseite vom Typ RELEASE (Zeile 33).

Das Hauptprogramm initialisiert zunächst das DSM-System (Zeilen 91 bis 94). Nach Auswertung der Programmparameter (Zeilen 96 bis 125) werden die Matritzen mit Pseudozufallszahlen gefüllt und die Zeitmessung gestartet (Zeilen 127 und 128). Anschließend werden gemäß der Argumente die Threads erzeugt (Zeilen 131 bis 138). Jeder von ihnen durchläuft die Funktion new_thread. Die übergebenen Funktionsparameter wurden in einem void-Pointer verpackt und in den Zeilen 47 bis 53 wieder ausgewertet. Die Ermittlung der zu bearbeitenden Zeilen der Zielmatrix wird zuerst durchgeführt (Zeilen 51 bis 63). Danach erfolgt die eigentliche Matrixmultiplikation (Zeilen 66 bis 70). Nach einer Barriere aller beteiligten Threads (Zeile 81) wird die Zeitmessung vom Startthread gestoppt und ausgegeben (Zeile 82).

7.1.1 Testlauf 1 - Kleine Matritzen

Bei dem ersten Test wurden ca. 500 Testläufe pro berechnetem Mittelwert durchgeführt. Sie liefen zu verschiedenen Wochenende- und Nachtzeiten, um eventuelle Fremdeinflüsse auszuschalten. Die Matrixgröße stieg von der Dimension (10x10) bis (200x200). Die Anzahl der Worker-Threads wurde von eins bis fünfzehn variiert. Zum Einsatz kam das einfache Testprogramm *test_release* (siehe Kapitel 7.1 auf Seite 56). Die Faktormatritzen *matrix1* und *matrix2* wurden zu Produktmatrix *matrix3* multipliziert.

#Threads	MRSW	MRMW
1	0,00630	0,01012
2	0,57190	0,46265
3	0,58026	0,50923

Tabelle 2: Vergleichslauf Threads 1-3 mit Matrix (60x60) (Zeitangaben in s)

Die Ergebnisse der Versuche von einem bis drei beteiligten Threads einer exemplarischen Matrix der Dimension (60x60) sind in der Tabelle 2 auf der Seite 59 dargestellt. Die großen Unterschiede von einem und mehreren beteiligten Worker-Threads können in einem Diagramm so nicht dargestellt werden. Die durchschnittliche Ausführungszeit des Testprogrammes mit dem MRSW-Protokoll dauert ca. 0,006 Sekunden. Mit dem trägen Release-Konsistenzmodell benötigt das Programm mit einem Thread ca. 0,01 Sekunde. Der erhöhte Rechenaufwand ist auf den zusätzlichen Ko- und Dekodierungsaufwand an der Barriere zurückzuführen.

Der Geschwindigkeitsverlust von zwei beteiligten Threads gegenüber der sequentiellen Abarbeitung ist mit der damit verbundenen Kommunikation der Threads zu erklären. Das Senden und Empfangen der Nachrichten erfolgt über die Verbindungsschicht TCP/IP. Im Vergleich zur eigentlichen Ausführungszeit des Programmcodes benötigt die Kommunikation ein Vielfaches an Zeit. Dieses Verhältnis sollte sich mit steigender Matrixgröße und gleichbleibender Anzahl der Worker-Threads zugunsten der eigentlichen Rechenzeit relativieren.

Die berechneten Mittelwerte des Testprogrammes *test_release* unter Verwendung des MRMW-Protokolles sind im Diagramm 26 auf Seite 60 grafisch veranschaulicht.

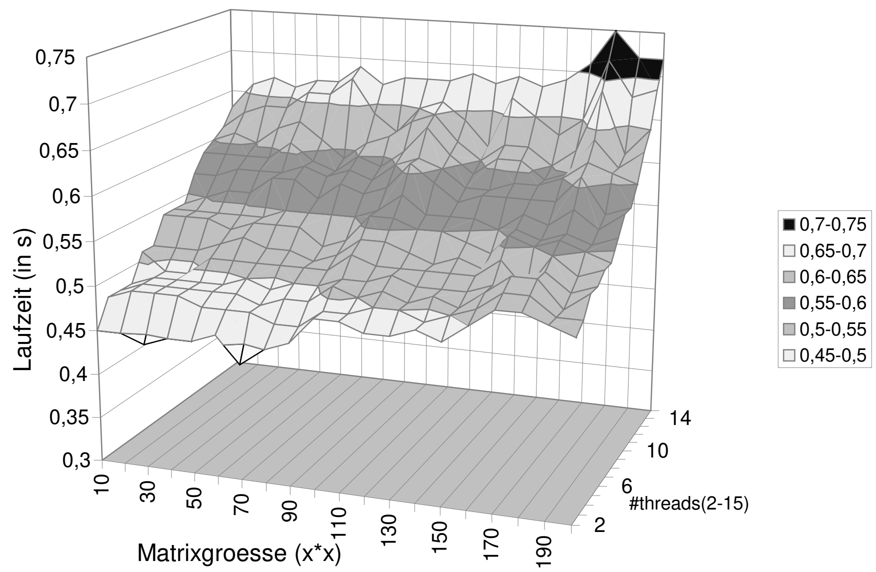

Abbildung 26: Laufzeiten mit dem MRMW-Protokoll (1)

Zu erkennen ist ein ansteigender Verlauf der Programmlaufzeit bei gleicher Matrixgröße und wachsender Anzahl der Worker-Threads. Selbst bei einer Matrix der Dimension (200x200) benötigen 15 Threads mehr Rechenzeit als zwei. Der Synchronisierungsaufwand der Threads ist in jedem Fall im Verhältnis zur eigentlichen Bearbeitungszeit so hoch, daß der Vorteil der nebenläufigen Problemabarbeitung wieder aufgehoben wird. Bei größeren Matritzen sollte sich dieses Verhältnis noch umkehren. Dies wird in einem späteren Kapitel untersucht werden.

Interessant sind die leicht schrägen Höhenlinien, die im Diagramm mit unterschiedlicher Farbgebung markiert sind. Bei einer Matrix der Dimension (10x10) unter Verwendung von 12 Worker-Threads beträgt die Programmlaufzeit ca. 0,55 Sekunden. Diese Zeit wird bei Berechnung einer (200x200)-Matrix schon mit Hilfe von 8 Threads erreicht. Die Höhenlinien zeigen an, daß die Oberfläche der dargestellten dreidimensionalen Funktion in Richtung X-Achse schwach ansteigend ist. Sie zeigen die höhere Rechenzeit, die direkt für die Berechnung der Zielmatrix verbraucht wird.

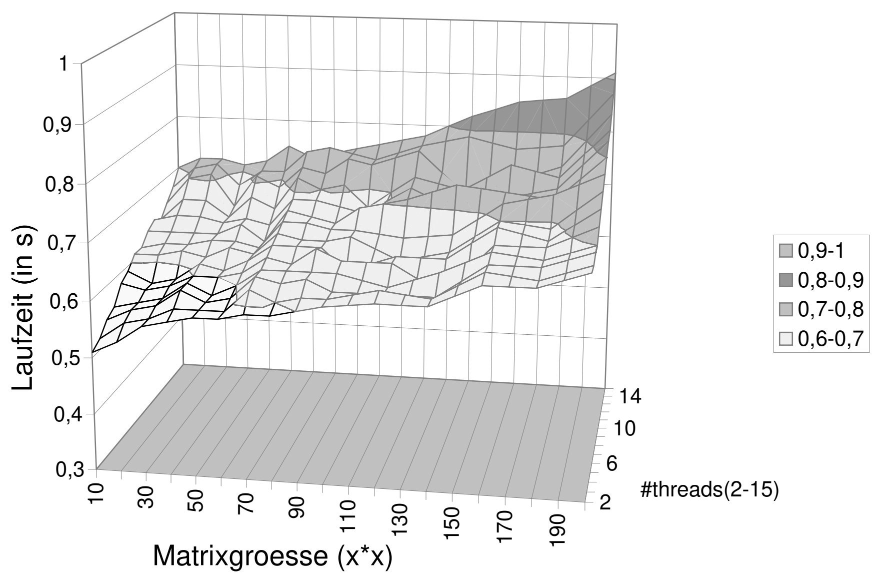

Abbildung 27: Laufzeiten mit dem MRSW-Protokoll (1)

Die durchschnittlichen Programmlaufzeiten des Testprogrammes *test_release* unter Verwendung des MRSW-Protokolles sind im Diagramm 27 auf Seite 61 erkennbar. Für das Diagramm gelten die gleichen Eigenschaften, die bereits das Diagramm 26 auf Seite 60 zeigt. Es ist eine leichte Steigung der Ergebniskurve in Richtung der X- und Y-Achse sichtbar. Der Grad der Neigung ist unter Verwendung des MRSW-Protokolles höher als bei Benutzung des trägen Release-Konsistenzmodelles mit dem MRMW-Protokoll. Bei steigender Anzahl von Worker-Threads steigt der Synchronisationsaufwand der Threads untereinander, was direkt die Gesamtperformance des Testprogrammes beeinflußt. Dieser Aufwand fehlt bei der Benutzung des MRMW-Protokolles.

Von Interesse ist ein direkter Laufzeitvergleich des MRSW- und MRMW-Protokolles. Im Diagramm 28 auf Seite 62 wird ein geringer, aber grundsätzlicher Geschwindigkeitsvorteil des MRMW-Protokolles mit dem trägen Release-

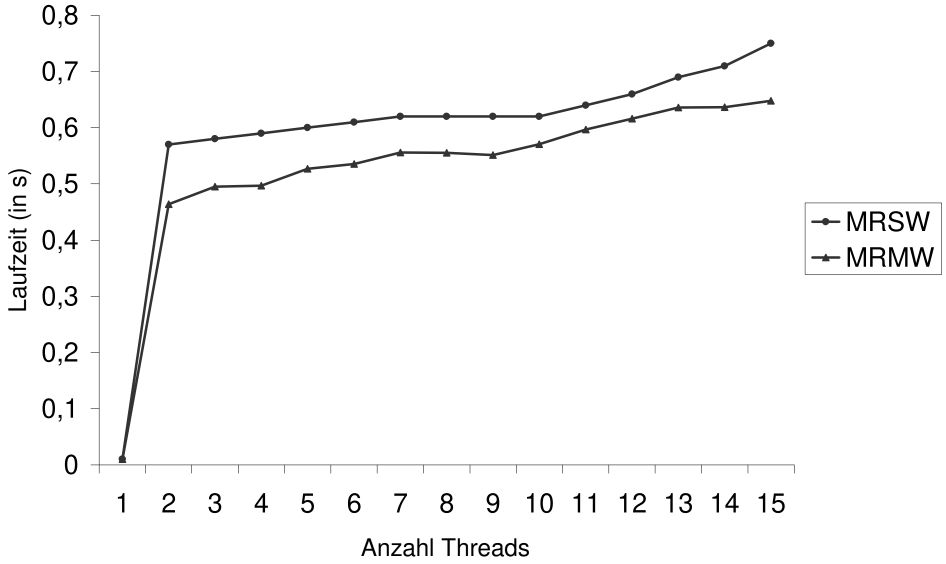

Abbildung 28: Vergleich MRSW- und MRMW-Protokoll (1)

-Konsistenzmodell gegenüber dem MRSW-Protokoll sichtbar. Es ist zu vermuten, daß die Zugriffszeit auf die Zielmatrix im Verhältnis zur Signallaufzeit der Nachrichten im DSM-System zu klein ist, um deutlichere Unterschiede erkennen zu können. Dies wird in den späterenen Kapiteln mit wesentlich größeren Matritzen untersucht werden.

Grundsätzlich besteht bei allen Performancetests ein messbarer Geschwindigkeitsvorteil des Testprogrammes bei der Verwendung des MRMW-Protokolles. Unter Benutzung des MRSW-Protokolles bewerben sich alle teilnehmenden Threads fast gleichzeitig um das Schreibtoken für die Speicherseite der Zielmatrix. Einer dieser Threads bekommt das Token und beschreibt diese für den Zeitraum, in dem er die Zielmatrix berechnet. Alle anderen Threads warten in dieser Zeit untätig auf die Freigabe des Schreibtokens. Dann gelangt er an die Barriere und gibt das Schreibtoken für einen anderen Thread frei. Die Produktmatrix wird fast sequentiell abgearbeitet. Dies gilt für Datenstrukturen, die völlig der Größe nach auf eine Speicherseite passen. Wesentlich umfangreichere Datenstrukturen werden in einem späteren Kapitel untersucht. Es sei hier auf das Kapitel 7.1.3 auf Seite 69 verwiesen.

Unter Verwendung des MRMW-Protokolles erhalten alle beteiligten Worker-Threads vollen Schreibzugriff auf ihre Kopie der entsprechenden Speicherseite. Sie beschreiben echt nebenläufig jeweils ihren Teil der Zielmatrix. Dieser Fakt bedingt die kürzere Gesamtbearbeitungsgeschwindigkeit der Testprogramme mit dem RELEASE-Protokoll. Natürlich dürfen sie bei dem Schreibzugriff auf die Speicherseite vom Typ RELEASE nicht identische Speicherbereiche parallel beschreiben. Ein solches Verhalten ist lt. Voraussetzung nicht definiert. Nach Beendigung des Schreibzugriffes kodieren alle Threads, die von ihnen durchgeführten Veränderungen der Speicherseite in einen Diffstring und synchronisieren sich an der Barriere. Dort erhalten sie die Diffs der anderen beteiligten Threads und arbeiten dessen Veränderungen in ihre Kopien der Speicherseiten ein. Unter der Voraussetzung, daß alle Threads an der Barriere teilnehmen, sind alle beteiligten Threads nach der Barriere über alle schreibenden Speicherzugriffe informiert. Der gesamte verteilte Speicher ist konsistent (siehe Kapitel 5 auf Seite 29). Mit zunehmender Anzahl von Worker-Threads sind mehr Nachrichten für die Synchronisation an der Barriere nötig. Dies erklärt den leichten Anstieg der Programmdauer mit steigender Anzahl der Threads.

Kritischerweise ist anzumerken, daß das aktuelle Beispiel ausgewählt worden ist, weil es die Vorteile des MRMW-Protokolles besonders effizient nutzt. Im Gegensatz zu üblichen Programmen besteht fast die gesamte Ausführungszeit aus miteinander konkurrierenden schreibenden Zugriffen auf eine gemeinsame Speicherseite. Selbst der Ko- und Dekodierungsaufwand hält sich in Grenzen, da ein Thread nur den Speicherblock verändert, der die Zielzeilen enthält. Dazu wird innerhalb dieser Zeilen *jedes* Element der Zielmatrix neu in die Zielmatrix geschrieben. Bei der Diff-Generierung wird die erste veränderte Speicherstelle gesucht. Von dieser Stelle aus wird der nächste unveränderte Speicherblock ermittelt (siehe Kapitel 6.2 auf Seite 43 und Kapitel 6.3 auf Seite 49). Der dazwischenliegende Speicherbereich wird in einem Diff kodiert. In unserem Beispiel ist die Produktmatrix vor der Bearbeitung leer, d.h. mit Nullen gefüllt. Nach der Multiplikationsoperation ist *jedes* Element der Zielmatrix von Null verschieden. Es entsteht ein einzelner Diff, der die gesamten Änderungen eines Threads umfaßt. Daraus resultieren bei n Threads nur n Diffstrings, deren Dekodierungsaufwand relativ minimal ist.

Bei sonst üblichen Schreibzugriffen entstehen Veränderungen der Speicherseiten, die nicht unbedingt zusammenhängen. Weil der kodierende Thread nicht weiß, ob auf die zwischen zwei Veränderungen liegenden Speicherstellen von anderen Threads nebenläufig schreibend zugegriffen wird, darf er diese unveränderten Zellen nicht mit in seinen Diffstring aufnehmen. Daraus entsteht eine Kette von einzelnen Diffstrings, die jeder einen Overhead in Form von Kopfinformationen besitzen. Der damit verbundene Speicherbedarf der Diffs in den Messages und der Dekodierungsaufwand der von den Threads durchgeführt wird, sind wesentlich größer als in diesem Beispiel.

Die Auswirkungen einer effizienteren Programmstruktur wird in einem der folgenden Kapitel untersucht werden (siehe 7.1.5 auf Seite 77).

7.1.2 Testlauf 2 - Einfluß der Seitengröße

Die Größe einer Speicherseite im DSM-System ist begrenzt bzw. nicht beliebig vergrößerbar. Größere Variablen und Strukturen werden im bestehenden DSM-System auf mehreren, aufeinander folgenden Seiten untergebracht. Wenn mehr Pages für eine Struktur verwendet werden und mehrere Threads auf verschiedene Stellen der Struktur schreibend zugreifen, sinkt die Wahrscheinlichkeit des Auftretens von möglichen Konfliktsituationen. Die Anzahl dieser Konflikte (*false sharing*) ist abhängig von der Menge der verwendeten Speicherseiten und der beteiligten Threads. Je größer die Datenstruktur und je geringer die Anzahl der Worker-Threads, desto höher ist die Wahrscheinlichkeit für echt nebenläufiges Arbeiten. Es ist zu vermuten, daß sich unter Verwendung des MRSW-Protokolles diese Wahrscheinlichkeiten direkt auf die Laufzeit des Testprogrammes auswirken. Die gewählten Intervalle der Matrixgrößen im Kapitel 7.1.1 auf Seite 59 sind zu grob, als daß solch geringe Performanceschwankungen sichtbar wurden. Die Testreihe wurde für zwei Threads und einer variablen Matrixdimension im Bereich (30x30) bis (100x100) mit ausreichender Genauigkeit wiederholt.

Für das Testprogramm *test_release* ist die Größe der Produktmatrix *matrix3* ausschlaggebend. Ein Element der Produktmatrix besitzt den Datentyp *long*. Er benötigt vier Bytes Speicherplatz. Die Größe einer Speicherseite im DSM-System ist mit 4096 Bytes festgelegt. Abzüglich einiger Bytes für Kopfinformationen stehen den Programmen ca. 4080 Bytes für Datenstrukturen auf der ersten Seite zur Verfügung. Aus Gründen der Einfachheit werden laut Kapitel 7.1 auf Seite 56 ausschließlich Matritzen quadratischer Größe verwendet. Der Speicherbedarf der Zielmatrix *matrix3* kann linear mit folgender Formel berechnet werden:

$$Speicherbedarf = Zeilen * Spalten * sizeof(ElementTyp) \qquad (3)$$

Eine Matrix einer Dimension von (32x32) besitzt einen Speicherbedarf von knapp 4096 Bytes, also etwas über einer Seite. Überschreitet die Dimension der Produktmatrix die Größe von (45x45), verbraucht die Datenstrutur bereits über 8100 Bytes und belegt bereits zwei komplette Speicherseiten.

Bei einer Benutzung von zwei Threads und einer Matrixdimension von (46x46) hat jeder Worker-Thread eine eigene Speicherseite zum Beschreiben der Zielmatrix zur Verfügung. Die Darstellung 29 auf Seite 66 stellt den Platz im verteilten gemeinsamen Speicher grafisch dar. Die Seiten bestehen aus Blöcken fester Größe. Die Speicherseite, in der die Zielmatrix

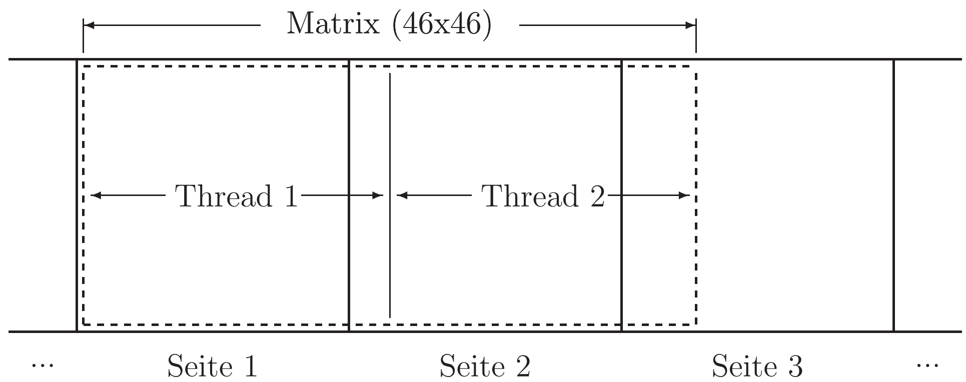

Abbildung 29: Speicheraufbau einer Matrix mit Dimension(46x46)

matrix3 beginnt, wird als *Seite 1* bezeichnet. Die Matrix der Dimension (46x46) reicht über zwei Speicherseiten bis zum ersten Drittel der dritten Seite. Zwei Threads errechnen nebenläufig das Produkt der Matritzen *matrix1* und *matrix2* und schreiben das Produkt in die Produktmatrix *matrix3*. Bei Beginn der Berechnungen bekommt der erste Thread (bezüglich ihrer ID) das Schreibtoken für die erste Speicherseite, während der zweite Thread mit der *Seite 2* beginnt. Haben beide Threads ihre Page abgearbeitet, versuchen sie schreibend auf die zugehörige nächste Seite zuzugreifen. Der zweite Thread bekommt das Token für *Seite 3* sofort, während der erste Worker-Thread noch kurz auf die Freigabe von *Seite 2* durch den zweiten Thread warten muß. Es ist nur eine Synchronisation an der Stelle nötig, an der beide Bereiche aneinanderstoßen.

Zwei benachbarte Threads, die mit dem MRSW-Protokoll und einer ausreichend großen Datenstruktur arbeiten, konkurrieren nur auf der Seite miteinander, die die Grenzen ihrer zugeteilten Datenbereiche beinhaltet.

Im Diagramm 30 auf Seite 67 zeigt sich das verringerte *false sharing* in einem Performancegewinn bei Überschreiten der Matrixgröße der oben definierten Grenze. In der Grafik sind zwei Funktionen dargestellt. Die Erste definiert den Speicherbedarf einer Produktmatrix in Abhängigkeit ihrer Zeilen- bzw. Spaltenzahl. Die zweite Funktion zeigt die Laufzeit des Testprogrammes *test_release* unter der Benutzung von zwei Threads mit zugehöriger abhängiger Matrixgröße. Bei Überschreiten der Matrixdimension (46x46) ist ein Absinken der durchschnittlichen Laufzeit zu erkennen. Vor dieser Schwel-

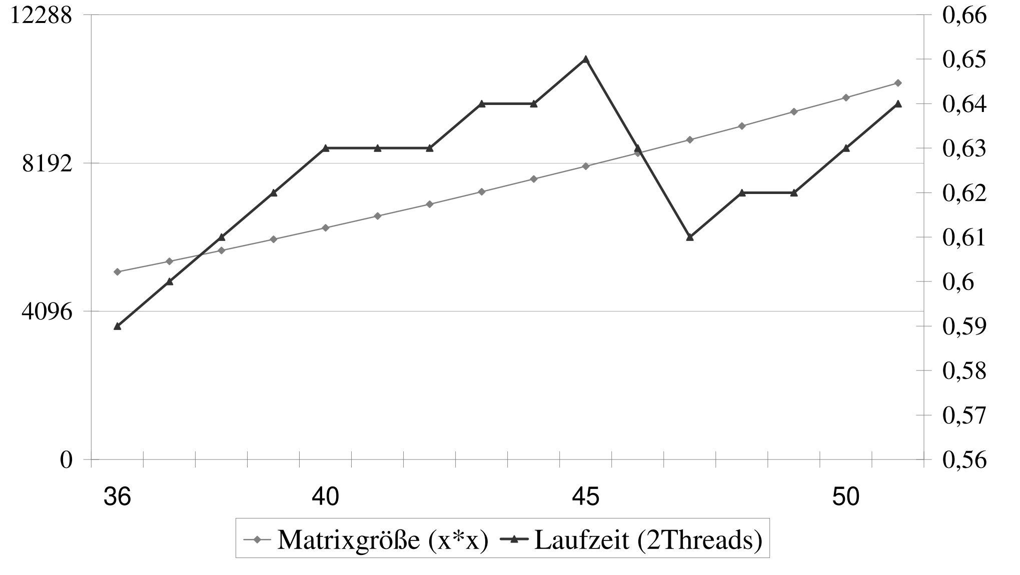

Abbildung 30: Einfluß der Seitengröße (1)

le lag der Beginn des zu beschreibenden Teils der Zielmatrix des zweiten Threads noch auf der gleichen Speicherseite wie der des ersten Threads. In dem Falle, daß der erste Thread zuerst das Schreibtoken erhielt, mußte der andere Thread erst auf die Beendigung des gesamten Rechnungsvorganges seines Vorgängerthreads warten.

Die Testreihe wurde mit Matritzen der Dimension (30x30) bis (92x92) durchgeführt. In Diagramm 31 auf Seite 68 werden die Ergebnisse der gesamten Testreihe dargestellt. Die erste Funktion zeigt das Überschreiten einer neuen Grenze und die zusätzliche Benutzung einer neuen Speicherseite an. Die waagerechten Gitternetzlinien verdeutlichen den Übergang. Die andere Funktion beschreibt die Performance der getesteten Reihen. Bei Überschreiten einer der oben genannten Grenzen erfolgt eine leichte Absenkung der Ablaufgeschwindigkeit des Testprogrammes.

Bei der Benutzung von drei Worker-Threads fällt der Performancegewinn geringer aus, da durch die höhere Threadanzahl eine andere Aufteilung der

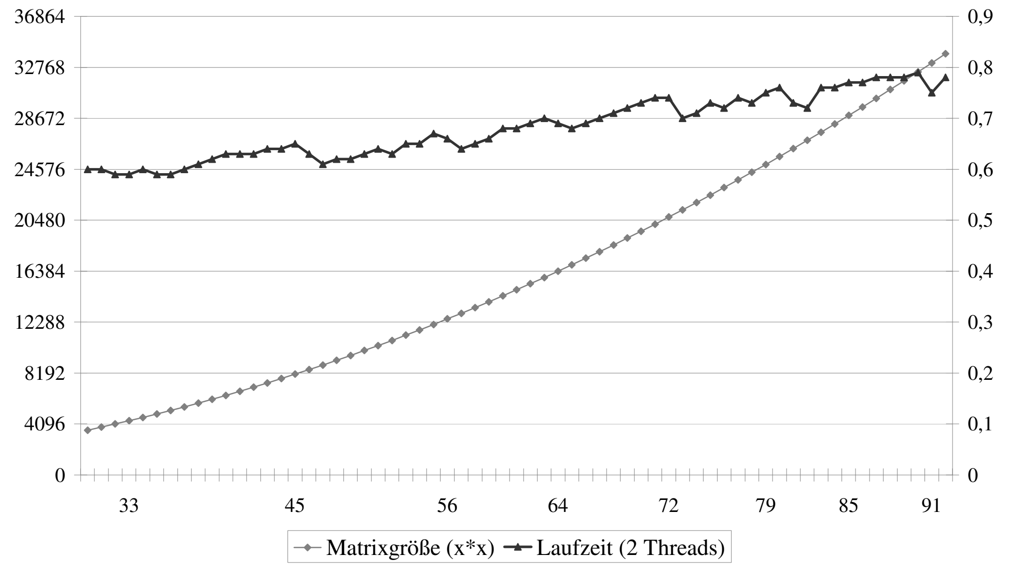

Abbildung 31: Einfluß der Seitengröße (2)

Zielmatrix erfolgt und die Wahrscheinlichkeit für das Auftreten von *false sharing* höher ist. Bei der Verwendung von zwei Speicherseiten (Matrix (46x46)) können sich drei Threads diese zwei Seiten nicht ohne *false sharing* teilen. Bei drei verwendeten Seiten gelingt dieses schon mit drei Threads aber nicht mehr mit vier. Der jeweilige Performancegewinn fällt dabei geringer aus als bei zwei Threads, da die eigentliche Berechnungszeit geringer ist. Der Geschwindigkeitsvorteil wird also flacher bei steigender Matrixgröße und Threadanzahl. Die entstehende Kurve ist mit Wellen vergleichbar, die durch einen Stein im Wasser entstehen. Der Ausschlag flacht sehr schnell ab.

7.1.3 Testlauf 3 - Große Matritzen

Bei den ersten Versuchen wurde deutlich, daß die eigentliche Zugriffszeit auf die Produktmatrix zu kurz war, um deutliche Geschwindigkeitsunterschiede zu erkennen. Eine wesentlich größere Matrix könnte die Berechnungs- und Zugriffszeiten verlängern und vielleicht deutlichere Effizienzunterschiede zwischen dem MRSW- und MRMW-Protokoll aufzeigen. Zumindest die Vorteile nebenläufiger Programmabarbeitung sollten sichtbar werden.

Wie in vorigen Abschnitt erläutert, ist die Größe einer Speicherseite im DSM-System begrenzt (siehe Kapitel 7.1.2 auf Seite 65). Je größer die Datenstruktur und je geringer die Anzahl der Worker-Threads, desto höher ist die Wahrscheinlichkeit für ein echt nebenläufiges Arbeiten. Im maximalen Verhältnis von Strukturgröße und Threadanzahl teilen sich nur zwei Threads eine Seite. In diesem Fall ist die Datenstruktur so groß, daß die zugeteilten abzuarbeitenden Datenbereiche über mehrere Speicherseiten reichen. Ein einzelner Thread b, der die Zeilen x bis y abarbeiten soll, gerät dann nur mit dem *vorigen* Thread a in Konflikt, wenn b auf Zeile x und a auf Zeile *x-1* zugreifen möchte. Der verwendete Begriff *vorig* ist im Sinne der zugeordneten ID-Nummer des Threads zu verstehen. Äquivalent bestehen konkurrierende Zugriffe von Thread b mit *nachfolgenden* Thread c bei Zugriff auf die Zeilen x und *x+1*.

Benötigt die Zielmatrix mehr Speicherseiten als Worker-Threads vorhanden sind, wartet unter Verwendung des MRSW-Protokolles jeweils ein Thread höchstens auf das Ende eines anderen. Die Größe einer Speicherseite im DSM-System ist mit 4096 Bytes festgelegt. Ein Element der Produktmatrix besitzt den Datentyp *long* und verbraucht vier Bytes Speicherplatz. Die maximale Anzahl von Threads in diesem DSM-System sind systembedingt 15 Threads (siehe Kapitel 7 auf Seite 7). Die minimale Größe der Produktmatrix, mit der dieses fast nebenläufige Arbeiten möglich ist, berechnet sich aus diesen Parametern:

$$x = \sqrt{(\#Threads * Seitengröße)/sizeof(ElementTyp)} \qquad (4)$$

Die Variable x definiert die minimale Dimension der Produktmatrix. Die geltenden Beschränkungen ergeben folgenden Wert:

$$x = \sqrt{(15 * 4096)/4} \approx 124 \qquad (5)$$

Für den Elementtyp *long* können die Threads fast nebenläufig auf die Ziel-

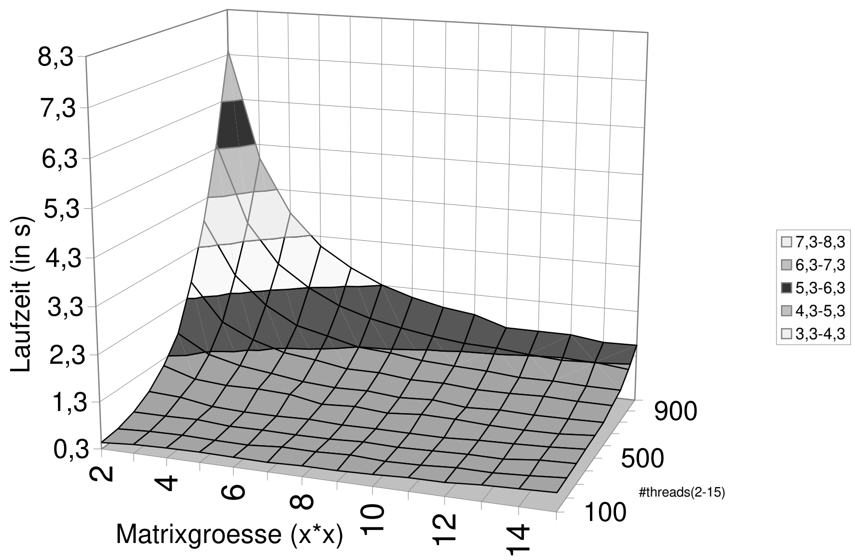

Abbildung 32: Laufzeiten mit dem MRMW-Protokoll (2)

matrix ab der Dimension (124x124) zugreifen. Der Elementtyp *char* verlangt die minimale Dimension (248x248). Ab dieser Größe werden keine Performancegewinne durch reduziertes *false sharing* mehr erreicht.

Im Diagramm 32 auf Seite 70 ist der durchschnittliche Verlauf der Matrixmultiplikation mit einer Zielmatrix von der Dimension (200x200) bis (900x900) dargestellt. Getestet wurden Matritzen bis zu einer Dimension von (3000x3000). Die Verläufe dieser Tests setzen sich exponentiell fort, sind aber grafisch nicht gut darstellbar. Am deutlichsten ist die stark erhöhte Ausführungszeit für wenig Threads bei sehr großen Matritzen. Hier überwiegt die Rechenzeit der Multiplikation die Signallaufzeit von Nachrichten im System stark.

Mit steigender Anzahl der Worker-Threads reduziert sich die Gesamtausführungszeit erheblich. Interessant ist das Verhältnis zwischen gewonnener Geschwindigkeit und Anzahl der verwendeten Worker-Threads. In Diagramm 33 auf Seite 71 ist ein Beispiel einer Zielmatrix von der Dimension

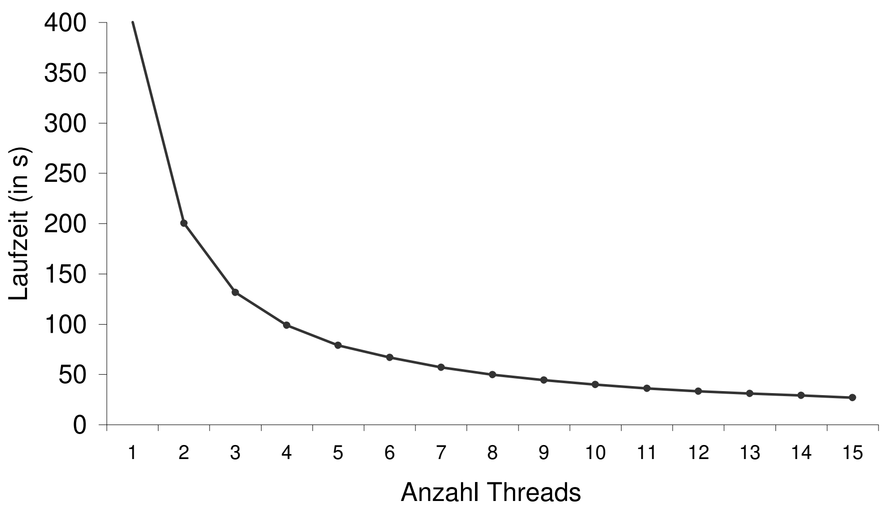

Abbildung 33: Laufzeiten mit dem MRMW-Protokoll mit 2 Threads

(3000x3000). Die Anzahl der Threads wurde von eins bis fünfzehn variiert. Der Performancegewinn mit wenigen Threads ist gravierend, er wird immer schwächer je mehr Threads an der Berechnung beteiligt wird. Ab dem fünften teilnehmenden Thread relativiert sich der entstehende Performancegewinn bereits erheblich, ab dem zehnten Thread lohnt es den Mehraufwand kaum noch. Bei angenommener weiterer Erhöhung der Zahl der Worker-Threads wird die Funktion wieder steigend werden. In diesem Fall wird der Synchronisationsaufwand die nebenläufige Berechnungszeit wieder übersteigen. Alle Berechnungen unterschiedlichster Matrixgrößen zeigen den gleichen exponentiellen Verlauf.

Unter Verwendung des MRSW-Protokolles zeigt die Grafik 34 auf Seite 72 einen ähnlichen Verlauf. Bei sehr großen Matritzen besteht bei einer im Verhältnis zur Matrixgröße geringen Zahl von Threads nur wenig Konkurrenz. Der starke Geschwindigkeitsgewinn bei der Berechnung einer sehr großen Matrix mit der Dimension (2000x2000) durch relativ wenige Threads

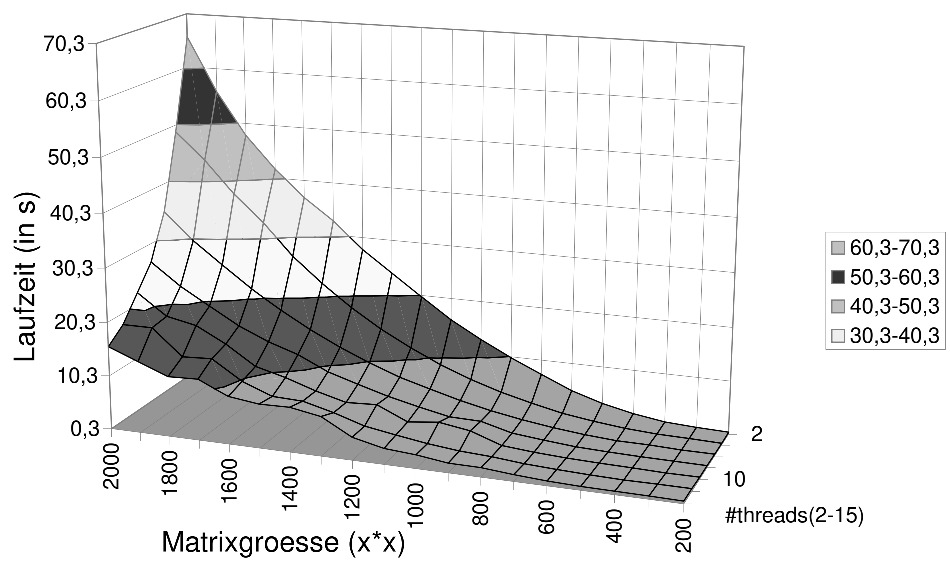

Abbildung 34: Laufzeiten mit dem MRSW-Protokoll (2)

ist auf der linken Seite des Diagrammes deutlich zu erkennen. Unter theoretischer Hinzunahme weiterer Threads würde sich diese Funktion wie die im Diagramm 33 auf Seite 71 verhalten und weiter abflachen. Bei kleineren Matritzen wird das erreichbare Optimum mit wesentlich weniger Worker-Threads erreicht. Für die Berechnung von Matritzen bis zu einer Größe von (1000x1000) sind im wesentlichen bereits fünf Worker-Threads ausreichend, weitere Threads bringen keinen besonderen Performancegewinn mehr. Bei einer wesentlich größeren Produktmatrix würden sich auch mehr als 15 Threads als effektiv erweisen.

Ein direkter Vergleich beider Protokolle wird im Diagramm 35 auf Seite 73 gezeigt. Dargestellt wird das Laufzeitverhalten einer exemplarischen Matrix der Dimension (2000x2000). Es ist ein deutlicher Geschwindigkeitsvorteil bei der Benutzung des MRMW-Protokolles sichtbar. Es gelten die gleichen Kritikpunkte am gewählten Testprogramm *test_release*, die schon am Ende des Kapitels 7.1.1 auf Seite 59 festgestellt wurden. Dieses Testprogramm wur-

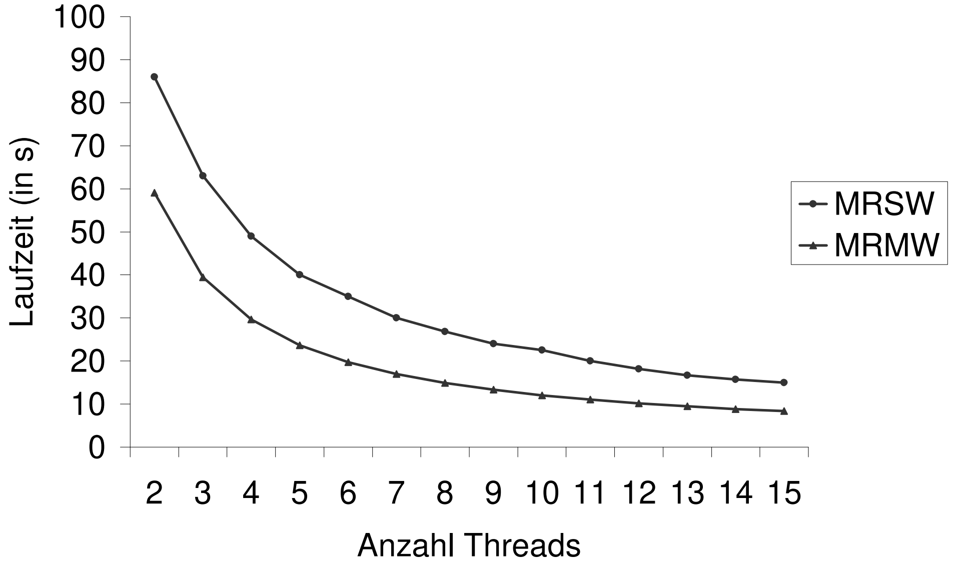

Abbildung 35: Vergleich MRSW- und MRMW-Protokoll (2)

de wegen seiner besonderen Eignung für den Test des MRMW-Protokolles entworfen. Ein deutlich optimierter Quellcode wird unter Kapitel 7.1.5 auf Seite 77 getestet.

Wie in diesem Kapitel behandelt, sinkt der Performancegewinn mit steigender Anzahl der Worker-Threads. Die Kurven der Funktionen nähern sich einem gedachten Minimum, das nicht überschritten werden kann. Mit steigenden Threadzahlen würde die Gesamtlaufzeit des Testprogrammes wieder steigen. Der Geschwindigkeitsvorteil des MRMW-Protokolles beruht im wesentlichen auf der wegfallenden Synchronisation für den schreibenden Zugriff auf die verteilten gemeinsamen Speicherseiten. Der Abgleich der Schreibzugriffe erfolgt dort erst in der Barriere. Bei einer solch großen Matrix ist die Zahl der Konfliktsituationen relativ gering. Deshalb fällt der Performanceunterschied nicht gravierender aus.

7.1.4 Testlauf 4 - Mittlere Matritzen

Im Kapitel 7.1.1 auf Seite 59 wurde kleine Matritzen bis zu der Dimension (200x200) untersucht. Je mehr Threads das gestellte Problem abarbeiteten, desto größer wurde die Gesamtlaufzeit des Testprogrammes. Kapitel 7.1.3 auf Seite 69 behandelte sehr große Matritzen. Dort wurde deutlich, daß viele Worker-Threads Matritzen dieser Größenordnung effizienter als wenig Threads abarbeiten können. Zwischen diesen beiden Extrema muß eine Matrixdimension existieren, in der sich dieses Verhältnis umkehrt.

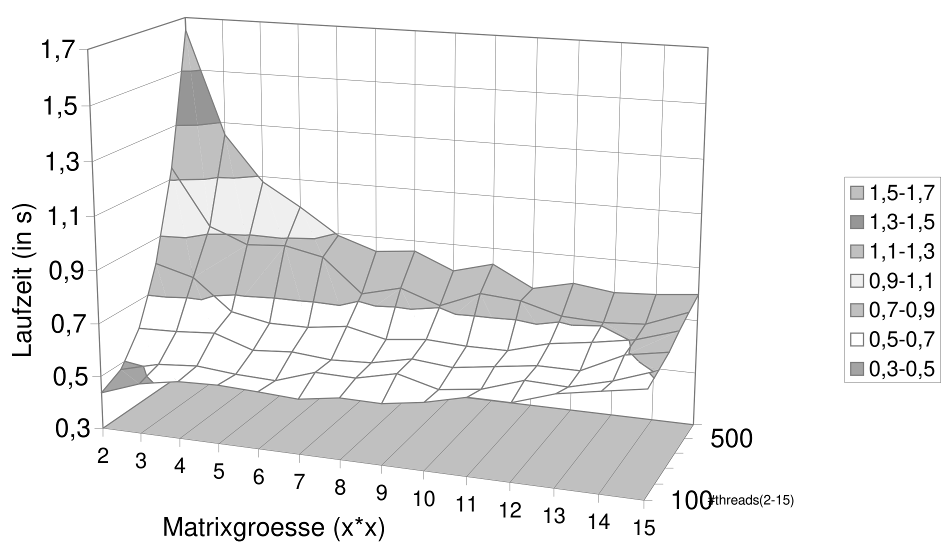

Abbildung 36: Laufzeiten mit dem MRMW-Protokoll (3)

In der Grafik 36 auf Seite 74 wird ein vergrößerter Auschnitt des Testlaufes aus Kapitel 7.1.3 auf Seite 69 dargestellt. Unter Verwendung des MRMW-Protokolles wird die Dimension der Produktmatrix von (100x100) bis zu (500x500) variiert. Bei der kleinsten Dimension (100x100) der Zielmatrix ist mit größer werdender Threadzahl eine steigende Tendenz der Gesamtgeschwindigkeit des Testprogrammes sichtbar. Die maximale Dimension

(500x500) zeigt einen stark sinkenden Verlauf. Die Matritzengrößen der Dimensionen (200x200) und (300x300) markieren den Wendepunkt zwischen beiden Tendenzen. Sie zeigen bei noch relativ wenigen Threads einen sinkenden Verlauf und erreichen bei der Benutzung von ca. sieben bis zehn Worker-Threads ein lokales Minimum. Unter Hinzunahme zusätzlicher Threads verschlechtert sich die Gesamtlaufzeit des Testprogrammes wieder. Der wachsende Synchronisationsaufwand übersteigt die Zeitersparnis bei der verteilten Durchführung der Multiplikation. Bei dieser Matrixgröße benötigen zwei Threads fast die gleiche Ausführungszeit wie fünfzehn. Erst bei Matritzen oberhalb der Dimension (300x300) lohnt sich eine verteilte Berechnung mit mehr Worker-Threads.

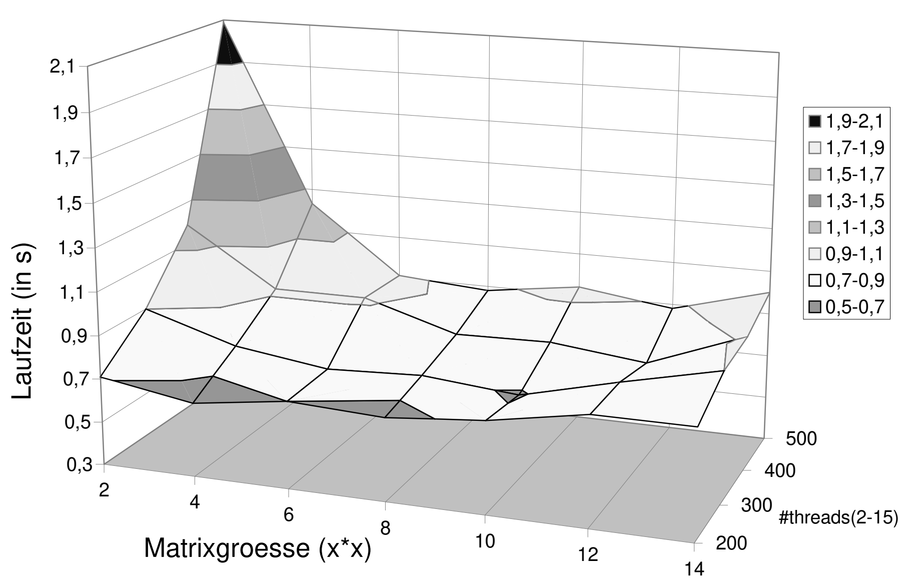

Abbildung 37: Laufzeiten mit dem MRSW-Protokoll (3)

Bei Verwendung des MRSW-Protokolles wird im Diagramm 37 auf Seite 75 ein ähnlicher Verlauf gezeigt. Auch hier liegt das lokale Minimum bei einer Zielmatrix der Dimension (300x300) und bei der Benutzung von ca. neun Threads. Dieser 'Knick' im Verlaufsgraphen ist vermutlich vom verwendeten

Protokoll unabhängig. Der Elementtyp der Produktmatrix und der eigentliche Berechnungsaufwand beeinflussen diesen Umschwung dagegen stark.

7.1.5 Testlauf 5 - Effizienterer Code

Im oberen Kapitel 7.1.1 auf Seite 59 wurde bereits kritisch auf das verwendete Testprogramm eingegangen. Während der gesamten Zeit, in der der Thread seinen Teil der Zielmatrix berechnet, wird mit Verwendung des MRSW-Protokolles die gesamte Speicherseite dieses Teiles für alle anderen Worker-Threads blockiert. Unter Benutzung des MRMW-Protokolles existiert diese Blockierung nicht. Dafür wird dem Programmierer die Beschränkung auferlegt, nicht nebenläufig auf identische Speicherstellen zwischen zwei Synchronisationspunkten schreibend zuzugreifen (Kapitel 3.3 auf Seite 15).

Unabhängig vom gestellten Problem wird versucht werden, eine effizientere Programmierung durchzusetzen. Dazu muß unter Verwendung des MRSW-Protokolles der schreibende Zugriff auf die gemeinsamen Speicherseiten minimiert werden. Das MRMW-Protokoll wird weitgehend durch folgende Beschränkungen simuliert:

- Es werden zwei Synchronisationspunkte in Form von Barrieren eingeführt.
- Zwischen zwei aufeinander folgenden Synchronisationspunkten darf nicht schreibend auf identische Speicherbereiche zugegriffen werden.
- Nach Ausführung des ersten Synchronisationspunktes wird eine lokale Kopie der zu beschreibenden Seite angelegt.
- Die eigentlichen Berechnungen werden durchgeführt. Es wird nur schreibend auf die lokal gepufferte Kopie der Speicherseite zugegriffen.
- Kurz vor dem zweiten Synchronisationspunkt werden die veränderten Teile der lokalen Kopie in die originale Zielseite geschrieben.

Die 'lokale Kopie' der gesamten Speicherseite wird effizient durchgeführt, indem nur die zu beschreibenden Variablen dieser Seite kopiert werden. Diese Kopien können bei Arrays mit relativ geringem Aufwand mittels des Befehls *memcpy* durchgeführt werden. Der Befehl *memcpy* ist allerdings in der Realität nicht sehr effizient implementiert. Der Zeitraum des letzten Schreibvorganges, der den eigentlichen konkurrierenden Schreibzugriff auf die verteilte Seite bedeutet, wird aber minimiert. Diese Verkürzung verringert die Wahrscheinlichkeit des Auftretens von *false sharing* und minimiert die Wartezeit

```
void new_thread(void *arg)
{
  ... nichtrelevanten Teil des Codes gestrichen ...

  /* matrix multiplication */
  if (down>=up)
    for( i=up; i<=down; i++ )
      for( j=0; j<rows; j++ )
        for( matrix3[i][j]=0,k=0; k<rows; k++ )
          matrix3[i][j] += (matrix1[i][k]*matrix2[k][j]);

  dsm_barrier(&barrier, max);

  ... nichtrelevanten Teil des Codes gestrichen ...
}
```

Abbildung 38: Matrixmultiplikation, ineffektiv

eines Threads auf die Beendigung des Schreibzugriffes der gemeinsamen verteilten Speicherseite durch einen anderen Worker-Thread bis zum Erhalt des Schreibtokens.

Die zentrale Funktion des Testprogramm *test_release* ist die Methode *new_thread*. In ihr wird die Matrixmultiplikation durchgeführt. Die verteilte gemeinsame Speicherseite um die alle Worker-Threads konkurrieren, ist die Produktmatrix *matrix3*.

Im Listing 38 auf Seite 78 sind die relevanten Teile dieser Methode der bisherigen Performancetests dargestellt. Der obere Teil zeigt die Schleife, in der die Multiplikation des jeweiligen Elementes aus den Quellmatritzen *matrix1* und *matrix2* durchgeführt wird. Während dieser Schleife blockiert der ausführende Thread die Speicherseite, in der sich der zu beschreibende Teil der Produktmatrix *matrix3* befindet. Danach erfolgt die Barriere, an der alle Threads diesen Codeabschnitt beenden. Die oberen Kapitel testeten ausführlich diesen Codeabschnitt.

Da in diesem Beispiel alle Threads ihren Lebenszyklus in dieser Funktion beginnen und die Zielmatrix *matrix3* mit Nullen gefüllt ist, wird hier kein erster Synchronisationspunkt benötigt. Die Barriere in Zeile 12 im Listing 38 wird als zweiter Synchronisationspunkt festgelegt. Der zu beschreibende

Teil der Produktmatrix wird durch die Variablen *up* und *down* bestimmt, die die Ergebniszeilen des Threads definieren. Die Variable *rows* symbolisiert die Breite der Zielmatrix.

Der lokale Puffer der Produktmatrix wird als

$$ElementTyp : buf[down - up + 1][rows] \tag{6}$$

deklariert (Zeile 6 im Listing 39 auf Seite 80). Der Rest der Produktmatrix wird vom Thread nicht benötigt, da er weder gelesen noch beschrieben wird. Der zu puffernde Teil der Zielmatrix wird über einen *memcpy*-Befehl in den Puffer kopiert (Zeile 8). Die Berechnung der Produktmatrix wird nun im Puffer *buf* zwischengespeichert. Die Zeile 19 schreibt das Ergebnis in die Zielmatrix zurück. Dieser *memcpy*-Befehl ist der einzige schreibende Zugriff auf die verteilte gemeinsame Seite, die die Produktmatrix enthält. Mit dem MRSW-Protokoll ist dies die einzige Codestelle, in der alle Threads miteinander um diese Seite konkurrieren.

Die Ergebnisse dieses Performancetests müssen in drei Bereiche unterteilt werden. Die Effektivität der Codeoptimierungen sind für kleine, große und sehr große Matritzen unterschiedlich. Stellvertretend für kleine Matritzen ist der Laufzeittest für eine Matrix der Dimension (20x20) in der Abbildung 40 auf Seite 81 dargestellt. Für Matritzen dieser Größenordnung entsteht eine vernachlässigbar kleine Berechnungszeit für die eigentliche Multiplikation. Für den, wenn auch nur sehr kurzen Seitenzugriff ist eine Synchronisation für die Threads mit dem MRSW-Protokoll nötig. Für sehr wenig beteiligte Worker-Threads ist noch eine minimale Laufzeitverbesserung des optimierten Quellcodes zu erkennen, die aber mit steigender Threadzahl gegen den des MRSW-Protokolles konvergiert.

Die Resultate für große Matritzen sind im Diagramm 41 auf Seite 82 abgebildet. Eine große Matrix ist mit maximaler Größe definiert, die noch auf eine Speicherseite passt. Die abgebildete Matrix der Dimension (64x64) besitzt 4096 Elemente. Der ElementTyp der Zielmatrix ist auf den Typ *char* reduziert worden, um eine maximal große Produktmatrix auf einer Speicherseite zu erhalten. Matritzen dieser Größenordnung besitzen eine relativ hohe Berechnungszeit der Multiplikation für geringe Threadzahlen. Für diesen Bereich ist eine maximale Laufzeitverbesserung möglich. Die relativ langen Berechnungen werden echt nebenläufig durchgeführt und für wenig Threads sind nicht viele Synchronisationen nötig. Diese Synchronisation fällt unter Verwendung des MRMW-Protokolles ersatzlos weg, deshalb ist ein ständiger

```
void new_thread(void *arg)
{
  ... nichtrelevanten Teil des Codes gestrichen ...

  {
    /* declaration buffer */
    long save[down-up+1][rows];

    /* copies the part of matrix3 to the buffer */
    memcpy ( &save[0][0], &matrix3[left][0],
            (down-up+1)*rows*sizeof(long) );

    /* matrix multiplication */
    if (down>=up)
      for( i=up; i<=down; i++ )
        for( j=0; j<rows; j++ )
          for( matrix3[i][j]=0,k=0; k<rows; k++ )
            save[up-i][j] += (matrix1[i][k]*matrix2[k][j]);

    /* copies back the buffer to the part of matrix3 */
    memcpy ( &matrix3[left][0], &save[0][0],
            (right-left+1)*rows*sizeof(long) );

  }

  dsm_barrier(&barrier, max);

  ... nichtrelevanten Teil des Codes gestrichen ...
}
```

Abbildung 39: Matrixmultiplikation, effektiv

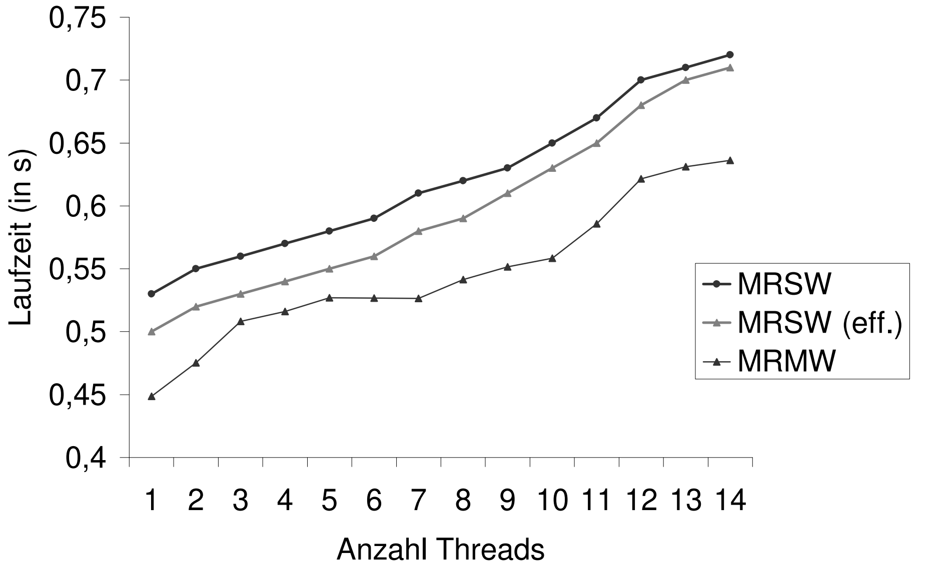

Abbildung 40: Matrix der Dimension (20x20)

Laufzeitvorteil mit diesem Protokoll erkennbar. Mit steigender Anzahl von Worker-Threads relativiert sich der Geschwindigkeitsgewinn des optimierten Quellcodes des MRSW-Protokolles zum originalen. Die eigentlichen Berechnungszeiten reduzieren sich, und der Synchronisationsaufwand der Threads wächst.

Abbildung 42 auf Seite 83 zeigt den Laufzeittest mit einer Matrix der Dimension (2000x2000). Sehr große Matritzen werden über viele Speicherseiten verteilt. Eine Beispielmatrix mit der oben gewählten Dimension mit insgesamt vier Millionen Elementen, einer Seitengröße von 4096 Bytes und einem Matrixelementtyp *char* belegt ca. 977 Speicherseiten. Bei Matritzen dieser Größenordnungen sind relativ lange Berechnungszeiten nötig. Die Multiplikation der einzelnen Threads erfolgt aber mit jedem Protokoll fast nebenläufig, da benachbarte Threads nur in der Nähe ihrer Start- und Endelemente um eine Speicherseite konkurrieren. Der Laufzeitunterschied sinkt bei allen Protokollen mit steigender Matrixgröße und verhältnismäßig wenig Threads auf

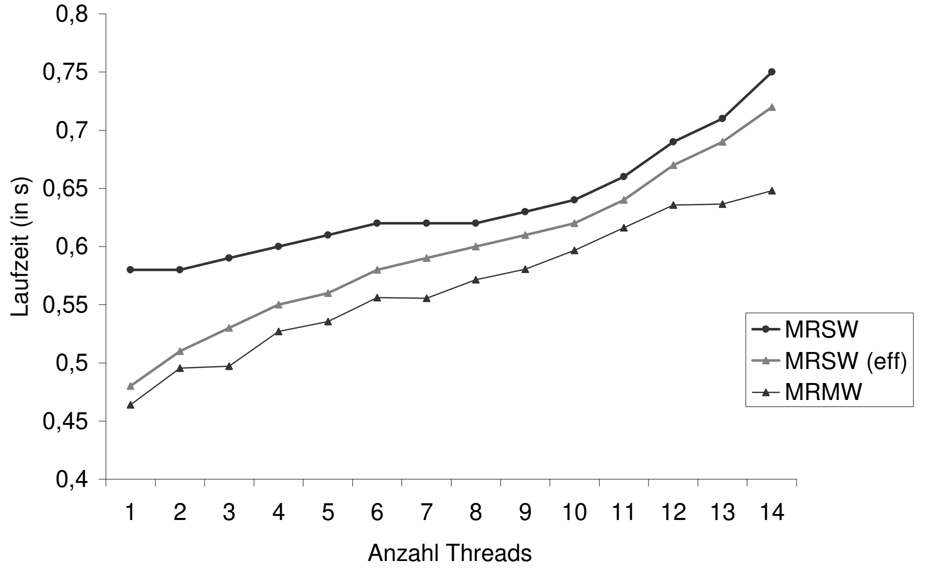

Abbildung 41: Matrix der Dimension (60x60)

Null zusammen.

Zusammenfassend muß auf die starke Abhängigkeit von dem gestellten Problem hingewiesen werden. Durch den Performancegewinn des optimierten Codes unter Verwendung des MRSW-Protokolles in nur mittleren Bereichen kann das Ergebnis nicht für bestimmte Aufgabenbereiche verallgemeinert werden. Programme, die extrem umfangreiche, lang dauernde Berechnungen auf einer Datenstruktur ausführen, die auf einer bzw. wenig Speicherseiten untergebracht ist und relativ wenig Worker-Threads zur Verfügung hat, kann vom optimierten Code profitieren. Positiv ist zu nennen, daß nur geringe Änderungen für die Umstellung dieses Beispieles notwendig waren. Der Aufwand für die Entwicklung eines MRMW-Protokolles ist in jedem Fall erheblich aufwändiger.

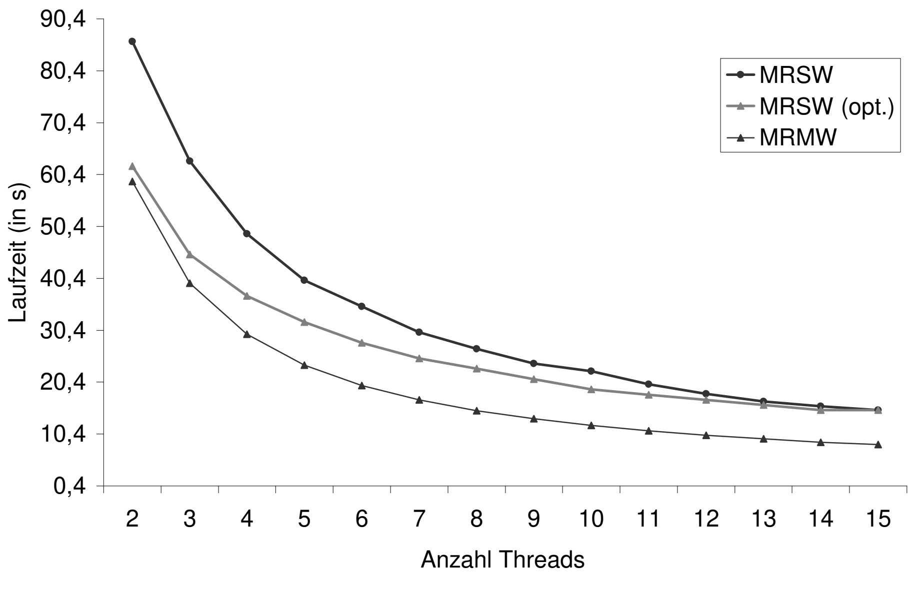

Abbildung 42: Matrix der Dimension (500x500)

7.2 Das Testprogramm FFT

Von der bereits existierenden SPLASH-2-Benchmark-Suite wird das Programm FFT zum Performancetest benutzt [32, 33]. In FFT (*Fast Fourier Transformation*) wird eine schnelle Fourier Transformation ausgeführt, wobei eine komplexe 1D-Version des *radix p sixstep FFT* Algorithmus [34, 35] zur Anwendung kommt. Die Transformation rechnet auf der Grundlage einer nxn-Matrix, die in diesem Test mit 4096 Datenpunkten (n) festgelegt wurde. Das Testprogramm wurde auf 4 Rechnern ausgeführt. In Abbildung 43 auf Seite 84 sind die Ausführungsgeschwindigkeiten der Programme unter Benutzung des MRSW- und MRMW-Protokolles dargestellt.

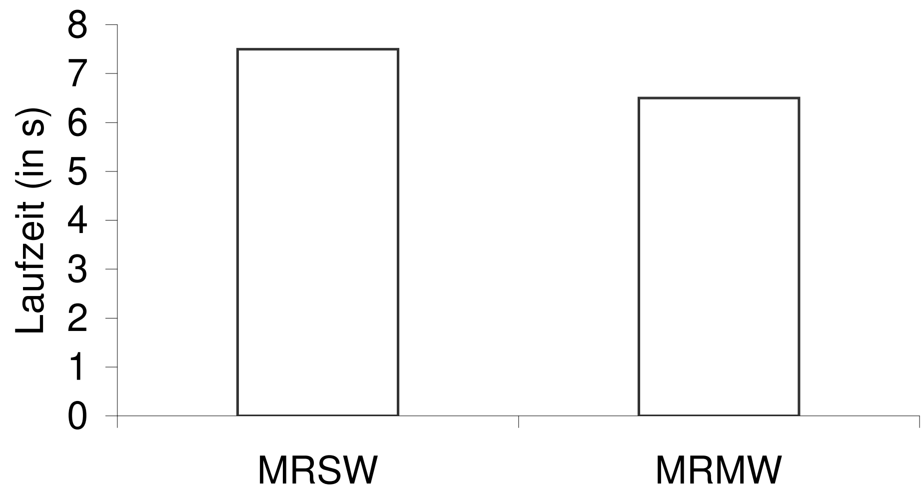

Abbildung 43: Vergleich FFT mit MRSW- und MRMW-Protokoll

Es ist nur ein geringer Unterschied der Programmlaufzeiten der beiden Protokolle zu erkennen. In Kapitel 7.1.5 auf Seite 77 wurde ausführlich auf die Implementation effizienteren Codes eingegangen. Diese Ideen sind im FFT bereits implementiert. Zu Beginn kopiert jeder Worker-Thread die Quellmatrix in eine lokale Kopie. Mit dieser Kopie werden die Berechnungen durchgeführt. Abschließend werden die Resultate zurück in die Zielmatrix kopiert. Dies ist der einzige konkurrierende Zugriff auf den gemeinsamen verteilten Speicher, der schreibend stattfindet. Demzufolge kann der Performanceunterschied nicht so gravierend ausfallen, da auch unter Verwendung des MRMW-Protokolles eine Synchronisation stattfinden muss.

Die Fast Fourier Transformation ist ein Algorithmus, der sehr gut auf mehrere Prozessoren verteilbar ist. Im Anhang B auf Seite 103 ist der Quellcode des Testprogrammes abgebildet.

8 Schlußwort / Ausblick

Gegenstand dieser Arbeit war eine Erweiterung der Funktionalität des bestehenden DSM-Systems [27, 28] um das MRMW-Protokoll. Dies wurde durch die Benutzung des trägen Release-Konsistenzmodelles erreicht. Es wurden der gegenseitige Ausschluß und Barrieren mit Hilfe des MRMW-Protokolles implementiert [29, 3].

Eine erfolgte Änderung am bestehenden DSM-System war die Erzeugung eines Moduls, welches Zeitstempel verwalten kann. Zusätzlich wurden die Module, die Diffs und Twins verwalten, den neuen Erfordernissen angepaßt.

Für die Implementation des MRMW-Protokolles wurden die Module *Timestamp* und *Release* erzeugt und die Module *Diff, Twin, Pagefunc, Pagetab* und *Barrier* verändert. Diese Arbeiten sind bereits vollständig abgeschlossen.

Nicht behandelt wurden alle Aspekte einer zufriedenstellenden Speicherfreigabe der erzeugten Diffstrings. Eine Zeichenkette, die die Veränderungen eines Prozesses an einer Speicherseite beinhaltet, kann erst dealloziiert werden, wenn sie von allen beteiligten Prozessen behandelt und in die jeweiligen Kopien der Speicherseiten eingefügt wurde. Nur nach erfolgreichem Abschluß einer Barriere, an der alle Prozesse beteiligt waren, können alle jemals erzeugten Diffstrings ersatzlos zerstört werden.

Ein Teil dieser Aufgaben wurden im Rahmen der gleichnamigen Studienarbeit zu diesem Thema erledigt. Zusätzlich zur Implementation wurden umfangreiche Vergleichsmessungen durchgeführt, in denen die Laufzeiteffizienz des MRMW-Protokolles mit dem bereits bestehenden MRSW-Protokoll verglichen wurde. Es wurde meist ein geringer Geschwindigkeitsvorteil unter Benutzung des trägen Release-Konsistenzmodelles gemessen. Dieser Vorteil wird durch zusätzliche Beschränkungen des Programmierers und einen erhöhten Implementationsaufwand erkauft. Zusätzlich zu diesen Nachteilen besteht eine extreme Problemabhängigkeit des Programmes im Verhältnis zu seiner Performance. Selbst bei Variation kleinster problemunabhängiger Programmablaufparameter wie z.B. der Speicherseitengröße ändert sich die Geschwindigkeit des Programmes zum Teil erheblich. Durch zusätzliche Optimierung (Verringerung der Zugriffe auf den gemeinsamen Speicherbereich) kann die Effizienz eines Programmes mit dem MRSW-Protokoll beschleunigt werden. Unter Verwendung des MRMW-Protokolles treten auch völlig neue Probleme auf. Durch die fehlende automatische Speicherfreigabe der kodierten Diffstrings kann die maximal mögliche Größe einer zu verschickenden Nachricht überschritten werden. In diesem Fall müßten zwei Nachrichten geschickt werden, was die Signallaufzeit verdoppelt.

Ein im Sinne einer globalen Uhr *gleichzeitiger* schreibender Zugriff auf eine identische Speicherstelle von zwei verschiedenen Threads kann es nie geben. Somit ist die Implementation eines MRMW-Protokolles immer eine komplexere Variation eines MRSW-Protokolles mit zusätzlichen Beschränkungen des Programmierers. Eine Umformulierung der Programmierpflichten zur Geschwindigkeitssteigerung hin (Minimierung des gemeinsamen Zugriffes, Berechnungen auf einer Speicherkopie) bzw. eine problemabhängige Optimierung der externen Parameter (Speicherseitengröße) könnte auch unter Verwendung des MRSW-Protokolles eine bessere Performance erzielt werden. Der zusätzliche Implementationsaufwand und die Problematik der Speicherfreigabe sollten ebenfalls mitbetrachtet werden. Durch die extreme Problemabhängigkeit bedingt, kann keine definitive Aussage über Vor- und Nachteile des jeweiligen Protokolles getroffen werden.

Abschließend werden die Vor- und Nachteile der beiden Protokolle gegenübergestellt. Es ist unter Verwendung des MRMW-Protokolles mit träger Release-Konsistenz in den meisten Fällen ein zum Teil deutlicher Geschwindigkeitsvorteil genüber dem MRSW-Protokoll meßbar. Dieser Performanceunterschied ist durch effizientere Programmierung unter Verwendung des MRSW-Protokolles bedingt relativierbar, indem der Zugriffsaufwand auf gemeinsam benutzte Speicherstellen minimiert wird. Zusätzlich muß auf den Aspekt der Transparenz eingegangen werden. Unter Verwendung des MRSW-Protokolles sind keine Quellcodeänderungen eines bestehenden nebenläufigen Programmes nötig. Dieser Vorteil ist bei dem effizienten MRSW-Protokoll nicht mehr gegeben. Bei der Verwendung des MRMW-Protokolles sind Änderungen des Programmes insoweit erforderlich, daß die bereits genannten Beschränkungen des Programmierers beachtet und eingehalten werden. Die Optimierungsmöglichkeiten des effizienten MRSW-Protokolles sind stark äbhängig von den Implementationen der verwendeten Befehle. Der in Kapitel 7.1.5 auf Seite 77 verwendete Befehl *memcpy* ist in der Realität relativ aufwendig realisiert. Zudem muss das betreffende Programm umfangreich analysiert und der Speicheraufbau näher betrachtet werden. Unter Verwendung des MRMW-Protokolles ist das bestehende DSM-System selbstoptimierend und damit wesentlich transparenter.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A comparison of entry consistency and lazy release consistency implementations. Technical Report TX 77005-1892, Dept. of CS, Rice University, 1995.

[2] S. V. Adve and M. D. Hill. Weak ordering—A new definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 2–14, 1990.

[3] C. Amza, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software dsm protocols that adapt between single writer and multiple writer. Technical report, Dept. of CS, Rice University, University of Rochester, 1995.

[4] C. Amza, A. L. Cox, K. Rajamani, and W. Zwaenepoel. Tradeoffs between false sharing and aggregation in software distributed shared memory. Technical report, Dept. of CS, Rice University, 1997.

[5] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, February 1996.

[6] G. R. Andrews. *Concurrent Programming – Principles and Practice*. Benjamin Cummings, 1991.

[7] G. R. Andrews and R. A. Olsson. *The SR Programming Language – Concurrency in Practice*. Benjamin Cummings, 1993.

[8] B. Bershad, M. Zekauskas, and W. Sawdon. The midway distributed shared memory system. In *COMPCON Conference Proceedings*, pages 528–537, 1993.

[9] N. Bershad and M. Zekausas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. TR CMU-CS-91-170, Carnegie-Mellon Univ., 1991.

[10] M. Dubios and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Trans. Software Engineering*, 16(6):660–673, June 1990.

[11] M. Dubios, C. Scheurich, and F. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, February 1988.

[12] K. Gharachorloo, D. Lenosik, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *International Symposium on Computer Architecture*, pages 15–26, 1990.

[13] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *Intl. Conference on Supercomputing*, pages 442–452, July 1988.

[14] P. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Distributed Computing Systems*, pages 302–311, 1990.

[15] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: a bridge between release consistency and entry consistency. In *Symposium on Parallel Algorithms and Architectures*, pages 277–287, June 1996.

[16] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Department of Computer Science, Rice University, December 1994.

[17] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *International Symposium on Computer Architecture*, pages 13–21, 1992.

[18] P. Keleher and C. Tseng. Enhancing software dsm for compiler-parallelized applications. In *International Parallel Processing Symposium*, pages 490–499, 1997.

[19] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. An evaluation of software-based release consistent protocols. *J. Parallel Distrib. Comput.*, 29:126–141, September 1995.

[20] L. Lamport. Time, clocks and ordering of events in distributed systems. *Comm. ACM*, 21(7):558–565, June 1978.

[21] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *ACM Trans. Comput. Systems*, 28:690–691, September 1979.

[22] L. Lamport. Concurrent reading and writing of clocks. In *ACM Trans. Comput. Systems*, pages 8(3):305–310, November 1990.

[23] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. Technical report, Digital Equipment Corporation, Systems Research Center, February 1993.

[24] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Systems*, 7(4):321–359, November 1989.

[25] K. Li and P. R. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings 1986 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, New York, NY, 1986. ACM.

[26] R. Lipton and J. Sandberg. Pram: A scalable shared memory. Technical Report CS-TR-180-88, Princeton Univ., 1988.

[27] F. Mueller. Distributed shared-memory threads: DSM-threads. In *Workshop on Run-Time Systems for Parallel Programming*, pages 31–40, April 1997.

[28] F. Mueller. On the design and implementation of DSM-threads. In *Proc. 1997 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 315–324, June 1997. (invited).

[29] F. Mueller. Prioritized token-based mutual exclusion for distributed systems. In *International Parallel Processing Symposium*, pages 791–795, 1998.

[30] A. S. Tanenbaum. *Distributed Operating System*. Prentice-Hall, 1995.

[31] A. S. Tanenbaum. *Verteilte Betriebssysteme*. Prentice-Hall, 1995.

[32] J.P. Singh, W.D. Weber, A. Gupta. *SPLASH: Stanford Parallel Applications for Shared-Memory*. Computer Architecture News, Vol. 20(1), pages 5-44, March 1992.

[33] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta. *The SPLASH-2 Programs: Characterization and Methodological Considerations*. Proceedings of the 22nd International Symposium on Computer Architecture, pages 24-36, Santa Margherita Ligure, Italy, June 1995.

[34] S.C. Woo, J.P. Singh, J.L. Hennessy. *The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors*. Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, ASP-LOS-VI, pages 219-229, October 1994.

[35] D.H. Bailey. *FFT's in External or Hierarchical Memory*. Journal of Supercomputing, Vol 4(1), pages 23-35, March 1990.

A Der Quellcode *test_release*

```
/****************************************************************/
/*                                                              */
/* Copyright (C) 1996 by Frank Mueller.                         */
/* This code was developed at Humboldt University Berlin.       */
/* Distributed by the author(s) under the following terms:      */
/*                                                              */
/* This code may not be distributed further without permission   */
/*      from Frank Mueller of the Humboldt-University zu Berlin, */
/*      mueller@informatik.hu-berlin.de or (+49) (30) 20181-276. */
/*                                                              */
/****************************************************************/
#include <pthread.h>
#include <dsm.h>
#include "commserv.h"
#define NPROC 90                         /* max==256   */
#define NROWS 90                         /* max==32565 */

DSM_DATA_BEGIN
dsm_thread_t thread[NPROC] = {};
dsm_threadattr_t attr = {};
dsm_barrier_t barrier = {};
int matrix1[NROWS][NROWS] = {};
int matrix2[NROWS][NROWS] = {};
DSM_RELEASE_BEGIN                        /* nur bei MRMW aktiv */
/*DSM_NEWPAGE*/                          /* nur bei MRSW aktiv */
int matrix3[NROWS][NROWS] = {};
DSM_DATA_END

char Name[100] = "";
void FillMatrices( char* name, int rows );
void DumpMatrices( int rows, int one, int two, int three );
void startTime();
void stopTime(long rows, long threads);
struct timespec starttime={}, stoptime={}, dtime={};
```

```
/************************************************************/
void new_thread(void *arg)
{
  unsigned long int r_l = (unsigned long int) arg;
  int rows  = (int) ( r_l >> 16),         /* size of matrices   */
      max   = (int) ((r_l >>  8) & 0xff), /* #threads           */
      id    = (int) ( r_l & 0xff),
      lines = rows / max,                 /* lines per node     */
      rest  = rows % max,
      up, down, i, j, k;

  /* sets up and down line of matrix  */
  if (id<rest) {
    up   = id * lines + id;
    down = up + lines;
  }
  else {
    up   = id * lines + rest;
    down = up + lines - 1;
  }

  /* matrix multiplication */
  if (down>=up)
    for( i=up; i<=down; i++ )
      for( j=0; j<rows; j++ )
        for( matrix3[i][j]=0,k=0; k<rows; k++ )
          matrix3[i][j] += (matrix1[i][k]*matrix2[k][j]);

    {   /* busy wait */
        int k, save=matrix3[up][0];
            for (k=0; k<25000*25000;k++)
                        matrix3[up][0]=0;
        matrix3[up][0] = save;
    }

    dsm_barrier(&barrier, max);
  if (id==0) stopTime(rows, max);
}
```

```
/*************************************************************/
int main(int argc, char **argv)
{
  int rows, max=0, hosts=0, id, params=0;
  unsigned long int arg;

  pthread_init();
  dsm_init(argc, argv);
  dsm_threadattr_init(&attr);
  dsm_barrier_init(&barrier);

  if (argc<=1) {
    fprintf(stderr, "use: %s infile #rows host1 [host2 ...]\n", argv[0]);
    fprintf(stderr, " or: %s infile #rows #threads\n", argv[0]);
    fprintf(stderr, " or: %s        #rows #threads\n", argv[0]);
    fprintf(stderr, " or: %s        #rows\n", argv[0]);
    exit(0);
  }

  /* command line param: filename to save the results  */
  if (!isdigit(argv[1][0]))
    strcpy( Name, argv[++params] ); // ++params == 1

  /* command line param: number of rows for all matrices */
  rows = atoi(argv[++params]);          // ++params == 2 or == 1
  if (rows > NROWS) {
    printf("ERROR: number of rows to high, maximum==%d\n", NROWS);
    exit(-1);
  }
  /* command line param: #threads or hostnames */
  if (argc<=params+1)
    max = rows;
  else
    if (isdigit(argv[++params][0]))
      max = atoi(argv[params]);
    else
      hosts = max = argc-params+1;


  if (max > NPROC) {
```

```
    printf("ERROR: too many DSM threads maximum==%d\n", NPROC);
    exit(-1);
  }

  FillMatrices( Name, rows );
    startTime();

  /* starts threads */
  arg = (rows<<16) + (max<<8);
  for (id=1; id<max; id++) {
    if (hosts)
      dsm_threadattr_setnode(&attr, argv[id+2]);
    dsm_thread_create(&thread[id],&attr,
                      (void *(*)(void*))new_thread,(void*)(arg+id));
  }
  new_thread( (void *)arg );

  dsm_exit(NULL);
}
```

```
/*************************************************************/
/*
 * FillMatrices()
 * fills matrix1 and matrix2 with random values
 * in every loop there are the same random values
 * (without srand())
 *
 * if a filename given, reads matrix1 and matrix2
 * from file
 */
void FillMatrices(char* filename, int rows )
{
  if (strlen(filename)==0) {
      int  i, j, reduce = 3276;
    /* filename not given, fills matrices with random values */
    for( i=0; i<rows; i++ )
      for( j=0; j<rows; j++ ) {
        matrix1[i][j] = rand()%reduce;
        matrix2[i][j] = rand()%reduce;
      }
  } else {
    /* filename given, reads matrices from file */
    FILE* file;
    int  i, j;

    if (!(file = fopen( filename, "r" ))) {
      printf( "FillMatrices(): file open error\n");
      exit ( -1 );
    }
    for( i=0; i<rows; i++ )
      for( j=0; j<rows; j++ )
        fscanf( file, " %6d", &matrix1[i][j] );
    for( i=0; i<rows; i++ )
      for( j=0; j<rows; j++ )
        fscanf( file, " %6d", &matrix2[i][j] );
    fclose( file );
  }
}

void startTime()
```

```
{
  clock_gettime(CLOCK_REALTIME, &starttime);
}

void stopTime(long rows, long threads)
{
  FILE* file;
  int i, j, empty;
  /* allgemeiner Test, ob alles richtig lief */
  for( i=0; i<rows; i++ ) {
    empty = 0;
    for( j=0; j<rows; j++ )
        if (matrix3[i][j]==0)
            empty++;
    if (empty==rows) {
        printf("FEHLER: Eine Zeile der Zielmatrix == 0 (Komm-fehler)");
        return;
    }
  }
  clock_gettime(CLOCK_REALTIME, &stoptime);
  if ((starttime).ts_nsec > (stoptime).ts_nsec) {     /* difference */
    (dtime).ts_sec = (stoptime).ts_sec - (starttime).ts_sec - 1;
    (dtime).ts_nsec = ((stoptime).ts_nsec-(starttime).ts_nsec)+1000000000;
  }
  else {
    (dtime).ts_sec = (stoptime).ts_sec - (starttime).ts_sec;
    (dtime).ts_nsec = (stoptime).ts_nsec - (starttime).ts_nsec;
  }
  /* Ergebnis speichern */
  if (file=fopen("time_protokoll.txt","a")) {
    fprintf(file,"rows:%6d threads:%6d sec=%u nsec=%u\n",
                  rows, threads, dtime.ts_sec, dtime.ts_nsec );
    fclose(file);
  } else
    printf("\n\nKann File 'time_protokoll.txt' nicht ffnen!\n\n");
}
```

B Der Quellcode *FFT*

```
/*************************************************************************/
/*                                                                       */
/*  Copyright (c) 1994 Stanford University                               */
/*                                                                       */
/*  All rights reserved.                                                 */
/*                                                                       */
/*  Permission is given to use, copy, and modify this software for any   */
/*  non-commercial purpose as long as this copyright notice is not       */
/*  removed.  All other uses, including redistribution in whole or in    */
/*  part, are forbidden without prior written permission.                */
/*                                                                       */
/*  This software is provided with absolutely no warranty and no         */
/*  support.                                                             */
/*                                                                       */
/*************************************************************************/

/*************************************************************************/
/*                                                                       */
/*  Perform 1D fast Fourier transform using six-step FFT method          */
/*                                                                       */
/*  1) Performs staggered, blocked transposes for cache-line reuse       */
/*  2) Roots of unity rearranged and distributed for only local          */
/*     accesses during application of roots of unity                     */
/*  3) Small set of roots of unity elements replicated locally for       */
/*     1D FFTs (less than root N elements replicated at each node)       */
/*  4) Matrix data structures are padded to reduce cache mapping         */
/*     conflicts                                                         */
/*                                                                       */
/*  Command line options:                                                */
/*                                                                       */
/*  -mM : M = even integer; 2**M total complex data points transformed.  */
/*  -pP : P = number of processors; Must be a power of 2.                */
/*  -nN : N = number of cache lines.                                     */
/*  -lL : L = Log base 2 of cache line length in bytes.                  */
/*  -s  : Print individual processor timing statistics.                  */
/*  -t  : Perform FFT and inverse FFT.  Test output by comparing the     */
/*        integral of the original data to the integral of the data      */
/*        that results from performing the FFT and inverse FFT.          */
```

```
/*  -o  : Print out complex data points.                                 */
/*  -h  : Print out command line options.                                */
/*                                                                       */
/*  Note: This version works under both the FORK and SPROC models        */
/*                                                                       */
/*************************************************************************/

#include <stdio.h>
#include <math.h>

/*#define PAGE_SIZE               4096*/
#define NUM_CACHE_LINES        65536
#define LOG2_LINE_SIZE             4
#define PI                         3.1416
#define DEFAULT_M                 10
#define DEFAULT_P                  1

#define SWAP(a,b) {double tmp; tmp=a; a=b; b=tmp;}

struct GlobalMemory {
  int id;
  LOCKDEC(idlock)
  BARDEC(start)
  int *transtimes;
  int *totaltimes;
  int starttime;
  int finishtime;
  int initdonetime;
} *Global = NULL;


int P = DEFAULT_P;
int M = DEFAULT_M;
int N = 0;                  /* N = 2^M                                */
int rootN = 0;              /* rootN = N^1/2                          */
double *trans = NULL;       /* trans is used as scratch space         */
double *umain = NULL;       /* umain is roots of unity for 1D FFTs    */
double *umain2 = NULL;      /* umain2 is entire roots of unity matrix */
int test_result = 0;
int doprint = 0;
```

```
int dostats = 0;
int transtime = 0;
int transtime2 = 0;
int avgtranstime = 0;
int avgcomptime = 0;
unsigned int transstart = 0;
unsigned int transend = 0;
int maxtotal=0;
int mintotal=0;
double maxfrac=0;
double minfrac=0;
double avgfractime=0;
int orig_num_lines = NUM_CACHE_LINES;     /* number of cache lines */
int num_cache_lines = NUM_CACHE_LINES;    /* number of cache lines */
int log2_line_size = LOG2_LINE_SIZE;
int line_size = 0;
int rowsperproc = 0;
double ck1 = 0.0;
double ck3 = 0.0;                       /* checksums for testing answer */
int pad_length = 0;

DSM_RELEASE_BEGIN
double *x = NULL;           /* x is the original time-domain data     */


void SlaveStart();
double TouchArray(double *,double *,double *,double *,int,int,int,int);
void FFT1D(int,int,int,double *,double *,double *,double *,int,int *,int,
       int,int,int,int,int,int,struct GlobalMemory *);
double CheckSum();
double drand48();
int log_2(int);
void printerr(char *);



#define PROF
```

```
main(argc, argv)

int argc;
char *argv[];

{
  int i;
  int j;
  int c;
  extern char *optarg;
  int m1;
  int factor;
  int pages;
  unsigned int start;

#ifdef PROF
printf("Global %x\n", &Global);
#endif
  CLOCK(start);

  while ((c = getopt(argc, argv, "p:m:n:l:stoh")) != -1) {
    switch(c) {
      case 'p': P = atoi(optarg);
                if (P < 1) {
                  printerr("P must be >= 1\n");
                  exit(-1);
                }
                if (log_2(P) == -1) {
                  printerr("P must be a power of 2\n");
                  exit(-1);
                }
            break;
      case 'm': M = atoi(optarg);
                m1 = M/2;
                if (2*m1 != M) {
                  printerr("M must be even\n");
                  exit(-1);
                }
            break;
      case 'n': num_cache_lines = atoi(optarg);
```

```
                orig_num_lines = num_cache_lines;
                if (num_cache_lines < 1) {
                  printerr("Number of cache lines must be >= 1\n");
                  exit(-1);
                }
            break;
      case 'l': log2_line_size = atoi(optarg);
                if (log2_line_size < 0) {
                  printerr("Log base 2 of cache line length in bytes must be >= 0\n");
                  exit(-1);
                }
            break;
      case 's': dostats = !dostats;
            break;
      case 't': test_result = !test_result;
            break;
      case 'o': doprint = !doprint;
            break;
      case 'h': printf("Usage: FFT <options>\n\n");
                printf("options:\n");
                printf("  -mM : M = even integer; 2**M total complex data points transformed.\n");
                printf("  -pP : P = number of processors; Must be a power of 2.\n");
                printf("  -nN : N = number of cache lines.\n");
                printf("  -lL : L = Log base 2 of cache line length in bytes.\n");
                printf("  -s  : Print individual processor timing statistics.\n");
                printf("  -t  : Perform FFT and inverse FFT.  Test output by comparing the\n");
                printf("        integral of the original data to the integral of the data that\n");
                printf("        results from performing the FFT and inverse FFT.\n");
                printf("  -o  : Print out complex data points.\n");
                printf("  -h  : Print out command line options.\n\n");
                printf("Default: FFT -m%1d -p%1d -n%1d -l%1d\n",
                       DEFAULT_M,DEFAULT_P,NUM_CACHE_LINES,LOG2_LINE_SIZE);
        exit(0);
            break;
    }
  }

#ifndef PROF
printf("Global %x\n", &Global);
#endif
```

```
N = 1<<M;
rootN = 1<<(M/2);
rowsperproc = rootN/P;
if (rowsperproc == 0) {
  printerr("Matrix not large enough. 2**(M/2) must be >= P\n");
  exit(-1);
}

line_size = 1 << log2_line_size;
if (line_size < 2*sizeof(double)) {
  printf("WARNING: Each element is a complex double (%d bytes)\n",2*sizeof(double));
  printf("  => Less than one element per cache line\n");
  printf("     Computing transpose blocking factor\n");
  factor = (2*sizeof(double)) / line_size;
  num_cache_lines = orig_num_lines / factor;
}
if (line_size <= 2*sizeof(double)) {
  pad_length = 1;
} else {
  pad_length = line_size / (2*sizeof(double));
}

if (rowsperproc * rootN * 2 * sizeof(double) >= PAGE_SIZE) {
  pages = (2 * pad_length * sizeof(double) * rowsperproc) / PAGE_SIZE;
  if (pages * PAGE_SIZE != 2 * pad_length * sizeof(double) * rowsperproc) {
    pages ++;
  }
  pad_length = (pages * PAGE_SIZE) / (2 * sizeof(double) * rowsperproc);
} else {
  pad_length = (PAGE_SIZE - (rowsperproc * rootN * 2 * sizeof(double))) /

               (2 * sizeof(double) * rowsperproc);
  if (pad_length * (2 * sizeof(double) * rowsperproc) !=
      (PAGE_SIZE - (rowsperproc * rootN * 2 * sizeof(double)))) {
    printerr("Padding algorithm unsuccessful\n");
    exit(-1);
  }
}
```

```
  Global = (struct GlobalMemory *) DSM_MALLOC(sizeof(struct GlobalMemory));
  x = (double *) DSM_MALLOC(2*(N+rootN*pad_length)*sizeof(double)+PAGE_SIZE);
  trans = (double *)DSM_MALLOC(2*(N+rootN*pad_length)*sizeof(double)+PAGE_SIZE);
  umain = (double *) DSM_MALLOC(2*rootN*sizeof(double));
  umain2 =(double *)DSM_MALLOC(2*(N+rootN*pad_length)*sizeof(double)+PAGE_SIZE);

  Global->transtimes = (int *) DSM_MALLOC(P*sizeof(int));
  Global->totaltimes = (int *) DSM_MALLOC(P*sizeof(int));

  if (Global == NULL) {
    printerr("Could not malloc memory for Global\n");
    exit(-1);
  } else if (x == NULL) {
    printerr("Could not malloc memory for x\n");
    exit(-1);
  } else if (trans == NULL) {
    printerr("Could not malloc memory for trans\n");
    exit(-1);
  } else if (umain == NULL) {
    printerr("Could not malloc memory for umain\n");
    exit(-1);
  } else if (umain2 == NULL) {
    printerr("Could not malloc memory for umain2\n");
    exit(-1);
  }

  x = (double *) (((unsigned) x) + PAGE_SIZE - ((unsigned) x) % PAGE_SIZE);
  trans = (double *) (((unsigned) trans) + PAGE_SIZE - ((unsigned) trans) % PAGE_SIZE);
  umain2 = (double *) (((unsigned) umain2) + PAGE_SIZE - ((unsigned) umain2) % PAGE_SIZE);

/* In order to optimize data distribution, the data structures x, trans,
   and umain2 have been aligned so that each begins on a page boundary.
   This ensures that the amount of padding calculated by the program is
   such that each processor's partition ends on a page boundary, thus
   ensuring that all data from these structures that are needed by a
   processor can be allocated to its local memory */

/* POSSIBLE ENHANCEMENT:  Here is where one might distribute the x,
   trans, and umain2 data structures across physically distributed
   memories as desired.
```

```
   One way to place data is as follows:

   double *base;
   int i;

   i = ((N/P)+(rootN/P)*pad_length)*2;
   base = &(x[0]);
   for (j=0;j<P;j++) {
    Place all addresses x such that (base <= x < base+i) on node j
    base += i;
   }

   The trans and umain2 data structures can be placed in a similar manner.

   */

  printf("\n");
  printf("FFT with Blocking Transpose\n");
  printf("   %d Complex Doubles\n",N);
  printf("   %d Processors\n",P);
  if (num_cache_lines != orig_num_lines) {
    printf("   %d Cache lines\n",orig_num_lines);
    printf("   %d Cache lines for blocking transpose\n",num_cache_lines);
  } else {
    printf("   %d Cache lines\n",num_cache_lines);
  }
  printf("   %d Byte line size\n",(1 << log2_line_size));
  printf("   %d Bytes per page\n",PAGE_SIZE);
  printf("\n");

  BARINIT(Global->start);
  LOCKINIT(Global->idlock);
  Global->id = 0;
  InitX(N, x);                 /* place random values in x */

  if (test_result) {
    ck1 = CheckSum(N, x);
  }
  if (doprint) {
```

```
    printf("Original data values:\n");
    PrintArray(N, x);
  }

  InitU(N,umain);               /* initialize u arrays*/
  InitU2(N,umain2,rootN);

  /* fire off P processes */
  for (i=1; i<P; i++) {
    CREATE(SlaveStart);
  }
  SlaveStart();

/*
  WAIT_FOR_END(P-1)
*/

  if (doprint) {
    if (test_result) {
      printf("Data values after inverse FFT:\n");
    } else {
      printf("Data values after FFT:\n");
    }
    PrintArray(N, x);
  }

  transtime = Global->transtimes[0];
  printf("\n");
  printf("                 PROCESS STATISTICS\n");
  printf("            Computation      Transpose     Transpose\n");
  printf(" Proc          Time            Time        Fraction\n");
  printf("    0        %10d     %10d      %8.5f\n",
         Global->totaltimes[0],Global->transtimes[0],
         ((double)Global->transtimes[0])/Global->totaltimes[0]);
  if (dostats) {
    transtime2 = Global->transtimes[0];
    avgtranstime = Global->transtimes[0];
    avgcomptime = Global->totaltimes[0];
    maxtotal = Global->totaltimes[0];
    mintotal = Global->totaltimes[0];
```

```
    maxfrac = ((double)Global->transtimes[0])/Global->totaltimes[0];
    minfrac = ((double)Global->transtimes[0])/Global->totaltimes[0];
    avgfractime = ((double)Global->transtimes[0])/Global->totaltimes[0];
    for (i=1;i<P;i++) {
      if (Global->transtimes[i] > transtime) {
        transtime = Global->transtimes[i];
      }
      if (Global->transtimes[i] < transtime2) {
        transtime2 = Global->transtimes[i];
      }
      if (Global->totaltimes[i] > maxtotal) {
        maxtotal = Global->totaltimes[i];
      }
      if (Global->totaltimes[i] < mintotal) {
        mintotal = Global->totaltimes[i];
      }
      if (((double)Global->transtimes[i])/Global->totaltimes[i] > maxfrac) {
        maxfrac = ((double)Global->transtimes[i])/Global->totaltimes[i];
      }
      if (((double)Global->transtimes[i])/Global->totaltimes[i] < minfrac) {
        minfrac = ((double)Global->transtimes[i])/Global->totaltimes[i];
      }
      printf("  %3d        %10d     %10d      %8.5f\n",
             i,Global->totaltimes[i],Global->transtimes[i],
             ((double)Global->transtimes[i])/Global->totaltimes[i]);
      avgtranstime += Global->transtimes[i];
      avgcomptime += Global->totaltimes[i];
      avgfractime += ((double)Global->transtimes[i])/Global->totaltimes[i];
    }
    printf("  Avg        %10.0f     %10.0f      %8.5f\n",
           ((double) avgcomptime)/P,((double) avgtranstime)/P,avgfractime/P);
    printf("  Max        %10d     %10d      %8.5f\n",
       maxtotal,transtime,maxfrac);
    printf("  Min        %10d     %10d      %8.5f\n",
       mintotal,transtime2,minfrac);
  }
  Global->starttime = start;
  printf("\n");
  printf("                 TIMING INFORMATION\n");
  printf("Start time                        : %16d\n",
```

```
    Global->starttime);
printf("Initialization finish time        : %16d\n",
    Global->initdonetime);
printf("Overall finish time               : %16d\n",
    Global->finishtime);
printf("Total time with initialization    : %16d\n",
    Global->finishtime-Global->starttime);
printf("Total time without initialization : %16d\n",
    Global->finishtime-Global->initdonetime);
printf("Overall transpose time            : %16d\n",
       transtime);
printf("Overall transpose fraction        : %16.5f\n",
       ((double) transtime)/(Global->finishtime-Global->initdonetime));
printf("\n");

if (test_result) {
  ck3 = CheckSum(N, x);
  printf("              INVERSE FFT TEST RESULTS\n");
  printf("Checksum difference is %.3f (%.3f, %.3f)\n",
     ck1-ck3, ck1, ck3);
  if (fabs(ck1-ck3) < 0.001) {
    printf("TEST PASSED\n");
  } else {
    printf("TEST FAILED\n");
  }
}

  {
      FILE* f;
      printf ("1\n");fflush(stdout);
      printf ("%s\n",dsm_isDSMRelease(x) ? "REL" : "DSM");fflush(stdout);
      printf ("2\n");fflush(stdout);
      printf ("%16d\n",Global->starttime);fflush(stdout);
      printf ("%16d\n",Global->initdonetime);fflush(stdout);
      printf ("%16d\n",Global->finishtime);fflush(stdout);
      printf ("%16d\n",Global->finishtime-Global->starttime);fflush(stdout);
      printf ("%16d\n",Global->finishtime-Global->initdonetime);fflush(stdout);
      printf ("%16d\n",transtime);fflush(stdout);
      printf ("%%16.5f\n",((double) transtime)/(Global->finishtime-Global->initdonetime));fflush(stdout);
```

```
        if (f = fopen("result.txt","a")) {
            fprintf( f, "%s %16d %16d %16d %16d %16d %16d %16.5f\n",
                dsm_isDSMRelease(x) ? "REL" : "DSM",
                Global->starttime,
                Global->initdonetime,
                Global->finishtime,
                Global->finishtime-Global->starttime,
                Global->finishtime-Global->initdonetime,
                transtime,
                ((double) transtime)/(Global->finishtime-Global->initdonetime) );
            fclose(f);
        }
    }

#ifdef PROF
    printf("exit\n");
#endif
}


void SlaveStart()
{
  int i;
  int j;
  int MyNum;
  double error;
  double *upriv;
  int initdone;
  int finish;
  int l_transtime=0;
  int MyFirst;
  int MyLast;

  LOCK(Global->idlock);
    MyNum = Global->id;
    Global->id++;
  UNLOCK(Global->idlock);

#ifdef PROF
  printf("%d >> start barrier I", MyNum);
```

```
#endif

/* POSSIBLE ENHANCEMENT:  Here is where one might pin processes to
   processors to avoid migration */

  BARRIER(Global->start, P);

#ifdef PROF
  printf("%d << start barrier I", MyNum);
#endif

  upriv = (double *) malloc(2*(rootN-1)*sizeof(double));
  if (upriv == NULL) {
    fprintf(stderr,"Proc %d could not malloc memory for upriv\n",MyNum);
    exit(-1);
  }
  for (i=0;i<2*(rootN-1);i++) {
    upriv[i] = umain[i];
  }

  MyFirst = rootN*MyNum/P;
  MyLast = rootN*(MyNum+1)/P;

  TouchArray(x, trans, umain2, upriv, N, MyNum, MyFirst, MyLast);

#ifdef PROF
  printf("%d >> start barrier II", MyNum);
#endif

  BARRIER(Global->start, P);

#ifdef PROF
  printf("%d << start barrier II", MyNum);
#endif

/* POSSIBLE ENHANCEMENT:  Here is where one might reset the
   statistics that one is measuring about the parallel execution */

  if ((MyNum == 0) || (dostats)) {
    CLOCK(initdone);
```

```
  }

  /* perform forward FFT */
  FFT1D(1, M, N, x, trans, upriv, umain2, MyNum, &l_transtime, MyFirst,
    MyLast, pad_length, P, test_result, doprint, dostats, Global);

  /* perform backward FFT */
  if (test_result) {
    FFT1D(-1, M, N, x, trans, upriv, umain2, MyNum, &l_transtime, MyFirst,
      MyLast, pad_length, P, test_result, doprint, dostats, Global);
  }

  if ((MyNum == 0) || (dostats)) {
    CLOCK(finish);
    Global->transtimes[MyNum] = l_transtime;
    Global->totaltimes[MyNum] = finish-initdone;
  }
  if (MyNum == 0) {
    Global->finishtime = finish;
    Global->initdonetime = initdone;
  }

  SIGNAL_END(MyNum, P-1)
}


double TouchArray(x, scratch, u, upriv, N, MyNum, MyFirst, MyLast)

double *x;
double *scratch;
double *u;
double *upriv;
int N;
int MyNum;
int MyFirst;
int MyLast;

{
  int i,j,k;
  double tot = 0.0;
```

```
  /* touch my data */
  for (j=0;j<2*(rootN-1);j++) {
    tot += upriv[j];
  }
  for (j=MyFirst; j<MyLast; j++) {
    k = j * (rootN + pad_length);
    for (i=0;i<rootN;i++) {
      tot += x[2*(k+i)] + x[2*(k+i)+1] +
             scratch[2*(k+i)] + scratch[2*(k+i)+1] +
         u[2*(k+i)] + u[2*(k+i)+1];
    }
  }
  return tot;
}


double CheckSum(N, x)

int N;
double *x;

{
  int i,j,k;
  double cks;

  cks = 0.0;
  for (j=0; j<rootN; j++) {
    k = j * (rootN + pad_length);
    for (i=0;i<rootN;i++) {
      cks += x[2*(k+i)] + x[2*(k+i)+1];
    }
  }

  return(cks);
}


InitX(N, x)
```

```
int N;
double *x;

{
  int i,j,k;

  srand48(0);
  for (j=0; j<rootN; j++) {
    k = j * (rootN + pad_length);
    for (i=0;i<rootN;i++) {
      x[2*(k+i)] = drand48();
      x[2*(k+i)+1] = drand48();
    }
  }
}


InitU(N, u)

int N;
double *u;

{
  int q;
  int j;
  int base;
  int n1;

  for (q=0; 1<<q<N; q++) {
    n1 = 1<<q;
    base = n1-1;
    for (j=0; j<n1; j++) {
      if (base+j > rootN-1) {
    return;
      }
      u[2*(base+j)] = cos(2.0*PI*j/(2*n1));
      u[2*(base+j)+1] = -sin(2.0*PI*j/(2*n1));
    }
  }
}
```

```
InitU2(N, u, n1)

int N;
double *u;
int n1;

{
  int i,j,k;
  int base;

  for (j=0; j<n1; j++) {
    k = j*(rootN+pad_length);
    for (i=0; i<n1; i++) {
      u[2*(k+i)] = cos(2.0*PI*i*j/(N));
      u[2*(k+i)+1] = -sin(2.0*PI*i*j/(N));
    }
  }
}


BitReverse(M, k)

int M;
int k;

{
  int i;
  int j;
  int tmp;

  j = 0;
  tmp = k;
  for (i=0; i<M; i++) {
    j = 2*j + (tmp&0x1);
    tmp = tmp>>1;
  }
  return(j);
}
```

```
void FFT1D(direction, M, N, x, scratch, upriv, umain2, MyNum, l_transtime,
           MyFirst, MyLast, pad_length, P, test_result, doprint, dostats,
       Global)

int direction;
int M;
int N;
int *l_transtime;
double *x;
double *upriv;
double *scratch;
double *umain2;
int MyFirst;
int MyLast;
int pad_length;
int P;
int test_result;
int doprint;
int dostats;
struct GlobalMemory *Global;

{
  int i;
  int j;
  int m1;
  int n1;
  int flag = 0;
  unsigned int clocktime1;
  unsigned int clocktime2;

  m1 = M/2;
  n1 = 1<<m1;

  BARRIER(Global->start, P);

  if ((MyNum == 0) || (dostats)) {
    CLOCK(clocktime1);
  }
```

```
  /* transpose from x into scratch */

#ifdef PROF
  printf("%d >> Transpose I", MyNum);
#endif

  Transpose(n1, x, scratch, MyNum, MyFirst, MyLast, pad_length);

#ifdef PROF
  printf("%d << Transpose I", MyNum);
#endif

  if ((MyNum == 0) || (dostats)) {
    CLOCK(clocktime2);
    *l_transtime += (clocktime2-clocktime1);
  }

  /* do n1 1D FFTs on columns */
  for (j=MyFirst; j<MyLast; j++) {
#ifdef PROF
      printf("%d >> FFT1DOnce on column %d", MyNum,j);
#endif

    FFT1DOnce(direction, m1, n1, upriv, &scratch[2*j*(n1+pad_length)]);

#ifdef PROF
      printf("%d << FFT1DOnce on column %d AND >> Twiddle ...", MyNum,j);
#endif

    TwiddleOneCol(direction, n1, N, j, umain2, &scratch[2*j*(n1+pad_length)],
          pad_length);

#ifdef PROF
    printf("%d << TwiddleOneCol on column %d", MyNum, j);
#endif
  }

  BARRIER(Global->start, P);
```

```
  if ((MyNum == 0) || (dostats)) {
    CLOCK(clocktime1);
  }

#ifdef PROF
  printf("%d >> Transpose II", MyNum);
#endif

  /* transpose */
  Transpose(n1, scratch, x, MyNum, MyFirst, MyLast, pad_length);

#ifdef PROF
  printf("%d << Transpose II", MyNum);
#endif

  if ((MyNum == 0) || (dostats)) {
    CLOCK(clocktime2);
    *l_transtime += (clocktime2-clocktime1);
  }

#ifdef PROF
  printf("%d >> 2. loop with FFT1DOnce", MyNum);
#endif

  /* do n1 1D FFTs on columns again */
  for (j=MyFirst; j<MyLast; j++) {
    FFT1DOnce(direction, m1, n1, upriv, &x[2*j*(n1+pad_length)]);
    if (direction == -1)
      Scale(n1, N, &x[2*j*(n1+pad_length)]);
  }

#ifdef PROF
  printf("%d << 2. loop with FFT1DOnce", MyNum);
#endif

  BARRIER(Global->start, P);

  if ((MyNum == 0) || (dostats)) {
    CLOCK(clocktime1);
  }
```

```
  /* transpose back */
  Transpose(n1, x, scratch, MyNum, MyFirst, MyLast, pad_length);

  if ((MyNum == 0) || (dostats)) {
    CLOCK(clocktime2);
    *l_transtime += (clocktime2-clocktime1);
  }

  BARRIER(Global->start, P);

#ifdef PROF
  printf("%d >> copy to X", MyNum);
#endif

  /* copy columns from scratch to x */
  if ((test_result) || (doprint)) {
    for (j=MyFirst; j<MyLast; j++) {
      CopyColumn(n1, &scratch[2*j*(n1+pad_length)], &x[2*j*(n1+pad_length)]);
    }
  }

#ifdef PROF
  printf("%d << copy to X", MyNum);
#endif

  BARRIER(Global->start, P);
}


TwiddleOneCol(direction, n1, N, j, u, x, pad_length)

int direction;
int n1;
int N;
int j;
double *u;
double *x;
int pad_length;
```

```
{
  int i;
  double omega_r;
  double omega_c;
  double x_r;
  double x_c;
  double r1;
  double c1;
  double r2;
  double c2;

  for (i=0; i<n1; i++) {
    omega_r = u[2*(j*(n1+pad_length)+i)];
    omega_c = direction*u[2*(j*(n1+pad_length)+i)+1];
    x_r = x[2*i];
    x_c = x[2*i+1];
    x[2*i] = omega_r*x_r - omega_c*x_c;
    x[2*i+1] = omega_r*x_c + omega_c*x_r;
  }
}


Scale(n1, N, x)

int n1;
int N;
double *x;

{
  int i;

  for (i=0; i<n1; i++) {
    x[2*i] /= N;
    x[2*i+1] /= N;
  }
}


Transpose(n1, src, dest, MyNum, MyFirst, MyLast, pad_length)
```

```
int n1;
double *src;
double *dest;
int MyNum;
int MyFirst;
int MyLast;
int pad_length;

{
  int i;
  int j;
  int k;
  int l;
  int m;
  int blksize;
  int numblks;
  int firstfirst;
  int h_off;
  int v_off;
  int v;
  int h;
  int n1p;
  int row_count;

  blksize = MyLast-MyFirst;
  numblks = (2*blksize)/num_cache_lines;
  if (numblks * num_cache_lines != 2 * blksize) {
    numblks ++;
  }
  blksize = blksize / numblks;
  firstfirst = MyFirst;
  row_count = n1/P;
  n1p = n1+pad_length;
  for (l=MyNum+1;l<P;l++) {
    v_off = l*row_count;
    for (k=0; k<numblks; k++) {
      h_off = firstfirst;
      for (m=0; m<numblks; m++) {
        for (i=0; i<blksize; i++) {
      v = v_off + i;
```

```
        for (j=0; j<blksize; j++) {
      h = h_off + j;
          dest[2*(h*n1p+v)] = src[2*(v*n1p+h)];
          dest[2*(h*n1p+v)+1] = src[2*(v*n1p+h)+1];
        }
      }
  h_off += blksize;
    }
    v_off+=blksize;
  }
}

for (l=0;l<MyNum;l++) {
  v_off = l*row_count;
  for (k=0; k<numblks; k++) {
    h_off = firstfirst;
    for (m=0; m<numblks; m++) {
      for (i=0; i<blksize; i++) {
    v = v_off + i;
        for (j=0; j<blksize; j++) {
          h = h_off + j;
          dest[2*(h*n1p+v)] = src[2*(v*n1p+h)];
          dest[2*(h*n1p+v)+1] = src[2*(v*n1p+h)+1];
        }
      }
  h_off += blksize;
    }
    v_off+=blksize;
  }
}

v_off = MyNum*row_count;
for (k=0; k<numblks; k++) {
  h_off = firstfirst;
  for (m=0; m<numblks; m++) {
    for (i=0; i<blksize; i++) {
      v = v_off + i;
      for (j=0; j<blksize; j++) {
        h = h_off + j;
        dest[2*(h*n1p+v)] = src[2*(v*n1p+h)];
```

```
          dest[2*(h*n1p+v)+1] = src[2*(v*n1p+h)+1];
    }
      }
      h_off += blksize;
    }
    v_off+=blksize;
  }
}


CopyColumn(n1, src, dest)

int n1;
double *src;
double *dest;

{
  int i;

  for (i=0; i<n1; i++) {
    dest[2*i] = src[2*i];
    dest[2*i+1] = src[2*i+1];
  }
}


Reverse(N, M, x)

int N;
int M;
double *x;

{
  int j, k;

  for (k=0; k<N; k++) {
    j = BitReverse(M, k);
    if (j > k) {
      SWAP(x[2*j], x[2*k]);
      SWAP(x[2*j+1], x[2*k+1]);
```

```
    }
  }
}


FFT1DOnce(direction, M, N, u, x)

int direction;
int M;
int N;
double *u;
double *x;

{
  int j;
  int k;
  int q;
  int L;
  int r;
  int Lstar;
  double *u1;
  double *x1;
  double *x2;
  double omega_r;
  double omega_c;
  double tau_r;
  double tau_c;
  double x_r;
  double x_c;

  Reverse(N, M, x);

  for (q=1; q<=M; q++) {
    L = 1<<q; r = N/L; Lstar = L/2;
    u1 = &u[2*(Lstar-1)];
    for (k=0; k<r; k++) {
      x1 = &x[2*(k*L)];
      x2 = &x[2*(k*L+Lstar)];
      for (j=0; j<Lstar; j++) {
    omega_r = u1[2*j];
```

```
        omega_c = direction*u1[2*j+1];
    x_r = x2[2*j];
        x_c = x2[2*j+1];
    tau_r = omega_r*x_r - omega_c*x_c;
    tau_c = omega_r*x_c + omega_c*x_r;
    x_r = x1[2*j];
        x_c = x1[2*j+1];
    x2[2*j] = x_r - tau_r;
    x2[2*j+1] = x_c - tau_c;
    x1[2*j] = x_r + tau_r;
    x1[2*j+1] = x_c + tau_c;
      }
    }
  }
}


PrintArray(N, x)

int N;
double *x;

{
  int i, j, k;

  for (i=0; i<rootN; i++) {
    k = i*(rootN+pad_length);
    for (j=0; j<rootN; j++) {
      printf(" %4.2f %4.2f", x[2*(k+j)], x[2*(k+j)+1]);
      if (i*rootN+j != N-1) {
        printf(",");
      }
      if ((i*rootN+j+1) % 8 == 0) {
        printf("\n");
      }
    }
  }
  printf("\n");
  printf("\n");
}
```

```
void printerr(s)

char *s;

{
  fprintf(stderr,"ERROR: %s\n",s);
}


int log_2(number)

int number;

{
  int cumulative = 1;
  int out = 0;
  int done = 0;

  while ((cumulative < number) && (!done) && (out < 50)) {
    if (cumulative == number) {
      done = 1;
    } else {
      cumulative = cumulative * 2;
      out ++;
    }
  }

  if (cumulative == number) {
    return(out);
  } else {
    return(-1);
  }
}
```