Hauptseminar
Multicore-Programmierung
SS 2009

MPI und OpenMP auf Multicoresysteme

Cheng Xu
28. Mai 2009

Inhaltsverzeichnis

1. Optimierte MPI kollektive Kommunikationen für Multicore-Systeme

1.1 Einführung

Message Passing Interface(MPI) ist ein sprachenunabhängiges Kommunikationsprotokoll, welches einen Standard für das Schreiben paralleler Programme darstellt. MPI unterstützt Punkt-zu-Punkt und kollektive Kommunikation. Bei Punkt-zu-Punkt Kommunikation (z.B.: MPI_Send, MPI_Recv usw.) sendet ein Prozess eine Nachricht an genau einen Prozess innerhalb eines Kommunikators. Kollektive Kommunikationsoperationen (z.B.: MPI_Bcast, MPI_Gather usw.) sind Kommunikationen, an denen alle Prozesse aus der Gruppe des verwendeten Kommunikators beteiligt sind.

1.2 Motivation

Kollektive Kommunikation ist eine sehr wichtige und häufig genutzte Komponente in MPI. Eine fünfjährige Studie an der Universität Stuttgart hat gezeigt, dass über 40% der Ausführungszeit von Routinen des MPI auf die kollektiven Kommunikationsroutinen MPI_Allreduce und MPI_Reduce entfallen[1]. Eine Optimierung dieser und generell aller kollektiven Operationen wäre im Bezug auf die Performance von parallelen Programmen, die auf MPI aufbauen, von sehr großem Nutzen.

1.3 Algorithmen

Als in HPC(High-performance computing) Systemen sowohl die Anzahl der Knoten als auch die Anzahl der Cores pro Knoten rapide anstiegen, so beschränkt die Skalierbarkeit der kollektiven Kommunikationen viele Applikationen. Die Multi-Core Knoten bieten eine Möglichkeit zur Erhöhung der Skalierbarkeit Kollektiver Kommunikationen. Durch die Implementierung von Algorithmen, die den Memory-Traffic reduzieren sollen, kann die Skalierbarkeit erweitert werden. Memory-Traffic, Cache-Konflikt und Synchronisation begrenzen die Skalierbarkeit und die Performance von *shared-memory* kollektiven Kommunikationen. Das Ziel der

Algorithmen ist die Reduzierung des Datenverkehrs durch die Beschränkung der Zugriffe auf den Speicher und die Balance zwischen Synchronisation und Speicherzugriffskosten zu halten.
Das Verfahren beschreibt die Zuweisung bestimmter Größe von Segmenten in gemeinsam genutzten Speicher für jeden MPI Kommunikator. Die Segmente sind benachbart in virtuellen Speichern. Diese werden beim Starten des Kommunikators zugewiesen und wieder durch Vernichtung des Kommunikators freigegeben. Diese Speicher-Segmente werden in zwei Teile aufgeteilt, Kontrollbereich (control region) und Datenbereich (data region). Im Kommunikator hat jeder Prozess einen Kontrollbereich und einen Datenbereich. Der Kontrollbereich besitzt ein Flag, welches von den Algorithmen benutzt wird um andere Prozesse zu signalisieren. Im Datenbereich schreiben die Prozesse die Daten. Ein Prozess kann Daten aus den Kontroll- und Datenbereich anderer Prozessen auslesen. Jeder Prozess darf aber nur in seinen eigenen Kontrollbereich und Datenbereich schreiben.

1.3.1 Optimierte MPI_Allreduce

Im genormten Algorithmus für Allreduce führt nur ein Prozessorkern die Reduktion aus. Andere Prozessorkerne stehen im Leerlauf. Um die Rechenleistungen für Mehrprozessorsysteme auszunutzen werden optimierte Algorithmen gebraucht. Die Idee der Algorithmen ist die Zerlegung und Ausführung in mehreren Teilen, damit jeder Prozessorkern sich an der Aufgabe beteiligen kann.

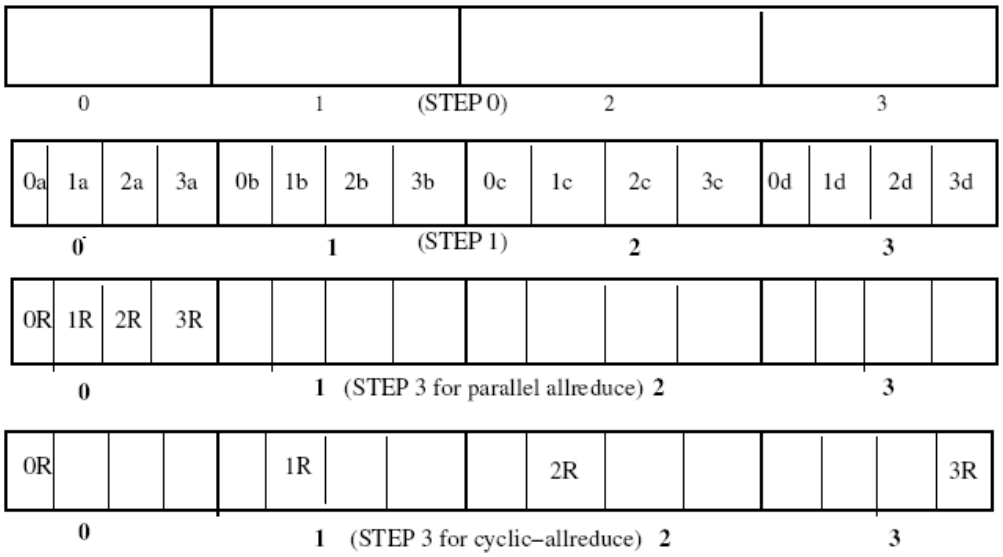

Abbildung 1. Opertimierter shared memory Allreduce

Wir stellen hier einen optimierten Algorithmus mit einem Beispiel (Abbildung 1) vor. Im Beispiel ist eine Reduktion zwischen vier Prozessen zu unternehmen. Am Anfang der Induktion kopiert jeder Prozess seine Daten in unterschiedliche Blöcke der gemeinsam genutzten Speicher (Step 0 in der Abbildung). Dann wird jeder Block in vier Unterblöcke aufgeteilt (Step 1 in der Abbildung). Jeder Prozess arbeitet an einem

Unterblock eines jeden Blocks. Zum Beispiel reduziert Prozess 0 Unterblöcke 0a, 0b, 0c und 0d. Das Ergebnis wird danach in den Unterblock 0R speichert. Prozess 1 reduziert Unterblöcke 1a, 1b, 1c, 1d und speichert das Ergebnis in 1R. Im dritten Schritt kopieren alle Prozesse die Ergebnisse in ihren lokalen Speicher.
Das kann aber problematisch sein, wenn mehrere Prozesse gleichzeitig auf einen Block zugreifen. Deswegen haben wir einen anderen Algorithmus entwickelt: *Cyclic-allreduce*. Der 0. und 1. Schritt vom *Cyclic-allreduce* Algorithmus sind gleich mit dem vorherigen Algorithmus. Die Prozesse greifen auf die Blöcke aber in einer anderen Reihenfolge zu. Prozess 0 reduziert 0a, 0b, 0c und 0d. Prozess 1 reduziert 1b, 1c, 1d, 1a usw. Im dritten Schritt kopiert Prozess 0 die Ergebnisse in den lokalen Speicher in Reihenfolge 0R, 1R, 2R und 3R. Prozess 1 kopiert die Ergebnisse in Reihenfolge 1R, 2R, 3R, 0R usw.

1.3.2 Optimierter MPI-Bcast

Der genormte Algorithmus für MPI_Bcast verwendet eine *scatter* Operation gefolgt von einem *all-to-all* Broadcast oder MPI_Allgather. Dieser Algorithmus ist gut geeignet für Clusters, in denen ein Knoten nur einen Prozess hat. Für die Knoten, auf denen mehrere Prozesse laufen, wird ein neuer Algorithmus gebraucht. Weil die Prozesse auf einem Knoten einen gemeinsamen Speicher haben, entstehen unnötige Kommunikationen mit dem alten Algorithmus.

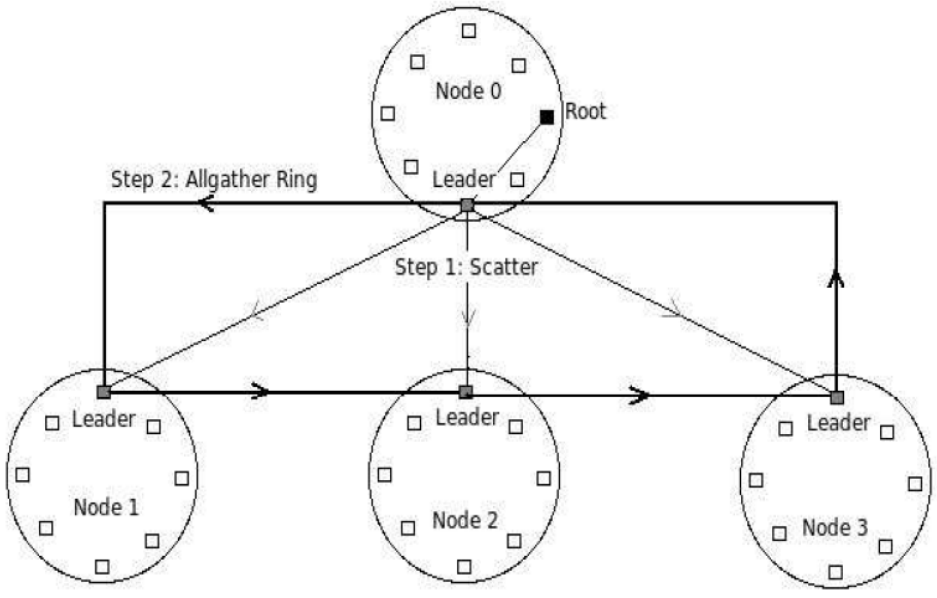

Abbildung 2. Optimierter Broadcast Algorithmus

Im neuen Algorithmus wird bei jedem Konten ein Prozess als ein *Leader* des Knotens ausgewählt. Im ersten Schritt des Algorithmus kopiert der *root* Prozess die Daten in den gemeinsam genutzten Buffer. Der *Leader* sendet die Daten an alle anderen *Leader* mit einer Scatter-Operation. Im zweiten Schritt wird die Ring-Topologie verwendet.

Ein *Leader* sendet die Nachrichten an den nächsten *Leader* mit MPI_Allgather. Es wird n-1 (n: Anzahl der Leader) Schritt gebraucht, bis alle *Leader* die Daten vollständig erhalten haben. Im letzten Schritt werden die Daten von *Network Buffer* in den *User Buffer* kopiert. Dann können alle Prozesse des Knotens auf die Daten zugreifen. Die Kommunikationen zwischen der Prozessen eines gleichen Knoten wurden gespart.

2. OpenMP auf Cell-Prozessor

2.1 Was ist Cell

In den vergangenen drei Jahren haben die Prozessor-Hersteller zahlreiche Multicore-Chips entwickelt um eine bessere Performance zu erzielen. Als eine der heterogenen Architekturen integriert der Cell-Prozessor neun Prozessorkerne, lokale Speicher und Kommunikationshardware in einen Chip. Der Cell ist für rechenintensive und breitbandige Anwendungen, wie Multimedia, Computerspiele, Videos und andere Formen digitaler Inhalte optimiert. Zum Beispiel wird der Cell in Sonys Spielekonsole Playstation 3 als Hauptprozessor verwendet. Der Cell-Prozessor basiert auf acht einzelnen synergistic Processor Elements (SPEs), die extrem umfangreiche Fließkomma-Berechnungen verarbeiten können und über einen 64-Bit Power Processor Element(PPE) gesteuert werden. Dem PPE stehen 512 KB L2-Cache zur Verfügung und von Betriebssystemen wie zum Beispiel Linux unterstützt werden. Jeder SPE besteht aus einer Synergistic Processing Unit(SPU), einem lokalen Speicher von 256KB und ein *Memory Flow Controller* (MFC), der DMA -Übertragungen[2] zum Hauptspeicher oder zu anderen SPEs steuert. Die neun Prozessorkerne sind über einen *Element Interface Bus* (EIB) gekoppelt. Der Zugriff auf den Hauptspeicher erfolgt über einen *Memory Interface Controller* (MIC).

2.2 Probleme

2.2.1 Threads und Synchronisation

Threads in PPE unterscheiden sich in Leistungsfähigkeiten von den Threads in SPEs. Das System soll so entworfen werden, damit alle Threads effizient synchronisiert

werden können.
In Cell-System erzeugen PPE und SPEs Threads. Der Master-Thread wird aber immer in PPE ausgeführt. Er ist zuständig für die Erzeugung der Threads, Zuweisung und Ausführungsplanung der Aufgaben und Initialisierung der Synchronisations-operationen. Da die SPEs kein Betriebssystem unterstützen, behandelt der Master-Thread auch alle Anfragen an die Service des Betriebssystems. Der PPE wird so als ein Manager-Prozessor entworfen, der die Aktivität der SPE-Prozessoren verwaltet. Das vereinfacht auch den Code für SPEs, die in den kleinen Lokalspeicher SPEs gespeichert werden muss.
Die Erzeugung und Synchronisation der Threads werden mit *Cell Software Development Kit*(SDK) [3] implementiert. Der PPE Master-Thread erzeugt SPE-Thread nur wenn in der Ausführungszeit eine parallele Struktur auftritt. Wenn ein SPE-Thread gestartet und initialisiert wurde, wartet er auf die Aufgabenzuweisung des PPEs. Der SPE-Thread führt die Aufgaben aus und wartet auf weitere Aufgaben bis alle Aufgeben beendet werden. Im Systemspeicher gibt es eine entsprechende Warteschlange von Aufgaben (*task queue*) für jeden Thread. Wenn der Master-Thread einem Thread eine Aufgabe zuweist, schreibt er die Informationen der Aufgabe in die entsprechende Warteschlange. Sobald der SPE-Thread eine Aufgabe aus der Warteschlange geholt hat, ändert er den Status der Aufgabe in der Warteschlange, damit weiß der Master-Thread dass der Platz in der Warteschlange wieder verwendbar ist.
Der Cell-Prozessor bietet einen Mechanismus für effiziente Kommunikation und Synchronisation zwischen der Cores. Jeder SPE-Prozessor beinhaltet einen *Memory Flow Controller*(MFC). Dieser MFC hat zwei blockierte *Outbound Mailbox Queues* und eine nicht blockierte *Inbound Mailbox Queue*. Mit den Mailboxen können die Nachrichten aus 32-bit Werten zwischen der Cores effizient ausgetauscht werden. Wenn der Master-Thread einem SPE-Thread Aufgaben zuweist, teilt er dem SPE-Thread die Anzahl der Aufgaben durch die Mailbox mit.

2.2.2 Codegenerierung

Da die Codes für PPE und SPE unterschiedlich sind, müssen wir noch den PPE-Code vom SPE-Code trennen. Diese Aufgabe übernimmt der Cell OpenMP Compiler.

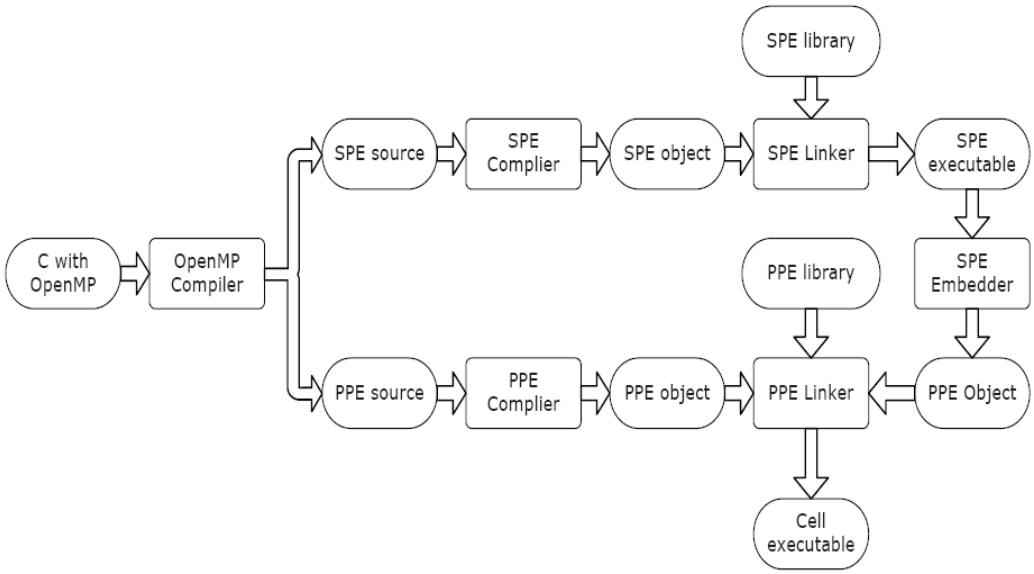

Abbildung 3. OpenMP Compiler Framework für Cell

Die Abbildung 3 zeigt das OpenMP Compiler Framework auf dem Cell an. Der source-to-source OpenMP-Compiler generiert aus den sequenziellen C-Programmen mit OpenMP zwei Sourcecodes. Einer davon ist der PPE Sourcecode. Der andere ist Sourcecode für SPE. Sie werden dann von PPE-Compiler bzw. SPE-Compiler kompiliert. PPE-Compiler bzw. SPE-Compiler generieren aus dem Sourcecode ein PPE-Objekt bzw. SPE-Objekt. Das PPE-Objekt wird dann mit PPE-Bibliothek von PPE Linker gekoppelt, damit steht der ausführbare Code für PPE zur Verfügung. Im Vergleich zum PPE-Objekt wird SPE-Objekt mit SPE-Bibliothek gekoppelt und zur Ausführbarkeit gebracht.

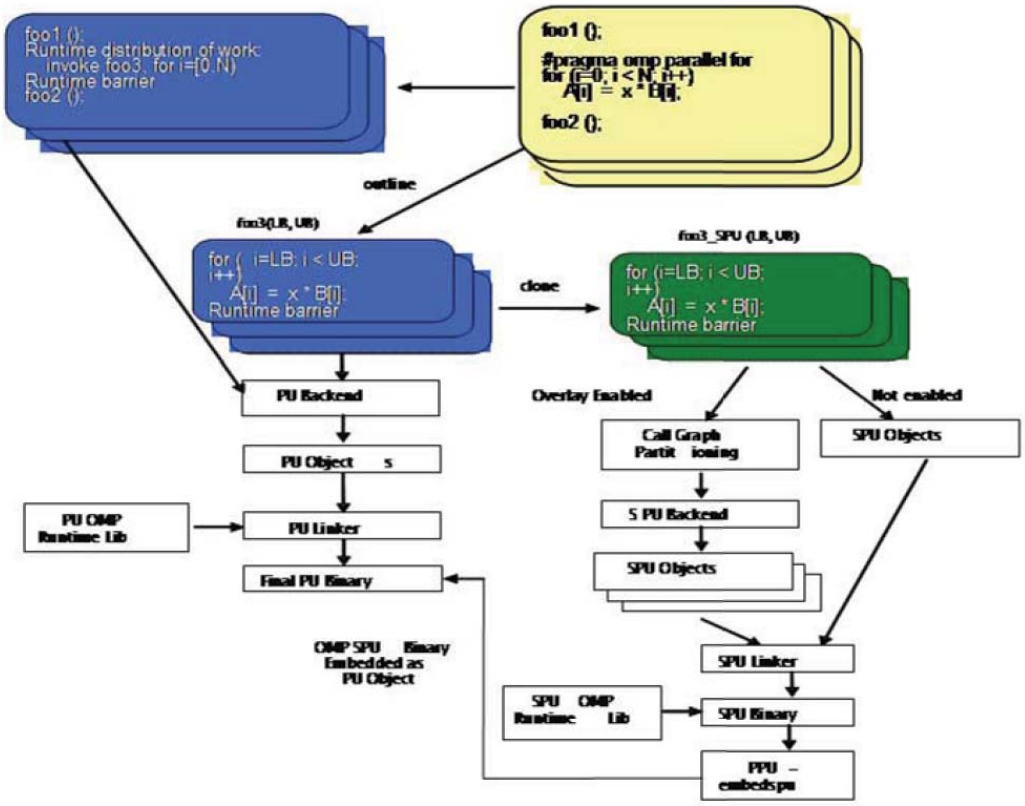

Abbildung 4. Codegenerierung

In der Abbildung 4 wird ein Beispiel gegeben, wie die Codegenerierung verläuft. Jedes Code-Statement, das eine OpenMP parallele Konstrukt entspricht, wird von der Source-Code getrennt. Jedes parallele Konstrukt wird in eine neue Funktion umgewandelt. Die Funktion kann die Grenzwerte der Schleife als Parameter verwenden. Im Originalcode fügt der Compiler einen OpenMP runtime library Aufruf zu der neuen Funktion hinzu. Während der Ausführungszeit wird die runtime Funktion diese neue Funktion indirekt aufrufen. Wenn es nötig ist, fügt der Compiler auch Synchronisationsoperationen wie Barrier in die runtime Funktion ein.
Weil PPE und SPE beide Aufgaben parallele ausführen kann, kopieren wir die Funktion so, dass SPE auch eine Kopie der Funktion besitzt. Die Funktion wird dann von PPE-Compiler bzw. SPE-Compiler kompiliert und zum ausführbaren Code gebracht.

2.2.3 Memory Model

In Cell-Prozessor besitzt jeder SPE nur 256K direkt zugreifbaren Lokalspeicher für Code und Daten. Nur die privaten Variablen werden im Lokalspeicher gesichert. Alle anderen Variablen befinden sich im Systemspeicher. Das SPE kann auf sie durch DMA-Operationen zugreifen. Es gibt zwei Mechanismen für DMA-Übertragung: *static buffering* und *compiler-controlled software cache*. Bei beiden Mechanismen

kann das SPE die globalen Daten in den Lokalspeicher kopieren. Der SPE-Thread darf die lokale Kopie lesen oder überschreiben.
Reguläre Referenzen sind solche Referenzen, die sich in einer Schleife befinden und mit affinem Ausdruck der Induktionsvariablen der Schleife formuliert werden können. Außerdem sind die regulären Referenzen immer unabhängig von anderen Iterationen. Für reguläre Referenzen auf gemeinsam genutzte Daten/Datenstrukturen verwenden wir einen temporären Buffer im SPE Lokalspeicher. Für read-Referenzen wird der Buffer vor der Ausführung der Schleife durch eine DMA *get*-Operation initialisiert. Und für write-Referenzen wird der Wert im Buffer nach der Ausführung der Schleife durch eine DMA put-Operation kopiert.
Für irreguläre Referenzen auf gemeinsam genutzte Systemspeicher verwenden wir eine *compiler-controlled software cache* um die Daten in den Systemspeicher zu schreiben oder aus dem Systemspeicher auszulesen. Der Compiler fügt eine Zeile mit Referenz und entsprechendem Wert in der *software cache* ein. Wenn eine Cache Zeile mit der richtigen Referenz gefunden wurde, wird der Wert verwendet. Ist das nicht der Fall, so wird ein *Cache Miss* ausgegeben. Nach dem Miss fügt der Compiler eine Zeile ohne Wert in die Cache ein. Bei einem „*Read*“ lesen wir die Daten in Systemspeicher mit einer DAM get-Operation ein. Um die Daten zu speichern, schreiben wir die Daten in den Cache mit *dirty Bits*. Die *dirty Bits* geben an, welche Bytes modifiziert wurden. Die neuen Daten werden dann mit einer DMA put-Operation in den Systemspeicher geschrieben, nachdem ein cache flush aufgerufen wurde oder nachdem die Cache Zeile gelöscht werden muss um Platz für andere Daten freizulassen.

3. Transactional Memory und OpenMP

Es ist besonders schwierig die OpenMP-Codes zu schreiben, die die aktiven parallelen Threads gut synchronisiert haben, da critical regions(locks), atomic regions und barriers verwendet werden. Eine ineffiziente Anordnung der Datenzugriffe und Synchronisation-Primitiven begrenzen die Produktivität der Programmierer und die Performance der Anwendungen für Multi-Core Prozessoren. Das Transactional Memory(TM) ist ein entscheidender Mechanismus zur Lösung von Problemen. Mit *Transactional Memory* bezeichnet man ein neues Hauptspeicherkonzept, das bei Mehrprozessorsystemen zum Einsatz kommen soll. Ziel ist es die Schwierigkeiten der Synchronisierung und Koordination, die bei parallelen Berechnungen entstehen, vom Programmierer in den Compiler und Hardware zu verlagern. TM ermöglicht mehreren Threads gleichzeitigen Zugriff auf eine Adresse im gemeinsam genutzten Speicher. Falls Konflikte beim Zugriff entdeckt werden, wird die Transaktion rückgängig

gemacht. Bisher befindet sich Transactional Memory noch im Forschungsstadium. Transactional Memory kann entweder komplett in Software (STM) [5], als Hardware (HTM) [6] oder mit Hardware-Unterstützung (Hybrid-HTM) [7], [8], [9], [10] implementiert werden. Bestehende Hardware Transaktionale Speichersystem gibt es zurzeit nur als Forschungsprototypen.
Im Vergleich zum TM ist ein lock-basierter Mechanismus viel pessimistischer. Während mit Mutual-Exclusion nur ein Thread in einer bestimmten Zeit ein Lock halten darf, können mehrere Threads mit TM versuchen, gleichzeitig auf gemeinsame Ressourcen zuzugreifen. Weil Konflikte bei vielen Programmen sehr selten auftreten [4], so ist der optimistische TM ein viel sinnvolleres Programmierungsmodell für die Zukunft.

3.1 OpenMP Extensionen für TM

Als Transaktion bezeichnet man eine feste Folge von Operationen, die als eine Einheit betrachtet werden. Insbesondere wird für Transaktionen gefordert, dass sie entweder vollständig ausgeführt werden(commit) oder gar keine Auswirkung haben(abort). Um Transaktionen in den OpenMP-Code zu integrieren, werden ein paar Open-Extensionen für TM verwendet. Eine wichtige Extension ist ein *pragma*, welches den Code abgrenzt, der eine Transaktion enthält.

```
#pragma omp transaction [ exclude (list) | only (list) ]
    structured-block
```

Mit dieser Extension können Programmierer die Folge von Anweisungen (engl. *statement*) festlegen, die als eine Transaktion ausgeführt werden sollen. Die optionale *exclude* Klausel gibt eine Liste von Variablen an, für die keine Überprüfung auf Konflikte nötig ist, d.h., eine Sicherungskopie unnötig. Im Gegensatz dazu kann der Programmierer die *only* Klausel verwenden, die dann die Originalwerte von Variablen in der Liste speichert, somit können die Werte nach dem Konflikt wieder zurückgesetzt werden.
Für Deklaration einer Transaktion gibt es noch andere Möglichkeiten:

```
#pragma omp for transaction [ exclude (list) | only (list) ]
    for (. . . , . . . , . . . )
        structured-block

und

#pragma omp sections transaction    [ exclude (list) | only (list) ]
#pragma omp section
    structured-block
#pragma omp section
    structured-block
```

Während im ersten Fall jede Iteration der Schleife eine Transaktion ist, ist jede Sektion im zweiten Fall eine Transaktion.
Für OpenMP 3.0 ist ein neues Ausführungsmodell der Aufgaben vorgesehen. Die Aufgaben werden in mehrere kleine Aufgaben aufgeteilt, die von jedem Thread der Arbeitsgruppe ausgeführt werden können. Eine Aufgabe kann wieder neue Aufgaben erzeugen. Um eine Aufgabe als eine Transaktion zu deklarieren, verwendet man eine ähnliche Klausel.

```
#pragma omp task transaction [ exclude (list) | only (list) ]
    structured-block
```

3.2 Ausführungsumgebung für TM: Nebelung

Eine STM Bibliothek namens Nebelung wurde implementiert, die eine vollständige Ausführungsumgebung für TM bietet. Die aktuelle Implementierung der Bibliothek hat keine Unterstützung auf der Hardware-Ebene. Nebelung definiert ein paar Funktionen vor, um den Code, der vom Mercurium generiert wird, zu unterstützen.

```
Transaction* createtx ();
void starttx (Transaction *tr);
status committx (Transaction *tr);
void destroytx (Transaction *tr);
void aborttx (Transaction *tr);
void retrytx (Transaction *tr);
void* readtx (Transaction *tr, void *addr, int blockSize);
void* writetx (Transaction *tr, void *addr, void *obj, int blockSize);
```

Semantik der Funktionen sind folgende: *createtx* erstellt die benötigte Datenstrukturen für die Ausführung einer Transaktion, während *destroytx* die Datenstrukturen wieder freigibt. Mit *starttx* wird die Transaktion gestartet. *Committx* macht die Ergebnisse(writes) der Transaktion sichtbar für andere Transaktionen. Mit *aborttx* kann man die Transaktion abrechen. *Retrytx* bricht die Transaktion ab und startet sie neu. Die Funktionen die Speicherzugriffe ausführen sind *readtx* und *writetx*.

```
{   Transaction* t = createtx(); while (1) {
   starttx (t);
   if (setjmp (t->context) == TRANSACTION_STARTED) {
```

(a)

```
         if (COMMIT_SUCCESS == committx (t)) break;
         else aborttx (t);
       } else aborttx (t);
   }
   destroytx (t);
}
```

(b)

```
startTransaction();
// transaction body
endTransaction();
```

(c)

Abbildung 5: Macros für (a) Start und (b)Beendung eine Transaktion.
(c)Code für die Transaktion wird umschlossen von den Marcos.

Abbildung 5 zeigt ein paar Zeilen Beispielcode für eine Transaktion auf. Die Transaktion wird gestartet durch die Funktion *createtx*, gefolgt von der Funktion *starttx*. Diese wird durch die Funktion *destroytx* beendet. Der Körper der Transaktion wird vom Start und Ende umschlossen.
Die wichtigsten Teile der Bibliothek sind die Funktionen *readtx* und *writetx*. *Writetx* bekommt die Daten, die Größe der Daten in Byte und die Adresse, wo die Daten abgespeichert werden sollen. Die Funktion *readtx* bekommt die Adresse, wo die Daten ausgelesen werden sollen und die Größe der Daten. *Readtx* gibt einen Pointer auf der Adresse zurück, wo die Daten abgespeichert werden. Je nach Implementierung muss der zurückgegebene Pointer nicht identisch mit der Adresse sein, wo die Daten ausgelesen worden sind. Das heißt, man kann die Daten in eine neue Adresse kopieren und den Pointer an die neue Adresse zurückgeben.
In Nebelung wurde lazy Konflikterkennung(lazy conflict detection) implementiert. Die Transaktion führt alle Speicheroperationen lokal aus. Nebelung überprüft auf Konflikt mit anderen Transaktionen erst bei *commit*. Wenn es einen Konflikt erkannt hat, commitet er die aktuelle Transaktion und bricht andere Transaktionen ab (abort).

3.3 Source-to-source OpenMP Mercurium compiler

Mercurium ist ein OpenMP source-to-source Übersetzer. Er wandelt den OpenMP-Code um, so dass die STM-Bibliothek den Code auch unterstützt. Es wird für jeden Speicherzugriff im Block der Transaktion eine geeignet STM-Bibliothek Funktion eingesetzt. Abbildung 6 gibt einen Beispielcode an. Jede Codezeile im Block, die auf den Speicher zugreift, wird in eine passende Funktion umgewandelt.

```
int f(int); int correct(int* a, int* b, int* x){
   int fx;
#pragma omp transaction exclude (fx) {
   fx = f(*x);
   a += fx;
   b -= fx;
   }
}
```

(original code)

```
int correct(int* a, int* b, int* x){
   int fx;
   { startTransaction();
      {
      fx = f(*read(t, x));
      write(t, &a, *read(t, &a) + fx);
      write(t, &b, *read(t, &b) - fx);
      }
   endTransaction();
   }
}
```

(transactional code)

Abbildung 6

3.4 Offene Probleme

3.4.1 Verschachtelte Transaktionen

Das erste Problem bei der Integration des TMs in OpenMP ist die Verschachtelung der Transaktionen. Eine verschachtelte Transaktion ist eine Transaktion, die vollständig von einer anderen Transaktion umschlossen wird. Die innere Transaktion sieht die Veränderungen, die von der äußeren gemacht werden. Es gibt zwei verschiedene Implementierungen von Verschachtelungen: offene Verschachtelung und geschlossene Verschachtelung. Im geschlossen verschachtelten TM-System führt ein Abbruch der inneren Transaktion nicht automatisch zu einem Abbruch der äußeren Transaktion. Schließt die innere Transaktion ihre Aktionen erfolgreich ab, sind die gemachten Änderungen nur für die äußere Transaktion sichtbar. Für das ganze System ist die Änderung erst sichtbar, wenn die äußerste Transaktion erfolgreich terminiert. Im Gegensatz dazu wird bei offenen Transaktionen die Änderung der inneren Transaktion sofort nach deren Termination im ganzen System sichtbar. Diese Änderungen bleiben

bestehen auch wenn die äußere Transaktion später abgebrochen wird. Das heißt, andere Transaktionen können die Änderungen gleich sehen und die aktuellen Daten bearbeiten. Bei der offenen Verschachtelung wird es mehr Nebenläufigkeit erzielt. Eine verschachtelte Transaktion ist aber eine Herausforderung für Programmierer, weil zusätzliche Aktionen nötig sind, wenn die äußerste Transaktion commitet und eine innere Transaktion abbricht. Außerdem kann das Programmieren eines zusätzlichen Codes kann kompliziert sein.

3.4.2 I/O-Operationen in einer Transaktion

Das zweite Problem ist die Verwendung der I/O-Operationen in einer Transaktion. I/O-Operationen in *Critical Section* ist kein Problem für OpenMP, da die Operationen nie rückgängig gemacht werden. Beim TM-System ist das aber anders, weil diese Transaktion manchmal abbricht. Das ist problematisch, wenn in einer Transaktion ein System-Aufruf Zeichen im Terminal ausgibt und die Transaktion nachher abbricht, weil die Rücknahme der Ausgabe im Terminal das System unglaubwürdig macht. Man glaubt, die beste Lösung ist ein Verbot der I/O-Operationen in der Transaktion.

3.4.3 Verschachtelung der Transaktionen und OpenMP Konstrukt.

Eine Transaktion kann sich in einem OpenMP parallelen Konstrukt befinden. Das bringt uns keine Schwierigkeit, da jeder Thread nur eine Transaktion ausführen muss. In einem anderen Fall, wird es viel komplizierter wenn ein OpenMP paralleles Konstrukt in einem Transaktion-Block auftritt. Das Problem ist, dass die OpenMP Threads jeder Zeit abbrechen können. Wenn ein von der Transaktion erzeugtes Thread aufgrund eines Konflikts abgebrochen ist, müssen alle anderen von der Transaktion erzeugten Threads wegen der Atomarität der Transaktion auch abbrechen.

Referenzen

[1] R. Rabenseifner: *Optimization of Collective Reduction Operati ons.* International Conference on Computational Science 2004: 1 -9
[2] Kistler, M., Perrone, M., Petrini, F.: CELL multiprocessor communication network:

Built for Speed. IEEE Micro 26(3) (May/June 2006)

[3]. SDK for Cell, http://www-128.ibm.com/developerworks/power/cell

[4]. Harris, T., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing Memory Transactions. In: PLDI 2006. ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (June 2006)

[5]. Shavit, N., Touitou, D.: Software Transactional Memory. In: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, pp. 204–213 (1995)

[6] 8. Herlihy, M., Eliot, J., Moss, B.: Transactional Memory: Architectural Support for
Lock-Free Data Structures. In: Proc. of the 20th Int'l Symp. on Computer Architecture
(ISCA 1993), May 1993, pp. 289–300 (1993)

[7]. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybid
Transactional Memory. In: Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (October 2006)
[8]. Kumar, S., Chu, M., Hughes, C.J., Kundu, P., Nguyen, A.: Hybrid Transactional Memory. In: Proceedings of ACM Symp. on Principles and Practice of Parallel Programming (March 2006)

[9]. Saha, B., Adl-Tabatabai, A., Jacobson, Q.: Architectural Support for Software Transactional Memory. In: 39th International Symposium on Microarchitecture (MICRO) (2006)
[10]. Shriraman, A., Marathe, V.J., Dwarkadas, S., Scott, M.L., Eisenstat, D., Heriot, C.,
Scherer III, W.N., Spear, M.F.: Hardware Acceleration of Software Transactional Memory. In: TRANSACT 2006 (2006)