MPI und OpenMP auf Multicoresysteme

Hauptseminar
Multicore-Programmierung
SS 2009

Cheng Xu
28.Mai 2009

Inhalt

1. Transactional Memory und OpenMP
2. OpenMP auf Cell-Prozessor
3. Optimierte MPI kollektive Kommunikationen für Multicore-Systeme

Transactional Memory und OpenMP

- Was ist Transactional Memory(TM)?

  Ein neues Hauptspeicherkonzept, das bei Mehrprozessorsystemen zum Einsatz kommen soll.
- Im Forschungsstadium

Ziel

Die Schwierigkeiten der Synchronisierung und Koordination, die bei parallelen Berechnungen entstehen, vom Programmierer in den Compiler und auf die Hardware zu verlagern.

TM vs. Lock-basierter Mechanismus

- Lock-basierter Mechanismus

  In einer bestimmten Zeit darf nur ein Thread der critical section auf gemeinsame Ressourcen zugreifen.
- TM

  Mehrere Prozesse versuchen gleichzeitig auf gemeinsame Ressourcen zuzugreifen.

  Konflikte treten selten auf [1]

  Rollback bei Konflikten

  Erzielt mehr Nebenläufigkeiten

[1]. 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)

Transaktion

- Was ist eine Transaktion?

  Eine feste Folge von Operationen, die als eine Einheit betrachtet wird.

  Sie ist entweder vollständig oder überhaupt nicht ausgeführt wird.

Integration der Transaktion in den OpenMP-Code

- Extension wurde gebraucht.

  Ein pragma wird verwendet um den Code einer Transaktion abzugrenzen.

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

  exclude Klausel: eine Liste von Variablen, für die keine Überprüfung auf Konflikte nötig ist.

  only Klausel: eine Liste von Variablen, die überprüft werden müssen.

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
```

Ausführungsumgebung für TM : Nebelung

- Nebelung:

  Eine STM Bibliothek

  Sie bietet eine vollständige Ausführungsumgebung für TM

  keine Unterstützung auf der Hardware-Ebene

Funktionen der Bibliothek

```
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);
```

Beispiel

```
{   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)

Nebelung

In Nebelung wurde lazy Konflikterkennung(lazy conflict detection) implementiert.

Die Transaktion führt alle Speicheroperationen lokal aus.

Überprüfung auf Konflikt erst beim commit.

Wenn ein Konflikt erkannt wird, commitet er die aktuelle Transaktion und bricht andere Transaktionen ab (abort).

Source-to-source OpenMP Mercurium compiler

- Mercurium
  - Ein OpenMP source-to-source Übersetzer
  - Es wird für jeden Speicherzugriff in der Transaktion eine geeignete STM-Bibliothek Funktion eingesetzt

Beispiel

```
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)

Offene Probleme

- Verschachtelte Transaktionen
- I/O-Operationen in einer Transaktion
- Verschachtelung der Transaktionen und OpenMP Konstrukt.

Verschachtelte Transaktionen

- Zwei verschiedene Implementierungen
  - Offene Verschachtelung

    Die Änderung der inneren Transaktion ist sofort nach deren Termination im ganzen System sichtbar.

    Diese Änderungen bleiben bestehen auch wenn die äußere Transaktion später abgebrochen wird.

    Erzielt mehr Nebenläufigkeit.

  - Geschlossene Verschachtelung

    Ein Abbruch der inneren Transaktion führt nicht automatisch zu einem Abbruch der äußeren Transaktion.

    Schließt die innere Transaktion ihre Aktionen erfolgreich ab, so 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.

Problemstellung der verschachtelten Transaktionen

Zusätzliche Aktionen sind nötig, wenn die äußerste Transaktion commitet und eine innere Transaktion abbricht.

Das Programmieren einer zusätzlichen Codes kann sehr kompliziert sein.

I/O-Operation in einer Transaktion

Bei TM-Systemen bricht Transaktion manchmal ab.

I/O-Operationen die vor dem Abbruch aufgeführt werden sind problematisch.

Es ist sehr schwierig von einer I/O-Operation (z.B. eine Ausgabe im Terminal) zurückzutreten.

Führt zu Unglaubwürdigkeit des Systems.

Beste Lösung : Verbot der I/O-Operation in der Transaktion.

Verschachtelung der Transaktionen und OpenMP Konstrukt

- Eine Transaktion in einem OpenMP parallelen Konstrukt

  ⇒ Jeder Thread führt eine Transaktion aus

- Ein OpenMP paralleles Konstrukt in einem Transaktion-Block

  Abbruch eines OpenMP-Threads

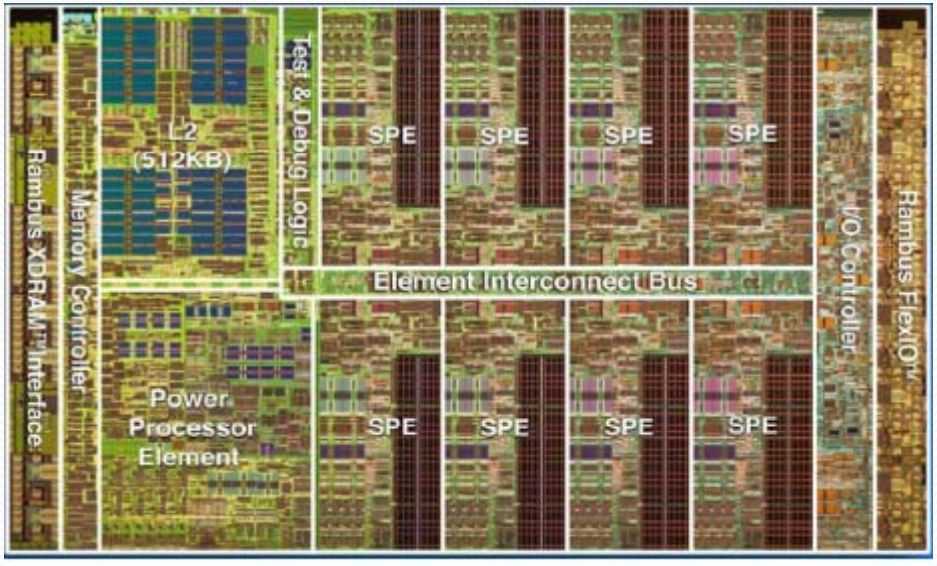
  ⇒ Abbruch aller anderen von der Transaktion erzeugten Threads wegen der Atomarität der Transaktion

OpenMP auf Cell-Prozessor

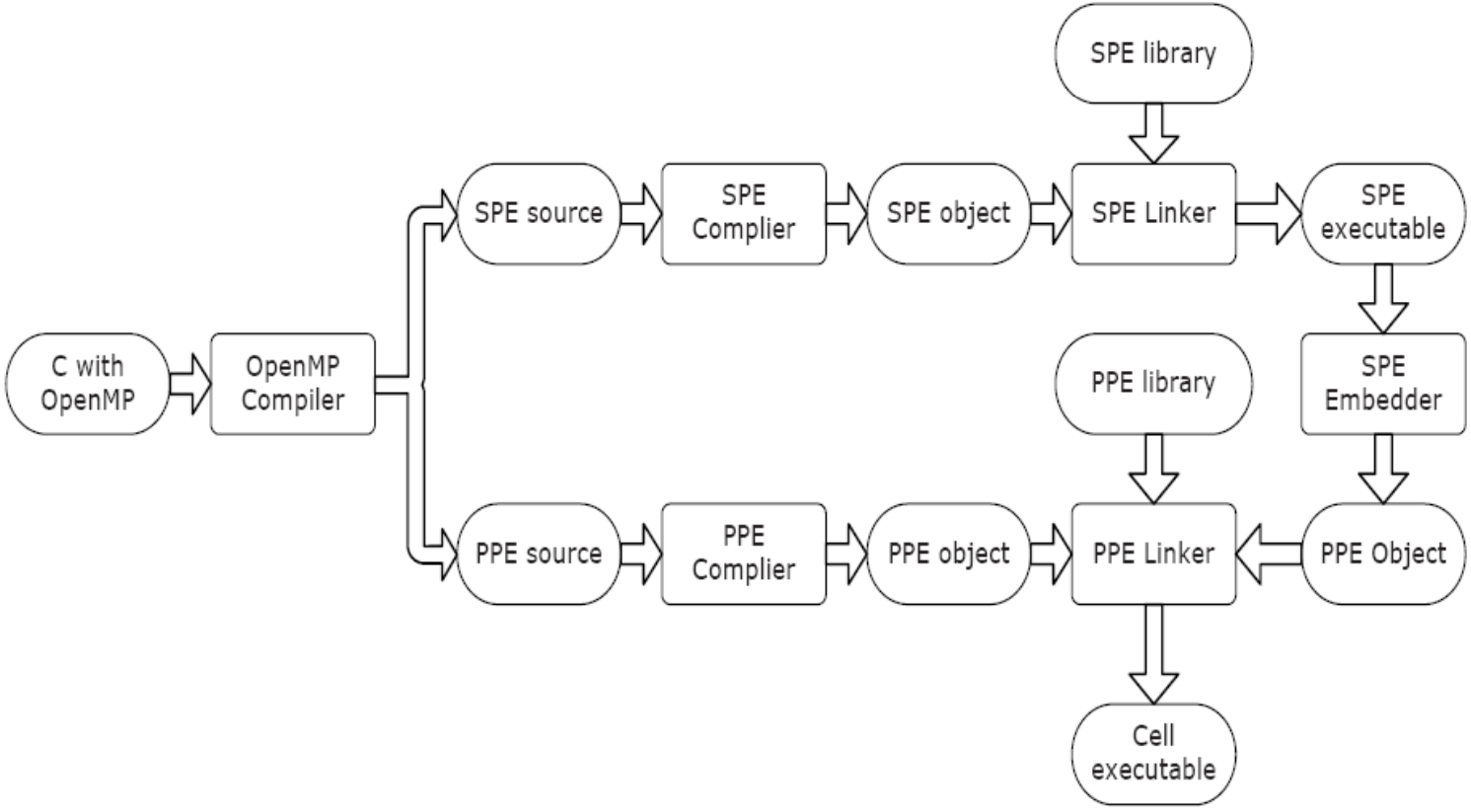
- Was ist ein Cell?
  - Ein Multicore-Chip, der von IBM gemeinsam mit Sony und Toshiba entwickelt wurde
  - Hauptprozessor der Spielkonsole Playstation 3
  - Cell ist für rechenintensive Anwendungen, wie Multimedia, Computerspiele, Videos und andere Formen digitaler Inhalte optimiert.

Aufbau

- Acht einzelne synergistic Processor Elements (SPEs), die extrem umfangreiche Fließkomma-Berechnungen verarbeiten können.
- Ein 64-Bit Power Processor Element(PPE)
- Die neun Prozessorkerne sind über einen *Element Interface Bus* (EIB) gekoppelt.
- Dem PPE stehen 512 KB L2-Cache zur Verfügung



Der OpenMP Compiler Framework für Cell



Probleme

- Um OpenMP auf Cell zu implementieren gibt es drei Probleme:
  - Threads und Synchronisation
  - Codegenerierung
  - Speichermodell

Threads und Synchronisation

- Erzeugung und Synchronisation der Threads wurden mit *Cell Software Development Kit*(SDK) [2] implementiert
- Threads
  - PPE und SPEs erzeugen beide Threads
  - Ein Master-Thread in PPE

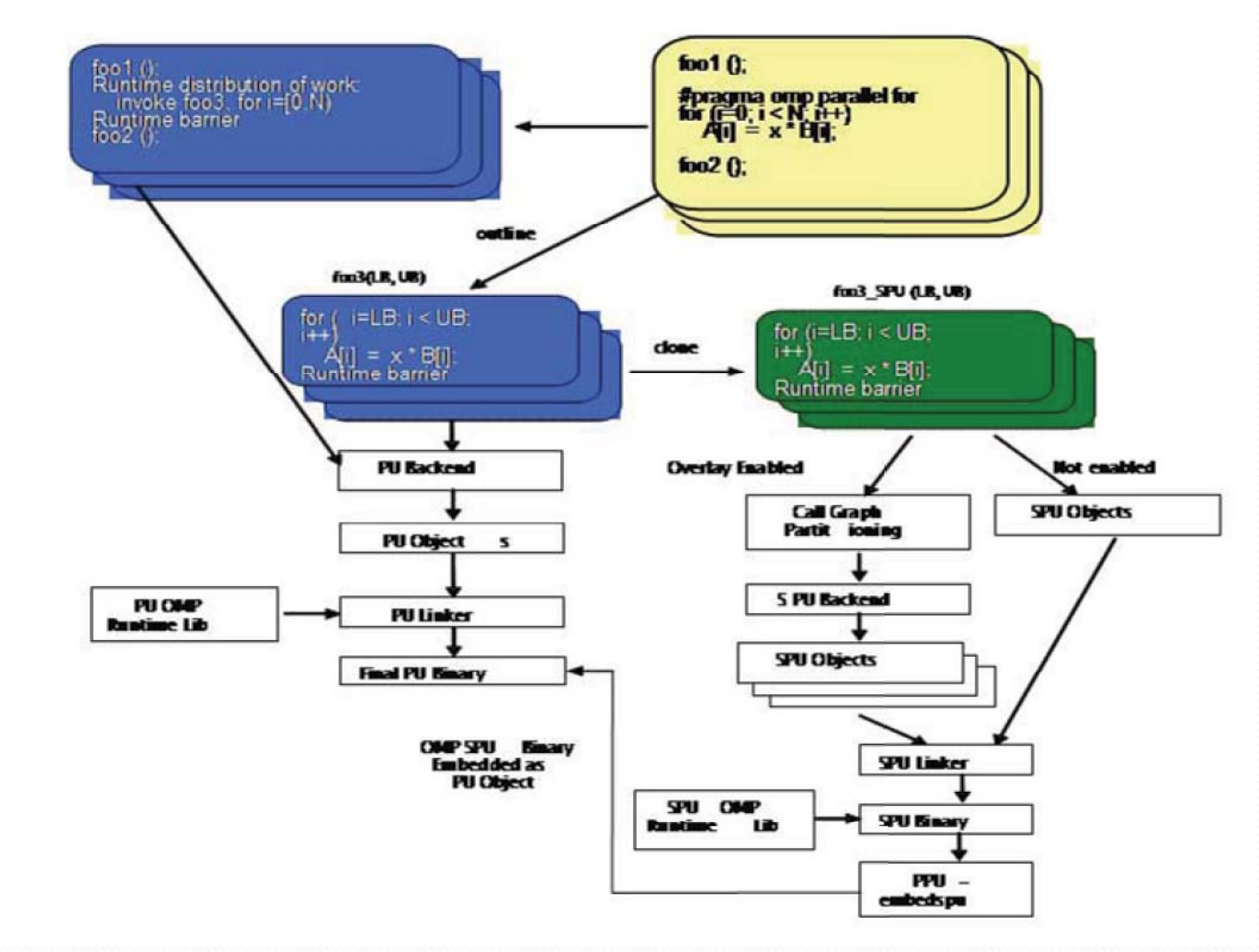
    Zuständig für Erzeugung der Threads, Zuweisung der Aufgaben und Initialisierung der Synchronisationsoperationen
  - Task Queue für jeden Thread

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

Threads und Synchronisation

- Synchronisation
  - *Memory Flow Controller*(MFC)
  - zwei blockierte *Outbound Mailbox Queues* und eine nicht blockierte *Inbound Mailbox Queue*
  - 32-bit Nachrichten

Codegenerierung



Speichermodell

- 256K direkt zugreifbarer Lokalspeicher für jeden SPE
- Nur private Variablen in Lokalspeicher
- Alle anderen Variablen befinden sich in Systemspeicher. Zugriff auf diese durch DMA-Operationen *
- Zwei Mechanismen für DMA-Übertragung:
  - *static buffering*
  - *compiler-controlled software cache*
- Eine lokale Kopie in Lokalspeicher

* **Direct Memory Access** (**DMA**) bezeichnet in der Computertechnik eine Zugriffsart, die über ein Bussystem direkt auf den Speicher zugreift.

Speichermodell

- Reguläre Referenzen
  - Temporär Buffer im SPE Lokalspeicher
  - Für read: DMA *get*-Operation
  - Für write: DMA put-Operation
- Irreguläre Referenzen
  - *compiler-controlled software cache*
  - Für read: DMA *get*-Operation
  - Für write: Erst *dirty* Bits in die Daten einfügen und dann die DMA put-Operation

Optimierte MPI kollektive Kommunikationen für Multicore-Systeme

- Message Passing Interface(MPI) ist ein Standard, der den Nachrichtenaustausch bei parallelen Berechnungen auf verteilten Computersystemen beschreibt.
- Punkt-zu-Punkt- und Kollektive Kommunikation
- Kommunikator
  - bestimmt eine Gruppe von Prozessen
  - Umgebung für Punkt-zu-Punkt- und Kollektive Kommunikation.
  - z.B. : MPI_COMM_WORLD: umfasst die Gruppe aller Prozesse

Motivation

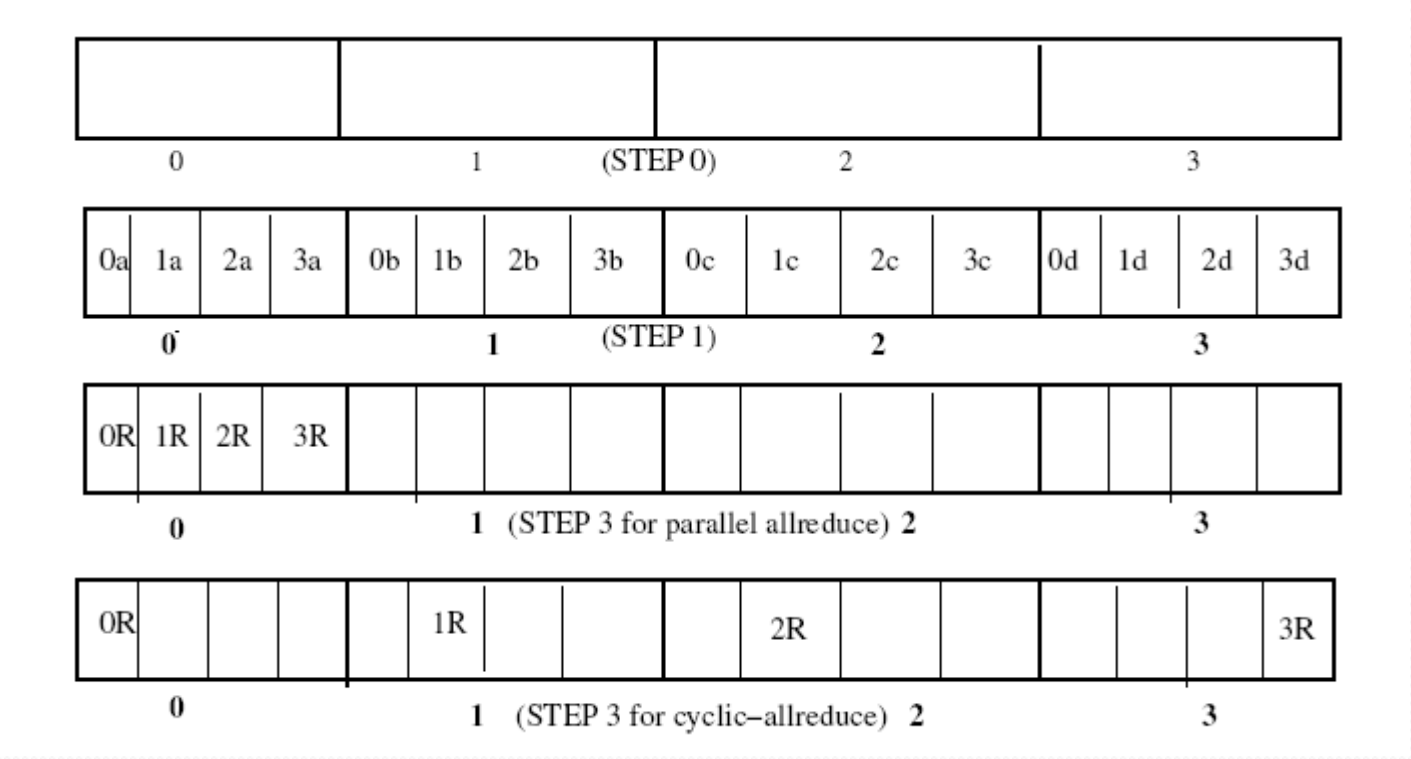
- Über 40% der Ausführungszeit von Routinen des MPI entfallen auf die kollektiven Kommunikationsroutinen MPI_Allreduce und MPI_Reduce.
- Eine Optimierung dieser und generell aller kollektiven Operationen wäre im Bezug auf die Performance von parallelen Programmen, die auf MPI aufbauen, von sehr großem Nutzen.

Algorithmen

- Ziel:
  - *Memory Traffic* reduzieren
  - Reduzierung des Datenverkehrs im Speicher

- Allgemeines Verfahren: eine bestimmte Größe von Segmenten in gemeinsam genutzten Speicher wird jedem MPI Kommunikator zuweisen.

Optimierter MPI_Allreduce



Optimierter MPI_Bcast