TECHNISCHE UNIVERSITÄT DRESDEN

Center for Information Services and High Performance Computing (ZIH)

# Linux Cluster in Theorie und Praxis

*Parallelisierung und Cell Programmierung*

03. Dezember 2009

INF 1046
Nöthnitzer Straße 46
01187 Dresden
0351 - 463 38783

Stefan Höhlig, Andy Georgi

ZIH
Center for Information Services & High Performance Computing

## Foliensatz

**Verfügbarkeit der Folien**

Vorlesungswebseite:
`http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/lehre/ws0910/lctp`



## Inhalt I





## Inhalt II

## Motivation

- Beschleunigung von Berechnungen
- Erhöhung des Durchsatzes
- Erfüllung großer Speicheranforderungen
- Beispiele:
  - Grand Challenges in der Grundlagenforschung:
    - Entschlüsselung menschlicher Gene
    - Kosmogenese
    - Globale Klimaveränderungen
    - Biopolymere
  - Simulation von Wetter und Klima
  - Medizinische Bildgebung
  - Windkanalsimulation
  - ...



## Lösungsansätze

- Ausnutzung des technischen Fortschritts (schnellere Hardware, mehr Speicher...)
- Effizientere Verarbeitung (Optimierungen, effiziente Algorithmen...)
- Vervielfachung der Ressourcen (parallele Rechnersysteme)

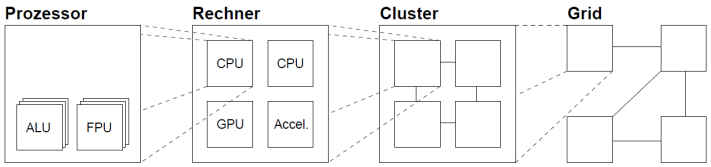


## Lösungsansätze

- Ausnutzung des technischen Fortschritts (schnellere Hardware, mehr Speicher...)
- Effizientere Verarbeitung (Optimierungen, effiziente Algorithmen...)
- **Vervielfachung der Ressourcen (parallele Rechnersysteme)**

## Ebenen der Parallelität

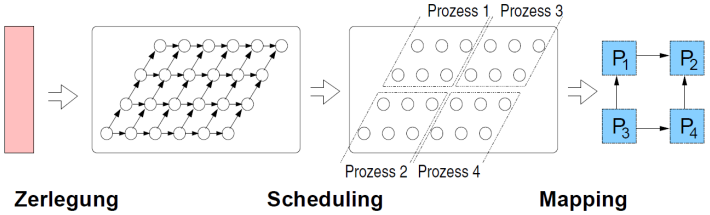
- Funktionseinheiten eines Prozessors
- Mehrere Prozessoren eines Rechners
- Clustering
- Grid-Computing






## Parallelisierung von Programmen

Wesentliche Schritte bei der Parallelisierung eines sequentiellen Algorithmus:






## Daten- vs. Taskparallelität

- **Datenparallelität:** Anwendung einer Instruktion auf mehrere Elemente einer Datenstruktur
- **Taskparallelität:** Simultane Verarbeitung voneinander unabhängiger Instruktionsströme
- Zudem können einzelne Tasks ggf. von mehreren Prozessoren - d.h. datenparallel - ausgeführt werden



## Implizite Parallelisierung

- **Idee:** Parallelisierende Compiler
- **Probleme:**
  - Automatisierte Analyse von Abhängigkeiten
  - Lastverteilung
  - Kommunikationsorganisation
- **Fazit:** Implizite Parallelisierung ist nur eingeschränkt anwendbar

## Explizite Parallelisierung

- Verwendung von expliziter Parallelisierung in der Praxis
- Gängige Programmiermodelle:
  - **Message Passing Interface (MPI):** Bibliotheksfunktionen für nachrichtenorientierte Kommunikation und Synchronisation von Prozessen
  - **OpenMP:** Definition paralleler Programmteile (insbesondere datenparallele Schleifen) durch Compilerdirektiven
  - **CUDA:** Programmierung von GPU's durch Befehlserweiterungen der Sprache C
  - **OpenCL:** Offener Standard für parallele Programmierung heterogener Rechnersysteme







## Historie I

- Problem: Portierung paralleler Programme auf andere Architekturen nur mit hohem Aufwand möglich
- Idee: Eine von der Architektur unabhängige und standardisierte Programmierschnittstelle
- Umsetzung: 1993-94 durch das MPI-Forum [MPIFORUM]
- Veröffentlichung von MPI 1.0 1994
- Unterstützte C und Fortran 77
- Offene und herstellerspezifische Implementierungen waren schnell verfügbar und MPI wurde der de-facto Standard für Message Passing auf Parallelrechnern



## Historie II

- Erneutes Treffen des MPI-Forum 1995 mit dem Ziel MPI-2 zu spezifizieren
- Zunächst erfolgten allerdings Fehlerkorrekturen mit MPI 1.1
- Standardisierung von MPI 2.0 wurde 1997 beendet
- Darin enthaltene Neuerungen:
  - Parallele I/O
  - Remote Direct Memory Access (RDMA)
  - Dynamisches Prozessmanagement
  - Zusätzliche Unterstützung von C++ und Fortran 90
  - Externe Schnittstele für Debugger, Profiler und Tracing-Tools
- Aktueller Stand vom 04. September 2009: MPI 2.2

# Punkt-zu-Punkt-Kommunikation I

- Bereitstellung grundlegender Mechanismen zum Senden und Empfangen von Nachrichten sowie zur Handhabung komplexer Datentypen
- Jeder Nachricht werden folgende Informationen hinzugefügt:
  - Rang des Senders und Empfängers innerhalb des Kommunikators
  - Ein Message Tag zur Identifikation der Nachricht
  - Der Kommunikator, welcher den Kontext beschreibt in dem die Kommunikation stattfindet
  - Datentyp der zu übertragenen Nachricht, wodurch korrektes Byteordering und Datenkonvertierung in heterogenen Systemen sichergestellt wird



# Punkt-zu-Punkt-Kommunikation II

Allgemeiner Kommunikationsablauf:

1. Daten dem Sendepuffer entnehmen und erzeugen einer Nachricht
2. Übertragung der Nachricht an den Empfänger
3. Schreiben der eingegangenen Nachricht in den Empfangspuffer

Blockierende und nicht-blockierende Kommunikation:

- Eine blockierende Operation kehrt erst zurück, wenn die Nachricht an den Empfänger übermittelt bzw. in einem Puffer zwischengespeichert wurde
- Nicht-blockierende Funktionen kehren dagegen sofort zurück, liefern aber zusätzlich ein Request-Handle mit dessen Hilfe geprüft werden kann, ob die Operation beendet wurde



# Punkt-zu-Punkt-Kommunikation III

Kommunikationsmodi:

- **Standard:**
  - Entscheidung über Pufferung der Nachrichten durch MPI
  - Senden der Nachricht vor Ausführung der Empfangsoperation möglich
- **Buffered:**
  - Bereitstellung des Puffers durch den Nutzer
  - Start der Sendeoperation unabhängig von einer Empfangsoperation
- **Synchronous:**
  - Beenden der Sendeoperation erst möglich sobald Daten in den Empfangspuffer kopiert werden
  - Bei Verwendung von blockierenden Operationen erfolgt implizit eine Synchronisierung von Sender und Empfänger
- **Ready:**
  - Sendeoperation kann erst nach Aufruf einer passende Empfangsoperation beginnen
  - Häufig erfolgt eine Gleichsetzung mit dem Standard-Mode



# Punkt-zu-Punkt-Kommunikation IV

| | blockierend | nicht-blockierend |
|---|---|---|
| Standard Send | MPI_Send | MPI_Isend |
| Buffered Send | MPI_Bsend | MPI_Ibsend |
| Synchronous Send | MPI_Ssend | MPI_Issend |
| Ready Send | MPI_Rsend | MPI_Irsend |
| Receive | MPI_Recv | MPI_Irecv |

Tabelle: Überblick Punkt-zu-Punkt-Kommunikationsfunktionen

## Kollektive Operationen

- Umfasst Funktionen für den Datenaustausch, an dem mehrere Prozesse einer Gruppe beteiligt sind
- Die Qualität des Algorithmus zur Verteilung der Daten ist abhängig von der MPI-Implementierung
- Enthaltene Funktionen (Auszug):
  - `MPI_Barrier` - Blockiert bis alle Prozesse der Gruppe diese Routine gerufen haben
  - `MPI_Alltoall` - Jeder Prozess der Gruppe sendet die gleiche Datenmenge an alle anderen Prozesse
  - `MPI_Bcast` - Sendet eine Nachricht von einem Prozess an alle anderen Prozesse des angegebenen Kommunikators
  - `MPI_Gather` - Sammelt Daten einer Prozessgruppe ein
  - `MPI_Scatter` - Möglichst gleichmäßige Verteilung der Daten unter allen Prozessen der Gruppe
  - `MPI_Reduce` - Führt eine globale Operation aus



## Virtuelle Topologien

- Generell linearer Namensraum (Durchnummerierung der Prozesse von Null bis n-1)
- Spiegelt häufig nicht die logische Kommunikationsstruktur wieder
- Abhängig vom zugrunde liegenden Problem oder Algorithmus treten vielmehr Strukturen wie zwei- und dreidimensionale Gitter, Ringe, Hypercubes oder allgemeine Graphen auf
- Abbildung dieser logischen Anordnung der Prozesse in MPI mit Hilfe virtueller Topologien möglich
- Funktionen zum Erzeugen von Topologien (Auszug):
  - `MPI_Graph_create` - Erzeugt eine Graph-Topologie
  - `MPI_Cart_create` - Erzeugt alle Arten kartesischer Topologien z.B. Gitter, Ring, Hypercube
  - `MPI_Dims_create` - Liefert eine ausgeglichene Aufteilung einer Anzahl von Prozessen auf eine kartesisches Struktur



## Hello World in C

```c
/* C Example */
#include <stdio.h>
#include <mpi.h>


int main (argc, argv)
{
  int rank, size;

  MPI_Init (&argc, &argv);  /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);    /* get current process id */
  MPI_Comm_size (MPI_COMM_WORLD, &size);    /* get number of processes */
  printf( "Hello world from process %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```

Kompilieren: `mpicc -o mpicc_example mpicc_example.c`
Ausführung: `mpirun -np 4 mpicc_example`

Ausgabe:

Hello world from process 2 of 4
Hello world from process 3 of 4
Hello world from process 0 of 4
Hello world from process 1 of 4



## Hello World in Fortran

```
      C  Fortran example
         program hello
         include 'mpif.h'
         integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)

         call MPI_INIT(ierror)
         call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
         call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
         print*, 'node ', rank, ': Hello world'
         call MPI_FINALIZE(ierror)
         end
```

Kompilieren: `mpif90 -o mpif90_example mpif90_example.f`
Ausführung: `mpirun -np 4 mpif90_example`

Ausgabe:

node 1: Hello world
node 0: Hello world
node 3: Hello world
node 2: Hello world

# Quellen I


- [MPIFORUM] Message Passing Interface Forum
  Webseite, 2009 `http://www.mpi-forum.org/`
- [MPI09] Message Passing Interface Forum
  MPI: A Message-Passing Interface Standard Version 2.2, Sep 2009
  `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`
- [OMPI09] Open MPI: Open Source High Performance Computing
  A High Performance Message Passing Library, 2009
  `http://www.open-mpi.org/`








# Motivation

**Problem:**

- Multicore CPUs allgemein verbreitet
- Ausnutzung der schnellen Kommunikationsmöglichkeiten in Multicore und Multi-CPU Systemen
- Doch Anwendungen meist single threaded

**Lösungsmöglichkeit:**

- Parallelisierung mittels OpenMP



# OpenMP allgemein

- **Open M**ulti-**P**rocessing
- Spezifikation von Direktiven, Bibliotheksfunktionen und Umgebungsvariablen
- Schnittstelle für Shared Memory-Programmierung
- Seit 1997 durch Universitäten, Hersteller von Hard- und Software entwickelt
- Unterstützung für C/C++ und Fortran
- Homepage: `http://openmp.org/wp/`

## Funktionsweise von OpenMP

- Thread-basierte Parallelisierung mittels Compiler-Direktiven
- **S**ingle **P**rogram **M**ultiple **D**ata-Programmiermodell
- Bietet Konstrukte zur
  - Arbeitsaufteilung
  - Synchronisation
  - Deklaration von gemeinsamen und privaten Variablen
- Arbeitet nach dem Fork-Join-Modell (Gemeinsamer Addressraum!)
- Ohne OpenMP-fähigen Compiler: serielle Abarbeitung
- Bibliotheksfunktionen in **omp.h**



## Direktivenstruktur

**C/C++:**

```
#pragma omp DIREKTIVE [KLAUSELN...]
```

**Fortran:**

```
!$OMP DIREKTIVE [KLAUSELN...]
```

- Direktiven werden ignoriert, wenn unbekannt
- Einsteiger Tutorial:
  `http://www.bero-software.de/tutorials_programming/omp.pdf`
- Referenz: `https://computing.llnl.gov/tutorials/openMP/`



## Ausgewählte Direktiven I

**omp parallel**

- Master Thread erzeugt weitere Threads (fork)
- Nachfolgender Block wird parallel von **allen** Threads ausgeführt
- Nach Blockende:
  - Beendigung der erzeugten Threads (join)
  - Master Thread arbeitet allein weiter



## Ausgewählte Direktiven II

**omp for**

- Parallelisierung von for-Schleifen
- Aufteilung aller Iterationen in Blöcke und Verteilung auf die einzelnen Threads
- Voneinander unabhängige Abarbeitung
- Verschiedene Verteilungsstrategien mittels **schedule()** möglich
- Iterationsvariable ist automatisch **private** deklariert und darf nicht im Rumpf geschrieben werden

# Ausgewählte Direktiven III

**omp sections**

- Kennzeichnet parallel aber nichtiterativ abarbeitbare Blöcke
- Implizite Synchronisation an section-Ende
- Synchronisation kann mit **nowait** Parameter umgangen werden



# Ausgewählte Direktiven IV

**omp critical**

- Kennzeichnet einen kritischen Abschnitt
- Dieser Abschnitt kann zu einem Zeitpunkt nur von einem Thread bearbeitet werden

**omp barrier**

- Blocklose Direktive
- Blockiert bis alle Prozesse diese Direktive gerufen haben



# Ein OpenMP Beispielprogramm in C

```
1   #ifndef _OPENMP
2   #include <omp.h>
3   #endif
4
5   #include <stdio.h>
6
7   int main ()
8   {
9       int i;
10
11      #pragma omp parallel
12      {
13          #pragma omp for
14          for (i=0; i<4; ++i)
15          {
16              int thread_id = omp_get_thread_num();
17              printf("Hello World, I'm number %d\n", thread_id);
18
19              if(thread_id == 0)
20                  printf("There are %d threads\n", omp_get_num_threads());
21          }
22      }
23      return 0;
24  }
```



# Ein OpenMP Beispielprogramm in C

```
1   #ifndef _OPENMP
2   #include <omp.h>
3   #endif
4
5   #include <stdio.h>
6
7   int main ()
8   {
9       int i;
10
11      #pragma omp parallel for
12      for (i=0; i<4; ++i)
13      {
14          int thread_id = omp_get_thread_num();
15          printf("Hello World, I'm number %d\n", thread_id);
16
17          if(thread_id == 0)
18              printf("There are %d threads\n", omp_get_num_threads());
19      }
20
21  return 0;
22  }
```

# Quellen I


[PaPr] Thomas Rauber, Gudula Rünger
Parallele Programmierung - 2. Auflage, 2007
Springer Verlag Berlin Heidelberg

[OMPH] OpenMP Homepage
The OpenMP API specification for parallel programming, 2009
`http://openmp.org/wp/`








# Zur Erinnerung

**Heterogenes Prozessordesign:**

- PPE: Power/PowerPC kompatibler 64 Bit RISC Prozessor mit SMT
- SPE: Spezialisierter 32 Bit Prozessor mit eigenem Befehlssatz

**Speicher:**

- SPE verfügt über Local Store, der Instruktionen und Daten hält
- Nur 256 MB XDR Hauptspeicher bei PlayStation 3

**Problem**

Wie vereint man zwei zueinander inkompatible Befehlssätze in einem Programm?



# Compiler für PPE

**ppu-gcc:**

- Zentraler Bestandteil der Toolchain
- Modifizierter GCC
- Cross Compiler
- Enthält Optimierungen für Cell BE
- Erzeugtes Programm kann direkt ausgeführt werden

## Compiler für SPE

**spu-gcc:**

- Modifizierter GCC
- Cross Compiler
- SPE Programm kann:
  - Direkt auf SPE gestartet werden (mittels SPUFS)
  - In PPE Programm eingebettet werden (mittels embedspu)
  - Zur Laufzeit von PPE Programm geladen werden (mittels spezieller Bibliotheksfunktionen)



## ppu-embedspu

- Tool zum Verbinden von PPU Code und SPU Code zu einer Binärdatei
- Verbindet Symbolnamen eines PPE Programms mit den Funktionen eines SPE Programms
- Nicht zwingend erforderlich



## IBM XLC

- C/C++ Compiler von IBM
- Benötigt GCC
- Cross Compiler
- Verwendet OpenMP Direktiven zur Identifizierung von SPE Code
- Letzte Version in Cell SDK 3.0 enthalten



## Der System Simulator

- Laufzeitumgebung entwickelt von IBM
- Unterstützt je nach Version alle gängigen IBM Prozessorarchitekturen
- Dient zur Simulation von Programmen und der Analyse deren Laufzeitverhaltens
- Enthält Tools zur Code-Optimierung
- Bestandteil des Cell SDK

## libspe2

- SPE Runtime Management Library Version 2
- Enthält Funktionen zum Steuern des SPE-Programmflusses innerhalb von PPE Programmen
- Inkompatibel zu Version 1.x



## Quellen I


[CBEPr] Cell BE Programming Tutorial
Cell BE Programming Tutorial v3.0, 2007
`https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/FC857AE550F7EB83872571A80061F788`


[XLC] Using the Single-Source Compiler
IBM XL C/C++ for Multicore Acceleration for Linux, V9.0 Runtime Environment, 2007
`http://www-01.ibm.com/support/docview.wss?uid=swg24017599`







## Cell Programmierkonzepte

**IBM publiziert 7 Konzepte:**

- Function-Offload Model
- Device-Extension Model
- Computation-Acceleration Model
- Streaming Model
- Shared-Memory Multiprocessor Model
- Asymmetric-Thread Runtime Model
- User-Mode Thread Model

## Methode I: SPE Programme direkt ausführen

**Benötigt:**

- SPUFS
  - Funktionalität des Linux Kernels
  - Kernel-Treiber bildet virtuelles Dateisystem auf die vorhandenen SPEs ab
  - SPE Programme über Dateisystembefehle ausführbar

**Vorteile:**

- Kein PPE Programm-Code nötig
- Einfachste Methode SPEs zu nutzen

**Nachteil:**

- Langsamere und unflexiblere Ausführung



## SPE Beispielprogramm

```
1       #include <stdio.h>
2
3       int main(unsigned long long spe, unsigned long long argp, unsigned long long envp)
4       {
5           printf("Hello world!\n");
6
7           return 0;
8       }
```



## Methode II: Lade SPE Programm zur Laufzeit

**Benötigt:**

- PPE Programm-Code
- Ein oder mehrer SPE Programm-Codes
- Libspe2

**Vorteile:**

- Einfache Handhabung
- Keine Anforderungen an Betriebssystem

**Nachteile:**

- Höherer Programmieraufwand
- Anfälliger für Fehler zur Laufzeit



## Ablauf im PPE Programm

**Ablauf:**

1. Öffne SPE Programmdatei
2. Erzeuge SPE context
3. Lade SPE Programm in LS
4. Führe SPE Programm aus
5. Zerstöre SPE context
6. Schließe SPE Programmdatei

## Benötigte Funktionen

- spe_program_handle_t * **spe_image_open**(const char *filename);
- spe_context_ptr_t **spe_context_create**(unsigned int flags, spe_gang_context_ptr_t gang);
- int **spe_program_load**(spe_context_ptr_t spe, spe_program_handle_t *program);
- int **spe_context_run**(spe_context_ptr_t spe, unsigned int *entry, unsigned int runflags, void *argp, void *envp, spe_stop_info_t *stopinfo);
- int **spe_context_destroy**(spe_context_ptr_t spe);
- int **spe_image_close**(spe_program_handle_t *program);



## Methode III: SPE und PPE Programm verschmelzen

**Benötigt:**

- PPE Programm-Code
- Ein oder mehrer SPE Programm-Codes
- Libspe2
- ppu-embedspu

**Vorteile:**

- Nur ein Binary am Ende
- Leichter zu Programmieren als Methode 2

**Nachteil:**

- Komplexeres Vorgehen beim Kompilieren/Linken



## Ein Beispiel

**PPE Programm Kompilieren:**
ppu-gcc -c ppu_prog.c

**SPE Programm Kompilieren und Linken:**
spu-gcc spu_prog.c -o spu_prog

**SPE Programm in PPE Objekt einbetten:**
ppu-embedspu spu_handle spu_prog spu_prog.eo

**Finaler Linkvorgang:**
ppu-gcc ppu_prog.o spu_prog.eo -o cbe_prog -lspe2



## Methode IV: SPE Code via OpenMP-Direktiven

**Benötigt:**

- IBM XLC Single Source Compiler
- PPE Programm-Code
- Programmteile zur Ausführung auf SPEs mittels OpenMP-Direktiven kennzeichnen

**Vorteile:**

- Nur ein Binary am Ende
- Keine exakte Kenntnis über Architektur und Programmierung nötig
- Somit die einfachste Methode

**Nachteile:**

- Komplexeres Vorgehen beim Kompilieren/Linken
- OpenMP-Kenntnisse benötigt

# HTLBFS I

- Hauptspeicher besteht aus Rahmen (physisch) bzw. Seiten (virtuell)
- Bspw. 4 KiB Seitengröße bei x86
- Adressauflösung ist teuer
- Daher: **T**ranslation **L**ookaside **B**uffer als Cache
- **Problem:** TLB extrem klein (häufige TLB misses)
- **Lösung:** Unterstützung größerer Speicherseiten durch Hardware



# HTLBFS II

- **H**uge **T**ranslation **L**ookaside **B**uffer **F**ile **S**ystem (HTLBFS, auch Huge TLB FS)
- Fähigkeit des Linux Kernels größe Speicherseiten zu allokieren
- Bspw. 2/4 MiB bei x86, 16 MiB bei PPC
- Nutzung über *mmap()* oder shared memory-Programmierung



# Quellen I

[CBEPP] Sony Computer Entertainment Inc
Cell Programming Primer, 2007
`aptitudeinstallcell-programming-primer`

[CBEPr] Cell BE Programming Tutorial
Cell BE Programming Tutorial v3.0, 2007
`https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/FC857AE550F7EB83872571A80061F788`

[SPUFS] SPU File System
Spufs: The Cell Synergistic Processing Unit as a virtual file system, Juni 2005
`http://www.ibm.com/developerworks/power/library/pa-cell/`

[MSCHED] R. Krishnakumar
HugeTLB - Large Page Support in the Linux Kernel
`http://linuxgazette.net/155/krishnakumar.html`