Parallel Processing in a Nutshell – OpenMP & MPI kurz vorgestellt

16. Juni 2009

- Multi-Core Prozessoren halten Einzug (nicht nur) auf dem Desktop
- Neuer Rechencluster des CITEC (16 Nodes mit je 8 Kernen -> 128 Kerne)
- Ziele:

- Multi-Core Prozessoren halten Einzug (nicht nur) auf dem Desktop
- Neuer Rechencluster des CITEC (16 Nodes mit je 8 Kernen -> 128 Kerne)
- Ziele:
  - Zeitkritische Anwendungen von Multi Core profitieren lassen

- Multi-Core Prozessoren halten Einzug (nicht nur) auf dem Desktop
- Neuer Rechencluster des CITEC (16 Nodes mit je 8 Kernen -> 128 Kerne)
- Ziele:
  - Zeitkritische Anwendungen von Multi Core profitieren lassen
  - “Langläufer” optimal auf vorhandene Rechenresourcen verteilen
- Lösungen (naiv)

- Multi-Core Prozessoren halten Einzug (nicht nur) auf dem Desktop
- Neuer Rechencluster des CITEC (16 Nodes mit je 8 Kernen -> 128 Kerne)
- Ziele:
  - Zeitkritische Anwendungen von Multi Core profitieren lassen
  - “Langläufer” optimal auf vorhandene Rechenresourcen verteilen
- Lösungen (naiv)
  - Zeitkritische Sektionen in mehrere Threads aufteilen

- Multi-Core Prozessoren halten Einzug (nicht nur) auf dem Desktop
- Neuer Rechencluster des CITEC (16 Nodes mit je 8 Kernen -> 128 Kerne)
- Ziele:
  - Zeitkritische Anwendungen von Multi Core profitieren lassen
  - “Langläufer” optimal auf vorhandene Rechenresourcen verteilen
- Lösungen (naiv)
  - Zeitkritische Sektionen in mehrere Threads aufteilen
  - Verteilen der Jobs per Hand oder in Skripten

- Multi-Core Prozessoren halten Einzug (nicht nur) auf dem Desktop
- Neuer Rechencluster des CITEC (16 Nodes mit je 8 Kernen -> 128 Kerne)
- Ziele:
  - Zeitkritische Anwendungen von Multi Core profitieren lassen
  - “Langläufer” optimal auf vorhandene Rechenresourcen verteilen
- Lösungen (naiv)
  - Zeitkritische Sektionen in mehrere Threads aufteilen
  - Verteilen der Jobs per Hand oder in Skripten

- API (C, C++, Fortran), von diversen Hard- und Softwareherstellern entwickelt und behütet (AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, SGI, Sun, Microsoft, ...)
- OpenMP besteht aus:
  - Compiler-Direktiven (#PRAGMA)

- API (C, C++, Fortran), von diversen Hard- und Softwareherstellern entwickelt und behütet (AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, SGI, Sun, Microsoft, ...)
- OpenMP besteht aus:
  - Compiler-Direktiven (#PRAGMA)
  - Runtime-Library-Routinen

- API (C, C++, Fortran), von diversen Hard- und Softwareherstellern entwickelt und behütet (AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, SGI, Sun, Microsoft, ...)
- OpenMP besteht aus:
  - Compiler-Direktiven (#PRAGMA)
  - Runtime-Library-Routinen
  - Umgebungsvariablen

- API (C, C++, Fortran), von diversen Hard- und Softwareherstellern entwickelt und behütet (AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, SGI, Sun, Microsoft, ...)
- OpenMP besteht aus:
  - Compiler-Direktiven (#PRAGMA)
  - Runtime-Library-Routinen
  - Umgebungsvariablen
- Multi-Threading auf Unix/Windows NT Plattformen mit SHARED Memory

- API (C, C++, Fortran), von diversen Hard- und Softwareherstellern entwickelt und behütet (AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, SGI, Sun, Microsoft, ...)
- OpenMP besteht aus:
  - Compiler-Direktiven (#PRAGMA)
  - Runtime-Library-Routinen
  - Umgebungsvariablen
- Multi-Threading auf Unix/Windows NT Plattformen mit SHARED Memory
- OpenMP per Kommandozeilenoption bspw. des GCC einschaltbar (-fopenmp)
- Vorteile:

- API (C, C++, Fortran), von diversen Hard- und Softwareherstellern entwickelt und behütet (AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, SGI, Sun, Microsoft, ...)
- OpenMP besteht aus:
  - Compiler-Direktiven (#PRAGMA)
  - Runtime-Library-Routinen
  - Umgebungsvariablen
- Multi-Threading auf Unix/Windows NT Plattformen mit SHARED Memory
- OpenMP per Kommandozeilenoption bspw. des GCC einschaltbar (-fopenmp)
- Vorteile:
  - Programm compiliert/läuft auch seriell

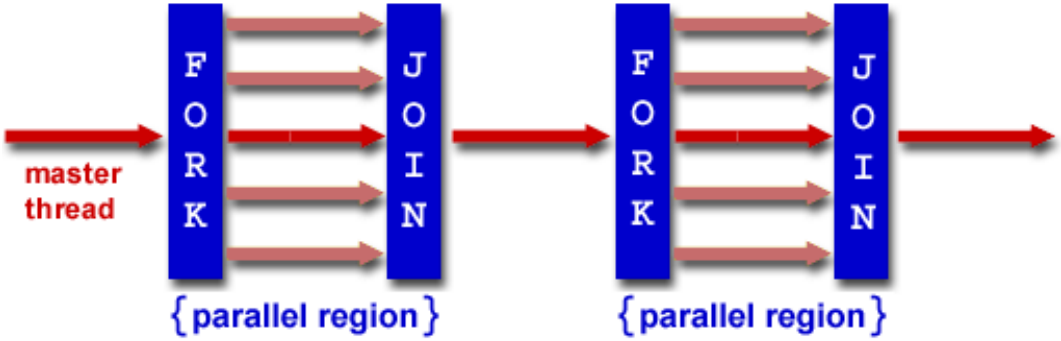
- API (C, C++, Fortran), von diversen Hard- und Softwareherstellern entwickelt und behütet (AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, SGI, Sun, Microsoft, ...)
- OpenMP besteht aus:
  - Compiler-Direktiven (#PRAGMA)
  - Runtime-Library-Routinen
  - Umgebungsvariablen
- Multi-Threading auf Unix/Windows NT Plattformen mit SHARED Memory
- OpenMP per Kommandozeilenoption bspw. des GCC einschaltbar (-fopenmp)
- Vorteile:
  - Programm compiliert/läuft auch seriell
  - Programmteile lassen sich leicht parallelisieren

- API (C, C++, Fortran), von diversen Hard- und Softwareherstellern entwickelt und behütet (AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, SGI, Sun, Microsoft, ...)
- OpenMP besteht aus:
  - Compiler-Direktiven (#PRAGMA)
  - Runtime-Library-Routinen
  - Umgebungsvariablen
- Multi-Threading auf Unix/Windows NT Plattformen mit SHARED Memory
- OpenMP per Kommandozeilenoption bspw. des GCC einschaltbar (-fopenmp)
- Vorteile:
  - Programm compiliert/läuft auch seriell
  - Programmteile lassen sich leicht parallelisieren
  - Threadanzahl muss nicht (aber kann) vorher festgelegt werden

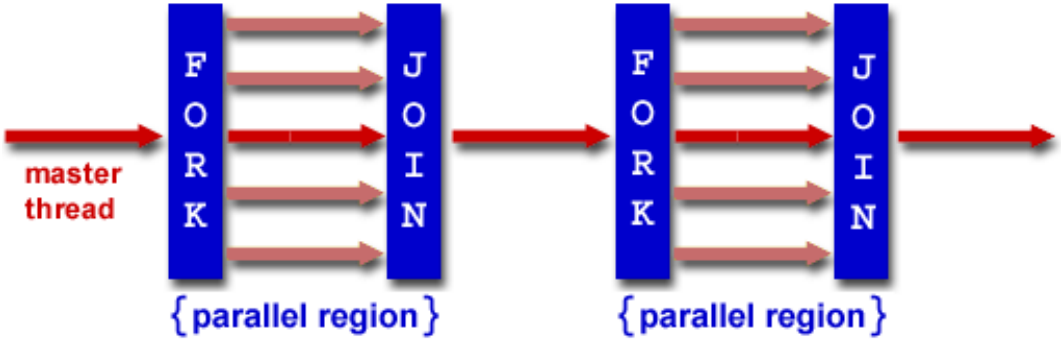
- API (C, C++, Fortran), von diversen Hard- und Softwareherstellern entwickelt und behütet (AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, SGI, Sun, Microsoft, ...)
- OpenMP besteht aus:
  - Compiler-Direktiven (#PRAGMA)
  - Runtime-Library-Routinen
  - Umgebungsvariablen
- Multi-Threading auf Unix/Windows NT Plattformen mit SHARED Memory
- OpenMP per Kommandozeilenoption bspw. des GCC einschaltbar (-fopenmp)
- Vorteile:
  - Programm compiliert/läuft auch seriell
  - Programmteile lassen sich leicht parallelisieren
  - Threadanzahl muss nicht (aber kann) vorher festgelegt werden

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[])
{
    #pragma omp parallel
    {
      printf("Hallo Welt!\n");
    }
    return (EXIT_SUCCESS);
}
```

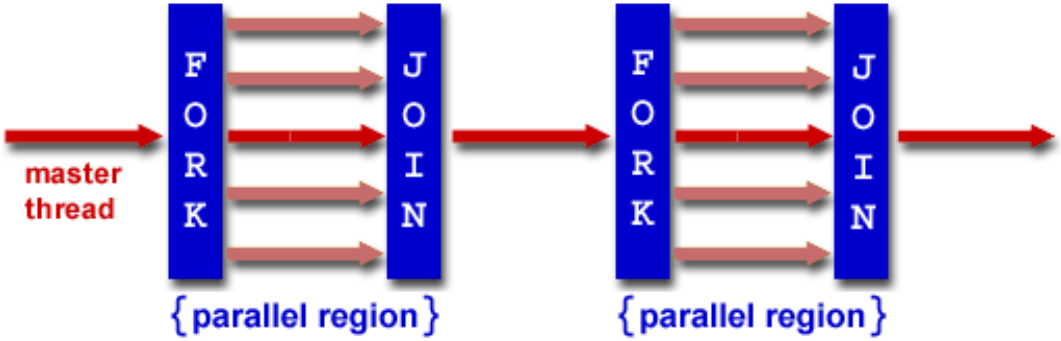
```
mschoepf@caesar$ gcc -o HelloWorld -fopenmp
HelloWorld.c
mschoepf@caesar$ ./HelloWorld
Hallo Welt!
Hallo Welt!
Hallo Welt!
Hallo Welt!
mschoepf@caesar$ ssh arminius ./HelloWorld
Hallo Welt!
```



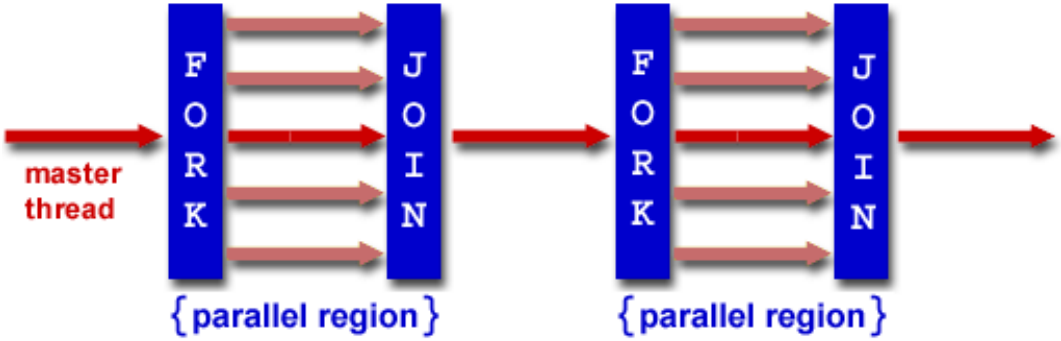
- Durch explizite Compileranweisungen werden Programmsektionen in Threads (“team”) aufgeteilt.
- Der Speicherbereich wird “geshared”



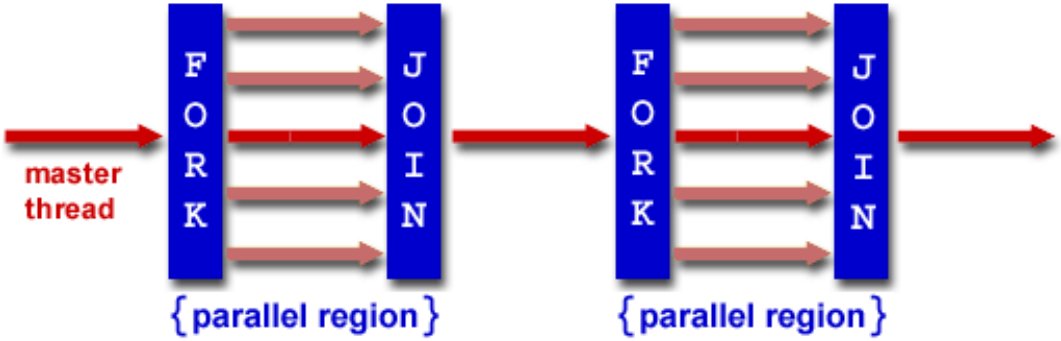
- Durch explizite Compileranweisungen werden Programmsektionen in Threads (“team”) aufgeteilt.
- Der Speicherbereich wird “geshared”
- Verschachtelte Parallelisierung von der API möglich



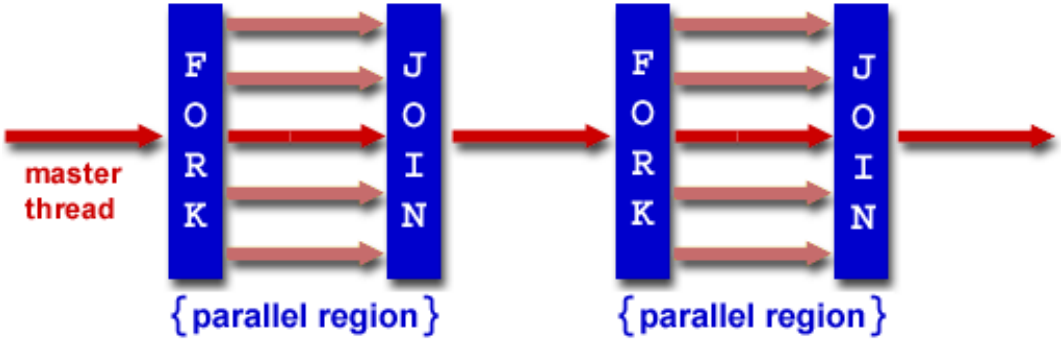
- Durch explizite Compileranweisungen werden Programmsektionen in Threads (“team”) aufgeteilt.
- Der Speicherbereich wird “geshared”
- Verschachtelte Parallelisierung von der API möglich
- Dynamische Erhöhung/Reduzierung der Anzahl der Threads



- Durch explizite Compileranweisungen werden Programmsektionen in Threads (“team”) aufgeteilt.
- Der Speicherbereich wird “geshared”
- Verschachtelte Parallelisierung von der API möglich
- Dynamische Erhöhung/Reduzierung der Anzahl der Threads
- KEINE explizite Unterstützung von I/O



- Durch explizite Compileranweisungen werden Programmsektionen in Threads (“team”) aufgeteilt.
- Der Speicherbereich wird “geshared”
- Verschachtelte Parallelisierung von der API möglich
- Dynamische Erhöhung/Reduzierung der Anzahl der Threads
- KEINE explizite Unterstützung von I/O
- Threads können ihre Daten “cachen”. Bei geteilten Variablen muss ggf. explizit “geflushed” werden



- Durch explizite Compileranweisungen werden Programmsektionen in Threads (“team”) aufgeteilt.
- Der Speicherbereich wird “geshared”
- Verschachtelte Parallelisierung von der API möglich
- Dynamische Erhöhung/Reduzierung der Anzahl der Threads
- KEINE explizite Unterstützung von I/O
- Threads können ihre Daten “cachen”. Bei geteilten Variablen muss ggf. explizit “geflushed” werden

```
#pragma omp <Direktive> [Klausel, ...  ]
```

- parallel – erstellt Team von Threads
- for – verteilt Schleife aufs Team
- sections – leitet Block mit Unteraufgaben ein
- single – Block für einen Thread
- parallel for – Kombi parallel & for
- task – Erstellen von tasks

```
#pragma omp <Direktive> [Klausel, ...  ]
```

- master – Block für Master-Thread
- critical – Block für einen Thread gleichzeitig
- barrier – Barriere für alle Threads
- taskwait – Warte auf beenden der tasks
- atomic – garantiere “Atomare” Speicherbehandlung
- flush – Schreibe Thread-Variablen
- ordered – Durchlaufe Block in Schleife sortiert
- threadprivate – Kopiere Variablen in für T. private

```
#pragma omp <Direktive> [Klausel, ...  ]
```

Klauseln:

- shared (list)
- private (list)
- firstprivate (list) – Initialisierung aller private
- lastprivate (list) – Update nach Ende der Region
- reduction (operator:list) – Update nach Ende der Region nach Regel

```
int main (int argc, char** argv)
{
  int a[5],i,x=1;
  #pragma omp parallel for shared (x)
  for (i=0; i < 5; i++)
    {
     printf ("Schritt %d\n",i);
     sleep (1);
     a[i] = x = x+x;
    }
  for (i=0;i<5;printf ("a[%d] = %d\n",i,a[i++]));
  return (EXIT_SUCCESS);
}
```

```
mschoepf@caesar$ gcc -o zweitesbeispiel-omp -fopenmp
zweitesbeispiel.c
mschoepf@caesar$ gcc -o zweitesbeispiel-serial zweitesbeispiel.c
mschoepf@caesar$ time ./zweitesbeispiel-serial
Schritt 0
Schritt 1
Schritt 2
Schritt 3
Schritt 4
a[1] = 2
a[2] = 4
a[3] = 8
a[4] = 16
a[5] = 32
real 0m5.001s
```

```
mschoepf@caesar$ time ./zweitesbeispiel-omp
Schritt 0
Schritt 2
Schritt 4
Schritt 1
Schritt 3
a[1] = 2
a[2] = 16
a[3] = 4
a[4] = 32
a[5] = 8
real 0m2.002s
```

```
int main(int argc, char** argv)
{
  int id,i;
  omp_set_num_threads(4);
  #pragma omp parallel for private(id)
  for (i = 0; i < 4; ++i)
  {
    id = omp_get_thread_num();
    printf("Hello World from thread %d\n", id);
    #pragma omp barrier
    if (id == 0)
      printf("There are %d threads\n",
      omp_get_num_threads());
  }
}
```

```
mschoepf@caesar: $ gcc -o drittesbeispiel -fopenmp
drittesbeispiel.c
mschoepf@caesar: $ ./drittesbeispiel
Hello World from thread 2
Hello World from thread 1
Hello World from thread 3
Hello World from thread 0
There are 4 threads
```

Was OpenMP NICHT kann:

- Auf verteilten Speicher zugreifen
- Speichereffizienz garantieren

Was OpenMP NICHT kann:

- Auf verteilten Speicher zugreifen
- Speichereffizienz garantieren
- Garantie gleichen Verhaltens verschiedener Implementationen

Was OpenMP NICHT kann:

- Auf verteilten Speicher zugreifen
- Speichereffizienz garantieren
- Garantie gleichen Verhaltens verschiedener Implementationen
- Prüfen von Datenkonsistenz, Abhängigkeiten, deadlocks, race conditions

Was OpenMP NICHT kann:

- Auf verteilten Speicher zugreifen
- Speichereffizienz garantieren
- Garantie gleichen Verhaltens verschiedener Implementationen
- Prüfen von Datenkonsistenz, Abhängigkeiten, deadlocks, race conditions
- Synchronisierte I/O

Was OpenMP NICHT kann:

- Auf verteilten Speicher zugreifen
- Speichereffizienz garantieren
- Garantie gleichen Verhaltens verschiedener Implementationen
- Prüfen von Datenkonsistenz, Abhängigkeiten, deadlocks, race conditions
- Synchronisierte I/O

- Message Passing Interface - Standard zur Nachrichtenübertragung
- Ziele: Effizienz, Portabilität, Flexibilität

- Message Passing Interface - Standard zur Nachrichtenübertragung
- Ziele: Effizienz, Portabilität, Flexibilität
- MPI Forum: IBM, Intel, HP, nCUBE's Vertex, p4, Zipcode, Chimp, PVM, Chameleon, PICL, ...

- Message Passing Interface - Standard zur Nachrichtenübertragung
- Ziele: Effizienz, Portabilität, Flexibilität
- MPI Forum: IBM, Intel, HP, nCUBE's Vertex, p4, Zipcode, Chimp, PVM, Chameleon, PICL, ...
- Seit 1995: MPI 1.0, danach MPI 1.1, 1.2 und 2.0

- Message Passing Interface - Standard zur Nachrichtenübertragung
- Ziele: Effizienz, Portabilität, Flexibilität
- MPI Forum: IBM, Intel, HP, nCUBE's Vertex, p4, Zipcode, Chimp, PVM, Chameleon, PICL, ...
- Seit 1995: MPI 1.0, danach MPI 1.1, 1.2 und 2.0
- Freie Implementationen verfügbar: lam-mpi, MPICH, OpenMPI (lam-mpi Nachfolger) ...

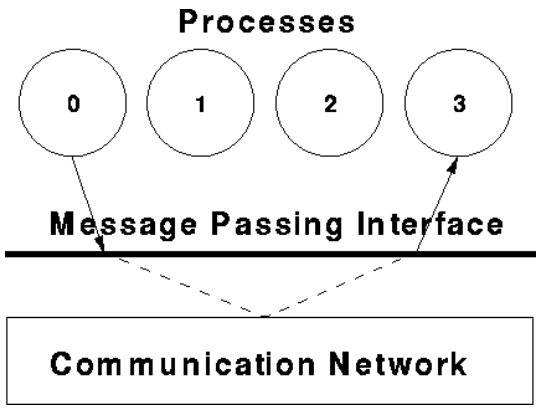
- Message Passing Interface - Standard zur Nachrichtenübertragung
- Ziele: Effizienz, Portabilität, Flexibilität
- MPI Forum: IBM, Intel, HP, nCUBE's Vertex, p4, Zipcode, Chimp, PVM, Chameleon, PICL, ...
- Seit 1995: MPI 1.0, danach MPI 1.1, 1.2 und 2.0
- Freie Implementationen verfügbar: lam-mpi, MPICH, OpenMPI (lam-mpi Nachfolger) ...
- Eine API mit Vorwärtskompabilität

- Message Passing Interface - Standard zur Nachrichtenübertragung
- Ziele: Effizienz, Portabilität, Flexibilität
- MPI Forum: IBM, Intel, HP, nCUBE's Vertex, p4, Zipcode, Chimp, PVM, Chameleon, PICL, ...
- Seit 1995: MPI 1.0, danach MPI 1.1, 1.2 und 2.0
- Freie Implementationen verfügbar: lam-mpi, MPICH, OpenMPI (lam-mpi Nachfolger) ...
- Eine API mit Vorwärtskompabilität
- API gesichert durch Gremium (MPI Forum)

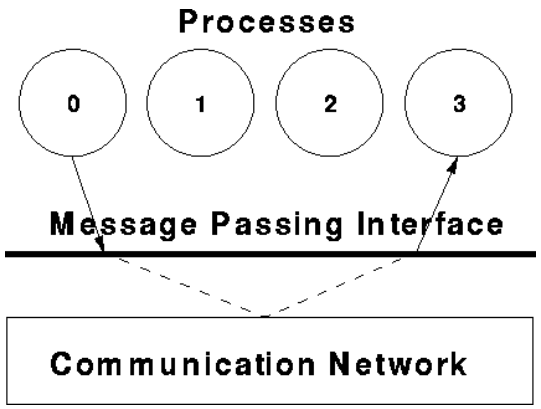
- Message Passing Interface - Standard zur Nachrichtenübertragung
- Ziele: Effizienz, Portabilität, Flexibilität
- MPI Forum: IBM, Intel, HP, nCUBE's Vertex, p4, Zipcode, Chimp, PVM, Chameleon, PICL, ...
- Seit 1995: MPI 1.0, danach MPI 1.1, 1.2 und 2.0
- Freie Implementationen verfügbar: lam-mpi, MPICH, OpenMPI (lam-mpi Nachfolger) ...
- Eine API mit Vorwärtskompabilität
- API gesichert durch Gremium (MPI Forum)
- Damit Sourcecode-kompabilität

- Message Passing Interface - Standard zur Nachrichtenübertragung
- Ziele: Effizienz, Portabilität, Flexibilität
- MPI Forum: IBM, Intel, HP, nCUBE's Vertex, p4, Zipcode, Chimp, PVM, Chameleon, PICL, ...
- Seit 1995: MPI 1.0, danach MPI 1.1, 1.2 und 2.0
- Freie Implementationen verfügbar: lam-mpi, MPICH, OpenMPI (lam-mpi Nachfolger) ...
- Eine API mit Vorwärtskompabilität
- API gesichert durch Gremium (MPI Forum)
- Damit Sourcecode-kompabilität
- C, C++, Fortran

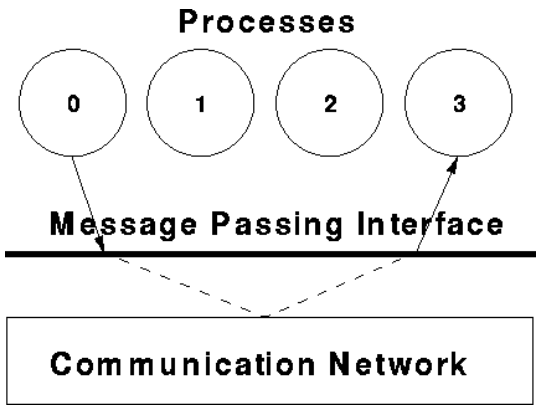
- Message Passing Interface - Standard zur Nachrichtenübertragung
- Ziele: Effizienz, Portabilität, Flexibilität
- MPI Forum: IBM, Intel, HP, nCUBE's Vertex, p4, Zipcode, Chimp, PVM, Chameleon, PICL, ...
- Seit 1995: MPI 1.0, danach MPI 1.1, 1.2 und 2.0
- Freie Implementationen verfügbar: lam-mpi, MPICH, OpenMPI (lam-mpi Nachfolger) ...
- Eine API mit Vorwärtskompabilität
- API gesichert durch Gremium (MPI Forum)
- Damit Sourcecode-kompabilität
- C, C++, Fortran



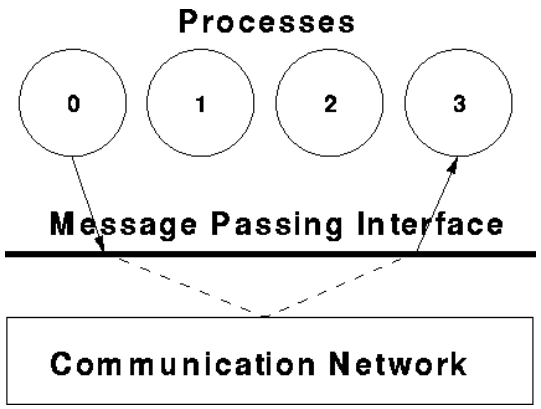
- Kommunikation auf Prozessebene
- Shared Memory, TCP, ...
- Implementationen übernehmen:



- Kommunikation auf Prozessebene
- Shared Memory, TCP, ...
- Implementationen übernehmen:
  - Verteilen der Prozesse (bspw. mit rsh/ssh)



- Kommunikation auf Prozessebene
- Shared Memory, TCP, ...
- Implementationen übernehmen:
  - Verteilen der Prozesse (bspw. mit rsh/ssh)
  - Wahl des Kommunikationsmediums



- Kommunikation auf Prozessebene
- Shared Memory, TCP, ...
- Implementationen übernehmen:
  - Verteilen der Prozesse (bspw. mit rsh/ssh)
  - Wahl des Kommunikationsmediums

Vorbedingungen

- Erster MPI-Aufruf: `MPI_Init ()`
- Bewirkt u.a. Erstellung von `MPI_COMM_WORLD`

Vorbedingungen

- Erster MPI-Aufruf: `MPI_Init ()`
- Bewirkt u.a. Erstellung von `MPI_COMM_WORLD`
- Letzter (MPI)-Aufruf `MPI_Finalize ()`

Vorbedingungen

- Erster MPI-Aufruf: `MPI_Init ()`
- Bewirkt u.a. Erstellung von `MPI_COMM_WORLD`
- Letzter (MPI)-Aufruf `MPI_Finalize ()`
- MPI-Programme können mit extra Wrappern gebaut werden (`mpicc, mpicxx, mpif77`).

Vorbedingungen

- Erster MPI-Aufruf: `MPI_Init ()`
- Bewirkt u.a. Erstellung von `MPI_COMM_WORLD`
- Letzter (MPI)-Aufruf `MPI_Finalize ()`
- MPI-Programme können mit extra Wrappern gebaut werden (`mpicc, mpicxx, mpif77`).
- MPI-Programme werden mit einem Helfer verteilt und gestartet: `mpirun -np 4 -hostfile hostfile.txt executable`

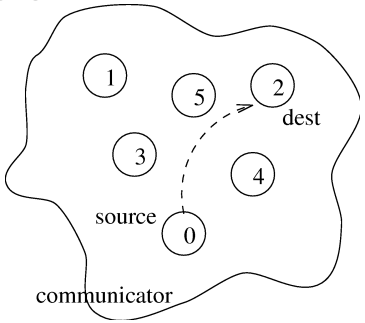
Vorbedingungen

- Erster MPI-Aufruf: `MPI_Init ()`
- Bewirkt u.a. Erstellung von `MPI_COMM_WORLD`
- Letzter (MPI)-Aufruf `MPI_Finalize ()`
- MPI-Programme können mit extra Wrappern gebaut werden (`mpicc, mpicxx, mpif77`).
- MPI-Programme werden mit einem Helfer verteilt und gestartet: `mpirun -np 4 -hostfile hostfile.txt executable`

Erstes Beispiel

```
#include <mpi.h>
int main (int argc, char** argv)
{
  int ierror, rank, size;
  ierror = MPI_Init (&argc,&argv);
  if (ierror != MPI_SUCCESS)
    schade ();
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Finalize();
  return (EXIT_SUCCESS);
}
```

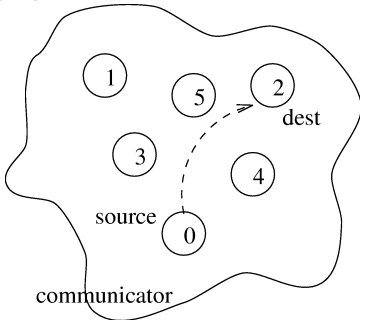
Point-to-Point Kommunikation



synchron Sender blockiert, bis Empfänger Nachricht erhalten hat

asynchron Sender schickt Nachricht ohne auf die Rückmeldung des Empfängers zu warten

Point-to-Point Kommunikation

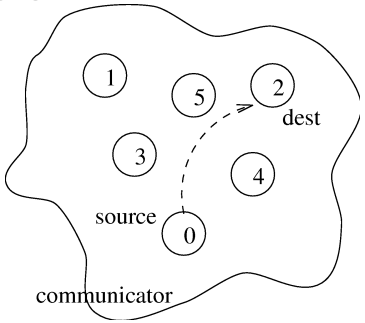


synchron Sender blockiert, bis Empfänger Nachricht erhalten hat

asynchron Sender schickt Nachricht ohne auf die Rückmeldung des Empfängers zu warten

Wichtig In MPI braucht jedes SEND ein entsprechendes RECEIVE.

Point-to-Point Kommunikation



synchron Sender blockiert, bis Empfänger Nachricht erhalten hat

asynchron Sender schickt Nachricht ohne auf die Rückmeldung des Empfängers zu warten

Wichtig In MPI braucht jedes SEND ein entsprechendes RECEIVE.

Send (blocking)

- `MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm))` (Standard: Entweder buffered oder synchron)
- `MPI_Ssend (...)` (Synchron)

Send (blocking)

- `MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm))` (Standard: Entweder buffered oder synchron)
- `MPI_Ssend (...)` (Synchron)
- `MPI_Bsend (...)` (Buffered)

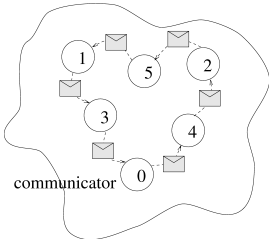
Send (blocking)

- `MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm))` (Standard: Entweder buffered oder synchron)
- `MPI_Ssend (...)` (Synchron)
- `MPI_Bsend (...)` (Buffered)
- `MPI_Rsend (...)` (Ready)

Send (blocking)

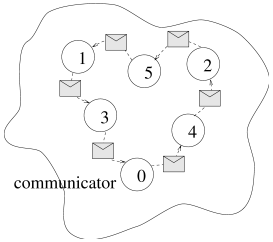
- `MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm))` (Standard: Entweder buffered oder synchron)
- `MPI_Ssend (...)` (Synchron)
- `MPI_Bsend (...)` (Buffered)
- `MPI_Rsend (...)` (Ready)

Send (non-blocking)



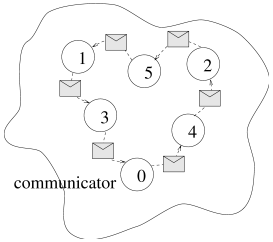
- `MPI_Isend (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
- `MPI_Issend (...)` (Synchron)

Send (non-blocking)



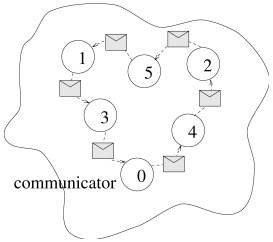
- `MPI_Isend (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
- `MPI_Issend (...)` (Synchron)
- `MPI_Ibsend (...)` (Buffered)

Send (non-blocking)



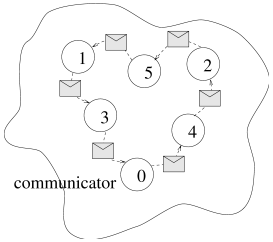
- `MPI_Isend (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
- `MPI_Issend (...)` (Synchron)
- `MPI_Ibsend (...)` (Buffered)
- `MPI_Irsend (...)` (Ready)

Send (non-blocking)



- `MPI_Isend (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
- `MPI_Issend (...)` (Synchron)
- `MPI_Ibsend (...)` (Buffered)
- `MPI_Irsend (...)` (Ready)
- Statusabfrage mit `MPI_Wait()` oder `MPI_Test()`

Send (non-blocking)



- `MPI_Isend (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
- `MPI_Issend (...)` (Synchron)
- `MPI_Ibsend (...)` (Buffered)
- `MPI_Irsend (...)` (Ready)
- Statusabfrage mit `MPI_Wait()` oder `MPI_Test()`

Receive:

- `MPI_Recv ()` (blocking)
- `MPI_Irecv ()` (non-blocking)

Receive:

- `MPI_Recv ()` (blocking)
- `MPI_Irecv ()` (non-blocking)
- Es können beide Varianten gemischt benutz werden.

Receive:

- `MPI_Recv ()` (blocking)
- `MPI_Irecv ()` (non-blocking)
- Es können beide Varianten gemischt benutz werden.
- `MPI_Waitany ()`
- `MPI_Testany ()`
- `MPI_Waitall ()`
- `MPI_Testall ()`
- `MPI_Waitsome ()`
- `MPI_Testsome ()`

Receive:

- `MPI_Recv ()` (blocking)
- `MPI_Irecv ()` (non-blocking)
- Es können beide Varianten gemischt benutz werden.
- `MPI_Waitany ()`
- `MPI_Testany ()`
- `MPI_Waitall ()`
- `MPI_Testall ()`
- `MPI_Waitsome ()`
- `MPI_Testsome ()`

```
#define to_right 201
#define to_left  102
int main (int argc, char *argv[])
{
  int right, left, rank, my_rank, size, other, i, sum;
  MPI_Status  send_status, recv_status;
  MPI_Request request;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  right = (my_rank + 1) % size;
  left = (my_rank - 1) % size;
  sum = 0;
  rank = my_rank;
  for( i = 0; i < size; i++) {
    MPI_Issend(&rank, 1, MPI_INT, right, to_right ,MPI_COMM_WORLD,
    &request);
    MPI_Recv(&other, 1, MPI_INT, left, to_right, MPI_COMM_WORLD,
    &recv_status);
    MPI_Wait(&request, &send_status);
    sum += other;
    rank = other;
  }
  printf ("PE␣%d:\t␣Sum␣=␣%d\n", rank, sum);
  MPI_Finalize();
  return (EXIT_SUCCESS);
}
```

- Datentypen: Basic, Vektoren, Structs, ...
- Topologien: Prozesse bspw. Carthesisch anordnenbar bishin zu allgemeinen Graphen
- `MPI_Barrier()` analog zu OpenMP

Deutliche Unterschiede zu Point-to-Point:

- Immer über einen Communicator (z.B. MPI_COMM_WORLD)
- Alle Prozesse müssen kommunizieren
- Mit oder ohne Synchronisation
- Nur “blocking” möglich
- Keine “Tags”
- Receive buffer müssen exakt die richtige Größe haben
- `MPI_Bcast()`, `MPI_Scatter()`, `MPI_Gather`, `MPI_Reduce()`, `MPI_All_to_all()`

- 2003: 16 PCs unter MPI brachten 0.5 GFLOPS (AG NI)
- 2009: 1 QuadCore bringt 3.2 GFLOPS (OpenMP) bzw. 2.0 (MPI) (vgl. 1.2 GFLOPS Single Thread)
- Mittels OpenMP einfache Parallelisierung (parallel for / sections)
- Mittel MPI bspw. durch Master-Worker Aufteilung Nutzung großer Rechenrescourcen
- BTW: OpenMP & MPI können zusammen benutzt werden

- Writing Message-Passing Parallel Programs with MPI - University of Edinburgh - http://www.lrz-muenchen.de/services/software/parallel/mpi/epcc-course/
- MPI-Primer / Developing with LAM-MPI - University of Ohio - http://www.lam-mpi.org
- Cluster Quick Start Version 0.1 - Douglas Eadline - http://www.xtreme-machines.com/x-cluster-qs.html
- Message Passing Interface (MPI) FAQ - comp.parallel.mpi - http://www.faqs.org/faqs/mpi-faq/
- MPI Forum - http://www.mpi-forum.org
- Wikipedia
- www.openmp.org