Marcell Kehmstedt
m.kehmstedt@tu-bs.de

Message Passing Interface

Das **Message Passing Interface** (MPI) ist ein Standard, der einen Nachrichtenaustausch für parallele Prozesse beschreibt. Die Prozesse laufen typischerweise verteilt in einem über ein Netzwerk zusammengeschaltetes Computersystem (wie dem AKB-Cluster). Es abstrahiert von der zugrundeliegenden Netzwerkkommunikation und organisiert die Zuteilung zu den Prozessoren. Es sind eine Reihe von verschiedenen Implementierungen verfügbar, die aber alle dem Standard genügen und zumindest die darin beschriebene Schnittstelle exportieren. Nachfolgend werden einige der wesentlichen Funktionen dieser Schnittstelle kurz vorgestellt.

MPI im Programm verwenden

Alle Symbole im Zusammenhang mit MPI sind in der Headerdatei `mpi.h` deklariert.
Bevor ein Programm auf irgendeine Funktionalität von MPI zugreifen kann, muss zunächst

```
MPI_Init(&argc, &argv);
```

aufgerufen werden.
Entsprechend wird nach der letzten Verwendung einer MPI Funktion aufgeräumt:

```
MPI_Finalize();
```

Prozessorganisation

Jeder Prozess wird von MPI unter einer ID (rank) geführt. Diese IDs sind gewöhnliche Integer und werden von 0 aufsteigend den Prozessen zugeordnet.
Die ID des jeweiligen Prozesses erfragt man mittels:

```
int myRank;
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
```

`MPI_COMM_WORLD` ist ein sog. Kommunikator, welcher an dieser Stelle (noch) nicht näher betrachtet werden soll.
Neben der ID des jeweiligen Prozesses kann auch die Anzahl aller parallelen Prozesse des selben Jobs erfragt werden:

```
int numProcesses;
MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
```

Programmablauforganisation

Generell führen alle parallelen Prozesse dasselbe Programm aus. Da man typischerweise nicht einfach mehrfach dasselbe berechnen will, sondern eine Kooperation der Prozesse zur gemeinsamen Lösung einer großen Aufgabe erreichen will, ist es erforderlich, dass die Prozesse sich selbst kennen und von den anderen Prozessen unterscheiden können. Häufig nimmt dabei ein Prozess eine Sonderstellung ein (meist derjenige mit ID 0) und übernimmt als sog. „master process“ die Zuteilung der Aufgaben und die Bereitstellung der Berechnungsantwort. Eine typische Programmstruktur sieht daher wie folgt aus:

Marcell Kehmstedt
m.kehmstedt@tu-bs.de

```
if (myRank == 0) // master process
{ /* organize computation */ }
else if (myRank == 1) // another process (optional)
{ /* doing something special */ }
else // all the other processes
{ /* doing all the same */}
```

Punkt-zu-Punkt Kommunikation

Es gibt mehrere Arten der Kommunikation mit MPI. Eine davon ist die Punkt-zu-Punkt Kommunikation, bei der genau **ein** Prozess genau **einem** anderen Prozess eine Nachricht schickt. Die entsprechenden Operationen unter MPI sind:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm)
```

- buf: Zeiger auf den Sendepuffer
- count: Anzahl der Elemente im Sendepuffer (nicht Anzahl an Bytes)
- datatype: Datentyp der Elemente im Sendepuffer (z.B. `MPI_DOUBLE`)
- dest: ID (rank) des Zielprozesses
- tag: Nachrichtenmarkierung (wie ein farbiger Briefumschlag)
- comm: Kommunikator (für den Anfang zunächst `MPI_COMM_WORLD`)

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source
    , int tag, MPI_Comm comm, MPI_Status *status)
```

- source: ID (rank) des Prozesses, von dem eine Nachricht erwartet wird
  (mit `MPI_ANY_SOURCE` wird von einem beliebigen Prozess empfangen)
- tag: erwartete Markierung der Nachricht
  (mit `MPI_ANY_TAG` wird eine Nachricht mit beliebiger Markierung empfangen)
- status: Zeiger auf Statusstruktur
  (in dieser Struktur werden Informationen über die Nachricht abgelegt)

Beide genannten Operationen sind **blockierend** und **asynchron**. Das bedeutet:
`MPI_Send`

- blockiert bis die Nachricht verschickt bzw. zwischengepuffert wurde.
- kann aufgerufen werden, bevor ein `MPI_Recv` die Nachricht empfängt.

`MPI_Recv`

- blockiert bis eine passende Nachricht empfangen wurde.
- kann aufgerufen werden, bevor ein `MPI_Send` eine Nachricht verschickt.

Marcell Kehmstedt
m.kehmstedt@tu-bs.de

Globale Kommunikation

Neben der Punkt-zu-Punkt Kommunikation gibt es auch Möglichkeiten mit mehreren Prozessen gleichzeitig zu kommunizieren. An dieser Stelle soll zunächst nur die eine nachfolgende Operation zur globalen Kommunikation vorgestellt werden, welche dazu dient, Daten eines Prozesses auf alle anderen Prozesse zu replizieren.

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root,
    MPI_Comm comm)
```

Der Prozess mit der ID `root` stellt in `buf` die Daten bereit, die per Broadcast auf alle anderen Prozesse repliziert werden sollen. Alle anderen Prozesse halten ihrerseits einen ausreichend dimensionierten Empfangspuffer bereit, in dem sie die Daten von `root` empfangen wollen. `count` muss somit bei allen Prozessen gleich sein. Ebenso sollen auch die übrigen Parameter bei allen Prozessen gleich sein. Nach der Rückkehr der Funktion befinden sich in den angegebenen Puffern `buf` aller Prozesse dieselben Daten.

Parallele Prozesse mit Parastation

Auf dem AKB-Cluster ist das Batchsystem Parastation installiert. Eine kurze Einführung in die Job- und Modulverwaltung wurde bereits auf dem ersten Aufgabenblatt behandelt. Es folgen einige zusätzliche Erläuterungen zu der Verwendung von MPI Jobs mit Parastation.

Kompilieren von MPI Programmen

Um MPI Programme übersetzen und auch starten zu können, muss zuvor das Modul `lib/parastation` geladen worden sein:

```
module load lib/parastation
```

Ein C-Programm kann anschließend mit dem Kommando `mpicc` übersetzt werden. `mpicc` ist ein Wrapper, der `icc` aufruft und um die MPI Bibliotheken und Header ergänzt. Beispiel:

```
mpicc -o program exercise1.c gettime.c
```

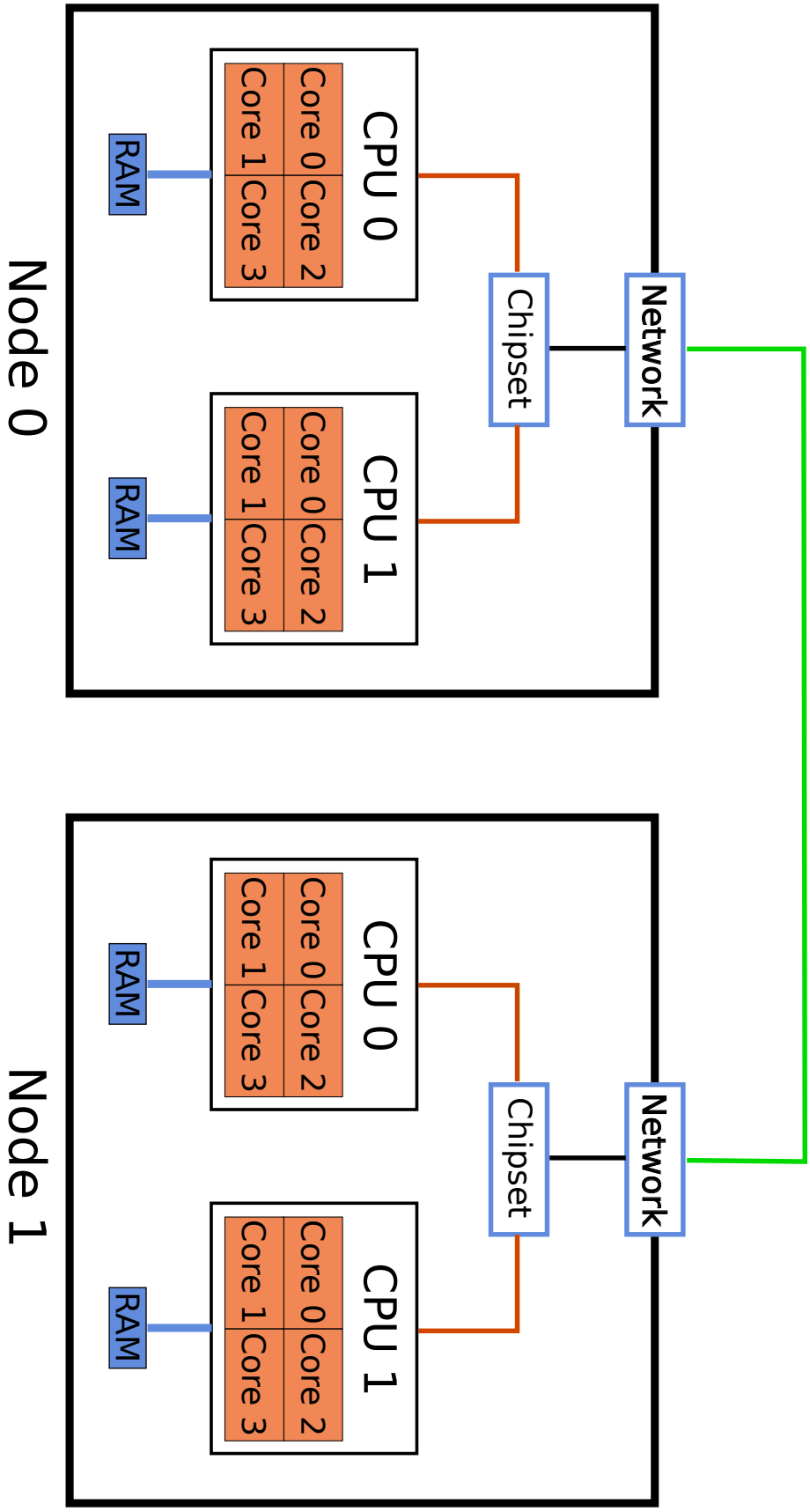
Anfordern paralleler Jobs

Das Batchsystem kennt unterschiedliche Queues, denen Jobs zur Abarbeitung zugeteilt werden. Jede Queue hat ihre eigenen Ressourcenlimits. So gibt es auf dem AKB-Cluster u.a. die beiden Queues `Serial` und `Parallel`. Parastation übernimmt die Zuordnung automatisch anhand der im Jobfile angeforderten Ressourcen. Es folgt ein einfaches Jobfile, das die Ausführung eines MPI-Programms `program` mit insgesamt 4 Prozessen anfordert. Die 4 Prozesse sollen sich auf 2 Knoten (nodes) mit jeweils zwei Prozessoren pro Knoten (ppn) verteilen. Mit Prozessor ist dabei ein eigenständiger Rechenkern gemeint. Verfügt ein Knoten z.B. über zwei Sockets mit je einem Dualcore-Prozessor, so zählen diese zusammen als vier Prozessoren für den jeweiligen Knoten.

```
#!/bin/sh
#PBS -l nodes=2:ppn=2
#PBS -o job.out
#PBS -e job.err
mpiexec -np=4 program
```

Marcell Kehmstedt
m.kehmstedt@tu-bs.de

Abbildung 1: Beispielarchitektur eines Systems mit verteiltem Speicher



Marcell Kehmstedt
m.kehmstedt@tu-bs.de

Aufgabe 1 *Hello MPI World*

Schreiben Sie ein MPI-Programm, das auf mehr als einem Prozess den folgenden Text ausgibt:

```
Hello World! Greetings from process <pid>.
```

<pid> ist dabei durch die ID des jeweiligen Prozesses zu ersetzen.

(2 Punkte)

Aufgabe 2 Ping-Pong

Schreiben Sie ein Programm, das eine Nachricht von einem Prozess A zu einem Prozess B und wieder zurück sendet („Ping-Pong“).

a) Messen Sie die mittlere Latenz für ein Ping-Pong zwischen zwei Prozessen auf
   1. demselben Knoten.
   2. zwei verschiedenen Knoten.

b) Messen Sie die mittlere Bandbreite einer Nachrichtenübermittlung zwischen zwei Prozessen wieder auf
   1. demselben Knoten.
   2. zwei verschiedenen Knoten.

   Beachten Sie den Einfluss der zuvor bestimmten Latenz auf Ihre Bandbreitenmessung. Führen Sie beide Messungen jeweils für verschiedene Nachrichtenlängen zwischen 4 und 100 Kilobyte in Schritten von 4 Kilobyte durch und erstellen Sie je einen Plot.

(10 Punkte)

Aufgabe 3 Summation von Vektorelementen

Schreiben Sie ein Programm, das die Summe aller Elemente eines Vektors x mithilfe von MPI in parallelen Prozessen berechnet und das Ergebnis anschließend in allen Prozessen bereitstellt. Die Summenbildung lässt sich auf die Berechnung von Teilsummen zurückführen, die von den P Prozessen in parallel berechnet werden können. Es ergeben sich Teilergebnisse, die zur Gesamtsumme final zu summieren sind.
Dazu:

- Initialisieren Sie einen Vektor x der Länge N im Prozess mit der ID 0 (master process) mit Zufallszahlen und replizieren Sie ihn anschließend über alle Prozesse.
- Berechnen Sie geeignete Teilsummen in allen Prozessen.
- Verteilen Sie die Teilergebnisse aller Prozesse an jeweils alle anderen Prozesse, so dass jeder Prozess schließlich über alle Teilergebnisse verfügt.
- Summieren Sie in allen Prozessen die Teilergebnisse zum Endergebnis.

a) Messen und plotten Sie die für die Summation benötigte Zeit (ohne Initialisierung und Replikation) jeweils für $N = 10^7$ und $P = 2, \ldots, 8$.

b) Messen Sie die Zeit, die eine serielle Implementierung benötigt und vergleichen Sie diese mit den zuvor bestimmten Zeiten. Berechnen Sie jeweils den Speedup.

(10 Punkte)