Alexander Vondrous, Britta Nestler, Michael Selzer,
Johannes Hötzer, Walter Werner, Constantin Heisler
Fachgebiet Informatik, Hochschule Karlsruhe - Technik und Wirtschaft

Kurzanleitung
High Performance Computing im WS 16/17

Arbeitsumgebung einrichten

1. Loggen Sie sich mit ssh auf dem ProStudium Cluster an (Öffnen Sie dazu ein Terminal):

   ```
   ssh labor-hpc<GR-NR>@iwi-i-hpc01.hs-karlsruhe.de
   ```

   (Ersetzen Sie <GR-NR> durch Ihre Gruppennummer.)

2. Kopieren Sie den Übungscode auf das home Verzeichnis des ProStudium Clusters:

   ```
   cp -r ../labor-gruppe/lab1 .
   ```

3. Wechseln Sie mit dem Befehl "cd" (change directory) in das lab1 Verzeichnis.

4. Kompilieren Sie den Quelltext so wie im Übungsblatt angegeben.

5. Für das Editieren des Quelcodes oder das Betrachen der Ergebnise in Paraview, mounten Sie ihr ProStudium Cluster Verzeichnis (Öffnen Sie dazu ein neues Terminal):

   ```
   mkdir mnt
   sshfs -o idmap=user \
         labor-hpc<GR-NR>@iwi-i-hpc01.hs-karlsruhe.de: mnt
   ```

   (Ersetzen Sie <GR-NR> durch Ihre Gruppennummer.) Der zweite Befehl kann in einer Zeile geschrieben werden ohne das Backslash \. Das "mnt" Verzeichnis sollte nun im Homeverzeichnis sichtbar sein.

Arbeiten mit dem ProStudium Cluster

Für den reibungslosen Ablauf der Programme und der Zuordnung der Rechenressourcen wird SLURM (Simple Linux Utility for Resource Management) verwendet. Es stellt Benutzern und Administratoren Schnittstellen zum Verwalten und Verwenden der Cluster Ressourcen zur Verfügung. Aus Benutzersicht sind die häufigsten Anwendungen Starten, Beenden und Beobachten von Jobs.

sinfo - Status der Rechenknoten

Bevor ein Job gestartet wird, kann mit sinfo überprüft werden, wie viele Rechenknoten momentan belegt, frei oder ni cht erreichbar sind. Freie Knoten befinden sich im Zustand "`idle`" und belegte Knoten im Zustand "`alloc`".

Beispielausgabe

```
PARTITION AVAIL  TIMELIMIT NODES  STATE NODELIST
producti*    up   infinite     2   fail cnode0_15,cnode1_01
producti*    up   infinite    25  alloc cnode0_[01-02,04-11,13-1...
producti*    up   infinite    17   idle cnode0_[12,17],cnode1_[0...
producti*    up   infinite     3   down cnode1_[07,15-16]
```

25 Knoten sind bereits belegt, 17 Knoten stehen noch zur Verfügung und 5 Knoten sind nicht erreichbar.

sbatch - Starten eines Jobs

Wenn die Anzahl der zu verwendenden Rechenknoten feststeht, dann kann ein Job mit "`sbatch`" gestartet werden. Jeder Job erhält eine eindeutige Job ID nach dem Ausführen von `sbatch`. Sollten alle Ressourcen belegt sein, wird der Job in eine Warteschlange eingereiht. Auf dem ProStudium Cluster entspricht ein Rechenknoten 8 CPUs.

Beispiel

```
sbatch -N 2 hpc batch hello
sbatch: Submitted batch job 8586
```

Das Programm "`hello`" wird auf 2 Rechenknoten (= 16 CPUs) ausgeführt und erhält die ID 8586. Die Ausgabe wird in die Textdatei "`slurm-8586.out`" (`slurm-JOBID.out`) geschrieben. Das Skript "hpc batch" dient SLURM und den Administartoren zur leichteren Verwaltung der Jobs.

squeue - Jobs Status abfragen

Programme mit hoher Rechenzeit benötigen selbst auf einem Hochleistungsrechner mehrere Tage, wenn die Ressourcen zur Verfügung stehen. Die Ausgabe von "`squeue`" zeigt jeden Job, der sich gerade in der Warteschlange, in der Ausführung oder dem Aufräumvorgang befindet.

Beispielausgabe

```
JOBID PARTITION     NAME     USER  ST       TIME  NODES NODELIST(REASON)
 8584 productio      hpc hase0001  PD       0:00     30 (Resources)
 8014 productio      hpc esel0001   R 17-23:32:02      1 cnode0_16
 8531 productio      hpc anna0001   R 3-18:37:55     12 cnode0_[01-...
 8574 productio      hpc jojo0001   R 2-14:11:38     12 cnode0_[09-...
```

Der Job des Benutzers `hase0001` wartet auf Ressourcen (30 Knoten) und der Status ist PD (pending). Alle anderen Jobs werden ausgeführt und befinden sich im Status R (running).

scancel - Job beenden

Befindet sich ein Job in der Warteschlange oder in der Ausführung und es wird festgestellt, dass ein Fehler in der Konfiguration vorliegt oder das bisherige einen Abbruch rechtfertig, so kann mit "`scancel`" der Job beendet oder aus der Warteschlange entfernt werden. Es wird die eindeutige Job ID zum Entfernen des Jobs benötigt und es sollte nicht möglich sein Jobs von anderen Benutzern zu entfernen.

Beispiel

```
scancel 2342
```

Der Job mit der ID 2342 wird aus dem System entfernt.

OpenMP Zusammenfassung

Grundstruktur

Für die Steuerung der parallelen Verarbeitung werden Direktiven eingesetzt, die auf Codeblöcke angewendet werden. Der Aufbau beginnt in C und C++ immer mit "`#pragma omp`" und wird von einer Direktive wie z.B. "`parallel`" gefolgt. Eine Direktive kann mit mehreren Parametern (clauses) konfiguriert werden.

Beispiel (ohne Parameter)

```
#pragma omp parallel
{
  printf(”Hello World“);
}
```

Jeder Thread gibt auf der Konsole "Hello World" aus.

parallel Direktive - Parallele Ausführung

Mit der Direktive "`parallel`" wird angegeben, dass der Anweisungsblock parallel ausgeführt wird. Jeder Thread führt die gleichen Anweisungen aus sofern keine explizite Arbeitszuweisung stattfindet.

Mit den Parametern "`private(Liste mit Variablen)`" und "`shared (Liste mit Variablen)`" können die gemeinsamen und privaten Variablen angegeben werden.

Beispiel

```
int a;
int b;
int c;

a = 4;
#pragma omp parallel shared(a) private(b,c)
{
  c = 3;
  b = a + b;
  ...
}
```

Die Variable "`a`" wird von allen gemeinsam geteilt. Eine Änderung von "`a`" ist somit in allen Threads sichtbar. Änderungen von den privaten Variablen "`b`" und "`c`" sind nicht sichtbar für andere Threads.
VORSICHT! Es kann leicht zu fehlerhaften Verarbeitung kommen, wenn das Lesen und Schreiben von geteilten Variablen nicht durch entsprechende Mechanismen geschützt wird.

for Direktive - Parallele Schleife

Für die Verteilung von Arbeit innerhalb eines parallelen Bereiches (parallel Direktive) kann die "`for`" Direktive eingesetzt werden. Jeder Iterationsschritt oder eine Menge von Iterationsschritten wird auf die gegebenen Threads verteilt. Es ist zu beachten das die "`#pragma omp for`" Anweisung direkt vor der zur parallelisierenden

for-Schleife platziert wird.

Beispiel

```
int i;
int a[10];
int b[10];

#pragma omp parallel shared(a, b) private(i)
{
  #pragma omp for
  for (i = 0; i < 10; i++) {
    a[i] = b[i] + 3;
    ...
  }
}
```

Jeder Thread erhält einen Teil oder einen Block an Iterationen. Zu beachten ist, dass die Berechnungen in jeder Iteration voneinander unabhängig sein sollen.

Die Parameter "`schedule(static)`" und "`schedule(dynamic)`" steuern die Verteilung der Iterationen. Im statischen Fall (`static`) werden die Iterationen in festen Blöcken jedem Thread reihum (round-robin) zugeteilt. Bei dem dynamischen Fall (`dynamic`) werden die zu bearbeitenden Iterationen dynamisch zugeteilt.

critical Direktive - Kritischer Bereich

Ein kritischer Bereich kann in einem parallelen Block nur von einem Thread durchlaufen werden und ist ein Mechanismus um das fälschliche Überschreiben von Variablen zu vermeiden.

Beispiel

```
int a = 3;
int b = 4;

#pragma omp parallel shared(a, b)
{
  #pragma omp critical
  {
    a = a + b;
  }
}
```

OpenMP Funktionen

Innerhalb des parallen Block ist es notwendig die Thread ID und die Gesamtanzahl Threads zu bestimmen. Die folgenden Funktionen ermöglichen die Bestimmung.

```
int num_threads = omp_get_num_threads();
int id = omp_get_thread_num();
```

Sperrmechanismen

Für das Sperren stehen in OpenMP Locks zu Verfügung. Sie entsprechen einem Mutex und erlauben bzw. Sperren die Verarbeitung. Ein Lock kann sich in den zwei Zuständen "gesperrt" und "nicht gesperrt" befinden. Wird ein Lock durch einen Thread gesperrt, können andere Thread das Lock nicht passieren, bis das Lock entsperrt ist. Zum Sperren der Locks wird die Funktion '`omp_set_lock(...)`' und '`omp_unset_lock(...)`' verwendet.

Beispiel

```
omp_lock_t mutex;
omp_init_lock(&mutex);

#pragma omp parallel shared(mutex)
{
  omp_set_lock(&mutex);
  printf("Welcome to the critical section.\n");
  omp_unset_lock(&mutex);
}

omp_destroy_lock(&mutex);
```

MPI Zusammenfassung

Webseite mit Beschreibung aller MPI Funktionen http://mpi.deino.net/.

MPI - Initialisieren und Aufräumen

Funktion: `int MPI_Init(int* argc, char ***argv);`
Anwendung: `MPI_Init(&argc, &argv);`
Initialisiert die Ausführungsumgebung und synchronisiert alle Programmparameter, die auch an main(...) übergeben werden.

`int MPI_Finalize();`
Räumt die Ausführungsumgebung auf.

MPI - ID/Rang und Anzahl der Prozesse ermitteln

`MPI_Comm_size(MPI_comm comm, int *size);`
Gibt die Größe der Gruppe "comm" zurück. Die Gruppe, die standardmäßig alle Prozesse enthält wird adressiert mit MPI_COMM_WORLD.

`MPI_Comm_rank(MPI_comm comm, int *id);`
Gibt die ID/den Rang des Prozesses an, der den Prozess eindeutig innerhalb der Gruppe bestimmt. Die Gruppe, die standardmäßig alle Prozesse enthält lautet MPI_COMM_WORLD.

MPI Datenaustasuch

In diesem Abschnitt werden die Punkt-zu-Punkt Operationen `MPI_Send` und `MPI_Recv`, sowie die globale Kommunikationskoperation `MPI_Reduce` vorgestellt.

MPI_Send und MPI_Recv

```
int MPI_Send(
void *smessage,          // Sendepuffer
int count,               // Anzahl der zu sendenden Elemente
MPI_Datatype datatype,   // Typ der zu sendenden Elemente
int dest,                // Rang des Zielprozesses
int tag,      // Art der Nachricht, auf die der Empfänger reagieren soll
MPI_Comm comm)// Gruppe von Prozessen, denen der Empfägner zugeordnet ist

int MPI_Recv(
void *rmessage,         // Empfangspuffer
int count,              // Anzahl der zu empfangenen Elemente
MPI_Datatype datatype,  // Typ der zu empfangenen Elemente
int source,            // Rang der Quelle
int tag,               // Art der zu erwartenden Nachricht
MPI_Comm comm,  // Gruppe von Prozessen, der der Sender zugeordnet ist
MPI_Status *status)  // Informationen über tatsächlich empfangene Daten
```

MPI_Reduce

MPI_Reduce ist eine Akkumulationsoperation, für die jeder beteiligte Prozess Daten zur Verfügung stellt, die dann mit der angegebenen Operation (z.B. Summe, Produkt, Maximum, ...) verknüpft werden. Das Ergebnis erhält der mit dem Parameter `root` festgelegter Wurzelprozess.

```
int MPI_Reduce(
void *sendbuf,      // Sendepuffer
void *recvbuf,      // Empfangspuffer
int count,          // Anzahl der Elemente
MPI_Datatype type, // Typ der Elemente
MPI_Op op,          // Operation z.B. Summe, Maximum, ...
int root,           // Rang des Wurzelprozesses
MPI_Comm comm)     // Gruppe von beteiligten Prozessen)
```

MPI Operationen: MPI_MAX, MPI_PROD, MPI_SUM, MPI_MIN, ...

Beispiel: Senden und Empfangen

Der Wert 42 soll vom Sender mit dem Rang 4 an den Empfänger mit dem Rang 5 gesendet werden. Das gemeinsam genutzte Tag lautet 123. Das Tag dient zur Unterscheidung der Nachrichten und kann ein beliebiger ganzzahliger (int) Wert sein. Wichtig ist, dass Sender und Empfänger den selben Tag benutzen um Daten auszutauschen, weil es sonst zum Deadlock kommt.

Sender

```
int somevalue = 42
MPI_Send(&somevalue, 1, MPI_INT, 5, 123, MPI_COMM_WORLD);
```

Empfänger

```
int receivevalue;
MPI_Status status;
MPI_Recv(&receivevalue, 1, MPI_INT, 4, 123, MPI_COMM_WORLD,&status);
```

MPI und C Datentypen

MPI Datentypen	**C Datentypen**
MPI_INT	int
MPI_LONG	long
MPI_DOUBLE	double

Starten mit mehreren Prozessoren

Alle Programme können auf dem Loginrechner des Clusters gestartet werden.

Ausführung des Programms mit 5 Prozessoren auf dem Loginrechner

```
mpirun -np 5 hello
```

Ausführen des Programms mit 2 Knoten (1 Knoten = 8 Prozesse) auf dem Cluster. Die Ausgabe wird in eine Textdatei der Form "`slurm-JOBID.out`" geschrieben.

```
sbatch -N 2 hpc batch hello
```