PHILIPPS-UNIVERSITÄT MARBURG
Fachbereich Mathematik und Informatik
Prof. Dr. R. Loogen
J.Berthold

D-35032 Marburg
Hans Meerwein Straße
Lahnberge
23.April 2007

Übungen zu „Parallele und Verteilte Algorithmen“, Sommer 2007

Software zu den Übungen

In unserer Vorlesung stehen nicht die Programmiertechniken, sondern die Algorithmen im Vordergrund. Dennoch brauchen wir geeignete Programmierwerkzeuge, um sinnvolle praktische Übungsaufgaben zu stellen. Für die Programmieraufgaben kommen *Linux als Betriebssystem* und *Sprachen mit MPI-Anbindung* (C und Fortran, evtl. Java) sowie paralleles Haskell infrage.
Parallele Algorithmen werden in der Regel in für Cluster-Systeme geeigneten Sprachen entwickelt, meistens unter Linux. Häufig werden etablierte imperative Sprachen wie Fortran und C sowie geeignete Middleware zur Kommunikation und Synchronisation eingesetzt. Fortgeschrittene Programmiersprachen zur Parallelverarbeitung wie Skelettbibliotheken und parallel-funktionale Sprachen nutzen abstraktere Konzepte zur Konzentration auf den Algorithmus.

C + OpenMPI

Der MPI-Standard (siehe auch unter wikipedia) definiert ein Interface für den Datenaustausch mit verteiltem Speicher sowie eine Fortran- und C-Schnittstelle (seit MPI-2 auch C++). MPI ist ein Quasi-Standard in der wissenschaftlichen Programmierung, aus verschiedenen Implementierungen der Vergangenheit sind im wesentlichen noch zwei allgemein relevant:

- Open MPI: aus verschiedenen Implementierungen hervorgegangen, darunter Los-Alamos-MPI (lampi) und LAM-MPI (lam)
- mpich: die erste Implementierung überhaupt. Federführende Autoren des Standards waren hier beteiligt.

Konfiguration

Am Fachbereich ist unter Linux die MPI-Implementierung Open MPI installiert. Um sie zu benutzen, müssen die folgenden Variablen in der Shell-Konfigurationsdatei (normalerweise *.tcshrc*) gesetzt werden:

TC-Shell
```
setenv LD_LIBRARY_PATH /app/lang/parallel/openmpi-1.2/lib
setenv MANPATH $MANPATH:/app/lang/parallel/openmpi-1.2/man
setenv PATH /app/lang/parallel/bin:${PATH}
```

Das Setzen von LD_LIBRARY_PATH ist erforderlich, damit beim Programmstart dynamisch die entsprechenden Bibliotheken gefunden werden. Der MANPATH muss dagegen nicht unbedingt gesetzt werden, falls man keine Anleitung braucht.
In `/app/lang/parallel/bin` (wahlweise auch `/app/lang/parallel/openmpi-1.2/bin`) befinden sich die nötigen Werkzeuge zum Übersetzen und Starten von Programmen.

Übersetzen und Starten von Programmen

Grundlegende MPI-Konzepte

Der MPI-Standard enthält eine Unmenge von Operationen, von denen die meisten allerdings für spezielle Aufgaben gedacht sind und nicht unbedingt erforderlich sind. Als Basis für einfache MPI-Programme mit Peer-to-Peer-Kommunikation reichen wenige Grundoperationen aus.
MPI-Kommunikation erfolgt über sog. *Kommunikatoren* (Typ `MPI_Comm`), welche die Prozesse gruppieren und nach verschiedenen Kontexten trennen können. Ein vordefinierter `MPI_COMM_WORLD` enthält alle gestarteten Prozesse und reicht für einfache Programme in der Regel aus.
Hier die C-Prototypen der grundlegenden Funktionen:

- `int MPI_Init(int* argv, char*** argc);`
  wird zu Beginn des MPI-Programms aufgerufen und sollte die Programmparameter erhalten und verändern dürfen.

- `int MPI_Finalize();`
  wird am Ende des MPI-Programms aufgerufen, um den laufenden Prozess von MPI abzumelden.

- `int MPI_Comm_size(MPI_Comm comm, int* size);`
  um die Anzahl n der in Kommunikator `comm` enthaltenen Prozesse (= verfügbare Kommunikationspartner) zu ermitteln.

- `int MPI_Comm_rank(MPI_Comm comm, int* rank);`
  um die eigene Adresse $(0 \ldots (n-1))$ im Kommunikator `comm` zu ermitteln.

- ```
  int MPI_Send(void* data, int k, MPI_Datatype t,
               int p, int tag, MPI_Comm comm);
  ```
  sendet eine Nachricht mit `k` Datenelementen vom Typ `t` an die Prozess `p` im Kommunikator `comm`. Die Nachricht ist mit einer Zahl `tag` markiert.

- ```
  int MPI_Recv(void* data, int k, MPI_Datatype t,
              int p, int tag, MPI_Comm comm, MPI_Status status);
  ```
  empfängt (maximal!) `k` Elemente vom Typ `t` in einer Nachricht mit Markierung `tag` von Prozess `p` im Kommunikator `comm`. Mit `p=MPI_ANY_SOURCE` kann von einem beliebigem Sender empfangen werden, mit `tag=MPI_ANY_TAG` beliebig markierte Nachrichten. Der Empfangsstatus enthält in diesem Fall die Information, welcher Sender und welche Markierung tatsächlich empfangen wurde, daneben die Anzahl der tatsächlich empfangenen Elemente. Falls diese Information nicht interessiert, verwendet man Parameter `MPI_STATUS_IGNORE`.

Alle Funktionen liefern einen Fehlercode als Rückgabe, der allerdings im Normalfall nicht ausreicht, um einen Fehler zu beheben bzw. das Programm in einen sicheren Zustand zurückzuführen.
MPI bietet viele weitere Funktionen, die hier nicht beschrieben sind; essenziell sind hier die sog. *kollektiven Operationen*. Sie erlauben eine wesentlich bessere Strukturierung des Programms und bieten optimierte Implementierungen für Standardaufgaben wie etwa das Aufteilen von Daten auf alle Prozesse und deren Kombination.

Wichtig: Die genannten Sende- und Empfangsoperationen gehen "eigentlich" von einer *synchronen* Kommunikation aus, d.h. Nachrichten werden nicht gepuffert. Daher kann es schnell zu einem Deadlock kommen wie etwa in diesem einfachen MPI-Programm:

```
// mpiDeadlock
#include "mpi.h"
#include <stdio.h>
#include <string.h>

void main(int argc, char *argv[]) {
  int rank;
  int size;
  int* data;

  // Datenmenge aus Aufrufparameter:
  size = argc>1?atoi(argv[1]):100;
  fprintf(stdout, "Bestellte Groesse: %d\n", size);

  if (size < 2) exit(1);

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  data = (int*) malloc(size*sizeof(int)); // Speicher reservieren
  memset(data, rank % 256, size*sizeof(int)); // daten fuellen

  // sende und empfange Daten vom Nachbarn
  //   (Nachbar: letztes Bit des Rangs invertiert)
  fprintf(stdout, "Sende\n");
  MPI_Send(data,size,MPI_INT, rank ^ 1, 0, MPI_COMM_WORLD);
  fprintf(stdout, "Empfange\n");
  MPI_Recv(data,size,MPI_INT, rank ^ 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  fprintf(stdout, "Erhalten: %x\n", data[1]);

  free(data);

  MPI_Finalize();
}
```

Dass das Programm ohne Parameter dennoch terminiert, liegt daran, dass der Standard eine optimierte (z.B. gepufferte) Kommunikation erlaubt. Man kann aber nicht davon ausgehen, dass das Programm stets funktionieren wird!
(Ab welcher Größe terminiert es nicht mehr? Testen Sie mehrere Implementierungen!)

Ein einfaches MPI-Programm

Wenn das folgende Programm in mehreren Instanzen unter MPI gestartet wird, werden von allen beteiligten Prozessen aus Nachrichten an den "Hauptprozess" (mit Nummer 0) gesendet und dort ausgegeben. Die Nachrichten enthalten den Rechnernamen, auf dem die sendende Instanz ausgeführt wird.

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
```

```
  int rank, size;
  char message[MPI_MAX_PROCESSOR_NAME + 22],
       name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("! Knoten %d von %d startet.\n", rank, size);

  if (!rank) { // Hauptknoten: rank == 0
    int recvRank;
    MPI_Status status; /* mpi.h: struct _status{int MPI_SOURCE;int MPI_TAG;
                                              int MPI_ERROR;int st_length} */
    int i;
    for (i = 1; i < size; i++) {
      // empfange von irgendwem eine Nachricht (Tag 4711) mit einem String
      MPI_Recv(message, 35, MPI_CHAR,
               MPI_ANY_SOURCE, 4711, MPI_COMM_WORLD, &status);
      recvRank = status.MPI_SOURCE;
      printf("Knoten %d sagt: %s\n", recvRank, message);
    }
  } else { // andere Knoten:
    int len;
    MPI_Get_processor_name(name,&len);
    sprintf(message,"Knoten %.2d auf Rechner %s.",
            rank, name);
    // Senden der Nachricht "message"
    // (Länge 22 + Name,Datentyp char, Tag 4711) an rank 0.
    MPI_Send(message,35,MPI_CHAR,0,4711,MPI_COMM_WORLD);
    }
  // alle beenden das Programm.
  MPI_Finalize();
  return 0;
}
```

Übersetzen von Programmen

Programme in C und MPI werden mit einem speziellen Compileraufruf `mpicc` übersetzt. Dieser ruft den eigentlichen Compiler, hier `gcc`, mit passenden Parametern, die man statt dessen auch selbst angeben könnte. Mit `mpicc -showme` wird der Aufruf angezeigt.
Auch wenn nur einzelne Module eines größeren Programms übersetzt werden, sollte `mpicc` verwendet werden.

TC-Shell

```
berthold@maseru:>mpicc -showme
gcc -I/app/lang/parallel/openmpi-1.2/include/openmpi -I/app/lang/parallel/openmp
i-1.2/include -pthread -L/app/lang/parallel/openmpi-1.2/lib -lmpi -lopen-rte -lo
pen-pal -ldl -Wl,--export-dynamic -lnsl -lutil -lm -ldl
berthold@maseru:>
berthold@maseru:>mpicc -O2 -o mpiSimple mpiSimple.c
berthold@maseru:>
```

Starten

Der Befehl `mpirun`[1] startet ein MPI-Programm. Als wichtigster Parameter wird mit `-np` gefolgt von einer Zahl (gleichwertig: `-np, -c` + Zahl) angegeben, auf wie vielen "Knoten" (Rechnern im System) das Programm laufen soll.
Die Rechner, auf denen das Programm gestartet werden soll, können in einem Hostfile angegeben werden, eine weitere Option von `mpirun`

TC-Shell
```
berthold@maseru:>cat hostfile8
# Rechnerraum D4, alle DualCore
boende slots=2
luena slots=2
tamale slots=2
yola slots=2

# Rechnerraum D5, Hyperthreaded oder DualCore
alexandria slots=2
banjul slots=2
lubumbashi slots=2
ogbomosho slots=2
berthold@maseru:>
```

Die Knoten (Nodes) im System können dabei mehrere Programminstanzen ("slots") beherbergen. Im Hostfile wird angegeben, wie viele slots zur Verfügung stehen. Falls diese nicht reichen, werden slots doppelt besetzt (die Angabe im Hostfile ist nur ein "Vorschlag"), außer man verhindert dies explizit durch Angabe von `max_slots` im Hostfile bzw. Option `--nooversubscribe` für `mpirun`.
Beispiel: Start eines Programms in 4 Slots, 8 Hosts à 2 slots im Hostfile:

TC-Shell
```
berthold@maseru:>mpirun -np 4 -hostfile hostfile8 mpiSimple
! Knoten 0 von 4 startet.
Knoten 1 sagt: Knoten 01 auf Rechner boende.
! Knoten 1 von 4 startet.
! Knoten 2 von 4 startet.
Knoten 2 sagt: Knoten 02 auf Rechner luena.
! Knoten 3 von 4 startet.
Knoten 3 sagt: Knoten 03 auf Rechner luena.
berthold@maseru:>man mpirun
...
```

Beispiel: Start eines Programms in 4 Slots. Kein Hostfile, alle lokal.

TC-Shell
```
berthold@maseru:>mpirun -np 4 mpiSimple
! Knoten 0 von 4 startet.
! Knoten 1 von 4 startet.
Knoten 1 sagt: Knoten 01 auf Rechner maseru.
Knoten 2 sagt: Knoten 02 auf Rechner maseru.
Knoten 3 sagt: Knoten 03 auf Rechner maseru.
! Knoten 2 von 4 startet.
! Knoten 3 von 4 startet.
berthold@maseru:>man mpirun
...
```

[1] Der Befehl `mpirun` ist im MPI-Standard lediglich als Empfehlung enthalten. Parameter können je nach Implementierung sehr verschieden sein.

Hosts können mit der Option `-host` sowie ihren Namen (durch Komma getrennt) auch explizit bei Aufruf von `mpirun` angegeben werden.

TC-Shell
```
berthold@maseru:>mpirun -np 3 -host maseru,yola --nooversubscribe mpiSimple
--------------------------------------------------------------------------
There are not enough slots available in the system to satisfy the 3 slots
that were requested by the application:
  mpiSimple

Either request fewer slots for your application, or make more slots available
for use.
--------------------------------------------------------------------------
[maseru:26690] [0,0,0] ORTE_ERROR_LOG: Out of resource in file ../../../../..
/orte/mca/rmaps/round_robin/rmaps_rr.c at line 179
[maseru:26690] [0,0,0] ORTE_ERROR_LOG: Out of resource in file ../../../../..
/orte/mca/rmaps/round_robin/rmaps_rr.c at line 584
[maseru:26690] [0,0,0] ORTE_ERROR_LOG: Out of resource in file ../../../../or
te/mca/rmaps/base/rmaps_base_map_job.c at line 210
[maseru:26690] [0,0,0] ORTE_ERROR_LOG: Out of resource in file ../../../../..
/orte/mca/rmgr/urm/rmgr_urm.c at line370
[maseru:26690] mpirun: spawn failed with errno=-2
berthold@maseru:>
```

Jede Menge weitere Optionen nennt `mpirun` bei Aufruf ohne Parameter, die manpage enthält auch zahlreiche Beispiele.

Java + MPI = MPJ

Trotz der schon in den Jahren von MPI-2 großen Mode Java ist MPI seitens des MPI-Forums nie offiziell mit Java verbunden worden. Es gab aber mehrere Initiativen, diese Verbindung zu schaffen. Da MPI nicht objektorientiert konzipiert wurde, erscheint eine Anbindung zunächst auch schwierig, aber andererseits existiert seit MPI-2 eine C++ Anbindung, welche ähnliche Probleme lösen muss.
Die Basis der OO-Implementierung sind MPI-Kommunikatoren, welche die Sende- und Empfangsoperationen ausführen:

```
public class Comm extends java.lang.Object {
  public void Send(java.lang.Object buf, int offset, int count,
                   Datatype datatype, int dest, int tag)
          throws MPIException
  public Status Recv(java.lang.Object buf, int offset, int count,
                     Datatype datatype, int source, int tag)
          throws MPIException
  ...
}
```

Die versandten und empfangenen Daten müssen, wenngleich im Typ einfach `Object`, stets eindimensionale Arrays passender MPI-Basistypen (oder daraus abgeleiteter Typen) sein. Der `offset` bestimmt die Stelle im Array, an der die empfangenen Daten beginnen (in C mit Pointerarithmetik realisierbar).

Konfiguration

Unter den existierenden Java-MPI-Bindungen sticht "MPJ-Express" insofern heraus, als dass sie allein auf Java aufbaut. Einige andere Prototypen arbeiteten mit JNI-Aufrufen von C-Funktionen einer darunterliegenden MPI-Implementierung und sind somit nicht portabel.
Eine "Installation" (erforderliche Klassen, Dokumentation, Scripts etc.) von MPJ ist unter `/app/lang/parallel/mpj` zu finden. Allerdings braucht jeder Benutzer eine eigene Kopie davon, weil zur Laufzeit Verwaltungsdaten in Unterverzeichnisse geschrieben werden.
Kopieren Sie daher vor der Benutzung dieses Verzeichnis (mindestens `/lib` und `bin`) in Ihr Homeverzeichnis an eine geeignete Stelle. Setzen Sie ferner (in *.tcshrc*, nicht temporär) die beiden Umgebungsvariablen

TC-Shell

```
setenv MPJ_HOME <Ort Ihrer Kopie>
setenv PATH ${MPJ_HOME}/bin:${PATH}
```

Beispielprogramm

Das nachfolgende Beispiel entspricht der Funktionalität des C-Programms *mpiDeadlock.c* aus dem vorherigen Abschnitt.

Java + MPI

```
/*
 * Created on Apr 20, 2007 by berthold
 *
 */
import mpi.*;
import java.util.Arrays;

public class mpiDeadlock {

        /**
         * @param args
         */
        public static void main(String[] args) {
                int rank;
                int size;
                int data[];
                int i;

                MPI.Init(args);
                rank = MPI.COMM_WORLD.Rank();

                for (i=0;i < args.length; i++)
                        System.out.println(i + ": " + args[i]);

                size = 100;
                if (args.length >= 3) { // Argument 3, nicht 0. MPJ-Magix
                        size = Integer.parseInt(args[3]);
                }
                System.out.printf("Bestellte Groesse: %d\n", size);

                if (size < 2) return;

                data = new int[size];
                i = rank % 256;
```

```
            i += (i<<8);
            i += (i<<16);
            Arrays.fill(data, i);

            System.out.printf("Sende\n");
            MPI.COMM_WORLD.Send(data, size, 0, MPI.INT, rank ^ 1, 0);
            System.out.printf("Empfange\n");
            MPI.COMM_WORLD.Recv(data, size, 0, MPI.INT, rank ^ 1, 0);
            System.out.printf(rank + "Erhalten: 0x%x\n", data[1]);

            MPI.Finalize();
        }

    }
```

Programmstart

Zur Ausführung eines MPJ-Programms muss zunächst mit `mpjboot` das System hochgefahren werden. Dann kann mit `mpjrun.sh` (unter Angabe der Maschinen in einem "Hostfile") ein Programm gestartet werden. Danach wird mit `mpjhalt` das System wieder heruntergefahren.

TC-Shell

```
berthold@maseru:>cat ../4machines
boende.mathematik.uni-marburg.de
yola.mathematik.uni-marburg.de
luena.mathematik.uni-marburg.de
oran.mathematik.uni-marburg.de
berthold@maseru:>mpjboot ../4machines
Starting mpjd...
Starting mpjd...
Starting mpjd...
Starting mpjd...
berthold@maseru:>mpjrun.sh -machinesfile ../4machines -np 2 mpiDeadlock 200
11:23:15.604 EVENT  Starting Jetty/4.2.23
11:23:15.619 EVENT  Started HttpContext[/]
11:23:15.622 EVENT  Started SocketListener on 0.0.0.0:15000
...
berthold@maseru:>mpjhalt ../4machines
Stopping mpjd...
Stopped mpjd.
Stopping mpjd...
Stopped mpjd.
Stopping mpjd...
Stopped mpjd.
Stopping mpjd...
Stopped mpjd.
berthold@maseru:>
```

Weitere Informationen

Open-MPI: `http://www.open-mpi.org`
LAM-MPI und Tutorials:`http://www.lam-mpi.org`
MPJ: `http://acet.rdg.ac.uk/projects/mpj/index.php`