1 MPI: Message Passing Interface

- Portable Bibliothek zum Nachrichtenaustausch
- Wurde 1993-94 durch ein intern. Gremium entwickelt
- 1997 wurde der Standard überarbeitet (MPI2)
- Open-Source Implementieren gibt beispielweise von:**MPICH**[1] und **OpenMPI**[2]
- Eigenschaften von MPI:
  - Direkte Anbindung an C, C++ und Fortran
  - verschiedene Arten von Punkt-zu-Punkt Kommunikation
  - globale Kommunikation
  - Daten Umwandlung in heterogenen Systemen
  - Vierte Netze & Topologien möglich

1.1 Beispiel

Hello World

```
1  #include <mpi.h>
   #include <iostream>
3
   int main(int argc, char **argv)
5  {
     int *buf, i, rank, nints, len;
7    char hostname[256];

9    MPI_Init(&argc,&argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11   gethostname(hostname,255);
     std::cout << "Hello␣world!␣␣I␣am␣process␣number:␣" << rank
13             << "␣on␣host␣" << hostname << std::endl;
     MPI_Finalize();
15   return 0;
   }
```

Kompilieren eines Programms

- Das Beispiel ist im SPMD-Stil geschrieben; der MPI Standard legt dies nicht fest und man kann auch andere Paradigmen verwenden.
- Kompilieren eines MPI C-Programmes und starten mit 8 Prozessen:

  ```
  mpicc -o hello hello.c
  mpirun -machinefile machines -np 8 hello
  ```

  Die Liste der Computer steht in der Datei machines.

  Cür C++ Programme heisst der Compiler

  ```
  mpicxx -o hello hello.cc
  ```

[1] http://www-unix.mcs.anl.gov/mpi/mpich

[2] http://www.open-mpi.org/

Initialisierung und Beenden

Bevor irgend ein MPI Befehl aufgerufen werden darf, muss MPI mit `MPI_Init` initialisiert werden, damit MPI das parallele Programm starten kann.

```
int MPI_Init(int *argc, char ***argv)
```

Nach dem letzten MPI Aufruf wird `MPI_Finalize` ausgeführt, um alles Prozesse ordentlich zu beenden.

```
int MPI_Finalize(void)
```

1.2 Kommunicatoren und Topologien

MPI erlaubt es Kommunikation auf einem Subset der gestarten Prozesse durchzuführen, indem *virtuelle* Netzwerke, sogenannte Kommuikatoren, angelegt werden.

- `MPI_Comm` beschreibt einen Kommunikator, eine Menge von Prozessen $\{0, \ldots, P-1\}$.
- Des vordefinierte Netzwerk `MPI_COMM_WORLD` enthält alle gestarten Prozesse.
- *Virtuelle Topologien:* Ein Kommunikator kann zusätzlich eine spezielle Struktur erhalten, z.B. ein mehrdimensionales Feld, oder ein allgemeiner Graph.
- *Kontext:* Jeder Kommunikator definiert seinen eigenen Kommunikationskontext.

Rank und Size

Die Anzahl der Prozesse in einem Kommunikator wird mit `MPI_Comm_size` bestimmt:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Innerhalb eines Kommunikators hat jeder Prozess eine eindeutige Nummer, diese wird mit `MPI_Comm_rank` bestimmt:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

```
1  #include <stdio.h>
   #include <string.h>
3  #include <mpi.h>  // provides MPI macros and functions

5
   int main (int argc, char *argv[])
7  {
       int my_rank;
9      int P;
       int dest;
11     int source;
       int tag=50;
13     char hostname[256];
       MPI_Status status;
15
       MPI_Init(&argc,&argv);  // begin of every MPI program
17
       MPI_Comm_size(MPI_COMM_WORLD,&P);        // number of processes
19     MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);  // my process number

21     gethostname(hostname,255);

23     // number of current process always between 0 and P-1
       if (my_rank!=0)
25     {
           dest = 0;
27         MPI_Send(hostname,strlen(hostname)+1,MPI_CHAR, // Send data
                    dest,tag,MPI_COMM_WORLD);             // (blocking)
29     }
```

```
    else
    {
        for (source=1; source<P; source++)
        {
            MPI_Recv(hostname,256,MPI_CHAR,source,tag, // Receive data
                     MPI_COMM_WORLD,&status);          // (blocking)
            std::cout << "Reveived a message from process " << P
                      << " on machine " << hostname;
        }
    }

    MPI_Finalize();  // end of every MPI program

    return 0;
}
```

Ausgabe des Beispiel Programms (mit P=8)

```
I am process 0 of 8

I am process 1 of 8

I am process 2 of 8

I am process 3 of 8

I am process 4 of 8

I am process 5 of 8

I am process 6 of 8

I am process 7 of 8
```

1.3 Blockierende Kommunikation

Die Entsprechung zu **send** und **recv** bieten

```
int MPI_Send(void *message, int count, MPI_Datatype dt,
             int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *message, int count, MPI_Datatype dt,
             int src, int tag, MPI_Comm comm,
             MPI_Status *status);
```

Die ersten drei Parameter, `message`, `count` und `dt`, beschreiben die eigentlichen Daten. `message` ist ein Zeiger auf ein Feld mit `count` Elementen des Typs `dt`. Die Angabe des Datentyps erlaubt die automatische Umwandlung durch MPI. Die Parameter `dest`, `tag` und `comm` beschreiben das Ziel, bzw. die Quelle der Nachricht.

MPI biete verschiedene Varianten von `MPI_Send` (`MPI_BSend`, `MPI_SSend`, `MPI_RSend`), die wir aber jetzt nicht weiter diskutieren wollen.

`MPI_ANY_SOURCE` und `MPI_ANY_TAG` können verwendet werden, um belibige Nachrichten zu empfangen. Damit enthält `MPI_Recv` die Funktionalität von **recv_any**.

Datenumwandlung

MPI erlaubt die Verwendung in heterogenen Netzen. Hierbei ist es nötig manche Daten an die Darstellung auf der fremden Architektur anzupassen.

MPI definiert die architektur-unabhängigen Datentypen:

MPI_CHAR, MPI_UNSIGNED_CHAR, MPI_BYTE MPI_SHORT, MPI_INT, MPI_LONG, MPI_LONG_LONG_INT, MPI_UNSIGNED, MPI_UNSIGNED_SHORT, MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE and MPI_LONG_DOUBLE.

Der MPI Datentyp `MPI_BYTE` wird *nie* konvertiert.

Status

```
typedef struct {
    int count;
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
} MPI_Status;
```

`MPI_Status` ist ein zusammengesetzter Datentyp, der Informationen über die Anzahl der empfangenen Objekte, den Quellprozess, das Tag und den Fehler Status enhält.

Guard Funktion

Die Guard Funktion **rprobe** liefert

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm,
               int *flag, MPI_Status *status);
```

Es ist eine nicht-blockierende Funktion, die überprüft, ob eine Nachricht vorliegt. `flag` erhält den Wert `true` ($\neq 0$) wenn eine Nachricht mit passendem source und tag empfangen werden kann. Auch hier können `MPI_ANY_SOURCE` und `MPI_ANY_TAG` verwendet werden.

1.4 Nicht-blockierende Kommunikation

Die Funktionen **asend** und **arecv** bietet MPI als

```
int MPI_ISend(void *buf, int count, MPI_Datatype dt,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *req);
int MPI_IRecv(void *buf, int count, MPI_Datatype dt,
              int src, int tag, MPI_Comm comm,
              MPI_Request *req);
```

`MPI_Request` speichert den Status einer Kommunikation, wie unsere **msgid**.

`MPI_Request`-Objekte

Der Status einer Nachricht kann mit Hilfe der `MPI_Request`-Objekte und folgender Funktion geprüft werden:

```
int MPI_Test(MPI_Request *req,int *flag, MPI_Status *status);
```

`flag` wird auf `true` ($\neq 0$) gesetzt, wenn die Kommunikation, die `req` beschreibt abgeschlossen ist. In diesem Fall enthält `status` weiter Informationen.

1.5 Globale Kommunikation

MPI bietet ebenfalls Funktionen zur globalen Kommunikation, welche alle Prozesse eines Kommunikators einschließen.

```
1    int MPI_Barrier(MPI_Comm comm);
```

implementiert eine Barriere; alle Prozesse werden blockiert, bis der letzte Prozess die Funktion ausgeführt hat.

```
1    int MPI_Bcast(void *buf, int count, MPI_Datatype dt,
                   int root, MPI_Comm comm);
```

verteilt eine Nachricht an alle Prozesse eine Kommunikators (Einer-an-Alle Kommunikation).

Einsammeln von Daten

MPI hat eine Reihe verschiedener Funktionen um Daten von verschiedenen Prozessen einzusammeln. Z.B:

```
int MPI_Reduce(void *sbuf, void *rbuf, int count,
2          MPI_Datatype dt, MPI_Op op, int root, MPI_Comm comm);
```

kombiniert die Daten im Sende-Puffer `sbuf` aller Prozesse durch die assoziative Operation `op` (z.B. `MPI_SUM`, `MPI_MAX` oder `MPI_MIN`). Das Ergebnis erhält der Prozesse `root` in seinen Empfang-Puffer `rbuf`.

2 Zeitmessung

MPI bietet direkt Funktionen zur Zeitmessung:

```
double MPI_Wtime();
```

Der Rückgabewert sind Sekunden seit einem „beliebigen" Zeitpunkt in der Vergangenheit. Die Zeit für eine spezielle Operation läßt sich also mit

```
1 double start = MPI_Wtime();
  expensive_funktion();
3 std::cout << "elapsed␣time␣=␣" << MPI_Wtime() - start << std::endl;
```

bestimmen.

References

[1] *MPI: Dokumentation der verschiedenen Message-Passing Interface Standards* http://www.mpi-forum.org/docs/

[2] *MPICH-A Portable Implementation of MPI* http://www-unix.mcs.anl.gov/mpi/mpich

[3] *Open MPI: Open Source High Performance Computing* http://www.open-mpi.org/

[4] *Liste von MPI Tutorials* http://www-unix.mcs.anl.gov/mpi/tutorial/