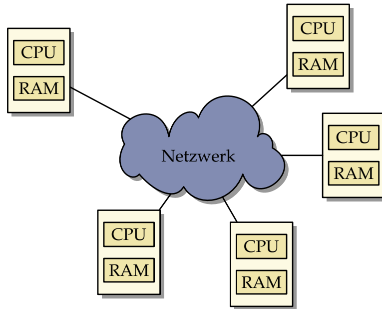


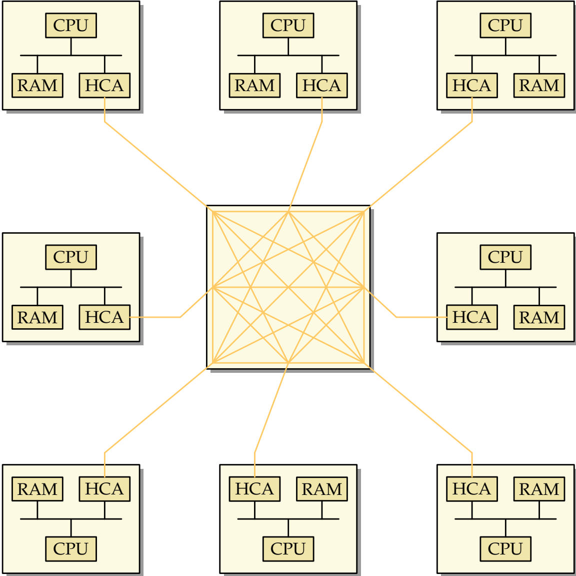
- Multicomputer bestehen aus einzelnen Rechnern mit eigenem Speicher, die über ein Netzwerk miteinander verbunden sind.
- Ein direkter Zugriff auf fremden Speicher ist nicht möglich.
- Die Kommunikation kann daher nicht über gemeinsame Speicherbereiche erfolgen. Stattdessen geschieht dies durch den Austausch von Daten über das Netzwerk.

- Eine traditionelle Vernetzung einzelner unabhängiger Maschinen über Ethernet und der Verwendung von TCP/IP-Sockets erscheint naheliegend.
- Der Vorteil ist die kostengünstige Realisierung, da die bereits vorhandene Infrastruktur genutzt wird und zahlreiche Ressourcen zeitweise ungenutzt sind (wie etwa Pools mit Desktop-Maschinen).
- Zu den Nachteilen gehört
  - die hohe Latenzzeit (ca. $150\mu s$ bei GbE auf Pacioli, ca. $500\mu s$ über das Uni-Netzwerk),
  - die vergleichsweise niedrige Bandbreite,
  - das Fehlen einer garantierten Bandbreite und
  - die Fehleranfälligkeit (wird von TCP/IP automatisch korrigiert, kostet aber Zeit).
  - Ferner fehlt die Skalierbarkeit, wenn nicht erheblich mehr in die Netzwerkinfrastruktur investiert wird.

- Mehrere Hersteller schlossen sich 1999 zusammen, um gemeinsam einen Standard zu entwickeln für Netzwerke mit höheren Bandbreiten und niedrigeren Latenzzeiten.
- Infiniband ist heute die populärste Vernetzung bei Supercomputern: Zwei Supercomputer der TOP-10 und 210 der TOP-500 verwenden Infiniband (Stand: November 2012).
- Die Latenzzeiten liegen im Bereich von 140 *ns* bis 2,6 μs.
- Brutto-Bandbreiten sind zur Zeit bis ca. 56 Gb/s möglich. (Bei Pacioli: brutto 2 Gb/s, netto mit MPI knapp 1 Gb/s.)
- Nachteile:
  - Keine hierarchischen Netzwerkstrukturen und damit eine Begrenzung der maximalen Rechnerzahl,
  - alles muss räumlich sehr eng zueinander stehen,
  - sehr hohe Kosten insbesondere dann, wenn viele Rechner auf diese Weise zu verbinden sind.

- Bei einer Vernetzung über Infiniband gibt es einen zentralen Switch, an dem alle beteiligten Rechner angeschlossen sind.
- Jede der Rechner benötigt einen speziellen HCA (*Host Channel Adapter*), der direkten Zugang zum Hauptspeicher besitzt.
- Zwischen den HCAs und dem Switch wird normalerweise Kupfer verwendet. Die maximale Länge beträgt hier 14 Meter. Mit optischen Kabeln und entsprechenden Adaptern können auch Längen bis zu ca. 100 Meter erreicht werden.
- Zwischen einem Rechner und dem Switch können auch mehrere Verbindungen bestehen zur Erhöhung der Bandbreite.
- Die zur Zeit auf dem Markt angebotenen InfiniBand-Switches bieten zwischen 8 und 864 Ports.

Topologie bei InfiniBand



Latenzzeiten bei InfiniBand

Die extrem niedrigen Latenzzeiten werden bei InfiniBand nur durch spezielle Techniken erreicht:

- Die HCAs haben direkten Zugang zum Hauptspeicher, d.h. ohne Intervention des Betriebssystems kann der Speicher ausgelesen oder beschrieben werden. Die HCAs können dabei auch selbständig virtuelle in physische Adressen umwandeln.
- Es findet kein Routing statt. Der Switch hat eine separate Verbindungsleitung für jede beliebige Anschlusskombination. Damit steht in jedem Falle die volle Bandbreite ungeteilt zur Verfügung. Die Latenzzeiten innerhalb eines Switch-Chips können bei 200 Nanosekunden liegen, von Port zu Port werden beim 648-Port-Switch von Mellanox nach Herstellerangaben Latenzzeiten von 100-300 Nanosekunden erreicht.

Auf Pacioli werden auf Programmebene (mit MPI) Latenzzeiten von unter 5 μs erreicht.

- Mit einer Rechenleistung von 17,59 Petaflop/s ist die Installation seit November 2012 der leistungsstärkste Rechner.
- Titan besteht aus 18.688 einzelnen Knoten des Typs Cray XK7, die jeweils mit einem AMD-Opteron-6274-Prozessor mit 16 Kernen und einer Nvidia-Tesla-K20X-GPU bestückt sind.
- Jede der GPUs hat 2.688 SPs (Stream-Prozessoren).
- Jeweils vier Knoten werden zu einer Blade zusammengefasst, die in insgesamt 200 Schränken verbaut sind.
- Für je zwei Knoten gibt es für die Kommunikation jeweils einen *Cray-Gemini-interconnect*-Netzwerkknoten, mit 10 Außenverbindungen, die topologisch in einem drei-dimensionalen Torus organisiert sind (vier redundant ausgelegte Verbindungen jeweils in den x- und z-Dimensionen und zwei in der y-Dimension, die topologisch in einem drei-dimensionalen Torus organisiert sind.
- Die Verbindungsstruktur wurde speziell für MPI ausgelegt und erlaubt auch zusätzlich die dreidimensionale Adressierung fremder Speicherbereiche.

- 12 Knoten mit jeweils 2 AMD-Opteron-252-Prozessoren (2,6 GHz) und 8 GiB Hauptspeicher.
- 20 Knoten mit jeweils 2 AMD-Opteron-2218-Prozessoren mit jeweils zwei Kernen und 8 GiB Hauptspeicher.
- 4 Knoten mit 2 Intel-Xeon-E5-2630-CPUs mit jeweils sechs Kernen und 128 GiB Hauptspeicher.
- Zu jedem Knoten gibt es eine lokale Festplatte.
- Über einen Switch sind sämtliche Knoten untereinander über Infiniband vernetzt.
- Zusätzlich steht noch ein GbE-Netzwerk zur Verfügung, an das jeder der Knoten angeschlossen ist.
- Pacioli bietet zwar auch IP-Schnittstellen für Infiniband an, aber diese sind nicht annähernd so effizient wie die direkte Schnittstelle.

- MPI (*Message Passing Interface*) ist ein Standard für eine Bibliotheksschnittstelle für parallele Programme.
- 1994 entstand die erste Fassung (1.0), 1995 die Version 1.2 und seit 1997 gibt es 2.0. Kürzlich (im September 2012) erschien die Version 3.0. Die Standards sind öffentlich unter `http://www.mpi-forum.org/`
- Der Standard umfasst die sprachspezifischen Schnittstellen für Fortran und C. (Es wird die C-Schnittstelle in C++ verwendet. Alternativ bietet sich die Boost-Library an: `http://www.boost.org/doc/libs/1_53_0/doc/html/mpi.html`).
- Es stehen mehrere Open-Source-Implementierungen zur Verfügung:
  - OpenMPI: `http://www.open-mpi.org/` (wird von Sun verwendet, auf Theseus und Thales)
  - MPICH: `http://www.mpich.org/`
  - MVAPICH: `http://mvapich.cse.ohio-state.edu/` (spezialisiert auf Infiniband, auf Pacioli)

- Zu Beginn wird mit n die Zahl der Prozesse festgelegt.
- Jeder Prozess läuft in seinem eigenen Adressraum und hat innerhalb von MPI eine eigene Nummer (*rank*) im Bereich von 0 bis $n - 1$.
- Die Kommunikation mit den anderen Prozessen erfolgt über Nachrichten, die entweder an alle gerichtet werden (*broadcast*) oder individuell versandt werden.
- Die Kommunikation kann sowohl synchron als auch asynchron erfolgen.
- Die Prozesse können in einzelne Gruppen aufgesplittet werden.

Simpson-Verfahren mit MPI

mpi-simpson.cpp

```
int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);

   int nofprocesses; MPI_Comm_size(MPI_COMM_WORLD, &nofprocesses);
   int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   // process command line arguments
   int n; // number of intervals
   if (rank == 0) {
      cmdname = argv[0];
      if (argc > 2) usage();
      if (argc == 1) {
         n = nofprocesses;
      } else {
         istringstream arg(argv[1]);
         if (!(arg >> n) || n <= 0) usage();
      }
   }
   // ...

   MPI_Finalize();

   if (rank == 0) {
      cout << setprecision(14) << sum << endl;
   }
}
```

mpi-simpson.cpp

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

   int nofprocesses; MPI_Comm_size(MPI_COMM_WORLD, &nofprocesses);
   int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

- Im Normalfall starten alle Prozesse das gleiche Programm und beginnen alle mit *main*(). (Es ist auch möglich, verschiedene Programme über MPI zu koordinieren.)
- Erst nach dem Aufruf von *MPI_Init*() sind weitere MPI-Operationen zulässig.
- *MPI_COMM_WORLD* ist die globale Gesamtgruppe aller Prozesse eines MPI-Laufs.
- Die Funktionen *MPI_Comm_size* und *MPI_Comm_rank* liefern die Zahl der Prozesse bzw. die eigene Nummer innerhalb der Gruppe (immer ab 0, konsekutiv weiterzählend).

mpi-simpson.cpp

```
// process command line arguments
int n; // number of intervals
if (rank == 0) {
   cmdname = argv[0];
   if (argc > 2) usage();
   if (argc == 1) {
      n = nofprocesses;
   } else {
      istringstream arg(argv[1]);
      if (!(arg >> n) || n <= 0) usage();
   }
}
```

- Der Hauptprozess hat den *rank* 0. Nur dieser sollte verwendet werden, um Kommandozeilenargumente auszuwerten und/oder Ein- und Ausgabe zu betreiben.

mpi-simpson.cpp

```
   // broadcast number of intervals
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

- Mit der Funktion *MPI_Bcast* kann eine Nachricht an alle Mitglieder einer Gruppe versandt werden.
- Die Funktion bezieht sich auf eine Gruppe, wobei *MPI_COMM_WORLD* die globale Gesamtgruppe repräsentiert.
- Der erste Parameter ist ein Zeiger auf das erste zu übermittelnde Objekt. Der zweite Parameter nennt die Zahl der zu übermittelnden Objekte (hier nur 1).
- Der dritte Parameter spezifiziert den Datentyp eines zu übermittelnden Elements. Hier wird *MPI_INT* verwendet, das dem Datentyp **int** entspricht.
- Der letzte Parameter legt fest, welcher Prozess den Broadcast verschickt. Alle anderen Prozesse, die den Aufruf ausführen, empfangen das Paket.

mpi-simpson.cpp

```
// broadcast number of intervals
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

double value = 0; // summed up value of our intervals;
if (rank < n) {
   int nofintervals = n / nofprocesses;
   int remainder = n % nofprocesses;
   int first_interval = rank * nofintervals;
   if (rank < remainder) {
      ++nofintervals;
      if (rank > 0) first_interval += rank;
   } else {
      first_interval += remainder;
   }
   int next_interval = first_interval + nofintervals;

   double xleft = a + first_interval * (b - a) / n;
   double x = a + next_interval * (b - a) / n;
   value = simpson(f, xleft, x, nofintervals);
}
double sum;
MPI_Reduce(&value, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

mpi-simpson.cpp

```
double sum;
MPI_Reduce(&value, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

- Mit der Funktion *MPI_Reduce* werden die einzelnen Ergebnisse aller Prozesse (einschließlich dem auswertenden Prozess) eingesammelt und dann mit einer auszuwählenden Funktion aggregiert.
- Der erste Parameter ist ein Zeiger auf ein Einzelresultat. Der zweite Parameter verweist auf die Variable, wo der aggregierte Wert abzulegen ist.
- Der dritte Parameter liegt wieder die Zahl der Elemente fest (hier 1) und der vierte den Datentyp (hier *MPI_DOUBLE* für **double**).
- Der fünfte Parameter spezifiziert die aggregierende Funktion (hier *MPI_SUM* zum Aufsummieren) und der sechste Parameter gibt an, welcher Prozess den aggregierten Wert erhält.

MPI unterstützt folgende Datentypen von C++:

MPI_CHAR	**signed char**
MPI_SIGNED_CHAR	**signed char**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_SHORT	**signed short**
MPI_INT	**signed int**
MPI_LONG	**signed long**
MPI_UNSIGNED_SHORT	**unsigned short**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned long**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**
MPI_LONG_DOUBLE	**long double**
MPI_WCHAR	*wchar_t*
MPI_BOOL	*bool*
MPI_COMPLEX	*Complex*<**float**>
MPI_DOUBLE_COMPLEX	*Complex*<**double**>
MPI_LONG_DOUBLE_COMPLEX	*Complex*<**long double**>

Makefile

```
CXX :=          mpiCC
CXXFLAGS :=     -fast -library=stlport4
CC :=           mpiCC
CFLAGS :=       -fast -library=stlport4
```

- Die Option *mpi* sollte in *~/.options* genannt werden. Ggf. hinzufügen und erneut anmelden.
- Dann ist */opt/SUNWhpc/HPC8.2.1c/sun/bin* relativ weit vorne im Suchpfad.
- Statt den C++-Compiler von Sun mit *CC* direkt aufzurufen, wird stattdessen *mpiCC* verwendet, das alle MPI-spezifischen Header-Dateien und Bibliotheken automatisch zugänglich macht.
- Die Option *-fast* schaltet alle Optimierungen ein. Die Warnung, die deswegen ausgegeben wird, kann ignoriert werden.
- Hinweis: *mpiCC* unterstützt noch nicht den aktuellen C++-Standard von 2012.

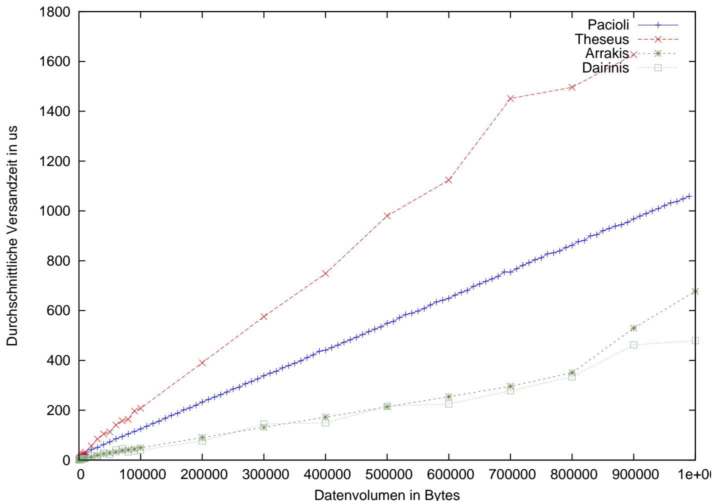
Ausführen von MPI-Anwendungen auf Theseus


```
theseus$ f
Makefile  mpi-simpson.C
theseus$ make mpi-simpson
mpiCC -fast    mpi-simpson.C   -o mpi-simpson
CC: Warning: -xarch=native has been explicitly specified, or implicitly specif
theseus$ time mpirun -np 1 mpi-simpson 10000000
3.1415926535902

real    0m0.95s
user    0m0.87s
sys     0m0.03s
theseus$ time mpirun -np 4 mpi-simpson 10000000
3.1415926535897

real    0m0.39s
user    0m1.02s
sys     0m0.14s
theseus$
```

- Mit *mpirun* können MPI-Anwendungen gestartet werden.
- Wenn das Programm ohne *mpirun* läuft, dann gibt es nur einen einzigen Prozess.
- Die Option *-np* spezifiziert die Zahl der zu startenden Prozesse. Per Voreinstellung starten die alle auf der gleichen Maschine.

```
theseus$ cat my-machines
malaga
rom
prag
lille
theseus$ time mpirun -hostfile my-machines -np 4 mpi-simpson 10000000
3.1415926535897

real    0m3.03s
user    0m0.31s
sys     0m0.06s
theseus$
```

- Die Option *-hostfile* ermöglicht die Spezifikation einer Datei mit Rechnernamen. Diese Datei sollte soviel Einträge enthalten, wie Prozesse gestartet werden.
- Bei OpenMPI werden die Prozesse auf den anderen Rechnern mit Hilfe der *ssh* gestartet. Letzteres sollte ohne Passwort möglich sein. Entsprechend sollte mit *ssh-keygen* ein Schlüsselpaar erzeugt werden und der eigene öffentliche Schlüssel in *~/.ssh/authorized_keys* integriert werden.
- Das reguläre Ethernet mit TCP/IP ist jedoch langsam!



- Pacioli: 8 Prozesse, Infiniband. Gemeinsamer Speicher: Theseus: 6 Prozesse; Arrakis: 2 Prozesse (AMD-Opteron Dual-Core, 3 GHz); Dairinis: 4 Prozesse (Intel Quad-Core, 2,5 GHz)

Warum schneidet die Pacioli mit dem Infiniband besser als die Theseus ab?

- OpenMPI nutzt zwar gemeinsame Speicherbereiche zur Kommunikation, aber dennoch müssen die Daten beim Transfer zweifach kopiert werden.
- Das Kopieren erfolgt zu Lasten der normalen CPUs.
- Hier wäre OpenMP grundsätzlich wesentlich schneller, da dort der doppelte Kopieraufwand entfällt.
- Sobald kein nennenswerter Kopieraufwand notwendig ist, dann sieht die Theseus mit ihren niedrigeren Latenzzeiten besser aus: 2,2 μs vs. 4,8 μs bei Pacioli. (Arrakis: 0.8 μs; Dairinis: 0.9 μs).

- Bei inhomogenen Rechnerleistungen oder bei einer inhomogenen Stückelung in Einzelaufgaben kann es sinnvoll sein, die Last dynamisch zu verteilen.
- In diesem Falle übernimmt ein Prozess die Koordination, indem er Einzelaufträge vergibt, die Ergebnisse aufsammelt und – sofern noch mehr zu tun ist – weitere Aufträge verschickt.
- Die anderen Prozesse arbeiten alle als Sklaven, die Aufträge entgegennehmen, verarbeiten und das Ergebnis zurücksenden.
- Dies wird an einem Beispiel der Matrix-Vektor-Multiplikation demonstriert, wobei diese Technik in diesem konkreten Fall nicht viel bringt.

mpi-gemv.cpp

```
int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);

   int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   int nofslaves; MPI_Comm_size(MPI_COMM_WORLD, &nofslaves);
   --nofslaves; assert(nofslaves > 0);

   if (rank == 0) {
      int n; double** A; double* x;
      if (!read_parameters(n, A, x)) {
         cerr << "Invalid input!" << endl;
         MPI_Abort(MPI_COMM_WORLD, 1);
      }
      double* y = new double[n];
      gemv_master(n, A, x, y, nofslaves);
      for (int i = 0; i < n; ++i) {
         cout << " " << y[i] << endl;
      }
   } else {
      gemv_slave();
   }

   MPI_Finalize();
}
```

Rolle eines Sklaven

mpi-gemv.cpp

```
static void gemv_slave() {
   int n;
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   double* x = new double[n];
   MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

   double* row = new double[n];
   // receive tasks and process them
   for(;;) {
      // receive next task
      MPI_Status status;
      MPI_Recv(row, n, MPI_DOUBLE, 0, MPI_ANY_TAG,
         MPI_COMM_WORLD, &status);
      if (status.MPI_TAG == FINISH) break;
      // process it
      double result = 0;
      for (int i = 0; i < n; ++i) {
         result += row[i] * x[i];
      }
      // send result back to master
      MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
   }
   // release allocated memory
   delete[] x; delete[] row;
}
```

mpi-gemv.cpp

```
int n;
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
double* x = new double[n];
MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- Zu Beginn werden die Größe des Vektors und der Vektor selbst übermittelt.
- Da alle Sklaven den gleichen Vektor (mit unterschiedlichen Zeilen der Matrix) multiplizieren, kann der Vektor ebenfalls gleich zu Beginn mit *Bcast* an alle verteilt werden.

mpi-gemv.cpp

```
      MPI_Status status;
      MPI_Recv(row, n, MPI_DOUBLE, 0, MPI_ANY_TAG,
         MPI_COMM_WORLD, &status);
      if (status.MPI_TAG == FINISH) break;
```

- Mit *MPI_Recv* wird hier aus der globalen Gruppe eine Nachricht empfangen.
- Die Parameter: Zeiger auf den Datenpuffer, die Zahl der Elemente, der Element-Datentyp, der sendende Prozess, die gewünschte Art der Nachricht (*MPI_ANY_TAG* akzeptiert alles), die Gruppe und der Status, über den Nachrichtenart ermittelt werden kann.
- Nachrichtenarten gibt es hier zwei: *NEXT_ROW* für den nächsten Auftrag und *FINISH*, wenn es keine weiteren Aufträge mehr gibt.

mpi-gemv.cpp

```
MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

- *MPI_Send* versendet eine individuelle Nachricht synchron, d.h. diese Methode kehrt erst dann zurück, wenn der Empfänger die Nachricht erhalten hat.
- Die Parameter: Zeiger auf den Datenpuffer, die Zahl der Elemente (hier 1), der Element-Datentyp, der Empfänger-Prozess (hier 0) und die Art der Nachricht (0, spielt hier keine Rolle).

mpi-gemv.cpp

```
static void
gemv_master(int n, double** A, double *x, double* y,
            int nofslaves) {
   // broadcast parameters that are required by all slaves
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

   // send out initial tasks for all slaves
   int* tasks = new int[nofslaves];
   // ...

   // collect results and send out remaining tasks
   // ...

   // release allocated memory
   delete[] tasks;
}
```

- Zu Beginn werden die beiden Parameter n und x, die für alle Sklaven gleich sind, mit *Bcast* verteilt.
- Danach erhält jeder der Sklaven einen ersten Auftrag.
- Anschließend werden Ergebnisse eingesammelt und – sofern noch etwas zu tun übrig bleibt – die Anschlußaufträge verteilt.

mpi-gemv.cpp

```
// send out initial tasks for all slaves
   // remember the task for each of the slaves
int* tasks = new int[nofslaves];
int next_task = 0;
for (int slave = 1; slave <= nofslaves; ++slave) {
   if (next_task < n) {
      int row = next_task++; // pick next remaining task
      MPI_Send(A[row], n, MPI_DOUBLE, slave, NEXT_ROW,
         MPI_COMM_WORLD);
      // remember which task was sent out to whom
      tasks[slave-1] = row;
   } else {
      // there is no work left for this slave
      MPI_Send(0, 0, MPI_DOUBLE, slave, FINISH, MPI_COMM_WORLD);
   }
}
```

- Die Sklaven erhalten zu Beginn jeweils eine Zeile der Matrix A, die sie dann mit x multiplizieren können.

mpi-gemv.cpp

```
// collect results and send out remaining tasks
int done = 0;
while (done < n) {
   // receive result of a completed task
   double value = 0; // initialize it to get rid of warning
   MPI_Status status;
   MPI_Recv(&value, 1, MPI_DOUBLE,
      MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
   int slave = status.MPI_SOURCE;
   int row = tasks[slave-1];
   y[row] = value;
   ++done;
   // send out next task, if there is one left
   if (next_task < n) {
      row = next_task++;
      MPI_Send(A[row], n, MPI_DOUBLE, slave, NEXT_ROW,
         MPI_COMM_WORLD);
      tasks[slave-1] = row;
   } else {
      // send notification that there is no more work to be done
      MPI_Send(0, 0, MPI_DOUBLE, slave, FINISH, MPI_COMM_WORLD);
   }
}
```

mpi-gemv.cpp

```
      MPI_Status status;
      MPI_Recv(&value, 1, MPI_DOUBLE,
         MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      int slave = status.MPI_SOURCE;
```

- Mit *MPI_ANY_SOURCE* wird angegeben, dass ein beliebiger Sender akzeptiert wird.
- Hier ist die Identifikation des Sklaven wichtig, damit das Ergebnis korrekt in y eingetragen werden kann. Dies erfolgt hier mit *status.Get_source*().