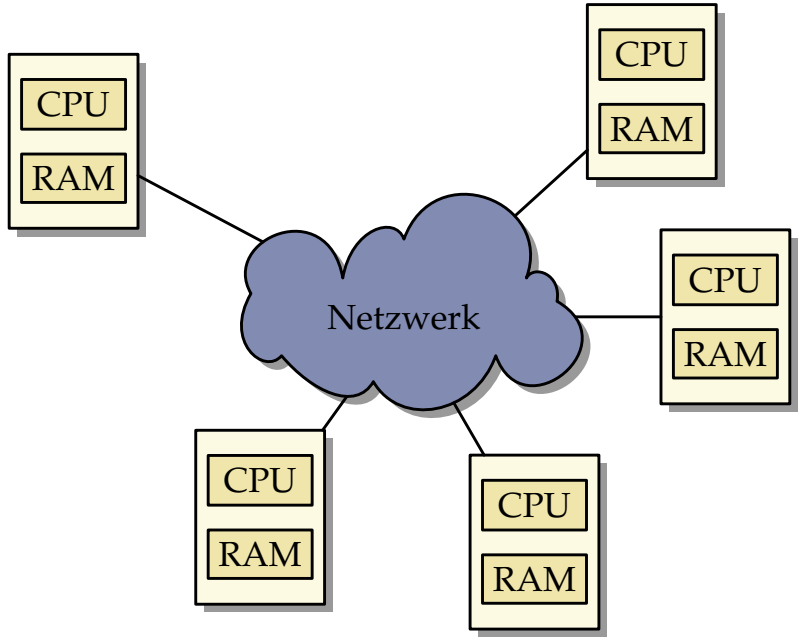


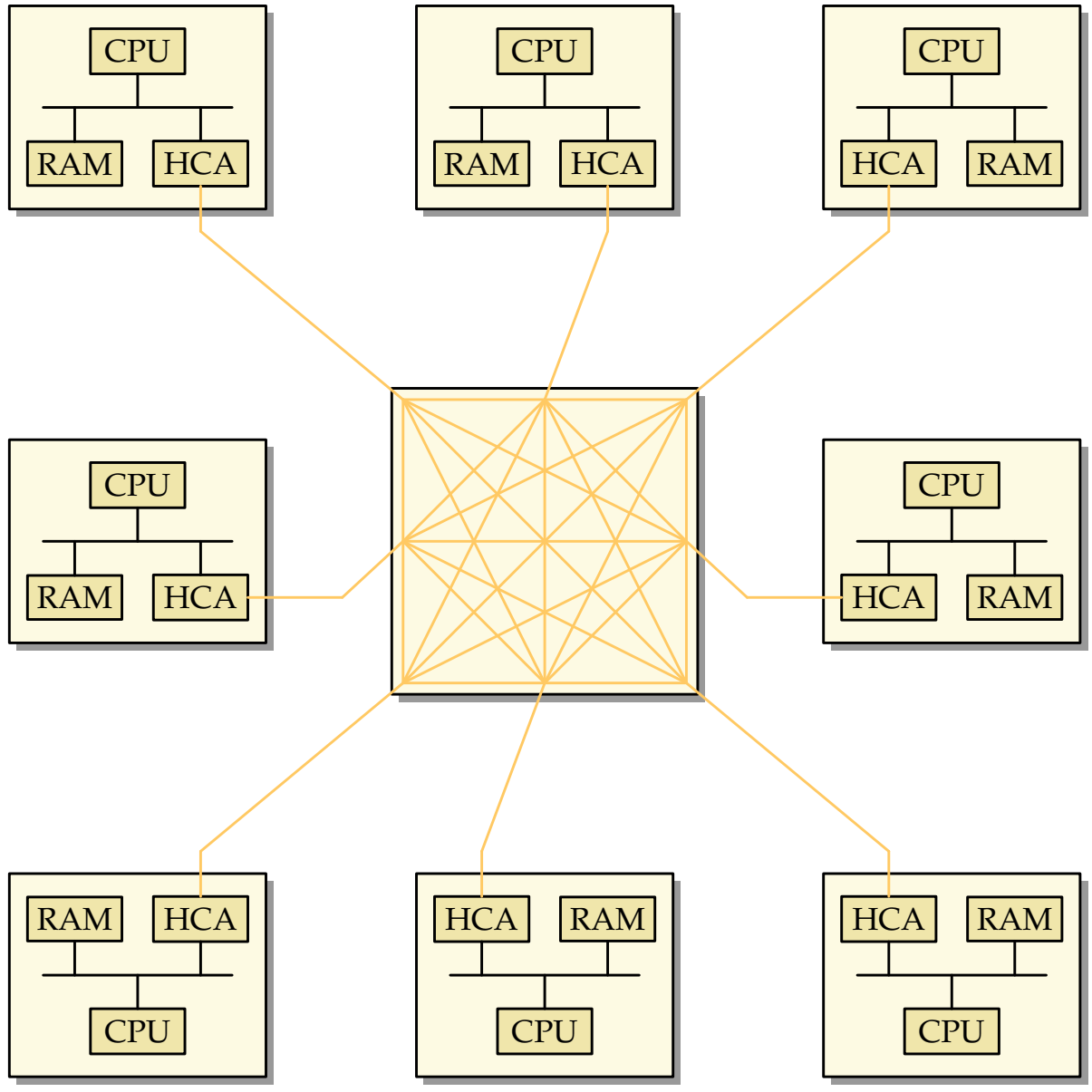
- Multicomputer bestehen aus einzelnen Rechnern mit eigenem Speicher, die über ein Netzwerk miteinander verbunden sind.
- Ein direkter Zugriff auf fremden Speicher ist nicht möglich.
- Die Kommunikation kann daher nicht über gemeinsame Speicherbereiche erfolgen. Stattdessen geschieht dies durch den Austausch 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. 150μs bei GbE auf Pacioli, ca. 500μs 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 eine der populärsten Vernetzungen bei Supercomputern: 178 der TOP-500 verwenden Infiniband (Stand: Juni 2017).
- 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

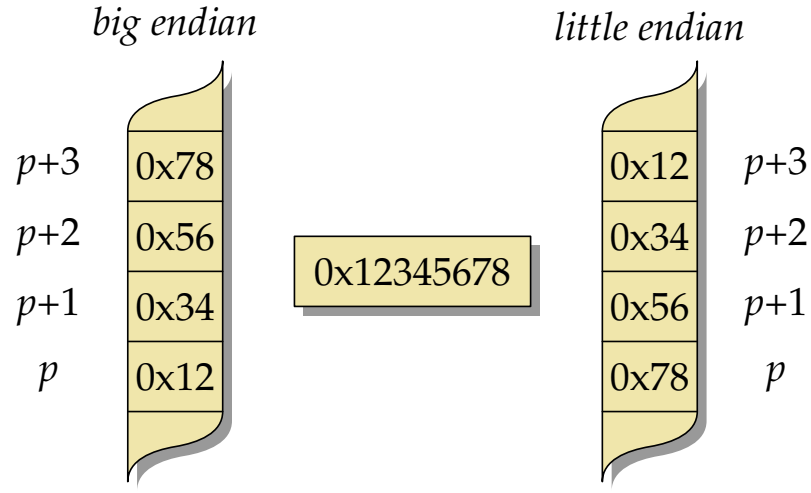


Die extrem niedrigen Latenzzeiten werden bei InfiniBand nur durch spezielle Techniken erreicht:

- 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.

- Da einzelne Rechner unterschiedlichen Architekturen angehören können, werden möglicherweise einige Datentypen (etwa ganze Zahlen oder Gleitkommazahlen) unterschiedlich binär repräsentiert.
- Wenn die Daten mit Typinformationen versehen werden, dann wird die Gefahr von Fehlinterpretationen vermieden.
- Die Übertragung von Daten gibt auch die Gelegenheit, die Struktur umzuorganisieren. Beispielsweise kann ein Spaltenvektor in einen Zeilenvektor konvertiert werden.
- Auch die Übertragung dynamischer Datenstrukturen ist möglich. Dann müssen Zeiger in Referenzen umgesetzt werden.
- Die Technik des Verpackens und Auspackens von Datenstrukturen in Byte-Sequenzen, die sich übertragen lassen, wird Serialisierung oder *marshalling* genannt.



- Bei *little endian* sind im ersten Byte die niedrigstwertigen Bits (hier 0*x78*).
- Die Reihenfolge ist bei *big endian* genau umgekehrt, d.h. die höchstwertigen Bits kommen zuerst (hier 0*x12*).
- Da der Zugriff immer byte-weise erfolgt, interessiert uns nur die Reihenfolge der Bytes, nicht der Bits.
- Zu den Plattformen mit *little endian* gehört die x86-Architektur von Intel, während die SPARC-Architektur normalerweise mit *big endian* operiert.

- Prinzipiell hat sich als Repräsentierung das Zweier-Komplement durchgesetzt:

$$a = \sum_{i=1}^{n-1} a_i 2^{i-1} - a_n 2^n$$

- Wertebereich: $\left[-2^{n-1}, 2^{n-1} - 1\right]$
- Dann bleibt nur noch die Entscheidung über die Größe von n und die Reihenfolge der einzelnen Bytes.
- Bei letzterem wird traditionell *big endian* gewählt (*network byte order*), siehe RFC 791, *Appendix B*, und RFC 951, 3. Abschnitt.

- Durch die *Google Protocol Buffers* wurde eine alternative Repräsentierung populär, bei der ganze Zahlen mit einer variablen Anzahl von Bytes dargestellt werden.
- Von den acht Bits wird das höchstwertige nur als Hinweis verwendet, ob die Folge fortgesetzt wird oder nicht: 1 = Folge wird fortgesetzt, 0 = Folge ist mit diesem Byte beendet.
- Die anderen sieben Bits (deswegen zur Basis 128) werden als Inhalt genommen, wobei sich Google für *little endian* entschied, d.h. die niedrigstwertigen Bits kommen zuerst.
- Dieses Verfahren ist jedoch ungünstig für negative Zahlen im Zweierkomplement, da dann für die -1 die maximale Länge zur Kodierung verwendet werden muss.
- Beispiel: 300 wird kodiert als Folge der beiden Bytes 0xac und 0x02 (binär: 1010 1100 0000 0010).

- Bei vorzeichenbehafteten ganzen Zahlen verwenden die *Google Protocol Buffers* die sogenannte Zickzack-Kodierung, die jeder ganzen Zahl eine nicht-negative Zahl zuordnet:

ganze Zahl	Zickzack-Kodierung
0	0
-1	1
1	2
-2	3
2147483647	4294967294
-2147483648	4294967295

- Das bedeutet, dass das höchst- und das niedrigstwertige Bit jeweils vertauscht worden sind. Bei 32-Bit-Zahlen sieht das dann so aus:

  $(n << 1) \hat{} (n >> 31)$

- IEEE-754 (auch IEC 60559 genannt) hat sich als Standard für die Repräsentierung von Gleitkommazahlen durchgesetzt.
- Eine Gleitkommazahl nach IEEE-754 besteht aus drei Komponenten:
  - dem Vorzeichen s (ein Bit),
  - dem aus q Bits bestehenden Exponenten $\{e_i\}_{i=1}^{q}$,
  - und der aus p Bits bestehenden Mantisse $\{m_i\}_{i=1}^{p}$.
- Für **float** und **double** ist die Konfiguration durch den Standard festgelegt, bei **long double** ist keine Portabilität gegeben:

Datentyp	Bits	q	p
float	32	8	23
double	64	11	52
long double		≥ 15	≥ 63

- Festzulegen ist hier nur, ob die Binärrepräsentierung in *little* oder *big endian* übertragen wird.

- Kommunikation ist entweder bilateral (der Normalfall) oder richtet sich an viele Teilnehmer gleichzeitig (*multicast*, *broadcast*).
- Bei einer bilateralen Kommunikation ergibt sich aus der Asymmetrie der Verbindungsaufnahme eine Rollenverteilung, typischerweise die eines Klienten und die eines Diensteanbieters.
- Diese Rollenverteilung bezieht sich immer auf eine konkrete Verbindung, d.h. es können zwischen zwei Kommunikationspartnern mehrere Verbindungen mit unterschiedlichen Rollenverteilungen bestehen.
- Über ein Protokoll wird geregelt, wie die einzelnen Mitteilungen aussehen und in welcher Abfolge diese gesendet werden dürfen.

- Klassischerweise existieren Netzwerkdienste, die angerufen werden können:
  - Diese sind sinnvoll, wenn die gleichen Aufgaben wiederkehrend zu lösen sind.
  - Es bleibt aber das Problem, wie diese Dienste gefunden werden und wie eine sinnvolle Lastverteilung zwischen konkurrierenden Anwendungen erfolgt.
- Bei variierenden Aufgabenstellungen muss ggf. eine Anwendung erst auf genügend Rechnerresourcen verteilt werden:
  - Wie erfolgt die Verteilung?
  - Wird die Umfang der Ressourcen zu Beginn oder erst im Laufe der Anwendung festgelegt?
  - Wie erfolgt die Verbindungsaufnahme untereinander?

- 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. Im September 2012 erschien die Version 3.0, die bei uns bislang nur auf der Thales unterstützt wird. Aktuell ist 3.1. 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_64_0/doc/html/mpi.html`).
- Es stehen mehrere Open-Source-Implementierungen zur Verfügung:
  - OpenMPI: `http://www.open-mpi.org/` (bei uns überall installiert)
  - MPICH: `http://www.mpich.org/`
  - MVAPICH: `http://mvapich.cse.ohio-state.edu/` (spezialisiert auf Infiniband)

- 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*), an Prozessgruppen (*multicast*) oder individuell versandt werden.
- Die Kommunikation kann sowohl synchron als auch asynchron erfolgen.
- Die Prozesse können in einzelne Gruppen aufgesplittet werden. Ein Prozess kann mehreren Gruppen angehören. Alle Prozesse gehören der globalen Gruppe an.

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 und 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 vorletzte 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([](double x) -> double {
      return 4 / (1 + x*x);
   }, 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	**char**
MPI_SIGNED_CHAR	**signed char**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_SHORT	**signed short**
MPI_INT	**signed int**
MPI_LONG	**signed long**
MPI_LONG_LONG	**signed long long int**
MPI_UNSIGNED_SHORT	**unsigned short**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned long**
MPI_UNSIGNED_LONG_LONG	**unsigned long long int**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**
MPI_LONG_DOUBLE	**long double**
MPI_WCHAR	**wchar_t**
MPI_CXX_BOOL	**bool**
MPI_CXX_FLOAT_COMPLEX	*std*::*complex*<**float**>
MPI_CXX_DOUBLE_COMPLEX	*std*::*complex*<**double**>
MPI_CXX_LONG_DOUBLE_COMPLEX	*std*::*complex*<**long double**>

Übersetzen von MPI-Anwendungen auf unseren Maschinen

Makefile

```
CC :=           mpicc
CXX :=          mpic++
CXXFLAGS :=     -g -Ofast -std=c++11
CPPFLAGS :=     -std=c++11
LDFLAGS :=      -std=c++11
```

- Wir verwenden OpenMPI auf unseren Rechnern.
- Statt den Übersetzern *g++* (und ggf. *gcc*) sind *mpic++* und *mpicc* zu verwenden.
- Dann sind alle MPI-spezifischen Header-Dateien und Bibliotheken automatisch zugänglich.
- Die Option *-Ofast* schaltet alle Optimierungen ein.

```
thales$ ls
Makefile  mpi-simpson.cpp
thales$ make
mpic++ -g -Ofast -std=c++11 -std=c++11  -c -o mpi-simpson.o mpi-simpson.cpp
mpic++ -o mpi-simpson -std=c++11 mpi-simpson.o
thales$ time mpirun -np 1 mpi-simpson 100000000
3.1415926535901

real    0m1.636s
user    0m1.563s
sys     0m0.038s
thales$ time mpirun -np 4 mpi-simpson 100000000
3.1415926535897

real    0m0.510s
user    0m1.604s
sys     0m0.108s
thales$
```

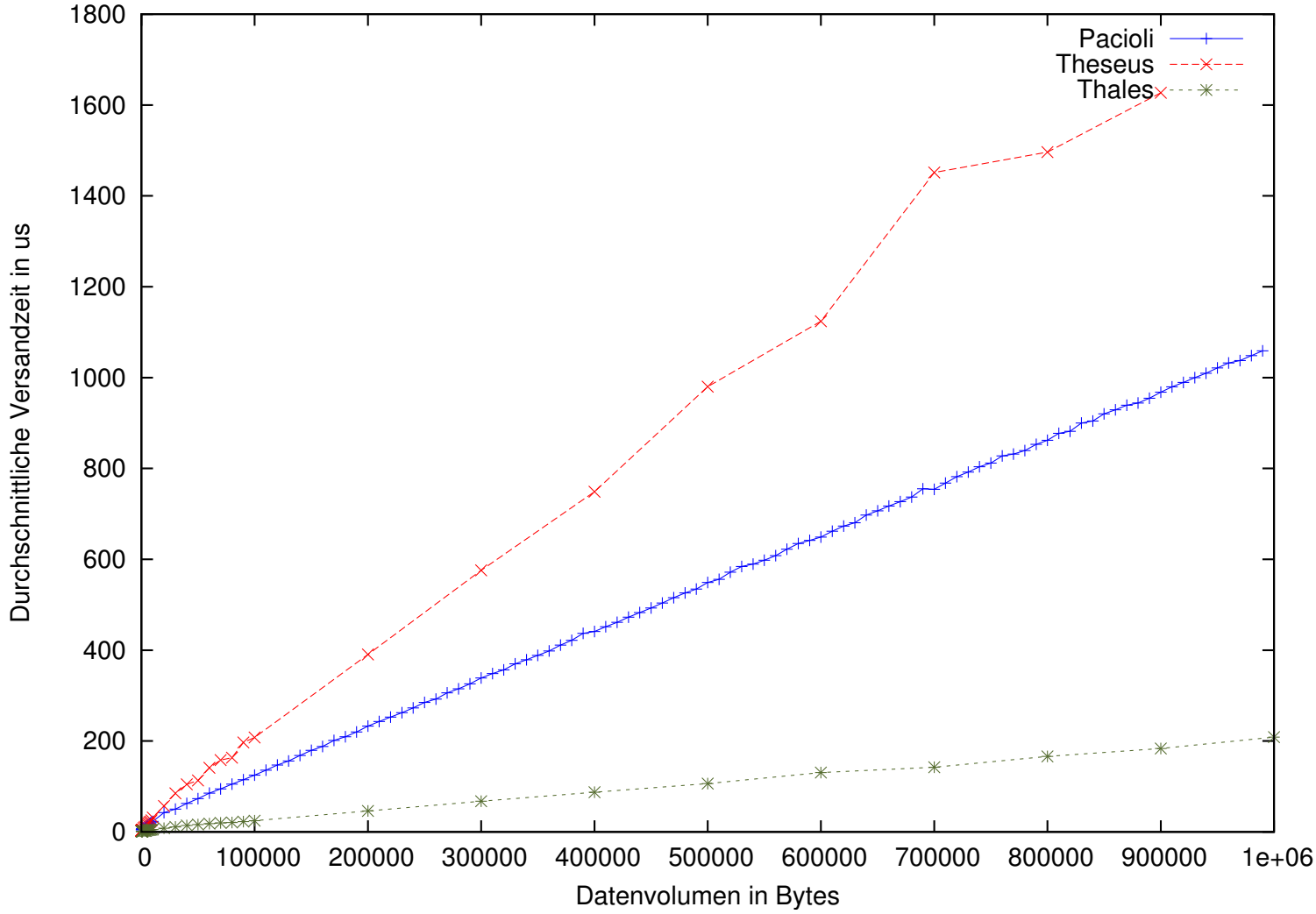
- Mit *mpirun* können MPI-Anwendungen gestartet werden.
- Wenn das Programm ohne *mpirun* läuft, dann gibt es nur einen einzigen Prozess.
- Die Option *-np* spezifiziert die Zahl der zu startenden Prozesse. Per Voreinstellung starten die alle auf der gleichen Maschine.

```
heim$ cat my-machines
multscher
syrlin
wolbach
heim
heim$ time mpirun -hostfile my-machines -np 4 \
> mpi-simpson 10000000 2>/dev/null
3.1415926535899

real    0m0.364s
user    0m0.136s
sys     0m0.040s
heim$
```

- Die Option *-hostfile* ermöglicht auf den Suns 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!

Gemessene Versandzeiten bei MPI



- Pacioli: 8 Prozesse, Infiniband. Gemeinsamer Speicher: Theseus: 6 Prozesse; Thales: 8 Prozesse (2 Intel X5650-Prozessoren, 2,6 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. (Thales: 0,62 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 aus Gründen der Einfachheit an einem Beispiel der Matrix-Vektor-Multiplikation demonstriert, wobei diese Technik in diesem konkreten Beispiel wegen des Kopieraufwands nichts bringt.

Rollenverteilung

Angenommen, die Matrix habe m Zeilen und uns stehen n Sklaven zur Verfügung. Der Einfachheit halber wird $m > n$ angenommen. Dann sehen die Rollen wie folgt aus:

- Master:
  - Verteile den Wert n und den Vektor an alle n Sklaven.
  - Versende jedem der n Sklaven eine Zeile der Matrix.
  - Insgesamt $m - n$ Mal: Empfange von irgendeinem Sklaven einen Wert des Resultatsvektors und schicke in Antwort eine weitere Zeile der Matrix.
  - Insgesamt n Mal: Empfange von irgendeinem Sklaven einen Wert des Resultatsvektors und signalisiere in der Antwort das Ende.
- Sklave:
  - Empfange den Wert n und den Vektor.
  - Für jede erhaltene Matrixzeile wird das entsprechende Skalarprodukt berechnet und verschickt.
  - Der Prozess endet, wenn das Ende signalisiert wird.

Seien $n = 2$ und $m = 4$. Dann kann das so in CSP übertragen werden:

$$
\begin{aligned}
P &= Master \parallel (Slave \,|||\, Slave) \\
Master &= broadcast\_parameters \rightarrow broadcast\_parameters \rightarrow \\
&\quad exchange\_row \rightarrow exchange\_row \rightarrow \\
&\quad exchange\_value \rightarrow exchange\_row \rightarrow \\
&\quad exchange\_value \rightarrow exchange\_row \rightarrow \\
&\quad exchange\_value \rightarrow finish \rightarrow \\
&\quad exchange\_value \rightarrow finish \rightarrow \\
&\quad SKIP_{\alpha Master} \\
Slave &= broadcast\_parameters \rightarrow WorkingSlave \\
WorkingSlave &= exchange\_row \rightarrow exchange\_value \rightarrow WorkingSlave \mid \\
&\quad finish \rightarrow SKIP_{\alpha Slave}
\end{aligned}
$$

MPI und CSP kommen sich in der Ausdrucksform hier sehr nahe:

- Die Datenübertragung erfolgt im einfachsten Falle synchron. Die *exchange_row*- und *exchange_value*-Ereignisse entsprechen jeweils einer Paarung von *MPI_Send* und *MPI_Recv*, die ebenfalls synchron erfolgen sollten.
- Bei *MPI_Send* und *MPI_Recv* wird zusätzlich noch eine Markierung in Form eines ganzzahligen Werts mit übertragen, der die Art der Nachricht charakterisiert (*tag value*). Dieser Wert kann verwendet werden, um auf der Seite des Sklaven das Empfangen einer weiteren Zeile von dem Empfangen des Endesignals unterscheiden zu können. Im Beispiel werden hier die Werte *NEXT_ROW* und *FINISH* verwendet.

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);
   }
}
```

Beachtenswert ist hier, dass bei der Übertragung eines Arrays die Länge dynamisch gewählt weden kann:

- Beim Versenden der nächsten Matrixzeile werden nebem dem *tag value* noch *n* Werte übermittelt:

```
MPI_Send(A[row], n, MPI_DOUBLE, slave, NEXT_ROW, MPI_COMM_WORLD);
```

- Beim Übermitteln des Endesignals wird als Array-Länge die 0 angegeben, d.h. es wird nur *FINISH* übertragen:

```
MPI_Send(0, 0, MPI_DOUBLE, slave, FINISH, MPI_COMM_WORLD);
```

Die Kombination von ganzzahligen Paketarten (hier *NEXT_ROW* oder *FINISH*) mit dynamischen Arrays vermeidet die Aufsplittung solcher Pakete in getrennte Header- und Datenpakete, die die Latenzzeiten erhöhen würden.

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 durch das Auslesen von *status.MPI_SOURCE*.