

## 5 Der *Local Area Multicomputer* (LAM-MPI)

### 5.1 MPI-Umgebung an der Hochschule Fulda

- Parallelverarbeitung in verteilten (heterogenen) Umgebungen

1. vernetzte Rechner ohne *Network File System* (NFS)

⇒ eigenständige *HOME*-Verzeichnisse auf jedem Rechner

⇒ jedes Programm muss auf jedem Rechner übersetzt werden

2. vernetzte, homogene Rechner mit NFS

⇒ ein *HOME*-Verzeichnis für alle Rechner

⇒ jedes Programm wird nur auf einem Rechner übersetzt

3. vernetzte, heterogene Rechner mit einer NFS-Domäne für jeden Rechnertyp (ggf. auch homogene Rechner in verschiedenen NFS-Domänen)

⇒ eigenständige *HOME*-Verzeichnisse in jeder NFS-Domäne

⇒ jedes Programm muss einmal in jeder NFS-Domäne übersetzt werden

4. vernetzte, heterogene Rechner mit nur einer NFS-Domäne

⇒ ein *HOME*-Verzeichnis für alle Rechner(typen)

⇒ jedes Programm muss einmal auf jedem Rechnertyp übersetzt werden

⇒ Programme für verschiedene Rechnertypen müssen z. B. in verschiedenen Verzeichnissen gespeichert werden

(an der HS Fulda in  $\$HOME/\{SYSTEM\_ENV\}/\{MACHINE\_ENV\}/bin$  mit  $SYSTEM\_ENV = SunOS, Linux, usw.$  und  $MACHINE\_ENV = sparc, x86, usw.$ ; das Verzeichnis ist in  $PATH$  enthalten, damit die Programme gefunden werden.)

- **Programmerzeugung erfolgt immer über *Makefile* oder *GNUmakefile***  
(wurde am Ende des Kapitels "C für Java-Programmierer" im Skript zur Lehrveranstaltung "Betriebssystem-Praktikum" erklärt)
- **bei MPI wird der Compiler durch das spezielle Programm *mpicc* (*compiler wrapper*) aufgerufen**

```
# GNUmakefile for heterogeneous environments for the programs
# ...
# Usage:
#   (g)make          make both programs
#   (g)make clean    remove object files
#   (g)make clean_all  remove object files and executables
#
# Possible procedure of operation:
#   make
#   make clean          remove object files for this platform
#   ssh <new host>      <----+
#   cd <this directory> |
#   make                | pass through all platforms
#   make clean          |
#   exit                ----+
#
# ...

# You should always use the wrapper compiler "mpicc". "mpicc -show"
# or "mpicc -showme" (not available with MPICH) print the "default
# compiler" and the options which are compiled into "mpicc". Later
# you can specify additional options for the compiler.

CC      = mpicc
...
# necessary files

FILE1   = hello_2_mpi
FILE2   = hello_2_slave_mpi
...
# Determine operating system and processor architecture. The variables
# MYOS, MYARCH, BINARY, CC, CFLAGS, and LDFLAGS will be set according
# to the platform.

include GNUmakefile.env
...
ifeq ($(SYSTEM_ENV), Cygwin)
    TARGET1 = $(HOME)/$(SYSTEM_ENV)/$(MACHINE_ENV)/bin/$(FILE1).exe
...

```

(Sie können **GNUmakefile.env** im Programmarchiv zu dieser Lehrveranstaltung anschauen, wenn es Sie interessiert, wie die Rechnerumgebung bestimmt wird und wie die verschiedenen Umgebungsvariablen gesetzt werden müssen. Die *Shell*-Skripte ***make\_compile*** und ***make\_clean\_all*** unterstützen Sie bei der Verwaltung Ihrer Programme in einer heterogenen Welt. Passen Sie die Rechnerliste in den Skript-Dateien bitte an, bevor Sie sie das erste Mal benutzen.)

- auf entfernten Rechnern müssen Kommandos ausgeführt werden können, ohne dass ein Passwort eingegeben werden muss
  - die Benutzerumgebung muss hierfür in Abhängigkeit vom zu benutzenden Kommunikationsmechanismus vorbereitet werden
  - typische Kommunikationsmechanismen
    - ◆ *Remote Shell* (rsh; wird an der HS Fulda nicht unterstützt)
      - ⇒ die Datei `$HOME/.rhosts` muss erzeugt werden
        - geringe Sicherheit (Daten sind nicht verschlüsselt)
        - schwache Authentifizierung (über "reservierte *Ports*")
        - `chmod 600 $HOME/.rhosts`  
(aus Sicherheitsgründen darf nur der Eigentümer Zugriff auf diese Datei haben; sie erlaubt ein *login* auf fremde Rechner **ohne** Passwort!)
      - ⇒ "`rsh <Rechnername> uname -a`" muss für alle Rechner der Datei "`$HOME/.rhosts`" ohne Fehler, Warnungen oder unerwartete Ausgaben funktionieren
    - ◆ *Secure Shell* mit rechnerbasierter Authentifizierung (ssh)
      - ⇒ die Datei `$HOME/.shosts` muss erzeugt werden
        - Systemadministrator muss die öffentlichen Schlüssel der vertrauenswürdigen Rechner sammeln und speichern
        - sichere, verschlüsselte Datenübertragung
        - sehr strenge Authentifizierung
        - `chmod 600 $HOME/.shosts`  
(aus Sicherheitsgründen darf nur der Eigentümer Zugriff auf diese Datei haben; sie erlaubt ein *login* auf fremde Rechner **ohne** Passwort!)
      - ⇒ "`ssh <Rechnername> uname -a`" muss für alle Rechner der Datei "`$HOME/.shosts`" ohne Fehler, Warnungen oder unerwartete Ausgaben funktionieren

- ◆ *Secure Shell* mit benutzerbasierter Authentifizierung (ssh)
  - ⇒ der Benutzer muss einen privaten und einen öffentlichen Schlüssel erzeugen
  - alle öffentlichen Schlüssel des Benutzers müssen in der Datei *\$HOME/.ssh/authorized\_keys* gesammelt werden (bei OpenSSH)
    - (falls der Benutzer nur einen Benutzernamen auf allen Rechnern verwendet, reicht ein Schlüsselpaar aus; das obige Verzeichnis inklusive aller Dateien muss auf allen Rechnern sichtbar sein (NFS) bzw. eingerichtet werden)
  - sichere, verschlüsselte Datenübertragung
  - sehr strenge Authentifizierung
  - Benutzung der privaten Schlüssel zur Authentifizierung muss immer mit der *SSH-Passphrase* erlaubt werden
  - die Authentifizierung kann an einen Authentifizierungs-Agenten übertragen werden
    - (notwendig für MPI)
- ⇒ "ssh <Rechnername> uname -a" muss für alle gewünschten Rechner ohne Fehler, Warnungen oder unerwartete Ausgaben funktionieren

- Aktivierung der Umgebung für MPI
  - folgende Entscheidungen treffen
    - ◆ welcher Kommunikationsmechanismus soll benutzt werden  
(**empfohlen:** *Secure Shell* mit rechnerbasierter Authentifizierung)
    - ◆ welche MPI-Implementierung soll benutzt werden
  - Datei *\$HOME/.cshrc* entsprechend anpassen
    - ⇒ Kommentarzeichen vor "set MPI" entfernen  
(beginnen Sie mit der aktuellsten Version von LAM-MPI)
    - ⇒ ggf. Kommentarzeichen vor "set USESSH" entfernen  
(benutzen Sie zunächst weiterhin die vorausgewählte Variante)
  - richten Sie die Kommunikationsumgebung ein  
(Dieser Punkt entfällt, wenn Sie *Secure Shell* bereits in der Lehrveranstaltung "Betriebssystempraktikum" eingerichtet haben.)

**Beispiel:** *Secure Shell* mit rechnerbasierter Authentifizierung

```
cp -i /opt/global/config/shosts $HOME/.shosts
```

```
chmod 600 $HOME/.shosts
```

in *\$HOME/.shosts* xxxx durch eigenen Benutzernamen ersetzen

- *Logout* und *Login* durchführen
  - ⇒ abhängig vom gewählten Kommunikationsmechanismus wird das Passwort und ggf. die *SSH-Passphrase* abgefragt
  - ⇒ Umgebungsvariablen werden gesetzt bzw. erweitert (z. B. *LAMHOME* oder *PATH*)
  - ⇒ Details können */opt/global/cshrc\** und */opt/global/mpi.csh* entnommen werden
  
- ggf. "ssh" auf je einen Rechner mit anderem Betriebssystem oder anderer Architektur durchführen, der dasselbe *HOME*-Verzeichnis benutzt, damit die erforderliche Umgebung eingerichtet wird (an der Hochschule Fulda nicht erforderlich)
  
- ggf. die obigen Schritte für alle *HOME*-Verzeichnisse wiederholen, die nicht über NFS erreichbar sind (an der Hochschule Fulda nicht erforderlich)

- weitere Hinweise
  - *csh* oder *tcsh* **muss** als *Login-Shell* benutzt werden  
(nur für diese *Shell* werden die erforderlichen Skripte zur Einrichtung der Umgebung zur Verfügung gestellt)
  - jede *C-Shell* benutzt eine *Hash*-Tabelle für alle Kommandos, die über die Umgebungsvariable *PATH* erreichbar sind
    - ⇒ nach der Übersetzung eines **neuen** Programms muss die *Hash*-Tabelle neu erzeugt bzw. aktualisiert werden  
(andernfalls wird die ausführbare Datei des neuen Programms nicht gefunden, wenn sie sich nicht im aktuellen Verzeichnis befindet)
    - ⇒ die *Hash*-Tabelle wird mit dem Kommando *rehash* aktualisiert
  - alle Rechner müssen dieselbe Zeit haben
    - ⇒ unterschiedliche Uhrzeiten können zu Problemen führen
    - ⇒ die Uhrzeiten sollten z. B. über *xntp* synchronisiert werden

- einige Fehler und deren Beseitigung
  - das Laufzeitsystem von MPI liefert u. U. einige Ausgaben und dann *Permission denied*
    - ⇒ die Datei `~/ .rhosts` bzw. `~/ .shosts` fehlt auf einem Rechner, enthält fehlerhafte Einträge oder zu wenig Einträge
    - ⇒ auf einem Rechner enthält die Datei `~/.ssh/authorized_keys` fehlerhafte oder zu wenig Einträge oder es fehlen Dateien für die benutzerbasierte Authentifizierung der *Secure Shell*
  - das Laufzeitsystem von MPI liefert u. U. einige Ausgaben und dann *Command not found*
    - ⇒ falls heterogene Rechner an der Parallelverarbeitung beteiligt sind oder die Rechner zu unterschiedlichen NFS-Domänen gehören oder eigenständig sind, haben Sie u. U. vergessen, das Programm auf **jedem** unterschiedlichen/eigenständigen System zu übersetzen
    - ⇒ u. U. haben Sie vergessen, *rehash* einzugeben, so dass das Programm auf dem lokalen Rechner nicht gefunden wird

- in heterogenen Umgebungen können Probleme mit inkompatiblen Datentypen auftreten

Betriebs-system	Compiler	double	long double
Solaris Sparc	Sun C/C++ v5.7 (Visual Studio 10)	8 Bytes, 53 Bit Mantisse	16 Bytes, 113 Bit Mantisse
	GNU gcc v4.1.1	8 Bytes, 53 Bit Mantisse	16 Bytes, 113 Bit Mantisse
Solaris x86	Sun C/C++ v5.8 (Visual Studio 11)	8 Bytes, 53 Bit Mantisse	12 Bytes, 64 Bit Mantisse
	GNU gcc v4.1.1	8 Bytes, 53 Bit Mantisse	12 Bytes, 64 Bit Mantisse
Linux x86	Intel C/C++ 9.0	8 Bytes, 53 Bit Mantisse	12 Bytes, 64 Bit Mantisse
	GNU gcc v4.0.2	8 Bytes, 53 Bit Mantisse	12 Bytes, 64 Bit Mantisse
Cygwin x86	GNU gcc v3.4.4	8 Bytes, 53 Bit Mantisse	12 Bytes, 64 Bit Mantisse
Windows x86	Microsoft Visual Studio 2005	8 Bytes, 53 Bit Mantisse	8 Bytes, 53 Bit Mantisse
	Borland C/C++ v5.5	8 Bytes, 53 Bit Mantisse	10 Bytes, 64 Bit Mantisse
	DevC++ v4.9.9.2	8 Bytes, 53 Bit Mantisse	12 Bytes
	lcc-win32 v3.3	8 Bytes, 53 Bit Mantisse	12 Bytes, 64 Bit Mantisse

(Die Compiler unterstützen für "long double" 64-Bit IEEE-754-Gleitkommazahlen (53-Bit Mantisse), 80-Bit-IEEE-754-Gleitkommazahlen (64-Bit Mantisse) oder 128-Bit-IEEE-754-Gleitkommazahlen (113-Bit Mantisse). "MS VS 2005" liefert für 64-Bit-Prozessoren dieselben Ergebnisse, wenn man die 64-Bit-Umgebung über "\Programme\Microsoft Visual Studio 8\VC\bin\amd\vcvarsamd64.bat" einrichtet. Die "UNIX"-Compiler unterstützen für AMD64-Prozessoren weiterhin 80-Bit-Gleitkommazahlen, wobei "long double" dann teilweise 16 Bytes groß ist. DevC++ gibt für "long double" nur die Größe der Datenstruktur korrekt aus. Die Werte der Tabelle wurden mit dem Programm *Floatingpoint.c* aus dem Programmarchiv ermittelt.)

⇒ nur *double* ist für heterogene Umgebungen geeignet

(eine Berechnung von  $\pi$  mit `MPI_LONG_DOUBLE` in einer heterogenen Umgebung liefert im Allgemeinen unerwartete Ergebnisse:  $\pi$  kann beliebig groß werden und verschiedene virtuelle Rechner liefern im Allgemeinen verschiedene Ergebnisse)

**Aufgabe:**

- 1) Richten Sie die Benutzerumgebung für MPI ein.
- 2) Benutzen Sie die Beispielprogramme zu dieser Lehrveranstaltung.  
Laden Sie ggf. die Datei *prog.zip* bzw. *prog.tar.gz* von der Webseite <http://www2.hs-fulda.de/~gross/parallel/<Datei>> und packen Sie sie aus:

"unzip prog.zip" bzw. "tar zxvf prog.tar.gz"

⇒ es wird ein Verzeichnis *prog* mit mehreren Unterverzeichnissen erzeugt

- 3) Testen Sie Ihre neue Umgebung
  - a) `cd prog/mpi/hello_1`
  - b) passen Sie die Datei *hosts.lam-mpi* an
    - ◆ der erste Eintrag für "<Rechnername> <Benutzername>" wird durch den Rechnernamen Ihres lokalen Rechners und Ihren Benutzernamen ersetzt
    - ◆ fügen Sie zwei weitere Einträge hinzu
  - c) führen Sie *(g)make* und *rehash* aus
  - d) starten Sie einen virtuellen Rechner und führen Sie das Programm aus

(die Vorgehensweise ist am Anfang der Datei *hello\_1\_mpi.c* beschrieben)

⇒ wenn Sie die Umgebung richtig eingerichtet haben, erhalten Sie die Ausgabe des Programms

## 5.2 MPI-Grundlagen

- der *Message-Passing Interface* Standard wird vom *Message-Passing Interface Forum* entwickelt (ca. 100 Personen aus ca. 50 Organisationen)
  - April 1992: *Workshop on Standards for Message Passing in a Distributed Memory Environment* (Williamsburg, Virginia)
    - ⇒ Diskussion der grundlegenden Eigenschaften
    - ⇒ Einrichtung einer Arbeitsgruppe zur Erarbeitung eines Standards
  - November 1992: vorläufiger Konzeptvorschlag MPI-1  
Konstituierung des MPI-Forums
  - Juni 1994: MPI 1.0
  - Juni 1995: MPI 1.1 (Korrekturen, Klarstellungen)
  - Juli 1997: MPI 1.2 (Korrekturen, Klarstellungen; enthalten in MPI-2, Kap. 3)  
MPI 2.0 (Erweiterungen von MPI-1)
  - Juni 2008: MPI 2.1

- Entwicklungsziele
  - leicht handhabbare, einfach portierbare, leistungsfähige und flexible Schnittstelle zur Nachrichtenübertragung
  - standardisierte Mechanismen zur Kommunikation in Systemen mit verteiltem Speicher (distributed memory)
  - effiziente Kommunikation
    - ◆ Speicher-Speicher-Kopien vermeiden
    - ◆ Überlagerung von Kommunikation und Berechnung
    - ◆ Auslagerung auf Kommunikations-Coprozessoren (falls verfügbar)
  - Unterstützung heterogener Umgebungen
  - Unterstützung der Sprachen *C* und *Fortran 77*
  - Schnittstellensemantik unabhängig von der Programmiersprache
  - es soll eine zuverlässige Kommunikationsschnittstelle zur Verfügung stehen (der Benutzer muss sich nicht um Übertragungsfehler kümmern)
  - die Schnittstelle soll sich nicht übermäßig von vorhandenen Schnittstellen (z. B. PVM) unterscheiden, aber Erweiterungen für eine größere Flexibilität bieten
  - die Schnittstelle soll von vielen Herstellern ohne größere Änderungen ihrer Kommunikations- und System-Software implementiert werden können
  - die Schnittstelle soll so entworfen werden, dass man sie *thread-safe* implementieren kann

- seit Mitte 1994 wird an Implementierungen von MPI-1 gearbeitet
  - MPICH      Implementierung der *Mississippi State University*
  - LAM        Implementierung des *Ohio Supercomputer Center* der *Ohio State University* (als Projekt bis ca. Ende 1997)

seit ca. 1998 als *OpenSource*-Projekt an der Universität *Notre Dame*

– ...

### ⇒ LAM

- ◆ vollständige MPI-1 Implementierung mit zusätzlichen MPI-2 Funktionen
  - dynamische Erzeugung von Prozessen
  - direkter entfernter Speicherzugriff (one-sided communication)
  - ...
- ◆ grafische Oberfläche zum Ausführen, Testen und visualisieren von MPI-Programmen (xmpi)
- ◆ Quellcode-kompatibel zu anderen Implementierungen (solange nur Funktionen von MPI-1 benutzt werden)
- ◆ unterstützt heterogene Multicomputersysteme
- ◆ unterstützt Multiprozessorsysteme (ab lam-6.5)

- LAM-MPI besteht aus zwei Teilen
  - einem Dämon-Prozess *lamd* auf jedem Rechner des Multi-computers
  - einer Bibliothek mit den MPI-Funktionen (*libmpi.a*)
  
- Was ist im MPI-1 Standard enthalten?
  - Punkt-zu-Punkt Kommunikation
  - Gruppenkommunikation (z. B. *Broadcast*)
  - Prozessgruppen
  - Kontexte (Partitionierung des Kommunikationsraums)
  - Prozesstopologien (z. B. 2D- oder 3D-Gitter)
  - Verwaltung der Umgebung (z. B. Fehlerbehandlung, Uhren)
  - Schnittstelle zur Leistungsmessung (*profiling interface*)
  - Schnittstellen für *C* und *Fortran 77*

- Was ist im MPI-1 Standard nicht enthalten?
  - explizite Operationen für gemeinsamen Speicher (*shared memory*)
  - Operationen, die eine große Unterstützung durch das Betriebssystem erfordern (z. B. unterbrechungsgesteuertes Empfangen)
  - Werkzeuge zur Programmerstellung
  - Hilfsmittel zum *Debuggen*
  - explizite Unterstützung von *Threads*
  - Unterstützung zur Prozessverwaltung
  - Ein-/Ausgabefunktionen

- Was ist in der MPI-2 Erweiterung des Standards enthalten?
  - dynamische Prozesserzeugung und -verwaltung
  - direkter entfernter Speicherzugriff (remote memory access (RMA)  $\Rightarrow$  one-sided communication)  

Normalerweise ruft bei einer Nachrichtenübertragung ein Prozess eine *Send*-Operation und ein anderer die zugehörige *Receive*-Operation auf. Jetzt kann ein Prozess alleine für die Nachrichtenübertragung verantwortlich sein.
  - neue/erweiterte Gruppenkommunikationsoperationen
  - Unterstützung von *Threads*
  - parallele Dateioperationen
  - Schnittstellen für *C++* und *Fortran 90*
  - ...
  
- MPI ist aufwärts kompatibel  
(jedes MPI-1.1-Programm ist ein MPI-1.2-Programm usw.)

- grundlegende Konzepte
  - alle Prozesse werden beim Start des Programms mit *mpiexec* erzeugt
  - ausgefeiltes Konzept für Prozessgruppen
    - ◆ alle Prozesse werden durch eine fortlaufende Nummer identifiziert (*process rank*, Zahl zwischen 0 und n-1 bei n Prozessen)
    - ◆ Prozessgruppen definieren einen Namensraum für Prozesse bei Punkt-zu-Punkt Operationen und Gruppenkommunikationsoperationen  
(ein Prozess kann in verschiedenen Prozessgruppen unterschiedliche Prozessnummern haben)
    - ◆ alle Prozesse sind in der Basisprozessgruppe enthalten
    - ◆ neue Gruppen können über Prozessnummernbereiche oder über Mengenoperationen auf bestehende Prozessgruppen gebildet werden
    - ◆ Prozesse können verschiedenen Kommunikationskontexten angehören
    - ◆ Prozessgruppen können unabhängig von Kommunikationskontexten benutzt werden
    - ◆ Kommunikationsoperationen können nur mit Kommunikatoren aufgerufen werden

- Kommunikatoren (*communicator*)
  - ◆ repräsentieren einen Kommunikationskontext  
(Namensraum für alle Kommunikationsoperationen)
  - ◆ es gibt zwei Typen
    - interne Kommunikatoren (*intra-communicators*) für Operationen innerhalb einer Prozessgruppe
    - übergreifende Kommunikatoren (*inter-communicators*) für Punkt-zu-Punkt Operationen zwischen zwei Prozessgruppen
  - ◆ am häufigsten werden interne Kommunikatoren benutzt, die folgende Attribute beinhalten
    - die Prozessgruppe
    - die Topologie, die die logische Anordnung der Prozesse innerhalb der Gruppe beschreibt
  - ⇒ der Kommunikator enthält alle für die Kommunikation erforderlichen Daten (Zustände, Prozessnummern, ...)
  - ◆ für die Basisprozessgruppe gibt es den Kommunikator `MPI_COMM_WORLD`
  - ◆ *source* und *destination* in einer Kommunikationsoperation beziehen sich immer auf die Prozessnummer eines Prozesses innerhalb der Gruppe, die der Kommunikator identifiziert

- Kommunikationskontexte
  - ◆ repräsentieren verschiedene, sichere Kommunikationswelten
  - ◆ die Kontexte entsprechen einer Partitionierung des Kommunikationsadressraums
  - ◆ jede Kommunikation zwischen Prozessen läuft in einem Kommunikationskontext ab
  - ◆ verschiedene Kommunikationskontexte können teilweise oder vollständig überlappende Prozessgruppen repräsentieren
    - ⇒ ein Prozess wird in verschiedenen Kommunikationskontexten u. U. durch unterschiedliche Prozessnummern repräsentiert
  - ◆ jeder Kommunikationskontext unterstützt eine unabhängige Folge von Kommunikationsoperationen
    - ⇒ ein Prozess kann mit einem anderen Prozess u. U. über verschiedene Kommunikationskontexte kommunizieren
    - ⇒ die Kommunikationsoperationen eines Prozesses in verschiedenen Kontexten sind voneinander unabhängig
  - ◆ es ist sichergestellt, dass sich Punkt-zu-Punkt Operationen und Gruppenkommunikationsoperationen gegenseitig nicht stören

- virtuelle Topologie
  - ◆ erlaubt eine logische Anordnung der Prozesse
    - in 2D- und 3D-Gitterstrukturen
    - in allgemeinen Graphen
  - ◆ unterstützt die Anforderungen paralleler Algorithmen an bestimmte Nachbarschaftsbeziehungen zwischen Prozessen (z. B. beim Lösen von partiellen Differentialgleichungen)
  
- es können beliebig strukturierte Nachrichten übertragen werden, die nicht einmal zusammenhängend sein müssen
  - ◆ die Nachricht wird durch eine Anfangsadresse, einen Datentyp und die Anzahl der Daten des Datentyps festgelegt
  - ◆ aufgrund des Datentyps müssen die Daten nicht mehr zusammenhängend sein (z. B. die Spaltenlemente einer zeilenweise gespeicherten Matrix)
  - ◆ MPI-Bibliothek sorgt automatisch für alle erforderlichen Datenkonvertierungen in einer heterogenen Umgebung

- vordefinierte Datentypen
  - ◆ MPI\_CHAR
  - ◆ MPI\_SHORT
  - ◆ MPI\_INT
  - ◆ MPI\_LONG
  - ◆ MPI\_UNSIGNED\_CHAR
  - ◆ MPI\_UNSIGNED\_SHORT
  - ◆ MPI\_UNSIGNED
  - ◆ MPI\_UNSIGNED\_LONG
  - ◆ MPI\_FLOAT
  - ◆ MPI\_DOUBLE
  - ◆ MPI\_LONG\_DOUBLE
  - ◆ MPI\_BYTE
  - ◆ MPI\_PACKED
  
- selbstdefinierte Datentypen
  - ◆ Sequenz von Paaren (*basic data type, offset*)
  - ◆ diese Sequenz heißt *typemap*
  - ◆ wird mit Hilfe entsprechender MPI-Funktionen erstellt

### ◆ Beispiel 1: Parameter für Mandelbrotmenge

```

struct { char    display[50];        /* name of display          */
        int     maxiter;            /* max # of iterations     */
        double  xmin, ymin;        /* lower left corner       */
        double  xmax, ymax;        /* upper right corner      */
        int     width, height;     /* window size in pixels   */
        } cmdline;

/* set up MPI data type */
int      blockcounts[4] = {50, 1, 4, 2};
MPI_Datatype types[4] = {MPI_CHAR, MPI_INT, MPI_DOUBLE, MPI_INT};
MPI_Aint   offsets[4];
MPI_Datatype cmdtype;          /* new data type for cmdline */

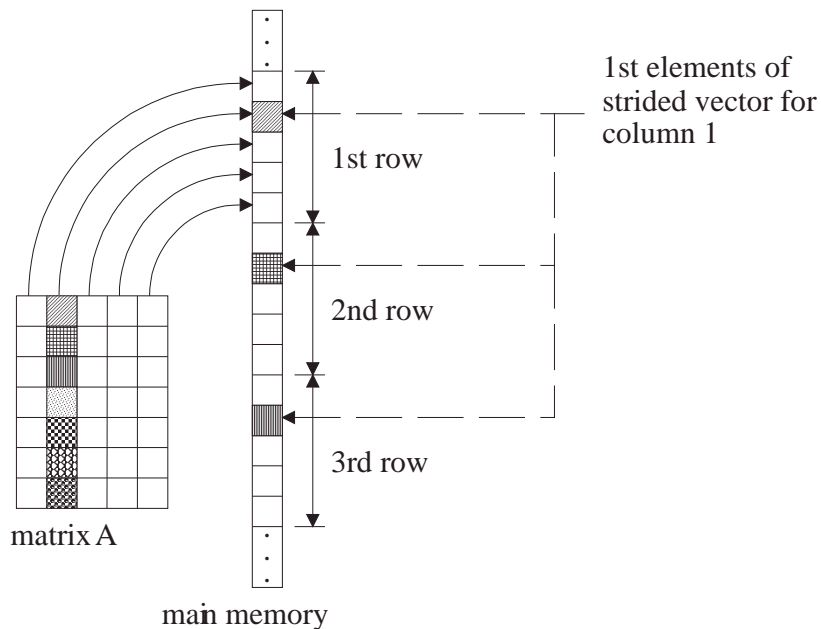
MPI_Address (&cmdline.display, &offsets[0]);
MPI_Address (&cmdline.maxiter, &offsets[1]);
MPI_Address (&cmdline.xmin,    &offsets[2]);
MPI_Address (&cmdline.width,   &offsets[3]);
/* adjust offsets to relative offsets */
for (i = 3; i >= 0; --i)
{
    offsets[i] -= offsets[0];
}
/* build the new type */
MPI_Type_struct (4, blockcounts, offsets, types, &cmdtype);
MPI_Type_commit (&cmdtype);

/* now we are ready to broadcast the structure */
MPI_Bcast (&cmdline, 1, cmdtype, 0, MPI_COMM_WORLD);

```

◆ **Beispiel 2:** verteilte Daten (*strided vector*)

geg.: zeilenweise gespeicherte (m, n)-Matrix



```

#define P          6          /* # of rows          */
#define Q         10         /* # of columns       */

double matrix[P][Q],
       column[P];
MPI_Datatype column_t,      /* column type (strided vector)*/
             tmp_column_t;  /* needed to resize the extent */

/* Build the new type for a strided vector and resize the extent
 * of the new datatype in such a way that the extent of the
 * whole column looks like just one element so that the next
 * column starts in matrix[0][i] in MPI_Scatter/MPI_Gather.
 */
MPI_Type_vector (P, 1, Q, MPI_DOUBLE, &tmp_column_t);
MPI_Type_create_resized (tmp_column_t, 0, sizeof (double),
&column_t);
MPI_Type_commit (&column_t);
MPI_Type_free (&tmp_column_t);

MPI_Scatter (matrix, 1, column_t, column, P, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

MPI_Gather (column, P, MPI_DOUBLE, matrix, 1, column_t, 0,
MPI_COMM_WORLD);

MPI_Send (&matrix[0][j], 1, column_t, ...);

```

– **Aufgabe 5 - 1:**

Erstellen Sie ein Programm, das die Spalten einer zeilenweise gespeicherten Matrix an Arbeitsprogramme schickt. Ein Arbeitsprogramm führt auf allen Spaltenelementen eine vorgegebene Operation aus (z. B. "quadrieren" der Elemente bei geraden Spaltennummern und "verdoppeln" der Elemente bei ungeraden Spaltennummern) und sendet das Ergebnis zurück an das Hauptprogramm, das dann die Ergebnismatrix ausgibt.

– verschiedene Arten der Punkt-zu-Punkt Kommunikation

◆ blockierend/nicht blockierend (Immediate)

(z. B.: MPI\_Send/MPI\_Isend)

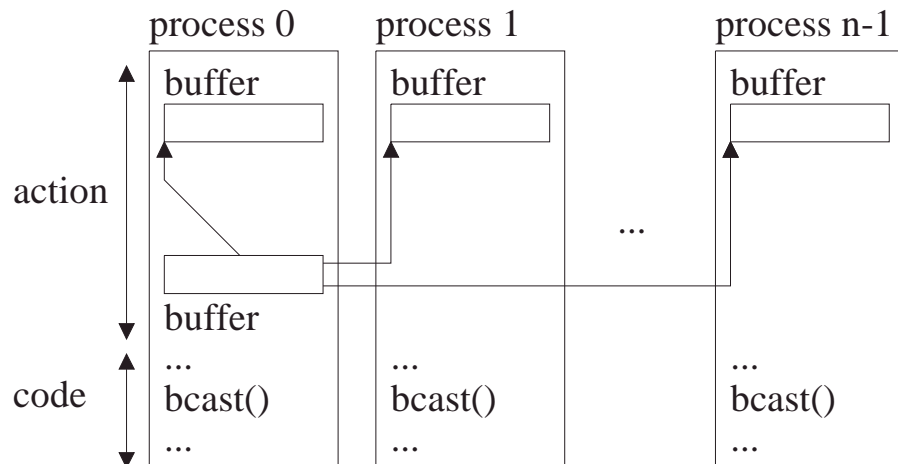
- *blockierend*: Rückkehr vom *Send*-Aufruf, sobald die Daten im Empfangspuffer des Empfängers **oder** in einem temporären Systempuffer gespeichert sind (MPI entscheidet, ob die Nachricht zwischengespeichert wird)
- *nicht blockierend*: sofortige Rückkehr; später muss geprüft werden, ob das Senden/Empfangen erfolgreich war
- blockierende und nicht blockierende Sende- und Empfangsaufrufe können beliebig gemischt werden

- ◆ verschiedene Modi: *standard*, *buffered*, *synchronous*, *ready*  
(z. B.: MPI\_Send, MPI\_Bsend, MPI\_Ssend, MPI\_Rsend/MPI\_Isend, MPI\_Ibsend, MPI\_Issend, MPI\_Irsend)
  - *standard*: siehe oben
  - *buffered*: falls der Empfangsprozess noch nicht auf die Daten wartet, **muss** MPI die Daten zwischenspeichern
  - *synchronous*: der Sender wird blockiert bis der Empfänger die Daten bearbeiten will, d. h., wenn blockierende *Send*- und *Receive*-Operationen benutzt werden, findet ein Rendezvous statt
  - *ready*: die *Send*-Operation darf nur gestartet werden, wenn der Empfänger bereits auf die Nachricht wartet  
(Dient der Leistungssteigerung, da MPI nicht überprüfen muss, ob der Empfänger bereits eine *Receive*-Operation gestartet hat. Falls der Empfänger nicht auf die Nachricht wartet, ist das Ergebnis der Operation undefiniert! In einem korrekten Programm kann jedes *Ready-Send* durch ein *Standard-Send* ersetzt werden, ohne das Verhalten des Programms zu ändern (u. U. wird es allerdings langsamer))

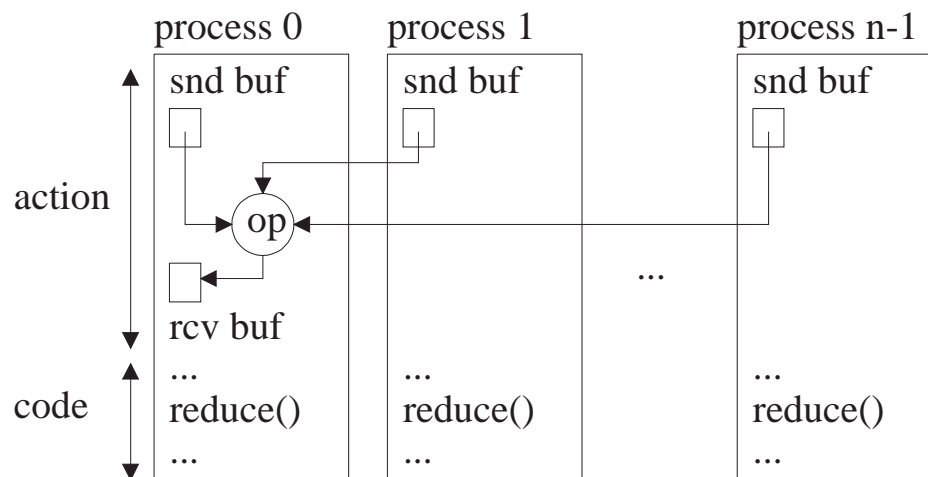
- Gruppenkommunikation (*collective communication*)
  - ◆ Datenübertragungen
    - Umverteilung der Daten zwischen den Prozessen
    - Rundumverteilung (*broadcast*), Streuung (*scattering*), Sammlung (*gathering*)
  - ◆ Gruppenoperationen (*collective computation operations*)
    - Minimum/Maximum von verteilten Daten bestimmen
    - logische Verknüpfungen von verteilten Daten
    - Summenbildung verteilter Daten
    - ...
  - ◆ MPI nutzt die Kenntnis über die Struktur des Kommunikationskontextes aus, um die Parallelität der Operationen zu optimieren

- ◆ grafische Darstellung einiger Gruppenkommunikationsoperationen

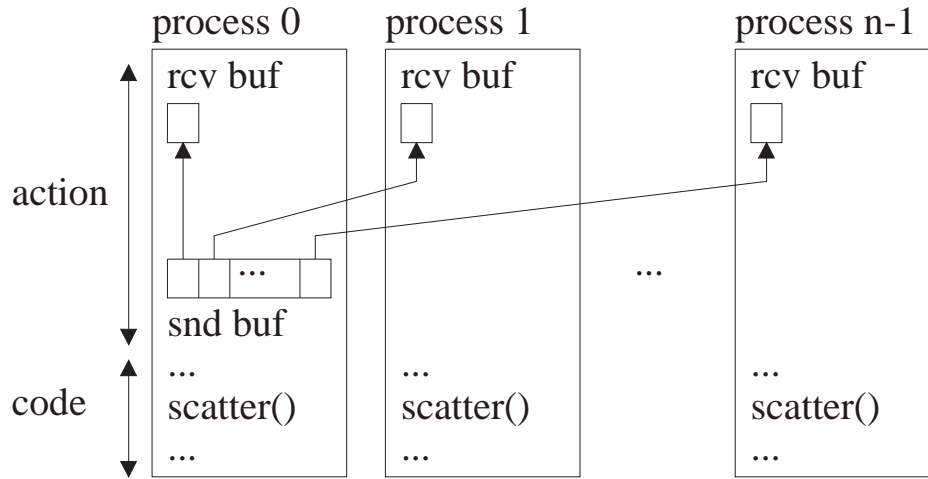
- MPI\_Bcast (buffer, count, datatype, root, communicator)



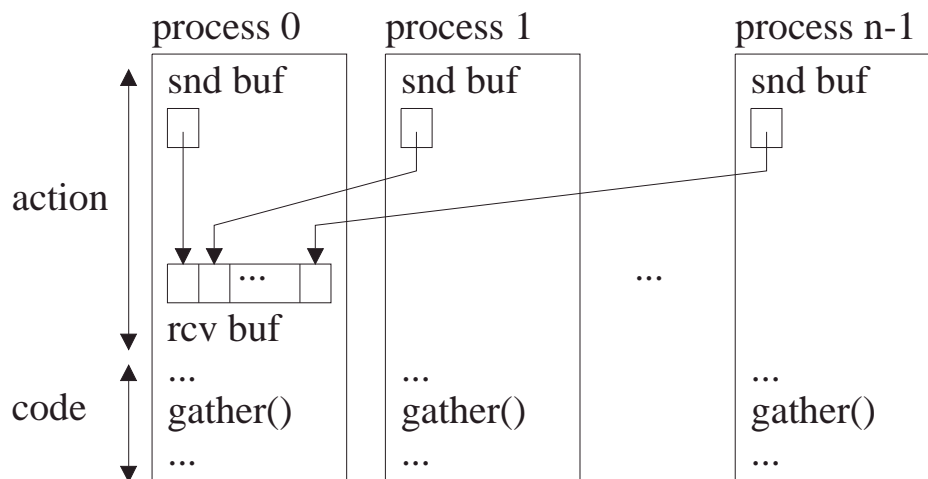
- MPI\_Reduce (sndbuf, rcvbuf, count, datatype, operation, root, communicator)



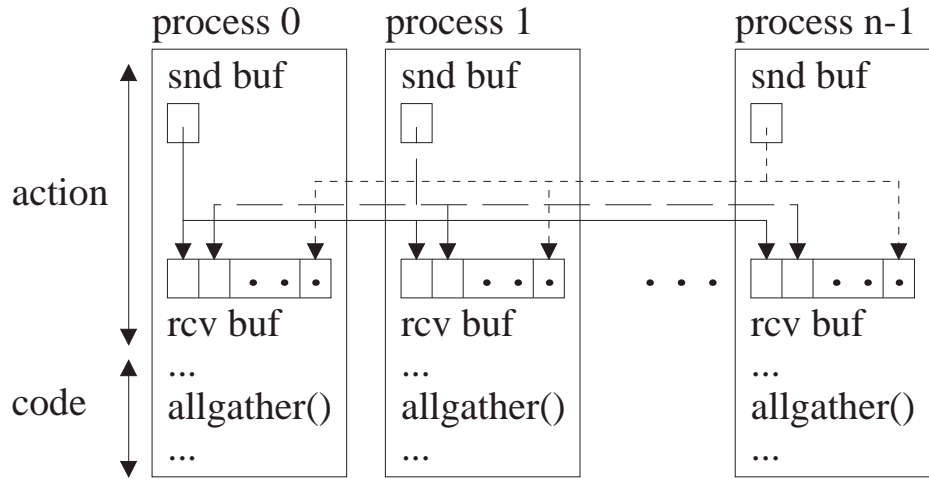
- MPI\_Scatter (sndbuf, sndcnt, sndtype, rcvbuf, rcvcnt, rcvtype, root, communicator)



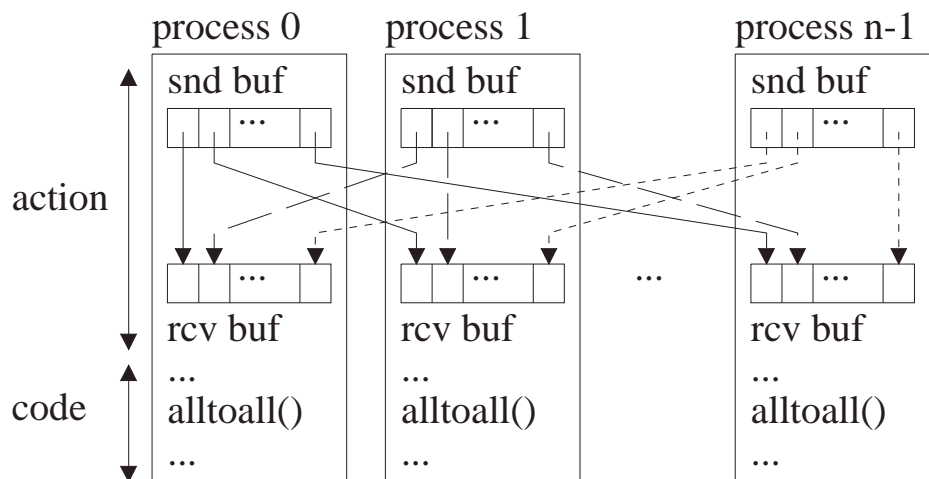
- MPI\_Gather (sndbuf, sndcnt, sndtype, rcvbuf, rcvcnt, rcvtype, root, communicator)



- MPI\_Allgather (sndbuf, sndcnt, sndtype, rcvbuf, rcvcnt, rcvtype, communicator)



- MPI\_Alltoall (sndbuf, sndcnt, sndtype, rcvbuf, rcvcnt, rcvtype, communicator)



## 5.3 Programmierschnittstelle

- ca. 125 Funktionen in MPI-1 und ca. 190 neue bzw. überarbeitete Funktionen in MPI-2, von denen im Allgemeinen mindestens die folgenden benötigt werden:
  - `MPI_Init`                    MPI initialisieren
  - `MPI_Comm_rank`            eigene Prozessnummer bestimmen
  - `MPI_Comm_size`            Anzahl der Prozesse bestimmen
  - `MPI_Send`                    Nachricht senden
  - `MPI_Recv`                    Nachricht empfangen
  - `MPI_Finalize`                MPI beenden
  
- `int MPI_Init (int *argc, char ***argv)`
  - *argc*                    Zeiger auf die Anzahl übergebener Argumente
  - *argv*                    Zeiger auf den Argumentvektor
  - Rückgabewert:    0    Operation erfolgreich beendet  
                  < 0   Fehler
  - initialisiert die MPI-Umgebung
  - der MPI-Standard spezifiziert keine Argumente auf der Kommandozeile, erlaubt den Implementierungen jedoch solche zu benutzen
  - die Handbuchseiten der verschiedenen Implementierungen geben an, ob Kommandozeilenparameter unterstützt werden
  - weitere Einzelheiten stehen in der Handbuchseite

- `int MPI_Comm_rank (MPI_Comm comm, int *rank)`
  - *comm*            Kommunikator
  - *rank*            Zeiger auf Variable für Prozessnummer innerhalb des Kommunikators
  - Rückgabewert:    0    Operation erfolgreich beendet  
                  < 0    Fehler
  - liefert die eigene Prozessnummer im Kommunikator
  - weitere Einzelheiten stehen in der Handbuchseite
  
- `int MPI_Comm_size (MPI_Comm comm, int *size)`
  - *comm*            Kommunikator
  - *size*            Zeiger auf Variable für die Anzahl der Prozesse innerhalb des Kommunikators
  - Rückgabewert:    0    Operation erfolgreich beendet  
                  < 0    Fehler
  - bestimmt die Anzahl der Prozesse in der Gruppe
  - weitere Einzelheiten stehen in der Handbuchseite

- `int MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - *buf* Adresse des ersten zu sendenden Datenelements
  - *count* Anzahl zu sendender Elemente
  - *datatype* Datentyp der zu sendenden Elemente
  - *dest* Prozessnummer des Empfängers
  - *tag* frei wählbares Kennzeichen ( $\geq 0$ ) zur Bezeichnung des Nachrichteninhalts
  - *comm* Kommunikator
  - Rückgabewert: 0 Operation erfolgreich beendet  
< 0 Fehler
  - sendet eine Nachricht im Standard-Modus
  - weitere Einzelheiten stehen in der Handbuchseite
  
- `int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - *buf* Adresse für das erste zu empfangende Element
  - *count* maximale Anzahl zu empfangender Elemente  
(die aktuelle Anzahl kann mit *MPI\_Get\_count* ermittelt werden)
  - *datatype* Datentyp der zu empfangenden Elemente
  - *source* Prozessnummer des Absenders der Nachricht, falls eine Nachricht eines bestimmten Absenders empfangen werden soll oder `MPI_ANY_SOURCE`, falls auf eine Nachricht eines beliebigen Absenders gewartet werden soll

- *tag* Kennzeichen für den Nachrichteninhalte ( $\geq 0$ ), falls eine Nachricht eines bestimmten Typs empfangen werden soll oder `MPI_ANY_TAG`, falls auf eine Nachricht beliebigen Inhalts gewartet werden soll
- *comm* Kommunikator
- *status* Zeiger auf Variable für Zustandsinformationen
- Rückgabewert: 0 Operation erfolgreich beendet  
< 0 Fehler
- empfängt eine Nachricht im Standard-Modus
- falls `MPI_ANY_SOURCE` oder `MPI_ANY_TAG` benutzt werden, können der Absender der Nachricht und der Nachrichtentyp über die Zustandsinformation ermittelt werden

```
typedef struct _status MPI_Status;
struct _status {
    int          MPI_SOURCE;    /* rank of source          */
    int          MPI_TAG;      /* message tag             */
    int          MPI_ERROR;    /* error code               */
    int          st_count;     /* top element count       */
    int          st_nelem;     /* low element count       */
    int          st_length;    /* message length          */
    MPI_Datatype st_dtype;     /* datatype                 */
};
```

- weitere Einzelheiten stehen in der Handbuchseite
- `int MPI_Finalize (void)`
  - Rückgabewert: 0 Operation erfolgreich beendet  
< 0 Fehler
  - beendet den Prozess und schließt die MPI-Umgebung
  - **jeder** Prozess muss diese Funktion am Ende ausführen
  - weitere Einzelheiten stehen in der Handbuchseite

- LAM-MPI-Version von "Hello World"

```

/* An MPI-version of the "hello world" program, which delivers some
 * information about its machine and operating system.
 *
 * Besides "libmpi" you have to bind "libnsl" and "libsocket" to the
 * program to get an executable using Sun's Solaris.
 *
 * Compiling:
 * mpicc -o hello_1_mpi hello_1_mpi.c [-lnsl -lsocket]
 *
 * Running:
 * LAM-MPI:
 * lamboot -v <boot schema file>
 * mpiexec -w [-s h] -np <number of processes> N hello_1_mpi
 * lamhalt
 * MPICH (e.g. for Cygwin):
 * mpiexec -np <number of processes> <program path>/hello_1_mpi
 *
 * File: hello_1_mpi.c Author: S. Gross
 * Date: 14.06.2007
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/utsname.h>
#include "mpi.h"

#define BUF_SIZE          255          /* message buffer size          */
#define MAX_TASKS        12          /* max. number of tasks          */
#define SENDTAG           1          /* send message command          */
#define EXITTAG           2          /* termination command          */
#define MSGTAG            3          /* normal message token          */

#define ENTASKS           -1          /* error: too many tasks          */

static void master (void);
static void slave (void);

int main (int argc, char *argv[])
{
    int mytid;                          /* my task id                      */

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &mytid);
    if (mytid == 0)
    {
        master ();
    }
    else
    {
        slave ();
    }
    MPI_Finalize ();
    return 0;
}

```

```

/* Function for the "master task". The master sends a request to all
 * slaves asking for a message. After receiving and printing the
 * messages he sends all slaves a termination command.
 *
 * input parameters:    not necessary
 * output parameters:  not available
 * return value:       nothing
 * side effects:       no side effects
 *
 */
void master (void)
{
    int          ntasks,          /* number of parallel tasks      */
               mytid,           /* my task id                    */
               num,             /* number of entries             */
               i;              /* loop variable                 */
    char         buf[BUF_SIZE + 1]; /* message buffer (+1 for '\0') */
    MPI_Status   stat;          /* message details               */

    MPI_Comm_rank (MPI_COMM_WORLD, &mytid);
    MPI_Comm_size (MPI_COMM_WORLD, &ntasks);
    if (ntasks > MAX_TASKS)
    {
        printf ("Error: Too many tasks. Try again with at most %d tasks.\n",
                MAX_TASKS);
        /* terminate all slave tasks */
        for (i = 1; i < ntasks; ++i)
        {
            MPI_Send ((char *) NULL, 0, MPI_CHAR, i, EXITTAG, MPI_COMM_WORLD);
        }
        MPI_Finalize ();
        exit (ENTASKS);
    }
    printf ("\n\nNow %d slave tasks are sending greetings.\n\n",
            ntasks - 1);
    /* request messages from slave tasks */
    for (i = 1; i < ntasks; ++i)
    {
        MPI_Send ((char *) NULL, 0, MPI_CHAR, i, SENDTAG, MPI_COMM_WORLD);
    }
    /* wait for messages and print greetings */
    for (i = 1; i < ntasks; ++i)
    {
        MPI_Recv (buf, BUF_SIZE, MPI_CHAR, MPI_ANY_SOURCE,
                 MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
        MPI_Get_count (&stat, MPI_CHAR, &num);
        buf[num] = '\0'; /* add missing end-of-string */
        printf ("Greetings from task %d:\n"
                " message type:\t%d\n"
                " msg length:\t%d characters\n"
                " message:\t%s\n\n",
                stat.MPI_SOURCE, stat.MPI_TAG, num, buf);
    }
    /* terminate all slave tasks */
    for (i = 1; i < ntasks; ++i)
    {
        MPI_Send ((char *) NULL, 0, MPI_CHAR, i, EXITTAG, MPI_COMM_WORLD);
    }
}

```

```

/* Function for "slave tasks". The slave task sends its hostname,
 * operating system name and release, and processor architecture
 * as a message to the master.
 *
 * input parameters:    not necessary
 * output parameters:  not available
 * return value:       nothing
 * side effects:       no side effects
 *
 */
void slave (void)
{
    struct utsname sys_info;          /* system information          */
    int mytid,                       /* my task id                  */
        more_to_do;
    char buf[BUF_SIZE];              /* message buffer              */
    MPI_Status stat;                 /* message details             */

    MPI_Comm_rank (MPI_COMM_WORLD, &mytid);
    more_to_do = 1;
    while (more_to_do == 1)
    {
        /* wait for a message from the master task          */
        MPI_Recv (buf, BUF_SIZE, MPI_CHAR, 0, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &stat);
        if (stat.MPI_TAG != EXITTAG)
        {
            uname (&sys_info);
            strcpy (buf, "hostname:\t\t");
            strncpy (buf + strlen (buf), sys_info.nodename,
                    BUF_SIZE - strlen (buf));
            strncpy (buf + strlen (buf), "\n\t\toperating system:\t",
                    BUF_SIZE - strlen (buf));
            strncpy (buf + strlen (buf), sys_info.sysname,
                    BUF_SIZE - strlen (buf));
            strncpy (buf + strlen (buf), "\n\t\trelease:\t\t",
                    BUF_SIZE - strlen (buf));
            strncpy (buf + strlen (buf), sys_info.release,
                    BUF_SIZE - strlen (buf));
            strncpy (buf + strlen (buf), "\n\t\tprocessor:\t\t",
                    BUF_SIZE - strlen (buf));
            strncpy (buf + strlen (buf), sys_info.machine,
                    BUF_SIZE - strlen (buf));
            MPI_Send (buf, strlen (buf), MPI_CHAR, stat.MPI_SOURCE,
                     MSGTAG, MPI_COMM_WORLD);
        }
        else
        {
            more_to_do = 0;          /* terminate          */
        }
    }
}

```

- der virtuelle Rechner kann über eine Konfigurationsdatei initialisiert werden (*boot schema file*)
- bei **einem** Programm kann dieses direkt mit *mpiexec* gestartet werden

- Realisierung von "Hello World" mit zwei Programmen (*Master*)  
(hello\_2\_mpi.c)

```

...
int main (int argc, char *argv[])
{
    int          ntasks,          /* number of parallel tasks */
               mytid,           /* my task id */
               num,             /* number of entries */
               i;               /* loop variable */
    char         buf[BUF_SIZE + 1]; /* message buffer (+1 for '\0') */
    MPI_Status   stat;           /* message details */

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &mytid);
    MPI_Comm_size (MPI_COMM_WORLD, &ntasks);
    if (ntasks > MAX_TASKS)
    {
        printf ("Error: Too many tasks. Try again with at most %d tasks.\n",
                MAX_TASKS);
        /* terminate all slave tasks */
        for (i = 1; i < ntasks; ++i)
        {
            MPI_Send ((char *) NULL, 0, MPI_CHAR, i, EXITTAG, MPI_COMM_WORLD);
        }
        MPI_Finalize ();
        exit (ENTASKS);
    }
    printf ("\n\nNow %d slave tasks are sending greetings.\n\n",
            ntasks - 1);
    /* request messages from slave tasks */
    for (i = 1; i < ntasks; ++i)
    {
        MPI_Send ((char *) NULL, 0, MPI_CHAR, i, SENDTAG, MPI_COMM_WORLD);
    }
    /* wait for messages and print greetings */
    for (i = 1; i < ntasks; ++i)
    {
        MPI_Recv (buf, BUF_SIZE, MPI_CHAR, MPI_ANY_SOURCE,
                 MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
        MPI_Get_count (&stat, MPI_CHAR, &num);
        buf[num] = '\0'; /* add missing end-of-string */
        printf ("Greetings from task %d:\n"
                " message type:\t%d\n"
                " msg length:\t%d characters\n"
                " message:\t%s\n\n",
                stat.MPI_SOURCE, stat.MPI_TAG, num, buf);
    }
    /* terminate all slave tasks */
    for (i = 1; i < ntasks; ++i)
    {
        MPI_Send ((char *) NULL, 0, MPI_CHAR, i, EXITTAG, MPI_COMM_WORLD);
    }
    MPI_Finalize ();
    return 0;
}

```

- Realisierung von "Hello World" mit zwei Programmen (*Slave*)

(hello\_2\_slave\_mpi.c)

```

...
int main (int argc, char *argv[])
{
    struct utsname sys_info;          /* system information          */
    int             mytid,            /* my task id                  */
            more_to_do;

    char            buf[BUF_SIZE];    /* message buffer              */
    MPI_Status      stat;             /* message details             */

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &mytid);
    srand ((unsigned int) time (NULL) * mytid * mytid);
    more_to_do = 1;
    while (more_to_do == 1)
    {
        /* wait for a message from the master task          */
        MPI_Recv (buf, BUF_SIZE, MPI_CHAR, 0, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &stat);
        if (stat.MPI_TAG != EXITTAG)
        {
            uname (&sys_info);
            strcpy (buf, "hostname:\t\t");
            strncpy (buf + strlen (buf), sys_info.nodename,
                    BUF_SIZE - strlen (buf));
            strncpy (buf + strlen (buf), "\n\t\toperating system:\t",
                    BUF_SIZE - strlen (buf));
            strncpy (buf + strlen (buf), sys_info.sysname,
                    BUF_SIZE - strlen (buf));
            strncpy (buf + strlen (buf), "\n\t\trelease:\t\t",
                    BUF_SIZE - strlen (buf));
            strncpy (buf + strlen (buf), sys_info.release,
                    BUF_SIZE - strlen (buf));
            strncpy (buf + strlen (buf), "\n\t\tprocessor:\t\t",
                    BUF_SIZE - strlen (buf));
            strncpy (buf + strlen (buf), sys_info.machine,
                    BUF_SIZE - strlen (buf));
            sleep (rand () % MAXWTIME);
            MPI_Send (buf, strlen (buf), MPI_CHAR, stat.MPI_SOURCE,
                     MSGTAG, MPI_COMM_WORLD);
        }
        else
        {
            more_to_do = 0;          /* terminate          */
        }
    }
    MPI_Finalize ();
    return 0;
}

```

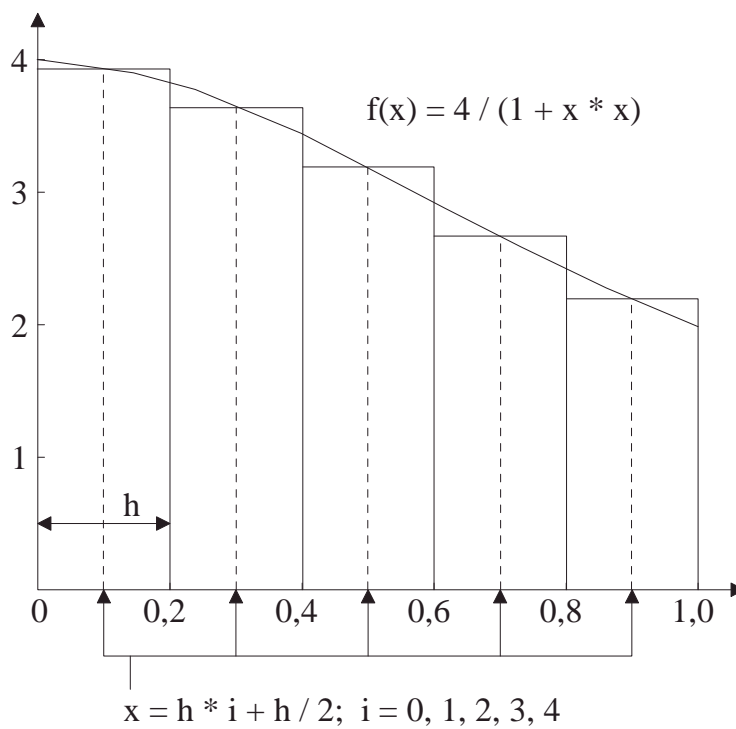
- bei mehreren Programmen müssen die einzelnen Programme auf die Rechner des virtuellen Rechners über eine Konfigurationsdatei verteilt werden (*application schema file*)

- ein MPI-Programm zur numerischen Integration  
(Lösung mit Gruppenkommunikationsoperationen)

**Aufgabe:** Berechnung von  $\pi$  durch Integration

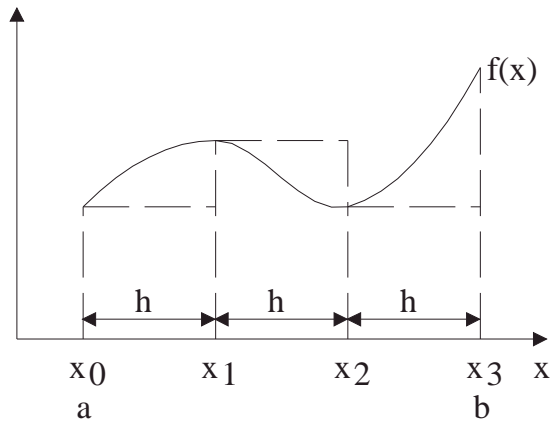
$$\int_0^1 \frac{1}{1+x^2} = \arctan(x) \Big|_0^1 = \arctan(1) - \arctan(0) = \frac{\pi}{4}$$

## Berechnung nach dem Tangenten-Trapez-Verfahren

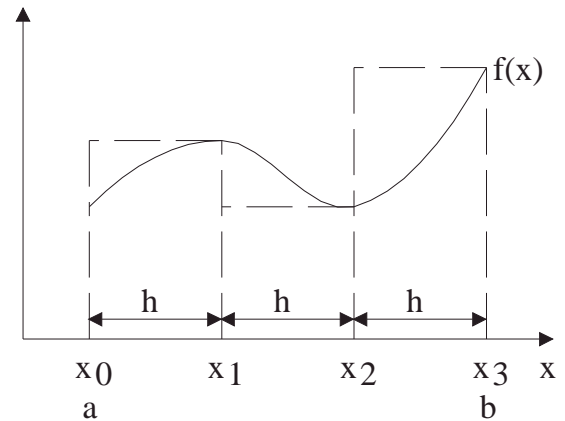


- Numerische Integration (kurze Wiederholung)  
(Newton-Côtes Formeln für Polynome vom Grad 0, 1 und 2)

### 0) Rechteckregeln

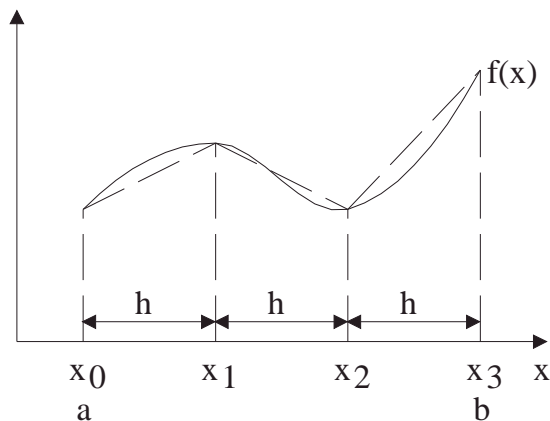


$$I = h \cdot f(x_0) + h \cdot f(x_1) + \dots + h \cdot f(x_{n-1})$$



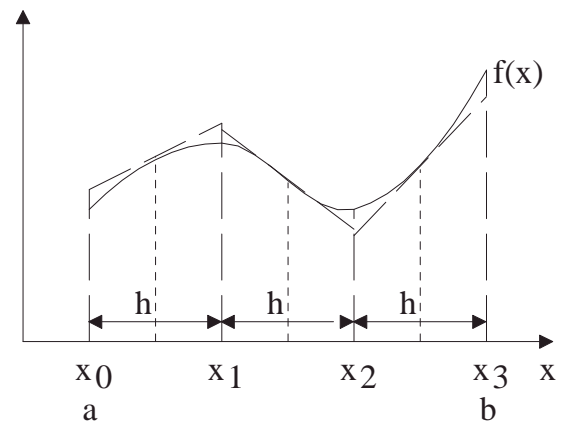
$$I = h \cdot f(x_1) + h \cdot f(x_2) + \dots + h \cdot f(x_n)$$

### 1) Trapezregeln



#### Sehnen-Trapez-Verfahren

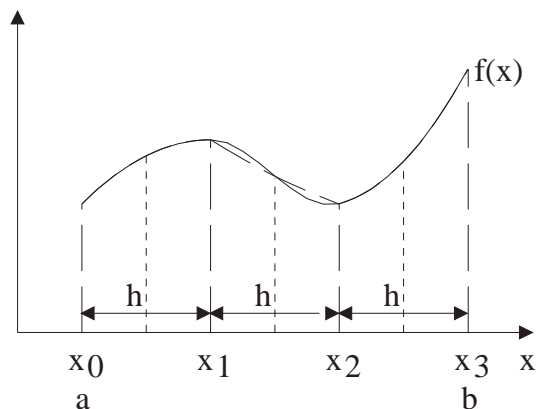
$$I = \frac{h}{2} (f(x_0) + f(x_0 + h)) + \dots + \frac{h}{2} (f(x_{n-1}) + f(x_{n-1} + h))$$



#### Tangenten-Trapez-Verfahren

$$I = h \cdot f(x_0 + \frac{h}{2}) + \dots + h \cdot f(x_{n-1} + \frac{h}{2})$$

## 2) Simpson-Regel



$$I = \frac{h}{6}(f(x_0) + 4f(x_0 + \frac{h}{2}) + f(x_0 + h)) + \dots + \frac{h}{6}(f(x_{n-1}) + 4f(x_{n-1} + \frac{h}{2}) + f(x_{n-1} + h))$$

In den Intervallen  $[x_0, x_1]$  und  $[x_2, x_3]$  ist bei der Simpson-Regel zwischen der Funktion  $f(x)$  und der Approximation bereits kein Unterschied erkennbar.

- das Programm (pi\_mpi.c)

```

...
#define f(x)      (4.0 / (1.0 + (x) * (x)))
#define PI_25    3.141592653589793238462643      /* 25 digits */

int main (int argc, char *argv[])
{
    int          alg,                /* algorithm */
           ntasks,                  /* number of parallel tasks */
           mytid,                   /* my task id */
           n,                        /* number of subintervals */
           err_cnt,                  /* input error counter */
           more_to_do,
           i;                        /* loop variable */
    double       mypi,               /* local sum of pi */
           pi,                       /* global sum of pi */
           h,                         /* length of subinterval */
           x,                          /* distinct points xi */
           t1, t2, t3, t4;           /* temporary values */
    double       scomm_time,         /* starttime for communication */
           scomp_time,              /* starttime for computation */
           dcomm_time,              /* duration of communication */
           dcomp_time, dcomp_tmp;   /* duration of computation */

    MPI_Init (&argc, &argv);
    scomm_time = MPI_Wtime ();
    MPI_Comm_rank (MPI_COMM_WORLD, &mytid);
    MPI_Comm_size (MPI_COMM_WORLD, &ntasks);
    dcomm_time = MPI_Wtime () - scomm_time;
    if (mytid == 0)
    {
        printf ("...", MPI_Wtick ());
        ...
    }
}

```

```

more_to_do = 1;
while (more_to_do == 1)
{
    if (mytid == 0)
    {
        printf ("...");
        ...
    }
    /* Each task m u s t call MPI_Bcast to learn about the algorithm
    * which should be used.
    */
    MPI_Bcast (&alg, 1, MPI_INT, 0, MPI_COMM_WORLD);
    /* Now we can start our real work!
    */
    scomm_time = MPI_Wtime ();          /* 1st part of communication time */
    MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    dcomm_time = MPI_Wtime () - scomm_time;
    if (n <= 0)
    {
        more_to_do = 0;                /* terminate */
    }
    else
    {
        scomp_time = MPI_Wtime ();
        h = 1.0 / (double) n;
        mypi = 0.0;
        switch (alg)
        {
            case 1:                    /* tangent-trapezoidal rule */
                t1 = h / 2;
                for (i = mytid; i < n; i += ntasks)
                {
                    x = h * (double) i;
                    mypi += h * f(x + t1);
                }
                break;

            case 2:                    /* chord-trapezoidal rule */
                ...
        }
        dcomp_tmp = MPI_Wtime () - scomp_time;
        /* Because the following MPI_Reduce call is for statistical
        * reasons only, it will not be integrated in the communication
        * time.
        */
        MPI_Reduce (&dcomp_tmp, &dcomp_time, 1, MPI_DOUBLE,
                    MPI_SUM, 0, MPI_COMM_WORLD);
        scomm_time = MPI_Wtime ();      /* 2nd part of communication time */
        MPI_Reduce (&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                    MPI_COMM_WORLD);
        dcomm_time += MPI_Wtime () - scomm_time;
        if (mytid == 0)
        {
            printf ("\nApproximation for Pi using %d intervals: %.16f\n"
                    "Error: %.1e\n",
                    n, pi, fabs ((double) pi - PI_25));
            ...
        }
    }
}
MPI_Finalize ();
return 0;
}

```

- parallele Matrizenmultiplikation bei gemeinsamem Speicher
  - sequentielle Lösung der Matrizenmultiplikation:  $C = A B$

```
...
double a[n][n], b[n][n], c[n][n];
...
for (i = 0; i < n; ++i)
{
    for (j = 0; j < n; ++j)
    {
        c[i][j] = 0.0;
        for (k = 0; k < n; ++k)
        {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}
...
```

- ◆ die Berechnungen der inneren Produkte sind unabhängig voneinander
  - ◆ alle inneren Produkte können parallel berechnet werden
  - ◆ alle Zeilen können parallel berechnet werden
  - ◆ alle Spalten können parallel berechnet werden
  - ◆ Blöcke von Zeilen und Spalten (Submatrizen) können parallel berechnet werden
- ⇒ Matrizenmultiplikation ist eine extrem parallele Anwendung  
(vergleichbar mit der Berechnung von Mandelbrot Mengen)

- parallele Berechnung der Zeilen der Ergebnismatrix

```

parfor (i = 0; i < n; ++i)
{
    for (j = 0; j < n; ++j)
    {
        c[i][j] = 0.0;
        for (k = 0; k < n; ++k)
        {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}

```

- ◆ bei  $n$  Prozessen berechnet jeder genau eine Zeile
- ◆ ein Prozess berechnet einen Zeilenblock, wenn weniger als  $n$  Prozesse zur Verfügung stehen

- parallele Berechnung der Spalten der Ergebnismatrix (Lösung 1)

```

for (i = 0; i < n; ++i)
{
    parfor (j = 0; j < n; ++j)
    {
        c[i][j] = 0.0;
        for (k = 0; k < n; ++k)
        {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}

```

⇒ schlechte Lösung, da für jede Zeile  $n$  Prozesse erzeugt werden, die jeweils ein Spaltenelement parallel berechnen

⇒ die beiden äußeren Schleifen müssen vertauscht werden

(die Reihenfolge von zwei Schleifen darf vertauscht werden, wenn die Schleifenkörper unabhängig voneinander sind und immer noch dasselbe Ergebnis liefern)

- parallele Berechnung der Spalten der Ergebnismatrix (Lösung 2)

```
parfor (j = 0; j < n; ++j)
{
    for (i = 0; i < n; ++i)
    {
        c[i][j] = 0.0;
        for (k = 0; k < n; ++k)
        {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}
```

- ◆ bei  $n$  Prozessen berechnet jeder genau eine Spalte
- ◆ ein Prozess berechnet einen Spaltenblock, wenn weniger als  $n$  Prozesse zur Verfügung stehen

- parallele Berechnung aller inneren Produkte

```
parfor (i = 0; i < n; ++i)
{
    parfor (j = 0; j < n; ++j)
    {
        c[i][j] = 0.0;
        for (k = 0; k < n; ++k)
        {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}
```

- ◆ zuerst wird ein Prozess für jede Zeile erzeugt
  - ◆ danach erzeugt jeder "Zeilen-Prozess" einen Prozess für jedes Spaltenelement
- ⇒  $n^2$  Prozesse werden erzeugt

- kann die innerste Schleife über  $k$  ebenfalls parallelisiert werden?
  - ⇒ in diesem Schleifenkörper wird das Matrixelement  $c[i][j]$  **sowohl gelesen als auch geschrieben**
  - ⇒ die Operationen sind voneinander abhängig
  - ⇒ nein
- parallele Matrizenmultiplikation bei verteiltem Speicher mit MPI
  - Annahme:  $\mathbf{A}$ ,  $\mathbf{B}$  und  $\mathbf{C}$  seien  $n \times n$  Matrizen und es gäbe genau  $n$  Arbeitsprozesse für die parallele Multiplikation
  - es werden ein Koordinierungsprozess und  $n$  weitere Prozesse erzeugt, die jeweils eine Zeile der Ergebnismatrix berechnen
  - Struktur des Koordinierungsprozesses

```
process coordinator
{
    double a[n][n],          /* source matrix a          */
           b[n][n],          /* source matrix b          */
           c[n][n];         /* result matrix c         */

    initialize matrices a and b;
    for (i = 0; i < n; ++i)
    {
        send row  $i$  of  $a$  to worker  $i$ ;
        send all of  $b$  to worker  $i$ ;
    }

    for (i = 0; i < n; ++i)
    {
        receive row  $i$  of  $c$  from worker  $i$ ;
    }
    print the result matrix  $c$ ;
}
```

## – Struktur eines Arbeitsprozesses

```

process worker
{
    double row_a[n],          /* row i of matrix a      */
          b[n][n],           /* all of matrix b       */
          row_c[n];          /* row i of matrix c     */

    receive initial values for vector a and matrix b;
    for (j = 0; j < n; ++j)
    {
        row_c[j] = 0.0;
        for (k = 0; k < n; ++k)
        {
            row_c[j] = row_c[j] + row_a[k] * b[k][j];
        }
    }
    send result vector c to coordinator process
}

```

## – MPI-Programm für eine parallele Matrizenmultiplikation

```

/* Matrix multiplication: c = a * b.
 * This program needs as many processes as there are rows in
 * matrix "a".
 * ...
 */

#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define P          4          /* # of rows              */
#define Q          6          /* # of columns/rows      */
#define R          8          /* # of columns            */

static void print_matrix (int p, int q, double **mat);

int main (int argc, char *argv[])
{
    int          ntasks,      /* # of parallel tasks    */
              mytid,         /* my task id             */
              i, j, k,       /* loop variables         */
              tmp;           /* temporary value        */
    double      a[P][Q], b[Q][R], /* matrices to multiply   */
              c[P][R],      /* matrix for result      */
              row_a[Q],     /* one row of matrix "a"  */
              row_c[R];     /* one row of matrix "c"  */

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &mytid);
    MPI_Comm_size (MPI_COMM_WORLD, &ntasks);

```

```

if ((ntasks != P) && (mytid == 0))
{
    printf ("\n\nWe need %d processes.\n"
           "Usage: mpiexec -w -np %d N %s\n\n\n",
           P, P, argv[0]);
}
if (ntasks != P)
{
    MPI_Finalize ();
    exit (EXIT_FAILURE);
}
if (mytid == 0)
{
    tmp = 1;
    for (i = 0; i < P; ++i)          /* initialize matrix a          */
    {
        for (j = 0; j < Q; ++j)
        {
            a[i][j] = tmp++;
        }
    }
    printf ("\n\n(%d,%d)-matrix a:\n\n", P, Q);
    print_matrix (P, Q, (double **) a);
    tmp = Q * R;
    for (i = 0; i < Q; ++i)          /* initialize matrix b          */
    {
        for (j = 0; j < R; ++j)
        {
            b[i][j] = tmp--;
        }
    }
    printf ("(%d,%d)-matrix b:\n\n", Q, R);
    print_matrix (Q, R, (double **) b);
}
/* send matrix "b" to all processes          */
MPI_Bcast (b, Q * R, MPI_DOUBLE, 0, MPI_COMM_WORLD);
/* send row i of "a" to process i          */
MPI_Scatter (a, Q, MPI_DOUBLE, row_a, Q, MPI_DOUBLE, 0,
             MPI_COMM_WORLD);
for (j = 0; j < R; ++j)          /* compute i-th row of "c"          */
{
    row_c[j] = 0.0;
    for (k = 0; k < Q; ++k)
    {
        row_c[j] = row_c[j] + row_a[k] * b[k][j];
    }
}
/* receive row i of "c" from process i          */
MPI_Gather (row_c, R, MPI_DOUBLE, c, R, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
if (mytid == 0)
{
    printf ("(%d,%d)-result-matrix c = a * b :\n\n", P, R);
    print_matrix (P, R, (double **) c);
}
MPI_Finalize ();
return 0;
}
...

```

Programm kann optimiert werden, da nur der Koordinierungsprozess (*task 0*) Speicher für **A** und **C** benötigt

– **Aufgabe 5 - 2:**

Erweitern Sie das obige Programm, so dass eine beliebige Anzahl Prozesse (zwischen  $1$  und  $n$ ) für die Berechnung benutzt werden kann (sehen Sie sich die Handbuchseiten für *MPI\_Scatterv* und *MPI\_Gatherv* an).

– Struktur des Arbeitsprozesses, wenn er anstelle der Gesamtmatrix **B** zu jedem Zeitpunkt nur eine Spalte der Matrix speichern soll

- ◆ der Koordinierungsprozess sendet Zeile  $i$  von Matrix **A** und Spalte  $i$  von Matrix **B** an Arbeitsprozess  $i$
- ◆ jeder Arbeitsprozess kann das Ergebnis für  $c[i][i]$  berechnen
- ◆ jeder Arbeitsprozess  $i$  benötigt alle anderen Spalten von **B**, um alle Elemente der Zeile  $i$  der Matrix **C** berechnen zu können
  - ⇒ Arbeitsprozesse müssen ringförmig angeordnet werden
  - ⇒ jeder Arbeitsprozess sendet seine Spalte von **B** an seinen Nachfolger
  - ⇒ jeder Arbeitsprozess erhält seine nächste Spalte von seinem Vorgänger
- ◆ die Berechnung ist nach  $n-1$  Schritten vollständig
- ◆ die Berechnung dauert länger als beim ersten Programm mit vollständiger Kopie von **B**, weil jetzt eine Synchronisation der Arbeitsprozesse erforderlich ist

```
process worker
{
    double row_a[n],          /* row i of matrix a      */
           col_b[n],          /* one column of matrix b */
           row_c[n],          /* row i of matrix c      */
           sum;               /* sum of inner product    */
    int     next_col;         /* next column of result   */

    next_col = i;            /* start with c[i][i]     */
    sum       = 0.0;
    receive row i of matrix a and column i of matrix b;
    /* compute c[i][i]      */
    for (k = 0; k < n; ++k)
    {
        sum = sum + row_a[k] * col_b[k];
    }
    c[next_col] = sum;
    /* circulate columns and compute rest of row i of c */
    for (j = 1; j < n; ++j)
    {
        send own column of b to successor;
        receive new column of b from predecessor;
        sum = 0.0;
        for (k = 0; k < n; ++k)
        {
            sum = sum + row_a[k] * col_b[k];
        }
        next_col = (next_col - 1 + n) % n;
        c[next_col] = sum;
    }
    send result vector c to coordinator process
}
```

### – Aufgabe 5 - 3:

Implementieren Sie die obige Lösung und vergleichen Sie die Laufzeit mit der Laufzeit des Programms aus Aufgabe 5-2.

## 5.4 Laufzeitumgebung

### 5.4.1 Umgebungsvariablen

- allgemeine Umgebungsvariablen

PATH	enthält die Verzeichnisse, in denen nach ausführbaren Programmen gesucht wird
MANPATH	das Verzeichnis mit den Handbuchseiten von MPI kann ergänzt werden
LD_LIBRARY_PATH	das Verzeichnis der MPI-Bibliothek kann hinzugefügt werden
XAPPLRESDIR	das Verzeichnis mit der Ressourcen-Datei <i>XMPI</i> kann ergänzt werden, damit <i>xmpi</i> die korrekten Einstellungen benutzt

- MPI-spezifische Umgebungsvariablen

LAMHOME	Installationsverzeichnis von MPI
LAMHCC	standardmäßig benutzt <i>mpicc</i> den Compiler, mit dem MPI übersetzt wurde (GNU C-Compiler); über diese Variable kann ein anderer Compiler gewählt werden (führt im Allgemeinen zu Fehlern, da <i>mpicc</i> an den Compiler <i>Flags</i> übergibt, die compiler-spezifisch sind)

LAMHCP	analog kann für <i>mpiCC</i> ein anderer C++-Compiler als <i>g++</i> gewählt werden
LAMHF77	analog kann für <i>mpif77</i> ein anderer Fortran-Compiler als <i>g77</i> gewählt werden
LAMBHOST	legt ggf. ein <i>boot schema file</i> fest, so dass es nicht auf der Kommandozeile angegeben werden muss
LAMRSH	hiermit kann die zu benutzende <i>Remote Shell</i> festgelegt werden ( <i>rsh</i> , <i>ssh</i> , ...)
LAMAPPLDIR	hiermit kann ein Verzeichnis für ein <i>application schema file</i> festgelegt werden

## 5.4.2 Konfigurationsdateien

- Standarddateien
  - `$LAMHOME/boot/bhost.def` *default boot schema file*
  - `$LAMHOME/boot/conf.lam` *default process schema file for lamboot*
  - `$LAMHOME/boot/conf.otb` *default process schema file for hboot*
  - `$LAMHOME/app-defaults/XMPI` *default resource file for xmpi*

- Konfigurationsdatei für den virtuellen Rechner (*boot schema file*)
  - notwendig, um einen virtuellen Rechner zu starten

```
# Boot schema file for a virtual computer. This file is useful, if
# you want to create a virtual computer with more than one computer.
# Use "lamboot" and "lamhalt" without parameters when you implement
# and test your program (all processes are created on your local
# computer). Use this file to start a homogeneous or heterogeneous
# multicompiler virtual computer (don't forget to create an
executable
# on every architecture before running "mpiexec".
#
#
# Usage: recon -v hosts.lam-mpi
#         lamboot -v host.lam-mpi
#         lamhalt (alternatively "wipe -v hosts.lam-mpi")
#
# File: hosts.lam-mpi                      Author: S. Gross
# Date: 18.10.2004
#

# The following n computers will be numbered from 0 to n-1. They can
# be selected in the "application schema file" individually with n0,
# n1, and so on or else as a group with e.g. n2-7. The CPU's of the
# computers will be numbered consecutively starting with c0. E.g., if
# you specify "cpu=4" for the first computer than c0-3 belong to n0.
# "cpu=n" specifies the number of "virtual" cpu's which are relevant
# for the number of processes started on a node if you use option "C"
# with "mpiexec" (it's allowed to have more virtual cpu's than
# physical cpu's). The local computer must be the first member of the
# following list!

#
# U n c o m m e n t   at least your local computer and change
# "user=..." to your user name.
#

# SunOS, sparc
#rs0.informatik.hs-fulda.de      cpu=1    user=fd1026
#rs1.informatik.hs-fulda.de      cpu=1    user=fd1026
...
```

- allgemeiner Aufbau der Konfigurationsdatei (siehe auch: *man bhost*):

```
# Kommentar
<Rechner>[cpu=<Anzahl CPU's>][user=<Benutzername>]
```

- *Default*: cpu = 1, user = <Login-Name>
- die Konfigurationsdatei kann auf der Kommandozeile oder in der Umgebungsvariablen LAMBHOST angegeben werden; andernfalls wird die Standarddatei *bhost.def* benutzt

- folgende Verzeichnisse werden in dieser Reihenfolge nach der Konfigurationsdatei durchsucht
  1. das lokale Verzeichnis
  2. \$LAMHOME/boot
- Prozessverteilungsdatei für den virtuellen Rechner (*application schema file*)
  - diese Datei ist notwendig, wenn verschiedene Prozesse auf dem virtuellen Rechner gestartet werden sollen oder wenn der Prozess mit der Nummer 0 auf einem speziellen Rechner laufen soll

```
# Application schema file for "hello_2_mpi, hello_2_slave_mpi".
#
# Usage: mpiexec -w -v app_hello_2.lam-mpi
#
# File: app_hello_2.lam-mpi          Author: S. Gross
# Date: 22.10.2004
#

# The "master" (task 0) m u s t be started on the local computer!
# Otherwise you run in trouble with "printf" and "scanf" (infinite
# loops or no outputs). Don't use this file on a remote computer
# after "rlogin", "telnet", or something else or be sure that your
# local computer is listed f i r s t in the boot schema file
# ("recon -v <boot schema file>" must name your local computer as
# "n0").

h hello_2_mpi

# Distribute four "slaves" on all available computers of the
# virtual computer.

N -np 4 hello_2_slave_mpi

# If the virtual computer consists of homogeneous computers only, you
# can automatically transfer the program from your local computer.
# Advantage: If you don't use NFS, you haven't to compile the program
#             on each computer.
#             If you use NFS, you haven't to rely upon a correct PATH
#             on each computer.
# Disadvantage: The transfer consumes time and network capacity.

#N -np 8 -s h hello_2_slave_mpi
```

- allgemeiner Aufbau der Prozessverteilungsdatei (LAM-MPI)  
(siehe auch: *man appschema*):

```
# Kommentar
[<where>][-np #][-s <node>][-wd <dir>][-x <env>] <prog>[<args>]
```

<where>	h	lokaler Knoten, auf dem das Kommando eingegeben wird ( <i>here</i> )
	o	Knoten, auf dem LAM mit <i>lamboot</i> gestartet wurde ( <i>origin</i> )
	N	alle Knoten
	C	alle CPU's
	n<Liste>	Liste von Knoten, z. B. n0; n4-7; n0,3,5,7-9
	c<Liste>	analog <i>n&lt;Liste&gt;</i> für CPU's
-np #		Anzahl zu erzeugender Prozesse
-s <node>		Im Allgemeinen wird das Programm auf dem Knoten gesucht, auf dem es ausgeführt wird. Falls kein gemeinsames Dateisystem (NFS) benutzt wird und die Zielrechner homogen sind, kann LAM das Programm von einem <i>source node</i> auf die anderen Rechner übertragen.
-wd <dir>		Bevor das Programm ausgeführt wird, wird in das Verzeichnis <dir> gewechselt. Diese Angabe hat Priorität gegenüber der entsprechenden Kommandozeilenoption von <i>mpiexec</i> .
-x <env>		exportiert Umgebungsvariable <env> an alle Knoten, bevor das Programm ausgeführt wird
<prog>		Name des auszuführenden Programms

<args>                    alle Angaben nach dem Programmnamen  
werden dem auszuführenden Programm als  
Argumente übergeben

Falls nur <where> angegeben ist, wird auf jedem Knoten/jeder CPU ein Prozess erzeugt. Falls nur "-np #" angegeben ist, wird die entsprechende Anzahl Prozesse auf alle Knoten/CPU's des virtuellen Rechners verteilt. Falls beide Angaben fehlen, wird ein Prozess auf dem lokalen Knoten erzeugt. Falls beide Angaben vorhanden sind, wird die entsprechende Anzahl Prozesse auf die angegebenen Knoten/CPU's verteilt.

– folgende Verzeichnisse werden in dieser Reihenfolge nach der Prozessverteilungsdatei durchsucht

1. Verzeichnis der Umgebungsvariablen LAMAPPLDIR
2. das lokale Verzeichnis
3. \$LAMHOME/boot

- Prozesse des virtuellen Rechners

(*process schema file*; siehe auch: *man procschema*)

– die Konfigurationsdatei enthält die Prozesse, die die LAM-Umgebung bilden

– wird im Wesentlichen nur von Systementwicklern zum Testen benutzt oder zur Einrichtung einer maßgeschneiderten Umgebung

– *conf.lam* ist die Standarddatei für *lamboot* und *conf.otb* die für *hboot*

### 5.4.3 Initialisierung des virtuellen Rechners

- Programme zur Verwaltung des virtuellen Rechners

(Überblick über alle Kommandos: *man lam*; Beschreibung der Kommandos: *man <Kommando>* oder *<Kommando> --help*)

- *lamboot* startet den virtuellen Rechner
- *lamgrow* fügt weitere Rechner zum virtuellen Rechner hinzu
- *lamshrink* entfernt Rechner aus dem virtuellen Rechner
- *recon* überprüft, ob *LAM* gestartet werden kann
- *lamnodes* Rechner/CPU's des virtuellen Rechners anzeigen
- *laminfo* gibt Informationen zur Konfiguration aus
- *lamhalt* beendet den virtuellen Rechner
- *(lam)wipe* beendet den virtuellen Rechner

*lamboot* startet LAM auf allen Rechnern, die in der *Default*-Konfigurationsdatei genannt werden (im Allgemeinen nur der lokale Rechner). Ein Benutzer kann eine eigene Konfigurationsdatei verwenden, indem er *lamboot* die Datei als Parameter übergibt oder indem er die Datei der Umgebungsvariablen *LAMBHOST* zuweist.

- Programme zur Verwaltung der Prozesse des virtuellen Rechners

- *mpirun* ⇒ startet *n* Kopien eines Prozesses  
⇒ mehrere Prozesse werden über eine Prozessverteilungsdatei gestartet
- *mpiexec* startet MPI-Prozesse
- *lamclean* entfernt alle Prozesse und Nachrichten aus dem virtuellen Rechner (kein Neustart erforderlich!)

- Programme zur Überwachung der Prozesse des virtuellen Rechners

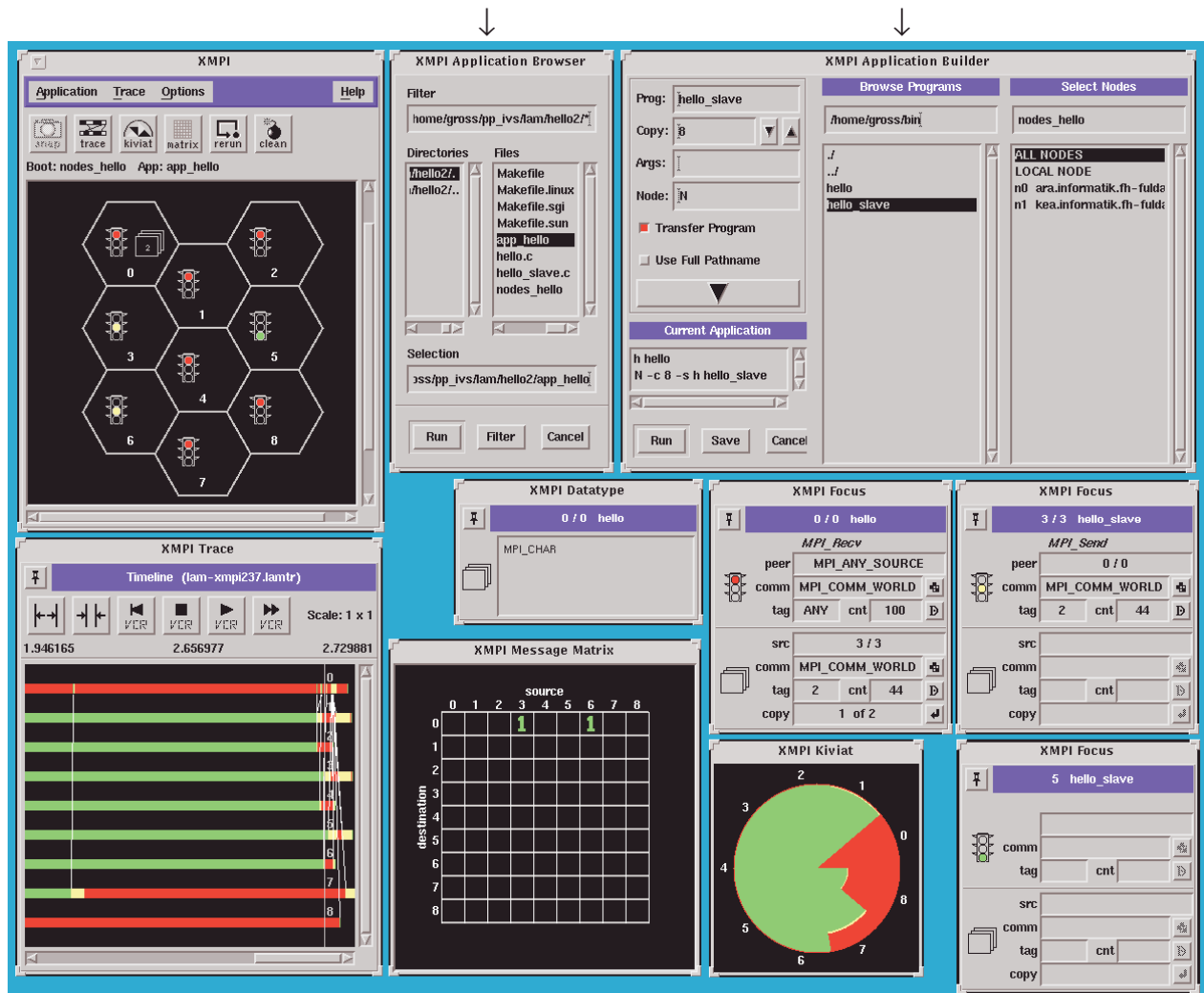
- *mpitask* liefert Angaben über die einzelnen Prozesse
- *mpimsg* liefert Angaben zu den Nachrichtenpuffern

- grafische Oberfläche *xmpi*
  - Aufruf: *xmpi*[<*boot schema file*>]
  - die Prozessverteilung kann mit *Browse&Run* eingelesen oder mit *Build&Run* interaktiv erstellt werden
  - im Überblicksbild wird durch eine Ampel angezeigt, ob der Prozess gerade aktiv oder blockiert ist
    - Ampel ist *rot*: Prozess ist in einer MPI-Routine blockiert
    - Ampel ist *gelb*: Prozess führt gerade eine MPI-Routine aus
    - Ampel ist *grün*: sonstige Aktivitäten
    - Ampel fehlt: Prozess hat MPI noch nicht initialisiert oder bereits verlassen
  - im Überblicksbild wird ggf. durch ein Rechteck mit einer Zahl angezeigt, wie viele Nachrichten im Nachrichtenpuffer sind
  - weitere Einzelheiten können der *Manual-Page* zu *xmpi* entnommen werden

## Application Menu

## Browse&amp;Run

## Build&amp;Run



(Im Trace-Diagramm können mit dem Zeitbalken bestimmte Zustände ausgewählt werden: linke Maustaste drücken, Zeitbalken verschieben, Maustaste loslassen  $\Rightarrow$  alle Diagramme zeigen die Werte für diesen Zustand an. Ein Klick mit der linken Maustaste verschiebt den Zeitbalken an die Mauszeigerposition. Einzelne Bereiche können ausgewählt und vergrößert dargestellt werden: rechte Maustaste drücken, Maus bewegen (es wird ein Rechteck aufgespannt), Maustaste loslassen. Ein Zoom kann auch über das Symbol  $\leftarrow \rightarrow$  in der Kopfzeile des Diagramms erreicht werden. Danach kann der gewünschte Bereich über den Rollbalken ausgewählt werden.)

## 5.5 Übersetzung und Ausführung von MPI-Programmen

- Übersetzung von MPI-Programmen
  - `mpicc` *Wrapper* für den C-Compiler
  - `mpiCC/mpic++` *Wrapper* für den C++-Compiler
  - `mpif77` *Wrapper* für den Fortran-Compiler
  
  - Beispiel: `mpicc -o pi_mpi pi_mpi.c`
- Programme müssen mit der Bibliothek *libmpi* gebunden werden  
(automatisch durch den *Wrapper*)
- Solaris benötigt zusätzlich die Bibliotheken *libnsl* (für XDR) und *libsocket*  
(automatisch durch den *Wrapper*)
- Verteilung und Start der Programme über *mpiexec* oder *xmpi*

**Aufgabe 5-4:**

Stabile Ehen: *Männer* und *Frauen* werden jeweils durch ein Feld von  $n$  Prozessen dargestellt. Jeder Mann bewertet jede Frau durch eine eindeutige Nummer zwischen  $1$  und  $n$  und jede Frau macht dasselbe für jeden Mann. Eine Paarung ist eine 1-zu-1-Beziehung zwischen Männern und Frauen. Eine Paarung ist stabil, wenn für zwei beliebige Männer  $m_1$  und  $m_2$  und deren Frauen  $w_1$  und  $w_2$  die beiden folgenden Bedingungen erfüllt sind:

1.  $m_1$  mag  $w_1$  lieber als  $w_2$  oder  $w_2$  mag  $m_2$  lieber als  $m_1$  und
2.  $m_2$  mag  $w_2$  lieber als  $w_1$  oder  $w_1$  mag  $m_1$  lieber als  $m_2$

Eine Lösung des Problems besteht aus  $n$  stabilen Paarungen. Implementieren Sie ein Programm, das das Problem der "Stabilen Ehen" löst. Die Männer fordern die Frauen zur Ehe auf. Eine Frau muss die erste Aufforderung annehmen, da sie nicht weiß, ob sie später ein besseres Angebot bekommt. Wenn Sie später von einem Mann zur Ehe aufgefordert wird, den sie lieber mag, kann sie das eingegangene Verhältnis lösen. Stellen Sie den Lösungsweg als Animation dar, so dass man alle Aktionen in "Echtzeit" verfolgen kann. Sorgen Sie dafür, dass die Prozesse langsam genug laufen, so dass man die Abläufe beobachten kann.