Programmiermodell Nachrichtenaustausch

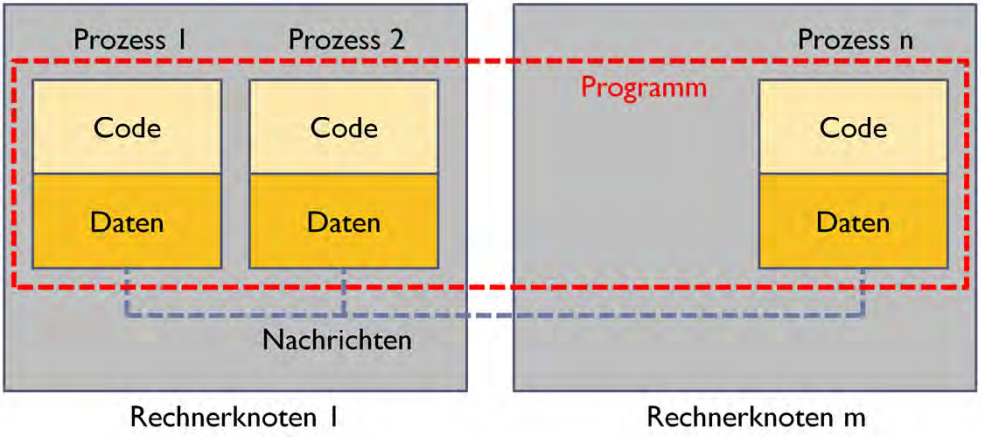
- Problemstellung
- Das Message Passing Interface (MPI)
- Ziele und Spezifikationsumfang
- Punkt-zu-Punkt-Kommunikation
- Abgeleitete Datentypen
- Kollektive Kommunikationen
- Gruppen, Kontexte, Prozesstopologien
- Bewertungen

Programmiermodell Nachrichtenaustausch
Die zehn wichtigsten Fragen

- Welche Kommunikationsschemata gibt es?
- Was sind die Ziele der MPI-Definition?
- Was enthält die MPI-Definition?
- Welche Variationen von Blockierungen gibt es bei den Funktionsaufrufen?
- Wie ist die Punkt-zu-Punkt-Kommunikation definiert?
- Wie funktioniert nichtblockierende Kommunikation?
- Was sind abgeleitete Datentypen?
- Was sind kollektive Kommunikationen?
- Was versteht man unter Prozesstopologien?
- Wie funktioniert das Profiling-Interface?

Problemstellungen

Die Codeteile des Programms in den Prozessen können identisch oder verschieden sein



Problemstellungen...

- Kompilieren für unterschiedliche Architekturen
- Laden des Codes auf unterschiedliche Knoten
- Start der Prozesse auf den Knoten
- Wechselseitiges Bekanntmachen der Prozesse
- Informationsaustausch zwischen Prozessen
- Optimierung der Kommunikationseffizienz
- Kommunikationsrelationen der Prozesse zueinander
- Überwachungsmöglichkeit der Abläufe

Laden und Starten des Codes

- Prinzipieller Aufruf (allgemeiner Fall)

  `spawn(<binary_name>,<node_list>,...);`

  - Ist ähnlich wie bei einer Thread-Erzeugung

- Wenn nur ein Programmcode existiert, dann z.B.

```
if (myid()==0)
then /* I´m the first */
  spawn(...); /* if others do not exist */
  send(init_data);
else /* I was spawned */
  receive(init_data);
fi
```

Nicht notwendigerweise nur ein Prozess pro Prozessor

Informationsaustausch

- Senden von Nachrichten

```
send(<to_proc_id>,<data>);
broadcast(<data>);
```

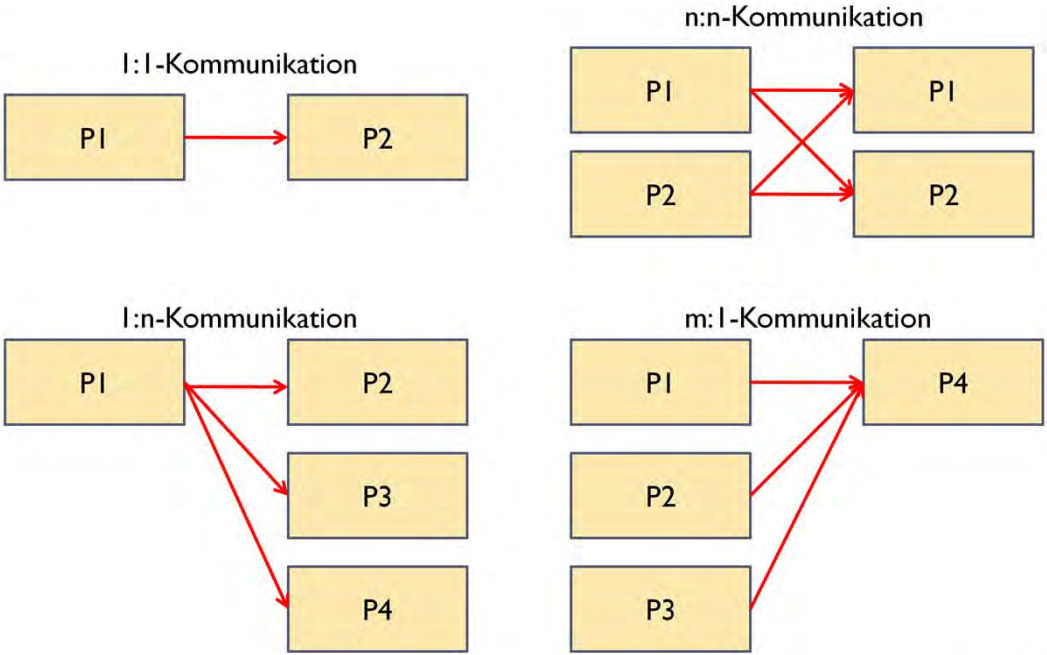
- Empfangen von Nachrichten

```
receive(<from_proc_id>,<data>);
testreceive(<from_proc_id>);
```

Charakteristisch: eigenhändiges Einfügen der Kommunikationsanweisungen in den Code

Hoher Aufwand aber auch hohe Leistungsausbeute

Kommunikationsschemata



Optimierung der Kommunikationseffizienz

Möglichst Senden und Empfangen nebenläufig abwickeln

Kombination aus Hardware und Software erforderlich

Existierende Ansätze Anfang der 90er

- **P4, Parmacs, Chameleon, NX, ...**

  Historie der Bibliotheken zum Nachrichtenaustausch
- **Parallel Virtual Machine (PVM)**

  Implementierung einer Bibliothek für nahezu alle Architekturen

  Lange Zeit de facto-Standard bei Clustern
- **Message Passing Interface (MPI)**

  Spezifikation einer Schnittstelle zum Nachrichtenaustausch

  De facto-Standard auf allen Hochleistungsrechnern und auf Cluster-Architekturen

Message Passing Interface (MPI)

- Vorangetrieben vom MPI-Forum (Firmen, Universitäten, ...)
- Beginn 1992

- MPI-Standard 1995 (nur Kommunikation)
- MPI-2-Standard 1997 (der nötige Rest)
- MPI-3.1-Standard 2015 (alles zusammen)
  - Dokument hat 868 Seiten

- Vorteile eines Standards: Portabilität, Einfachheit
  Vorher etwa ein Dutzend konkurrierende Ansätze
  - Alle funktional fast identisch aber syntaktisch unterschiedlich

- Probleme: Standardkonformität der Implementierungen

Ziele von MPI

- Entwurf einer Programmierschnittstelle (API)
- Unterstützung effizienter Kommunikationsmethoden
- Unterstützung heterogener Umgebungen
- Sprachanbindungen für Fortran77 und C/C++
  (jetzt auch für Java und Skript-Sprachen)
- Konstrukte nahe an bereits Existierendem
- Semantik der Schnittstelle soll sprachunabhängig sein
- Soll eine thread-sichere Implementierung gestatten
  - Wiedereintrittsfähige Routinen der Bibliotheksimplementierung!

Was MPI enthält

- Punkt-zu-Punkt-Kommunikation
- Kollektive Operationen
- Prozessgruppen
- Kommunikationskontexte
- Prozesstopologien
- Abfragefunktionen zur Programmumgebung
- Profiling-Schnittstelle

Was MPI (zunächst) nicht enthält

- Explizite Operationen für gemeinsamen Speicher
- Zusätzliche Unterstützung durch das Betriebssystem für z.B. unterbrechungsgesteuerte Kommunikation
- Explizite Unterstützung zur Prozessverwaltung
- Parallele Ein-/Ausgabe

MPI zunächst nur Nachrichtenaustausch

MPI-2 geht die obigen Punkte an

MPI-3 fasst alles in einem Standard zusammen

MPI-Spezifikationsmethode

- Aufrufe sprachunabhängig definiert
- Argumente mit IN, OUT oder INOUT annotiert

Z.B.
```
MPI_WAIT(request,status)
                INOUT request
                OUT   status
```
C:
```
int MPI_Wait(MPI_Request *request,
                    MPI_Status *status)
```
F77:
```
MPI_WAIT(REQUEST,STATUS,IERROR)
      INTEGER REQUEST,
                STATUS(MPI_STATUS_SIZE),
                IERROR
```

MPI Definitionen

MPI sehr sorgsam mit Problemen der Sprache

Wichtige Begriffe werden eindeutig definiert

- *Nonblocking*: Der Aufruf kehrt zurück, bevor die Operation abgeschlossen ist und bevor die Ressourcen wiederverwendet werden dürfen
- *Locally blocking*: Bei Rückkehr dürfen die lokalen Ressourcen wiederverwendet werden
  - Hängt nur vom lokalen Prozess ab
- *Globally blocking*: Bei Rückkehr ist die Kommunikationsoperation abgeschlossen
  - Hängt von anderen Prozessen ab
- *Collective*: Alle Prozesse einer Gruppe müssen den Aufruf ausführen

Punkt-zu-Punkt-Kommunikation

Senden

```
MPI_SEND(buf,count,datatype,dest,tag,comm)
   IN buf      Adresse des Sendepuffers
   IN count    Anzahl der Elemente im Puffer
   IN datatype Datentyp des Elements
   IN dest     Rangangabe des Ziels
   IN tag      Nachrichtenkennung
   IN comm     Kommunikator(Gruppe,Kontext)
```

Datentypen: int, long int, float, char, ...

Nachrichten bestehen aus Inhalt und Umschlag

Punkt-zu-Punkt-Kommunikation...

Empfangen

```
MPI_RECV(buf,count,datatype,source,tag,
        comm,status)
  OUT buf     Adresse des Empfangspuffers
  IN count    Anz. der Elemente im Puffer
  IN datatype  Datentyp des Elements
  IN source     Rangangabe der Quelle
  IN tag    Nachrichtenkennung
  IN comm  Kommunikator (Gruppe,Kontext)
 OUT status    Ergebnis des Empfangens
```

Punkt-zu-Punkt-Kommunikation...

Empfangen...

- Gesteuert durch den Umschlag
  `MPI_ANY_SOURCE`, **`MPI_ANY_TAG`** (Wildcard)

- Abfrage mittels
  `MPI_GET_SOURCE()`, **`MPI_GET_TAG()`**

Punkt-zu-Punkt-Kommunikation...

- Semantik der Kommunikation
  - Nachrichtenreihenfolge bleibt erhalten
- Datenumwandlung
  - In heterogenen Netzen automatische Umwandlung
- Modi
  - Normal: lokal blockierend
  - Ready Communication: Senden darf erst aufgerufen werden, wenn Empfangen schon bereit ist (effizientere Realisierung der Datenübertragung möglich)
  - Synchronous Communication: global blockierend; schließt ab, wenn der Empfang begonnen hat

Punkt-zu-Punkt-Kommunikation...

- Nichtblockierende Kommunikation
  - Verbesserte Effizienz durch Überlappung von Berechnung und Kommunikation
- Wichtige Unterscheidung
  - Blockierend / nichtblockierend
    (wann kehrt der Aufruf zurück)
  - Synchron / asynchron
    (wann ist der Auftrag ausgeführt)
  - Prinzip: der Aufruf wird mit einer Referenz versehen
    Durch Abfragen bzgl. der Referenz kann der Status der Ausführung ermittelt werden

Punkt-zu-Punkt-Kommunikation...

Nichtblockierend

```
MPI_ISEND(...,request)   immediate send
MPI_IRECV(...,request)   immediate receive
MPI_TEST(request,flag,status)
                         nichtblockierend
MPI_WAIT(request)        blockierend
MPI_CANCEL(request)
```

MPI „Hello World“

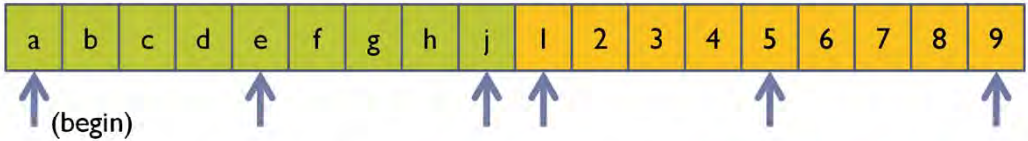
```
#include "mpi.h"
#include <stdio.h>

int main (int argc, char *argv[])
{
  int rank, size;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  printf("Hello World from process %d of %d\n",
                          rank, size );
  MPI_Finalize();
  return 0;
}
```

Abgeleitete Datentypen

- Verwendungszweck
  - Nachrichten mit gemischten Datentypen
  - Nachrichten mit nichtzusammenhängenden Bereichen
- Ein-/Auspacken der Nachrichten erfordert Rechenaufwand
- Effizienz hängt von der Hardware ab (z.B. Direct Memory Access, DMA)

Abgeleitete Datentypen…

Beispiel: Zwei Matrizen mit komplexen Zahlen
Aufgabe: Versende die beiden Diagonalen




```
MPI_TYPE_VECTOR(3/*blocks*/, 1/*element/block*/,
      4/*blockstride*/, MPI_COMPLEX, diag)
MPI_TYPE_CREATE_HVECTOR(2/*blocks*/, 1/*elm/blck*/,
      9*sizeof(MPI_COMPLEX),diag,doublediag)
MPI_TYPE_COMMIT(doublediag)
MPI_SEND(begin,1,doublediag,me,other,comm)
```

Kollektive Kommunikationen

Kollektive Kommunikationen werden immer von allen Mitgliedern einer Gruppe durchgeführt

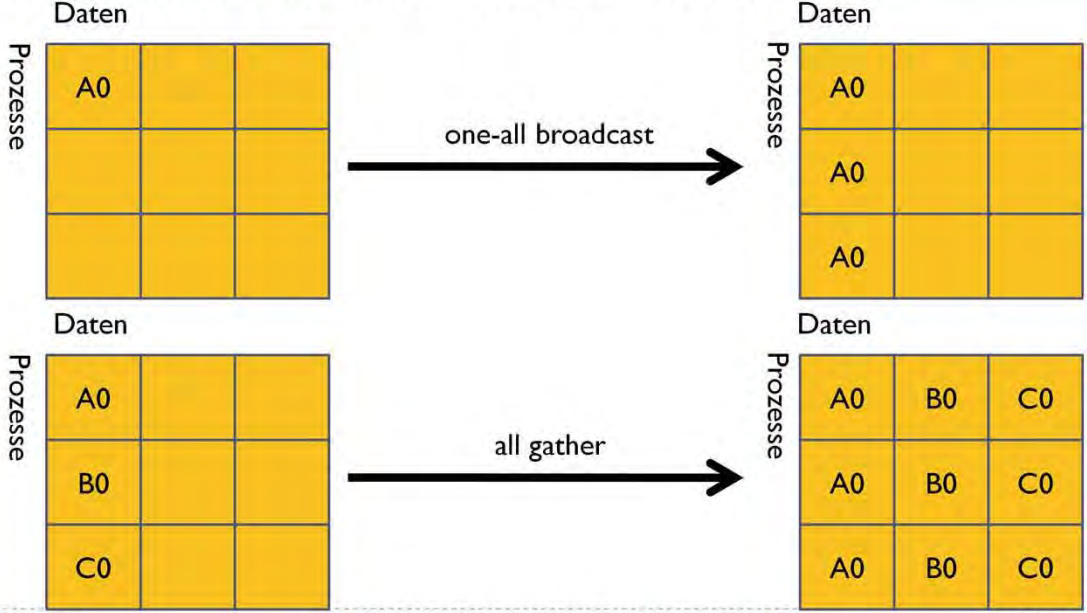
- Broadcast von einem an alle
- Barrierensynchronisation
- Daten einsammeln / verteilen
- Globale Berechnung von Funktionen

Möglicherweise durch spezielle Hardware unterstützt

Spielraum für Implementierungsoptimierungen

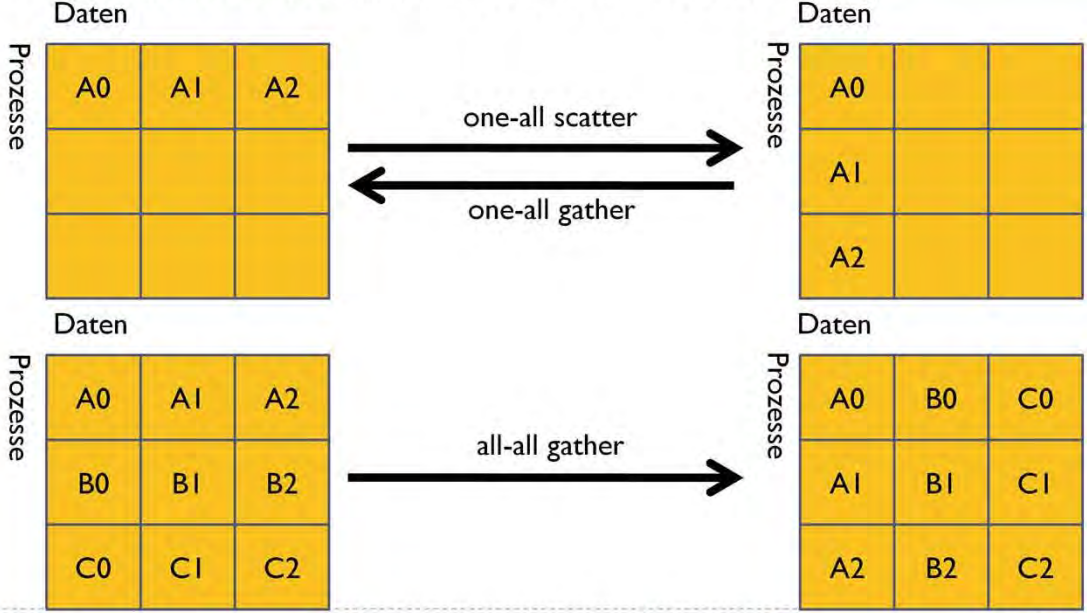
Kollektive Kommunikationen...

Kollektive Funktionen zur Datenverschiebung



Kollektive Kommunikationen...

Kollektive Funktionen zur Datenverschiebung



Kollektive Berechnungen

- Häufig müssen alle Prozesse dieselbe Funktion auf Daten anwenden, z.B. die Summenoperation
- Funktion **`MPI_REDUCE(...,op,...)`**

  Jeder Prozess trägt seinen Datenanteil bei

  Am Ende hat jeder Prozess das Endergebnis

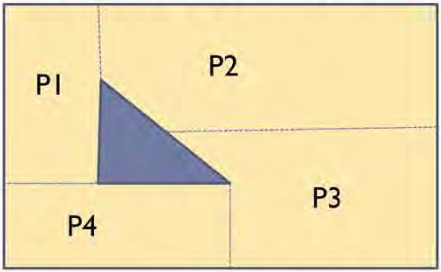
  `max, min, sum, product, AND, OR, XOR`
- Auswertereihenfolge beliebig
  - Evtl. Nichtdeterministisches Ergebnis
- In Parallelrechnern teilweise durch Hardware unterstützt
- Eigene Funktionen möglich (kritisch)

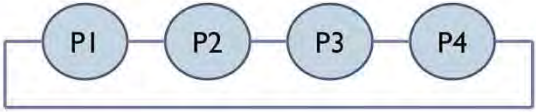
Gruppen, Kontexte, Kommunikatoren

- Neues Konzept, das es vorher nirgends gab
- Problem:
  - Drittanbieter entwickeln Bibliotheken mit Nachrichtenaustausch
  - Kennungen dieser Nachrichten und Rangangaben dürfen nicht mit dem Anwenderprogramm in Konflikt geraten
- Lösung
  - Gruppen fassen zusammengehörige Prozesse zusammen
  - Kontexte unterscheiden logische Teile des Programms
  - Kommunikator: faßt Gruppe und Kontext zusammen
  - Default-Kommunikator: `MPI_COMM_WORLD`

Prozeß-Topologien

Problem: Rangangaben sagen nichts über Beziehungen aus





- Benutzersicht: Nur bestimmte Kommunikationsmuster treten auf
  Zugriff auf Nachbarn über symbolische Namen
- MPI unterstützt die Topologieverwaltung

Profiling-Interface

- Möglichkeit zum Anschluss von Werkzeugen in MPI integriert
- Konzept
  - Unterstütze die Aktivierung von Überwachungen beim Aufruf von MPI-Funktionen
- Realisierung
  - Jede Funktion MPI_xyz muß auch über den Namen PMPI_xyz aufrufbar sein (*profiling*)

Profiling-Interface...

Beispiel: überwache Broadcast-Funktion

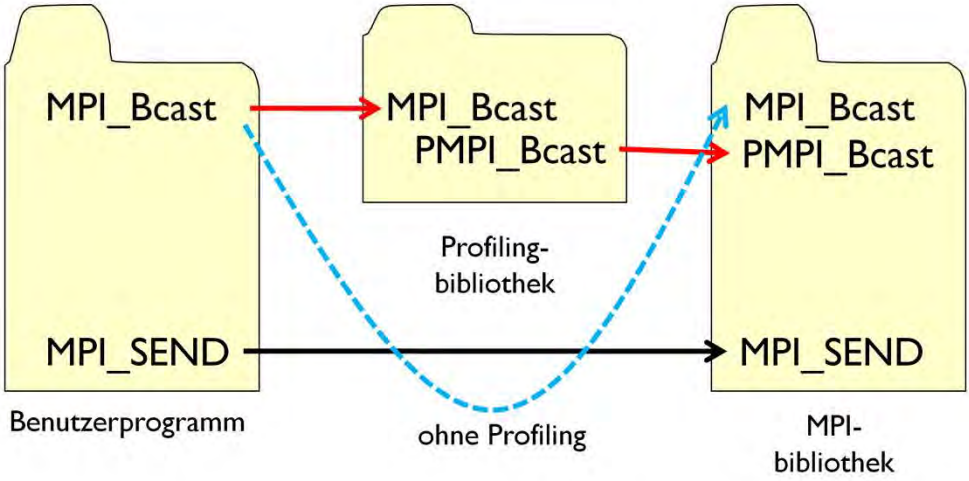
```
int MPI_Bcast(...)
  { int result;
    write_log_entry(...);
    start_timer();
    result=PMPI_Bcast(...);
    stop_timer(); write_log_entry(...);
    return result;
  }
```

Dies definiert eine Profiling-Version der Funktion

Profiling-Interface...

Der Trick: Reihenfolge beim Linken

zuerst: Profiling-Bibliothek, dann: libmpi



Bewertung MPI

- Spezifikation ausschließlich für Nachrichtenaustausch
- Sehr viele Funktionen
- Prozessverwaltung fehlt
- Kein dynamisches Prozesskonzept

  Keine Programme mit dynamisch variierender Prozessanzahl

Ausblick auf MPI-2

- MPI-2 ist eine Erweiterung zu MPI, nicht eine neue Version
- Umfasst Klarstellungen zu MPI und Erweiterungen
- Wichtige Erweiterung: Prozessverwaltung

  (Vorher machte jeder Hersteller was er wollte)
- Wichtige Erweiterung: Ein-/Ausgabe

  (Idee: äquivalent zu Senden und Empfangen von Nachrichten)
- Nachteil: sehr viele neue Funktionen

Vergleich der Ansätze

	Pthreads	OpenMP	MPI
Skalierbarkeit	Begrenzt	Begrenzt	Ja
Fortran / C und C++	Ja? / Ja	Ja / Ja	Ja / Ja
Hohe Abstraktion	Nein	Ja	Nein
Leistungsorientierung	Nein	Ja	Ja
Portierbarkeit	Ja	Ja	Ja
Herstellerunterstützung	Unix/SMP	Verbreitet	Verbreitet
Inkrement. Parallelisierung	Nein	Ja	Nein

Programmiermodell Nachrichtenaustausch
Zusammenfassung

- Relevante Probleme beim Nachrichtenaustausch:
  Kommunikationsschemata, Effizienz, Prozessverwaltung
- MPI ist eine Spezifikation eines API zum Nachrichtenaustausch
- Punkt-zu-Punkt-Kommunikation mit vielen Varianten möglich:
  synchron/asynchron, blockierend/nichtblockierend
- Abgeleitete Datentypen vereinfachen die Kommunikation
- Gruppen und Kontexte dienen zur wechselseitigen Abgrenzung von Programmteilen
- MPI-2 erweitert MPI um wesentliche Aspekte