Programmiermodell Nachrichtenaustausch

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

Literatur:

• Mark Snir et al.: MPI – The Complete Reference. Volume 1, The MPI Core. Second Edition. MIT-Press, 1998.
• William Gropp et al.: Using MPI – Portable Parallel Programming with the Message-Passing Interface. Second Edition. MIT-Press, 1999.

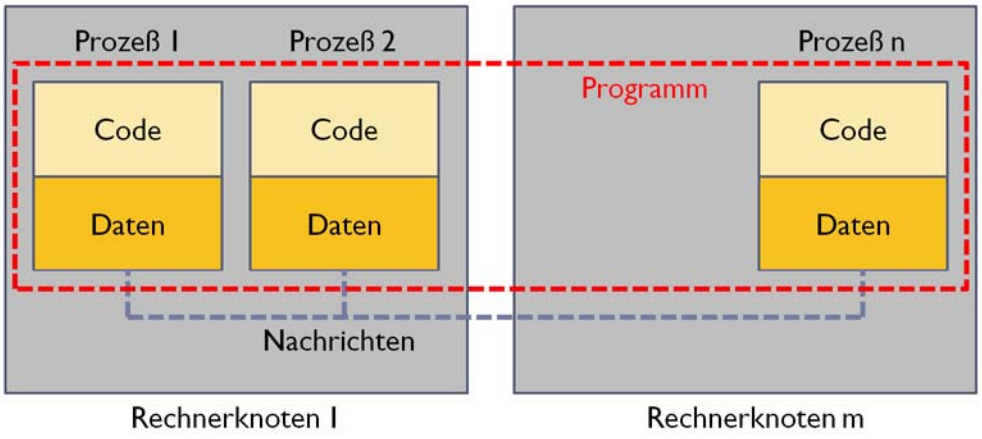
Prog´modell Nachrichtenaustausch

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



241 Hochleistungsrechnen - © Thomas Ludwig 02.05.2010

Nochmal zum Begriff des Rechnerknotens:
• Früher war ein Rechnerknoten ein einzelner Rechner, meist so eine Art PC, von denen man mehrere in einem Cluster zusammengebaut hat. D.h.: ein Knoten = ein Prozessor (=ein Core). Darauf bringt man einen Prozeß der Anwendung zum Ablauf.
• Heute ist ein Knoten z.B. am System „Blizzard“ des DKRZ ein Einschub mit 16 Prozessoren zu je 2 Cores, also ein 32-Prozessor-System.

Die Zuweisung von Jobs zum Rechner erfolgt durch die Angabe der Anzahl von Knoten, die man haben möchte. Es werden den Benutzern immer ganzzahlige Vielfache dieser Knoten zugewiesen.

Auf einem Knoten, der dann gemeinsamen Speicher hat, kann ein Prozeß nochmal in Threads unterteilt werden, die dann wiederum Code-Aufteilung und/oder Datenaufteilung folgen.

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

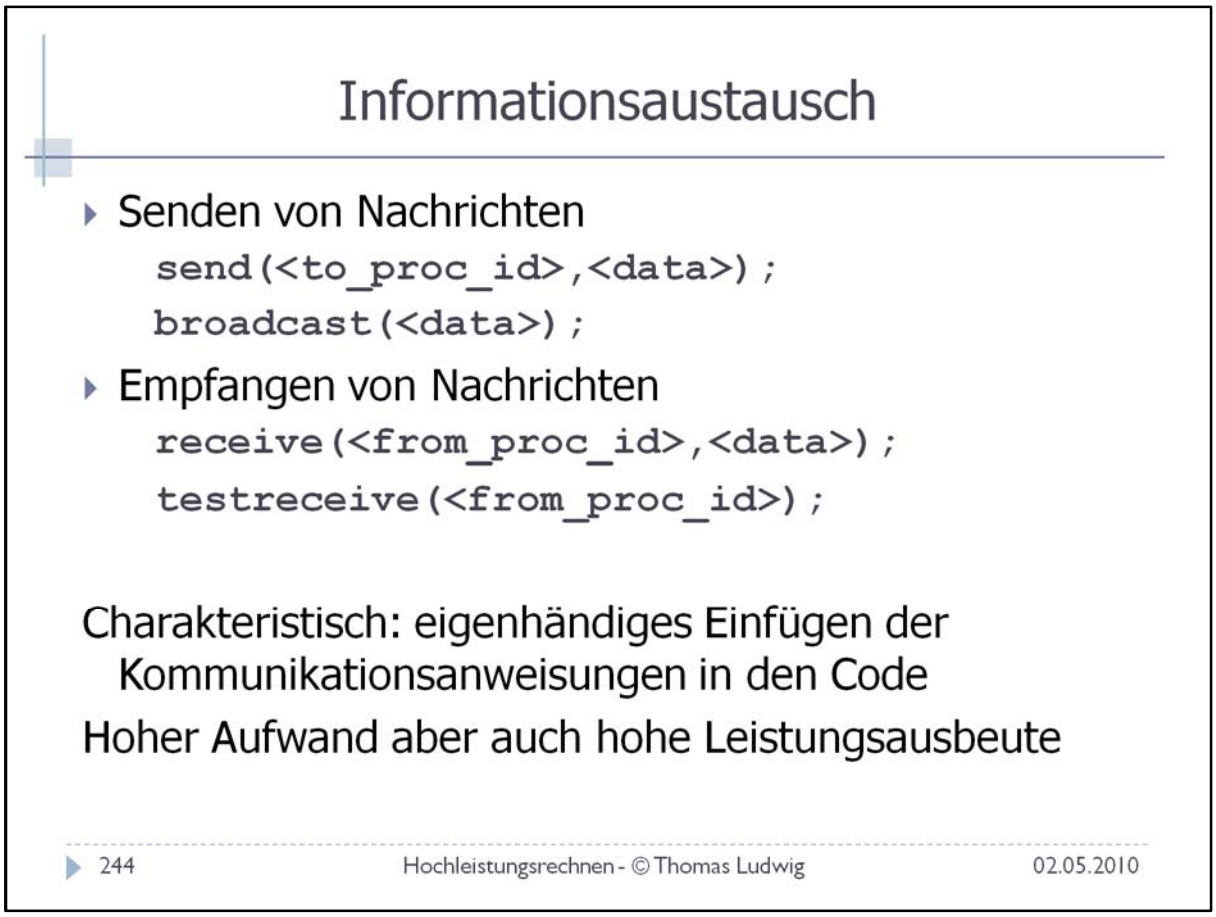
- Ähnlich zur Thread-Erzeugung

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

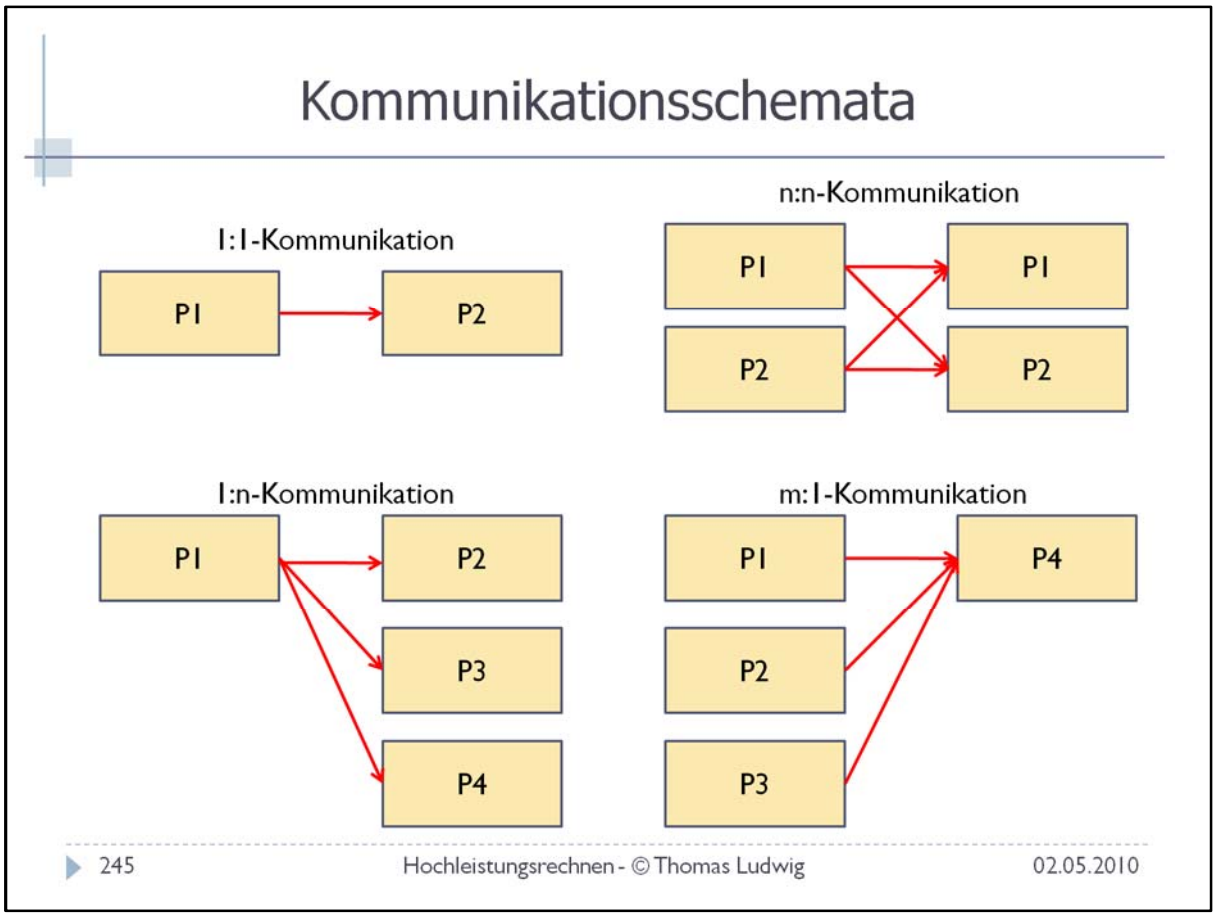
- Wenn nur ein Programmcode existiert

```
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



Die Programmierung des Nachrichtenaustausch wird auch als die Maschinenprogrammierung des parallelen Rechnens bezeichnet, weil sie auf einer sehr niedrigen Abstraktionsebene ansetzt.



1:1-Kommunikation: Ein Sender-Prozeß sendet an genau einen Empfänger-Prozeß. Der Sender kennt die Adresse vom Empfänger, der Empfänger die vom Sender.

1:n-Kommunikation: Ein Sender-Prozeß sendet an viele Empfänger-Prozesse mit einem einizigen Aufruf. Geht die Nachricht an alle anderen, so spricht man von Broadcast, geht sie an eine echte Teilmenge, so spricht man von Multicast.

m:1-Kommunikation: Insgesamt m Sender-Prozesse senden an ein und denselben Empfängerprozeß. Man unterscheidet syntaktisch Fälle, bei denen der Empfänger die Sender kennt und solche, wo das nicht der Fall ist. Ersterer wird verwandt, wenn z.B. Ergebnisse korrekt zusammengefügt werden müssen, letzterer, wenn man z.B. Teilergebnisse nur aufaddiert. Der zweite Fall kann meist mit besserer Effizienz implementiert werden.

n:n-Kommunikation wird verwendet, wenn eine Menge von Prozessen untereinander Daten austauschen, so daß jeder an Sende- und Empfangsoperationen beteiligt ist.

Alle komplexen Kommunikationsschemata können immer durch eine Menge 1:1-Kommunikationen dargestellt werden. Manchmal wird das auch so implementiert, das ist dann schnell und ineffizient. Vielfach wird aber gerade hier sehr viel Aufwand in interne Optimierungen der Bibliotheken gelegt.

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

Siehe: http://en.wikipedia.org/wiki/Parallel_Virtual_Machine

Message Passing Interface (MPI)

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

- MPI Standard 1995 (nur Kommunikation)
- MPI-2 Standard 1997 (der nötige Rest)

- Vorteile eines Standards: Portabilität, Einfachheit
  Vorher etwa ein Dutzend konkurrierende Ansätze
- Probleme: Standardkonformität der Implementierungen

Siehe:
- http://www.mpi-forum.org/
- http://en.wikipedia.org/wiki/Message_Passing_Interface

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!

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-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

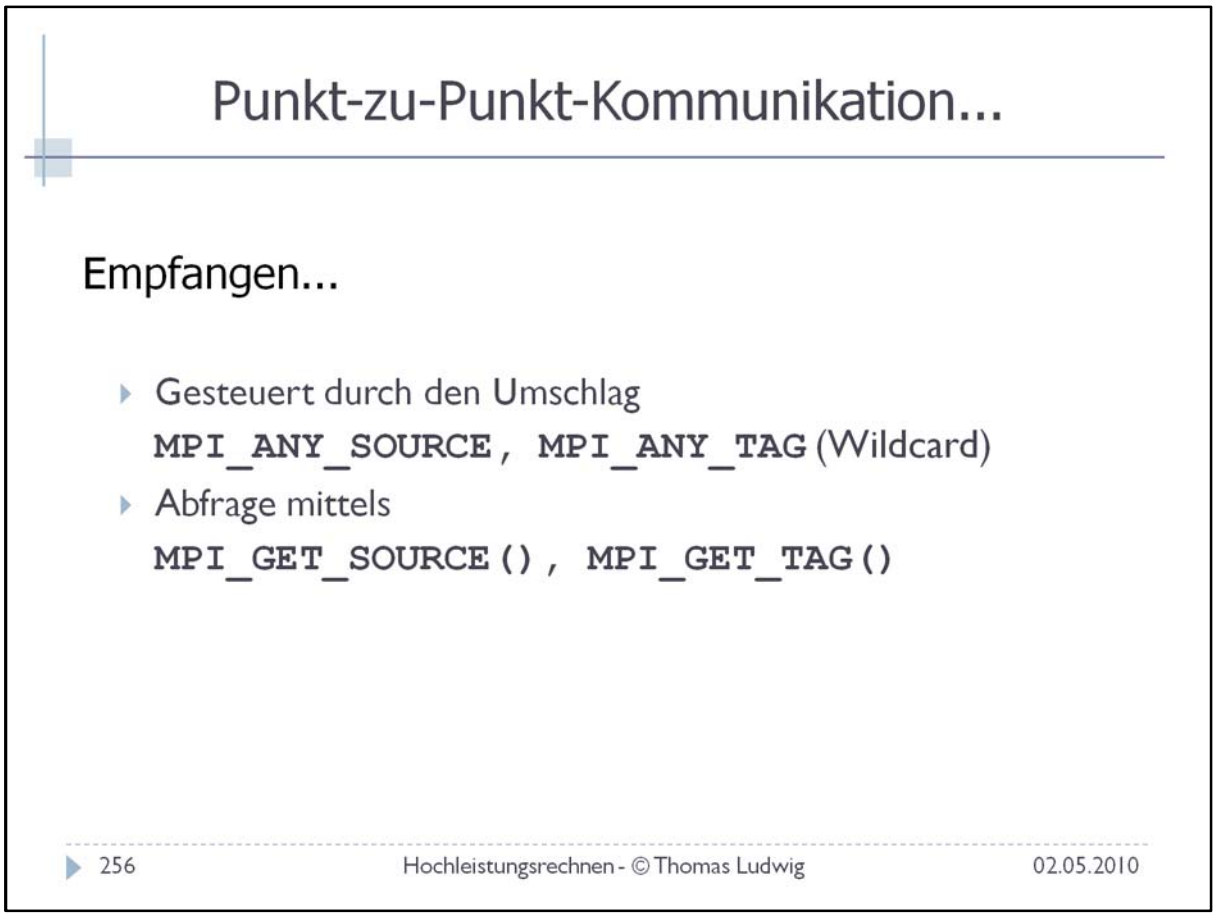
```
MPI_SEND(buf,count,datatype,dest,tag,comm)
   IN buf       Adresse des Sendepuffers
   IN count     Anzahl der Elemente im Puffer
   IN datatype  Datentyp des Elements
   IN dest      Rangangabe des Ziels
   IN tag       Nachrichtenkennung
   IN comm      Kommunikator(Gruppe,Kontext)
```

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     Kommunika. (Gruppe,Kontext)
OUT status    Ergebnis des Empfangens
```



Aufgabe: In MPI gibt es zum Senden kein `MPI_ANY_DEST`. Überlegen Sie, wie die Semantik hiervon sein sollte. Wofür könnte man das gebrauchen? Wie könnte man es trotzdem implementieren?

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

MPI „Hello World"

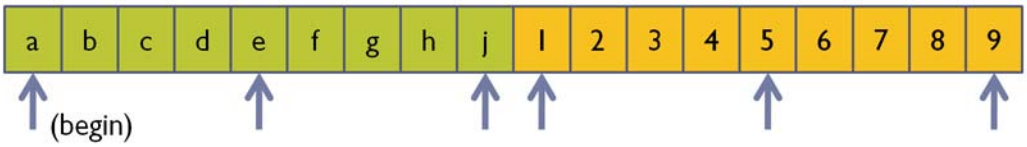
```
#include "mpi.h"
#include <stdio.h>

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)
```


MPI_TYPE_VECTOR hängt n Blöcke zusammen (hier: 3), die die Blocklänge l (hier: 1) haben und im Abstand von s Elementen des Ausgangsdatentyps (stride, Versatz) (hier: 4) liegen. Ausgangsdatentyp ist hier MPI_COMPLEX, der neue Datentyp ist hier diag.

MPI_TYPE_CREATE_HVECTOR mißt den Versatz in Byte.

Das Konzept stützt sich sehr auf die Kenntnis der speicherinternen Organisation der einzelnen Datentypen.

Kollektive Kommunikationen

Kollektive Kommunikationen werden immer von allen Mitgliedern einer Gruppe durchgeführt

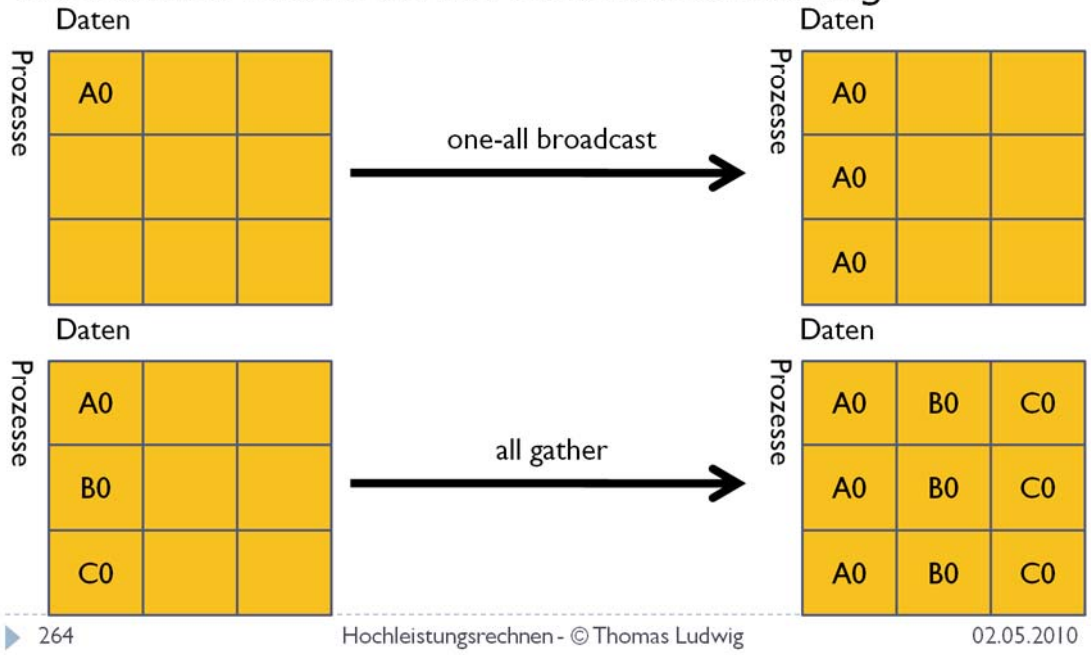
- Broadcast von einem an alle
- Barrierensynchronisation
- Daten einsammeln / verteilen
- Globale Berechnung von Funktionen

Möglicherweise durch spezielle Hardware unterstützt

Spielraum für Implementierungsoptimierungen

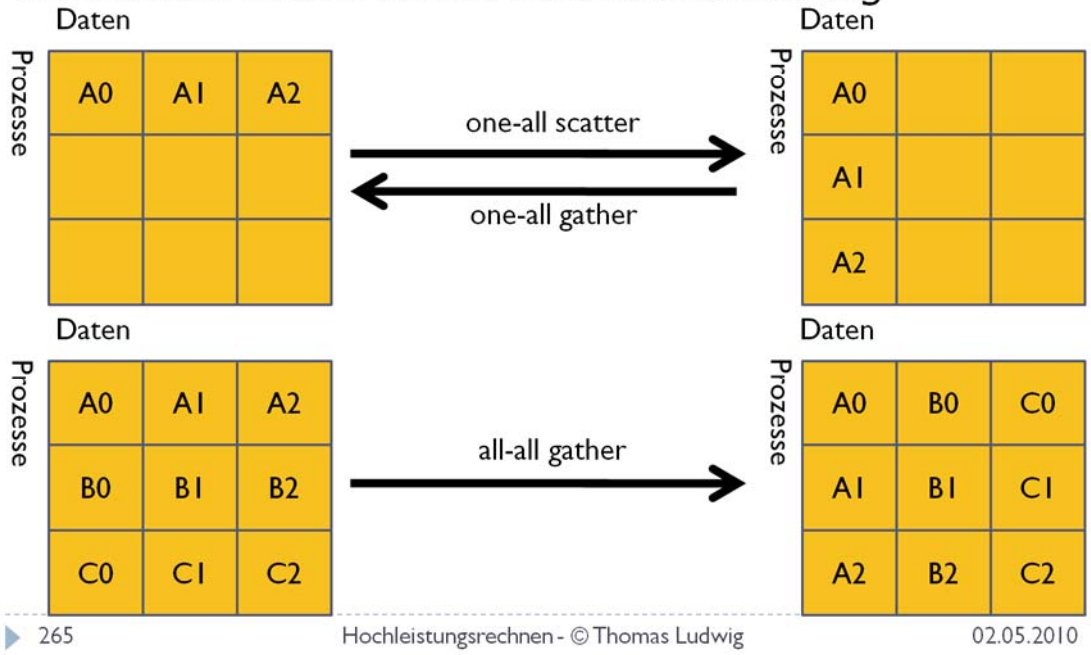
Kollektive Kommunikationen...

Kollektive Funktionen zur Datenverschiebung



Kollektive Kommunikationen...

Kollektive Funktionen zur Datenverschiebung



265 Hochleistungsrechnen - © Thomas Ludwig 02.05.2010

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)

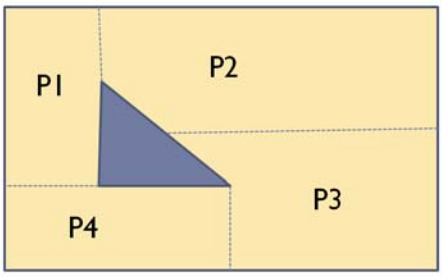
Die Operation op wird immer als assoziativ angenommen, d.h. a op (b op c) = (a op b) op c. Außerdem sind alle vordefinierten Funktionen auch kommutativ, d.h. a op b = b op a . Z.B. ist aber eine Gleitkommaaddition im Rechner nicht absolut kommutativ und assoziativ wegen der begrenzten Rechengenauigkeit.

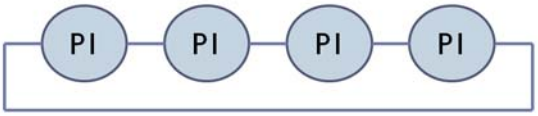
Gruppen, Kontexte, Kommunikatoren

- 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

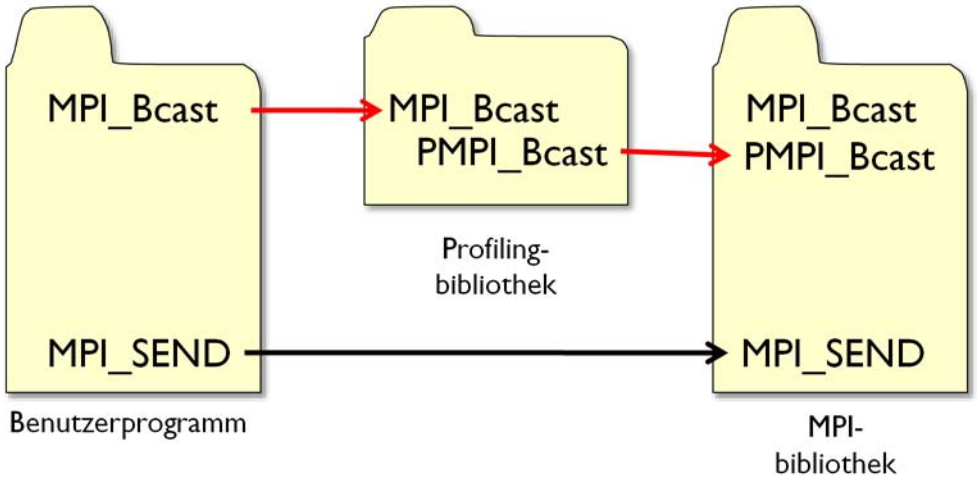
```
int MPI_Bcast(...)
  { int result;
    write_log_entry(...);
    start_timer();
    result=PMPI_Bcast(...);
    stop_timer(); write_log_entry(...);
    return result;
  }
```

Dies definiert eine Profiling-Version der Funktion

Profiling-Interface...

Der Trick: Reihenfolge beim Linken

Profiling-Bibliothek – libmpi – libpmpi



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

Prog´modell 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