Programmierung paralleler Systeme Message Passing mit PVM

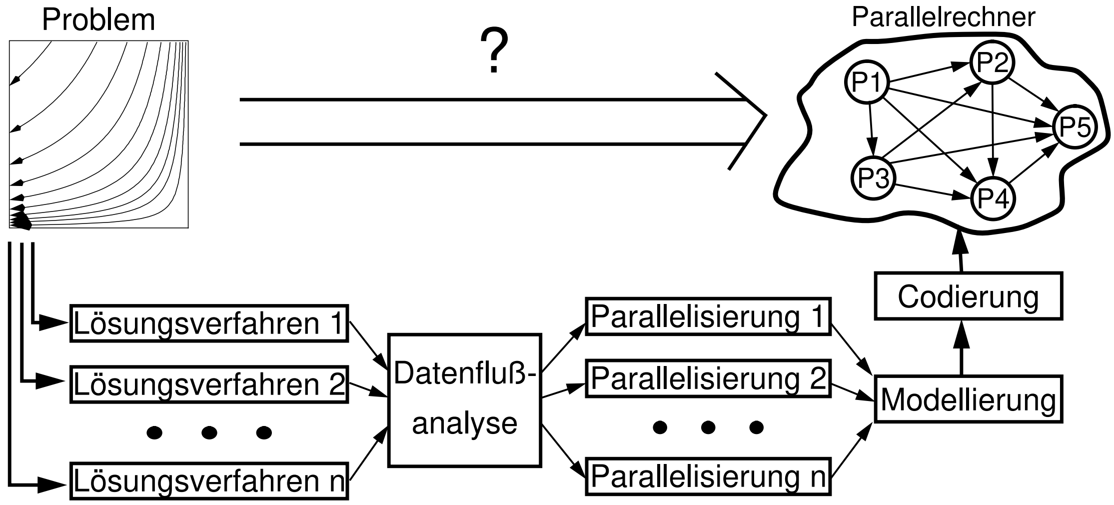
Frank Bellosa

Universität Erlangen-Nürnberg — IMMD IV
bellosa@informatik.uni-erlangen.de

Übersicht

- ❏ Programmiermodelle
- ❏ PVM Programmiermodell
- ❏ Einführung in PVM
- ❏ Debugging von PVM-Applikationen
- ❏ Tracing von PVM-Applikationen mit xpvm
- ❏ Installation am CIP-Pool (Workstation-Cluster)
- ❏ Installation am RRZE (Convex SPP)
- ❏ Tips und Tricks zur PVM-Programmierung
- ❏ Batch-Betrieb auf dem Convex SPP

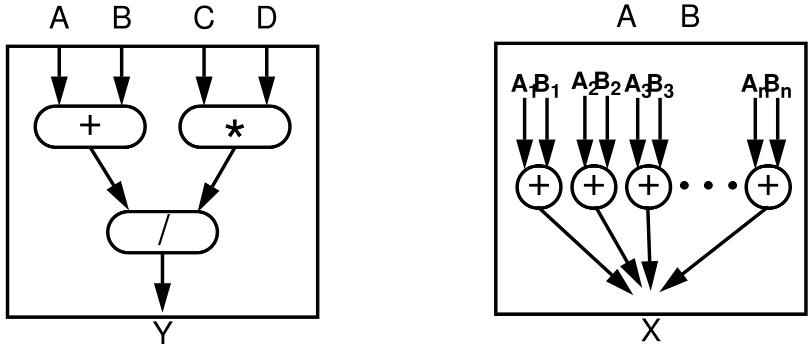
Vom Problem zum Programm



Vom Problem zum Programm

Datenflußanalyse

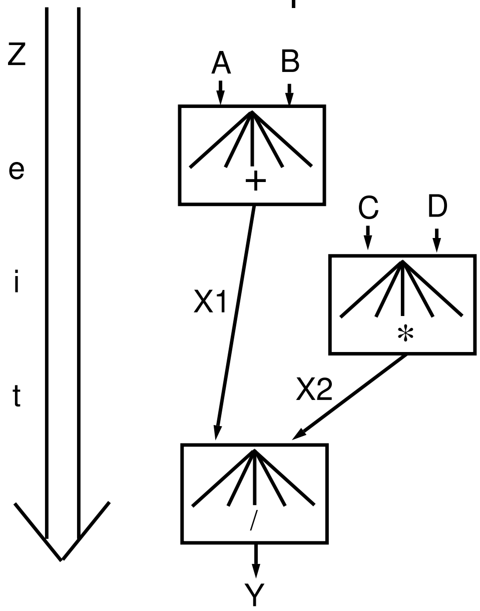
- ❏ Beschreibung des Datenflusses von der Eingabe bis zur Ausgabe
- ❏ Darstellung i.a. mit Hilfe von Datenflußgraphen (Knoten ≈ Operation, Kanten ≈ Datentransport)
- ❏ Verfeinerung bis auf die Ebene der Elementaroperationen möglich
  - ➡ Beschreibung des Abhängigkeitsverhältnisses von Funktionsblöcken
- ❏ Beispiel: Lösungsverfahren Y=(A+B)/(C∗D) mit A, B, C, D Vektoren



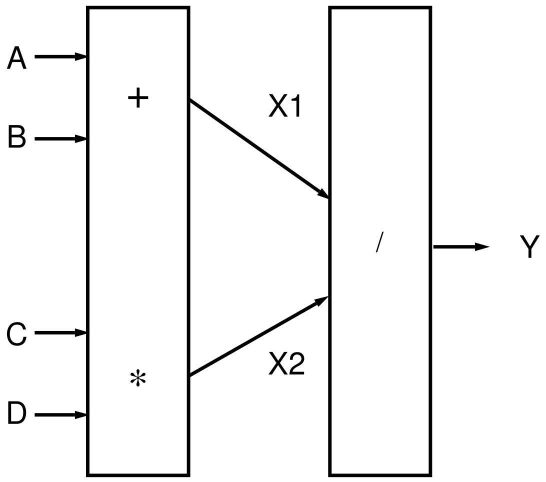
Vom Problem zum Programm

Parallelisierung

Datenparallelität



Prozeßparallelität



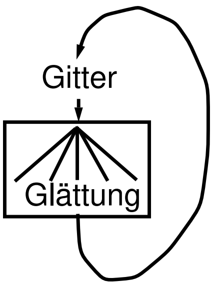
Vom Problem zum Programm

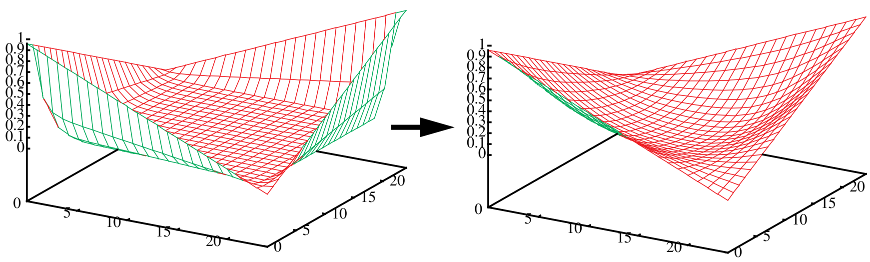
Beispiel

❏ Glättung eine Gitters

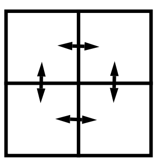
$$\text{gitter}[x, y] = (\text{gitter}[x+1, y] + \text{gitter}[x-1, y] + \text{gitter}[x, y-1] + \text{gitter}[x, y+1])/4$$

– Datenflußanalyse:





– Parallelisierung



– Gebietspartitionierung durch Bipartitionierung

↔ = Kooperation = Koordination + Kommunikation

Programmiermodelle

Parallele Prozesse + Message Passing

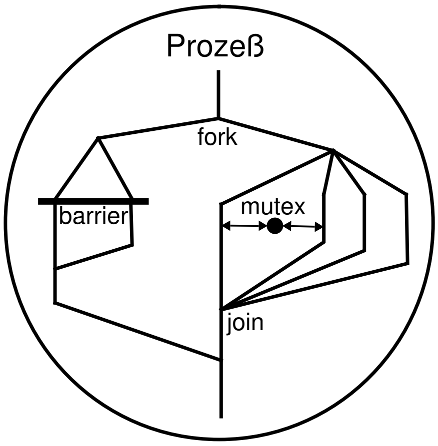
send(Botschaft, Empfänger-Adresse)

Botschaft = receive(Sender-Adresse)

Parallele Prozesse + Shared Memory

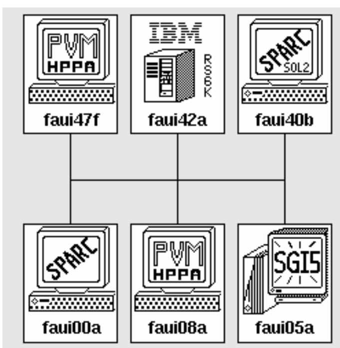
shm_id = shm_get(Schlüssel, Größe, Zugriffsrechte)	
ptr = shm_at(shm_id)	shm_dt(prt)
semaphore = sema_create(Schlüssel, Zugriffsrechte)	
wait(semaphore)	signal(semaphore)

1 Prozeß + Parallele Threads + Shared Memory

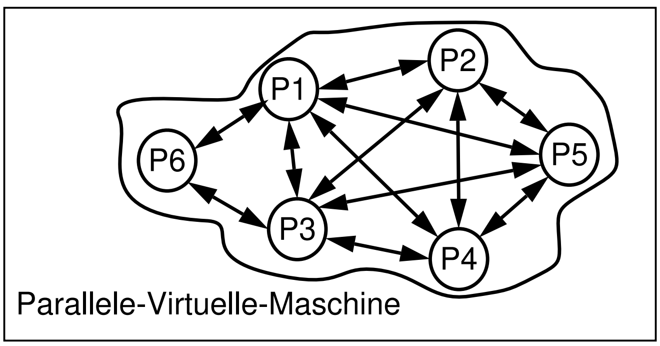


Einführung in PVM3

❏ Physikalische Sicht



❏ Logische Sicht



PVM3-Programmiermodell

- ❏ Sammlung von Rechen-Einheiten, die über expliziten Botschaftenaustausch kommunizieren und sich synchronisieren.
- ❏ Ausführende Instanz einer Rechen-Einheit ist ein (Unix-)Prozeß.
- ❏ Jeder Prozeß wird durch eine Prozeß-Id, die von der virtuellen Maschine vergeben wird, eindeutig gekennzeichnet.
- ❏ Ein Prozeß hat einen sequentiellen Kontrollfluß, der durch ein C- oder Fortran-Programm beschrieben wird. Die PVM3-Bibliothek wird vom Compiler dazugebunden.
- ❏ Die Partitionierung erfolgt explizit und manuell (Keine Unterstützung !)
- ❏ Die Verteilung der Prozesse auf die Rechenknoten erfolgt automatisch oder explizit.
- ❏ Es existiert **keine** implizite Lastverteilung.

Informationen zu PVM3

- ❏ FTP-Server am LRZ-München:
  `ftp ftp.lrz-muenchen.de    cd pub/comp/parallel/pvm3`
- ❏ FTP-Server am ZIP-Berlin:
  `ftp elib.zib-berlin.de    cd netlib`
- ❏ X-Window ORNL-netlib-Client:
  `/local/X11R5/bin/xnetlib` (SPP)
- ❏ News:
  comp.parallel.pvm, comp.parallel
- ❏ WWW-Seite zur Vorlesung:
  http://www4.informatik.uni-erlangen.de/IMMD-IV/Lehre/V_PPS.html
  http://www4.informatik.uni-erlangen.de/IMMD-IV/Lehre/PPS/spp.html

Elemente von PVM3

- ❏ Starten und Herunterfahren einer parallelen virtuellen Maschine
- ❏ Information
  - Konfiguration (Anzahl der zur Verfügung stehenden Knoten, ...)
  - Fehlermeldung (Ausgabe an die PVM3-Konsole)
- ❏ Prozeß-Management
  - Initialisierung (Prozeß-Generierung und Initialisierung)
  - Bildung von Prozeßgruppen
  - Terminierung
- ❏ Kommunikation
  - Senden (blockierend/nicht blockierend)
  - Empfangen (blockierend/nicht blockierend, mit Vorabfrage)
  - Broadcast
  - Multicast
- ❏ Synchronisation
  - Barriere

Starten einer parallelen virtuellen Maschine (CIP)

(1) Auswahl einer Menge von Rechenknoten (z.B. faui07a-faui07d)

(2) Anlegen eines hostfiles, das alle gewünschten Rechenknoten und eine Pfadangabe für den ausführbaren Code enthält.

```
# hostfile for CIP-Cluster
# HP-compute nodes
atlan
faui07b
faui07c
faui07d
```

(3) Starten der parallelen virtuellen Maschine

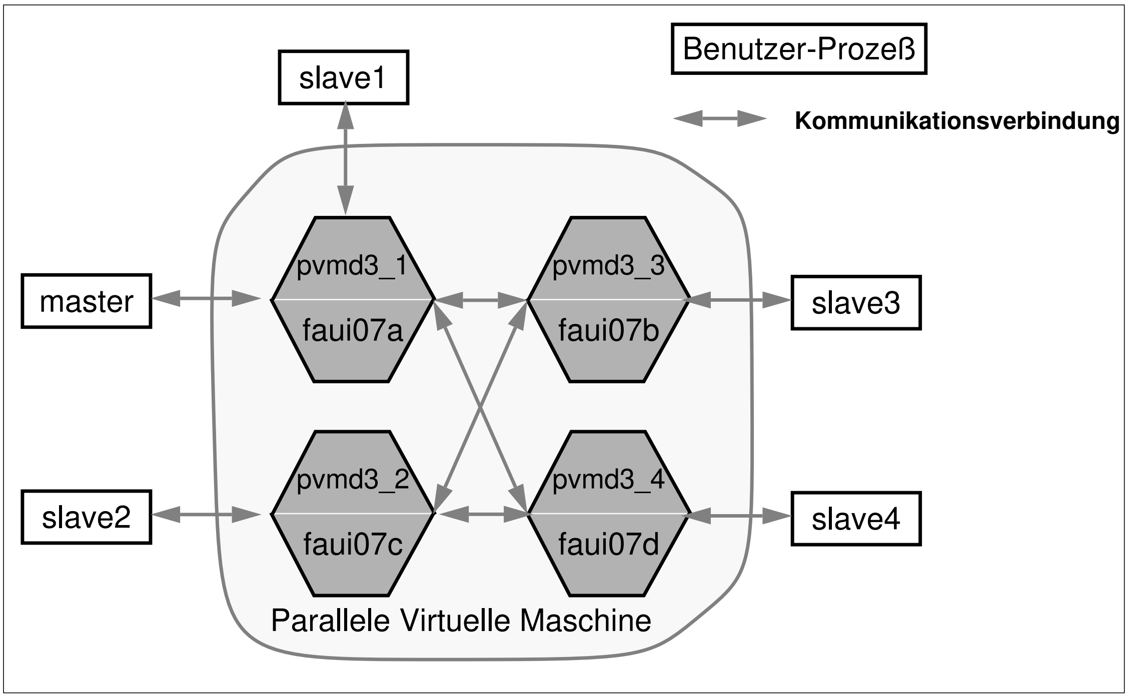
```
pvmd hostfile &
```

WICHTIG: pvmd **muß** auf einem der im hostfile enthaltenen Rechenknoten gestartet werden!

(4) Starten einer PVM-Konsole

```
pvm
```

Aufbau einer parallelen virtuellen Maschine



Probleme beim PVM-Start

❏ Starten der pvmd3s durch den Unix **`remsh`**-Mechanismus

- Eintrag aller Knoten in **`~/.rhosts`** oder **`/etc/hosts.equiv`** erforderlich, um ein Starten von Prozessen auf alle Knoten per **`remsh`** zu ermöglichen. Am CIP-Pool **`.rhosts`**-Datei erforderlich, da kein Eintrag in **`hosts.equiv`**
- Beim Starten eines Prozesses auf einem Rechenknoten darf keine Ausgabe durch **`.cshrc`** erfolgen!
  Einfacher Test:

```
> remsh faui07a hostname
faui07a
>
```

  Es darf bei diesem Test keinerlei weitere Ausgabe erscheinen!

  Wenn die .cshrc beim Einloggen eine Ausgabe durchführen soll, ist folgender Eintrag nützlich:

```
...... # Allgemeine Einträge
if ( { tty -s } && $?prompt) then
...... # Einträge für interaktiven Betrieb und Ausgabe
endif
```

Stoppen einer parallelen virtuellen Maschine

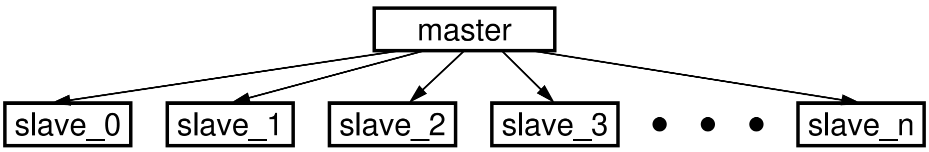
- Terminierung der PVM-Daemons (pvmd3) durch das "halt"-Kommando der Konsole
- Beseitigung der Dateien in /tmp , die der pvmd3 angelegt hat.

  Authentisierungsfile: **`-rw------- /tmp/pvmd.10407`**
  Logfile für stderr: **`-rw------- /tmp/pvml.10407`**

  Diese Dateien verhindern ein erneutes Starten eines PVM-Daemons
- **Empfehlung für CIP-Pool und Convex-SPP:**

  `pvm_clean hostfile`
  - Löschen der /tmp-Dateien
  - Killen aller pvmd3-Daemons

Aufbau einer parallelen Applikation mit PVM




```
/* master-Programm*/
#include <pvm3.h>
main()
{
 int mytid, nodes, narch, *slave_tid;
 struct hostinfo *hostp;
 mytid = pvm_mytid();
 pvm_config(&nodes, &narch, &hostp);
 slave_tid=(int *)malloc(nodes*sizeof(int));
 pvm_spawn("slave",(char**)0,PvmTaskDefault,
           "", nodes, slave_tids);
 pvm_joingroup("BARRIERE");
 pvm_barrier("BARRIERE",nodes+1);
 pvm_lvgroup("BARRIERE");
 pvm_exit();
}
```

```
/* slave-Programm*/
#include <pvm3.h>
main()
{
 int my_tid, nodes, narch;
 struct hostinfo *hostp;
 mytid = pvm_mytid();
 pvm_config(&nodes,&narch,&hostp)
 pvm_joingroup("BARRIERE");
 pvm_barrier("BARRIERE",nodes+1);
 pvm_lvgroup("BARRIERE");
 pvm_exit();
}
```

Prozeßinformation in PVM3

Anmelden bei der Parallelen Virtuellen Maschine

```
mytid = pvm_mytid()
```

↓

Prozeßinformation

```
pvm_config(int *nprocessors, int *narch, .. **hostp)
pvm_tasks(int where,int *nproc, struct ..**taskp);
int tid = pvm_parent()
int status = pvm_pstat(tid)
int mstat = pvm_mstat (char *host)
```

↓

Abmelden bei der Parallelen Virtuellen Maschine

```
pvm_exit()
```


Prozeßkontrolle in PVM3

Starten von neuen Prozessen

```
numt = pvm_spawn(char *task, char **argv, int flag,
                 char *where, int ntask, int *tids)
```

↓

Bildung von Prozeßgruppen

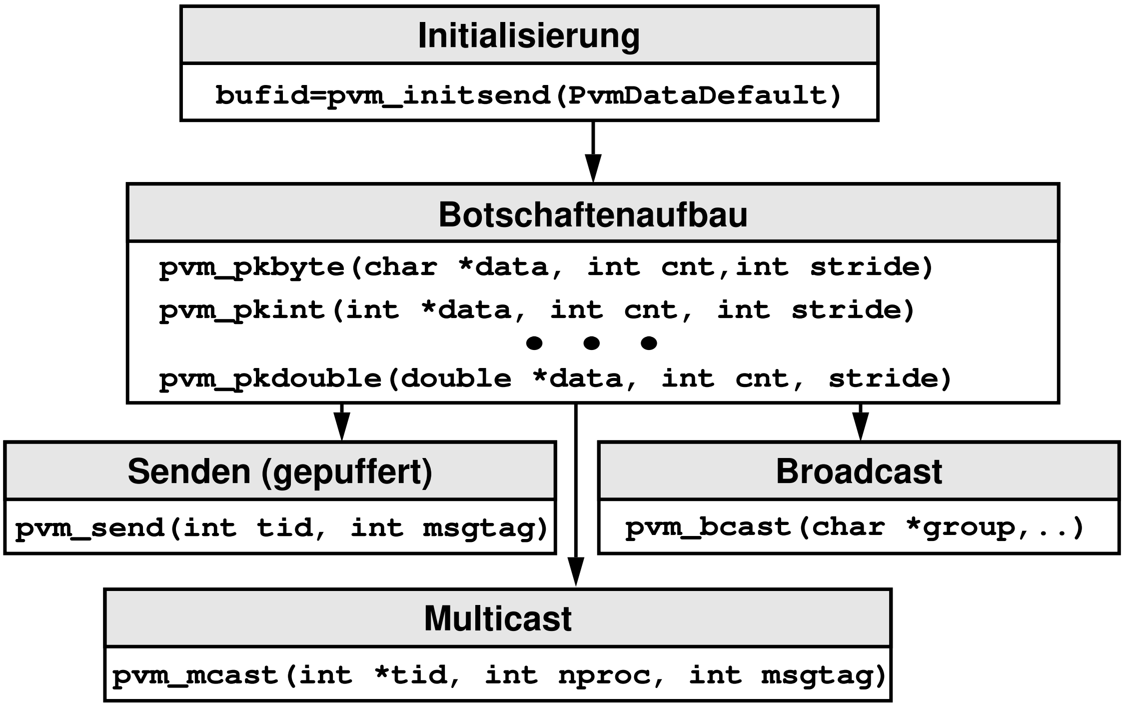
```
int inum = pvm_joingroup(char *groupname)
int size = pvm_gsize(char *groupname)
pvm_barrier(char *groupname, int count)
pvm_lvgroup(char *groupname)
```

↓

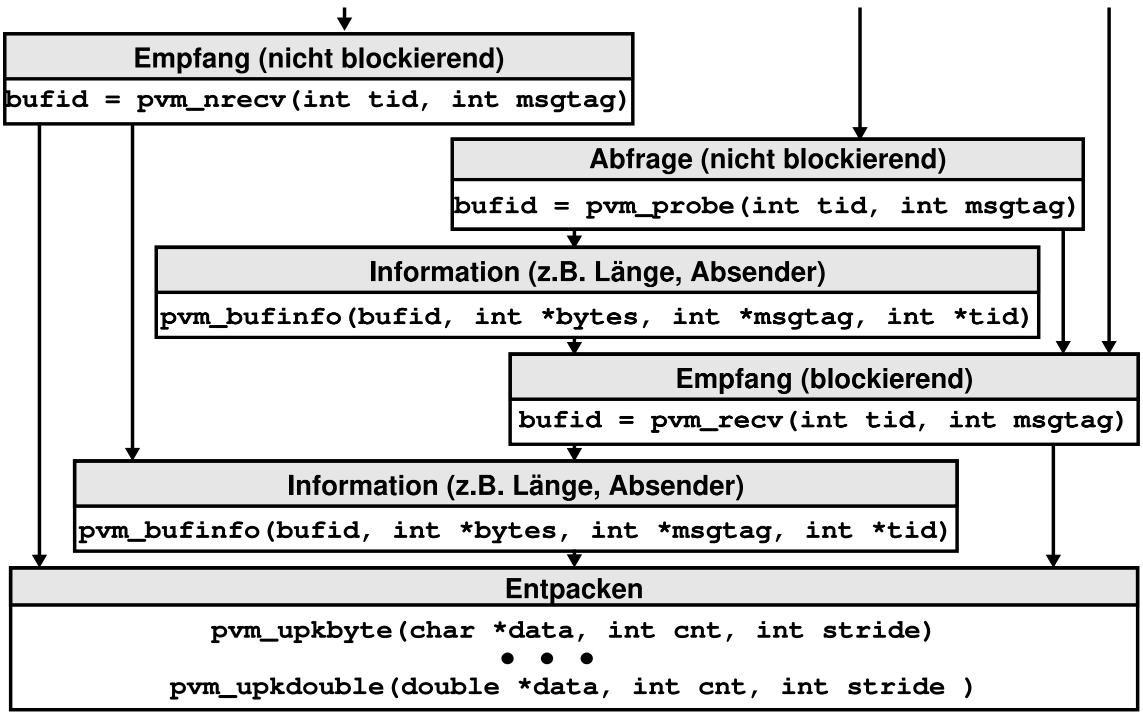
Terminieren von Prozessen

```
pvm_kill(int tid)
```

Botschaftenaustauch mit PVM3 (Senden)



Botschaftenaustauch mit PVM3 (Empfangen)



PVM Verzeichnisstruktur

❏ Empfohlene Benutzerumgebung wie in:

```
/home/gsm/misc/demo
```

❏ Wesentliche Environment Variablen:

```
setenv PVM_ROOT /local/pvm3          (CIP)
setenv PVM_ROOT /usr/convex/pvm      (SPP)
setenv PVM_ARCH `$PVM_ROOT/lib/pvmgetarch`
```

❏ Empfohlener Aufbau für das HOME-Directory:

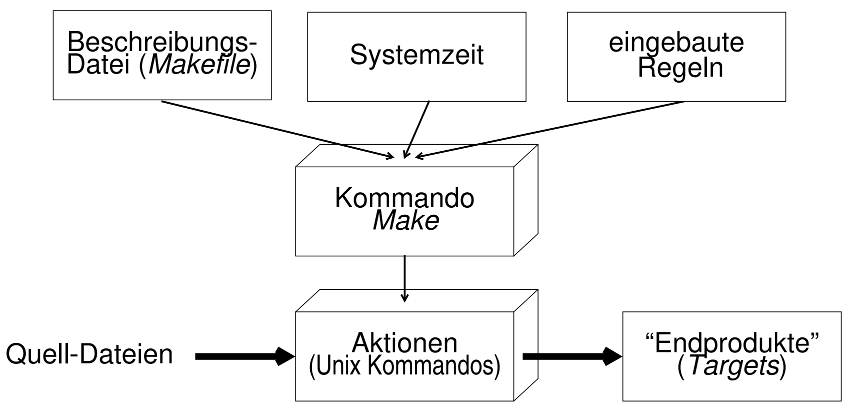
```
pvm3/bin/
           CSPP/
               pvmgs -> /usr/convex/pvm/lib/CSPP/pvmgs
           HPPA/
                grid
               pvmgs -> /local/pvm3/lib/HPPA/pvmgs
pvm3/bsp1/
            CSPP/
            HPPA/grid.o
            Makefile.aimk
            grid.c
```


Wozu ein Makefile ?

❏ Problemstellung

- große Zahl von einzelnen Quell-Dateien
- mit komplexen wechselseitigen Abhängigkeiten und
- einer Vielzahl von Verarbeitungsschritten von den Quell-Dateien bis zum "Endprodukt" (Dokument, Programm, ...)

❏ Funktionsweise



PVM-Makefiles

❑ Aufbau eines Makefile.aimk

```
# makefile for test-example
PVM_LIB     = -lgpvm3 -lpvm3
PVM_IDIR    = $(PVM_ROOT)/include
PVM_LDIR    = $(PVM_ROOT)/lib/$(PVM_ARCH)
XDIR        = ../../bin/$(PVM_ARCH)
VPATH       = ..
CFLAGS      = $(ARCHCFLAGS) -I$(PVM_IDIR) -L$(PVM_LDIR)
COMPILE = $(CC) $(CFLAGS) -o $@ $< $(PVM_LIB) -lm ; cp $@ $(XDIR)
PROGS       = test

default: $(PROGS)

test: test.c
      $(COMPILE)
.c.o:
      $(CC) $(CFLAGS) -c $<
```


PVM-Tools auf dem CIP-Pool/SPP

❑ **aimk** (architekture independent make) führt das zur augenblicklichen Architektur passende Makefile aus.

❑ **pvmgetarch** gibt die Architektur des gerade genutzten Rechners aus

❑ "Entfernen von Altlasten im Subcomplex (SPP)" : pvm_clean

❑ "Entfernen von Altlasten im CIP-Cluster" : pvm_clean hostfile

Tips und Tricks zur Parallelprogrammierung

(1) Alle Aktionen, die möglich sind, können fehlschlagen.

(2) Wenn Aktionen fehlschlagen, sollte man dies erkennen können.

(3) Programmläufe, die nicht korrekt terminieren, können nachfolgende beeinflussen

- ➡ Kontrolle, ob noch irgendwo Prozess unkontrollliert laufen.
- ➡ Testläufe auf wenigen (1-2) Prozessoren durchführen

(4) Soviel Strukturierung wie nur irgend möglich !!!

Entwurf des Programmrumpfes (Prozeß-Verwaltung und Kommunikation)

↓

Implementierung und Test der Einzelkomponenten mit Testdaten

↓

Zusammenfügen der Einzelkomponenten und Test des Programmrumpfes

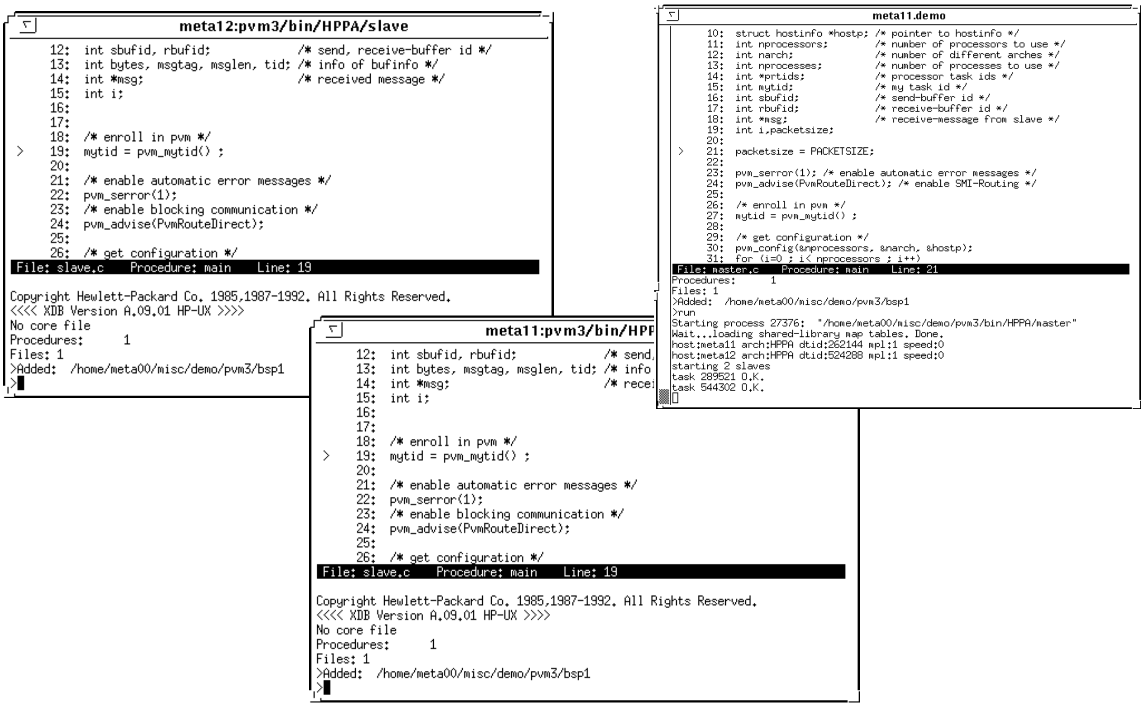
↓

Einfügen der "Numerik"

Debugging

- ❏ Am Besten gar nicht !
- ❏ Multi-Window-Debugging
  - Pro Prozeß ein xterm, in dem der Prozeß unter xdb-Debugger Kontrolle läuft
  - `pvm_spawn(task, argv, PvmTaskDebug, where, ntask, tids)`
  - Compiler-Option: -g (HP) , -cxdb (SPP)
  - Anlegen von **`.xdbrc`** bzw. **`.cxdbinit !`**
- ❏ In der Implementierungsphase:
  - Abfrage aller(!) Rückgabewerte und Ausgabe der genauer Ortsangabe, wenn der Rückgabewert nicht der ist, der er sein müßte.
  - Ständige Kontrollausgaben über den Zustand der einzelnen Prozesse.
- ❏ Modularisierung: Einsatz von Komponenten, die vorher genau mit Testdaten geprüft wurden.

Multi-Window-Debugging unter PVM3



Prozeß-Verwaltung I

❏ Jeder gestartete Prozeß muß sich bei der PVM-Maschine anmelden

```
mytid = pvm_mytid() ;
if (mytid < 0) {
    fprintf(stderr,"kein pvmd3 vorhanden\n");
    pvm_exit();
    exit(1);
   }
```

❏ Jeder Prozeß, den man glaubt gestartet zu haben, sollte überprüft werden

```
sprocesses = pvm_spawn("slave",(char**)0,PvmTaskDefault, "",
                        nprocesses, prtids);
if (sprocesses < nprocesses) {
    for (i = 0; i < nprocesses; i++){
        if (pvm_pstat(prtids[i])!= PvmOk)
           fprintf(stderr,"Prozess %d nicht gestartet\n",i);
        else pvm_kill(prtids[i]);
        }
    pvm_exit(); exit(1);
   }
```

Prozeß-Verwaltung II

❑ Der "Vaterprozeß" sollte erkennen können, wenn seine Söhne terminieren.

```
/* Dies ist der Sohnprozess */
#define ICH_HOEHRE_AUF 99
parent = pvm_parent();
pvm_notify(PvmTaskExit, ICH_HOERE_AUF, 1, &parent);
...
pvm_exit();


/* Dies ist der Vaterprozess */
#define ICH_HOERE_AUF 99
while ((bufid = pvm_nrecv(-1,ICH_HOEHRE_AUF)) > 0) {
       pvm_bufinfo(bufid, &bytes, &msgtag, &tid);
       printf("Prozess %x ist terminiert\n",tid);
      }
```


Prozeß-Verwaltung III

❑ Prozeßgruppen sollten mit einem für die Gruppen eindeutigen Namen gebildet werden. (~demo/pvm3/sc_spawn/sc_spawn.c)

```
/* Dies ist der Vaterprozess */
char groupname[256];
mytid = pvm_mytid();
sprintf(groupname,"group1%x",mytid);
pvm_joingroup(groupname);
pvm_barrier(groupname,pvm_gsize(groupname));
pvm_bcast(groupname, msgtag);
pvm_lvgroup(groupname);

/* Dies ist der Sohnprozess*/
char groupname[256];
parent= pvm_parent();
sprintf(groupname,"group1%x",parent);
pvm_joingroup(groupname);
pvm_barrier(groupname,pvm_gsize(groupname));
pvm_lvgroup(groupname);
```

❑ **Achtung: Sind alle Prozesse, die in der Gruppe sein sollten, schon darin?**

Prozeß-Start im Subcomplex (Convex SPP)

❏ Wieviele Prozessoren sind im Subcomplex?
(~demo/pvm3/sc_spawn/sc_spawn.c)

```
int get_num_procs()
{
 cnx_is_target_data_t target;
 cnx_is_thread_location_info_data_t     thread_location_info;
 cnx_is_sc_basic_info_data_t            sc_info;
 int entries, sc_id, pid;
 cnx_tid_t tid;

 pid = getpid(); tid = cnx_thread_self();
 cnx_sysinfo_target_thread(&target,pid,tid);
 entries = cnx_sysinfo (CNX_IS_THREAD_LOCATION_INFO,
                       &target, &thread_location_info, 1 ,
                       CNX_IS_THREAD_LOCATION_INFO_COUNT,0);
 cnx_sysinfo_target_subcomplex(&target,thread_location_info.scid);
 cnx_sysinfo (CNX_IS_SC_BASIC_INFO, &target, &sc_info, 1 ,
              CNX_IS_SC_BASIC_INFO_COUNT,0);
 return(sc_info.cpu_count);
}
```

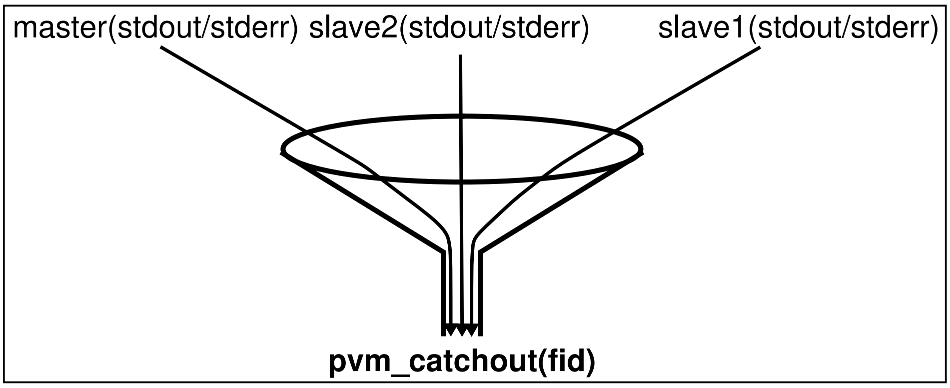

Ein/Ausgabe

❏ Die Ausgabe (**`stdout/stderr`**) des Masters wird auf stdout/stderr ausgegeben

❏ Die Ausgabe (stdout/st**`derr`**) eines Prozesses erfolgt

- lokal in die Datei /**`tmp/pvml.uid[.sc] oder`**
- in die Datei, die vom Parent-Process mit pvm_catchout angegeben wurde.
  (~demo/pvm3/sc_spawn/sc_spawn.c)

❏ **Die Ausgaben erfolgt nicht immer in der richtigen zeitlichen Reihenfolge !!**



Botschaftenaustausch I

❏ Unterschiedliche Architektur ➡ Unterschiedliche Zahlendarstellung

pvm_(u)pk.. - Funktion mit Angabe des Datentyps ermöglicht eine Konvertierung der Zahlendarstellung beim Botschaftenaustausch.

`pvm_initsend(PvmDataDefault)`: Konvertierung bei heterogener Architektur

`pvm_initsend(PvmDataRaw)`: keinerlei Konvertierung der Zahlendarstellung

❏ Verdoppelung der Datenmenge beim Versenden und Empfangen:

- Kopieren der Daten in den Sendepuffer (pvm_pk..)
- Kopieren der Daten aus dem Empfangspuffer (pvm_upk..)

Im CIP-Cluster darf ein Benutzerprozeß maximal 64 MB an Daten halten. Daher ist das Versenden einer 32 MBytes Matrix mit einer send-Operation nicht möglich.

➡ Aufteilen in kleiner Pakete, die hintereinander verschickt werden.

❏ Auf Shared Memory Systemen ist der Puffer-Bereich limitert:

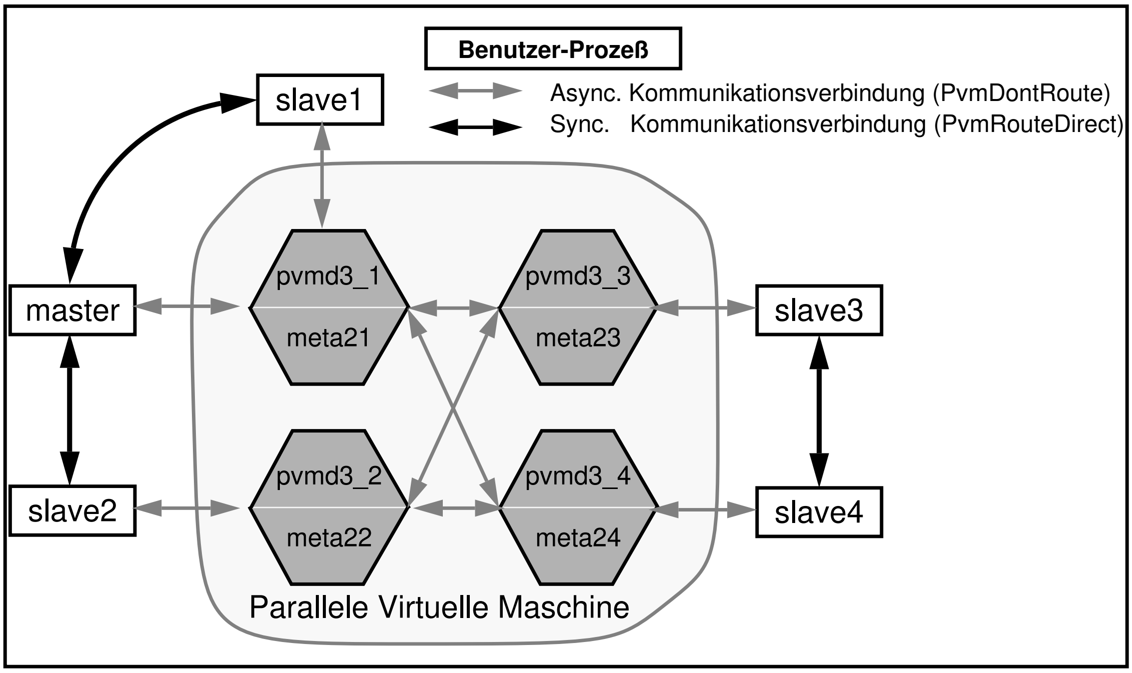
```
libpvm [t4001a]: da_new: shared buff full
libpvm [t4001a]: i_free: scribbled in 0x817728[-8]
```

➡ Verklemmung der Applikation möglich

```
setenv PVMBUFSIZE 10485760
```


Botschaftenaustausch II

❏ Asynchrone und synchrone Verbindung (im Cluster)

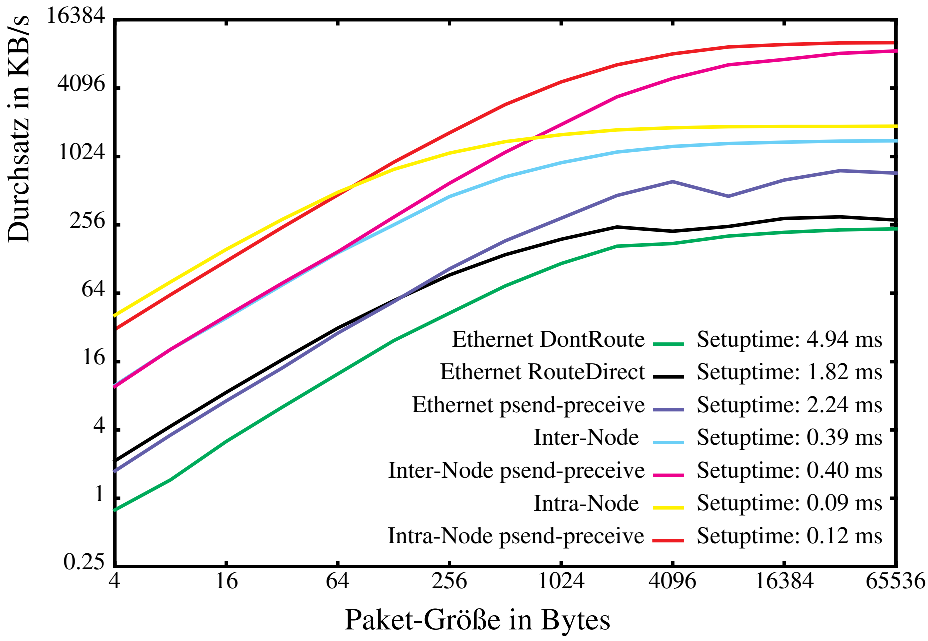


Botschaftenaustausch III

❏ Senden ohne Verpacken ist möglich und effizient!

- `pvm_psend(tid, tag, buf, len, datatype);`
- `pvm_preceive(tid,tag,buf,len,datatype,atid,atag,alen)`

PVM-Bandbreite im HP-Cluster / SPP



Strukturierung von parallelen Programmen I

❑ Kapselung von zusammengehörenden Daten in Strukturen

❑ Funktionen zum Senden/Empfangen von Strukturen

```
#define MATRIX 9999
typedef struct {
                 double *feld;
                 int zeilen, spalten;
                } matrix_typ;
...
int sende_matrix(matrix,tid)
int *tid;
matrix_typ *matrix;
{
 pvm_initsend(DataDefault);
 pvm_pkint(&(matrix->zeilen),1,1);
 pvm_pkint(&(matrix->spalten),1,1);
 pvm_pkdouble(matrix->feld,matrix->zeilen*matrix->spalten,1);
 return(pvm_send(*tid,MATRIX));
}
```


Strukturierung von parallelen Programmen II

```
matrix_typ empfange_matrix(tid)
int tid;
{
 matrix_typ matrix;
 int elements;

 pvm_recv(tid,MATRIX);
 pvm_upkint(&matrix.zeilen,1,1);
 pvm_upkint(&matrix.spalten,1,1);
 elements = matrix.zeilen*matrix.spalten;
 matrix.feld = (double *)malloc(elements*sizeof(double));
 pvm_upkdouble(matrix.feld,elements,1);
 return(matrix);
}
```

Strukturierung von parallelen Programmen III

❑ Vorsicht beim Byte-weisen Versenden von Strukturen (keine Konvertierung !)

```
#define STRUKTUR 8888
typedef struct {
                 int    izahl;
                 double dzahl;
                } strukturtyp;

int sende_struktur(struktur,tid)
stuktur_typ *struktur;
int tid;
{
 pvm_initsend(PvmDataDefault);
 pvm_pkbyte((char *)struktur,sizeof(strukturtyp),1);
 return(pvm_send(tid,STRUKTUR));
}
```

Vorteile des Byte-weisen Versendens von Strukturen:

- Sehr effizient, da wenige "Pack"-Operationen erfolgen
- Sende/Empfangs-Funktion ist unabhängig von der Strukturdefinition.

Strukturierung von parallelen Programmen IV

❑ Der Sender darf nicht terminieren bevor alle seine Botschaften gelesen wurden!

```
master()
{
      spawn(slaves);
      send_work_to_slave;
      receive(results);
      barrier();
}

slave()
{
      receive(task);
      work;
      send(result);
      barrier();
}
```

Verwendung von #include-Files

- Definition von mehrfach verwendeten Datentypen in #include-Files

```
/* structs.h */
#define STRUKTUR 8888
typedef struct {
                  int    izahl;
                  double dzahl;
                 } strukturtyp;
```

- Aufbau eines Makefile.aimk

```
# makefile for master-slave-example
PVM_LIB = -L$(PVM_ROOT)/lib/$(PVM_ARCH) -lgpvm3 -lpvm3
XDIR = ../../bin/$(PVM_ARCH)
VPATH= ..
COMPILE = $(CC) $(CFLAGS) -o $@ $< $(PVM_LIB) -lm ; cp $@ $(XDIR)
all: master slave
master: master.c structs.h
      $(COMPILE
slave: slave.c structs.h
      $(COMPILE)
```


Batch-Betrieb auf dem SPP-System

- Es stehen i.A. 2 Subcomplexe für parallele Jobs zur Verfügung(gsm8, gsm32)
- Die nqs-queues lauten **gsm8** und **gsm32**.
- Alle Batch-Aufträge werden auf einen der Prozessoren im Subcomplex gelenkt.
  ➡ Der "master"-Prozeß muß (!) als letzter Prozeß terminieren.
- Absetzen eines Auftrags: `qsub batchfile`
- Abfrage eines Jobs in der Warteschlange: `qstat -a gsm8`
- Abfrage eines laufenden Jobs: `qstat -a gsm8_b`
- Abbruch eine Batch-Jobs: `qmgr abort request REQUEST ID`

Aufbau eines PVM3-Batch-files

```
#!/bin/csh

# @$-q gsm32
# @$
# Löschen von "Altlasten"
/usr/lib/pvm/pvm_clean
# Starten der Virtuellen Maschine
/usr/lib/pvm/pvmd ~/pvm3/hostfile &
sleep 20
cd ~/pvm3/sc_spawn

# Starten des Group-Servers
pvmgs &
sleep 10
# Starten der Applikation
sc_spawn
# Terminieren verbliebender Prozesse
echo reset | /usr/lib/pvm/pvm
sleep 10
# Terminierung der Virtuellen Maschine
echo halt | /usr/lib/pvm/pvm
# Aufräumen
/usr/lib/pvm/pvm_clean ; rm core
```


Eigenschaften von MP-Bibliotheken

❏ Statische Prozeßverteilung (z.B. durch Eintrag in eine Konfigurationsdatei)

- Vorteile:
  - Effizienter Broadcast durch Aufbau einer Baumstruktur (O(log n))
  - Keine zusätzlichen Deamons erforderlich
- Nachteil:
  - Unflexibel
  - Nachstarten von Prozessen nicht möglich
  - Prozeßstart (meist über remsh) sehr zeitaufwending, da Authentisierung notwendig.
  - Keine gesichterte asynchrone Kommunikation möglich, da kein lokaler Daemon vorhanden ist, der die Botschaft zwischenspeichert und danach für eine sichere Übertragung sorgt.
- Vertreter:
  - TCGMSG (Argonne National Laboratory)

Eigenschaften von MP-Bibliotheken

❏ Dynamische Prozeßverteilung

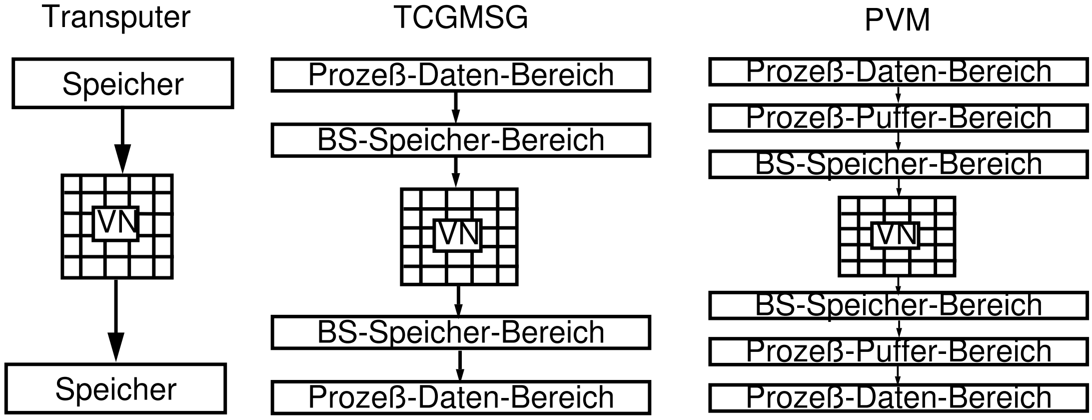
- Vorteile:
  - Flexibel
  - Prozeßkontrolle über einen Satz von Deamons möglich
  - Startupzeit für Prozesse sehr gering (1 Kommunikation + 1 fork())
  - Gesicherte Asynchrone Kommunikation möglich
  - Fehlererkennung/Fehlermeldung/Fehlertoleranzmaßnahmen möglich
- Nachteile:
  - Meist zusätzliche Deamons erforderlich
  - Aufwendiger Braodcast (O(n))
- Vertreter:
  - PVM (Oak Ridge National Laboratory)
  - Message Passing Interface (MPI) U.S. Department of Energy
- **Informationen zu PVM + MPI sind mit dem Programm xnetlib erhältlich !**

Eigenschaften von MP-Bibliotheken

❏ Unterstützung heterogener Systeme

– Automatische Konvertierung der Datenformate (PVM, TCGMSG)
z.B. Convex Native Floating Point ⟺ IEEE
Hinweis: CS-Tools der Firma MEIKO benötigen expilizite xdr-Kodierung

❏ Unterstützung synchroner Kommunikation mit Daten eines einzigen Typs ohne Kodierung/Dekodierung und Zwischenpufferung möglich



Eigenschaften von MP-Bibliotheken

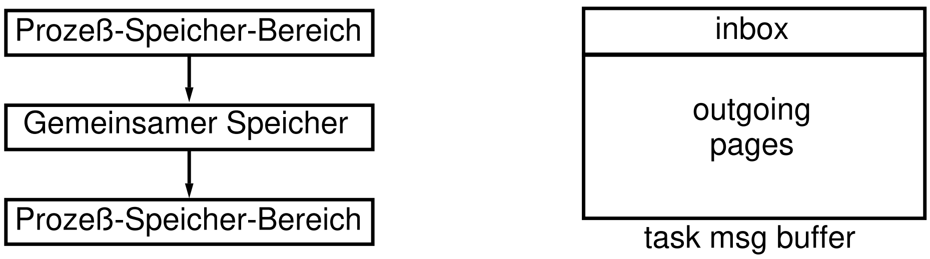
❏ Unterstützung von asynchroner Kommunikation

– Vorteil: Teilweises Verbergen von Lastungleichgewicht (Sender muß nicht auf Empfänger warten, der Empfänger jedoch auf den Sender.)

❏ Unterstützung von Shared Memory Architekturen

– Vorteil: Geringe Setupzeit und hoher Durchsatz durch Umgehung des Betriebssystems

– Beispiel: TCGMSG auf KSR2und Sequent Symmetry
PVM3 auf Convex SPP und KSR2



Schlußbemerkung

❏ Programmierung paralleler Systeme ist nicht einfach.

❏ Strukturierung bewirkt eine Fehlerreduzierung.

❏ Fehlerabfrage bewirkt eine Fehlerlokalisation.

❏ Fehlerquelle Nr. 1 sind Zeiger-Fehler

"Gehört zu jedem Zeiger auch ein Speicherbereich?"

❏ Fehlerquelle Nr. 2 sind Verklemmungen

"Können zur Laufzeit unauflösbare Abhängigkeiten entstehen?"

❏ ...

❏ Fehlerquelle Nr.x sind Systemfehler

☞ ausführliche Fehlerbeschreibung an: **`problems-spp@rrze`**

Viel Erfolg !

Übungsaufgabe zu PVM3

- ❏ Schreiben Sie ein PVM-Programm zur parallelen Glättung eines 2-D-Gitters.
- ❏ Sequentielles Beispiel: ~demo/Beispiel/grid.c
- ❏ Inititalisierung:

```
void init_grid(grid,dim)
double *grid;
int dim;
{
    int i,x,y;
    for (x = 0 ; x < dim ; x++)
        for(y = 0 ; y < dim ; y++)
            GRID(x,y) = 0.0;

    for (i = 0 ; i < dim ; i++)
        {
            GRID(0,i) = (1.0/dim)*(dim-1-i);
            GRID(i,(dim-1)) = (1.0/dim)*i ;
            GRID(i,0) = (1.0/dim)*(dim-1-i);
            GRID((dim-1),i) = (1.0/dim)*i;
        }
}
```


Übungsaufgabe zu PVM3

- ❏ Glättung

```
void smooth_grid(grid,dim,iterations)
double *grid;
int dim, iterations;
{
    int i,x,y;
    for (i = 0 ; i < iterations ; i++)
        {
            for (y = 1 ; y < dim-1 ; y++)
                for(x = 1 ; x < dim-1 ; x++)
                 GRID(x,y) =(GRID(x-1,y)+GRID(x+1,y)+GRID(x,y+1)+GRID(x,y-1)
                                      /4.0;
            print_grid(grid,dim,i);
        }
}
```

- ❏ Abgabe der Lösung erfolgt durch eine email an problems-spp@rrze mit Angabe des Verzeichnisses auf dem SPP-System bis zum 31.7.94.
    - Parallelisierung: Partitionierung des Gitters (z.B. in Streifen)
    - Das Lösungsprogramm soll mit einer beliebigen Prozessorzahl ablaufen können. Der Aufwand beträgt ca. 250 Zeilen kommentierter Code.

xpvm-Console