Programmierung paralleler Systeme
Message Passing mit PVM

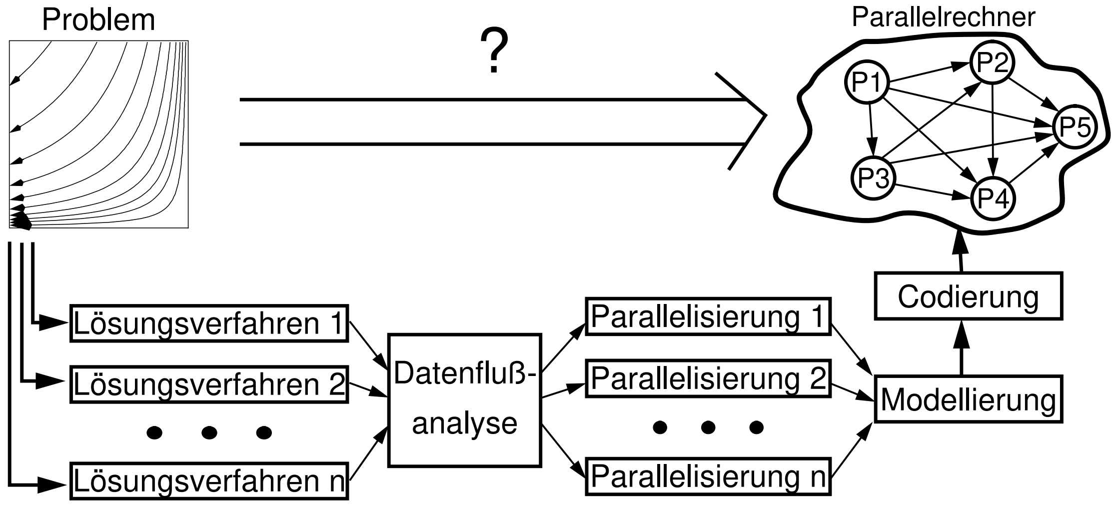
Frank Bellosa

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

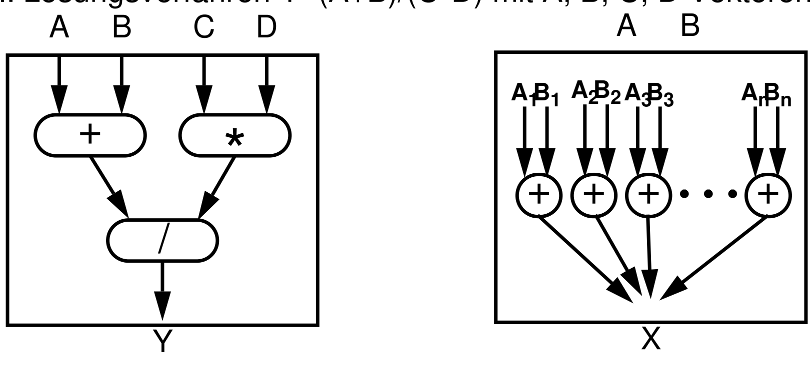
Vom Problem zum Programm



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

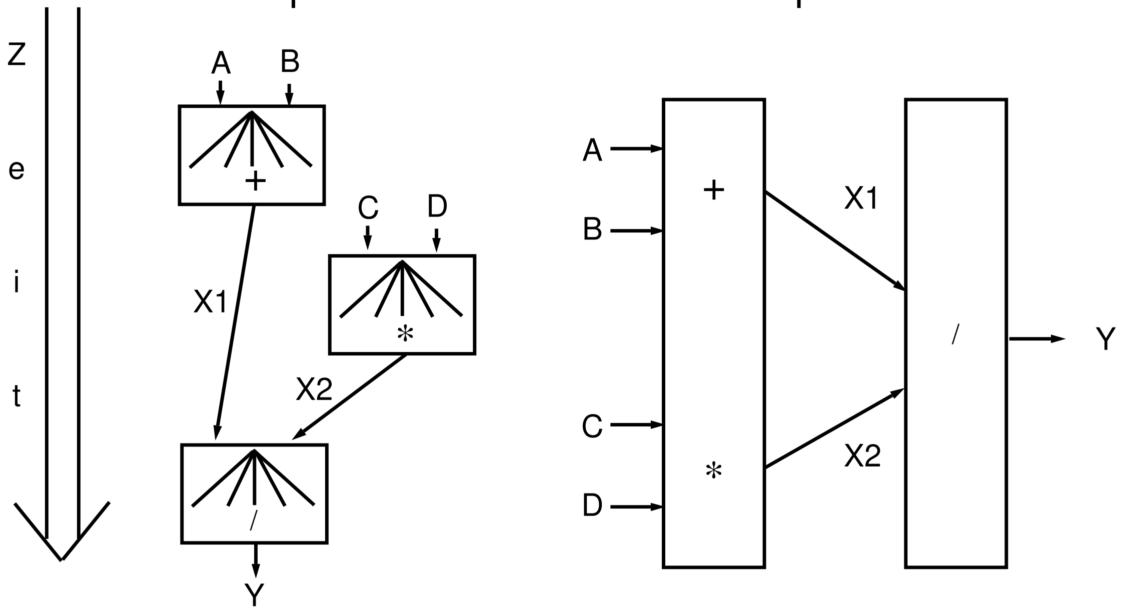


Vom Problem zum Programm

Parallelisierung

Datenparallelität

Prozeßparallelität



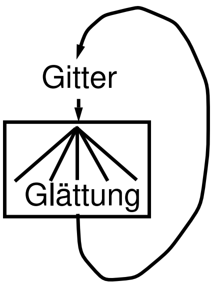
Vom Problem zum Programm

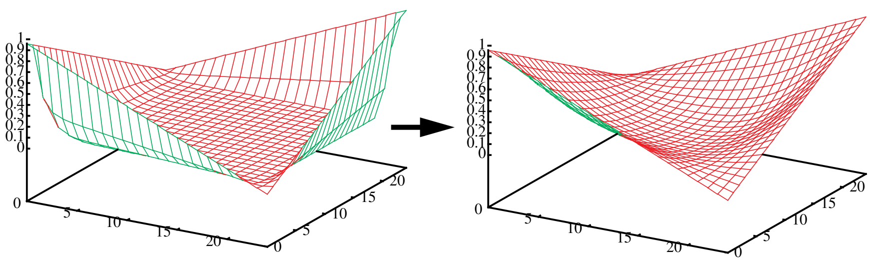
Beispiel

❑ Glättung eine Gitters

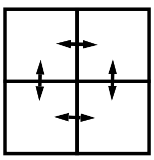
$$\text{gitter}[x, y] = (\text{gitter}[x+1, y] + \text{gitter}[x-1, y] + \text{gitter}[x, y-1] + \text{gitter}[x, y+1]) / 4$$

– Datenflußanalyse:





– Parallelisierung



– Gebietspartitionierung durch Bipartitionierung

↔ = Kooperation = Koordination + Kommunikation

Programmiermodelle

Parallele Prozesse + Message Passing

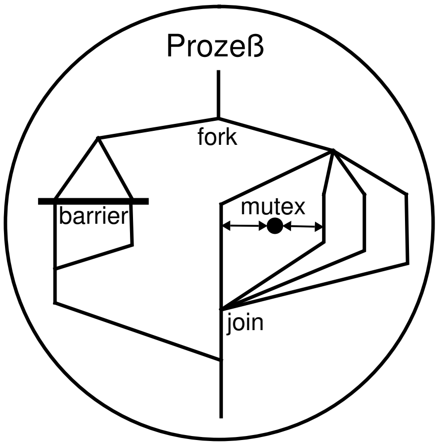
send(Botschaft, Empfänger-Adresse)

Botschaft = receive(Sender-Adresse)

Parallele Prozesse + Shared Memory

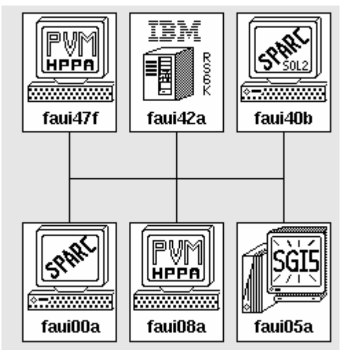
shm_id = shm_get(Schlüssel, Größe, Zugriffsrechte)	
ptr = shm_at(shm_id)	shm_dt(prt)
semaphore = sema_create(Schlüssel, Zugriffsrechte)	
wait(semaphore)	signal(semaphore)

1 Prozeß + Parallele Threads + Shared Memory

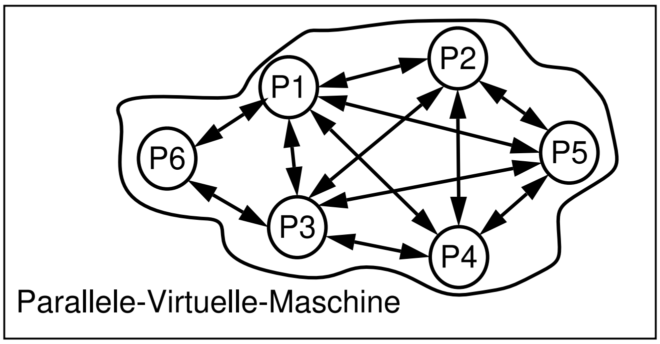


Einführung in PVM3

❑ 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

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

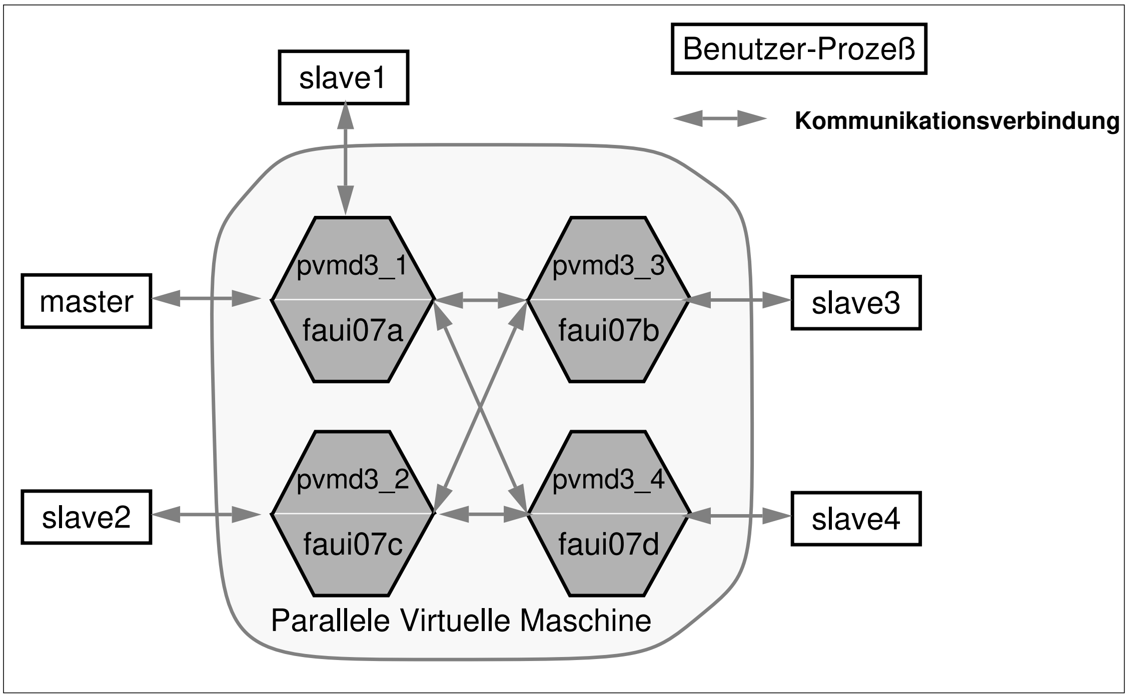
```
pvmd hostfile &
```

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.

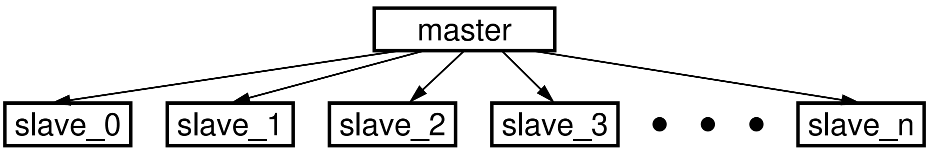
  Authentisierungsfile: `-rw------- /tmp/pvmd.10407`
  Logfile für stderr: `-rw------- /tmp/pvml.10407`

  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)
```

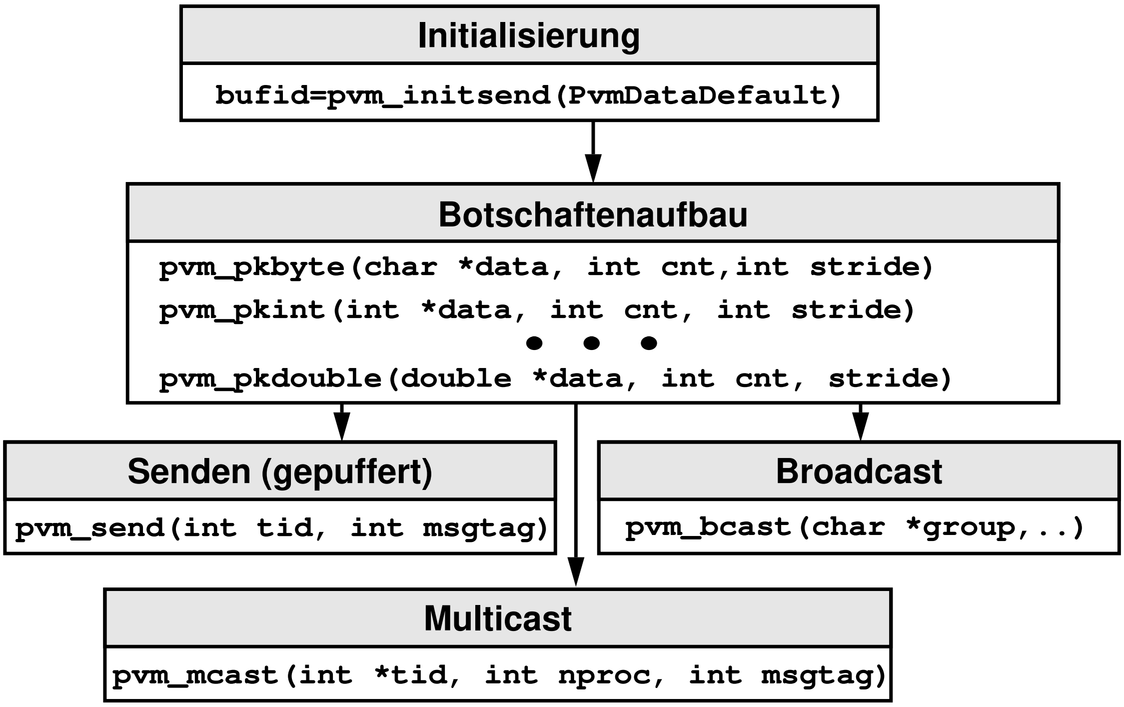
Bildung von Prozeßgruppen

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

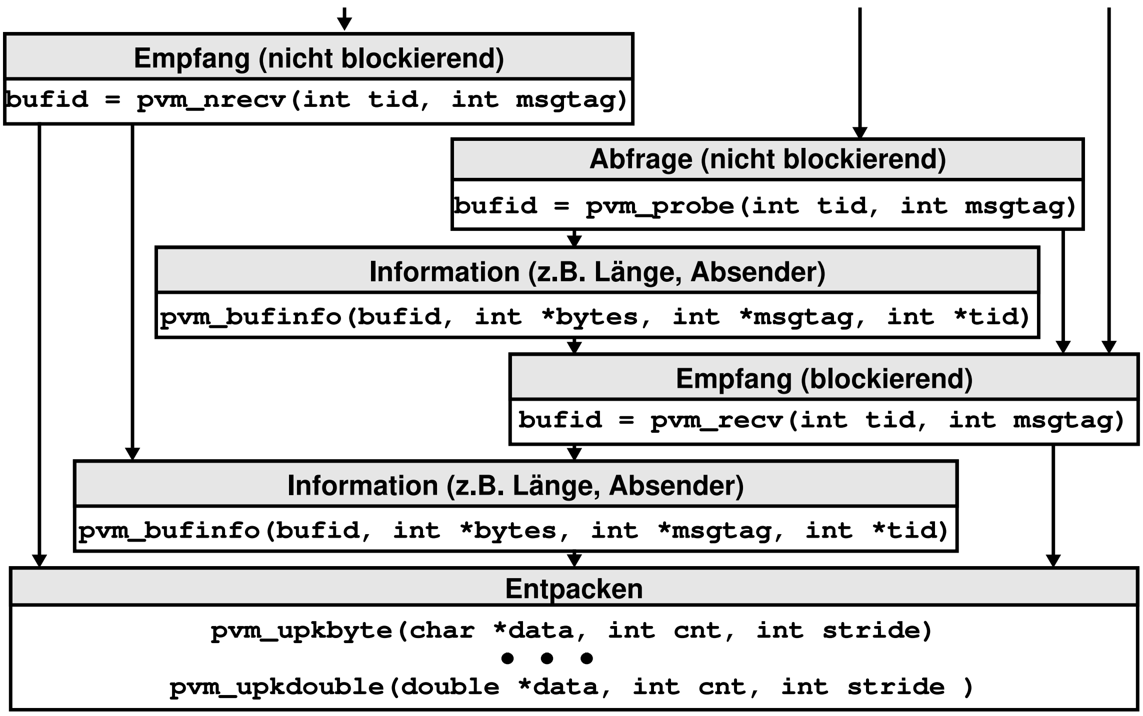
Terminieren von Prozessen

```
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:

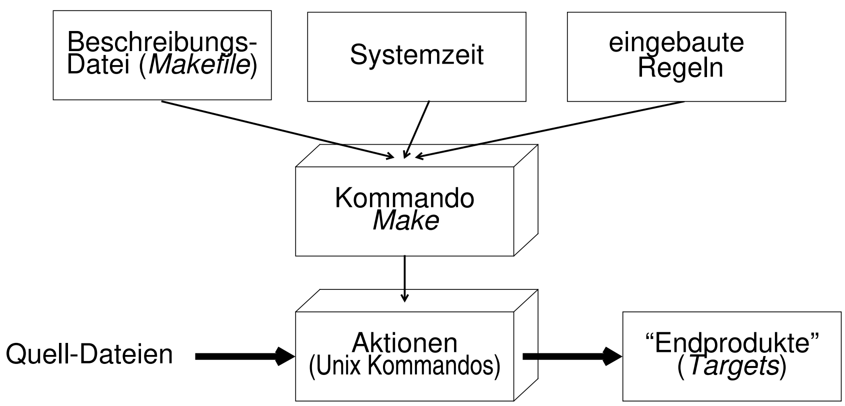
```
pvm3/bin/
           CSPP/
               pvmgs -> /usr/convex/pvm/lib/CSPP/pvmgs
           HPPA/
                grid
               pvmgs -> /local/pvm3/lib/HPPA/pvmgs
pvm3/bsp1/
            CSPP/
            HPPA/grid.o
            Makefile.aimk
            grid.c
```


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 master-slave-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