Entwicklung verteilter Anwendungen mit PVM

Peter Junglas

2. April 1993

Inhaltsverzeichnis

1 Einführung

- PVM = Parallel Virtual Machine
- Zielsetzung:
  - Zusammenschalten mehrerer (möglicherweise verschiedener) Workstations als "billigem Parallelrechner",
  - Verknüpfung von z.B. Graphik-Workstations, Vektorrechnern und Parallelrechnern zu einem großen Parallelrechner, um unterschiedliche Resourcen für ein Programm zusammenzubringen,
  - Portabilität von parallelen Programmen.
- Kooperation von:

Adam Beguelin	Carnegie Mellon University
Jack Dongarra	University of Tennessee
Al Geist	Oak Ridge National Laboratory
Weicheng Jiang	University of Tennessee
Robert Manchek	University of Tennessee
Vaidy Sunderam	Emory University

- Public Domain, bei der Netlib verfügbar
- Versionen für viele Workstation und Vektorrechner dabei, außerdem für folgende Parallelrechner :

  Intel iPSC860, Intel Paragon, TMC CM-5
  SMP-Systeme von Sun und SGI.

- wird von verschiedenen Herstellern für ihre Parallelrechner unterstützt, u.a. Convex, Cray, IBM, Intel, Parsytec, SGI, Thinking Machines.

1.1 Beispiele für Anwendungen

- Struktur-Analyse mit FEM
- Strömungsmechanik, Flugzeug-Design
- Molekül-Dynamik
- neuronale Netzwerke
- Simulation von Kernkraftwerken
- Computer-Tomographie
- Magneto-Hydrodynamik
- Hochenergiephysik
- Klimaforschung
- Implementierung verteilter Verfahren aus der linearen Algebra
- u.v.a.

1.2 System-Überblick

- Programm-Modell:
  - Menge von wechselwirkenden Komponenten (Tasks"), gleichartig oder verschieden, jede kann direkt mit jeder anderen kommunizieren
  - Verteilung von Tasks auf Prozessoren: transparent oder programmgesteuert
- Struktur:
  - Kontroll-Dämon (pvmd3)

    je einer auf jeder beteiligten Maschine
    Erzeugen und Vernichten von Tasks
    Weiterleiten von Messages
  - Bibliothek (libpvm3.a)

    Task Management
    Kommunikation
    Information
    Synchronisation
  - dynamische Prozeßgruppen extra

    Gruppenserver (pvmgs)
    Routinen zur Gruppenverwaltung und kollektive Operationen (libgpvm3.a)

1.3 Hilfssysteme für PVM

Xab:	Profiling und Debugging
Hence:	graphische Programmierung
XPVM	X11-Oberfläche und Profiling
DBPVM	komfortabler Debugger
DoPVM:	Programmierung mit verteilten Objekten
PVM++:	Objektorientiertes Toolkit für PVM
Qmanager:	automatische Lastverteilung
FT:	fehlertolerantes PVM

1.4 Alternative Systeme

- Express:

  ähnliche Struktur, aber auch "höhere" Befehle
  bessere Tools (Debugger, div. Profiler)
  kommerziell (ParaSoft, Vertrieb: Genias)

- P4:

  unterstützt Message Passing und Shared Memory
  Weiterentwicklung von PARMACS-Makros von Argonne
  Abspaltung: PARMACS kommerziell, GMD

- Linda:

  von David Gelernter et. al. (Yale) entwickelt
  anderes Konzept: Tupelspace (kein Message Passing)
  Compiler-Erweiterung (Macros) für C, Eiffel, PVM
  einige kommerzielle Anbieter und PD-Implementationen

- MPI:

  Quasi-Standard, April 94 verabschiedet
  Zielsetzung: Portabilität und Effizienz
  einige neuartige Konzepte, etwa für Bibliotheken
  MPP-Hersteller implementieren gerade
  erste PD-Versionen für Workstation-Cluster

2 Ein einfaches Beispielprogramm

2.1 Master-Slave-Modell

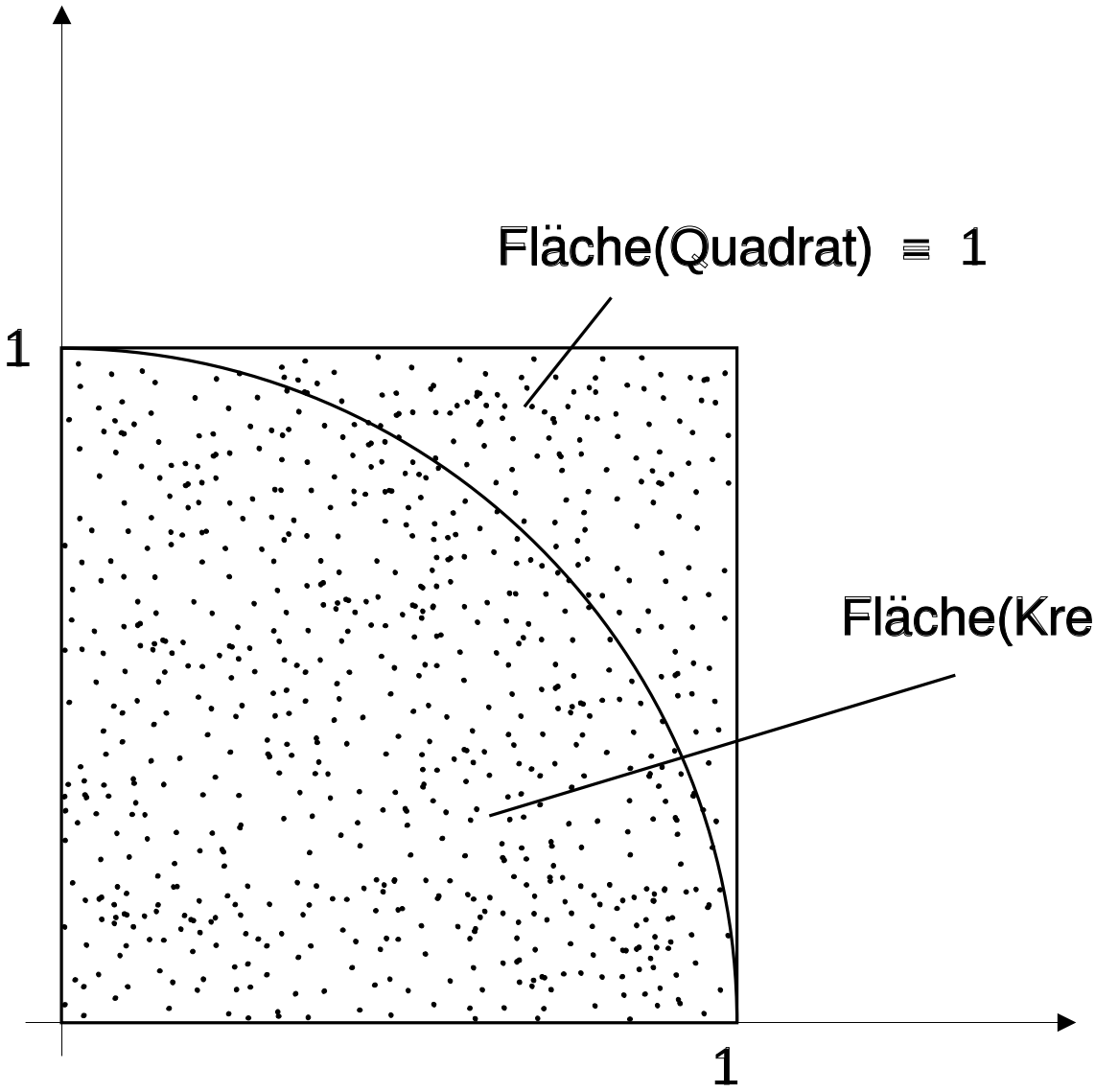
- Master:

  beim PVM-System anmelden
  Slaves starten
  Arbeit an die Slaves verteilen
  Teil-Ergebnisse der Slaves einsammeln
  Gesamt-Ergebnis berechnen und ausgeben

- Slave:

  beim PVM-System anmelden
  Arbeit vom Master empfangen
  Teil-Ergebnis berechnen
  Resultat an Master schicken
  beim PVM-System abmelden

2.2 Berechnung von Pi mit der "Montecarlo-Methode"



- Teilergebnis berechnen:

  bestimme N gleichverteilte Zufallspunkte im Quadrat
  [0, 1] x [0, 1]
  NT = Anzahl von Zufallspunkten im Viertelkreis
  $\pi \approx 4 * NT/N$

3 Arbeiten mit PVM

3.1 Installation

- Sourcen besorgen

  ftp von netlib.ornl.gov, Directory ~/pvm3
  ftp von ftp.rz.tu-harburg.de, Directory source/parallel
  (nicht das Manual ug.ps vergessen!)

- Auspacken (z.B. das tar-File)

  uncompress pvm3.3.tar.Z
  tar xvf pvm3.3.tar

- Übersetzen

  Environment-Variable PVM_ROOT setzen
  (am RZ: PVM_ROOT = /progsys/pvm3)
  make all"klappt fast immer

3.2 Vorbereitungen zur Benutzung von PVM

- Architektur PVM_ARCH der benutzten Maschinen feststellen (z.B. mit /progsys/pvm3/lib/pvmgetarch)

- Verzeichnisse anlegen:

  ~/pvm3/bin/$PVM_ARCH

- Umgebungsvariable setzen (am besten im Startup-Skript der Shell):

  setenv PVM_ROOT /progsys/pvm3
  setenv PVM_DPATH $PVM_ROOT/lib/pvmd
  setenv PVM_EXPORT DISPLAY

- Suchpfade ergänzen:

  setenv PATH "$PVM_ROOT/lib:$PATH"
  setenv MANPATH "$PVM_ROOT/man:$MANPATH"

3.3 Compilieren und Linken

- auf jeder beteiligten Architektur übersetzen
- benötigte Bibliotheken (in $PVM_ROOT/lib/$PVM_ARCH):

libpvm3.a	PVM3-Basisbibliothek
libfpvm3.a	Fortran-Interface zu PVM
libgpvm3.a	PVM-Gruppenfunktionen

- u.U. noch architekturspezifische Bibliotheken dazulinken
  (z.B. -lsun auf SGI)
- fertige Programme jeweils nach ~/pvm3/bin/$PVM_ARCH kopieren
- Unterstützung eines Makefiles für verschiedene Rechner: aimk

3.4 Definieren einer virtuellen Maschine

- Beschreibung der beteiligten Rechner im Hostfile
- einfaches Hostfile = Liste der Rechnernamen:

```
indi1.cip3s
sg02.cip3s
convex.rz
anton.rz
```

- mehrere Optionen hinter einem Hostnamen möglich:

lo=Userid	Angabe eines anderen Usernamens
so=pw	beim Starten wird das Login-Paßwort abgefragt
dx=Pfad	Pfad für den Dämon pvmd
ep=Pfad	Directory mit den Userprogrammen
wd=Pfad	Startdirectory für gespawnte Programme
sp=Wert	relative Geschwindigkeit eines Hosts

  Defaults:

dx	$PVM_DPATH oder $PVM_ROOT/lib/pvmd
ep	pvm3/bin/$PVM_ARCH:\ $PVM_ROOT/bin/$PVM_ARCH
sp	1000

- Optionen können für mehrere im Hostfile aufeinanderfolgende Maschinen gesetzt werden:

  * ep=bin/pvm lo=et42nn

- Optionen für Maschinen angeben, ohne dort gleich pvmd's zu starten:

  &convex lo=rztpj

3.5 Das Konsolprogramm pvm

- pvm [hostfile]

  verbindet sich mit der aktuellen virtuellen Maschine oder startet eine neue

- wichtige Kommandos:

help	Liste aller oder Erklärung eines Kommandos
add	Hinzufügen einer Maschine
delete	Entfernen einer Maschine
conf	Anzeige der Konfiguration der aktuellen Maschine
ps	Anzeige aller aktiven Benutzer-Prozesse
reset	Beenden aller PVM-User-Prozesse
spawn	Starten von PVM-Programm (s.u.)
quit	Verlassen von pvm (virtuelle Maschine läuft weiter)
halt	Beenden der virtuellen Maschine

- liest ~/.pvmrc-File

- Beispiel-Ausgaben:

  conf:

4 hosts, 1 data format

HOST	DTID	ARCH	MTU	SPEED
mufus	40000	HPPA	4096	1
anton.rz	80000	HPPA	4096	1
convex.rz	c0000	CNVX	4096	1
sg02.cip3s	100000	SGI	4096	1

  ps -a:

HOST	A.OUT	TID	PTID	FLAG
mufus	pi_s	44c7e	80285	0x04/c
anton.rz	pi	80285	(cons)	0x04/c
anton.rz	pi_s	80286	80285	0x04/c
sg02.cip3s	pi_s	100e28	80285	0x04/c

3.6 Starten von PVM-Programmen

- direkt von der Shell aus:

```
% lbpi 10000 100
```

  Ausgaben der Slaves landen im File /tmp/pvml.UID

- von der pvm-Konsole aus:

```
pvm> spawn lbpi 10000 100
```

- wichtige Optionen von spawn:

->	Ausgabe zur Konsole umlenken
->(file)	Ausgabe in File umlenken
->>(file)	Ausgabe an File anhängen
-?	Debugging einschalten
-@	Tracedaten erzeugen und zur Konsole schicken
-@(file)	Tracedaten erzeugen und in File schreiben

- von der pvm-Konsole aus Ereignisse zum Tracen auswählen:

```
pvm> trace [+|-] name
```

  alle Ereignisse auswählen mit '*'

4 Die PVM-Bibliothek im Überblick

4.1 Message Passing

pvm_send, pvm_mcast
pvm_recv, pvm_nrecv, pvm_trecv
pvm_probe
pvm_psend, pvm_precv
pvm_setmwid, pvm_getmwid, pvm_recvf

pvm_pk<TYPE>, pvm_upk<TYPE>
pvm_packf, pvm_unpackf

pvm_initsend, pvm_bufinfo
pvm_mkbuf, pvm_freebuf
pvm_getsbuf, pvm_getrbuf, pvm_setsbuf, pvm_setrbuf

Task Control
pvm_mytid, pvm_exit, pvm_spawn, pvm_kill
pvm_parent, pvm_pstat, pvm_tasks, pvm_tidtohost
pvm_sendsig

Group Library Functions
pvm_joingroup, pvm_lvgroup
pvm_getinst, pvm_gettid, pvm_gsize
pvm_barrier
pvm_bcast, pvm_gather, pvm_scatter
pvm_reduce

4.2 Virtual Machine Control

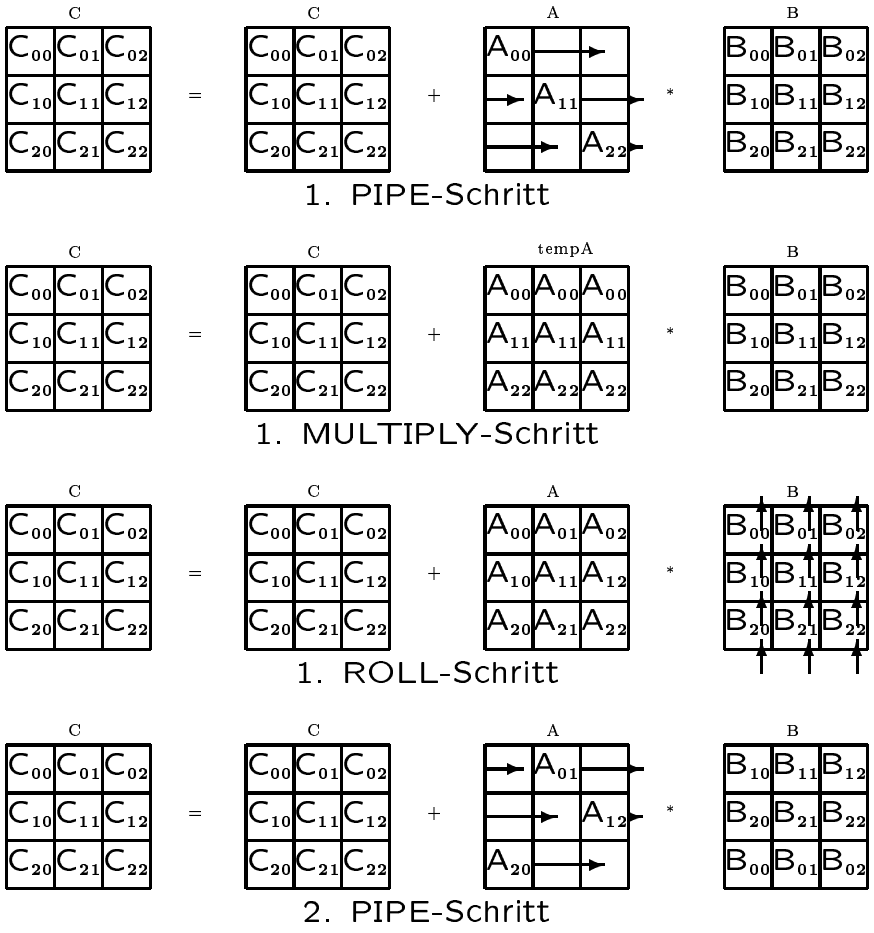
pvm_addhosts, pvm_delhosts, pvm_start_pvmd, pvm_halt
pvm_mstat, pvm_config, pvm_archcode
pvm_reg_hoster, pvm_reg_rm, pvm_reg_tasker

Miscellaneous
pvm_getopt, pvm_setopt, pvm_perror
pvm_catchout
pvm_notify
pvm_insert, pvm_lookup, pvm_delete
pvm_settmask, pvm_gettmask, pvm_hostsync

5 Matrix-Multiplikation – ein weiteres Beispiel

5.1 Der Pipe-Multiply-Roll-Algorithmus

- quadratisches Gitter von PxP Prozessoren"
- Zerlegung der quadratischen Matrizen in PxP Teilmatrizen
- Ausgangsteilmatrizen und Ergebnismatrix auf entsprechende Prozessoren verteilt
- 3 Phasen: Pipe, Multiply, Roll



5.2 Prozeß-Gruppen in PVM

- neues Feature seit PVM3.0: dynamische Prozeß-Gruppen
- Bestandteile:
  - eigene Bibliothek libgpvm3.a, basiert auf libpvm3.a
  - "Group Server" pvmgs, wird automatisch gestartet
  - Hilfsprogramm pvm_gstat zeigt Gruppenkonfigurationen
- Gruppe charakterisiert durch Namen, Mitglieder durch Instanz-Nummer
- Instanz-Nummern von 0 an aufsteigend, kleinste freie Nummer wird vergeben
- Jeder Prozeß (Gruppenmitglied oder nicht) darf jederzeit alle Gruppenbefehle ausführen außer pvm_lvgroup, pvm_barrier, pvm_reduce
- pvm_gstat (unter $PVM_ROOT/lib/$PVM_ARCH) zeigt aktuelle Gruppen, Mitglieder und Barrieren an:

```
group: gexamp, size: 3,  barrier_count 3, barrier_reached 2
tids:
0 0x42ef5       1 0x42ef8       2 0x42ef7
tids waiting on barrier:
0x42ef5 0x42ef8
```

5.3 Kollektive Operationen

- pvm_barrier:
  alle warten, bis die gewünschte Anzahl von Tasks (normalerweise die Gruppengröße) angekommen ist

- pvm_bcast:
  einer schickt die gleichen Daten an alle
  Achtung: empfangen wird ganz normal mit pvm_recv!

- pvm_scatter:
  einer schickt jeweils verschiedene Daten an alle

- pvm_gather:
  einer sammelt jeweils verschiedene Daten von allen ein

- pvm_reduce:
  eine globale Operation (Summe, Produkt, Minimum, Maximum) wird mit den Daten aller ausgeführt,
  das Ergebnis liegt bei einer ausgewählten Task

- Alle Operationen beziehen sich auf die aktuellen Gruppenmitglieder. Veränderungen der Gruppe während einer globalen Operation führen zu falschen Ergebnissen oder Deadlocks!

5.4 Bemerkungen zur Implementierung von matmul

- Output aller Tasks wird durch pvm_catchout nach STDOUT umgelenkt; er ist gepuffert, nicht synchronisiert.
- Wer ist der Master? pvm_parent (versagt bei spawn von der Konsole) oder Instanz 0 der Gruppe WORLD.
- pvm_barrier beim Startup nötig
  (sonst läuft der Master u.U. bis ins Pipe (gettid und send) voraus)
- Alternative Implementierung des Pipe:
  im Startup zeilenweise Gruppen bilden, dann broadcast
- Austausch im Roll in anderen Systemen (nicht PVM) u.U. problematisch
- Ohne beide Barriers im PrintTrace "Race Condition" für pvm_reduce

6 Debuggen und Profilen

6.1 Debuggen von PVM-Programmen

- Vorbereitungen:
  - in allen pvm_spawn-Aufrufen 3. Argument (flag) ersetzen durch flag | PvmTaskDebug"
  - mit Debug-Option (meist g") neu compilieren und linken
  - DISPLAY-Variable muß gesetzt sein, PVM_EXPORT muß DISPLAY und PATH enthalten (am besten in ∼/.pvmrc).
- von der Konsole starten mit

  pvm> spawn -? PROGRAMMNAME
- Falls sich das Programm nicht von der Konsole aus starten läßt:
  - DISPLAY-Variable in ∼/.cshrc explizit setzen !
  - erste PVM-Task explizit unter Debugger-Kontrolle starten
- beim Spawn wird für jede neue Task das Skript $PVM_ROOT/lib/debugger aufgerufen, das einen Debugger in einem eigenen Fenster startet.
- einige Debugger finden Source nicht, dort Sourcepfad explizit setzen z.B. im HP-UX-Debugger xdb:

  xdb> D pvm3/src/gexamp"

6.2 Generelle Vorgehensweise beim Debuggen

- zunächst, falls möglich, nur eine Task
  (serielle Fehler)
- dann 2 - 4 Tasks auf einer Maschine
  (Kommunikationsfehler wie mehrdeutige oder falsche Tags)
- danach 2 - 4 Tasks auf verschiedenen Maschinen
  (Timing-Fehler, schwer zu finden !)

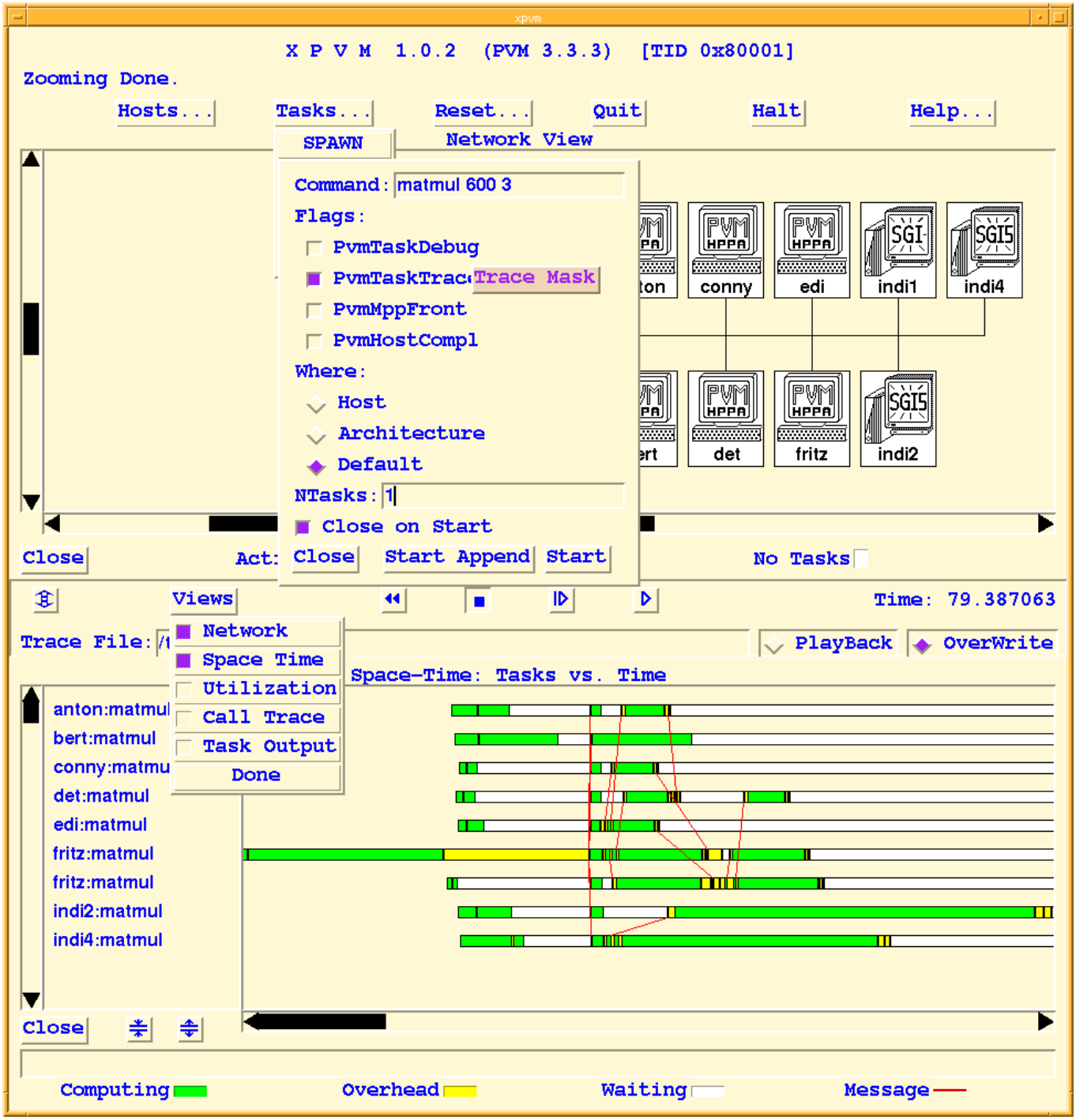
6.3 Laufzeitmessungen mit xpvm

- graphische Oberfläche zur Konsole:
  - Konfiguration der virtuellen Maschine incl. Graph
  - Starten von Tasks mit diversen Optionen
  - Beenden einzelner oder aller Prozesse
  - Senden von Signalen
  - Sammeln der Ausgaben
- Profiling-Tool:
  - ersetzt xab und Paragraph
  - Auswahl von Trace-Events
  - Synchronisation mit pvm_hostsync
  - einige Diagramme zur Darstellung der Trace-Daten:

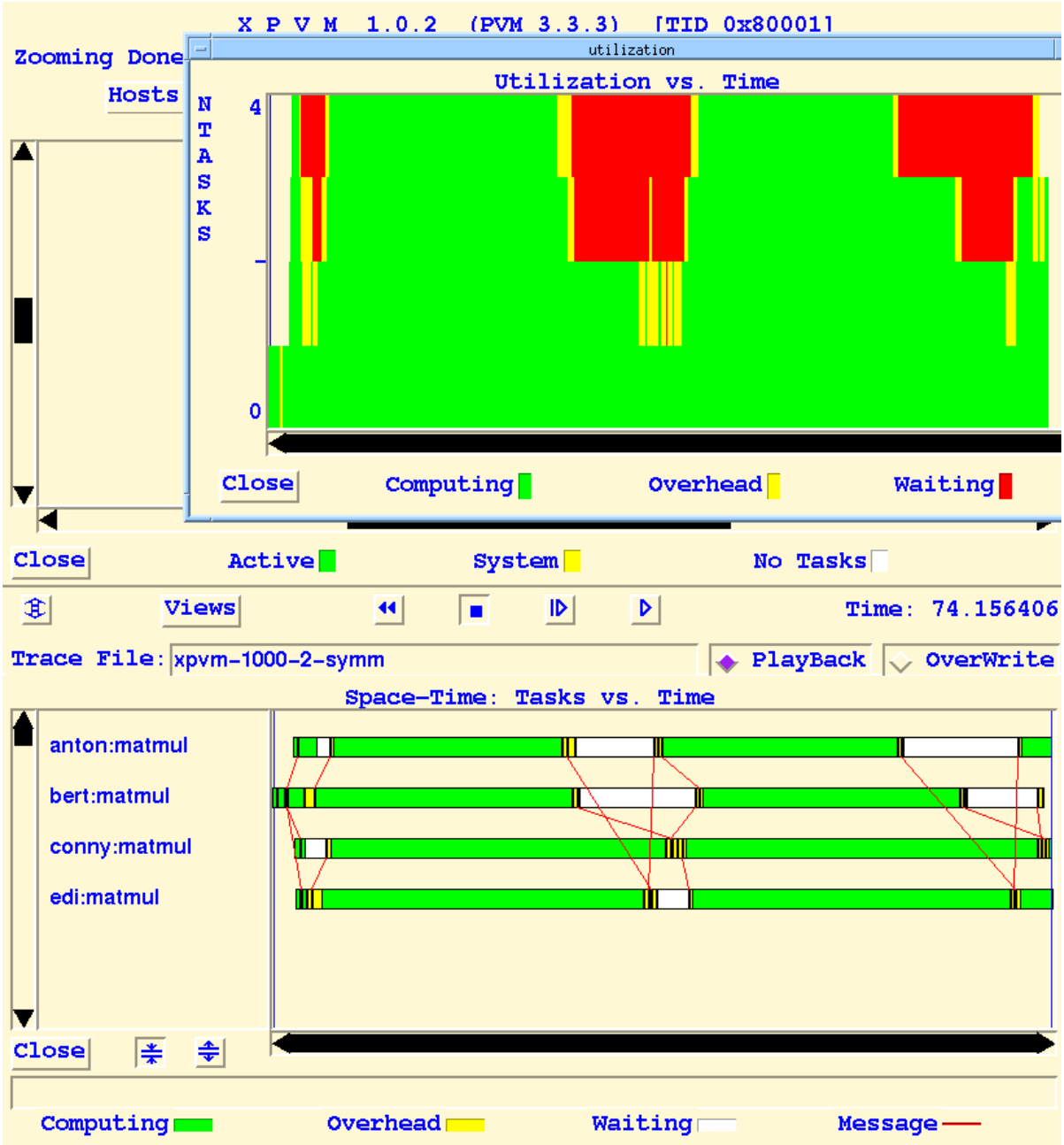
Call-Trace	PVM-Aufrufe pro Task als Text
Network	momentane Aktivität pro Prozessor
Space-Time	Aktivität und Nachrichten über die Zeit
Utilization	Auslastung über die Zeit

- noch Beta-Release:
  - Darstellung oft sehr langsam (Tcl/Tk-basiert)
  - weitere Diagramme geplant
  - z.Zt nur auf dem HP-Cluster installiert unter
    $PVM_ROOT/bin/$PVM_ARCH,
    braucht Umgebungsvariable
    setenv XPVM_ROOT $PVM_ROOT/xpvm

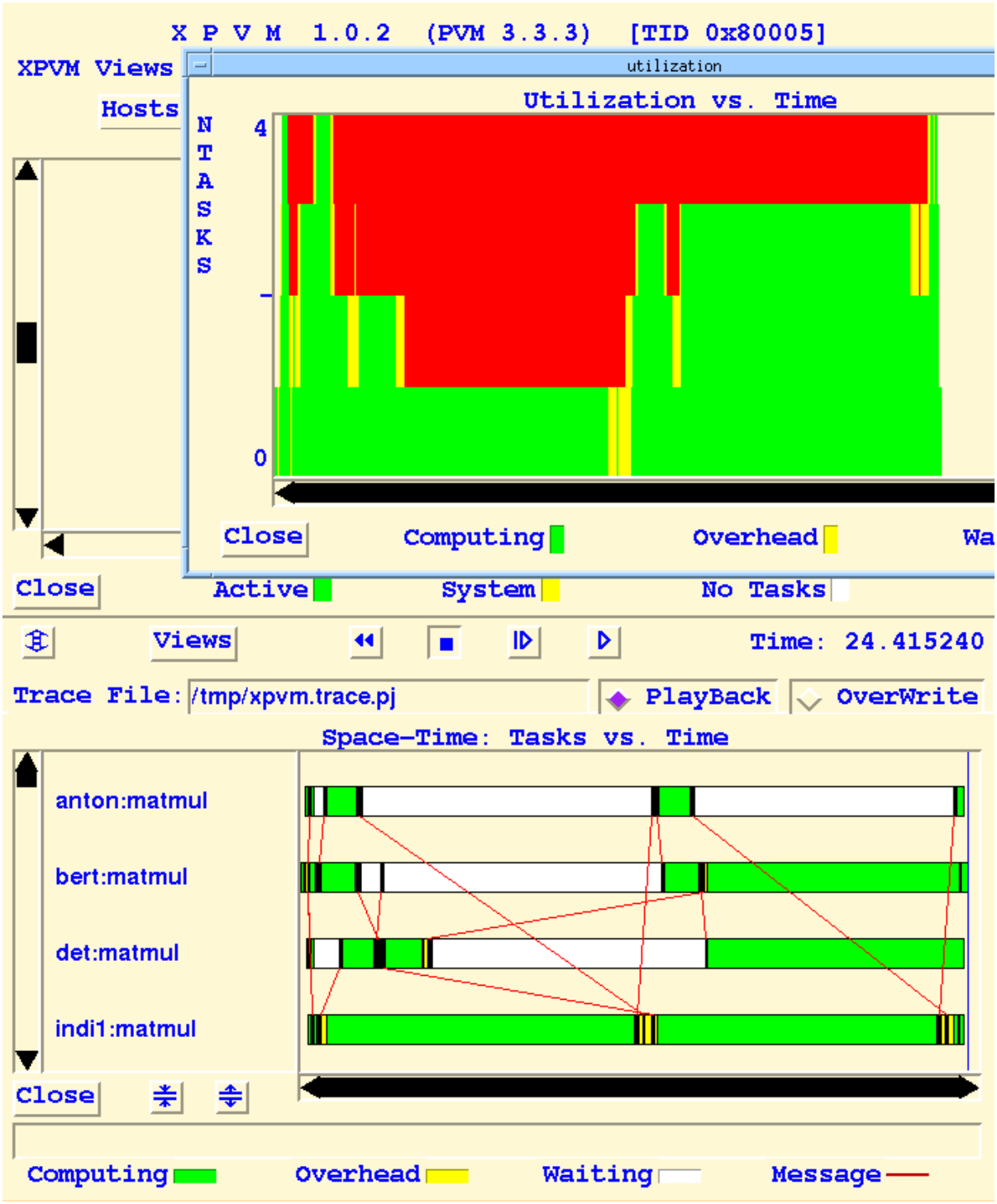
6.4 xpvm – Beispiel-Ausgabe



6.5 Laufzeitverhalten von matmul 1000x1000, 4Proz., symmetrisch

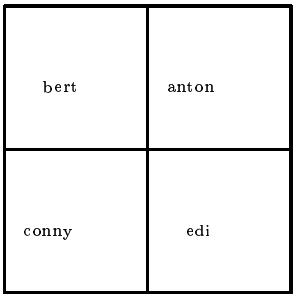


6.6 Laufzeitverhalten von matmul 500x500, 4Proz., unsymmetrisch



6.7 Laufzeit-Verhalten von matmul Anmerkungen

- Phasen des Algorithmus und Abbildung der Topologie auf die Prozessoren in beiden Beispielen gut erkennbar:



symmetrisch

unsymmetrisch

- Algorithmus hat relativ viel simultane Kommunikation, schlecht für Workstation-Cluster mit Bus-Netzwerk
- kein Load-Balancing → große Performance-Einbrüche bei verschieden schnellen Prozessoren
- "optimaler" Algorithmus zur Matrix-Multiplikation u.a. abhängig von Prozessoreigenschaften (Cache, Pipes,...), Netzwerktopologie und Routing-Verfahren
- Überlappung von Kommunikation und Berechnung wichtig

7 PVM-Interna

7.1 PVM-Dämon

- Master-Dämon:

  startet die anderen pvmd's
  verwaltet und synchronisiert die Host-Tabelle
  rekonfiguriert ggf. die Maschine
  lenkt I/O um nach /tmp/pvml.<UID>

- Menge von pvmd's eines Benutzers definiert seine "virtuelle Maschine"

- vermittelt Kommunikation:

  zwischen pvmd's über UDP, kein Routing
  übernimmt Kontrolle der Paketauslieferung
  zwischen pvmd und lokalen Tasks über TCP
  TCP direkt zwischen Tasks möglich

- entdeckt Ausfall eines Host:

  Symptom: keine Antwort auf Anfragen (time out)
  Host wird aus der Tabelle gestrichen
  Task-Anfragen bekommen Fehlercode PvmHostFail
  entsprechende Tasks werden nach pvm_notify benachrichtigt
  pvmd ohne Verbindung zum Master: räumt auf und stirbt

7.2 Task

- charakterisiert durch Task ID:
  - zerfällt in globalen und lokalen Anteil
  - globaler Anteil enthält hostnummer (global)
  - lokaler Anteil wird vom lokalen pvmd vergeben, z.B. UNIX-PID
  - TID eines pvmd: lokaler Teil = 0
- findet die Socket-Adresse ßeinespvmd in /tmp/pvmd.<UID>
- identifiziert sich beim pvmd durch Überprüfen gegenseitiger Schreibberechtigung

7.3 PVM auf MPP-Systemen

- nur ein pvmd auf dem MPP-System
- pvm_spawn angepaßt (z.B.: alloziert und lädt Nodes)
- Kommunikation:
  - zwischen inneren Knoten direkt mit internem Message-Passing
  - nach draußen: internes MP zum pvmd, von dort normal (UDP)
- auf einigen MPPs hersteller-eigene Anpassungen mit z.T. ganz anderer Funktionsweise

8 Übungen

- Übersetzen und starten Sie das Demo-Programm pi mit verschiedenen virtuellen Maschinen
- Lassen Sie das Programm matmul unter Debugger-Kontrolle laufen und testen Sie den jeweiligen Status des Gruppen-Servers mit pvm_gstat
- Probieren Sie das Programm xpvm am Programm matmul aus.
- Erweitern Sie das Programm pi so, daß der Master ein automatisches Load-Balancing durchführt.

8.1 Die wichtigsten xdb-Kommandos

- Voraussetzung: Compiliert und gelinkt mit der Option -g
- Aufruf: xdb progname
- Beenden: quit
- Liste aller Kommandos: help
- Starten eines Programms:

r [Argumente]	Programm (mit Argumenten) ausführen
r	mit den alten Argumenten neu starten
R	ohne Argumente neu starten

- Single Steps:

s	geht in Unterprogramme
S	führt Unterprogramme aus

- Breakpoints:

b line	setzt Breakpoint in Zeilennummer line
b routine	setzt Breakpoint in Routine
c	Weitermachen nach stop
lb	zeigt Breakpoints mit Nummer
db nummer / *	löscht Breakpoint nummer / alle

- Anzeigen des Source Codes:

v line /routine	Zeile line / Routine in der Mitte des Fensters
v	ein Fenster weiterblättern
V	aktuelle Zeile

- Anzeigen von Variablenwerten:

p expression	sehr universell

- Infos:

l	alle lokalen Variablen
lg	alle globalen Variablen
t	Stack

- Setzen von Variablen:
  p variable = value
- Startfile .xdbrc enthält Kommandos, die am Anfang ausgeführt werden sollen

A Sourcen der Beispielprogramme

A.1 pi_m.c

```
/*
 *      pi_m.c
 *
 *      Einfaches Beispiel fuer PVM
 *
 *      Berechnung von PI durch ein "Montecarlo"-Verfahren:
 *      Jeder Rechen-Prozess wuerfelt eine gewisse Anzahl von Werten aus,
 *      der Master-Prozess sammelt die Ergebnisse ein.
 *
 *      Aufruf:
 *              pi [Anzahl]
 *
 *       wobei Anzahl die Gesamtzahl der auszuwuerfelnden Punkte angibt
 */

#include "pi.h"

void main(int argc, char *argv[]) {
  /*
   * Master-Hauptprogramm:
   * Starten der Rechen-Prozesse
   * Verteilen der Aufgaben und Einsammeln der Ergebnisse
   */

  long   anzahl;                        /* Gesamtzahl der Punkte */
  long   p_anz;                         /* Zahl der Punkte pro Prozessor */
  long   treffer_einzel;                /* Zahl der Treffer eines Prozesses */
  long   treffer_gesamt = 0;            /* Gesamtzahl der Treffer */
  double pi;                            /* Ergebnis */
  int    nproc;                         /* Gesamtzahl der Rechen-Prozesse */
  int    *tids;                         /* Array der TIDs */
  int    i;

  pvm_mytid();                          /* Beim pvmd anmelden */
  pvm_config(&nproc, NULL, NULL);       /* Zahl der Prozessoren erfragen */

  /* TID-Array allozieren */
```

```
  tids = (int *) malloc(nproc * sizeof(int));

  /* soviele Node-Programme wie Prozessoren starten */
  pvm_spawn(SLAVE, NULL, PvmTaskDefault, NULL, nproc, tids);

  /* Anzahl der zu wuerfelnden Punkte pro Prozessor */
  if (argc != 2) {
    anzahl = DEFAULT_ANZAHL;
  } else {
    anzahl = 1000 * atol(argv[1]);
  }
  p_anz = (long) ceil(anzahl/nproc);
  anzahl = nproc * p_anz;              /* falls es nicht aufgeht */

  /* Arbeit an alle Rechen-Prozesse verteilen */
  pvm_initsend(PvmDataDefault);
  pvm_pklong(&p_anz, 1, 1);
  pvm_mcast(tids, nproc, MSGANZAHL);

  /* Einsammeln und Zusammensetzen der Ergebnisse */
  for (i = 0; i < nproc; i++) {
    pvm_recv(-1, MSGRESULT);
    pvm_upklong(&treffer_einzel, 1, 1);
    treffer_gesamt += treffer_einzel;
  }

  /* Bestimmen und Ausgeben des Ergebnisses */
  pi = (double)treffer_gesamt/(double)anzahl*4;
  printf("\nPI = %lf\n", pi);

  /* beim pvmd abmelden */
  pvm_exit();
}
```

A.2 pi_s.c

```
/*
 *      pi_s.c
 *
 *   Berechnung von PI durch "Montecarlo-Methode":
```

```
 *   Jeder Rechen-Prozess wuerfelt eine Anzahl von Zahlen im Einheitsquadrat
 *   und gibt die Zahl der Treffer im Viertelkreis zurueck.
 *   PI ist dann Trefferzahl/Gesamtzahl * 4
 */

#include "pi.h"

void main(int argc, char *argv[]) {
  /*
   * Hauptprogramm des Rechen-Prozesses
   * erledigt die Kommunikation mit dem Master
   */

  int  mastertid;                          /* TID des Masters */
  long anzahl;                             /* Gesamtzahl der Punkte */
  long treffer;                            /* Zahl der Treffer */

  long calc(long anzahl);                  /* eigentliche Rechnung */

  /* beim pvmd anmelden */
  pvm_mytid();

  /* TID des Masters holen */
  mastertid = pvm_parent();

  /* Anzahl der Punkte vom Master empfangen */
  pvm_recv(mastertid, MSGANZAHL);
  pvm_upklong(&anzahl, 1, 1);

  /* Ergebnis ausrechen */
  treffer = calc(anzahl);

  /* Ergebnis an Master schicken */
  pvm_initsend(PvmDataDefault);
  pvm_pklong(&treffer, 1, 1);
  pvm_send(mastertid, MSGRESULT);

  pvm_exit();
}

#include <sys/types.h>
```

```
long calc(long anzahl) {
  /*
   * eigentliche Berechnung:
   * anzahl Zufallspunkte im Einheitsquadrat werden ausgewuerfelt
   * und die Zahl der Treffer im Viertelkreis (x*x + y*y < 1)
   * zurueckgegeben.
   */

  double  x, y;                         /* Zufallspunkt im Einheitsquadrat */
  long    treffer = 0;                  /* Anzahl der Treffer */
  int     i;

  /* Zufallszahlen-Generator initialisieren */
  srand(getpid());

  for(i=0; i<anzahl; i++) {
    x = ((double) rand())/RAND_MAX;
    y = ((double) rand())/RAND_MAX;

    if ( x*x + y*y <= 1.0 ) {
      treffer++;
    }
  }

  return(treffer);
}
```

A.3 pi.h

```
/*
 *   pi.h   - Header fuer die einfache Version des PI-Programms
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pvm3.h>

#define SLAVE   "pi_s"              /* Name der Rechen-Programme */
#define MSGANZAHL  1
```

```
#define MSGRESULT  2

#define DEFAULT_ANZAHL 10000       /* Defaultwert fuer Anzahl */
```

A.4 matmul.c

```
/*
  matmul.c

  block matrix multiplication with the Pipe-Multiply-Roll Algorithm
  tasks are arranged as two-dimensional grid

  usage:
              matmul size gridsize

       size:       linear dimension of global array
       gridsize:   linear dimension of grid
*/

#include <stdio.h>
#include <string.h>
#include <math.h>
#include "pvm3.h"

#define PROGNAME "matmul"

#define BROADCAST 1       /* message type for broadcasting of initial data */
#define PIPE      2       /* message type for sending A matrices */
#define ROLL      3       /* message type for sending B matrices */
#define REDUCE    4       /* message type for reduce of local traces */

/*
 *  datatypes
 */
typedef struct {
  int   dim;        /* linear dimension of the quadratic grid */
  int   row;        /* row position of current task */
  int   col;        /* column position of current task */
} Grid;

/*
```

```
 *         function declarations
 */
void Startup(int argc, char *argv[], Grid *grid, int *mysize_p);
void MakeMatrices(int **A, int **B, int **C, int **tempA, int N, Grid grid);
void Pipe(int *matrix, int *temp, int size, int who, Grid grid);
void Multiply(int *A, int *B, int *C, int size, Grid grid);
void Roll(int *B, int size, Grid grid);
void PrintTrace(int *C, int size, Grid grid);
void Finalize(Grid grid);


void main(int argc, char *argv[]) {
  Grid grid;              /* geometric information of a task */
  int *C,*A,*B,*tempA;    /* submatrices used in multiplication */
  int localsize;          /* size of local submatrix */
  int i;                  /* loop counter */

  /* start tasks and broadcast infos */
  Startup(argc, argv, &grid, &localsize);

  /* initialize matrices */
  MakeMatrices(&A, &B, &C, &tempA, localsize, grid);

  /* iterate the Pipe-Multiply-Roll  */
  for (i=0; i<grid.dim; i++) {
    Pipe(A, tempA, localsize, i, grid);
    Multiply(tempA, B, C, localsize, grid);
    Roll(B, localsize, grid);
  }

  /* compute and print the trace of the result matrix */
  if (grid.col == grid.row) {
    PrintTrace(C, localsize, grid);
  }

  /* clean up and exit */
  Finalize(grid);
}

void Startup(int argc, char *argv[], Grid *grid, int *mysize_p) {
  /* get matrix size and dimension, start tasks and broadcast data */
```

```
int ptid;               /* parent's TID */
int *tids;              /* array of task id */
char *name;             /* name of program */
int nproc;              /* Number of processors */
int size;               /* linear matrix size */
int me;                 /* instance number in WORLD group */
int i,j;

pvm_mytid();                        /* enroll in PVM */
me = pvm_joingroup("WORLD");        /* join global group */

if (me == 0) {                      /* I am the master */
  name = PROGNAME;
  /* check arguments */
  if (argc != 3) {
    fprintf(stderr,"Usage: %s <matrix size> <grid dimension>\n",name);
    pvm_exit();
    exit(1);
  }

  /* Get matrix size, grid dimension and number of processes */
  size      = atoi(argv[1]);
  grid->dim = atoi(argv[2]);
  nproc     = grid->dim * grid->dim;

  if (size % grid->dim) {
    fprintf(stderr,"The grid dimension must divide the matrix size.\n");
    pvm_exit();
    exit(1);
  }

  /* start up other node programs and broadcast infos */
  if (nproc > 1) {
    pvm_catchout(stdout);
    tids = (int *) malloc( (nproc-1) * sizeof(int));
    pvm_spawn(name, NULL, PvmTaskDefault, NULL, nproc-1, tids);

    /* multicast size and grid dimension to all children */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&size, 1, 1);
```

```
      pvm_pkint(&grid->dim, 1, 1);
      pvm_mcast(tids, nproc-1, BROADCAST);
    }
  } else {    /* otherwise, receive data from node 0 */
    ptid = pvm_gettid("WORLD", 0);     /* who is the master? */
    pvm_recv(ptid, BROADCAST);
    pvm_upkint(&size, 1, 1);
    pvm_upkint(&grid->dim, 1, 1);
  }

  pvm_barrier("WORLD", grid->dim * grid->dim);   /* synchronize */

  /* compute my local info */
  *mysize_p  =   size / grid->dim;
  grid->row    =  me / grid->dim;
  grid->col    =  me % grid->dim;
}

void MakeMatrices(int **A, int **B, int **C, int **tempA, int N, Grid grid) {
  /* initialize the local submatrices of size NxN */
  /* globally: Aglob(i,j) = i+j, Bglob(i,j) = i-j     */
  /* locally : A(i,j) = i+j + (row+col)*dim */
  /* locally : B(i,j) = i-j + (row-col)*dim */

  int i,j, index;
  int len;
  int a_offset;
  int b_offset;

  len      = N*N*sizeof(int);
  a_offset = (grid.row + grid.col)*grid.dim;
  b_offset = (grid.row - grid.col)*grid.dim;

  /* Declare matrix storage in a one-dimensional array */
  *C     = (int *) malloc(len);
  *B     = (int *) malloc(len);
  *A     = (int *) malloc(len);
  *tempA = (int *) malloc(len);

  /* Initialize the matrices */
  for (i=0; i<N; i++) {
```

```
    for (j=0; j<N; j++) {
      index    = N*i + j;
      (*C)[index] = 0;
      (*A)[index] = i + j + a_offset;
      (*B)[index] = i - j + b_offset;
    }
  }
}

void Pipe(int *matrix, int *temp, int size, int round, Grid grid) {
  /* one node of each row sends its A matrix to all other nodes on the row */

  int i;
  int len;           /* number of ints to send in a message */
  int rcv_tid;       /* tid of receiving process */

  printf("Pipe starts on node (%d,%d).\n", grid.row, grid.col);
  len = size*size;

  /* If I am the broadcaster, send to all my neighbors */
  /*   and copy A explicitly */
  if (grid.col == (grid.row + round)%grid.dim) {
    pvm_initsend(PvmDataDefault);
    pvm_pkint(matrix, len, 1);
    for (i=0; i<grid.dim; i++) {
      if (i == grid.col) {
continue;           /* don't send to myself */
      }
      rcv_tid = pvm_gettid("WORLD", grid.row*grid.dim + i);
      pvm_send(rcv_tid, PIPE);
    }

    /* copy A to tempA (instead of receiving tempA) */
    memcpy(temp, matrix, len*sizeof(int));
  } else {
    /* receive the temp matrix */
    pvm_recv(-1, PIPE);
    pvm_upkint(temp, len, 1);
  }
}
```

```
void Multiply(int *A, int *B, int *C, int size, Grid grid) {
  /* multiplies the local submatrices */

  int i,j,k;
  int temp;

  printf("Multiply starts on node (%d,%d).\n", grid.row, grid.col);

  /* Multiply my subset of the matrices */
  for (i = 0; i < size; i ++) {
    for (j = 0; j < size; j++) {
      temp = 0;
      for (k = 0; k < size; k++) {
temp += A[i*size+k] * B[k*size+j];
      }
      C[i*size+j] += temp;
    }
  }
}

void Roll(int *B, int size, Grid grid) {
  /* rotates local B matrices one row upwards */

  int len;      /* number of bytes in a message to pass */
  int receiver; /* who to send to */
  int rcv_tid;  /* tid of receiver */

  printf("Roll starts on node (%d,%d).\n", grid.row, grid.col);
  len = size*size;
  if (grid.row != 0) {
    receiver = (grid.row - 1)*grid.dim + grid.col;
  } else {
    receiver = (grid.dim - 1)*grid.dim + grid.col;
  }
  rcv_tid = pvm_gettid("WORLD", receiver);

  pvm_initsend(PvmDataDefault);
  pvm_pkint(B, len, 1);
  pvm_send(rcv_tid, ROLL);

  pvm_recv(-1, ROLL);
```

```
  pvm_upkint(B, len, 1);
}

void PrintTrace(int *C, int size, Grid grid) {
  /* computes and prints the trace of the result matrix */
  /* called only by "diagonal" tasks */

  int i;
  int root_tid;  /* tid of the master process */
  int root;      /* instance number of master in group DIAG */
  int me_diag;   /* my instance number in group DIAG */
  int trace;     /* global trace of matrix C */

  trace = 0;

  me_diag = pvm_joingroup("DIAG");   /* join diagonal group */
  pvm_barrier("DIAG", grid.dim);     /* synchronize */

  /* compute local trace */
  for (i=0; i<size; i++) {
    trace += C[i*size+i];
  }
  printf("local trace on (%d,%d): %d\n", grid.row, grid.col, trace);

  /* root is the task with inst=0 in WORLD group (the master) */
  root_tid = pvm_gettid("WORLD", 0);
  root     = pvm_getinst("DIAG", root_tid);

  /* reduce local values to global trace */
  pvm_reduce(PvmSum, &trace, 1, PVM_INT, REDUCE, "DIAG", root);

  /* root prints the global result */
  if (me_diag == root) {
    printf("Total trace = %d\n", trace);
  }

  pvm_barrier("DIAG", grid.dim);  /* synchronize */
  pvm_lvgroup("DIAG");
}

void Finalize(Grid grid) {
```

```
  /* leave group and pvm */
  pvm_barrier("WORLD", grid.dim*grid.dim);  /* synchronize */
  pvm_lvgroup("WORLD");
  pvm_exit();
}
```