

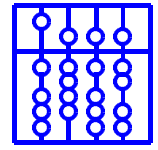
PVM — Parallel Virtual Machine

Thomas Ludwig
LRR-TUM

Institut für Informatik

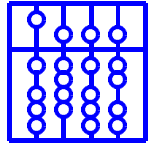
Technische Universität München

e-mail: ludwig@informatik.tu-muenchen.de



Inhaltsübersicht

● Einführung	2
● Die virtuelle Maschine	6
● Konzepte in PVM 3.3.x	7
● Hello World	10
● Wichtige PVM Routinen	12
● PVM Hostfile	15
● Die PVM-Konsole	16
● Dynamische Prozeßgruppen	17
● Nachrichtentransport in PVM	21
● Fehlersuche in PVM-Programmen	23
● Zusätzliche Informationen	24
● PVM vs. MPI	25



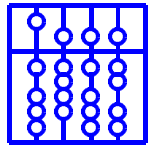
Einführung

Entwicklungsziele:

Optimale Unterstützung des heterogenen Netzwerk-Rechnens (Forschungsverbund amerikanischer Universitäten)

Entwicklerteam:

- Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam
- Oak Ridge National Lab, Univ. of Tennessee, Emory Univ., Carnegie Mellon Univ.



Entwicklungsziel von PVM:

PVM is an integrated set of software tools and libraries that emulates a general-purpose, flexible, heterogeneous concurrent computing framework on interconnected computers of varied architecture. The overall objective of the PVM system is to enable such a collection of computers to be used cooperatively for concurrent or parallel computation.

Entwicklungsgeschichte:

PVM 1.0 nie herausgegeben

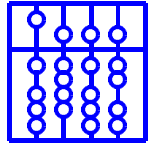
PVM 2.0 Februar 1991

PVM 3.0 Februar 1993: komplettes Redesign

PVM 3.2 August 1993: Grundlage für Herstellerversionen (z.B. IBM)

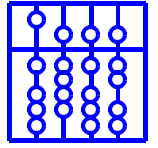
PVM 3.3 Juni 1994: Neue, umfassendere Kommunikationsaufrufe

PVM 3.4 1997: Thread safety, Authentifizierung, Kontexte



Grundprinzipien von PVM:

- Benutzer-konfigurierter Pool verwendeter Rechnerknoten beliebiger Architektur. Der Pool ist dynamisch veränderbar.
- Architektureigenschaften wahlweise verdeckt oder vollständig ausnutzbar.
- Prozeß-basiertes Verarbeitungsmodell: die Recheninstanz in PVM heißt *task* (meist gleich Prozeß) und alterniert zwischen Rechnen und Kommunizieren. Prozesse können beliebig an Knoten zugewiesen werden.
- Expliziter Nachrichtenaustausch zum Zwecke der Prozeßkooperation. Die Nachrichtengröße ist unbeschränkt.
- Unterstützung von Heterogenität.
- Unterstützung von Multiprozessorsystemen. Manche Hersteller bieten ein eigenes, optimiertes PVM für ihre Architekturen an.



Hauptbestandteile von PVM:

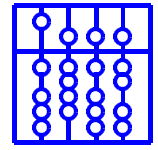
- Daemon-Prozeß `pvmd3` für jeden Knoten, der Element der virtuellen Maschine ist (ein `pvmd3` pro Benutzer).
- Bibliothek von PVM Routinen zum Zwecke des Nachrichtenaustauschs, der Verwaltung von Prozessen und der Handhabung der virtuellen Maschine.

Existierende Sprachanbindungen:

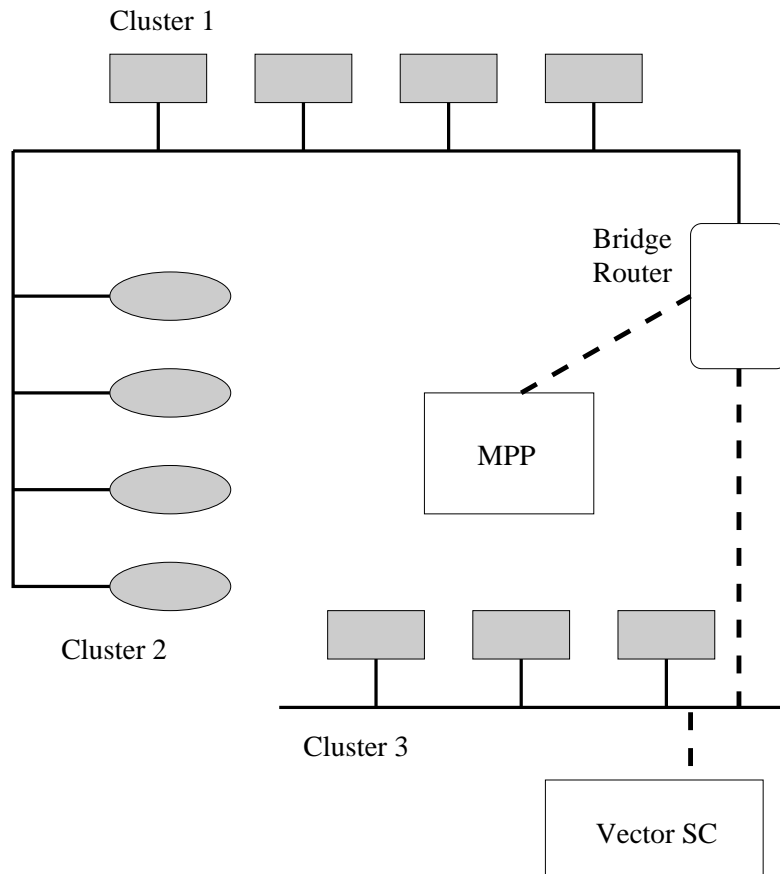
C, C++, Fortran, Tcl/Tk, CommonLisp, Java

Vorgehensweise:

- Kreiere die virtuelle Maschine durch Aufruf von `pvm`.
- Ausgangspunkt kann jeder beliebige Unix Rechner sein.
- Dieser Rechner wird zum Host-Rechner der Anwendung.
- Starte die zum Programm gehörigen tasks auf den Knoten der virtuellen Maschine.

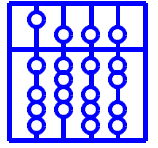


Die virtuelle Maschine



PVM sieht die Umgebung sowohl als abstrahierte virtuelle Maschine als auch als Menge von Rechnerknoten konkreter Architektur

⇒ Optimale Anpassung des Programms und des Programmierstils möglich.



Konzepte in PVM 3.3.x

Benutzerschnittstelle: hier nur für c

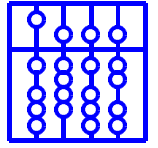
- Alle Funktionen beginnen mit `pvm`

Identifikatoren für Task-Objekte:

- Typ integer
- Eindeutig innerhalb einer virtuellen Maschine
- Werden vom lokalen Dämon gewählt
- Zusätzliche Namen für Prozeßgruppen

Prozeßkontrolle:

- Prozesse können sich dynamisch bei der virtuellen Maschine an- und wieder abmelden.
- Dynamische Verwaltung von Prozessen



Fehlertoleranz:

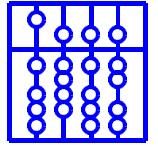
- Fehlerhafter Knoten werden automatisch erkannt
- Nur Basismechanismus, keine Fehlerbehandlung

Prozeßgruppen:

- Aufsetzend auf PVM implementiert
- Dynamisch bzgl. ihrer Größe
- Join- und leave-Operationen
- Verwendet für Barrieren, Broadcasts und globale Reduktionsfunktionen

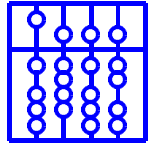
Signale:

- Unix-Signale (Vorsicht!)
- Speziell markierte Nachrichten



Kommunikation:

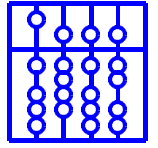
- Verpacken/Auspacken von Nachrichten
- Asynchrones blockierendes Senden/Empfangen: warte, bis Puffer gesendet/gefüllt
- Nichtblockierendes Empfangen: empfangen nur, wenn Daten vorhanden
- Multicast (zu einigen Prozessen), Broadcast (zu allen Prozessen)
- Nachrichtenreihenfolge garantiert (!)
- Markierte Nachrichten (tagged messages)



Hello World

```
#include "pvm3.h"
main ()
{
    int cc, tid, msgtag;
    char buf[100];
    printf("i'm t%x\n", pvm_mytid());
    cc = pvm_spawn("hello_other", (char**)0,
                  0, "", 1, &tid);
    if (cc == 1) {
        msgtag = 1;
        pvm_recv(tid, msgtag);
        pvmupkstr(buf);
        printf("from %tx: %s\n", tid, buf);
    } else
        printf("can't start hello_other\n");
    pvm_exit();
}
```

PVM Programm hello_other.c



```
#include "pvm3.h"

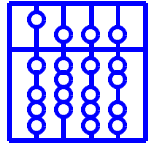
main ()
{
    int ptid, msgtag;
    char buf[100];

    ptid = pvm_parent();

    strcpy(buf, "hello world from ");
    gethostname(buf+strlen(buf), 64);
    msgtag = 1;
    pvm_initsend(PvmDataDefault);
    pvm_pkstr(buf);
    pvm_send(ptid, msgtag);

    pvm_exit();
}
```

PVM Programm hello_other.c



Wichtige PVM Routinen

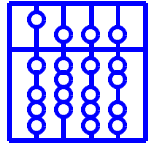
Starten einer Anwendung

```
int tid = pvm_mytid (void)
```

Übergibt den Prozeß an PVM, wenn er noch nicht früher übergeben worden ist. Liefert die eindeutige `tid` zurück, die der Dämon verwaltet.

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

Startet `ntask` Kopien des ausführbaren Programms `task`. Die Knoten, auf denen die Task erzeugt werden, sind durch `flag` und `where` bestimmt. Verschiedenste Varianten sind hier möglich. `numt` gibt die Anzahl der erfolgreich gestarteten Task-Objekte an, `tids` enthält die einzelnen Task-Identifikatoren



Kommunikation

```
int bufid = pvm_initsend (int encoding)
```

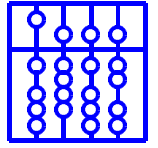
Löscht den default-Sendepuffer und spezifiziert die Art der Nachrichtenübertragung

```
int info = pvm_send (int tid, int msgtag)
```

Sendet die Nachricht im aktiven Sendepuffer zum PVM-Prozeß `tid`. `msgtag` dient zum Kennzeichnen der Nachricht. Der Aufruf kehrt zurück, sobald die Nachricht erfolgreich abgesendet(!) worden ist.

```
int bufid = pvm_recv (int tid, int msgtag)
```

Blockiert den aufrufenden Prozeß bis eine Nachricht mit Kennzeichen `msgtag` vom PVM-Prozeß `tid` empfangen wurde. Wahlweise können `tid` und `msgtag` auf **-1** gesetzt werden, um die Selektion auszuschalten. Die Nachricht steht im Puffer `bufid`



Pufferverwaltung

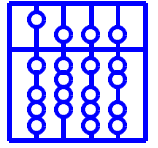
```
int info = pvm_pkTYPE (...)  
int info = pvm_pkstr (char *sp)  
int info = pvm_pkint (int *ip, int nitem,  
                     int stride)
```

Jede `pvm_pkTYPE` Routine packt Felder des Datentyps `TYPE` in den aktuellen Sendepuffer. `nitems` Elemente werden verpackt. Sie werden einem Feld (z.B. `ip`) entnommen, in dem sie im Abstand `stride` aufeinander folgen

```
int info = pvm_upkTYPE (...)  
int info = pvm_upkstr (char *sp)  
int info = pvm_upkint (int *ip, int nitem,  
                     int stride)
```

Jede Routine packt Daten des Typs `TYPE` aus dem aktiven Empfangspuffer aus. `nitems` Elemente werden ausgepackt und einem Feld (z.B. `ip`) im Abstand `stride` zugewiesen

Die Abfolge der `pvm_pk` Aufrufe beim Senden und der zugehörigen `pvm_upk*` Aufrufe beim Empfangen muß unbedingt übereinstimmen!*

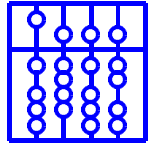


PVM Hostfile

Das Hostfile wird genutzt, um Rechner der virtuellen Maschine zu spezifizieren

```
#example hostfile
sunbode20 dx=/usr/local/bin/pvm3d
sunbode26
sunbode28
&ibode3 lo=ludwigt
&ibode4 lo=ludwigt
```

Im Hostfile können verschiedenste Login-Namen, Umgebungsvariablen und Pfade für die einzelnen Rechner gesetzt werden.



Die PVM-Konsole

```
% pvm hostfile
```

startet die PVM Konsole und liefert den Konsolenprompt.

```
pvm> help
```

liefert eine Übersicht über verfügbare Kommandos. *help command* liefert Hilfestellung zum Kommando *command*.

```
pvm> conf
```

liefert eine Tabelle, die die aktuelle virtuelle Maschine beschreibt.

```
pvm> quit
```

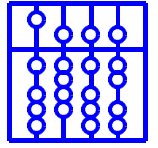
beendet die Konsolensitzung. PVM läuft weiter. Erneutes Starten der Konsole mit `pvm`.

```
pvm> halt
```

beendet die Konsolensitzung, stoppt die virtuelle Maschine und entfernt alle laufenden Tasks.

Ein Protokoll der Sitzung findet sich in

```
/tmp/pvml.<uid>
```



Dynamische Prozeßgruppen

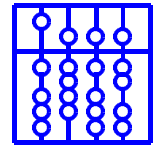
Gruppenkonzept

- Zusammenfassung beliebiger Mengen von Prozessen zu einer namentlich benannten Gruppe
- Prozesse können Gruppen jederzeit beitreten und sie verlassen (dynamische Gruppen)
- Prozesse können gleichzeitig Mitglied in mehreren Gruppen sein
- Jedes Mitglied in einer Gruppe hat eine eindeutige Mitgliedsnummer

Verwendung von Prozeßgruppen

- Globale Synchronisation
- Rundruf (Broadcast-Nachricht)
- Globale arithmetische Funktionen

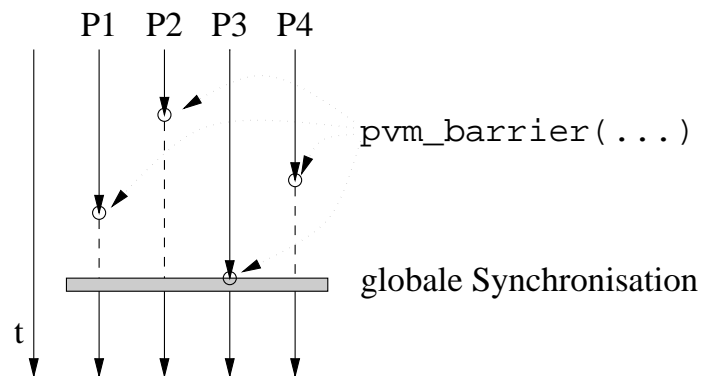
d.h. koordinierte Aktionen einer Menge von Prozessen



Barrieren

Globale Synchronisation aller Mitglieder einer Gruppe an einem Ausführungspunkt

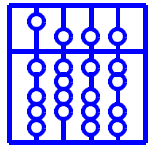
Beispiel



Vier Prozesse nehmen an einer globalen Synchronisation teil (Barriere). Die ersten drei am Aufruf eintreffenden werden so lange blockiert, bis der letzte auch den Aufruf absetzt

```
int info =
    pvm_barrier (char *group, int count)
```

- Aufrufer muß selbst Gruppenmitglied sein
- `count` Mitglieder müssen aufrufen; `count` darf zwar kleiner sein als die aktuelle Gruppengröße, sollte i.a. aber gleich sein



Rundruf

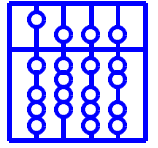
Sende die Nachricht im aktiven Puffer an alle Mitglieder einer Gruppe (aber nicht(!) an den Aufrufer selbst)

```
int info =
    pvm_bcast (char *group, int msgtag)
```

msgtag sollte die Nachricht eindeutig als Rundruf zu erkennen geben

Verhalten

- `pvm_bcast` ist asynchron. Blockiert nur so lange, bis die Nachrichten erfolgreich auf dem Weg sind
- Beim Aufruf werden die `tids` der aktuellen Gruppenmitglieder ermittelt. Sie erhalten die Nachricht einzeln zugeteilt (Effizienz abhängig von Zielarchitektur)



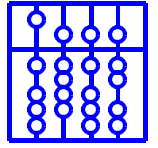
Globale arithmetische Operationen

- Hierbei wird eine arithmetische Operation über den Operanden der einzelnen Gruppenmitglieder ausgeführt
- Vordefinierte Funktionen in PVM: `PvmMax`, `PvmMin`, `PvmSum`, `PvmProduct`
- Eigene globale Funktionen können definiert werden

```
int info =
    pvm_reduce (void (*func)(), void *data,
               int nitem, int datatype,
               int msgtag,
               char *group, int root)
```

Ausführung

- Jedes Gruppenmitglied stellt `nitem` Datenelemente ab Adresse `*data` zur Verfügung (Typ durch `datatype` gegeben)
- Die Anwendung der Funktion `func` erfolgt auf Paare solcher Datenbereiche
- Das Gruppenmitglied mit Nummer `root` erhält das Ergebnis. Es überschreibt seine bereitgestellten Daten
- Außer für `root` ist der Aufruf nicht-blockierend



Nachrichtentransport in PVM

Beteiligte Komponenten:

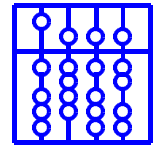
- Knoten-lokale Dämonprozesse (pvmd)
- Bibliotheksroutinen von PVM (libpvm3.a)
- Das parallele Programm

Varianten:

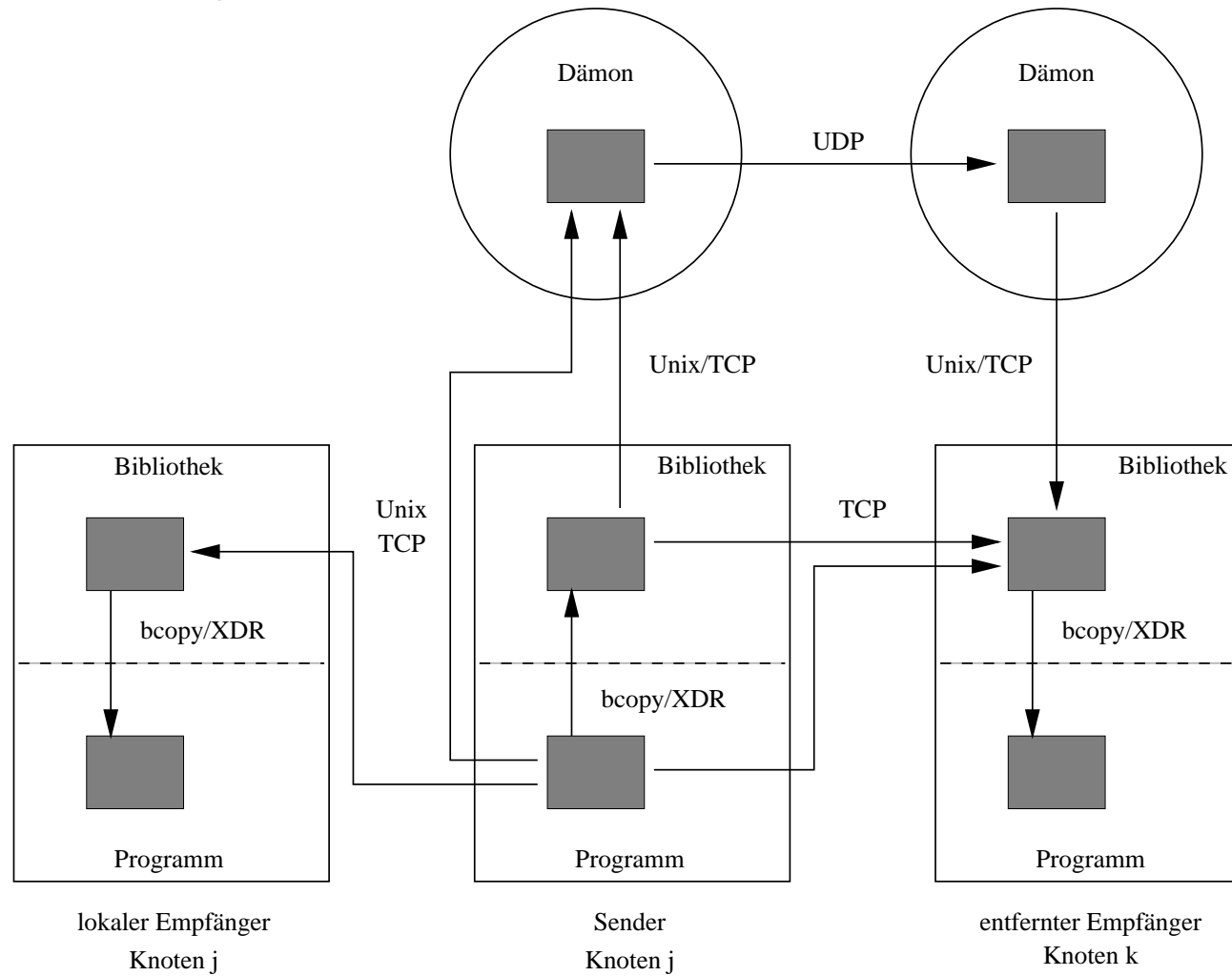
- Reguläres Datenversenden mit Einbeziehung des Dämons (zum Auffinden des Ortes des Empfängers)
 - Direktes Versenden zwischen zwei Tasks auf verschiedenen Knoten
 - Lokales Versenden auf einem Knoten
- Stets kritisch: Anzahl der Kopiervorgänge*

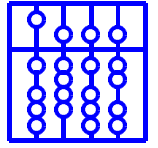
Heterogenität

Bei Maschinen mit verschiedenem Darstellungsformat für Daten ist eine Konvertierung zwischen diesen Formaten notwendig
⇒ XDR (erzeugt neuerliches Kopieren)



Nachrichtentransportwege





Fehlersuche in PVM-Programmen

Eines der traurigeren Kapitel...

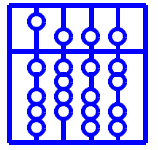
Die Konsole kann direkt von einem Debugger aus gestartet werden

Erzeugte Tasks müssen das Bit `PvmDebugTask` gesetzt bekommen.

⇒ PVM startet das Skript `pvm3/lib/debugger`

Vierstufiger Ansatz

1. Erst als sequentiellen Prozeß laufen lassen und Fehler entfernen
2. Mit 2-4 Prozessen auf einer Workstation. Fehler entfernen
3. Auf mehreren Workstations mit mehreren Prozessen. Fehler entfernen
4. Hoffen, daß es auf allen Konfigurationen läuft (meist vergeblich)



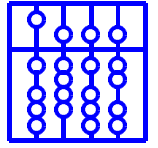
Zusätzliche Informationen

Man-Pages:

- `man pvm`

Im PVM-Buch im Web:

- Kapitel 7.7: Task Environment
- Kapitel 8.1: XPVM
- Kapitel 9.2: Getting PVM Running
- Kapitel 9.3: Compiling Applications
- Kapitel 9.4: Running Applications
- Kapitel 9.5: Debugging and Tracing



PVM vs. MPI

PVM

- Geringer Kommandoumfang; leicht zu erlernen
- PVM gut geeignet für heterogene Workstationnetze
- Dynamisches Prozeßkonzept notwendig für manche Algorithmen
- Dynamisches Prozeßkonzept Voraussetzung für SW-Fehlertoleranz
- Viele Entwicklungswerkzeuge (Public domain und kommerziell)

MPI

- Hoher Kommandoumfang; viele Möglichkeiten
- MPI gut geeignet für homogene Hochleistungsrechenumgebungen
- In MPI-1 nur statisches Prozeßkonzept
- Parallele E/A durch MPI-IO künftig gleich integriert
- Viele native Realisierungen auf Hochleistungsrechnern