

Verteilte Systeme

10. Middleware I

Verteiltes Programmieren leicht gemacht ...

Verteilte Middleware

- Vermittlerrolle zwischen verteilter Anwendung und Rechnernetz
- Teil moderner Systemsoftware

Einfache Handhabbarkeit

Leichte Portierbarkeit

Hohe Abstraktionen

- Gruppen
- Logische Uhren
- Transaktionen
- Verteilte Algorithmen
- ...

Wo ist die Mitte?

The diagram illustrates the architecture of a distributed system. It shows a central stack of four layers: Hardware (bottom, grey), Systemsoftware (second from bottom, grey), Middleware (third from bottom, white), and Verteilte Anwendung (top, grey). To the left, a node is represented by a stack of three grey boxes, with the top two containing the labels 'VA' and 'M'. To the right, another node is represented by a stack of four grey boxes. Lines connect the Hardware layer to both nodes, and the Middleware layer to both nodes. The top layer, Verteilte Anwendung, is connected to the right-hand node.

Verteilte Systeme, Wintersemester 2000/2001 Folie 10.2

Verteilte Middleware I

Einfache Middleware mit dem Funktionsumfang

- Verwaltung beteiligter Rechner
- Unicast
- Datenkonvertierung
- Verteilte Kontrollalgorithmen
- Einfache Multicast-Varianten

PVM = Parallel Virtual Machine

- Einer der ersten Middleware-Ansätze (1989)
- Leicht verständlich, Leicht installierbar, Breite Verfügbarkeit

MPI = Message Passing Interface

- Im Vergleich zu PVM "ausgereifter" und größerer Funktionsumfang
- Versuch eines Standards für nachrichtenbasierte Kommunikation

Primäres Einsatzgebiet

- Verteilung numerischer Simulationen wie z.B. Mehrgitterverfahren

10.1 Parallel Virtual Machine (PVM)

PVM

Anfang: 1989 am Oak Ridge National Laboratory (ORNL)

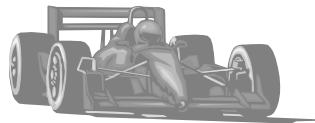
Abstrakte Schicht

- Realisiert auf einer benutzerdefinierten Menge von Rechnern in einem Netz die Sicht eines großen Rechners mit verteiltem Speicher

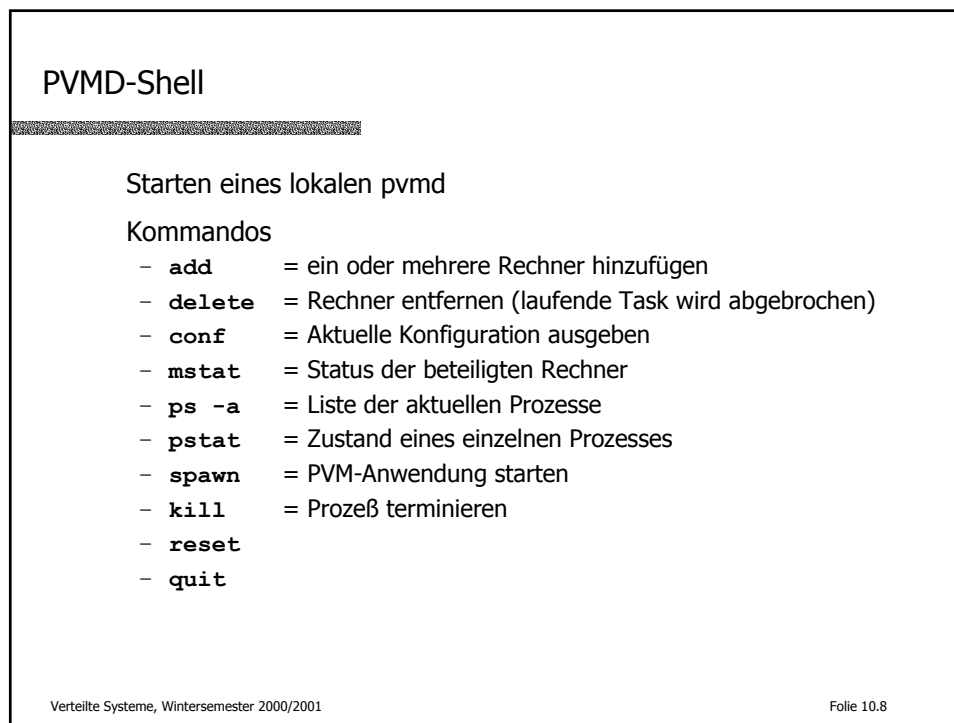
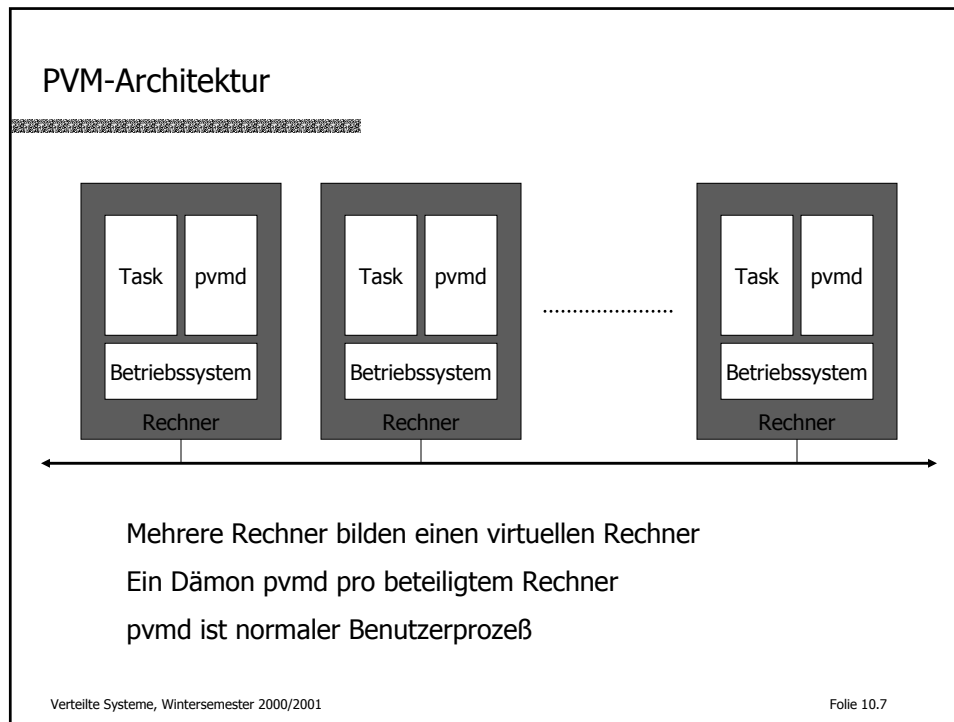
Funktionsumfang

- Verwaltung der Rechnermenge
- Verwaltung sogenannter Tasks
- Kommunikation
- Synchronisation
- Unterstützung für heterogene Systeme

Unterstützte Architekturen



Alliant FX/8
DEC Alpha
Sequent Balance
Sequent Symmetry
Intel-basierte UNIX-Systeme
Thinking Machines CM2
Thinking Machines CM5
Cray C-90
Cray YMP
Cray-2
Cray S-MP
HP-9000 Model 300, PA-RISC
Intel Paragon
Silicon Graphics IRIS
SUN 3, Sparcs
...



Skelett eines PVM-Prozesses

```

int main (int ac, char **av) {
    int tid;
    tid = pvm_mytid();
    ...
    pvm_exit();
}

```

Erster Aufruf von `pvm_mytid()` meldet den Prozeß an
 – Rückgabewert eindeutige Taskidentifikation

`pvm_exit()` meldet nur ab (keine Terminierung)

Erzeugung weiterer Prozesse

```

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

```

Werte für `flag`:

- `PvmTaskDefault`
pvmd wählt Rechner aus
- `PvmTaskHost`
In where steht Zielrechner
- `PvmTaskArch`
In where steht Architekturtyp
- `PvmTaskDebug`
Prozesse werden in einem Debugger gestartet
- `PvmTaskTrace`
Aufruf der erzeugten Tasks werden protokolliert

`ntask` = Anzahl der zu erzeugenden Prozesse

`numt` = Anzahl erfolgreich instanzierter Prozesse

Verwaltungsfunktionen

```
int tid = pvm_parent()
Identifikation des erzeugenden Prozesses sonst PvmNoParent

int pstat = pvm_stat(int tid)
Status einer Task

int mstat = pvm_mstat(char *host)
Informationen über den angegebenen Rechner

int info = pvm_config(...)
Informationen über den virtuellen Rechner

int info = pvm_tasks(...)
Informationen über bestimmte oder alle Tasks

pvm_kill(int tid)
Terminierung einer Task erzwingen
```

Verteilte Systeme, Wintersemester 2000/2001

Folie 10.11

Einfachstkommunikation: Signale

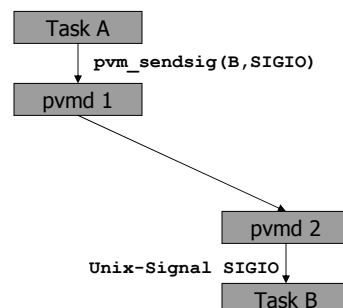
```
int info = pvm_sendsig (
    int tid,
    int signum
)
Sendet Signal signum an Task tid
```

```
int info = pvm_notify (
    int about,
    int msgtag,
    int ntask,
    int *tids
)

```

Benachrichtigung der angegebenen Tasks über besondere Ereignisse

- Task terminiert
- Rechner wurde ausgetragen oder stürzte ab
- Weitere Rechner kamen hinzu



Verteilte Systeme, Wintersemester 2000/2001

Folie 10.12

Nachrichtenkommunikation

Senden einer Nachricht:

- Aufbau eines Sendepuffers
- Einpacken der Nachricht (Datenkonvertierung)
- Absenden der Nachricht

Empfang einer Nachricht:

- Aufruf einer Empfangsoperation
- Auspacken der Nachricht

Alle Nachrichten besitzen eine Kennung (Message Tag)

Sende- und Empfangspuffer

Explizite Erzeugung und Freigabe

- `int bufid = pvm_mkbuf (int encoding)`
- `int info = pvm_freebuf (int bufid)`
- encoding legt die Konvertierungsmethode fest:
 - xdr-Konvertierung (PvmDataDefault)
 - Keine Konvertierung (PvmDataRaw)
 - Pointer auf Daten statt Daten (PvmDataInPlace)

Ein aktiver Sendepuffer pro Task

- `int old_bufid = pvm_setsbuf (int new_bufid)`
- `int bufid = pvm_getsbuf ()`

Ein aktiver Empfangspuffer pro Task

- `int old_bufid = pvm_setrbuf (int new_bufid)`
- `int bufid = pvm_getrbuf ()`

Ein- und Auspacken

Datenkonvertierung wird nicht direkt unterstützt

Basiskonvertierungen für

- Byte, Float, Double, Integer, Long, Short, String
- Grundformat

```
int info = pvm_pkdouble (
    double *data,
    int nitem,
    int stride
)

int info = pvm_upkdouble ( ... )
```

Explizite Nutzung der xdr-Routinen

- Datentypen definieren
- RPCGEN aufrufen

Senden und Empfangen

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

Die Nachricht im aktuellen Sendepuffer wird versendet

```
int info = pvm_mcast(int *tids, int ntids, int msgtag)
```

Einfacher Multicast

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

Nichtblockierendes Empfangen

- Keine Nachricht wenn bufid = 0
- tid = Nachricht von einem bestimmten Task oder -1
- msgtag = Bestimmter Nachrichtentyp oder -1

```
int bufid = pvm_recv(...)
```

Blockierendes Empfangen

```
int bufid = pvm_probe(...)
```

Asynchrones Empfangen

Weitere Funktionen

Aufbau direkter Kommunikationsverbindungen

- Zwei Task kommunizieren eng miteinander

Gruppenkommunikation

- Noch experimenteller Status
- Eigener Gruppenserver
- Keine Garantien
- Barrier-Synchronisation
 - Warten, bis eine vorgegebene Anzahl von Prozessen `pvm_barrier()` aufgerufen hat

Implementierung

- pvmds kommunizieren über UDP
- pvmd, lokale Tasks und direkte Verbindungen nutzen TCP

10.2 Message Passing Interface (MPI)

MPI

MPIF = Message Passing Interface Forum

- mehr als 40 Organisationen
- Beginn November 1992
- Viele PVM-Entwickler beteiligt

Vergleichbar PVM, aber ausgereifter und größerer Funktionsumfang

- Unicast
- Kollektive Operationen (Maximumsuche, Barrier, ...)
- Prozeßgruppen
- Kommunikationskontexte

Sprachbindungen

- C und C++
- Fortran

Kollektive Operationen

Multicast-Kommunikation

Barrier-Synchronisation

Scatter/Gather-Operationen

- Verteilen von Datenstücken auf mehrere Rechner
- Einsammeln verteilter Datenstücke

Globale Reduktionsoperationen

- Maximum, Minimum
- Summe, Produkt
- Logische Operationen
- Benutzerdefinierte Operationen möglich
- Varianten
 - Resultat kommt bei einem Prozeß an
 - Resultat kommt bei allen Prozessen an
 - Resultat kommt bei ausgewählten Prozessen an (Prefix Scan)

Zusammenfassung

Entwicklung einfacher verteilter Anwendungen wird deutlich erleichtert

- Im Bereich numerischer Simulationen sehr erfolgreich
- Gute Übungsgrundlage für Vorlesungen

Einsatz scheitert meist, wenn die verteilte Anwendung komplexer wird

- Keine vollständige Integration der Systemsoftware
- Work-Arounds
- ...

Offene Probleme

- Bessere und automatisierte Lösung der Datenkonvertierung
- Höhere Garantien bei Multicast-Kommunikation
- Stärkere Systemintegration

Übungsaufgaben

1. Implementieren Sie den in einer früheren Übung bereits besprochenen Raymond-Algorithmus für verteilten wechselseitigen Ausschluß unter Verwendung von PVM oder MPI.

Die Anzahl der beteiligten Prozesse (Rechner) sollte parametrisierbar sein.

Literatur

A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek,
V. Sunderam
PVM 3 User's Guide and Reference Manual
ORNL/TM-12187, 1993
typischerweise Teil einer PVM-Distribution