Gruppenfunktionen (1)

- Mehrere Tasks können zu einer Gruppe zusammengefaßt werden
- Operationen für Gruppen (Broadcast, Barrier-Synchronisation)
- Dynamische Gruppenkonfiguration
- Group-Server `pvmgs`, wird beim ersten Aufruf einer Gruppenfkt. gestartet
- Gather- und Scatter-Operationen zum Verteilen von Arrays
- Reduktions-Operationen innerhalb der Gruppe (`PvmMax`, `PvmMin`, `PvmSum`, `PvmProduct`)
- Gruppennamen sind Character-Strings
- Beispiele in `/sw/share/pvm3/gexamples`

Gruppenfunktionen (2) C

- `inum pvm_joingroup(*group)`
  der rufende Task wird in die Gruppe aufgenommen
- `info pvm_lvgroup(*group)`
  der rufende Task verläßt die Gruppe
- `size pvm_gsize(*group)`
  Anzahl der Tasks in der Gruppe
- `tid pvm_gettid(*group, inum)`
  Gruppen-spezifische Tasknummer
- `int pvm_barrier(*group, count)`
  synchronisiert Tasks einer Gruppe
- `info pvm_bcast(*group, msgtag)`
  Broadcast innerhalb einer Gruppe
- `info pvm_scatter(*result, *data, count, datatype, msgtag, *group, rootginst)`
  Ein Array wird auf die Tasks einer Gruppe verteilt
- `info pvm_gather(*result, *data, count, datatype, msgtag, *group, rootginst)`
  Daten von allen Tasks werden von einem Task gesammelt
- `info pvm_reduce(....)`
  Reduktionsoperation innerhalb der Gruppe

Gruppenfunktionen (2) Fortran

- `call pvmfjoingroup(group, inum)`
  der rufende Task wird in die Gruppe aufgenommen
- `call pvmflvgroup(group, info)`
  der rufende Task verläßt die Gruppe
- `call pvmfgsize(group, size)`
  Anzahl der Tasks in der Gruppe
- `call pvmfgettid(group, inum, tid)`
  Gruppen-spezifische Tasknummer
- `call pvmfbarrier(group, count, info)`
  synchronisiert Tasks einer Gruppe
- `call pvmfbcast(group, msgtag, info)`
  Broadcast innerhalb einer Gruppe
- `call pvmfscatter(result, data, count, datatype, msgtag, group, rootginst, info)`
  Ein Array wird auf die Tasks einer Gruppe verteilt
- `call pvmfgather(result, data, count, datatype, msgtag, group, rootginst, info)`
  Daten von allen Tasks werden von einem Task gesammelt
- `call pvmfreduce(....)`
  Reduktionsoperation innerhalb der Gruppe

Dynamische Konfiguration etc. (C)

- `info pvm_addhosts(**hosts, nhost, *infos)`
  bindet zusätzliche Hosts ein
- `info pvm_delhosts(**hosts, nhost, *infos)`
  löscht Hosts
- `pstat pvm_pstat(tid)`
  Status des PVM Prozesses
- `mstat pvm_mstat(*host)`
  ergibt Status eines Host (OK, NoHost, HostFail)
- `info pvm_config(*nhost, *narch, hostinfo, **hostp)`
  gibt Information über die augenblickliche Konfiguration
- `oldval pvm_setopt(what, value)`
  allgemeine Funktion um Parameter zu setzen (z.B. Routing-Verhalten, Debug, etc.)
- `bufid pvm_catchout(*ff)`
  Leitet den Output von Child-Tasks in eine Datei um

Dyn. Konfiguration etc. (Fortran)

- `call pvmfaddhost(host, info)`
  bindet zusätzliche Hosts ein
- `call pvmfdelhost(host, info)`
  löscht Hosts
- `call pvmfpstat(tid, pstat)`
  Status des PVM Prozesses
- `call pvmfmstat(host, mstat)`
  ergibt Status eines Host (OK, NoHost, HostFail)
- `call pvmfconfig(nhost, narch, dtid, name, arch, speed, info)`
  gibt Information über die augenblickliche Konfiguration
- `call pvmfsetopt(what, value, oldval)`
  allgemeine Funktion um Parameter zu setzen (z.B. Routing-Verhalten, Debug, etc.)
- `call pvmfcatchout(on off)`
  Leitet den Output von Child-Tasks in eine Datei um

Error Handling, Signale (C)

- `info pvm_perror(*msg)`
  schreibt die Fehlermeldung des letzten PVM-Aufrufes in `/tmp/pvml.<uid>`
- `info pvm_sendsig(tid, signum)`
  sendet UNIX-Signale an andere Prozesse (nur für Experten)
- `info pvm_notify(about, msgtag, ntask, *tids)`
  übermittelt ein Signal bei spezifizierten PVM-Events (`PvmTaskExit`, `PvmHostAdd`). Entsprechendes Receive muß vorhanden sein

Error Handling, Signale (Fortran)

- `call pvmfperror(msg, info)`
  schreibt die Fehlermeldung des letzten PVM-Aufrufes in `/tmp/pvml.<uid>`
- `call pvmfsendsig(tid, signum, info)`
  sendet UNIX-Signale an andere Prozesse (nur für Experten)
- `call pvmfnotify(about, msgtag, ntask, tids, info)`
  übermittelt ein Signal bei spezifizierten PVM-Events (`PvmTaskExit`, `PvmHostAdd`). Entsprechendes Receive muß vorhanden sein

Performance Aspekte

Message Passing

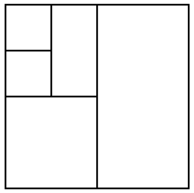
- Packing (`pvm_initsend`): 3 Modi
  - `PvmDataDefault`: XDR
  - `PvmDataRaw`: ohne Kodierung
  - `PvmDataInPlace`: kein physikalisches Kopieren, Definition eines Pointers (> Ver 3.2)
- Übertragungsprotokolle
  - Pvmd ↔ Pvmd: UDP-Sockets (paketorientiert)
  - Pvmd ↔ Task, Task ↔ Task: TCP (verbindungsorientiert)

  Task ↔ Task: Funktion `pvm_setopt()`

Programmiertechnik I

Struktur der Tasks (Prozesse)

- Crowd
  - Master ↔ Slave
  - Node only (SPMD = **S**ingle **P**rogramm **M**ultiple **D**ata)
- Baumstruktur
  Tasks werden dynamisch generiert (z.B. *branch-and-bound*, rekursives *divide-and-conquer*)



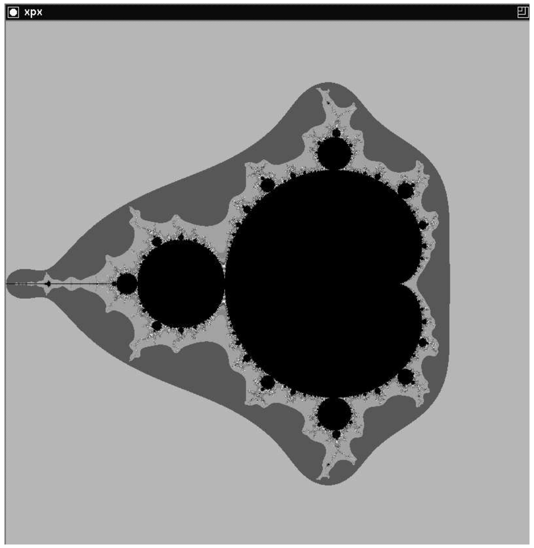
gr14

- Hybrid
  beliebige Mischform (Crowd, Tree, heterogene Applikationen)

Crowd (1)

Beispiel 1: Mandelbrotmenge

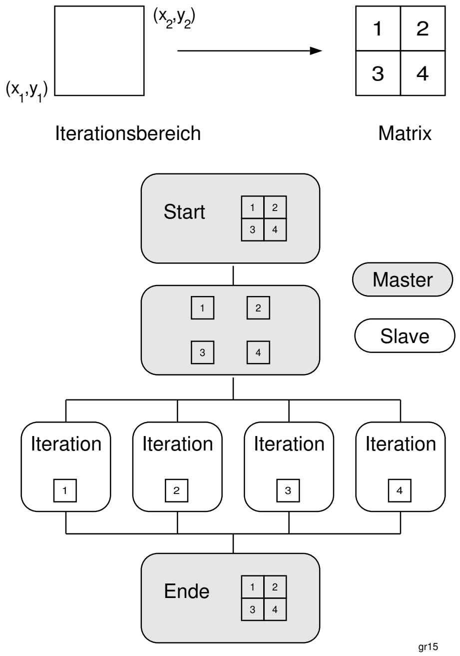
- *embarrassingly parallel*, d. h. einzelne Tasks sind praktisch unabhängig
- Lastbalancierung bei dynamischer Generierung der Prozesse möglich



Programmiertechnik

Crowd (2)

Implementierung: Mandelbrot

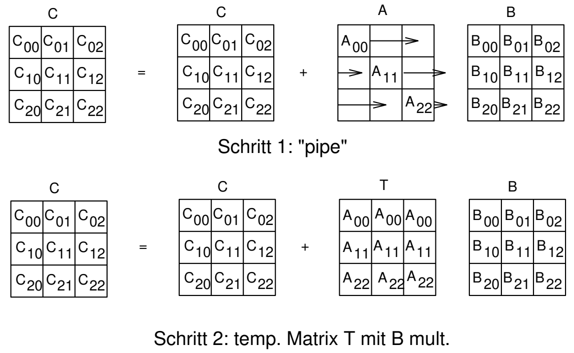


gr15

Programmiertechnik

Crowd (3)

Beispiel 2: Matrixmultiplikation $C = C + A \times B$



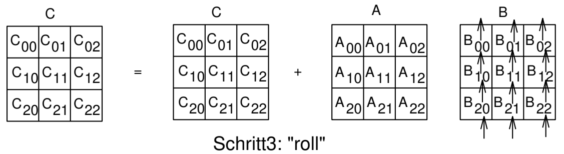
$C_{00}^{(1)} = A_{00} * B_{00}$	$C_{01}^{(1)} = A_{00} * B_{01}$	$C_{02}^{(1)} = A_{00} * B_{02}$
$C_{10}^{(1)} = A_{11} * B_{10}$	$C_{11}^{(1)} = A_{11} * B_{11}$	$C_{12}^{(1)} = A_{11} * B_{12}$
$C_{20}^{(1)} = A_{22} * B_{20}$	$C_{21}^{(1)} = A_{22} * B_{21}$	$C_{22}^{(1)} = A_{22} * B_{22}$

gr12

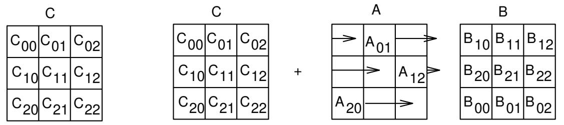
Programmiertechnik

Crowd (4)

Matrixmultiplikation (2)



Schritt3: "roll"



Schritt4: "pipe

$C^{(2)}_{00} = A_{00} * B_{00} + A_{01} * B_{10}$ $\quad$ $C^{(2)}_{01} = A_{00} * B_{01} + A_{01} * B_{11}$

$C^{(2)}_{10} = A_{11} * B_{10} + A_{12} * B_{20}$ $\quad$ $C^{(2)}_{11} = A_{11} * B_{11} + A_{12} * B_{21}$

$C^{(2)}_{20} = A_{22} * B_{20} + A_{20} * B_{00}$ $\quad$ $C^{(2)}_{21} = A_{22} * B_{21} + A_{20} * B_{01}$

$C^{(2)}_{02} = A_{00} * B_{02} + A_{01} * B_{12}$

$C^{(2)}_{12} = A_{11} * B_{12} + A_{12} * B_{22}$

$C^{(2)}_{22} = A_{22} * B_{22} + A_{20} * B_{02}$

gr13

Programmiertechnik

Crowd (5)

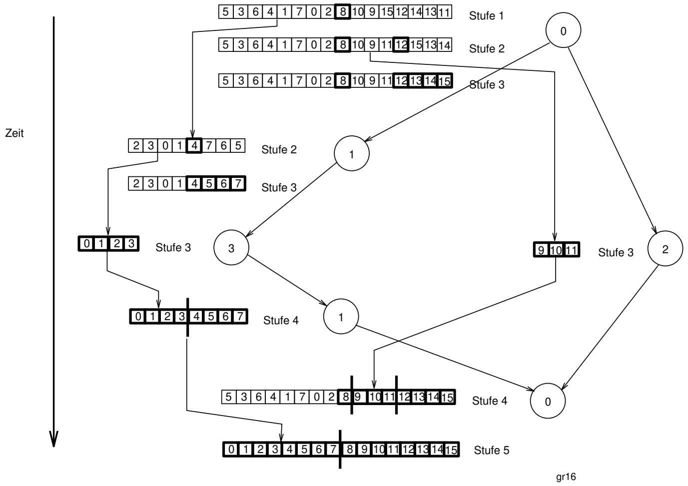
```
{Processor 0 starts up other processes}
if (<my processor number> = 0) then
  for i := 1 to MeshDimension*MeshDimension
    pvm_spawn(<component name>, . .)
  endfor
endif

forall processors Pij, 0 <= i,j < MeshDimension
  for k := 0 to MeshDimension-1
    {Pipe.}
    if myrow = (mycolumn+k) mod MeshDimension
      {Send A to all Pxy,x=myrow,y<>mycolumn}
      pvm_mcast((Pxy,x=myrow,y<>mycolumn),999)
    else
      pvm_recv(999)   {Receive A}
    endif
    {Multiply.  Running totals maintained in C.}
    Multiply(A,B,C)
    {Roll.}
    {Send B to Pxy, x=myrow-1, y=mycolumn}
    pvm_send((Pxy, x=myrow-1, y=mycolumn),888)
    pvm_recv(888)       {Receive B}
  endfor
endfor
```

Tree (1)

Sortieralgorithmus

- Liste wird bis zu einer gewissen Tiefe rekursiv aufgeteilt
- Teillisten werden sortiert und den Baum entlang wieder zusammengefügt



Tree (2)

Sortieralgorithmus (Teil 1)

```
{ Spawn and partition list based on
  a broadcast tree pattern. }
for i := 1 to N, such that 2^N = NumProcs
  forall processors P such that P < 2^i
    pvm_spawn(...) {process id P XOR 2^i}
    if P < 2^(i-1) then
      midpt: = PartitionList(list);
      {Send list[0..midpt] to P XOR 2^i}
      pvm_send((P XOR 2^i),999)
      list := list[midpt+1..MAXSIZE]
    else
      pvm_recv(999)    {receive the list}
    endif
  endfor
endfor
```

Tree (3)

Sortieralgorithmus (Teil 2)

```
{ Sort remaining list. }
Quicksort(list[midpt+1..MAXSIZE])

{ Gather/merge sorted sub-lists. }
for i := N downto 1, such that
                     2^N = NumProcs
  forall processors P such that P < 2^i
    if P > 2^(i-1) then
      pvm_send((P XOR 2^i),888)
      {Send list to P XOR 2^i}
    else
      pvm_recv(888)    {receive temp list}
      merge templist into list
    endif
  endfor
endfor
```


Programmiertechnik II

Verteilung der Arbeitslast

- Datenparalleler Ansatz
  - Das Programm soll auf speziellen Datenstrukturen operieren.
  - Diese Strukturen können unterteilt werden und in gewisser Weise unabhängig bearbeitetet werden
- Funktionelle Parallelisierung
  - Programmstruktur wird durch unterschiedliche Operationen oder Funktionen bestimmt (z.B. Input, Verarbeitung, Output, Nachbearbeitung)
  - Parallelisierung durch Verteilung der Funktionen

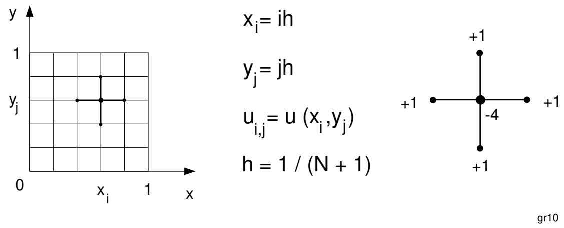
Portierung existierender Programme

- Sequentielles Programm
  - Wahl eines geeigneten Algorithmus
  - Wahl der Programmiertechnik (s.o.)
  - Wo bestehen Abhängigkeiten?
  - Aufteilung der Schleifen (Granularität: von Außen her)
  - Effizienz: Minimierung der Kommunikation
- shared memory parallel: Ersetze Synchronisations-Punkte durch Message Passing. Datenlayout anpassen
- distributed memory parallel: Kommunikationsroutinen ersetzen.

Bsp.: Part. Differentialgleichung (1)

Numerische Lösung der Gleichung

$$-u_{xx} - u_{yy} = f(x, y), \quad 0 < x, y < 1$$



Approximation durch Differenzenquotient

$$u_{xx}(x_i, y_i) = (u_{i-1,j} - 2u_{i,j} + u_{i+1,j})/h^2 + \text{err}$$

führt zu einer linearen Gleichung in jedem (inneren) Gitterpunkt:

$$4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = h^2 f_{i,j} + \text{err}$$

Bsp.: Part. Differentialgleichung (2)

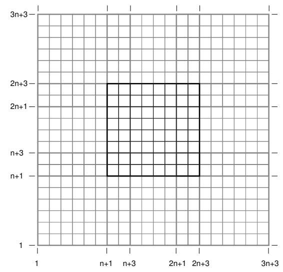
Lösung der linearen Gleichung durch iteratives Verfahren ($z_{i,j}^{(k)}$ ist Näherungswert der k-ten Iteration):

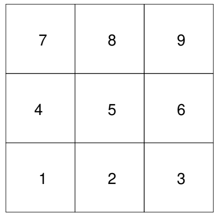
$$
\begin{aligned}
z_{i,j}^{(k+1)} &= [z_{i-1,j}^{(k)} + z_{i+1,j}^{(k)} + z_{i,j-1}^{(k)} + z_{i,j+1}^{(k)} \\
&\quad +h^2 f(x_i, y_j, z_{i,j}^{(k)})]/4
\end{aligned}
$$

Hintergrund (Jacobi-Iteration):

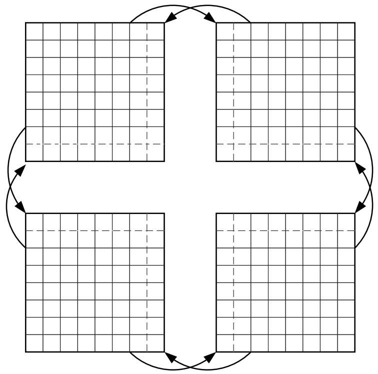
- Lineare Gleichung $Az = b$
- Iteration $Mz^{(k+1)} = Nz^{(k)} + b$ (M ist Diagonalmatrix ohne Nullelemente in der Diagonalen)
- zusätzliche Bedingungen an Matrizen N, M für Konvergenz
- für die Praxis nicht effizient genug

Gebietszerlegung





Kommunikation



gr11

Bsp.: Part. Differentialgleichung

Node: Nachbarknoten bestimmen

```
c - data(1) : Anzahl der Proz.
c - data(2) : SQR(Anzahl der Proz.)
c - data(3) : Eigene Nummer
    mynum = data(3)
    if (mynum .le. data(2)) then
       nexts = 0
    else
       nexts = mynum - data(2)
    endif
    if ((mynum + data(2)) .gt. data(1)) then
       nextn = 0
    else
       nextn = data(2) + mynum
    endif
    if (mod(mynum,data(2)) .eq. 0) then
       nexte = 0
    else
       nexte = mynum + 1
    endif
    if (mod(mynum,data(2)) .eq. 1) then
       nextw = 0
    else
       nextw = mynum - 1
    endif
```


Bsp.: Part. Differentialgleichung

Node: Iterationsschleife

```
c - boundary values
   call bound(Phi1,Phi2,ixlo,ixhi,iylo,iyhi,
  &           nextn,nexte,nexts,nextw,dx)
c - iteration
   do 10 ii = 1,iter/2
      call comp(Phi1,Phi2,ixlo,ixhi,iylo,iyhi,
                dx2)
      call commnode(Phi2,buf)
      call comp(Phi2,Phi1,ixlo,ixhi,iylo,iyhi,
                dx2)
      if (data(6) .eq. 1) then
        call commhost(Phi1,ixlo,ixhi,iylo,iyhi,
                      ii,hostnum)
      endif
      call commnode(Phi1,buf)
10 continue
   call commhost(Phi1,ixlo,ixhi,iylo,iyhi,
                 ii,hostnum)
```

`Phi1,Phi2`:	Näherungswerte (alt, neu)
`ixlo,ixhi,iylo,iyhi`:	Randkoordinaten
`buf`:	Send-Receive-Puffer

Bsp.: Part. Differentialgleichung

Node: Rechenkern

```
   subroutine comp(Phi1,Phi2,ixlo,ixhi,
                  iylo,iyhi,dx2)
   implicit none
   integer ixlo,ixhi,iylo,iyhi
   real*8 Phi1(ixlo:ixhi,iylo:iyhi)
   real*8 Phi2(ixlo:ixhi,iylo:iyhi),dx2
   integer jx, jy
   external source

c - compute intermediate data
   do 25 jx=ixlo+1,ixhi-1
     do 25 jy=iylo+1,iyhi-1
             Phi2(jx,jy)= 0.25 *(
  &        Phi1(jx-1,jy)+Phi1(jx+1,jy)+
  &        Phi1(jx,jy-1)+Phi1(jx,jy+1)
  &        -source(jx,jy)*dx2)
25 continue
   return
   end
```


Bsp.: Part. Differentialgleichung

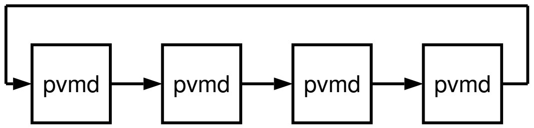
Kommunikation Node $\leftrightarrow$ Node

```
c - Kommunikation nach Norden
   if (nextn .ne. 0) then
      do 30 jx =  1, NGRID + 3
         buf(jx) = Phi(jx, NGRID + 1)
30    continue
      call pvmfinitsend(PVMTASKDEFAULT,info)
      call pvmfpack(REAL8,buf,NGRID+3,1,info)
      msgtype  = 1
      print *,'SEND FROM', mynum,inst(mynum),
  &            'TO',nextn,inst(nextn)
      call pvmfsend(inst(nextn),msgtype,info)
   endif
   if (nexts .ne. 0) then
      msgtype  = 1
      print *,'RECEIVE',mynum,inst(mynum)
      call pvmfrecv( -1, msgtype, info )
      call pvmfunpack(REAL8,buf,NGRID+3,1,info)
      do 31 jx =  1, NGRID + 3
         Phi(jx, 1) = buf(jx)
31    continue
   endif
c - Kommunikation nach Sued, West, Ost...
```

Andere MP-Dialekte

PVM ↔ MPI

- `pvm_send` nicht einfach durch `mpi_send()` ersetzen! `mpi_send()` wartet immer auf korrespondierendes *receive*.



gr17

- Bei `mpi_bsend()` muß man selbst für die Verwaltung der Puffer sorgen.
- Bei `mpi_rsend()` sendet man aus dem eigenen Adreßraum

Debugging

Voraussetzung: Erfahrung mit dem Debugging von seriellen Programmen (z.B. `dbx`, `xdbx`). Debugging von seriellen Programm ist eine Wohltat im Vergleich zum Debugging von parallelen Programmen!

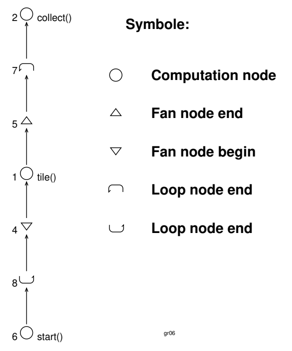
- Der `spawn`-Aufruf wird mit der Option `PvmTaskDebug` versehen
- Beim Programmstart werden durch `spawn` initiierte Tasks in einem `xterm` mit Debugger (Voreinstellung `dbx`) gestartet.
- Skript `$PVM_ROOT/lib/debugger` kann mit der Option `bx=` im Hostfile durch ein eigenes Skript ersetzt werden (damit können andere Debugger eingesetzt werden, z.B. `xdbx`)
- `pvm_setopt` hat Option die Mitprotokollierung von Debug-Daten in `/tmp/pvml.<uid>` zu ermöglichen

Debugging in 3 Stufen

1. Wenn möglich sollte das Programm zunächst in einer Version mit nur einer Task gestartet werden. (logische- oder Index-Fehler, die von der Parallelität unabhängig sind).
2. Ein Durchlauf des Programmes mit wenigen Tasks (2-4) auf einem Rechner. Mit entsprechenden Optionen in `pvm_setopt` können z.B. *message tags* in `/tmp/pvml.<uid>` untersucht werden. (Kommunikationsverhalten)
3. Ein Lauf des Programmes mit wenigen Tasks (2-4), aber auf mehreren Rechnern (auch unterschiedliche Rechnertypen). (durch das Netzwerk indizierte Fehler wie, Timeout etc. Fehler bei der Datenkonvertierung bei unterschiedlichen Rechnertypen.)

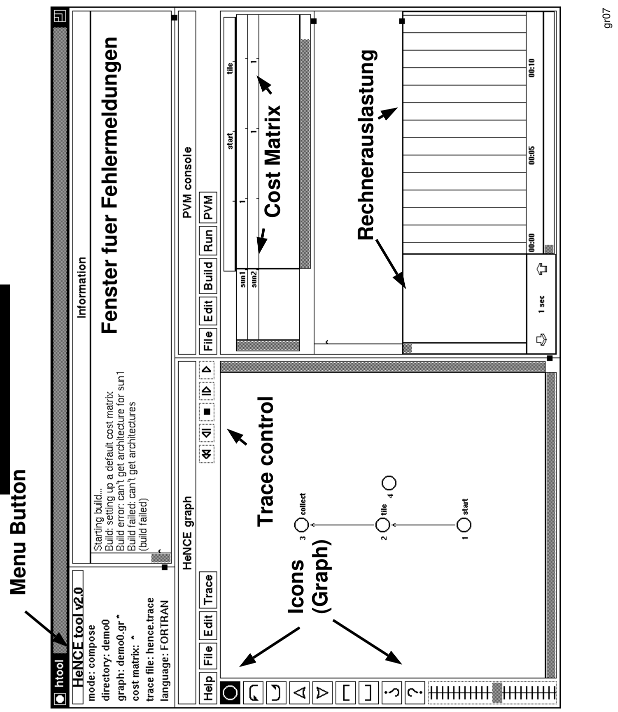
Tools: HeNCE

- graphische Oberfläche unter X-Window
- Parallelität einer Anwendung wird durch Graph definiert



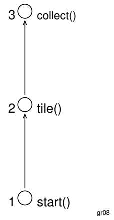
- Abhängigkeit der einzelnen Routinen wird durch Makrosprache definiert
- PVM-Direktiven werden automatisch eingestreut
- Ablauf der Applikation kann analysiert werden

HeNCE (2)



HeNCE - Beispiel

- Beispiel: Mandelbrot `mandlebrot`
  in `/sw/share/hence/hence-examples`



- Definition von Ein- Ausgabe-Parametern:

```
PROGRAM
NODE [ 218 347 ] 1
        NEW <> int height=256;
        NEW <> char pix[width][height];
        NEW <> int width=256;
        start(pix, width, height);
NODE [ 218 231 ] 2
        tile(pix, -2.000000, -2.000000,
2.000000, 2.000000, width, height);
NODE [ 218 115 ] 3
        collect(pix, width, height);
ARC 1 2
ARC 2 3
```

HeNCE - Beispiel (2)

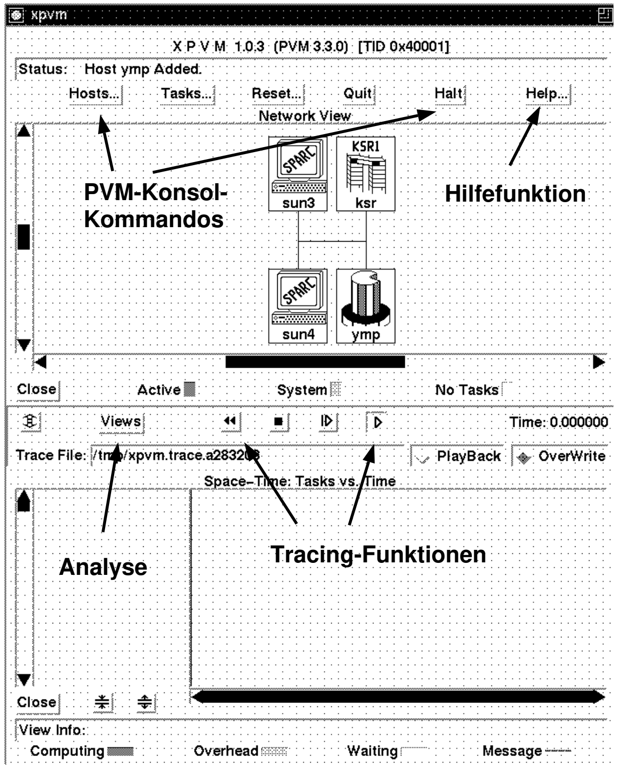
- Der Benutzer stellt Routinen zur Verfügung (C oder Fortran). `start.c`:

```
#include <stdio.h>

void
start (array, width, height)
char *array;
int width;
int height;
{
    fprintf (stderr,"w = %d h = %d\n",
             width, height);
}
```

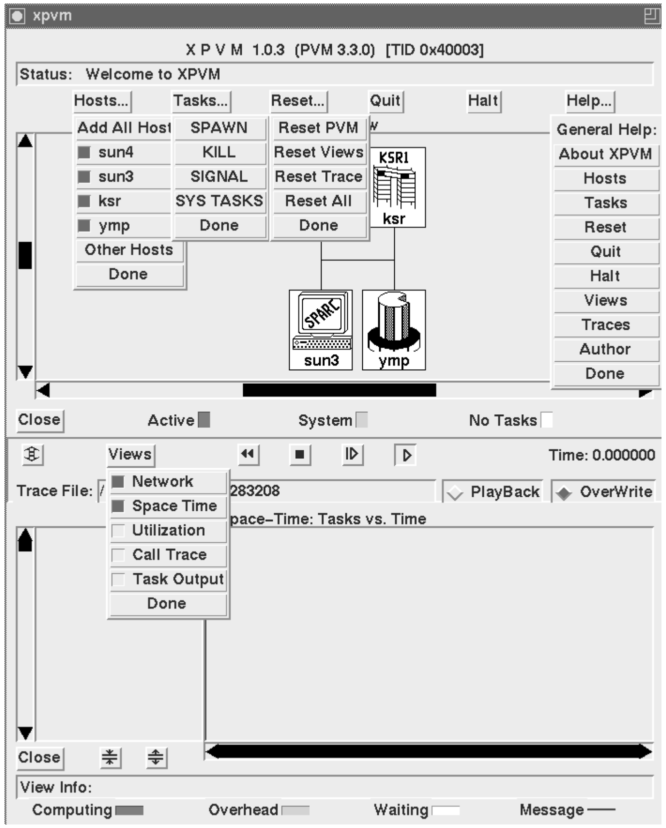
- HeNCE am LRZ:
  - Nur am Sun-Cluster
  - Aufruf: `hence`
  - Doku: `HeNCE-2.0-doc.ps` in `/sw/share/hence/HeNCE-2.0-doc.ps`
  - weitere Beispiele in `/sw/share/hence/hence-examples/`

XPVM (1)

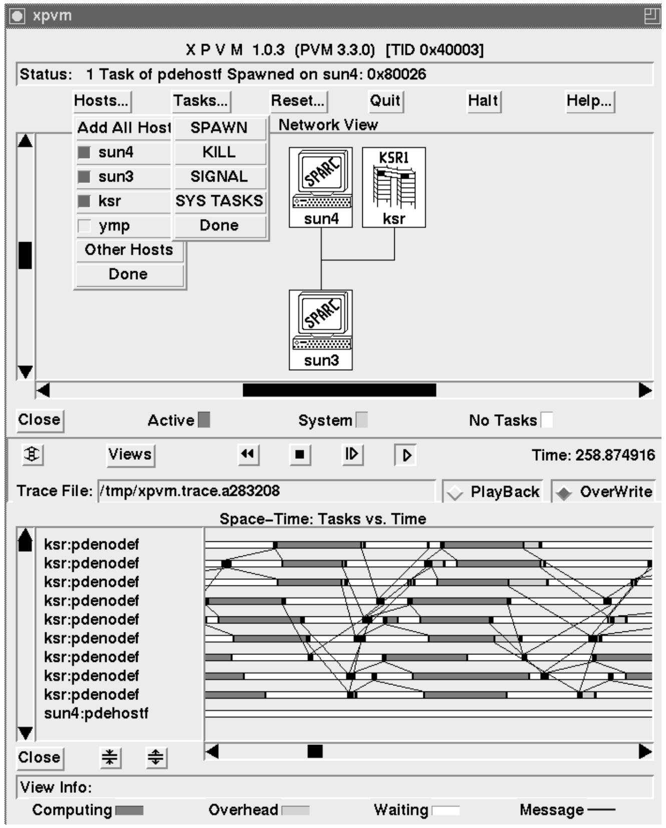


gr09

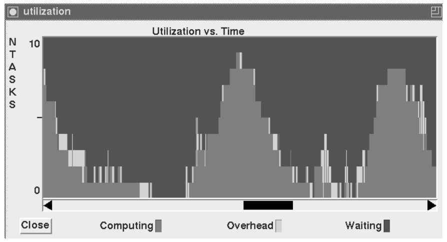
XPVM (2)

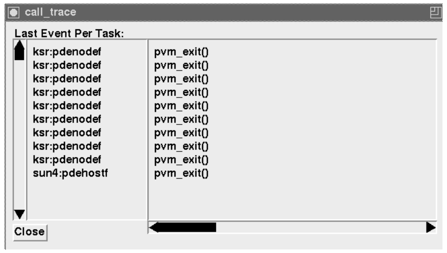

xpvm
X P V M 1.0.3 (PVM 3.3.0) [TID 0x40003]
Status: Welcome to XPVM
Hosts...
Tasks...
Reset...
Quit
Halt
Help...
Add All Host
sun4
sun3
ksr
ymp
Other Hosts
Done
SPAWN
KILL
SIGNAL
SYS TASKS
Done
Reset PVM
Reset Views
Reset Trace
Reset All
Done
General Help:
About XPVM
Hosts
Tasks
Reset
Quit
Halt
Views
Traces
Author
Done
KSR1
ksr
SPARC
sun3
ymp
Close
Active
System
No Tasks
Views
Time: 0.000000
Trace File:
283208
PlayBack
OverWrite
Network
Space Time
Utilization
Call Trace
Task Output
Done
pace–Time: Tasks vs. Time
Close
View Info:
Computing
Overhead
Waiting
Message

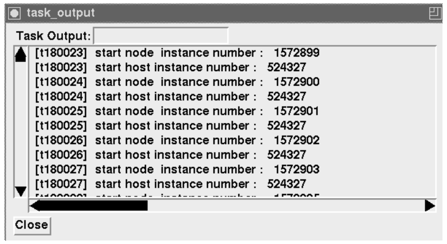
XPVM (3)


xpvm
X P V M 1.0.3 (PVM 3.3.0) [TID 0x40003]
Status: 1 Task of pdehostf Spawned on sun4: 0x80026
Hosts...
Tasks...
Reset...
Quit
Halt
Help...
Add All Host
sun4
sun3
ksr
ymp
Other Hosts
Done
SPAWN
KILL
SIGNAL
SYS TASKS
Done
Network View
SPARC
sun4
KSR1
ksr
sun3
Close
Active
System
No Tasks
Views
Time: 258.874916
Trace File: /tmp/xpvm.trace.a283208
PlayBack
OverWrite
Space–Time: Tasks vs. Time
ksr:pdenodef
ksr:pdenodef
ksr:pdenodef
ksr:pdenodef
ksr:pdenodef
ksr:pdenodef
ksr:pdenodef
ksr:pdenodef
ksr:pdenodef
sun4:pdehostf
Close
View Info:
Computing
Overhead
Waiting
Message

XPVM (4)







Tips

- Vorsicht bei I/O auf gleichnamige Dateien bei gemeinsamen Filesystemen (NFS, AFS)
- Die Dämonen sollten die gleiche Versionsnummer haben
- Bei Versions-Upgrade: PVM-Programme neu übersetzen (geänderte Include-Files)
- PVM-Rückgabe im Programm abfragen (Fehlervermeidung)
- Vorsicht mit den Datentypen, wenn unterschiedliche Rechnertypen eingesetzt werden

Internals

Messages

- Länge der Nachrichten ist beliebig
- Konvertierung mit XDR bei inkompatiblen Datenformaten
- Keine Quittung vom Empfänger (*asynchrones Message Passing*)
- Auslieferung von Messages ist zuverlässig (wenn Ziel existiert). Daten werden gepuffert, bis sie ,,empfangen" werden (solange Speicherplatz da ist)
- Blocking ist möglich
- Reihenfolge der Messages bleibt erhalten (bei point-to-point Kommunikation)

Internals

Fehlertoleranz

- Messages können verloren gehen bei Crash oder Netzfehler
- Also: Das Programm muß auf diese Situation vorbereitet sein:
  - Start eines neuen Servers und/oder Task
  - Message neu aufsetzen
- Notification (`pvm_notify()`): Eine Task kann vom Dämon Nachricht erhalten bei:
  - Crash oder Exit eines anderen Tasks
  - dynamischer Erweiterung (`add`)
- Semi-synchrones Message Passing:
  - $T_A \xrightarrow{\text{Mess.}} T_B$, $T_A \xleftarrow{\text{Quittung}} T_B$,
  - T_A wartet nach `send` auf Notification (=Fehler) oder Quittung (=o.k.)

Internals

Der Dämon

- Ein Dämon pro Host, Benutzer
- Message Routing, Prozeß-Kontrolle, Fehlererkennung, Überprüfung der Verbindungen
- Master-Slave Konzept bei mehreren Dämons (Verwaltung beim Master)
- Kommunikation mit Tasks oder anderen Dämons über Sockets
- Start: Konfiguration durch `hostfile` (optional).
  Error log: `/tmp/pvml.`*uid*
- Shutdown: Signal an eigene Tasks, Message an andere Dämons
- Slave-Pvmd: Shutdown, wenn Verbindung zum Master abbricht (verhindert Aufspaltung)

Internals

Start eines Slave-Pvmds

- Intern: `rsh`, `rexec()`, manuell
  ohne Passwort: Eintrag in `.rhosts` nötig
  mit Passwort:
  - `rexec()`: Hostfile-Eintrag:
    `so=pw`
  - Manueller Start (wenn `rsh`, `rexec()` nicht verfügbar sind): Hostfile-Eintrag:
    `so=ms`
    Beim Start `login ...`
- Ein Task ruft `pvm_addhosts` auf:
  - Master wird verständigt, liest gegebenfalls Voreinstellungen aus Hostfile
  - Fork eines Schatten-Dämons Pvmd’, der die Arbeit übernimmt
  - Komplexes Hin -und Her von Senden und Quittieren, bis der Slave funktionsfähig eingebunden ist, und alle anderen Slaves über die neue Konfiguration informiert sind

Internals

Resource Manager

- verantwortlich für Erzeugung und Scheduling von Tasks und Hosts
- verwendet Informationen über Lastverteilung im Cluster, CPU-Leistung etc.
- Seit Version 3.3: Benutzer kann eigenen RM einbinden (`pvm_reg_rm()`)
- beeinflußbar sind: `pvm_addhosts`, `pvm_delhosts`, `pvm_spawn`
- die einmal gewählte Verteilung von Tasks ist statisch
- dynamische Verteilung: MPVM und UPVM (J. Casas et al: Adaptive Load Migration systems for PVM, Supercomputing 1994)

Internals

Environment-Variablen

- `PVM_ROOT`: Wurzel der PVM-Installation
- `PVM_EXPORT`: Variablen die bei Spawn weitervererbt werden
  (Beispiel: `PVM_EXPORT=DISPLAY:SHELL`)
- `PVM_DPATH`: Installationspfad für Slave-Pvmds
- `PVM_DEBUGGER`: Pfad des Debugger-Scripts (Spawn)
  Alternative zu `bx`-Option im Hostfile
- `PVM_ARCH`: PVM-Architektur-Bezeichnung

AFS-Zugriff zum LRZ

Die LRZ-Installation kann von der lokalen WS ohne Login am LRZ eingesetzt werden!

- Zugriff auf die AFS-Zelle `lrz-muenchen.de` muß möglich sein
- `$PVM_ROOT=/afs/lrz-muenchen.de/sw/share/pvm3`
- `$PVM_HOME` auf das lokale PVM-Homeverzeichnis setzen
- Start des Dämonen: `$PVM_ROOT/pvm3.setup`
- Alternativ: Start mit Hostfile
  `dx= ...` (setzt `$PVM_ROOT`)
  `ep= ...` (setzt `$PVM_HOME`)

- SUN4, HPPA, RS6K, SGI5, CRAY

Internals

Beschränkung der Resourcen

- Aus Sicht des Dämons benötigt jede Task 2 Datei-Deskriptoren.
  (⇒ Beschränkung für die Zahl der Tasks (LRZ-Sun: ca 10 Tasks pro Host)
- Direct Routing: Pro Verbindung 1 Datei-Deskriptor
  (⇒ Beschränkung für die Zahl der Direktverbindungen)
- Pvmd allokiert dynamisch Speicher für die Pufferung der Messages.
  (⇒ bei einem Stau kann der Speicher überlaufen)
- Maximale Größe einer Message ≈ halber verfügbarer Speicher
  (wg. Platzbedarf beim Entpacken)

Protokolle

- TCP (verbindungsorientiertes Protokoll, Datei-Deskriptor bei Empfänger und Sender)
- UDP (packetorientiertes Protokoll, Sendeadresse im Paket enthalten)
- pvmd $\leftrightarrow$ pvmd (über Netz): UDP (bessere Skalierbarkeit, weniger Overhead, bessere Fehlertoleranz durch Quittieren)
- pvmd $\leftrightarrow$ Task, Task $\leftrightarrow$ Task (local): TCP (lokal kein Datenverlust)
- Ab Version 3.3: Lokale Kommunikation durch UNIX stream sockets (Beschleunigung um Faktor 2)
- Direct Message Routing: Task $\leftrightarrow$ Task über direkte TCP-Verbindung (vermeidet pvmd)
- Multicast: Fanout (kein Baum! Crash eines Knoten würde zu Verlust von Messages führen)

Literatur: Leffler, et al.: BSD UNIX Operating System