Tips und Tricks zur Parallelprogrammierung

(1) Alle Aktionen, die möglich sind, können fehlschlagen.

(2) Wenn Aktionen fehlschlagen, sollte man dies erkennen können.

(3) Programmläufe, die nicht korrekt terminieren, können nachfolgende beeinflussen

- ➡ Kontrolle, ob noch irgendwo Prozess unkontrollliert laufen.
- ➡ Testläufe auf wenigen (1-2) Prozessoren durchführen

(4) Soviel Strukturierung wie nur irgend möglich !!!

Entwurf des Programmrumpfes (Prozeß-Verwaltung und Kommunikation)

↓

Implementierung und Test der Einzelkomponenten mit Testdaten

↓

Zusammenfügen der Einzelkomponenten und Test des Programmrumpfes

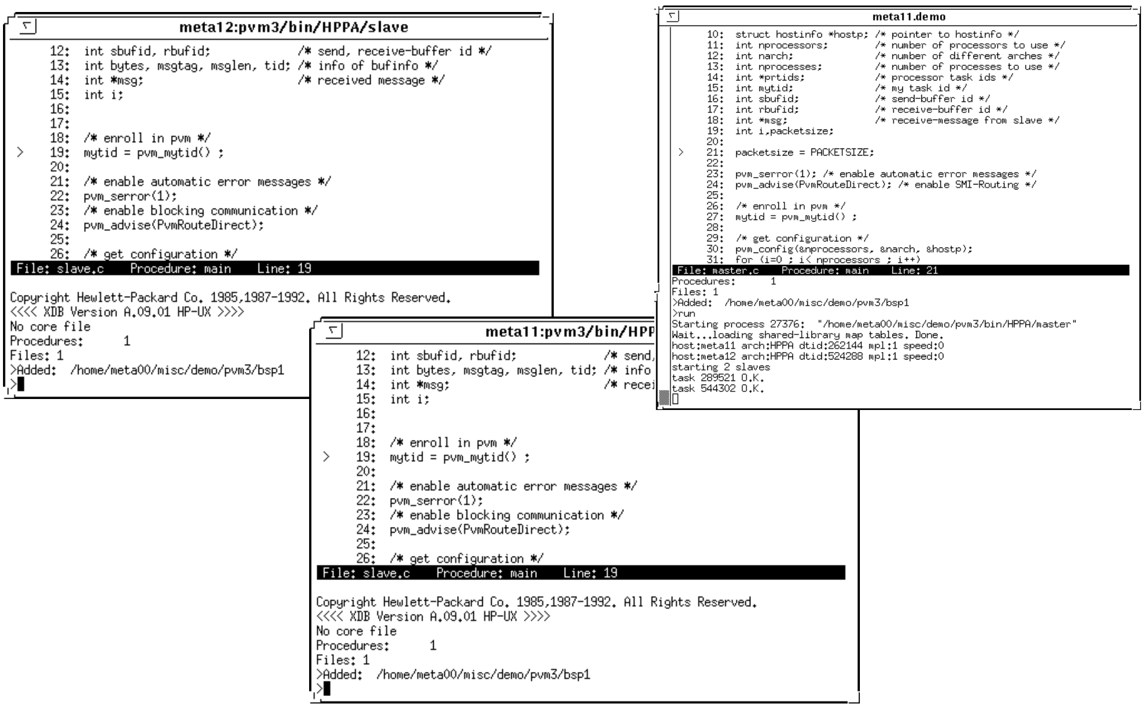
↓

Einfügen der "Numerik"

Debugging

- ❏ Am Besten gar nicht !
- ❏ Multi-Window-Debugging
  - Pro Prozeß ein xterm, in dem der Prozeß unter xdb-Debugger Kontrolle läuft
  - `pvm_spawn(task, argv, PvmTaskDebug, where, ntask, tids)`
  - Compiler-Option: -g (HP) , -cxdb (SPP)
  - Anlegen von **`.xdbrc`** bzw. **`.cxdbinit !`**
- ❏ In der Implementierungsphase:
  - Abfrage aller(!) Rückgabewerte und Ausgabe der genauer Ortsangabe, wenn der Rückgabewert nicht der ist, der er sein müßte.
  - Ständige Kontrollausgaben über den Zustand der einzelnen Prozesse.
- ❏ Modularisierung: Einsatz von Komponenten, die vorher genau mit Testdaten geprüft wurden.

Multi-Window-Debugging unter PVM3



Prozeß-Verwaltung I

❏ Jeder gestartete Prozeß muß sich bei der PVM-Maschine anmelden

```
mytid = pvm_mytid() ;
if (mytid < 0) {
    fprintf(stderr,"kein pvmd3 vorhanden\n");
    pvm_exit();
    exit(1);
   }
```

❏ Jeder Prozeß, den man glaubt gestartet zu haben, sollte überprüft werden

```
sprocesses = pvm_spawn("slave",(char**)0,PvmTaskDefault, "",
                        nprocesses, prtids);
if (sprocesses < nprocesses) {
    for (i = 0; i < nprocesses; i++){
        if (pvm_pstat(prtids[i])!= PvmOk)
           fprintf(stderr,"Prozess %d nicht gestartet\n",i);
        else pvm_kill(prtids[i]);
        }
    pvm_exit(); exit(1);
   }
```

Prozeß-Verwaltung II

❏ Der "Vaterprozeß" sollte erkennen können, wenn seine Söhne terminieren.

```
/* Dies ist der Sohnprozess */
#define ICH_HOEHRE_AUF 99
parent = pvm_parent();
pvm_notify(PvmTaskExit, ICH_HOERE_AUF, 1, &parent);
...
pvm_exit();


/* Dies ist der Vaterprozess */
#define ICH_HOERE_AUF 99
while ((bufid = pvm_nrecv(-1,ICH_HOEHRE_AUF)) > 0) {
       pvm_bufinfo(bufid, &bytes, &msgtag, &tid);
       printf("Prozess %x ist terminiert\n",tid);
      }
```


Prozeß-Verwaltung III

❏ Prozeßgruppen sollten mit einem für die Gruppen eindeutigen Namen gebildet werden

```
/* Dies ist der Vaterprozess */
char groupname[256];
mytid = pvm_mytid();
sprintf(groupname,"group1%x",mytid);
pvm_joingroup(groupname);
pvm_barrier(groupname,pvm_gsize(groupname));
pvm_bcast(groupname, msgtag);
pvm_lvgroup(groupname);

/* Dies ist der Sohnprozess*/
char groupname[256];
parent= pvm_parent();
sprintf(groupname,"group1%x",parent);
pvm_joingroup(groupname);
pvm_barrier(groupname,pvm_gsize(groupname));
pvm_lvgroup(groupname);
```

❏ **Achtung: Sind alle Prozesse, die in der Gruppe sein sollten, schon darin?**

Prozeß-Start im Subcomplex (Convex SPP)

❏ Wieviele Prozessoren sind im Subcomplex?

```
int get_num_procs()
{
 cnx_is_target_data_t target;
 cnx_is_thread_location_info_data_t     thread_location_info;
 cnx_is_sc_basic_info_data_t            sc_info;
 int entries, sc_id, pid;
 cnx_tid_t tid;

 pid = getpid(); tid = cnx_thread_self();
 cnx_sysinfo_target_thread(&target,pid,tid);
 entries = cnx_sysinfo (CNX_IS_THREAD_LOCATION_INFO,
                        &target, &thread_location_info, 1 ,
                        CNX_IS_THREAD_LOCATION_INFO_COUNT,0);
 cnx_sysinfo_target_subcomplex(&target,thread_location_info.scid);
 cnx_sysinfo (CNX_IS_SC_BASIC_INFO, &target, &sc_info, 1 ,
              CNX_IS_SC_BASIC_INFO_COUNT,0);
 return(sc_info.cpu_count);
}
```

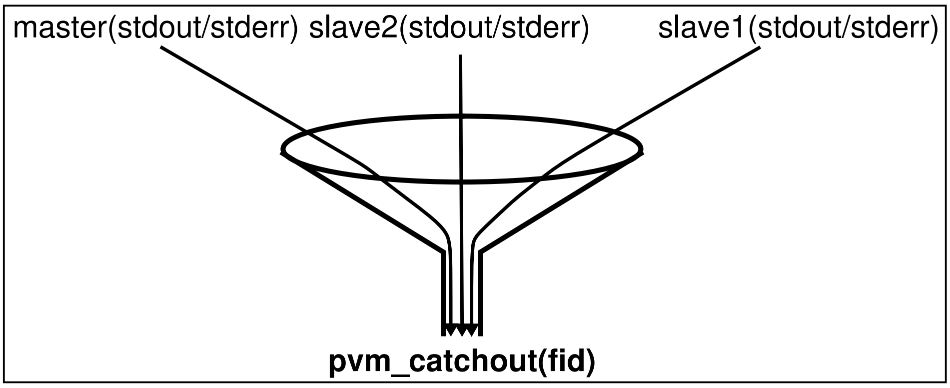

Ein/Ausgabe

❏ Die Ausgabe (**`stdout/stderr`**) des Masters wird auf stdout/stderr ausgegeben

❏ Die Ausgabe (stdout/st**`derr`**) eines Prozesses erfolgt

- lokal in die Datei /**`tmp/pvml.uid[.sc] oder`**
- in die Datei, die vom Parent-Process mit pvm_catchout angegeben wurde.

❏ **Die Ausgaben erfolgt nicht immer in der richtigen zeitlichen Reihenfolge !!**



Botschaftenaustausch I

❏ Unterschiedliche Architektur ➡ Unterschiedliche Zahlendarstellung

pvm_(u)pk.. - Funktion mit Angabe des Datentyps ermöglicht eine Konvertierung der Zahlendarstellung beim Botschaftenaustausch.

`pvm_initsend(PvmDataDefault)`: Konvertierung bei heterogener Architektur

`pvm_initsend(PvmDataRaw)`: keinerlei Konvertierung der Zahlendarstellung

❏ Verdoppelung der Datenmenge beim Versenden und Empfangen:

- Kopieren der Daten in den Sendepuffer (pvm_pk..)
- Kopieren der Daten aus dem Empfangspuffer (pvm_upk..)

Im CIP-Cluster darf ein Benutzerprozeß maximal 64 MB an Daten halten. Daher ist das Versenden einer 32 MBytes Matrix mit einer send-Operation nicht möglich.

➡ Aufteilen in kleiner Pakete, die hintereinander verschickt werden.

❏ Auf Shared Memory Systemen ist der Puffer-Bereich limitert:

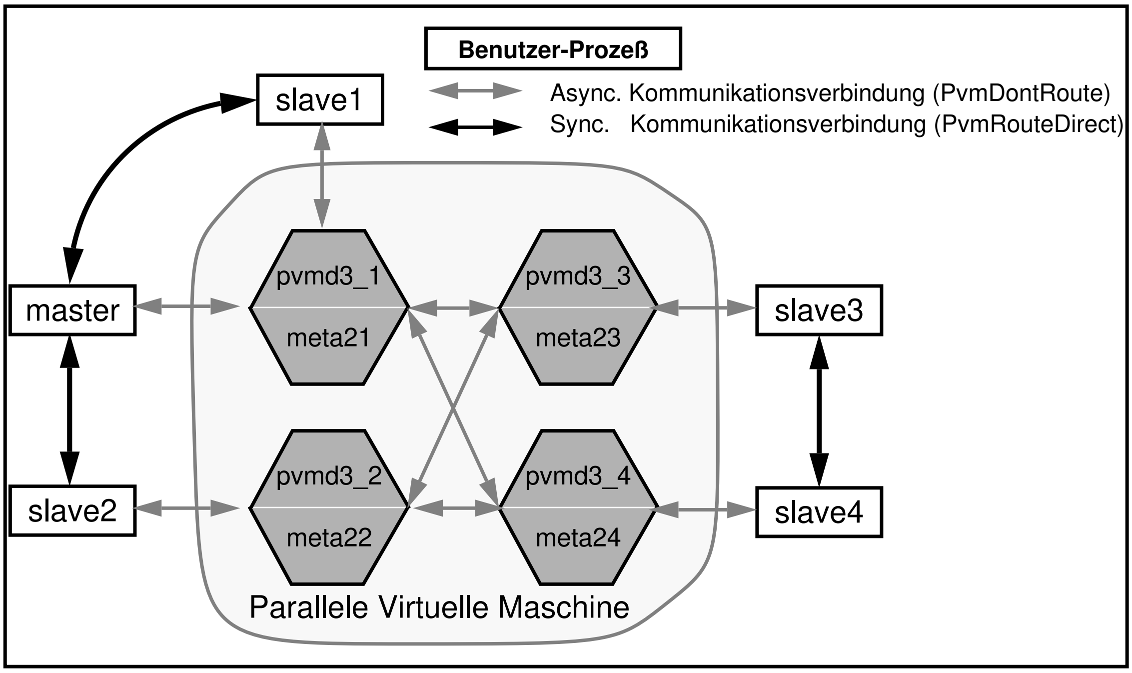
```
libpvm [t4001a]: da_new: shared buff full
libpvm [t4001a]: i_free: scribbled in 0x817728[-8]
```

➡ Verklemmung der Applikation möglich

```
setenv PVMBUFSIZE 10485760
```


Botschaftenaustausch II

❏ Asynchrone und synchrone Verbindung (im Cluster)



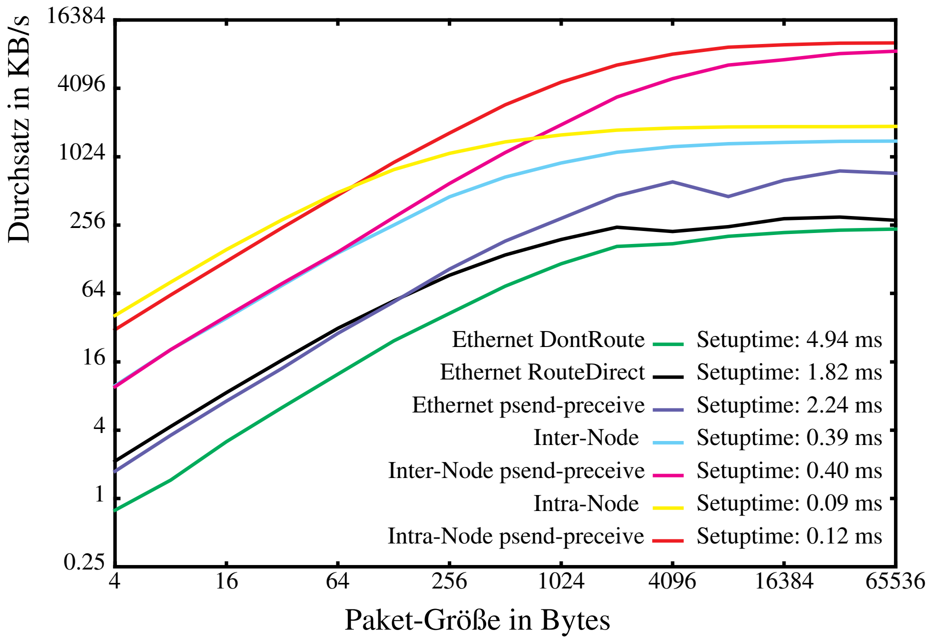
Botschaftenaustausch III

❏ Senden ohne Verpacken ist möglich und effizient!

- `pvm_psend(tid, tag, buf, len, datatype);`
- `pvm_preceive(tid,tag,buf,len,datatype,atid,atag,alen)`

❏

PVM-Bandbreite im HP-Cluster / SPP



Strukturierung von parallelen Programmen I

- Kapselung von zusammengehörenden Daten in Strukturen
- Funktionen zum Senden/Empfangen von Strukturen

```
#define MATRIX 9999
typedef struct {
                 double *feld;
                 int zeilen, spalten;
                } matrix_typ;
...
int sende_matrix(matrix,tid)
int *tid;
matrix_typ *matrix;
{
 pvm_initsend(DataDefault);
 pvm_pkint(&(matrix->zeilen),1,1);
 pvm_pkint(&(matrix->spalten),1,1);
 pvm_pkdouble(matrix->feld,matrix->zeilen*matrix->spalten,1);
 return(pvm_send(*tid,MATRIX));
}
```


Strukturierung von parallelen Programmen II

```
matrix_typ empfange_matrix(tid)
int tid;
{
 matrix_typ matrix;
 int elements;

 pvm_recv(tid,MATRIX);
 pvm_upkint(&matrix.zeilen,1,1);
 pvm_upkint(&matrix.spalten,1,1);
 elements = matrix.zeilen*matrix.spalten;
 matrix.feld = (double *)malloc(elements*sizeof(double));
 pvm_upkdouble(matrix.feld,elements,1);
 return(matrix);
}
```

Strukturierung von parallelen Programmen III

❏ Vorsicht beim Byte-weisen Versenden von Strukturen (keine Konvertierung !)

```
#define STRUKTUR 8888
typedef struct {
                 int    izahl;
                 double dzahl;
                } strukturtyp;

int sende_struktur(struktur,tid)
stuktur_typ *struktur;
int tid;
{
 pvm_initsend(PvmDataDefault);
 pvm_pkbyte((char *)struktur,sizeof(strukturtyp),1);
 return(pvm_send(tid,STRUKTUR));
}
```

Vorteile des Byte-weisen Versendens von Strukturen:

- Sehr effizient, da wenige "Pack"-Operationen erfolgen
- Sende/Empfangs-Funktion ist unabhängig von der Strukturdefinition.

Strukturierung von parallelen Programmen IV

❏ Der Sender darf nicht terminieren bevor alle seine Botschaften gelesen wurden!

```
master()
{
      spawn(slaves);
      send_work_to_slave;
      receive(results);
      barrier();
}

slave()
{
      receive(task);
      work;
      send(result);
      barrier();
}
```

Verwendung von #include-Files

- Definition von mehrfach verwendeten Datentypen in #include-Files

```
/* structs.h */
#define STRUKTUR 8888
typedef struct {
                 int    izahl;
                 double dzahl;
                } strukturtyp;
```

- Aufbau eines Makefile.aimk

```
# makefile for master-slave-example
PVM_LIB = -L$(PVM_ROOT)/lib/$(PVM_ARCH) -lgpvm3 -lpvm3
XDIR = ../../bin/$(PVM_ARCH)
VPATH= ..
COMPILE = $(CC) $(CFLAGS) -o $@ $< $(PVM_LIB) -lm ; cp $@ $(XDIR)
all: master slave
master: master.c structs.h
      $(COMPILE
slave: slave.c structs.h
      $(COMPILE)
```