Interface de programmation (environ 80 fonctions)

Manipulation des buffers de messages	Création, destruction des buffers compression/décompression de données
Passage de messages	Envois, réception, multicast
Contôle des processus	Création, destruction des tâches, table des tâches, propre tid, tid de la tâche parent
Gestion dynamique de groupe de processus	*avec la librairie de gestion de groupe,* adhérer, quitter un groupe, carte des membres d'un groupe, Broadcast, réduction globale
Configuration de la machine virtuelle	Ajout, retrait d'une machine, requête pour l'état d'une machine, démarrage et arrêt de la machine virtuelle
Divers	Obtention, établissement des options, notification de requête, récupération des offsets sur les horloges des machines

Contrôle de processus (1)

- int numt = pvm_spawn(char *task, char ** argv, int flags, char *where, int ntask, int *tids) : démarre une nouvelle tâche
  - *task : par défaut c'est $HOME/pvm3/bin/$PVM_ARCH/filename pour une configuration plus spécifique il faut utiliser l'option ep= du fichier de machine (le hostfile)*
  - *argv : tableau d'argument pour l'exécutable (par défaut les fichiers de données sont recherchés dans $HOME)*
  - *flags :*

0	*PvmTaskDefault*	*Round-robin*
1	*PvmTaskHost*	*machine spécifiée (“.” est locale)*
2	*PvmTaskArch*	*classe d'architecture nommée*
4	*PvmTaskDebug*	*le processus démarre en mode “debug”*
8	*PvmTaskTrace*	*le processus génère les données de trace*
16	*PvmMPPFront*	*démarre le processus sur le nœud frontal d'une machine //*
32	*PvmHostCompl*	*utilisation de l'ensemble de machines complémentaires //*

Contrôle de processus (2)

- int tid = pvm_mytids(void) : numéro d'identificateur de tâche (tid)
- int info = pvm_parent(void) : tid de la tâche mère
- int info = pvm_exit(void) : sortir proprement de pvm
- int info = pvm_kill(int tid) : termine la tâche pvm de numéro tid
- int info = pvm_stat(int tid) : état de la tâche tid

Communication de base en PVM (1)

- Envoi de message en 3 étapes :
  - int bufid = pvm_initsend(int encoding) : initialisation du buffer, RAZ du buffer courant, encoding :

PvmDataDefault	0	XDR
PvmDataRaw	1	pas d'encodage
PvmDataInplace	2	données restent sur place (explication)

  - int info = pvm_pk<type_de_données> (<data_type> * data, int num_items, int stride) : compactage des données selon différents formats
  - int info pvm_send(int tid, int msgtag)
    int info pvm_mcast(int * tids, int ncast, int msgtag) : envoi du buffer à la (aux) tâche(s)

Communication de base en PVM (2)

- Réception (deux étapes) :
  - int bufid = pvm_recv(int tid, int msgtag)
    int bufid = pvm_nrecv(int tid, int msgtag)
    réception bloquante et non bloquante
  - int info = pvm_up<type_de_données>(<data_type> * data, int num_item, int stride)
    décompression des messages dans les variables de l'utilisateur
- int bufid = pvm_probe(int tid, int msgtag)
  pour vérifier si un message de label msgtag de la tâche tid est arrivé. La valeur bufid peut ensuite être utilisée dans pvm_bufinfo par la suite, ou pour gérer les erreurs, ou une boucle d'attente.
- D'autres primitives de réception peuvent être utilisées :
  int bufid = pvm_trecv(int tid, int msgtag, struct timeval * tmout)
  équivalent à pvm_nrecv si le timeout est mis à zéro
  équivalent à pvm_recv si le timeout est mis à NULL

Communications à haute performance

- Deux primitives pour les messages avec peu de latence et grande vitesse :
  - int info = pvm_psend(int tid, int msgtag, void * vp, int cnt, int type)
  - int bufid = pvm_precv(int tid, int msgtag,void * vp, int cnt, int typ, int * rtid, int * rtag, int rlen)
- Compression et envoi d'un buffer de données d'un même type contigues
- Aussi rapide que les primitives "propriétaires" sur les machines parallèles.

Comunication collective

- Les primitives collectives travaillent sur les membres d'un même groupe :
  - int info = pvm_barrier(char * group, int count)
    barrière de synchronisation entre tâches d'un même groupe
  - int info = pvm_bcast(char * group, int msgtag)
    Broadcast d'un même message à toutes les tâches d'un groupe.
  - pvm_scatter(result, data, num_items, data_type, mstag, rootinst, group)
  - pvm_gatter(result, data, num_items, data_type, mstag, rootinst, group)
    Distribution et collecte de tableaux à travers un groupe de tâches
  - pvm_reduce((* func)(), data, num_items, data_type, mstag, group, rootinst)
    Opération de réduction sur un tableau distribué sur un groupe de tâches.
    Les fonctions prédéfinies sont : PvmMax, PvmMin, PvmSum, PvmProduct

Contrôle de la machine virtuelle

- int info = pvm_addhosts(char ** hosts, int nhost, int * infos)
  Ajout de nouvelles machines à la machine virtuelle.
- int info = pvm_delhosts(char ** hosts, int nhost, int * infos)
  Suppression de machines à la machine virtuelle.
- int info = pvm_halt(void)
  Stoppe tous les démons pvmd et tâches (shutdown)
- int mstat = pvm_mstat(char * host)
  Pour récupérer l'état d'une machine.
- int info = pvm_start_pvm(argc, argv, block)
  Pour lancer un nouveau "master" pvm

Gestion dynamique de groupe de processus

- int inum = pvm_joingroup(char * group)
  La tâche qui invoque cette fonction rejoint le groupe de tâches.
  inum est le numéro de la tâche au sein du groupe.
- int info = pvm_lvgroup(char * group)
  La tâche qui invoque cette fonction quitte le groupe de tâches.
- int size = pvm_gsize(char * group)
  Taille du groupe.
- int tid = pvm_gettid(char * group, int inum)
  Retourne le tid de la tâche de place inum au sein du groupe.
- int inum = pvm_getinst_pvm(char * group, int tid)
  Retourne le inum d'une tâche au sein d'un groupe.

Compilation d'applications

- Fichier d'en-tête :
  - les programmes C doivent inclure :

<pvm3.h>	Toujours
<pvmtev.h>	Pour manipuler les masques de traces

  - Spécifier la directory d'include :
    `gcc -I$PVM_ROOT/include ...`
- Compilation et édition de liens :
  - Les programmes doivent être liés avec :

libpvm3.a	Toujours
libgpvm3.a	Pour manipuler les fonctions de groupe

```c
/* hello.c */
#include <stdio.h>
#include "pvm3.h"

main() {
int cc, tid;
char buf[100];

printf("i'm t%x \n", pvm_mytid());
cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);
if (cc == 1) {
cc = pvm_recv(-1, -1);
pvm_bufinfo(cc, (int*)0, (int*)0,&tid);
pvm_upkstr(buf);
printf("from t%x: %s\n", tid, buf);
} else
printf("can't start hello_other\n");
pvm_exit();
exit(0);}
```



```c
#include "pvm3.h"
main()
{
int ptid;
char buf[100];

ptid = pvm_parent();
strcpy(buf, "hello, world from ");
gethostname(buf + strlen(buf), 64);
pvm_initsend(PvmDataDefault);
pvm_pkstr(buf);
pvm_send(ptid, 1);
pvm_exit();
exit(0);}
```

```
#include <stdio.h>
#include "pvm3.h"
#define SLAVENAME "slave1"

main()
{
    int mytid;                  /* my task id */
int tids[32]; /* slave task ids */
    int n, nproc, numt, i, who, msgtype, nhost, narch;
    float data[100], result[32];
    struct pvmhostinfo *hostp;
    /* enroll in pvm */
    mytid = pvm_mytid();
    /* Set number of slaves to start */
    pvm_config( &nhost, &narch, &hostp );
    nproc = nhost * 3;
    if( nproc > 32 ) nproc = 32 ;
printf("Spawning %d worker tasks ... " , nproc);
```



```
    /* start up slave tasks in master1.c */
    numt=pvm_spawn(SLAVENAME, (char**)0, 0, "", nproc, tids);
    if( numt < nproc ){
       printf("\n Trouble spawning slaves. Aborting. Error codes are:\n");
       for( i=numt ; i<nproc ; i++ ) {
          printf("TID %d %d\n",i,tids[i]);
       }
       for( i=0 ; i<numt ; i++ ){
          pvm_kill( tids[i] );
       }
       pvm_exit();
       exit(1);    }
printf("SUCCESSFUL\n");
    /* Begin User Program */
    n = 100;
    /* initialize_data( data, n ); */
    for( i=0 ; i<n ; i++ ){
       data[i] = 1.0;    }
```

```
    /* Broadcast initial data to slave tasks in master1.c*/
pvm_initsend(PvmDataDefault);
pvm_pkint(&nproc, 1, 1);
pvm_pkint(tids, nproc, 1);
pvm_pkint(&n, 1, 1);
pvm_pkfloat(data, n, 1);
    pvm_mcast(tids, nproc, 0);
```



```
    /* Wait for results from slaves */
    msgtype = 5;
    for( i=0 ; i<nproc ; i++ ){
       pvm_recv( -1, msgtype );
       pvm_upkint( &who, 1, 1 );
       pvm_upkfloat( &result[who], 1, 1 );
       printf("I got %f from %d; ",result[who],who);
       if (who == 0)
printf( "(expecting %f)\n", (nproc - 1) * 100.0);
       else
printf( "(expecting %f)\n", (2 * who - 1) * 100.0);

    }
    /* Program Finished exit PVM before stopping */
    pvm_exit();
}
```

```
#include <stdio.h>
#include "pvm3.h"
/* debut du programme esclave */
main()  {
    int mytid;       /* my task id */
    int tids[32];    /* task ids   */
int n, me, i, nproc, master, msgtype;
float data[100], result;
    float work(); /* definition de fonction ! */
    /* enroll in pvm */
mytid = pvm_mytid();
    /* Receive data from master */
    msgtype = 0;
    pvm_recv( -1, msgtype );
pvm_upkint(&nproc, 1, 1);
pvm_upkint(tids, nproc, 1);
pvm_upkint(&n, 1, 1);
pvm_upkfloat(data, n, 1);
```



```
    /* Determine which slave I am (0 -- nproc-1) */
    for( i=0; i<nproc ; i++ )
       if( mytid == tids[i] ){ me = i; break; }
    /* Do calculations with data */
    result = work( me, n, data, tids, nproc );
    /* Send result to master */
    pvm_initsend( PvmDataDefault );
    pvm_pkint( &me, 1, 1 );
    pvm_pkfloat( &result, 1, 1 );
    msgtype = 5;
    master = pvm_parent();
    pvm_send( master, msgtype );
    /* Program finished. Exit PVM before stopping */
    pvm_exit();}
```

```
float work(me, n, data, tids, nproc )
/* Simple example: slaves exchange data with left neighbor (wrapping) */
    int me, n, *tids, nproc;
    float *data;
{
    int i, dest;
    float psum = 0.0;
    float sum = 0.0;
    for( i=0 ; i<n ; i++ ) { sum += me * data[i]; }
    /* illustrate node-to-node communication */
    pvm_initsend( PvmDataDefault );
    pvm_pkfloat( &sum, 1, 1 );
    dest = me+1;
    if( dest == nproc ) dest = 0;
    pvm_send( tids[dest], 22 );
    pvm_recv( -1, 22 );
    pvm_upkfloat( &psum, 1, 1 );
    return( sum+psum );}
```


Load Balancing (1)

- Important pour avoir une application performante
- N'est pas réalisé automatiquement
- Statique :
  - Doit prédire le temps de calcul et la place mémoire !
  - Doit pouvoir tenir compte de machine avec des vitesses et des tailles mémoire différentes
  - La charge de travail des machines est imposée statiquement de façon externe

Load Balancing (2)

- Dynamique :
  - On peut faire un ordonnanceur simple avec un “pool” de tâches
    - Simple et cela marche assez bien
    - Diviser le travail en “petit” jobs
    - Fournir du travail à un processeur dès qu'il est inoccupé
    - Peut inclure une gestion de la tolérance aux pannes
  - Migration de travail
    - Sondage pour détruire depuis le maître
    - Peut être interrompu avec pvm_sendsig
    - Destruction du “travailleur” chère
  - Migration de tâche : à gérer soit même
- Même avec l'équilibrage de charge, s'attendre à des performances variables