TD 3

Exercices C/PVM (suite)

3.1 Calcul d'une opération associative

Soit $\oplus$ une operation associative quelconque sur des nombres entiers (par exemple, max, ou + etc.). On se propose d'implémenter un programme PVM parallèle calculant rapidement cette opération associative sur un tableau de nombres entiers A par la méthode "diviser pour régner". Pour cela on écrira un maître qui contient au début le tableau A (que l'on initialisera comme on voudra, et de taille 2^k, avec $k > 1$), puis qui lance récursivement deux fils avec chacun une moitié du tableau,..., ainsi de suite jusqu'à tomber sur des fils qui n'ont que deux nombres entiers. Ceux-ci effectuent l'opération associative et renvoient le résultat à leurs parents,..., et ainsi de suite. On pourra s'inspirer du code de FFT en annexe de cet exercice qui utilise la même méthode. Peut-on optimiser en réduisant le nombre de fils?

Remarque: Ceci est un exercice un peu académique dans le sens où certains de ces algorithmes dits de "réduction" sont déjà implémentés sous PVM.

Rappel:

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

- Permet de lancer `n` tâches PVM excútant le code `"programme"`
- Le `tid` de chacune des tâches est rangé dans une entrée du tableau `etid`
- Renvoie `numt <= n` le nombre de tâche(s) effectivement lancée(s)
- `argv` est la liste d'arguments (type ligne de commande UNIX) à passer à `"programme"`,
- `flag` donne des options à `pvm_spawn`,
- pour certaines de ces options `where` indique sur quel type d'hôte lancer les processus.

Exemple: Calcul récursif d'une transformée de fourier rapide (les sources sont accessibles sur la page web du cours, dans la section "corrigés").

"fft-startup" est à la racine de l'arbre des appels récursifs:

```
#include <stdlib.h>
#include <stdio.h>
#include "pvm3.h"
#define RESULT_TAG 1
```

```
#define MAX_LENGTH 128

main(argc, argv)
int argc;
char *argv[];
{
/* argv[i] is the unidimensional complex signal to be processed */
/* format is fft Re(x_1) Im(x_1) Re(x_2) ... Re(x_n) Im(x_n) */
int i,bufid,mtid,tid,numt;
float R[MAX_LENGTH];
        if ((mtid=pvm_mytid())<0)
           {
                printf("unexpected error occured!\n");
                exit(1);
           }
        setlinebuf(stdout);
       if (argc < 5)
          {
               printf("wrong number of parameters\n");
               exit(1);
          }
       numt = pvm_spawn("fft",&argv[1],PvmTaskDefault,"",1,
                                   &tid);
        if (numt != 1)
           {
                printf("could not spawn fft\n");
                exit(1);
           }
        bufid = pvm_recv(tid, RESULT_TAG);
        pvm_upkfloat(&R[0],argc-1,1);
       for (i=0;i<(argc-1)/2;i++)
          {
               printf(" %g %g",R[i],R[i+(argc-1)/2]); }
       printf("\n");
       pvm_exit();
       exit(0); }
```

Le programme récursif “fft” se charge du calcul effectif:

```
#include <math.h>
#include <stdio.h>
#include "pvm3.h"

#define RESULT_TAG 1
#define SIZE 64
#define MAX_LENGTH 128
#define PI 3.14159265358

main(argc, argv)
int argc;
char *argv[];
{
/* argv[i] is the unidimensional complex signal to be processed */
/* format is fft Re(x_1) Im(x_1) Re(x_2) ... Re(x_n) Im(x_n) */
```

```
/* sends to his parent an array of float with RESULT_TAG equal to 1 */
       int i,numt,bufid,mtid,ptid,lstid,rstid;
       float wr,wi;
       float Rr[MAX_LENGTH],Ri[MAX_LENGTH];
       float Xr[MAX_LENGTH];
       float Xi[MAX_LENGTH];
       float T1[MAX_LENGTH],T2[MAX_LENGTH];
       char *fargv1[MAX_LENGTH+1],*fargv2[MAX_LENGTH+1];
       if ((mtid=pvm_mytid())<0)
          {
               printf("unexpected error occured!\n");
               exit(1);
          }
       ptid=pvm_parent();
       setlinebuf(stdout);
       for (i=0;i<(argc-1)/2;i++)
          {
               sscanf(argv[2*i+1],"%g",&Xr[i]);
               sscanf(argv[2*(i+1)],"%g",&Xi[i]);
               if (i%2 == 0)
                  {
                       fargv1[i]=argv[2*i+1];
                       fargv1[i+1]=argv[2*(i+1)];
                  }
               else
                  {
                       fargv2[i-1]=argv[2*i+1];
                       fargv2[i]=argv[2*(i+1)];
                  }
          }
       fargv1[(argc-1)/2]=NULL;
        fargv2[(argc-1)/2]=NULL;
       if (argc==5)
          {
               Rr[1]=(Xr[0]-Xr[1])/2;
               Ri[1]=(Xi[0]-Xi[1])/2;
               Rr[0]=(Xr[0]+Xr[1])/2;
               Ri[0]=(Xi[0]+Xi[1])/2;
          }
       else
          {
               numt = pvm_spawn("fft",&fargv1[0],PvmTaskDefault,"",1,
                                              &lstid);
               if (numt != 1)
                  {
                       printf("could not spawn left son\n");
                       exit(1);
                  }
               numt = pvm_spawn("fft",&fargv2[0],PvmTaskDefault,"",1,
                                               &rstid);
               if (numt != 1)
                  {
                       printf("could not spawn right son\n");
                       exit(1);
```

```
            }
          bufid = pvm_recv(lstid, RESULT_TAG);
          pvm_upkfloat(&T1[0],(argc-1)/2,1);

          bufid = pvm_recv(rstid, RESULT_TAG);
           pvm_upkfloat(&T2[0],(argc-1)/2,1);
          for (i=0;i<(argc-1)/4;i++)
            {
                 wr=cos(-4*PI*i/(argc-1));
                 wi=sin(-4*PI*i/(argc-1));
                 Rr[i]=(T1[i]+wr*T2[i]-wi*T2[i+(argc-1)/4])/2;
                 Ri[i]=(T1[i+(argc-1)/4]+wr*T2[i+(argc-1)/4]+wi*T2[i])/2;
                 Rr[i+(argc-1)/4]=(T1[i]-wr*T2[i]+wi*T2[i+(argc-1)/4])/2;
                 Ri[i+(argc-1)/4]=(T1[i+(argc-1)/4]-wr*
                         T2[i+(argc-1)/4]-wi*T2[i])/2;
            }
      }
    pvm_initsend(PvmDataDefault);
    pvm_pkfloat(&Rr[0],(argc-1)/2,1);
    pvm_pkfloat(&Ri[0],(argc-1)/2,1);
    pvm_send(ptid,RESULT_TAG); pvm_exit(); exit(0); }
```

3.2 Client/Serveur

On considère une architecture client/serveur toute simple: un serveur de calculs S sait comment calculer certaines fonctions (on choisira ce que l'on veut) prenant un seul argument. Des clients, $C_1, \ldots, C_n$, veulent avoir accès au calcul de ces fonctions à la demande. Clients et serveur seront chacun des processus PVM. Pour simplifier, on supposera que l'on `spawn` le serveur au début, et que c'est lui qui crée ses clients.

- Le serveur exécute en boucle: `recv` d'ordres et d'arguments de n'importe quel client, effectue alors le calcul demandé puis renvoie le résultat,
- Chaque client exécute aussi en boucle un `send` d'ordres et d'arguments au serveur et fait un `recv` pour les résultats.

Quel est le problème avec le `recv` bloquant de PVM? On pourra améliorer le programme avec un `recv` non-bloquant, décrit en annexe.

Remarque: Cela peut servir entre autres à faire de l'équilibrage de charge "centralisé" où les clients demandent sitôt qu'il ont fini un calcul sur un sous-domaine d'autres calculs à effectuer au serveur.

Annexe: Réception non-bloquante:

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

- Réception asynchrone: si un message avec le bon tag est arrivé, le place dans le tampon `bufid`. Sinon renvoie `bufid=0`.
- Erreur si `bufid < 0`.

On peut vérifier si quelque chose est arrivé ou pas,

```
int info;
info=pvm_bufinfo(int bufid,int *bytes,int *msgtag,int *tid);
```

- Donne l'état du tampon de communication `bufid`: après exécution, `bytes` contient le nombre d'octets du message éventuellement reçu, `msgtag`, son tag et `tid` l'identificateur de tâche de l'expéditeur,
- <u>Erreur</u> si `info < 0`.

```
#include <stdio.h>
#include <pvm3.h>
#define MSG 1

main()
{
   int mtid,ctid,bufid,ok,i;
   if ((mtid=pvm_mytid())<0) { pvm_exit();
                               exit(1); }

   pvm_spawn("async2",(char **) 0,PvmTaskDefault,
                             (char *) 0,1,&ctid);
   fprintf(stdout,"I am spawning one child\n");
   fflush(stdout);

   ok = 0;
   i = -1;
   while (!ok)
      { i++;
        bufid = pvm_nrecv(ctid,MSG);
        if (bufid < 0) { pvm_perror("Pb with nrecv");
                         pvm_exit(); exit(1); }
        if (bufid != 0) ok=1;
        else if ((i % 60000) == 0)
                { fprintf(stdout,"I am computing...\n");
                  fflush(stdout);
                } }

   pvm_upkint(&i,1,1);
   fprintf(stdout,"I have received %d\n",i);
   fflush(stdout);
   pvm_exit();
   exit(1);
}
```

Esclave "async2":

```
#include <stdio.h>
#include <pvm3.h>
#define MSG 1

main()
{
   int i=42;
   sleep(4);
   fprintf(stdout,"I'm sending a message to my parent\n");
   fflush(stdout);
   pvm_initsend(PvmDataDefault);
   pvm_pkint(&i,1,1);
   pvm_send(pvm_parent(),MSG);
```

```
   pvm_exit();
   exit(1);
}
```

```
pvm> spawn -1 -> async1
[11]
1 successful
t40021
pvm> [11:t40021] I am spawning one child
[11:t40021] I am computing...
[11:t40021] I am computing...
[11:t40021] I am computing...
[11:t40021] I am computing...
[11:t40021] I am computing...
[11:t40022] I am sending a message to my parent
[11:t40022] EOF
[11:t40021] I have received 42
[11:t40021] EOF
[11] finished
```