TD “Calcul Parallèle”

Éric Goubault

11 février 2000

Table des matières

TD 1

Quelques exercices C

1.1 Calcul de la fonction de Fibonacci

On cherche à calculer et à afficher la suite de valeurs définie par,

- $u_0 = 1$,
- $u_1 = 1$,
- $u_{n+2} = u_{n+1} + u_n$.

Ecrire successivement un programme C itératif, puis un programme C récursif, puis un programme C récursif utilisant un argument entier supplémentaire appelé "accumulateur" permettant de ne pas recalculer trop souvent des itérées anciennes (à vous de réfléchir!).

1.2 Racine carrée

Codez la fonction racine carrée à l'aide de la méthode de Newton, c'est-à-dire où $\sqrt{a}$ s'obtient comme la limite de la suite récurrente associée à la fonction qui à x associe $\frac{x+\frac{a}{x}}{2}$.

TD 2

Quelques exercices C/PVM

2.1 Tri fusion

Reprendre l'exercice en fin des notes de TDs de la dernière fois. Une autre option pour le tri fusion parallèle est la suivante:

- Un maître M a un tableau local à trier. Il envoie la deuxième moitié à un esclave B,
- M et B trient leur morceau de tableau,
- (1) B envoie le premier élément de son tableau (donc le plus petit) à M,
- (2) M construit le tableau ordonné résultat du tri global en rangeant soit l'élément envoyé par B, soit les éléments de sa propre partie de tableau inférieurs à l'élément envoyé par B,
- retour à (1) jusqu'à la fin.

Pourrait-on proposer encore une autre méthode de tri fusion? (eu égard aux communications entre B et M en (1) et (2))

2.2 Calcul d'intégrale

Ecrire un programme C/PVM sur le modèle du calcul de π (maître/esclave) qui effectue l'intégration d'une fonction par la méthode des trapèzes,

$$\int_{x_0}^{x_1} f(x)dx \quad \cong \quad \frac{x_1 - x_0}{n} \left(\frac{f(x_0) + f(x_1)}{2} + \sum_{i=1}^{n-1} f\left(x_0 + i\frac{x_1 - x_0}{n} \right) \right)$$

2.3 Crible d'Erathostène

Tony Hoare a proposé il y a longtemps de calculer les nombres premiers inférieurs à n^2 en utilisant $n+2$ processus, qui recoivent de leur prédecesseur une suite croissante de nombres naturels dont le premier, p, est un nombre premier qui est imprimé, et transmettent à leur successeur les nombres de cette suite, dans le même ordre, à l'exclusion de p et de ses multiples.

En utilisant la bibliothèque PVM, écrire un programme C qui affiche la suite des nombres premiers par la méthode du crible d'Erathostène: un processus est chargé de générer les entiers naturels, dont on élimine d'abord les multiples de 2, puis 3, 5 etc. au moyen de processus filtrants successifs.

On recommande chaudement de montrer le programme (qui compile néanmoins) à un chargé de TP avant de l'essayer...

2.4 Equation de la chaleur

On considère l'équation de la chaleur sur l'ouvert $\Omega =]0,1[\times]0,1[$,

$$\rho c \frac{\partial u}{\partial t} = \mathrm{div}(\kappa(x).\nabla u) + S(t).\delta_{x_s}(x)$$

avec les conditions aux bords

$$u = 0$$

et la condition initiale

$$u(x,0) = 0$$

κ est la conductivité thermique du matériau considéré, c est la capacité calorifique massique et ρ la densité. $S(t)$ est une source de chaleur et δ_{x_s} est une masse de Dirac au point x_s. Ici, on va supposer κ constant dans tout le domaine.

On considère un schéma explicite en temps et une discrétisation explicite d'ordre deux en espace. On approxime donc u par ses valeurs discrétisées en les sommets (i,j) d'une grille régulière de pas h (en abscisse et ordonnée), et en des temps n (pas de temps Δt), soit $u^n_{i,j}$. L'équation discrétisée est maintenant:

$$\rho c \frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{\kappa}{h^2}(u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j+1} + u^n_{i,j-1} - 4u^n_{i,j})$$

Le critère de stabilité (expliqué également dans le prochain cours) de l'équation discrétisée nous donne une condition pour le choix du pas de temps:

$$\Delta t < \frac{\rho c h^2}{4\kappa}$$

Pour les applications numériques, on prendra $\frac{4\kappa\Delta t}{\rho c h^2} = \frac{1}{2}$ et on choisira h.

On souhaite écrire un programme maître/esclave permettant de calculer le schéma précédent,

- On suppose que l'on a N esclaves. Comment peut-on découper la grille en sous-grilles à faire calculer à chacun des esclaves?
- Pour les choix énumérés précédemment, comment les esclaves vont-ils communiquer entre eux?
- Ecrire un programme PVM maître gérant l'envoi des sous-grilles aux esclaves.
- Comment gérer la source de chaleur $S(t).\delta_{x_s}(x)$?
- Ecrire un programme PVM esclave gérant les communications avec le maître et avec les autres esclaves.
- Ecrire le schéma entièrement.
- Peut-on avoir des catastrophes numériques?

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 exćutant 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,
- Erreur 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
```

TD 4

Résolution de systèmes linéaires (C/PVM)

Au cours d'une résolution numérique de l'équation des ondes par un schéma implicite, on a à trouver la solution d'un système d'équations linéaires $Ax = b$. Après un algorithme de type pivot de Gauss que l'on n'étudiera pas ici, on se retrouve avec un système triangulaire supérieur $Ux = b$. On se propose ici de programmer la résolution parallèle de ce dernier système.

Soit donc à résoudre le système triangulaire supérieur

$$Ux = b$$

avec,

$$U = \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} & \cdots & U_{1,n} \\ 0 & U_{2,2} & U_{2,3} & \cdots & U_{2,n} \\ & & & \cdots & \\ 0 & 0 & \cdots & 0 & U_{n,n} \end{pmatrix}$$

et $U_{i,i} \neq 0$ pour tout $1 \leq i \leq n$.

On procède par "remontée" c'est à dire que l'on calcule successivement,

$$\begin{array}{rcl} x_n & = & \frac{b_n}{U_{n,n}} \\ x_i & = & b_i - \frac{\sum_{j=i+1}^{n} U_{i,j} x_j}{U_{i,i}} \end{array}$$

pour $i = n-1, n-2, \cdots, 1$.

On écrit les équations précédentes en pseudo-code C de la façon suivante:

```
x[n]=b[n]/U[n,n];
for (i=n-1;i>=1;i--)
   {
      x[i]=0;
      for (j=i+1;j<=n;j++)
         L: x[i]=x[i]+U[i,j]*x[j];
      x[i]=b[i]-x[i]/U[i,i];
   }
```

où `n` est la taille de la matrice U représentée par le tableau `U[i,j]`, où `x[i]` (`i` entre 1 et `n`) représente le vecteur "inconnu" x et `b[i]`, le vecteur second membre du système d'équations linéaires.

Programmer le code de remontée correspondant en PVM. On supposera que P_n est le maître qui contient au début tout le tableau `U` et le tableau `b` et crée ses esclaves $P_1,\cdots,P_{n-1}$. On pourra comme dans le cas séquentiel commencer par calculer `x[n]`. Ensuite, l'idée sera de placer chacune des composantes des tableaux rentrant en compte pour le calcul de la ième équation linéaire $(1 \leq i \leq n-1)$ [1] sur le ième processeur P_i. On pourra donc propager les valeurs déjà résolues des `x[j]`, `j`$\geq$`i`$+1$ aux processeurs P_k, $k \leq$`i`, qui effectueront une même étape de sommation élémentaire. A la fin il doit attendre le résultat de tous ses esclaves.

1. C'est celle qui sert à déterminer `x[i]` à partir des `x[j]` (`j`$\geq$`i`$+1$) lors de la remontée.