IF5-PAR2 : Rapport de TP1

Parallélisation de l'élimination de Gauss - Jordan

Jacquemin Thibault – Marleix Mathieu

Table des matières

Introduction

Ce rapport détaille notre implémentation d'une parallélisation de l'algorithme d'élimination de Gauss sous PVM. Nous allons d'abord présenter la manière théorique d'implémentation prévue puis comment nous avons chargé les données matricielles. Nous détaillerons ensuite la manière d'effectuer le calcul de l'élimination de Gauss pour enfin présenter et expliquer les résultats obtenus avec PVM lors des benchmarks.

Nous disposions d'un algorithme séquentiel de Gauss fournis et nous devions réussir son implémentation en parallèle. Le code source fourni peut être examiné en annexe. Nous disposions d'autre part d'un exemple d'algorithme de Token-Ring sous PVM.

I. Implémentation de l'élimination de Gauss-Jordan parallèle

Algorithme d'élimination de Gauss-Jordan

Pour accomplir une parallélisation de cet algorithme il est nécessaire au préalable de comprendre le fonctionnement de cet algorithme. Une implémentation séquentielle de cet algorithme sous Matlab est de la forme suivante :

```
real a(100,100),b(100)
integer n
for k = 1 to n-1 do
   for i = k+1 to n do
      for j = k+1 to n do
      a(i,j) = a(i,j) - a(i,k) / a(k,k) * a(j,k)
      endfor
   endfor
endfor
```

Le code de l'algorithme d'élimination de Gauss-Jordan possède trois boucles imbriquées. On étudie alors le code en C fourni :

```
for ( k=0; k<N-1; k++ ){ /* mise a 0 de la col. k */
    /* printf (". "); */
    if ( fabs(*(tab+k+k*N)) <= 1.0e-11 ) {
      printf ("ATTENTION: pivot %d presque nul: %g\n", k, *(tab+k+k*N) );
      exit (-1);
    }
    for ( i=k+1; i<N; i++ ){ /* update lines (k+1) to (n-1) */
      pivot = - *(tab+k+i*N) / *(tab+k+k*N);
      for ( j=k; j<N; j++ ){ /* update elts (k) - (N-1) of line i */
    *(tab+j+i*N) = *(tab+j+i*N) + pivot * *(tab+j+k*N);
      }
      /* *(tab+k+i*N) = 0.0; */
    }
  }
```

L'algorithme prend en entrée une matrice de données de taille N^2 et fourni en sortie la matrice triangularisée. On observe que chacune des lignes doit calculer un pivot en fonction de celles d'avant. On choisit de diviser notre matrice d'entrée par des blocs de taille fixés sur plusieurs sites différents. On suppose la taille de la matrice divisible par le nombre de processeurs, la taille des blocs sera de N / Nombre de processeurs. Cela implique un chargement et une sauvegarde spécifique des données.

Chargement de données multiprocesseurs

De manière à éviter les accès concurrents au fichier de stockage de la matrice, on associe la lecture et la séparation de donnée à la première tâche lancée. Celle-ci va venir lire les différentes valeurs des coefficients matriciels stockés dans le fichier (*scanf*), écrire ses valeurs dans un tableau local (*memcpy*) puis transmettre les valeurs suivantes aux différents processeurs de manière successive (*utilisation conjointe des fonctions pvm_gettid,pvm_ initsend,pvm_ pkdouble et pvm_send*).

Pendant que la première tâche lit les données, les autres tâches vont se placer en attente de réception à l'aide de la fonction bloquante de PVM (*pvm_recv*). Une fois les données reçues, elles vont venir les écrire à leur tour dans un tableau local.

Le code de la fonction de chargement est donc le suivant :

```
/**
 * Fonction de chargement de matrice en parallèle.
 * @arg char nom[], nom du fichier à ouvrir;
 * @arg double tab, tableau local utilisé pour les calculs futur;
 * @arg int me, numéro du processeur courant;
 * @arg int N, taille de ligne de la matrice;
 * @arg int NPROC, nombre de processeurs.
 * int tids[], liste des processeurs.
 */
void matrix_load ( char nom[], double *tab, int me, int N, int NPROC, int tids[]) {
  FILE *f;
  int i,j,idx,idxt = 0,dest,src,msgtag = me;
    const int NBLIGNES = N/NPROC;
  double * buffer;

    if ( (buffer = malloc(N*sizeof(double))) == NULL ) {
        fprintf(stderr,"Cant malloc %d bytes\n", N*sizeof(double));
        exit (-1);
    }
    if (me == 0){
        if ((f = fopen (nom, "r")) == NULL)
            perror ("matrix_load : fopen ");
        for (i=0; i<N; i++) {
          for (j=0; j<N; j++) {
            fscanf (f, "%lf", (buffer+j) );
          }
            idx = i % NPROC;
            if (idx == 0) {
                memcpy(tab+(idxt++)*N, buffer, N*sizeof(double));
            } else {
                pvm_initsend( PvmDataDefault );
                pvm_pkdouble( buffer, N, 1);
                pvm_send( tids[idx], i );
            }
        }
        fclose (f);
    } else {
        for (i=0; i<NBLIGNES; i++, msgtag += NPROC) {
            pvm_recv( tids[0], msgtag );
                pvm_upkdouble( buffer, N, 1 );
            idx = i % NPROC;
            memcpy(tab+(idxt++)*N, buffer, N*sizeof(double));
        }
    }
}
```


Une fois les données chargées, chacun des processeurs dispose de sa propre partie des données à traiter. On accélère donc le traitement du calcul pour chaque processeur ajouté. Reste à voir comment faire communiquer ces différents processus au sein du calcul de l'élimination de Gauss – Jordan.

Calcul du Gauss

Chaque ligne calcul un pivot en fonction de celui des lignes précédentes. Il est donc nécessaire que chaque ligne transmette son pivot aux suivantes. Une fois le chargement et la transmission des données terminé, la première tâche va commencer le calcul de son pivot puis va le transmettre à chacun des autres processeurs.

Bloqués en attente de réception, à chaque réception d'un pivot d'une autre ligne, les autres processeurs vont ajouter celui-ci au calcul de leur propre pivot. Une fois arrivé sur la ligne du processeur en question, celui-ci va alors packager son résultat et l'envoyer aux autres lignes.

Les calculs sont faits à l'aide d'une version légèrement modifiée de l'algorithme séquentiel où l'on va utiliser les données récupérées par diffusion.

Le code de l'élimination de Gauss-Jordan est donc la suivante :

```
/**
 * Calcul de l'élimination de Gauss - Jordan parallèle.
 * double * tab, tableau de donnée source de chacun des processeurs;
 * int N, taille de ligne du tableau;
 * int NPROC, nombre de processeurs;
 * int me, tâche courante;
 * int tids[], liste des processeurs.
 */
void gauss (int me, double * tab, int N , int NPROC, int tids[]) {
   int i,j,k,l, decalage = 0;
   double pivot, buffer[N];

   for(k=0;k<(N-1);k++) {
       if(k%NPROC==me) {
           memcpy(&buffer[k], tab + (k/NPROC)*N + k,(N-k)*sizeof(double));
           pvm_initsend(PvmDataDefault);
           pvm_pkdouble(&buffer[k],N-k,1);
           for(l=0; l<NPROC;l++){
               if(l != me){
                   pvm_send(tids[l], k);
               }
           }
       } else {
           pvm_recv(tids[k%NPROC], k);
           pvm_upkdouble(&buffer[k],N-k,1);
       }
       decalage = (k/NPROC) + ((me>(k%NPROC))?0:1);
       for(i=decalage;i<(N/NPROC);i++) {
           if ( fabs(buffer[k]) <= 1.0e-11 ) {
                   printf ("ATTENTION: pivot %d presque nul: %g\n", k, buffer[k] );
                   exit (-1);
           }
           pivot = (*(tab + i * N + k)/buffer[k]);
           for(j=k;j<N;j++){
               *(tab + i * N +j) = *(tab + i * N + j) - pivot*buffer[j];
           }
       }
   }
}
```


Sauvegarde de données multiprocesseurs

En comparaison avec le chargement des données multiprocesseurs évoqué plus haut, ici ce sont les différents processeurs qui vont transmettre leurs différents calculs à la première tâche exécutée. La première tâche exécuté est mise en attente de chaque envoi des autres processeurs jusqu'à réception de tous les pivots. La matrice va alors être concentrée par la tâche et écrite dans le fichier de résultat.

Le code de la sauvegarde parallèle est donc tel que :

```
/**
 * Sauvegarde parallèles d'un tableau sous PVM.
 * @arg char nom[], nom du fichier de sortie à écrire;
 * @arg double * tab, tableau des indices matriciels;
 * @arg int me, numéro du processeur courant;
 * @arg int N, nombre de ligne de la matrice;
 * @arg int NPROC, nombre de processeurs.
 */
void matrix_save ( char nom[], double *tab, int me, int N, int NPROC, int tids[] ) {
  FILE *f;
  int i, j, idx, idxt = 0, src, dest, msgtag = me;
    const int NBLIGNES = N/NPROC;
    double *buffer;
    if ( (buffer = malloc(N*sizeof(double))) == NULL ) {
        fprintf(stderr,"Cant malloc %d bytes\n", N*sizeof(double));
        exit (-1);
    }

    if (me == 0) {
        if ((f = fopen (nom, "w")) == NULL)
            perror ("matrix_save : fopen ");
        for (i=0; i<N; i++) {
            idx = i%NPROC;
            if (idx == 0) {

                memcpy(buffer,tab+(idxt++)*N,N*sizeof(double));
            } else {
                pvm_recv( tids[idx], i );
                pvm_upkdouble( buffer, N, 1 );
            }
        for (j=0; j<N; j++) {
                fprintf (f, "%8.2f\t", buffer[j] );
        }
        fprintf (f, "\n");
    }
    fclose (f);
    } else {
        for (i=0; i<NBLIGNES; i++, msgtag += NPROC) {
            pvm_initsend( PvmDataDefault );
            pvm_pkdouble( tab+i*N, N, 1);
            pvm_send( tids[0], msgtag );
        }
  }
}
```


II. Résultats d'exploitation

Validation du programme

En vérifiant notre programme par une exécution locale avec plusieurs tâches, nous avons pu observer que l'indice du flag de la fonction send nécessite d'être identique entre l'envoi et la réception pour permettre le depackaging des données. Nous avons donc intégré les modifications dans notre code permis permettant ce traitement.

Une fois que les résultats entre les matrices obtenues et celles fournies furent validés, nous sommes passés aux benchmarks des matrices de plus grande taille pour vérifier l'optimisation apportée par le traitement parallèle.

Benchmark

Pour obtenir des résultats cohérents ainsi que les plus optimaux possibles, nous avons décidés de nous passer de l'interface graphique de XPVM et utiliser directement PVM (L'interface graphique génère une charge processeur et ralenti le calcul). Pour nos différents tests, nous avons ajouté à la main les hôtes à l'aide de la commande « add [hostname] » puis nous avons lancé notre programme en spécifiant le nombre de thread à l'aide de la commande « spawn –[nombre de thread] -> [programme] [arguments] ».

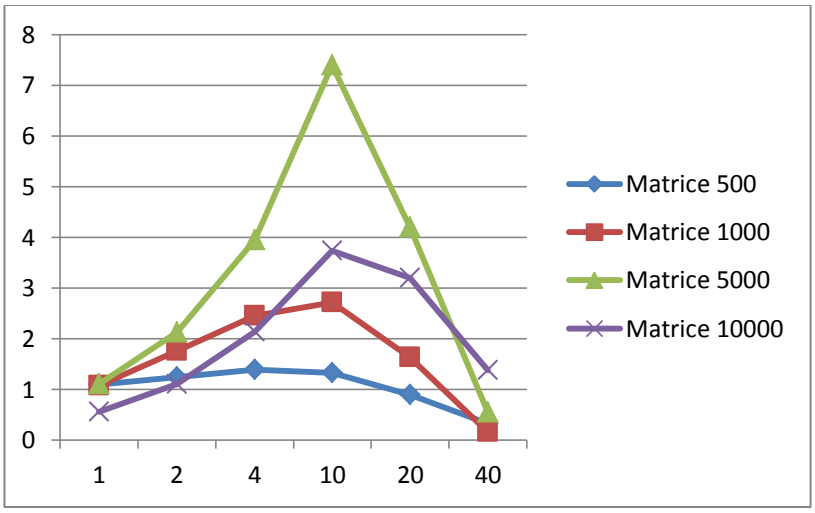
Les résultats entre les deux types de prélèvements ont justifiés notre choix puisque lors de test avec de grand nombre de machines, PVM avait non seulement des difficultés à actualiser l'interface mais aussi nous renvoyait des temps en moyenne 1,5 fois supérieurs à ceux pris avec PVM.

SpeedUp

Le benchmark a été effectué sur plusieurs salles, nous avons alors pu constater que le succès des performances dépendait fortement de la charge réseau et de l'occupation ou non des machines.

Nous avons pu constater une importante différence de résultat entre des mesures prisent lors que le réseau était inutilisé et lors d'une utilisation normale : Pour 10 hôtes donnée, le traitement d'une matrice (5000 : 5000) prend 39135,859 ms sur un réseau inutilisé et 60319,478 ms sur un réseau subissant une utilisation normale, soit une multiplication du temps de calcul par 2,5 fois.

Le graphique montrant le SpeedUp en fonction du nombre de machines allouées est tel que :

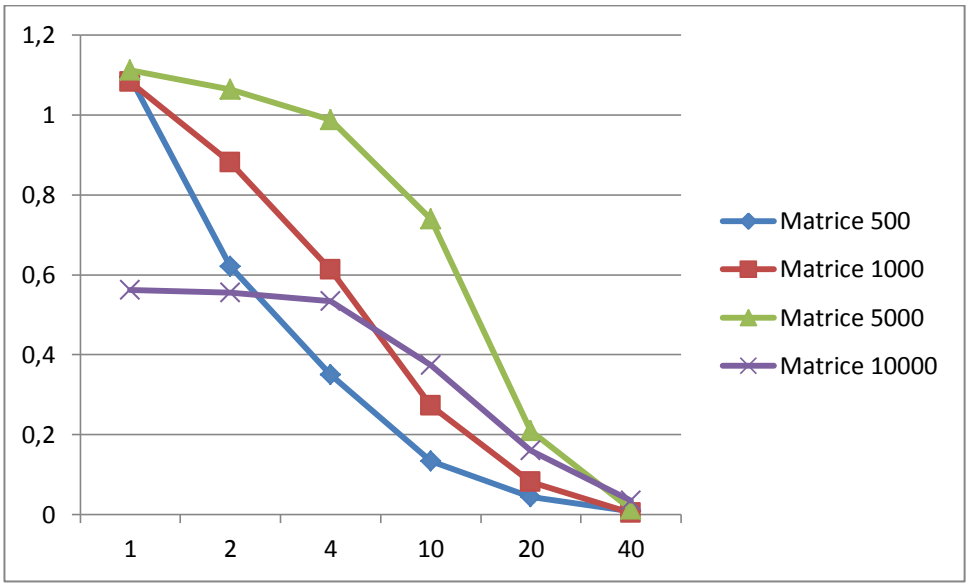


On constate bien que le parallélisme diminue de manière importante le calcul pour des matrices de grosses tailles. Cependant, l'augmentation du nombre de machine est valable jusqu'à une valeur critique : une fois cette valeur dépassé, le coût de communication devient plus important que le coût en calcul et le SpeedUp chute.

Nous avons aussi observé ceci en lançant plusieurs tâches sur un même système : les communications interprocessus ont pratiquement doublé le temps pour obtenir le résultat. Si nous pouvons lancer plusieurs tâches de calcul sur une machine, il est pertinent de lancer autant de tâche que le nombre de cœurs présents dans la machine.

Efficacité

Les résultats du calcul d'efficacité peuvent être vérifiés dans le graphique suivant :



Nous constatons qu'il est impossible d'obtenir un rendement parfait en fonction du nombre d'hôte et que même si le temps de calcul chute de manière importante avec la parallélisation, on constate une baisse successive du rendement.

Remarques

Il est théoriquement possible d'améliorer les performances en faisant du recouvrement calculs / communication. Le schéma impliquant que la dernière tâche commence le chargement de la matrice en lieu et place de la première permettrait aux autres tâches de commencer directement le calcul plutôt que d'attendre que la première tâche ait fini le chargement. Nous n'avons cependant pas eu le temps d'inclure cette modification à notre code.

Lorsque le pivot est nul, l'application est quittée. Dans le meilleur cas possible, celle-ci devrait sélectionner un nouveau pivot en permutant la ligne traitée avec une ligne du dessous. Cependant suite à la diffusion de la matrice, récupérer ces pivots complexifie le programme et nous n'avons pas eu le temps de le mettre en place dans le code durant ce TP.

Conclusion

Nous avons donc pu observer l'utilité de la parallélisation d'un algorithme tel que celui de l'élimination de Gauss – Jordan au cours de ce TP et comment elle peut être implémentée. Ces procédés permettent de réduire grandement le temps de calcul et sont de plus en plus utilisés pour résoudre des problèmes de grande taille limités par la puissance de calcul des machines actuelles.

Nous avons pu nous exercer à la programmation en parallèle sous PVM et constater les communications interprocessus obtenues à travers la version graphique de PVM, XPVM ainsi que les différences entre temps d'exécution pour des même programmes avec et sans interface graphique.

Annexe

Tableau de données du SpeedUp /Efficacité

Taille matrice	Séquentiel	1 host	2 hosts	4 hosts	10 hosts	20 hosts	40 hosts
500	279	254,833	224,505	199,599	209,155	310,565	835,859
1000	2254	2082,202	1279,574	918,981	828,582	1377,953	13793,56
5000	289554	260432,656	136124,935	73264,294	39135,859	68880,94	523371,308
10000	1178639	2099152,319	1060668,453	551572,685	315453,179	367900,255	854760,693
	SpeedUp	1 host	2 hosts	4 hosts	10 hosts	20 hosts	40 hosts
500		1,094834656	1,242734015	1,39780259	1,33393894	0,89836266	0,33378835
1000		1,082507845	1,761523757	2,45271665	2,72031012	1,63575971	0,16340959
5000		1,111819095	2,127119473	3,9521844	7,39868774	4,20368828	0,55324775
10000		0,561483314	1,11122283	2,13686978	3,73633578	3,20369172	1,37891109
	Efficacité	1 host	2 hosts	4 hosts	10 hosts	20 hosts	40 hosts
500		1,094834656	0,621367007	0,34945065	0,13339389	0,04491813	0,00834471
1000		1,082507845	0,880761879	0,61317916	0,27203101	0,08178799	0,00408524
5000		1,111819095	1,063559737	0,9880461	0,73986877	0,21018441	0,01383119
10000		0,561483314	0,555611415	0,53421744	0,37363358	0,16018459	0,03447278