Parallélisation de l'algorithme d'élimination de Gauss avec PVM

INF-5101A - Parallélisme et calculs distribués

Stela CARNEIRO ESPÍNDOLA
Morgane JOLY
Cécile POV

Responsable : M. Laurent PERROTON

October 19, 2019

Contents

1 Introduction

1.1 Objectif du TP

L'objectif de ce TP est de proposer une parallélisation de l'élimination de Gauss avec PVM. L'élimination de Gauss est un algorithme de triangulation de matrice qui peut être utilisé pour trouver les solutions d'un système d'équation linéaires du type A.X = B, avec :

- A est une matrice N x N ;
- X et B deux vecteurs de taille N.

L'algorithme consiste à annuler la partie triangulaire inférieure en itérant N-1 étapes (k=0,... n-2) qui annulent les élements i = k+1, N-1 de la colonne k de la matrice par substitution de la ligne k avec les lignes i = k+1 ... N-1.

Ressources : fichiers *gauss.c* (séquentiel) et *tokenring − sibling.c*.

Note : Tous les schémas explicatifs de ce document ont été réalisés avec Lucidchart.

1.2 L'élimination de Gauss

1.2.1 Rappel

Prenons comme exemple la matrice augmentée suivante, associée à un système donné :

$$A = \left(\begin{array}{ccc|c} 1 & -1 & -4 & 3 \\ 2 & -3 & 4 & 14 \\ 0 & 2 & 8 & 0 \end{array}\right) (L_0)(L_1)(L_2)$$

Choisissons comme ligne pivot la ligne L_0. On a donc k = 0. On va effectuer des opérations élémentaires sur les autres lignes, de telle sorte à avoir tous les nombres de la colonne k = 0 nuls, excepté pour le nombre pivot a_{kk} soit a_{00} ici. En d'autre termes, il faut donc que les éléments a_{01} et a_{02} valent 0.

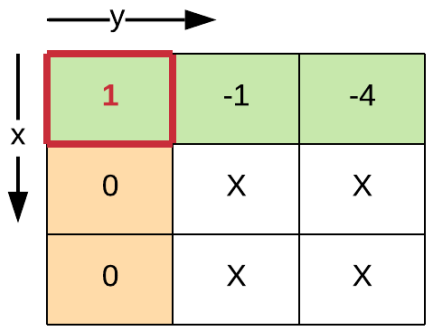

Figure 1: Premier pivot, avec k = 0 : en vert, la ligne pivot, en rouge le nombre pivot et en orange les valeurs à mettre à 0.

Première opération : L_1 <-L_1 - 2L_0 On obtient :

$$A = \left(\begin{array}{ccc|c} 1 & 1 & -4 & 3 \\ 0 & -1 & 12 & 8 \\ 0 & 2 & 8 & 0 \end{array}\right) (L_1)(L_2)(L_3)$$

Ici, nous avons bien a_{01} et a_{02} nuls. On peut alors changer de ligne pivot : on choisit la ligne L_1 avec k = 1. De la même façon, on va réaliser des opérations élémentaires de telle sorte à ce que tous les nombres "en dessous" du pivot $_{kk}$ soit a_{11} ici, soient nuls.

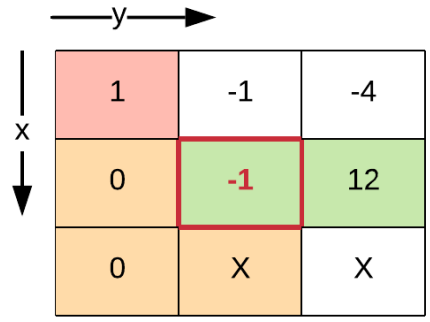

Figure 2: Deuxième pivot, avec k = 1 : en vert, la ligne pivot, en rouge le nombre pivot et en orange les valeurs à mettre à 0.

Deuxième opération : L_1 <-L_2 - 2L_1
On obtient :

$$A = \left(\begin{array}{ccc|c} 1 & -1 & -4 & 3 \\ 0 & -1 & 12 & 8 \\ 0 & 0 & 32 & 16 \end{array}\right) (L_0)(L_1)(L_2)$$

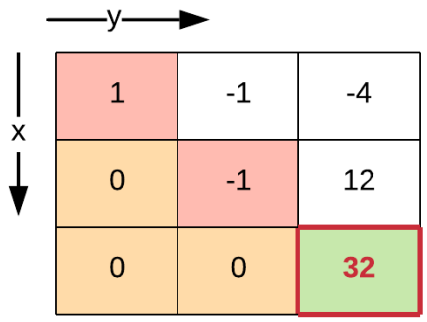

Figure 3: Deuxième pivot, avec k = 3 : en vert, la ligne pivot, en rouge le nombre pivot.

Ici, nous avons bien a_{21} nul. Changeons alors de ligne pivot : on choisit la ligne L_2 avec k = 2. Ici, il n'y a pas d'élément "en dessous" du pivot. De plus, toutes les élements de la matrice triangulaire inférieure valent 0, donc nous avons terminé notre élimination de Gauss.

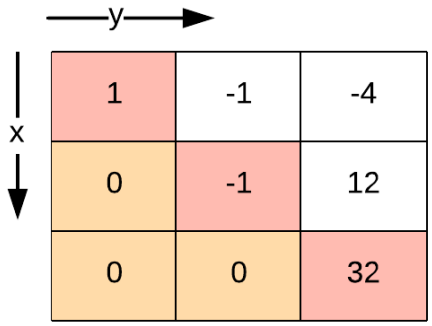

Figure 4: Matrice finale après élimination de Gauss. Tous les pivots sont situés sur la diagonale (en rouge) et la matrice triangulaire inférieure est nulle. (en orange)

1.2.2 Le programme séquentiel

La fonction gauss non parallélisée accède séquentiellement aux données de la matrice. Chaque ligne pivot est traité l'une après l'autre. Si un pivot est presque nul, cela veut dire que la matrice n'est pas inversible, on sort donc de l'algorithme. La matrice est parcourue en diagonale

- a[i][k] correspond à tab+k+i*N
- a[k][k] correspond à tab+k+k*N
- a[k][j] correspond à tab+j+k*N
- a[i][j] correspond à tab+j+i*N

```
void gauss ( double * tab, int N ) {
  int i,j,k;
  double pivot;

  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; */
    }
  }
  printf ("\n");
}
```

Dans la version non parallélisée chaque ligne est réactualisée à chaque étape du calcul dans la boucle et ne gène pas le bon déroulement du programme. L'enjeu va être de déterminer la meilleur manière de paralléliser ces opérations.

2 Distribution des données sur les processeurs

2.1 Que peut-on paralléliser ?

Pour construire notre programme parallèle, il faut dans un premier temps identifier, dans l'approche séquentielle :

- les opérations intrinsèquement séquentielles ;
- les opérations qui peuvent être parallélisées.

On constate que les étapes k ne peuvent être parallélisées, c'est-à-dire qu'on ne peut pas traiter les lignes pivots parallèlement l'une de l'autre : au contraire, le processus d'élimination de Gauss nous oblige à **traiter les lignes pivots l'une après l'autre**. En effet, pour mettre à 0 les nombres de la colonne en dessous du nombre pivot, il faut **modifier** les autres lignes de la matrice initiale. Paralléliser cela n'aurait donc pas de sens, puisque l'on travaillerait avec des lignes "non actualisées", non modifiées par les éventuelles opérations élémentaires. **On en conclut donc que les étapes k sont intrinsèquement séquentielles**.

On remarque cependant que l'étape qui consiste à mettre les éléments $a_{k+1,k}$ à $a_{n-1,k}$ à 0 (c'est-à-dire les éléments en dessous du pivot a_{kk}) peut-être parallélisée. En effet, si par exemple on effectue l'élimination de Gauss à la main pour la matrice suivante, et qu'on prend pour ligne pivot la ligne L_0 :

$$A_0 = \begin{pmatrix} 1 & 1 & -4 \\ 2 & -3 & 4 \\ 1 & 2 & 5 \end{pmatrix}$$

Pour rendre nuls les éléments a_{10} et a_{20}, on effectue les opérations élémentaires suivantes :

- L_1 <-L_1 - $2L_0$
- L_2 <-L_2 - L_0

On a alors :

$$A_1 = \begin{pmatrix} 1 & 1 & -4 \\ 0 & -1 & 12 \\ 0 & 1 & 9 \end{pmatrix}$$

En réalité, **l'ordre de ces 2 opérations importe peu** : que l'on traite la ligne L_1 ou L_2 avant, le résultat sera le même puisque l'on construit la nouvelle matrice A_1 à partir de A_0 . Ces calculs se font donc **indépendamment** des autres lignes à modifier.
Nous allons donc pouvoir paralléliser la sous-matrice (les boucles intérieures du programme).

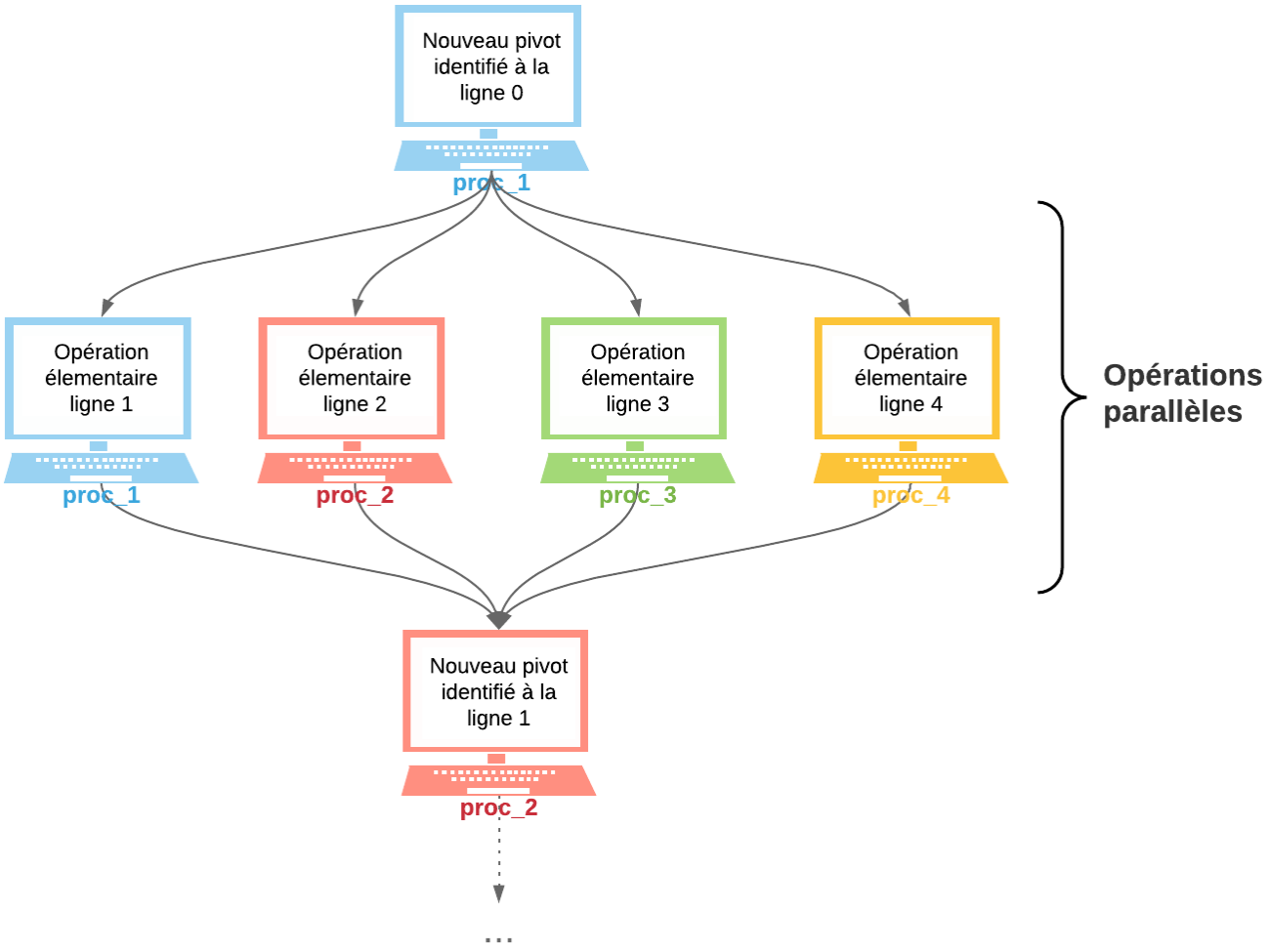

Figure 5: Les étapes à paralléliser et les étapes séquentielles à titre indicatif (celles-ci peuvent être amenées à être modifiées dans la suite du programme) pour une matrice carrée avec n = 4. Nous utilisons 4 processeurs : le processeur 1 réalise les opérations séquentielles.

2.2 Une première approche

Maintenant que nous savons quels calculs nous allons paralléliser, il faut maintenant réfléchir à la distribution des données entre les processeurs : quel processeur va s'occuper du calcul de telle ou telle ligne ?

Pour répondre à cette question, prenons pour exemple une matrice carrée avec n = 8 (matrice de petite taille). Nous avons 4 processeurs à notre disposition : P_0, P_1, P_2, P_4.

Une première idée consiste à affecter les 2 premières lignes de la matrice à P_0 ; les 2 suivantes sont prises en charge par P_1, celles d'après par P_2 etc, comme illustré ci-dessous :

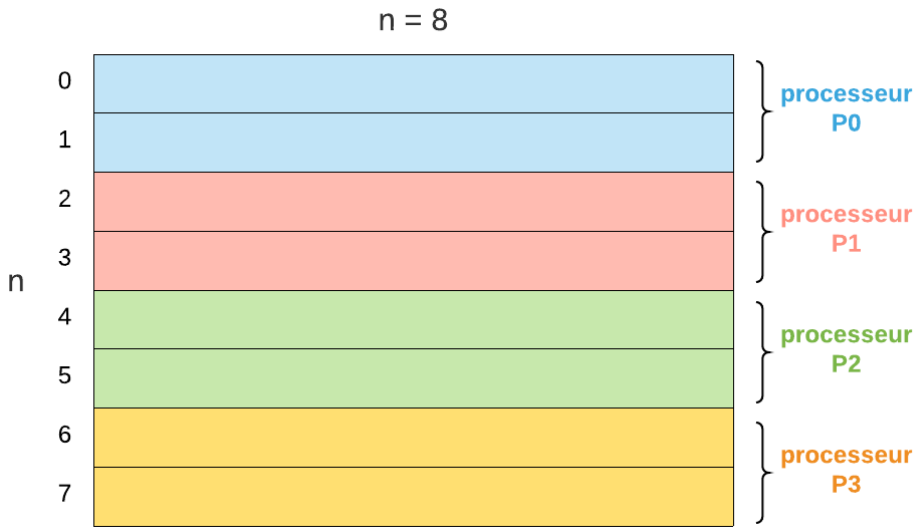

Figure 6: Distribution initiale des ressources, approche 2.

Pour évaluer la pertinence de notre approche, plaçons-nous dans le cas où une partie de la matrice a déjà été traitée par l'algorithme. Dans l'exemple ci-dessous, les itérations $k = 0$, $k = 1$ et $k = 3$ on déjà été calculés. (On rappelle que les pivots potentiels sont situés sur la diagonale de la matrice.)

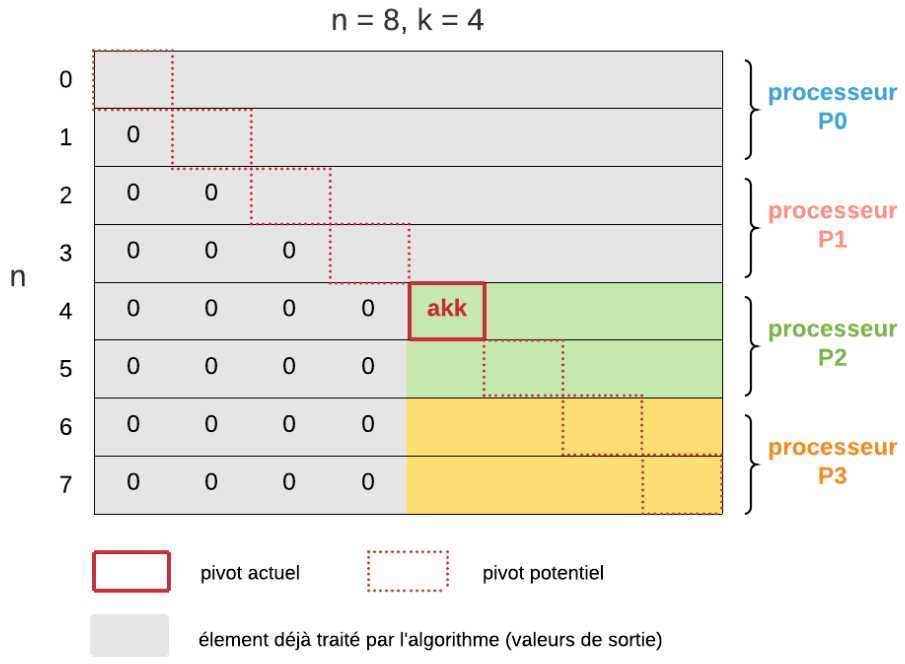

Figure 7: Distribution des ressources avec $k = 4$, en milieu d'exécution, approche 1.

On constate qu'à $k = 4$, seuls 2 processeurs sur 4 disponibles sont utilisés pour le calcul.

En effet, les processeurs P_0 et P_1 se chargeaient du traitement des premières lignes, et à l'étape $k = 4$, ces lignes qu'on leur avait affectées n'interviennent plus dans le calcul de la sous-matrice. On a rapidement perdu en parallélisme.

Avec cette approche, on perd en capacité de calcul proportionnellement à k.

2.3 Une seconde approche

Intéressons-nous alors à une seconde approche de distribution des lignes. Cette fois-ci, les 4 premières lignes sont distribuées entre les 4 processeurs, et nous procédons de la même façon pour les lignes suivantes. Ainsi, la ligne i, appartiendra au processeur p = i % NPROC (avec NPROC le nombre total de processeurs).

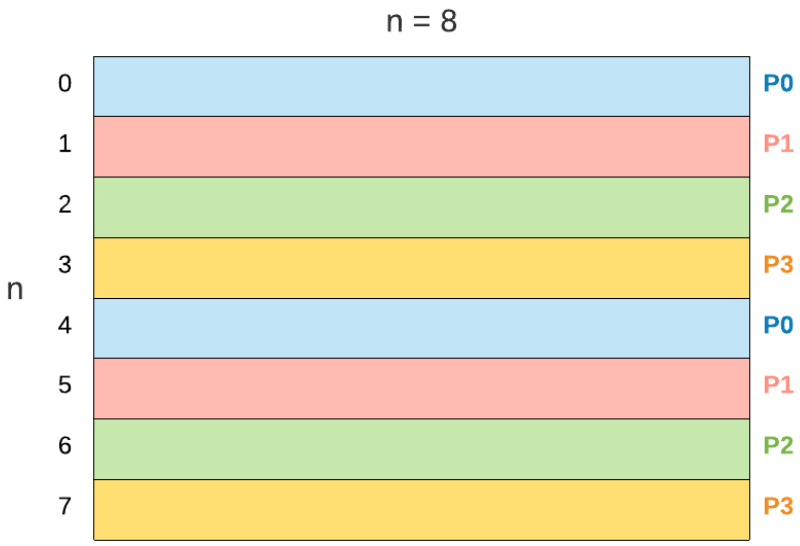

Figure 8: Distribution initiale des ressources, approche 2.

Lorsque $k = 4$, nous exploiterons encore toutes les ressources disponibles, comme illustré ci-dessous.

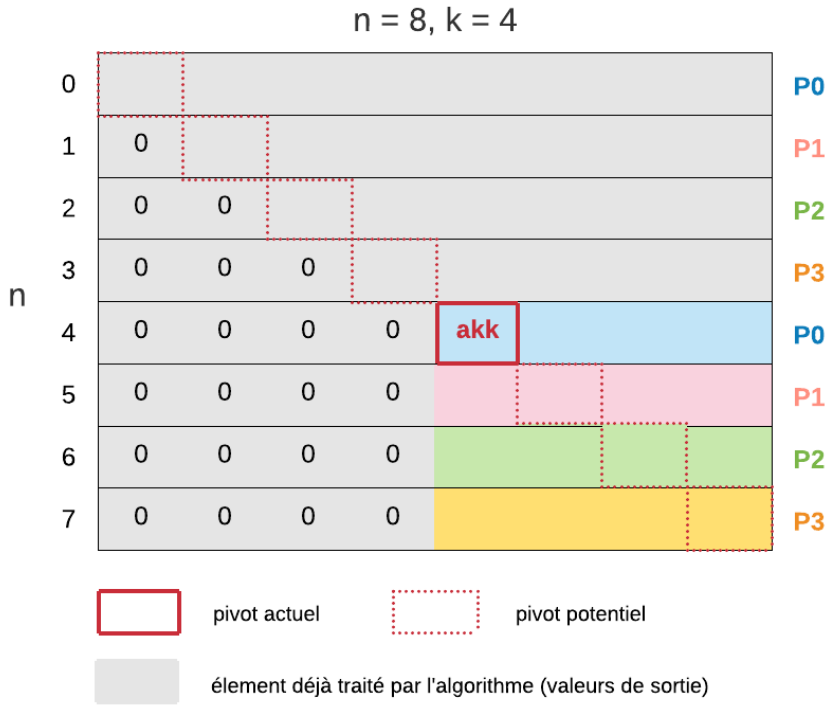

Figure 9: Distribution des ressources avec $k = 4$, en milieu d'exécution, approche 2.

On note cependant que quelque soit l'approche choisie, on perdra progressivement en parallélisme sur les k = NPROC dernières itérations.

3 Parallélisation de l'élimination de Gauss

3.1 Chargement et sauvegarde des données

Nous créons un fichier .txt contenant les valeurs de la matrice d'entrée qui est lu par notre programme. A la fin de l'algorithme, nous générons un fichier de sortie pour sauvegarder les résultats du calcul. Pour cela, nous avons créé deux fonctions qui traitent la lecture et écriture des fichiers en parallèle. (Confère annexe)

3.1.1 Chargement des données

Nous donnons au processeur d'index 0 la responsabilité de lire le fichier d'entrée et d'envoyer aux autres processeurs les lignes qui leur appartiennent. La condition *si me = 0* vérifie si le processeur qui exécute le programme est le celui d'index 0. Si c'est le cas, on a $me = 0$, et le processeur 0 ouvre le fichier d'entrée pour en faire la lecture.

Ensuite, à chaque ligne i de la matrice que le processeur lit, on calcule le numéro du processeur qui se chargera de faire les traitements sur la ligne en calculant $p = i\%NPROC$ (NPROC étant le nombre total de processeurs). La lecture de la ligne est gérée par le processeur 0. Si cette ligne lui appartient, (condition $p = 0$ vérifiée), on met la ligne dans notre tableau local. Sinon, on l'envoie au processeur p. Nous utilisons le tableau *tids* pour avoir le bon *tid* de chaque processeur.

À la fin, chaque processeur aura son tableau rempli avec toutes les lignes qui lui appartiennent.

```
1  void matrix_load ( char nom[], double *a_p, int N, int NPROC, int me, int tids
       []) {
2    FILE *f;
3    int i,j, l=0;
4    int p;
5    int msgtag = 4;
6
7    // Initialize the data matrix
8    double* data = (double*) malloc(N*sizeof(double));
9
10
11
12   // Assign rows to each processor
13   if(me == 0){
14     if ((f = fopen (nom, "r")) == NULL) {
15         perror ("matrix_load : fopen ");
16     }
17   }
18
19
20   for (i=0; i<N; i++)
21   {
22     p = i%NPROC; // numero du processeur a qui on doit envoyer la ligne i
23     if(me == 0){
```

```
24        for (j=0; j<N; j++)
25        {
26          fscanf (f, "%lf", (data+j));
27        }
28
29        if (p == 0)
30        {
31          memcpy((a_p+l*N), data, N*sizeof(double));
32          l++;
33        }
34
35        else{
36
37          pvm_initsend( PvmDataDefault ); // met a 0 le buffer d'envoi
38          pvm_pkdouble(data,N, 1 ); // met dans le paquet
39          pvm_send(tids[p],msgtag); //envoi a p
40
41        }
42
43      }
44      else{
45
46        if(me == p)
47        {
48          pvm_recv(tids[0], msgtag); // recoir du processeur 0
49          pvm_upkdouble( a_p+l*N, N, 1 ); // recoir la ligne
50          l++;
51        }
52      }
53
54    }
55
56    if(me == 0){
57      fclose(f);
58      }
59
60  }
```

3.1.2 Sauvegarde des données

Pour sauvegarder le fichier de résultat, on utilise une nouvelle fois le processeur d'index 0 comme tâche principale. Il s'occupe de l'ouverture et l'écriture du fichier de sortie. L'idée reste la même, si la ligne i appartient à le processeur 0, il l'écrit sur le fichier de sortie, sinon, il doit attendre l'envoi de la bonne ligne du processeur p.

```
1  void matrix_save ( char nom[], double *a_p, int N, int NPROC, int me, int tids
       [])  {
2    FILE *f;
3    int i,j, l = 0;// l = ligne deja lu
4    int msgtag = 4;
5    int p;
```

```
6   // Initialize the data matrix
7   double* data = (double*) malloc(N*sizeof(double));
8
9   if(me == 0) // si je suis 0 j'ecris
10  {
11
12    if ((f = fopen (nom, "w")) == NULL) { perror ("matrix_save : fopen "); }
13
14  }
15
16
17    for (i=0; i<N; i++)
18  {
19    p = i%NPROC; // numéro du processeur à qui on doit envoyer la ligne i
20    if(me == 0)
21    {
22      //si c'est ma ligne, j'ecris dans le fichier
23      if (p != 0)
24      {
25        pvm_recv(tids[p], msgtag);
26        pvm_upkdouble(data, N, 1 );
27      }
28      else
29      {
30        memcpy(data,(a_p+l*N),N*sizeof(double));
31        l++;
32      }
33
34      for (j=0; j<N; j++)
35      {
36        fprintf (f, "%8.2f ", *(data+j) );
37
38      }
39      fprintf (f, "\n");
40
41
42    }
43    else
44    {
45      if (me == p)
46      {
47
48        memcpy(data,(a_p+l*N),N*sizeof(double));
49        l++;
50        pvm_initsend( PvmDataDefault );
51        pvm_pkdouble(data,N, 1 );
52        pvm_send(tids[0],msgtag); //send data to processor 0
53      }
54    }
55  }
56
57
58  if(me == 0)
59    {
```

```
60          fclose (f);
61      }
62
63  }
```

3.1.3 Validation de l'approche avec XPVM

Avec le logiciel XPVM on peut facilement regarder les échanges de message entre les processeurs. Les lignes rouges représentent les envois et réceptions de données. La Figure 10 montre la fenêtre principale du logiciel. Au milieu, on peut voir les quatre machines connectées entre elles et en bas la progression du calcul de chaque tâche avec le temps.

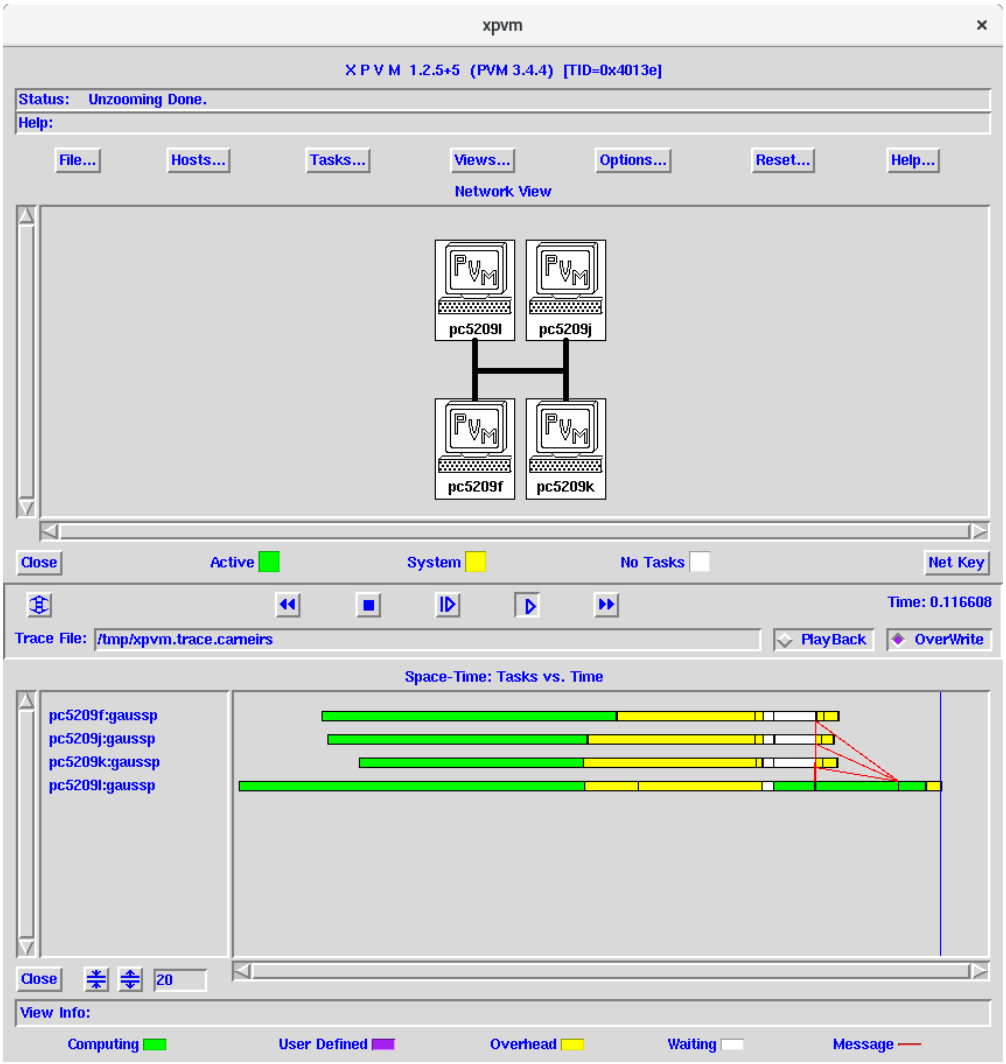

Figure 10: Éxécution de «matrix_load» et «matrix_save» avec une matrice de N = 4 et en utilisant 4 processeurs

Pour faire la validation de notre algorithme de chargement et sauvegarde de la matrice, on a utilisé une petite matrice de N = 4 et on a utilisé les quatres machines disponibles.

Dans ce cas, chaque processeur traitera une seule ligne. Ainsi, le processeur 0 dans la figure correspond à la tâche *pc5209l*. Elle fait l'envoi des lignes à tous les autres processeurs et reçoit d'eux les lignes déjà traitées.

Notre calcul de Gauss en parallèle fonctionne si et seulement si, il retourne exactement la même matrice que le programme *gauss.c* en séquentiel qui nous a été fourni. Nous créeons une fonction $matrix_save_simple$ qui permet d'écrire la matrice résultante dans un fichier.

On utilise la commande linux "diff fichier1 fichier2" pour savoir si les 2 fichiers obtenus par calcul séquentiel et parallèle sont les mêmes :

```
pc5209k:~/pvm3/verif>
pc5209k:~/pvm3/verif> diff matrix_4x4.txt.result output_gaussp.txt
pc5209k:~/pvm3/verif>
```

La commande diff indique que les 2 fichiers sont exactement identiques, notre calcul parallélisé est donc juste. Nous obtenons la même conclusion pour les matrices de plus grande taille.

3.2 Parallélisation du calcul de l'élimination de Gauss

3.2.1 L'algorithme de parallélisation

Comme il a été précisé précédemment, nous ne faisons pas la parallélisation de la boucle k. La variable pk nous indique qui est le processeur en possession de la ligne k. Ce processeur sera responsable de l'envoyer à tous les autres processeurs pour qu'ils aient accès aux informations de la ligne , utiles dans la formule pour le calcul de Gauss. Pour cela, on utilise la fonction *pvm_bcast* qui fait l'envoi à tous les processeurs du groupe concerné : nous réalisons un échange total.

Le processeur pk n'envoie que les colonnes à partir de la colonne k, et pour que les autres processeurs puissent accéder facilement au bon élément, ils reçoivent aussi le message dans un vecteur à partir de la colonne k.

Ensuite, à chaque ligne i, on fait le calcul avec le bon processeur pour chaque colonne j.

```
/**
* Calcule l'elimination de Gauss d'une matrice
* a_p : tableau local
* N : taille de la matrice d'entree
* NPROC : nombre de processeurs
* me : numero du processeur qui execute le programme
* tids[] tableau contenant les tids
*/
void gauss ( double * a_p, int N, int NPROC, int me , int tids[])
{
  int i,j,k, pk, pi;
  double pivot;
  double a[N]; // ligne pivot
  double akj, akk;
  int msgtag = 5;

```

```
17   for ( k=0; k<N-1; k++ )
18   {
19
20     pk = k%NPROC; // numero du processeur qui a la ligne pivot k
21
22     // Recuperer akk
23     if (me == pk) // si je suis le processeur qui a le pivot
24     {
25       memcpy(&a[k],(a_p+k+(k/NPROC)*N),(N-k)*sizeof(double)); // copie la
     ligne pivot dans le buffer de donnees k/___ : la bonne ligne-pivot dans le
      processeur pk
26       pvm_initsend( PvmDataDefault ); // met a 0 le buffer d'envoi
27       pvm_pkdouble(&a[k],(N-k), 1 ); // copie les (N-k) dernieres colonnes a
     partir de la colonne d'indice k
28       pvm_bcast(GRPNAME, msgtag);  // broadcast : echange total
29     }
30     else // si je ne l'ai pas
31     {
32       pvm_recv(tids[pk], msgtag);
33       pvm_upkdouble( &a[k], N-k, 1 );
34     }
35
36
37     for (i=k+1; i<N; i++) // on itere sur les lignes i en dessous de la ligne
     pivot
38     {
39         pi = i%NPROC;
40         if (me == pi) // si je suis le processeur qui a la ligne i
41           {
42
43             akk = a[k];
44             pivot =  *(a_p+k+(i/NPROC)*N)/akk; // la bonne ligne que l'on traite
      dans le processeur pi (qui n'a pas la ligne pivot)
45
46             for (j=k; j<N; j++) // pour chaque colonne de la ligne i
47             {
48               akj = a[j];
49             *(a_p+j+(i/NPROC)*N) = *(a_p+j+(i/NPROC)*N) - (pivot*akj);
50           }
51            }
52     }
53   }
54 }
```

Pour faciliter le calcul, on a utilisé des matrices qui n'ont pas de 0 dans leurs diagonales (pivot nul). Si on avait traité le cas où un pivot était nul, il aurait fallu :

- Mettre à 0 la colonne en dessous du pivot nul ;
- Trouver, une autre ligne pivot dans la colonne akk qui ne soit pas nul ;
- Sur la colonne k, il faut trouver le premier pivot non nul. Il aurait donc fallu faire un processus d'élection.

3.2.2 Validation de l'approche avec XPVM

En utilisant la même matrice de taille N = 4 d'auparavant, on regarde maintenant l'utilisation de la fonction *gauss* implémentée. On peut voir qu'il y a plus d'attentes à cause des échanges d'informations/communications (les lignes pivots).

Malheureusement, les communications effectuées lors de la fonction «broadcast» ne sont pas affichées comme des flèches rouges, comme c'est le cas pour les fonctions «send» par exemple. Cependant, on peut constater le bon fonctionnement de l'algorithme avec le fichier de sortie.

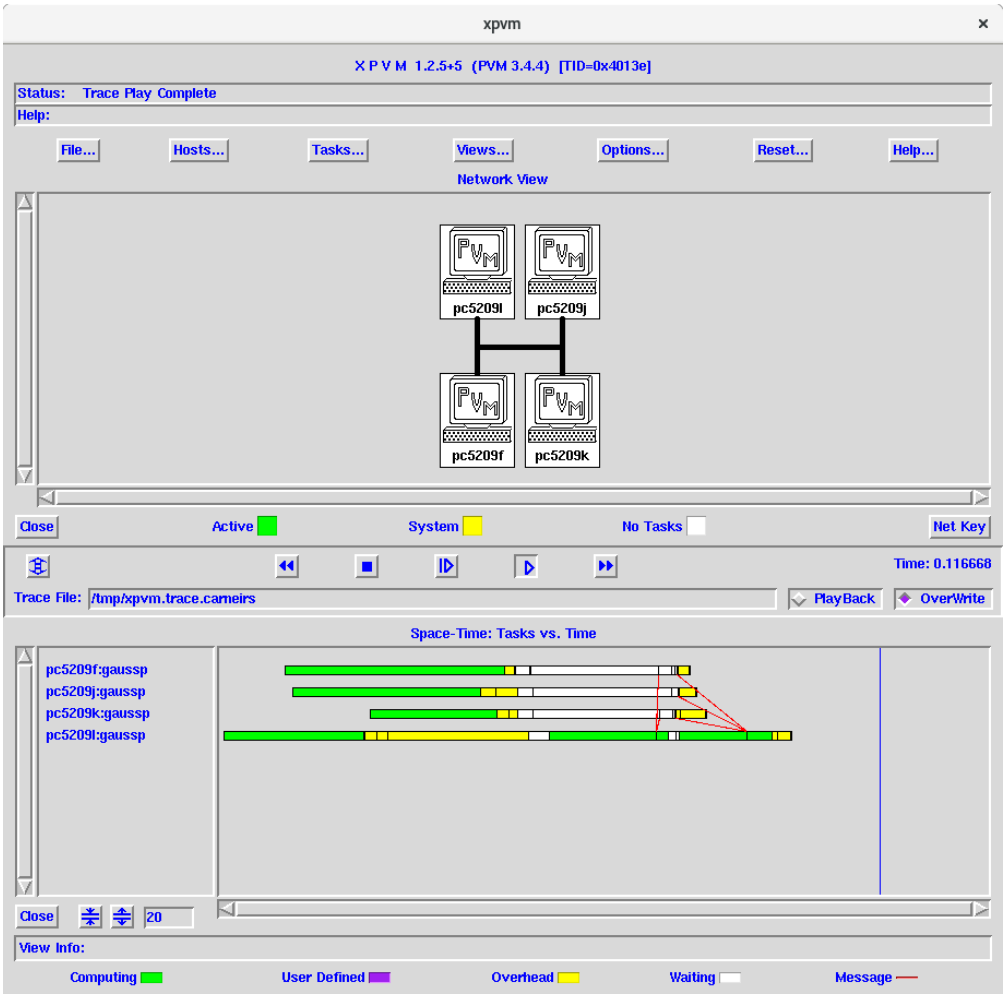

Figure 11: Exécution de l'algorithme complet (load, gauss et save) avec une matrice de N = 4 et en utilisant 4 processeurs
.Dans un cas «idéal», on aimerait qu'il y ait le plus de parties vertes possibles (computing) et le moins de parties blanches (waiting - attente)

4 Mesures de speedup et d'efficacité

Note : Dans cette partie, on distinguera 3 notions :

- le temps d'exécution : temps pris pour exécuter le programme ;
- le temps de calcul : temps d'exécution sans prendre en compte le temps de communication ;
- le temps/coût de communication : temps qui correspond à la communication entre les machines qui exécutent le programme,

4.1 Mesures du temps d'exécution et du speedup pour un réseau

4.1.1 Mesures du temps d'exécution

Pour évaluer les performances de notre algorithme parallèle, nous avons utilisé cinq matrices de taille N = 2^9 ... 2^{13}. Pour chaque matrice, nous avons pris les mesures avec un nombre de processeurs NPROC différent, à savoir avec NPROC = 1, 2, 4 et 8. Pour cette partie, il est important de ne rien imprimer (pas de printf), ne pas sauvegarder la matrice finale dans un fichier et utiliser PVM comme interface.

En comparant les temps d'exécution entre eux, nous obtenons les résultats suivants :

	NxN				
NPROC	**512x512**	**1024x1024**	**2048x2048**	**4096x4096**	**8192x8192**
1	0,44019	3,231435	19,844212	156,603299	1188,102253
2	0,59916	2,391952	13,435609	90,12059	660,600728
4	1,280189	2,482738	8,7201	50,160006	353,76619
8	1,440227	2,283325	7,280103	32,760251	206,840641

Table 1: Temps de calcul en secondes de les opérations de *load* et *gauss* avec l'algorithme parallèle proposé (Salle 5209)

Le Tableau 1 présente les temps d'exécution en secondes mesurés pour chaque matrice d'entrée, en faisant varier le nombre de processeurs. Nous pouvons observer qu'avec une matrice de taille N = 2^9 = 512, le coût de communication entre les machines est supérieur au temps de calcul, c'est pourquoi le temps d'exécution croit lorsque le nombre de processeurs augmente. Cependant, quand on a des matrices de taille supérieure, le temps d'exécution avec plusieurs machines est toujours inférieur à celui mesuré une seule machine. Plus la taille de la matrice croit, plus le speedup, la mesure du gain en vitesse, est élevé (jusque 5,77 pour une matrice de taille 8192 x 8192).

Pour mieux visualiser l'évolution du temps d'exécution en fonction du nombre de processeurs utilisés, on a produit deux graphes: le premier avec la courbe du temps d'exécution (Figure 22) et le second avec la courbe du «speedup» (Figure 14).

Nous avons utilisé une échelle logarithmique sur l'axe vertical pour faire l'affichage du graphe, car la différence entre le temps de calcul pour une matrice de taille 512x512 est x10000 inférieure par rapport à une matrice de taille 8192x8192. Ainsi, nous pouvons observer à la fois les grandes et petites variations. On constate que le temps d'exécution diminue lorsque le nombre de processeurs augmente pour les matrices de dimension 2048 x 2048 et celles de tailles supérieures. Pour celles de taille inférieure, le temps augmente ou reste environ le même.

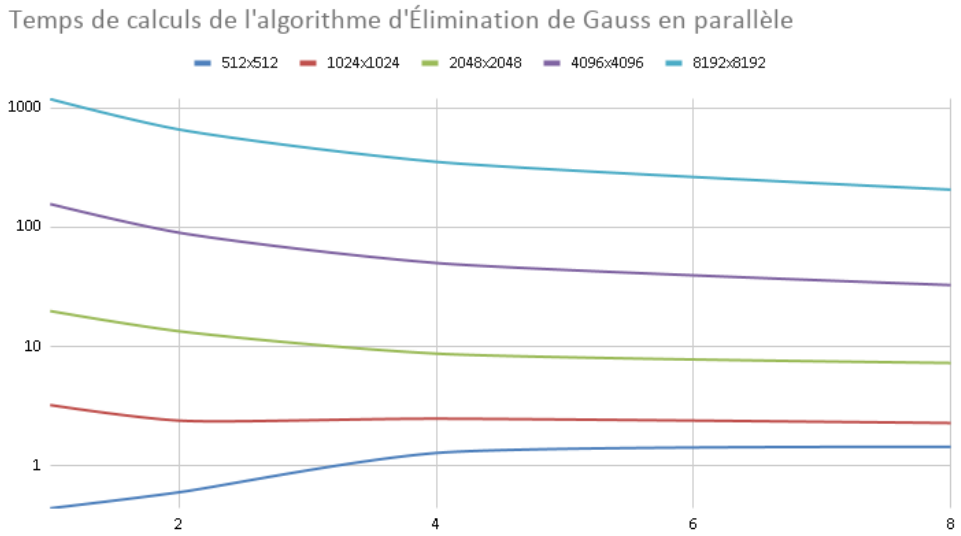

Figure 12: Temps de calcul en échelle logarithmique prise pour chaque matrice par rapport au nombre de processeurs utilisé (Salle 5209)

Le graphique en barres suivant représente les mêmes données, mais permet de mieux se rendre compte visuellement des écarts de temps :

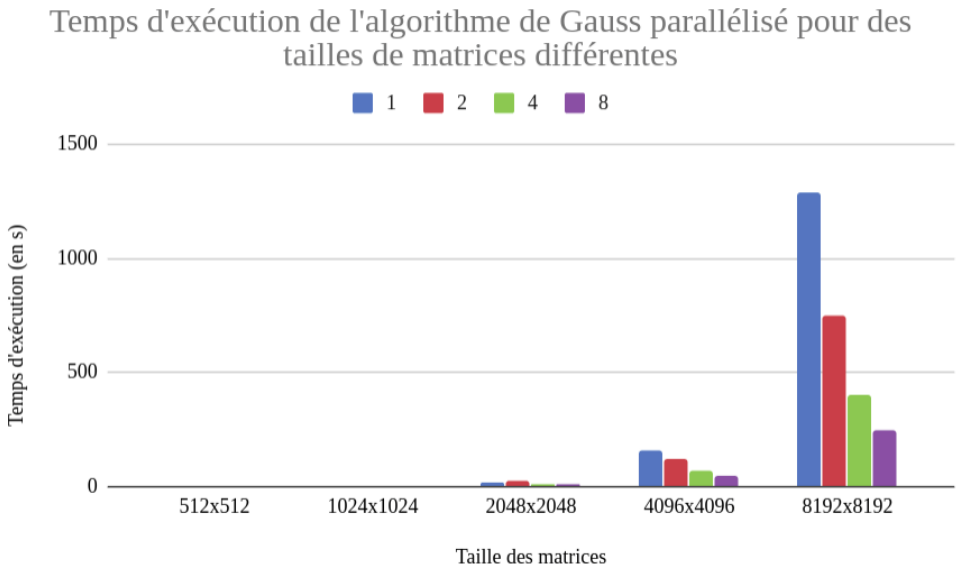

Figure 13: Bargraphe représenant le temps d'exécution de l'algorithme de Gauss parallélisé pour des matrices de taille différentes, et avec un nombre de processeur différent (1,2, 3 et 4.

4.1.2 Mesure du speedup

On rappelle que le Speed up (accélération) est la mesure du gain en vitesse, qui se mesure de la façon suivante : $S_p = \frac{t_1}{t_p}$

avec S_p le speedup calculé, t_1 le temps d'exécution avec un seul processeur t et t_p le temps d'exécution avec p processeurs.

En faisant l'affichage du «Speedup» par rapport au nombre de processeur, on peut bien visualiser l'optimisation obtenue. On peut constater que, par exemple, en utilisant 8 processeurs au lieu de 1, le calcul de la matrice 8192x8192 est cinq fois plus rapide.

Les résultats obtenus pour le speedup sont les suivants :

	Taille de la matrice utilisée N x N				
NPROC	**512x512**	**1024x1024**	**2048x2048**	**4096x4096**	**8192x8192**
1	1	1	1	1	1
2	0,73467855	1,350961474	1,476986417	1,803139982	1,798517929
4	0,3438476662	1,301561019	2,275686288	3,239633564	3,358439236
8	0,3056393194	1,415232172	2,725814731	4,960280646	5,744046466

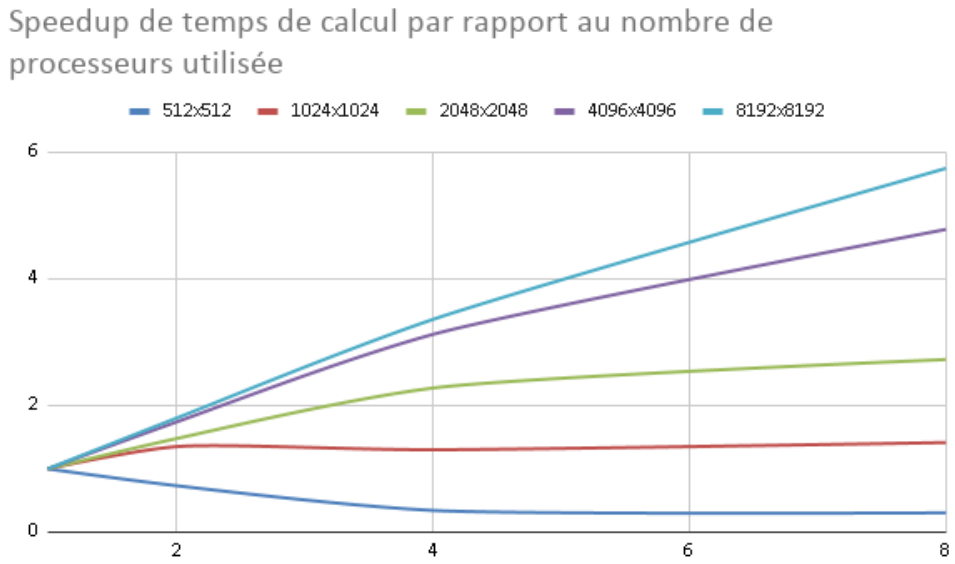

Figure 14: Speedup du temps de calcul par rapport au nombre de processeurs utilisé (Salle 5209)

Il est également important de noter qu'avec nos expérimentations, nous n'avons pas pu regarder ce qu'il se passe lorsque que plus de 8 processeurs sont mobilisés. Le résultat obtenu avec 8 processeurs ne correspond donc certainement pas au speedup maximum. Si on ajoute une n-ième machine de travail et qu'on se rend compte que le temps de calcul est inférieur au temps de communication, alors cela signifie que l'on a atteint le speedup maximal avec (n-1) machines.

4.2 Mesure d'efficacité pour un réseau

Le speedup mesure le gain en temps, mais il ne nous donne aucune indication sur la qualité de notre parallélisation, c'est-à-dire comment le programme parallèle utilise les processeurs alloués à la tâche.

Pour cela, il faut mesurer l'efficacité :

$E_p = \frac{1}{p}.S_p = \frac{t_1}{p.t_p}$

avec S_p le speedup calculé, t_1 le temps d'exécution avec un seul processeur t et t_p le temps d'exécution avec p processeurs.

Nous obtenons les résultats suivants :

	Taille de la matrice utilisée NxN				
NPROC	**512x512**	**1024x1024**	**2048x2048**	**4096x4096**	**8192x8192**
1	1	1	1	1	1
2	0,367339275	0,6754807371	0,7384932086	0,9015699908	0,8992589643
4	0,08596191656	0,3253902546	0,568921572	0,809908391	0,8396098091
8	0,03820491492	0,1769040215	0,3407268414	0,6200350808	0,7180058083

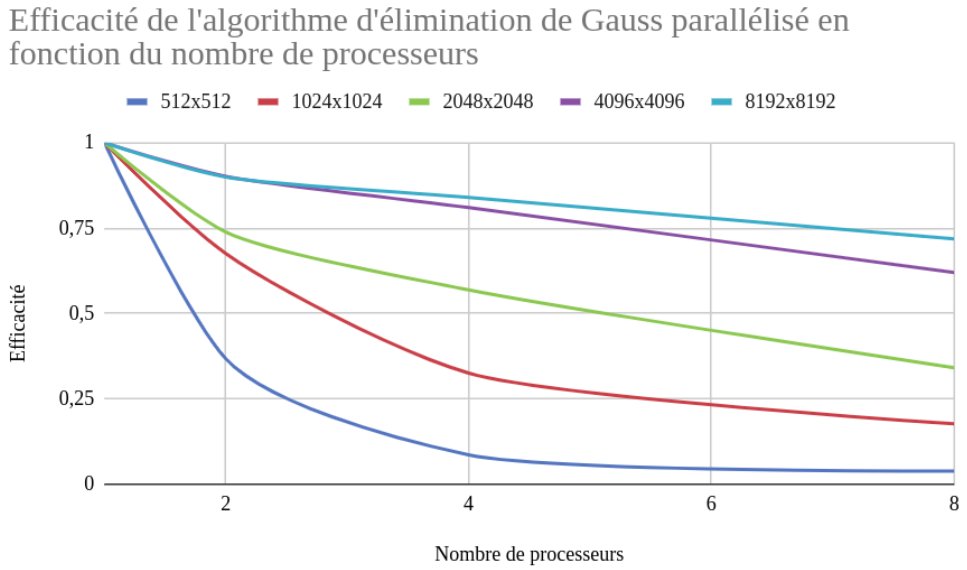

Figure 15: Efficacite de l'algorithme d'élimination de Gauss parallélisé en fonction du nombre de processeurs

L'efficacité correspond à un pourcentage, donc on aura toujours $E_p \leq 1$.
Si on a $E_p = 100\% = 1$, on exploite parfaitement tous les processeurs. On constate que lorsque le nombre de processeurs augmente, l'efficacité diminue. Dans notre cas, E_p diminue très rapidement pour des matrices de taille N inférieur à 1024.

Le nombre de processeurs nuit à l'efficacité : en effet, plus on a de processeurs, plus il est difficile de trouver des tâches à paralléliser.

4.3 Mesure de performances comparatives avec un autre réseau

4.3.1 Mesures de performance pour le deuxième réseau

Voici les résultats que nous obtenons sur des ordinateurs du réseau 5100 de l'ESIEE. Pour pouvoir faire une étude comparative exploitable, nous avons reproduit les mêmes conditions d'expérimentation que pour le réseau 5200, à savoir :

- à priori, pas d'autres utilisateurs connectés sur le réseau (expérimentations en heures

creuses)

- Pour 7 des 8 machines utilisées, nous n'avons lancé aucune autre tâche (la 8ème machine servant "d'interface" pour se connecter aux autres)

Nous obtenons les résultats suivants :

Pour le temps d'exécution :

	Taille de la matrice utilisée N x N				
NPROC	**512x512**	**1024x1024**	**2048x2048**	**4096x4096**	**8192x8192**
1	0,425526	2,772425	20,683149	156,603299	1285,292825
2	0,886319	3,343798	21,899783	121,3401487	752,340027
4	0,699818	2,896779	13,000141	70,496095	405,340042
8	0,817036	2,478455	9,670799	44,401558	245,860474

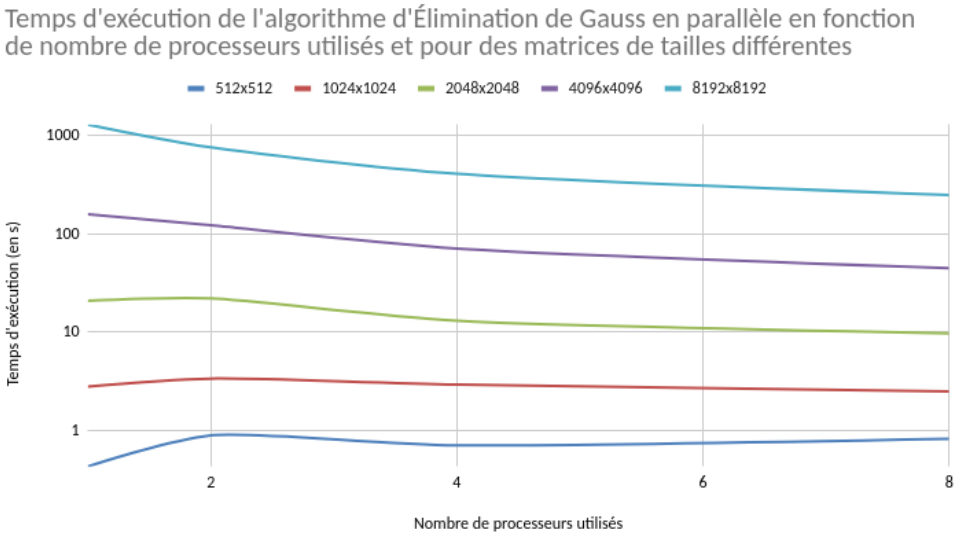

Figure 16: Temps de calcul en échelle logarithmique prise pour chaque matrice par rapport au nombre de processeurs utilisés et pour des matrices de taille différentes (Salle 5101)

Pour le calcul du speedup :

	Taille de la matrice utilisée N x N				
NPROC	**512x512**	**1024x1024**	**2048x2048**	**4096x4096**	**8192x8192**
1	1	1	1	1	1
2	0,480104793	0,8291245464	0,9444453856	1,290614036	1,708393517
4	0,6080523793	0,9570716302	1,590994205	2,221446436	3,170900212
8	0,5208167082	1,118610183	2,138721837	3,526977567	5,227732641

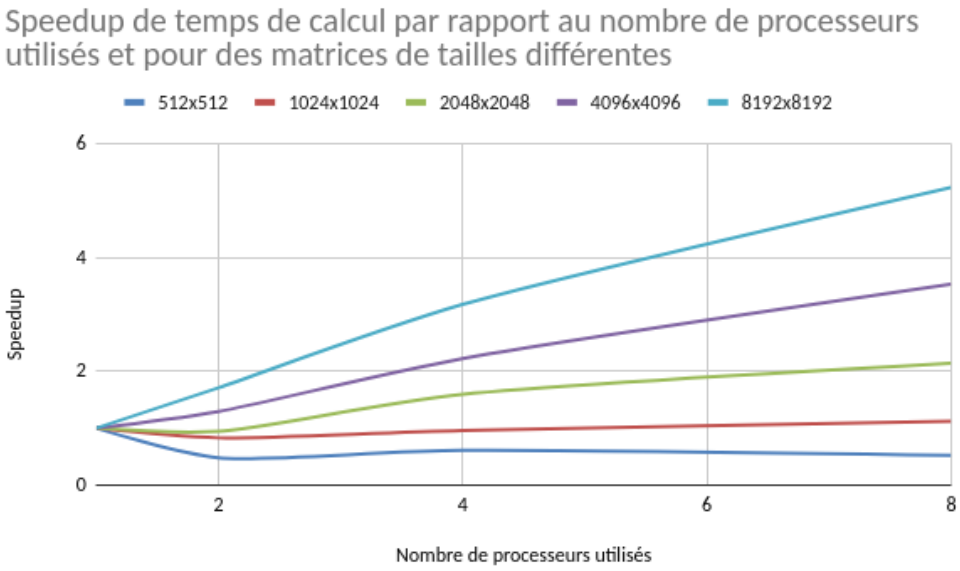

Figure 17: Temps de calcul en échelle logarithmique prise pour chaque matrice par rapport au nombre de processeurs utilisé (Salle 5101)

Il semblerait que les temps d'exécution soient légèrement plus élevés que pour les machines de la salle 5209. Traçons les bargraphes comparatifs suivants pour mieux visualiser cela.

4.3.2 Analyse comparative des résultats

Pour analyser les graphes suivants, on fera attention à l'échelle de l'axe vertical.

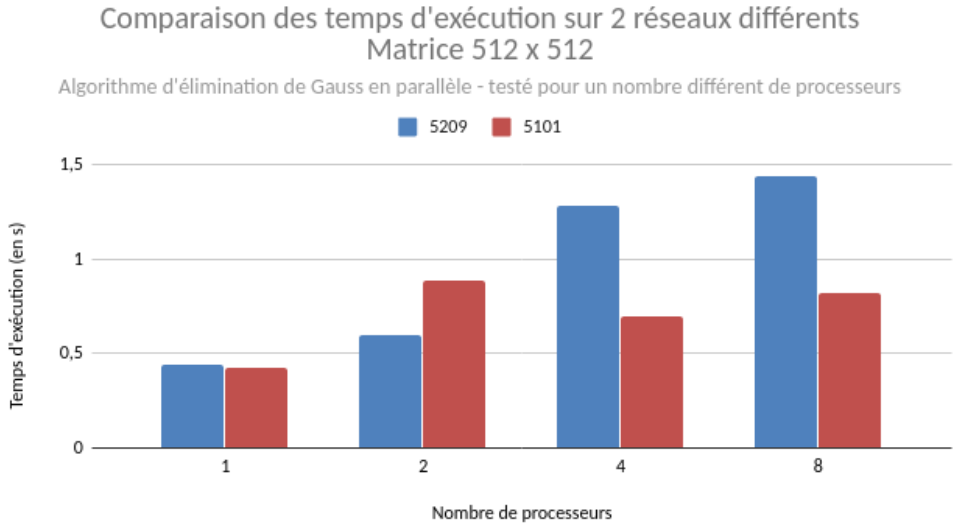

Figure 18: Comparaison des temps d'exécution sur les réseaux des salles 5209 et 5101 (Matrice 512 x 512)

Pour la matrice de taille 512 x 512, on constate que le temps d'exécution est approximativement le même, sauf avec 4 et 8 processeurs où les machines de la salle 5101 semblent prendre 2 fois moins de temps. Toutefois, il faut relativiser ce résultat car cette comparaison se fait à la seconde près.

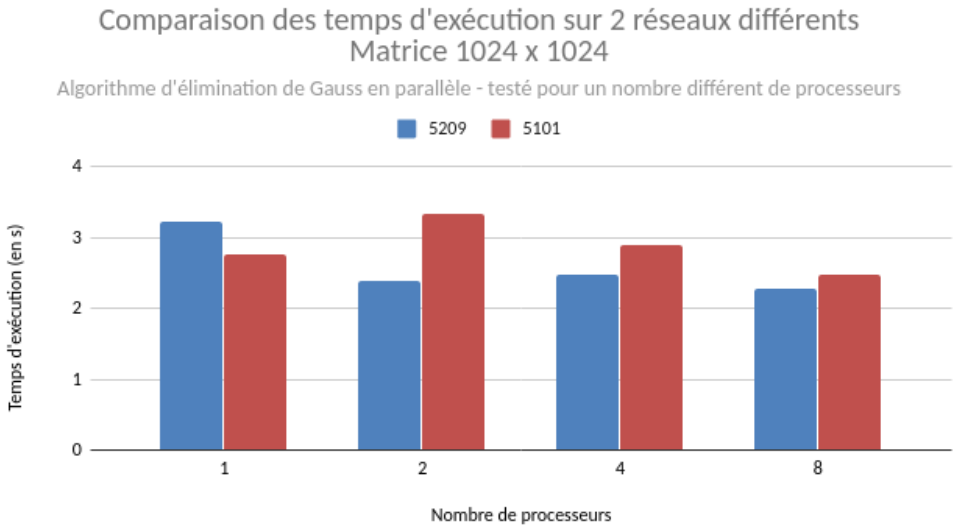

Figure 19: Comparaison des temps d'exécution sur les réseaux des salles 5209 et 5101 (Matrice 1024 x 1024)

En effet, en doublant la taille de la matrice, on remarque que le temps d'exécution est plus long pour le réseau de la salle 5101. Traçons les graphiques pour les matrices de taille supérieure :

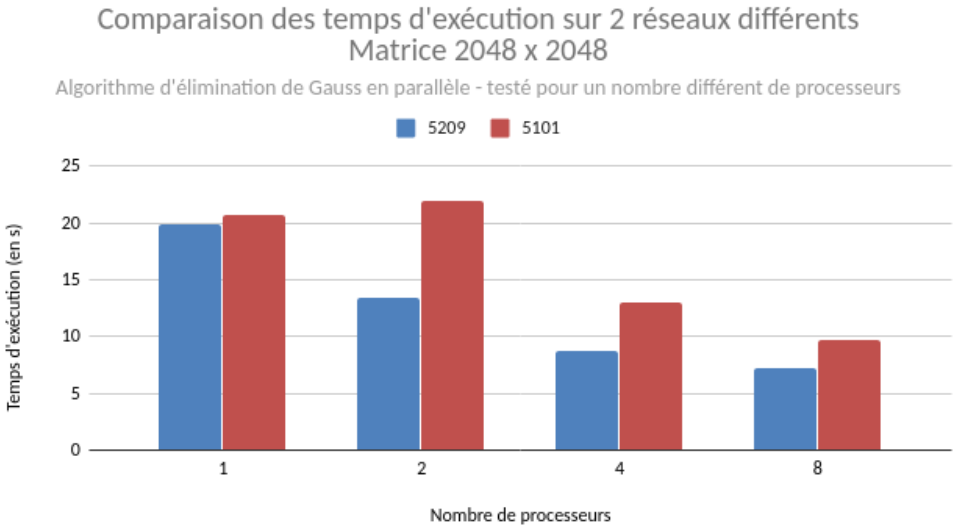

Figure 20: Comparaison des temps d'exécution sur les réseaux des salles 5209 et 5101 (Matrice 2048 x 2048)

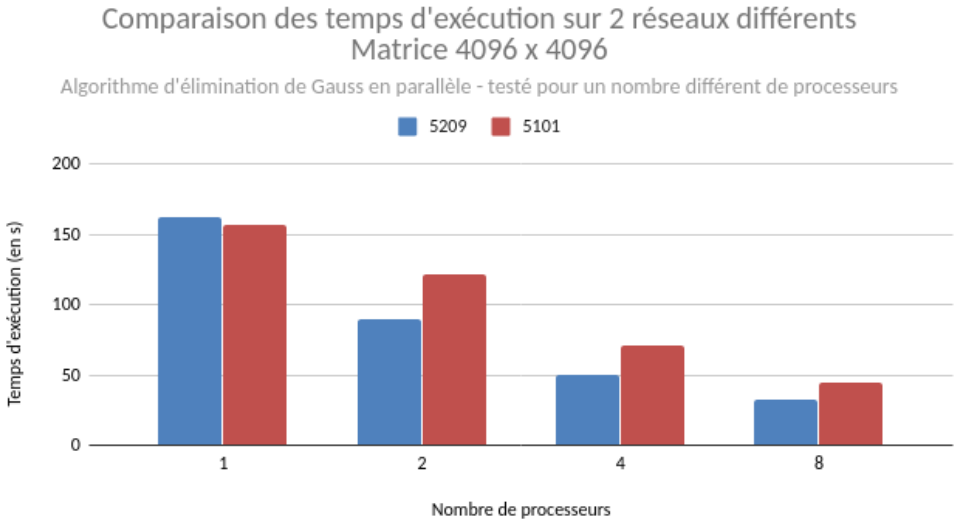

Figure 21: Comparaison des temps d'exécution sur les réseaux des salles 5209 et 5101 (Matrice 4096 x 4096)

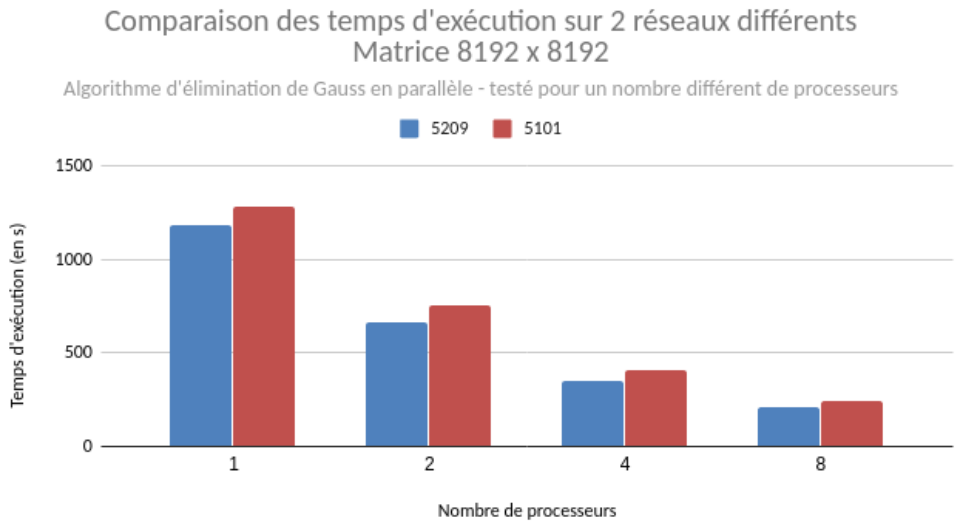

Figure 22: Comparaison des temps d'exécution sur les réseaux des salles 5209 et 5101 (Matrice 8192 x 8192)

On constate que le temps d'exécution est toujours plus elevé pour les machines de la salle 5101. Par exemple pour la matrice N = 8192 et avec 8 processeurs, on n'accélère le temps d'exécution que de 5,23 fois (speedup), alors que le gain est de 5,74 pour le réseau de la salle 5209.

4.3.3 Analyse comparative des architectures matérielles

Pour mieux comprendre ces résultats, intéressons nous à l'architecture matérielle des machines des deux réseaux.

Globalement, on rappelle que plusieurs critères déterminent la puissance et la vitesse d'une machine :

- La fréquence du processeur, qui correspond au nombre de cycles qu'il peut opérer en une seconde ; ainsi la fréquence est élevée, et plus l'ordinateur sera puissant ;
- La quantité de mémoire cache disponible, qui permet de mémoriser temporairement des données provenant d'une source en cours d'utilisation, afin de diminuer le temps d'un accès ultérieur à ces données directement, qui sans la mémoire cache doit se faire dans la mémoire principale ;
- Le nombre de coeurs sur le processeur, qui correspond au nombre d'unité d'unité de traitement ;
- La quantité de mémoire RAM.

Pour obtenir ces informations, on a utilisé les commandes suivantes : *lscpu* (informations générales), *nproc* (nombre de processeurs), $cat/proc/cpuinfo$ (information détaillée pour chaque processeur présent sur la machine), $free - g$ (quantité de mémoire RAM utilisée arrondi).

Ci-dessous un exemple d'exécution de *lscpu* pour le réseau 5100.

```
pc5209i:~> lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              8
On-line CPU(s) list:   0-7
Thread(s) per core: 2
Core(s) per socket: 4
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               60
Model name:          Intel(R) Xeon(R) CPU E3-1245 v3 @ 3.40GHz
Stepping:            3
CPU MHz:             1099.853
CPU max MHz:         3800,0000
CPU min MHz:         800,0000
BogoMIPS:            6784.81
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
```

```
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
```

On constate alors que bien que les machines de la 5101 possèdent plus de mémoire sur la 3ème couche de la mémoire cache (1024K contre 256K), les machines de la salle 5209 possèdent 2 fois plus de processeurs que celles de la 5101 (8 processeurs contre 4), et également une fréquence de processeur plus élevée (fréquence maximale : 3800,0000 Mhz contre 2900 Mhz). Les 2 configurations possèdent tous deux 4 coeurs par processeur.

On résume les principales caractéristiques relevées dans la tableau comparatif suivant :

	Salle 5101	**Salle 5209**
Architecture	64 bits	64 bits
Nombre de processeurs	4	8
Nombre de coeurs par processeur	4	4
Thread par coeur	1	2
Fréquence maximale du processeur	2900 Mhz	3800 Mhz
Mémoire cache	L1d : 32K L1i 32K L2 : 1024K L3 : 8448K	L1d : 32K L1i 32K L2 : 1024K L3 : 8192K
Mémoire RAM	15 GB	7 GB

Les machines de la salle 5209 possèdent donc une meilleure puissance de calcul pour notre programme, ce qui explique le temps d'exécution plus faible par rapport aux machines de la salle 5101.

5 Conclusion

Au cours de ce TP, nous avons proposé une implémentation en parallèle de l'algorithme d'élimination de Gauss, en nous inspirant de la version séquentielle fournie. Pour cela, nous avons utilisé la bibliothèque de passage de messages PVM (en C), qui nous a permis d'utiliser les ressources des processeurs de machines différentes, mais connectées sur le même réseau, en tant qu'une seule machine virtuelle unique. Nous avons testé notre programme sur 8 machines au total, et avons mesuré les performances de notre code en utilisant 2 outils : le speed-up, qui correspond au gain en vitesse, ainsi que l'efficacité.

En utilisant la console graphique de PVM, XPVM, nous avons pu observer les phénomènes et communications entre les machines pendant l'exécution du code, et également s'assurer du bon fonctionnement de notre programme (queue output vide à la fin par exemple)

Ainsi, ce TP nous a permis de mobiliser les connaissances acquises en cours, et également de constater l'intêret de paralléliser le programme dans le cas précis de l'algorithme d'élimination de Gauss.

6 Annexe

6.1 Programme principal

```
1
2  /*  Gauss Parallel using PVM 3.4
3      - uses sibling() to determine the nb of spawned tasks (xpvm and pvm> ok)
4      - uses group for token ring communication
5
6      Stela CARNEIRO ESPINDOLA et Cecile POV
7  */
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12 #include <sys/types.h>
13 #include <math.h>
14 #include "pvm3.h"
15
16 #define GRPNAME "tokenring"
17
18
19
20 void matrix_load ( char nom[], double *a_p, int N, int NPROC, int me, int tids
       [])
21 {
22   FILE *f;
23   int i,j, l=0;
24   int p;
25   int msgtag = 4;
26
27   // Initialize the data matrix
28   double* data = (double*) malloc(N*sizeof(double));
29
30   // Assign rows to each processor
31   if(me == 0){
32     if ((f = fopen (nom, "r")) == NULL) {
33         perror ("matrix_load : fopen ");
34     }
35   }
36
37
38   for (i=0; i<N; i++)
39   {
40     p = i%NPROC; // num du processeur a qui on doit envoyer la ligne i
41     if(me == 0)
42     {
43       for (j=0; j<N; j++)
44       {
45         fscanf (f, "%lf", (data+j));
46       }
47
48       if (p == 0)
```

```
49        {
50          memcpy((a_p+l*N), data, N*sizeof(double));
51          l++;
52        }
53        else
54        {
55
56          pvm_initsend( PvmDataDefault ); // met a 0 le buffer d'envoi
57          pvm_pkdouble(data,N, 1 ); // met dans le paquet
58          pvm_send(tids[p],msgtag); //envoi a p
59
60        }
61      }
62      else
63      {
64        if(me == p)
65        {
66          pvm_recv(tids[0], msgtag); // recoir du processeur 0
67          pvm_upkdouble( a_p+l*N, N, 1 ); // recoir la ligne
68          l++;
69        }
70      }
71    }
72
73    if(me == 0){
74      fclose(f);
75      }
76  }
77
78
79  void matrix_save_simple ( char nom[], double *tab, int N ) {
80    FILE *f;
81    int i,j;
82
83    if ((f = fopen (nom, "w")) == NULL) { perror ("matrix_save : fopen "); }
84    for (i=0; i<N; i++) {
85      for (j=0; j<N; j++) {
86        fprintf (f, "%8.2f ", *(tab+i*N+j) );
87      }
88      fprintf (f, "\n");
89    }
90    fclose (f);
91  }
92
93
94
95  void matrix_save ( char nom[], double *a_p, int N, int NPROC, int me, int tids
       [])  {
96    FILE *f;
97    int i,j, l = 0;// l = ligne deja lu
98    int msgtag = 4;
99    int p;
100   // Initialize the data matrix
101   double* data = (double*) malloc(N*sizeof(double));
```

```
102
103   if(me == 0) // si je suis 0 j'ecris
104   {
105
106     if ((f = fopen (nom, "w")) == NULL) { perror ("matrix_save : fopen "); }
107
108   }
109
110
111     for (i=0; i<N; i++)
112   {
113     p = i%NPROC; // numéro du processeur a qui on doit envoyer la ligne i
114     if(me == 0)
115     {
116       //si c'est ma ligne, j'ecris dans le fichier
117       if (p != 0)
118       {
119         pvm_recv(tids[p], msgtag);
120         pvm_upkdouble(data, N, 1 );
121       }
122       else
123       {
124         memcpy(data,(a_p+l*N),N*sizeof(double));
125         l++;
126       }
127
128       for (j=0; j<N; j++)
129       {
130         fprintf (f, "%8.2f ", *(data+j) );
131
132       }
133       fprintf (f, "\n");
134
135
136     }
137     else
138     {
139       if (me == p)
140       {
141
142         memcpy(data,(a_p+l*N),N*sizeof(double));
143         l++;
144         pvm_initsend( PvmDataDefault );
145         pvm_pkdouble(data,N, 1 );
146         pvm_send(tids[0],msgtag); //send data to processor 0
147       }
148     }
149   }
150
151
152   if(me == 0)
153     {
154       fclose (f);
155     }
```

```
156
157 }
158
159
160
161 /**
162 * Calcule l'elimination de Gauss d'une matrice
163 * a_p : tableau local
164 * N : taille de la matrice d'entree
165 * NPROC : nombre de processeurs
166 * me : numero du processeur qui execute le programme
167 * tids[] tableau contenant les tids
168 */
169 void gauss ( double * a_p, int N, int NPROC, int me , int tids[])
170 {
171         int i,j,k, pk, pi;
172         double pivot;
173         double a[N]; // ligne pivot
174         double akj, akk;
175         int msgtag = 5;
176
177         for ( k=0; k<N-1; k++ )
178         {
179
180           pk = k%NPROC; // numero du processeur qui a la ligne pivot k
181
182           // Recuperer akk
183           if (me == pk) // si je suis le processeur qui a le pivot
184           {
185             memcpy(&a[k],(a_p+k+(k/NPROC)*N),(N-k)*sizeof(double)); // copie
    la ligne pivot dans le buffer de donnees k/___ : la bonne ligne-pivot dans
     le processeur pk
186             pvm_initsend( PvmDataDefault ); // met a 0 le buffer d'envoi
187             pvm_pkdouble(&a[k],(N-k), 1 ); // copie les (N-k) dernieres
    colonnes a partir de la colonne d'indice k
188             pvm_bcast(GRPNAME,msgtag);  // broadcast : echange total
189           }
190           else // si je ne l'ai pas
191           {
192             pvm_recv(tids[pk], msgtag);
193             pvm_upkdouble( &a[k], N-k, 1 );
194           }
195
196
197           for (i=k+1; i<N; i++) // on itere sur les lignes i en dessous de la
    ligne pivot
198           {
199              pi = i%NPROC;
200              if (me == pi) // si je suis le processeur qui a la ligne i
201                {
202
203                 akk = a[k];
204                 pivot =  *(a_p+k+(i/NPROC)*N)/akk; // la bonne ligne que l'on
    traite dans le processeur pi (qui n'a pas la ligne pivot)
```

```
205
206                 for (j=k; j<N; j++) // pour chaque colonne de la ligne i
207                 {
208                   akj = a[j];
209                 *(a_p+j+(i/NPROC)*N) = *(a_p+j+(i/NPROC)*N) - (pivot*akj);
210               }
211                }
212           }
213         }
214 }
215
216
217
218
219 /* Simple example passes a token around a ring */
220 void dowork( int me, int tids[], int nproc, char* file_name, int N)
221 {
222      int token;
223      int src, dest;
224      int count  = 1;
225      int stride = 1;
226      int msgtag = 4;
227
228    double * a_p = (double *) malloc((N/nproc)*N*sizeof(double)); //Initialize
       matrix
229
230
231      matrix_load(file_name, a_p, N, nproc, me, tids);
232      gauss(a_p, N, nproc, me, tids);
233      matrix_save("output_cecile.txt", a_p, N, nproc, me, tids);
234 }
235
236
237
238
239
240
241 void main(int argc, char ** argv)
242 {
243     int NPROC = 8; //= atoi(argv[3]);    /* default nb of proc */
244     int mytid;                  /* my task id */
245     int *tids;                  /* array of task id */
246     int me;                     /* my process number */
247     int i;
248   int N = atoi(argv[1]);
249   char* file_name = argv[2];
250
251
252     /* enroll in pvm */
253     mytid = pvm_mytid();
254
255     /* determine the size of my sibling list */
256     NPROC = pvm_siblings(&tids);
257     printf(" NPROC: %d\n",NPROC);
```

```
258     /* WARNING: tids are in order of spawning, which is different from
259        the task index JOINING the group */
260 
261     me = pvm_joingroup( GRPNAME ); /* me: task index in the group */
262   printf("me: %d \n",me);
263     pvm_barrier( GRPNAME, NPROC );
264     pvm_freezegroup ( GRPNAME, NPROC );
265     for ( i = 0; i < NPROC; i++) {
266       tids[i] = pvm_gettid ( GRPNAME, i);
267     }
268 
269 
270 /*------------------------------------------------------------------------*/
271 /*            all the tasks are equivalent at that point                  */
272 
273      dowork( me, tids, NPROC, file_name, N );
274 
275      pvm_lvgroup( GRPNAME );
276      pvm_exit();
277 
278 }
```

6.2 Programme de génération de matrices aléatoires

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <math.h>
#include <string.h>

// Compiler : gcc -Wall -o rd_matrix random_matrix.c


int mkdir(const char *pathname, mode_t mode);

void save_file(int n, int*matrix)
{
        //create directory
        /*
        struct stat st = {0};

        if (stat("/matrices", &st) == -1)
        {
            mkdir("/matrices", 0700);
        }
        */

        char str_n[10];
        sprintf(str_n, "%d", n);

        char filename[100] = "matrix_";
        strcat (filename, str_n);
        strcat (filename,"x");
        strcat (filename, str_n);
        strcat (filename, ".txt");
        FILE *f = fopen (filename, "w");
        int i,j;


        if (f == NULL)
        {
                perror ("matrix_save : fopen ");
        }
        else
        {
                for(i=0; i<n; i++)
                {
                        for(j=0; j<n; j++)
                        {
                                fprintf(f,"%d ", *(matrix+j+i*n)); // n car nb colonnes
                        }
                        fprintf(f, "\n");
                }
```

```
52          }
53  }
54
55
56
57
58  int random_matrix(int n)
59  {
60          int* memoireAllouee = NULL;
61
62          memoireAllouee = (int *)malloc(sizeof(int)*n*n);
63          if (memoireAllouee == NULL) // On verifie si la memoire a ete allouee
64          {
65
66                  exit(0); // Erreur : on arrete tout !
67          }
68
69          //int rd_matrix[n][n];
70          int i, j;
71
72          srand(time(NULL));
73          for(i = 0; i<n; i++)
74          {
75                  for(j = 0; j<n; j++)
76                  {
77                          if (i != j) { *(memoireAllouee+j+i*n) = rand()%10;    }
    // nombre de 0 a 9
78                          else        { *(memoireAllouee+j+i*n) = (rand()%9)+1;}
    // nombre de 1 a 9
79
80                  }
81          }
82          save_file(n,memoireAllouee);
83          free(memoireAllouee);
84
85          return 0;
86  }
87
88
89
90
91  int main(int argc, char *argv[] )
92  {
93          int i;
94          for (i = 1; i<513; i = i*2)
95          {
96                  printf("%d \n",i);
97                  random_matrix(i);
98
99          }
100 }
```