Parallélisme et calculs distribués

Compte rendu E5

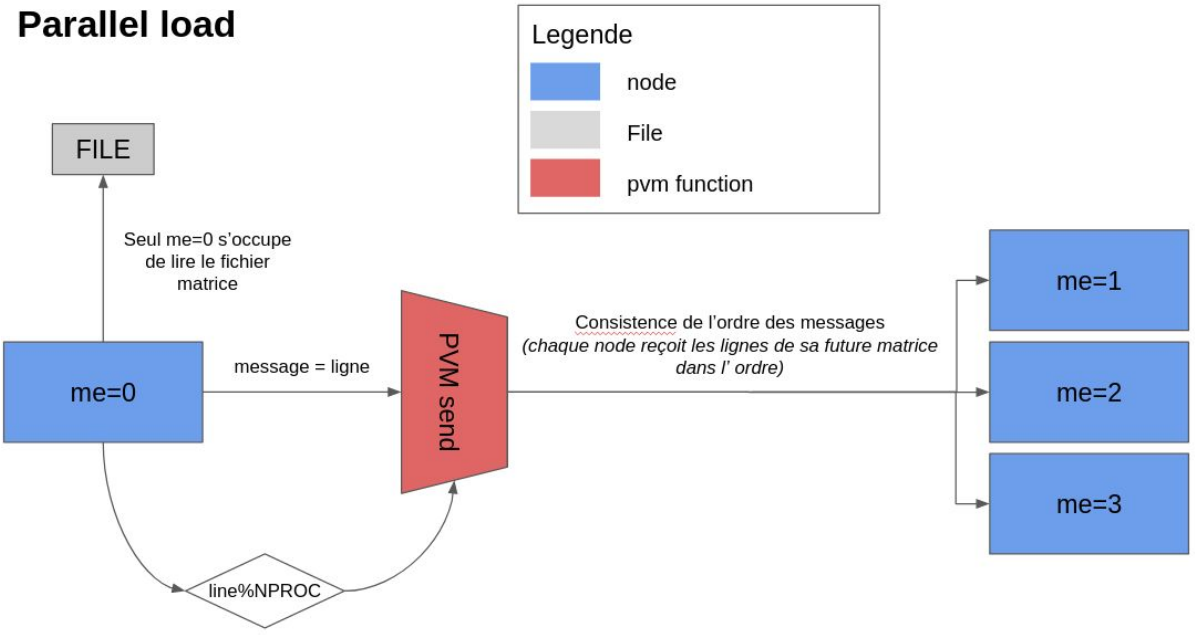
Alexandre Durrmeyer Jules Neghnagh-Chenavas

I. Diffusion et concentration de la matrice

A. Description du code

1. PARALLEL LOAD

Voici un schéma descriptif de ce que fait pload:

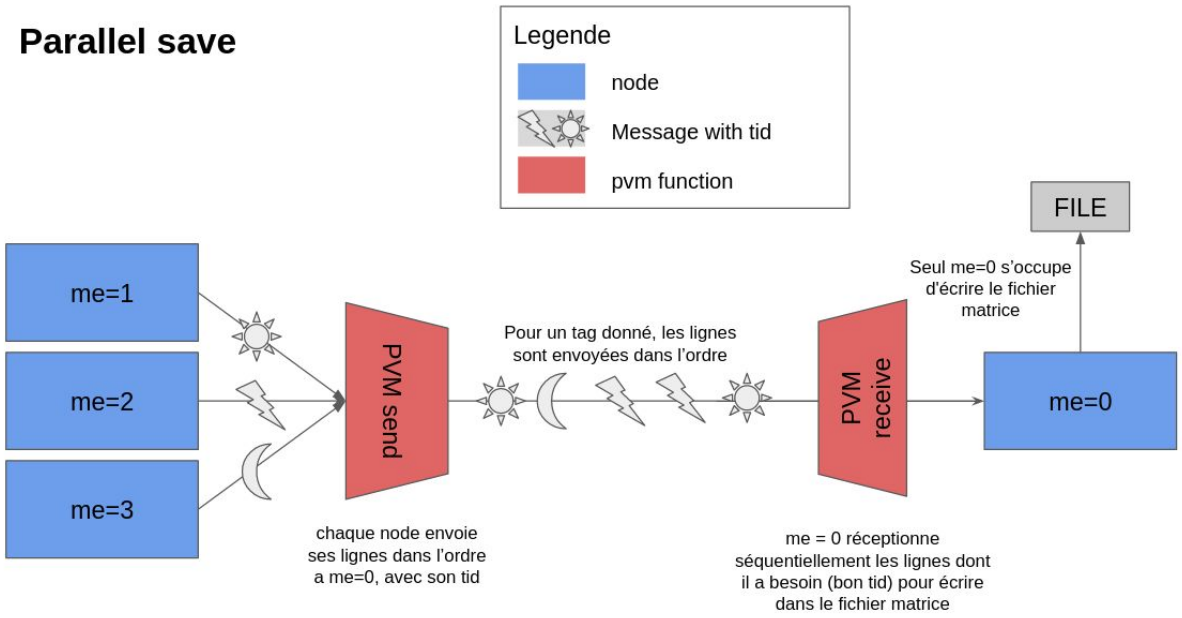


C'est la node 0 qui s'occupe de lire dans les lignes dans le fichier et d'envoyer les bonnes lignes au bonnes nodes, connaissant le nombre de nodes (via la ligne de commande pvm).

Chaque receveur (toutes les nodes sauf 0) vont donc recevoir une liste de ligne. pvm nous garantissant la consistance des messages, nous n'avons pas à nous soucier de l'ordre de réception de ceux-ci.

2. PARALLEL SAVE

Voici un schéma descriptif de ce que fait psave:



C'est me=0 qui s'occupe d'écrire la matrice. Les autres nodes vont envoyer séquentiellement leurs lignes à me=0.

me=0 va se charger d'écrire ligne par ligne dans un fichier résultat, et doit pour cela recevoir la ligne venant de la bonne node à l'aide de la fonction receive.

CODE:

Il s'agit d'exécuter l'algorithme du pivot de gauss en parallèle.

La méthode du pivot de gauss nous fait calculer des valeurs dans une sous matrice de taille N-1 de la matrice précédente M de taille N. Une parallélisation à cet endroit n'est pas judicieuse car le nombre de communications entre processus serait trop grand celles-ci seraient erratiques.

Nous parallélisons le calcul du pivot de gauss pour une sous matrice k donnée.

La méthode du pivot de gauss demande pour chaque élément M_{ij} deux valeurs de la première ligne de la matrice - ligne pivot - (en diagonale M_{kk} et au dessus M_{kj}), ainsi que sa valeur a gauche M_{ik}.

Pour paralléliser, on "découpe" la matrice selon des lignes (une séparation en ligne est idéale car nous aurons juste à envoyer notre ligne pivot aux autres processus, la taille des message est donc plus importante et leur nombre est réduit (coûts d'initialisations non négligeables…) Le processus ayant en mémoire la ligne pivot sachant k se charge de la broadcaster à tous les autres processus.

Nombre de processeurs actifs selon k avec entrelacement

	k=0	k=5	k=9
	0	0	0
	1	1	1
	2	2	2
	3	3	3
repart procs	4	4	4
	5	5	5
	6	6	6
	7	7	7
	8	8	8
	9	9	9
	10	10	10
	11	11	11
Procs actifs	**3**	**3**	**3**

Nombre de processeurs actifs selon k sans entrelacement

k	k=0	k=5	k=9
	0	0	0
	1	1	1
	2	2	2
	3	3	3
	4	4	4
repart procs	5	5	5
	6	6	6
	7	7	7
	8	8	8
	9	9	9
	10	10	10
	11	11	11
Procs actifs	**3**	**2**	**1**

Nous découpons notre matrice de manière entrelacée afin de conserver NPROC processus jusqu'à la fin (si le découpage était par “blocs”, au lieu d’ entrelacé, les dernieres étapes seraient entièrement faites par un seul processeur, ce qui n'a point d’intérêt.

ENTRELACEMENT

k
0
1
2
3
4
5
6
7
8
9
10
11

tab0	
i(p=0)	k stocké
0	0
1	3
2	6
3	9

tab1	
i(p=1)	k stocké
0	1
1	4
2	7
3	10

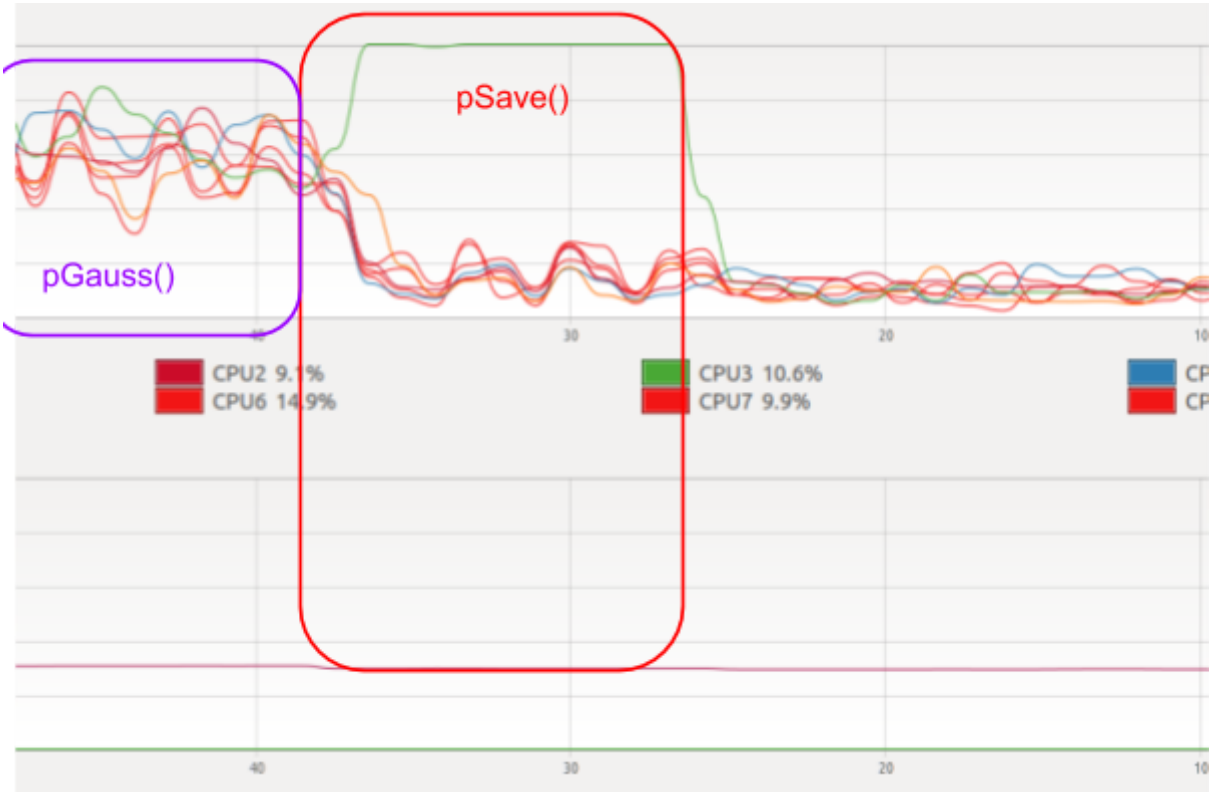
tab 2	
i(p=2)	k stocké
0	2
1	5
2	8
3	11

Le calcul des valeurs demandant des valeurs d' une autre ligne, il est nécessaire de recevoir (et donc d' envoyer) cette ligne a chaque itération de k.

II. Elimination de Gauss

A. Visualisation

On confirme visuellement que pvm utilise tous les processeurs de la machine a 100%.



III. Benchmarking

A. Description du code

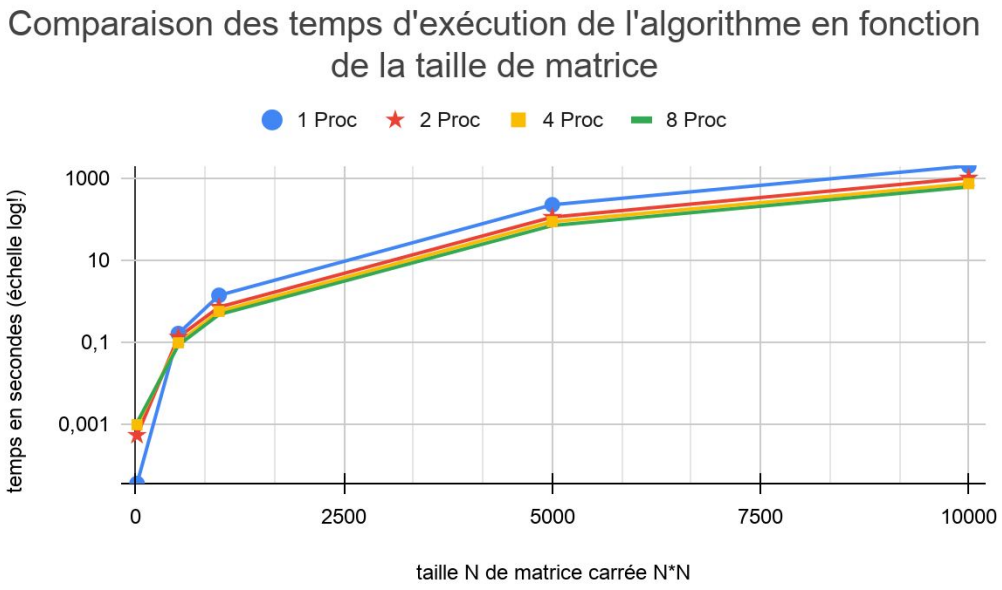
Pour les mesures de temps nous avons calculé le lapse de temps parcouru par la fonction pGauss() en utilisant respectivement les deux timestamp t1 et t2 avant et après l'appel. Lorsqu'il y a plusieurs processeurs nous notons le plus grand écart.
Nous aurions eu la possibilité de prendre le t1 minimum et le t2 maximum pour calculer le temps écoulé mais l'écart entre les processus est minime donc nous n'avons pas pris la peine.

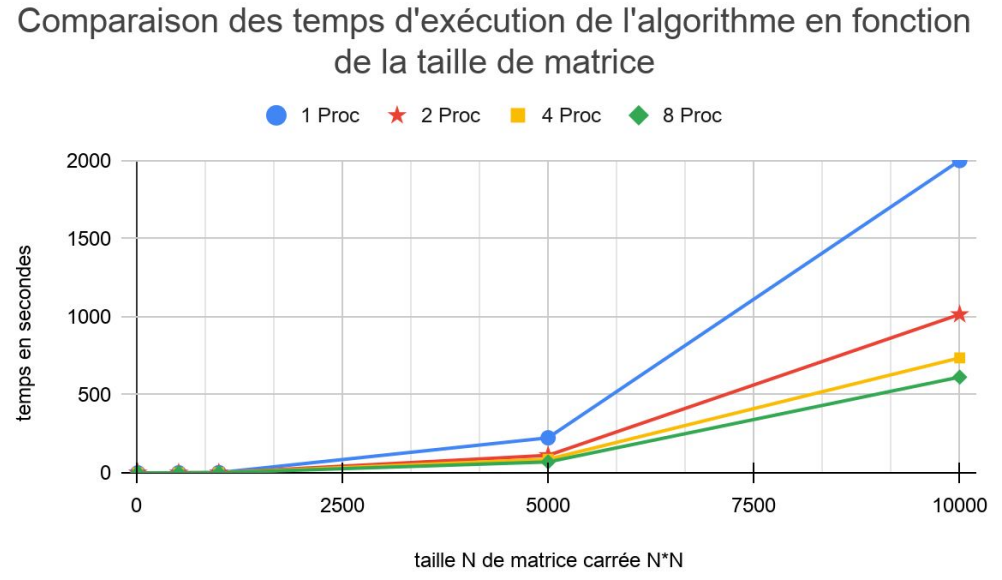
B. Speedup et Efficacité

Au lieu d'utiliser une matrice 500*500 nous avons utilisé une matrice 512*512 afin que celle-ci soit divisible par les 1/2/4/8 processeurs.

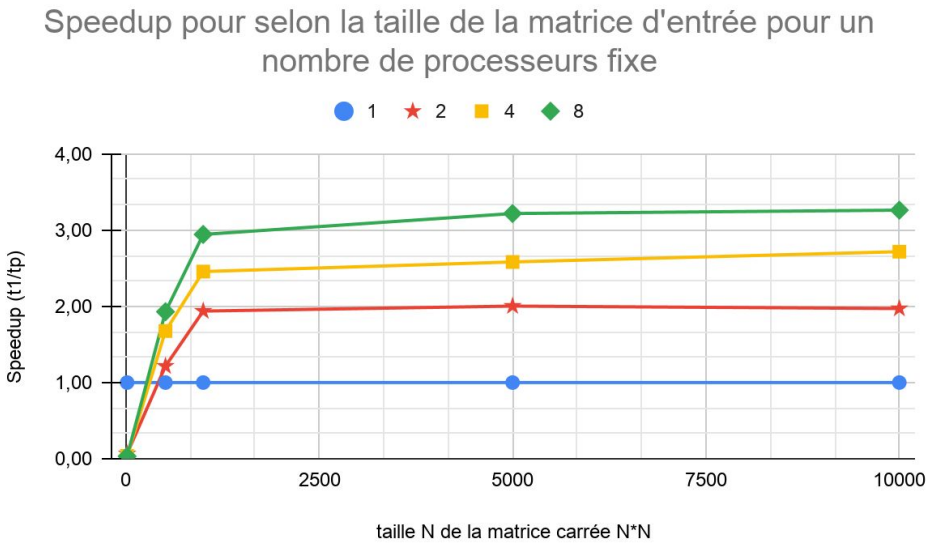
Temps d'exécution				
taille N	1 Proc	2 Proc	4 Proc	8 Proc
16	0,000	0,001	0,001	0,00108
512	0,161	0,132	0,096	0,083295

1000	1,378	0,711	0,561	0,468178
5000	224,182	111,909	86,806	69,663885
10000	2 020,000	1 014,435	736,044	613,058863

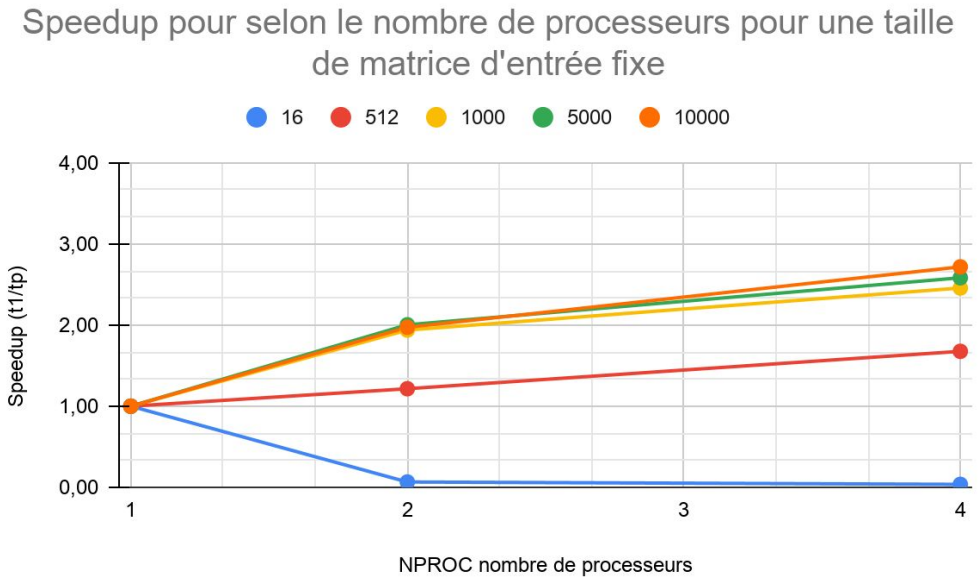


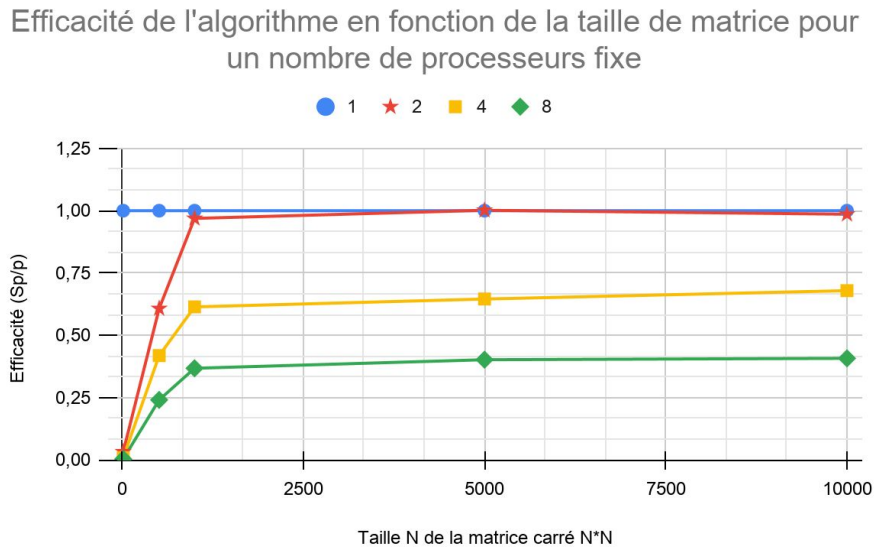


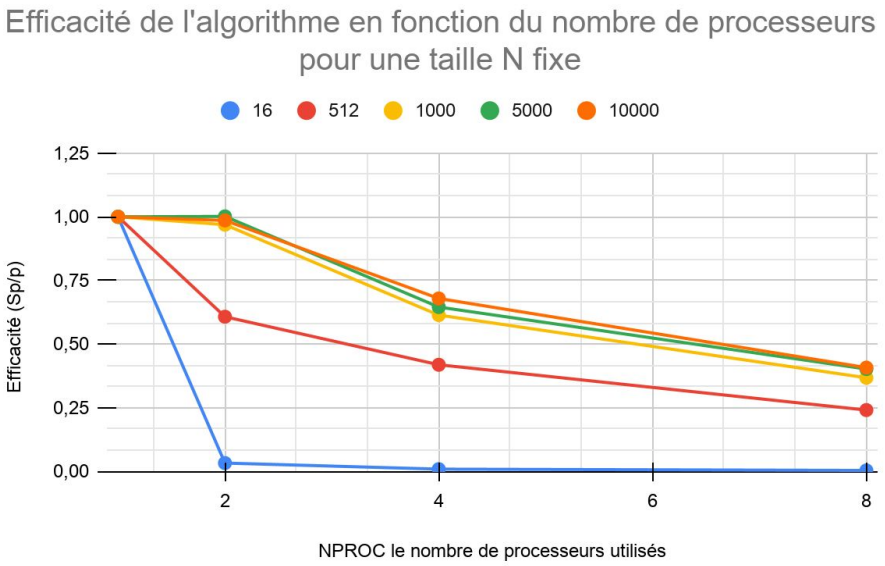
Ces graphes de temps d'exécution (respectivement en échelle logarithmique et linéaire) nous montrent que pour les petites matrices le temps d'exécution est plus court avec peu de processeurs, cela est dû aux coûts de communication.
Lorsque la matrice est plus grande le temps d'exécution devient avantageux avec plus de processeurs, grâce à la scalabilité de l'algorithme.



On remarque le speedup inférieur à 1 pour les petites matrices avec plusieurs processeurs. Dans le cas des petites matrices les coûts de communication rallongent le temps d'éxecution.







La taille de la matrice d’entrée n’a que peu d’influence sur l'efficacité pour un nombre de process fixé.
On remarque que l'efficacité diminue lorsque le nombre de processeurs augmente pour un N d’entré fixé.