Projet 1 : MS01

Clément Baillet et Paul Michel

Voici le problème que l'on cherche à résoudre : soit $\Omega =]0, a[\times]0, b[$, on cherche à calculer le champ $u(x, y)$ gouverné par

$$\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f(x, y), \quad \text{pour } (x, y) \in \Omega,$$

avec les conditions aux limites

$$u(x, 0) = U_0, \quad u(x, b) = U_0, \quad u(a, y) = U_0, \quad u(0, y) = U_0(1 + \alpha V(y)),$$

pour $x \in]0, a[$ et $y \in]0, b[$. On suppose que $V(y) = 1 - \cos(2\pi y/b)$ et $f(x, y) = 0$, et que les paramètres a, b et α sont strictement positifs, avec $\alpha < 1$. Dans toute la suite du rapport, les exemples sont réalisés pour les données suivantes : $U_0 = 1$, $\Omega =]0, 1[\times]0, 1[$, $\alpha = 0,5$. Notre but est d'utiliser les méthodes de Jacobi et de Gauss-Seidel pour résoudre ce problème, puis de paralléliser les codes en utilisant la bibliothèque MPI, afin de les tester sur le mésocentre Cholesky.

1 Méthode de Jacobi

Tout d'abord, on vérifie la convergence de notre code pour $N_x = N_y = 100$ et tol$= 10^{-10}$ (cf Figure 1).

```
clementbaillet@Clements-MacBook-Pro projet1_MS01 % ./buildClem/jacobi 100 100 50000 1e-10
Temps: 8.13254
Iterations: 33130
Error: 9.99769e-11
```

Figure 1: Convergence - méthode de Jacobi

Pour la suite, on fixe une tolérance en précision de 10^{-6} ainsi qu'un nombre maximal d'itérations à 10^4. En effectuant des tests pour N_x (respectivement N_y) variant dans

$$\{20, 50, 100, 200, 400, 800, 1600, 3200, 6400, 12800\},$$

avec N_y (respectivement N_x) fixé à 50, on observe une tendance $Tps \propto N_x^{1,15}$ avec $R^2 = 0.98$ (donc très bonne régression linéaire). On note qu'on trouve $1,15$ plutôt que 1 (complexité temporelle en $\mathcal{O}(N_x N_y)$ car matrice creuse).

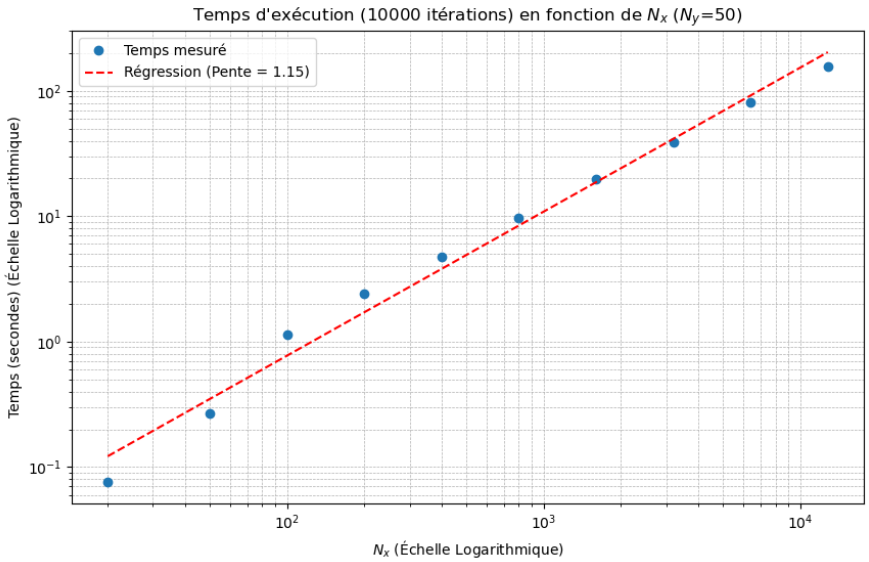

Figure 2: Evolution du nombre d'itérations en fonction de N_x - méthode de Jacobi

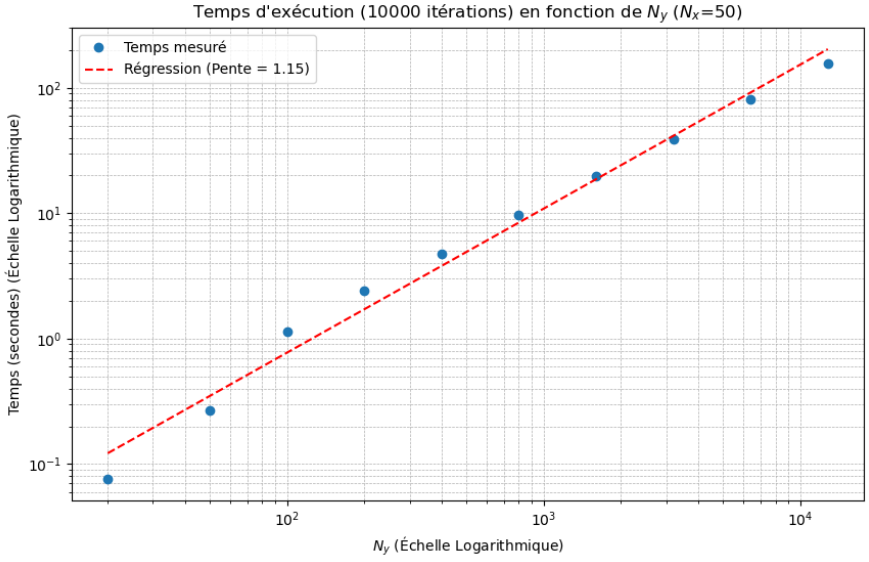

Figure 3: Evolution du nombre d'itérations en fonction de N_y - méthode de Jacobi

2 Méthode de Gauss-Seidel

Comme précédemment, on vérifie la convergence de notre code pour $N_x = N_y = 100$ et tol= 10^{-10} (cf Figure 4).

```
clementbaillet@Clements-MacBook-Pro projet1_MS01 % ./buildClem/gauss-seidel 100 100 50000 1e-10
Temps: 4.10404
Iterations: 17279
Error: 9.9988e-11
```

Figure 4: Convergence - méthode de Gauss-Seidel

De même que pour Jacobi, en effectuant des tests les mêmes tests, on observe une tendance $Tps \propto N_x^{1,28}$ avec $R^2 = 0.98$ (donc également une très bonne régression linéaire), avec de nouveau une pente légerement différente de 1, mais qui reste plausible.

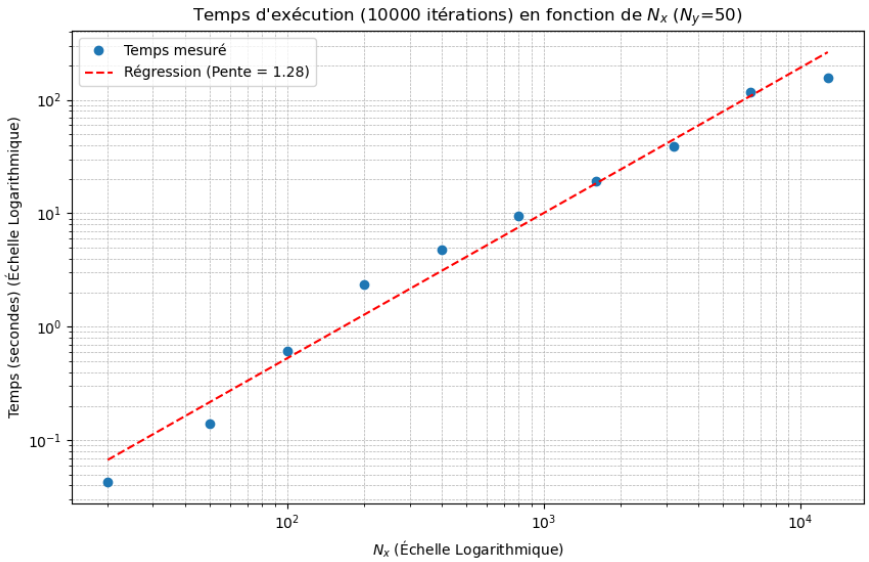

Figure 5: Evolution du nombre d'itérations en fonction de N_x - méthode de Gauss-Seidel

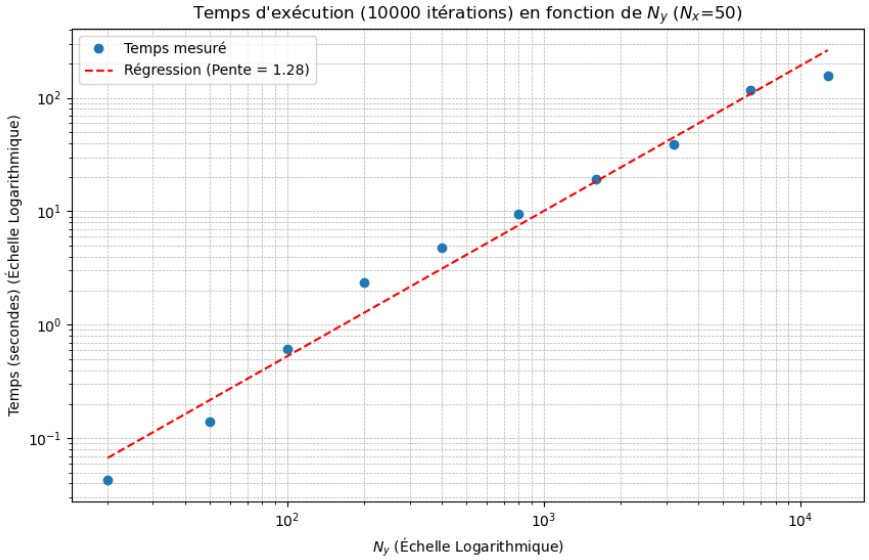

Figure 6: Evolution du nombre d'itérations en fonction de N_y - méthode de Gauss-Seidel

3 Comparaison des deux approches

Dans la méthode de Jacobi, on met à jour u à la fin de chaque boucle, alors qu'on le fait dans la boucle même pour Gauss-Seidel. Cela implique l'utilisation de 2 tableaux représentant u pour Jacobi, alors qu'un seul suffit pour Gauss-Seidel. Cette observation sur le plan logique des méthodes est confirmée par la différence du nombre d'itérations avant convergence : Gauss-Seidel est une

méthode presque deux fois plus rapide que celle de Jacobi en ce sens d'après les Figures 1 et 4. Ceci corrobore la théorie mathématique, mais nous ne la développerons pas dans ce rapport. Plus encore, en comparant le temps d'exécution entre les deux approches, on remarque une fois de plus que celle de Gauss-Seidel est légèrement plus rapide. La comparaison des deux précédentes parties sur l'évolution du nombre d'itérations en fonction de la taille de la grille se visualise sur la figure 7.

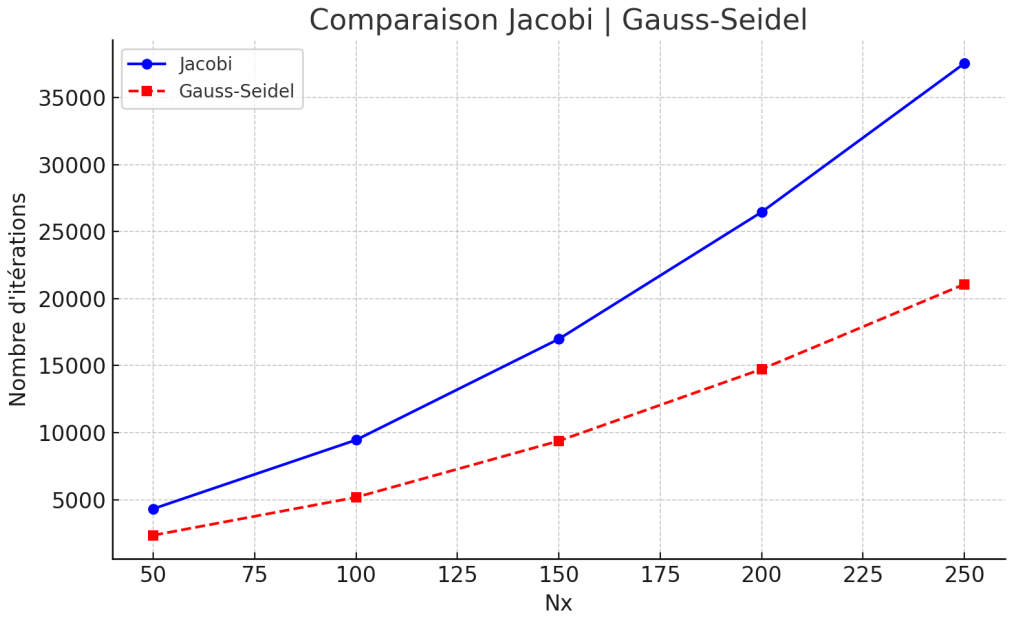

Figure 7: Comparaison de l'évolution du nombre d'itérations en fonction de N_x

4 Parallélisation du code : Jacobi

La parallélisation consiste à découper le domaine en p bandes, où sur chacune d'elle, un processus s'occupe des calculs. Dans cette sous-section uniquement, les tests ont été réalisés sur une grille 500x500 avec 1 jusqu'à 6 processus sur une machine de l'ENSTA. La Figure 8 permet l'étude de la scalabilité forte de la méthode de Jacobi.

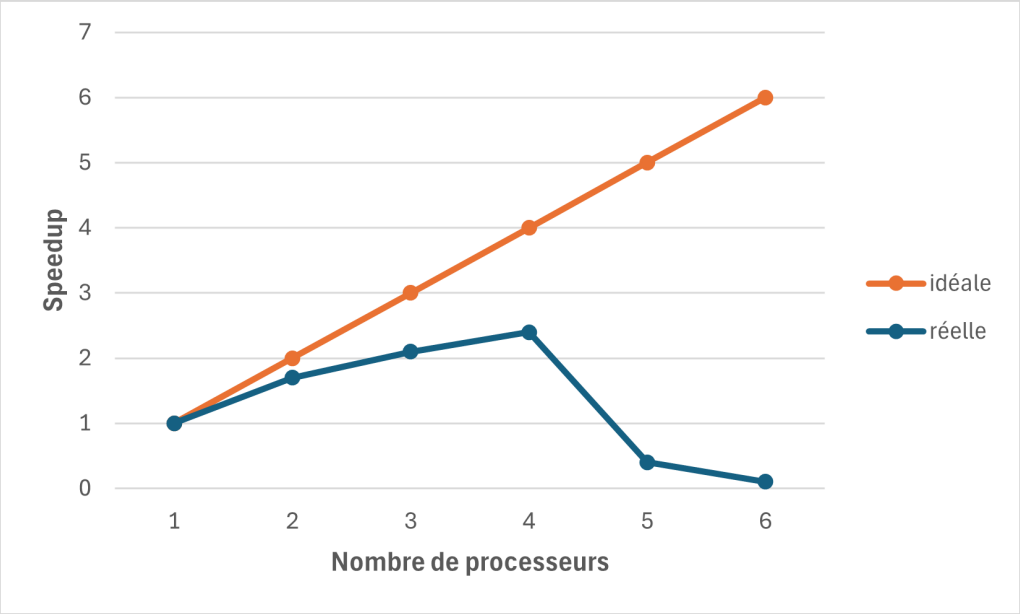

Figure 8: Scalabilité forte - méthode de Jacobi

On observe que le speedup croît avec le nombre de processus tant qu'on se sert de moins de 4 coeurs, limite physique du CPU après laquelle le speedup s'effondre. Sur la partie croissante, on observe une bonne scalabilité forte pour la méthode de Jacobi. On peut constater ces mêmes résultats en étudiant l'évolution de l'efficacité en fonction du nombre de processus sur la Figure 9.

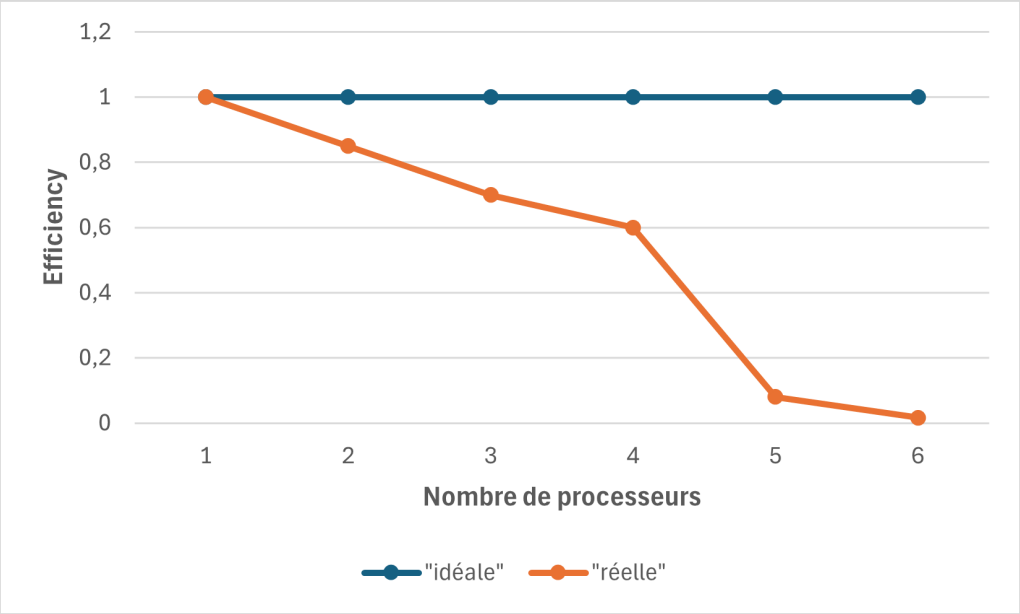

Figure 9: Scalabilité faible - méthode de Jacobi

Il est intéressant de remarquer qu'après avoir effectué des tests sur différentes tailles de grilles, on constate pour un petit format (50x50) un scaling moindre que pour des plus grandes grilles (300x300). Cela semble cohérent car une plus grande taille rend la parallélisation davantage intéressante.

5 Parallélisation du code : Gauss-Seidel

Nous avons parallélisé le code de Gauss-Seidel en utilisant la méthode dites ”Rouge/Noir”, qui consiste, en plus des bandes comme pour Jacobi, à découper le domaine en deux parties indépendantes dans la boucle for (une partie ”Rouge” et une partie ”Noire”). Bien que les différents tests aient montré que généralement Gauss-Seidel convergeait en moins d'itérations que Jacobi (idéalement 2x moins), la scalabilité forte de cette méthode est moins bonne que celle de Jacobi, notamment à cause d'un plus grand nombre de communications avec la méthode de parallélisation employée. Le graphique suivant montre les résultats obtenus pour la méthode de Gauss-Seidel et compare avec la scalabilité forte de Jacobi :

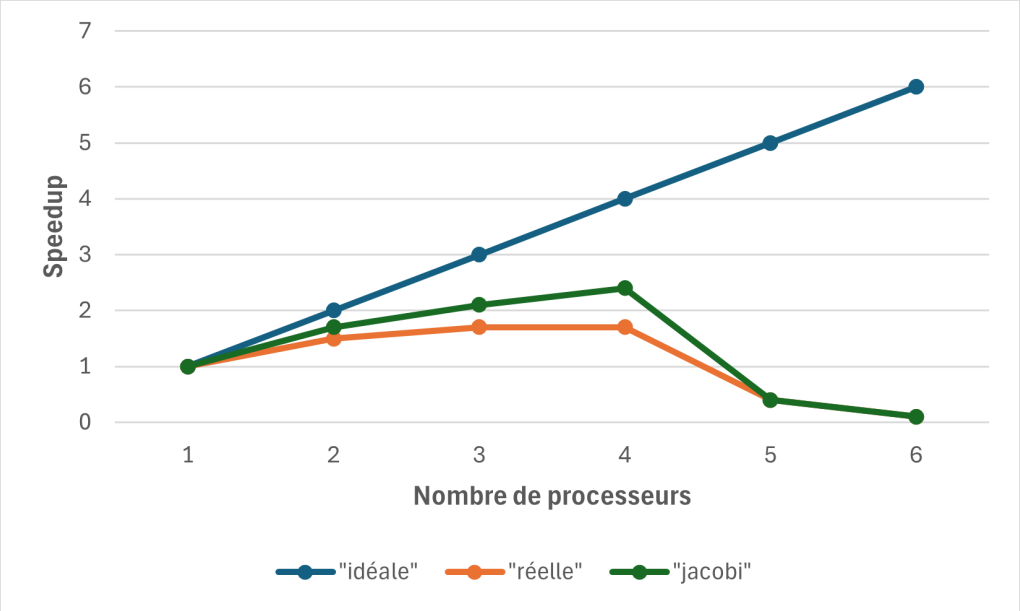

Figure 10: Scalabilité forte - Gauss-Seidel et comparaison

6 Tests avec Cholesky

Pour les tests sur le mésocentre Cholesky, dans un premier temps, nous avons testé la scalabilité forte pour une tolérance de 10^{-8}, et sur un domaine fixe de 256×256. Pour la scalabilité faible, nous avons augmenté progressivement la taille du domaine, en passant de 64×256 à 256×256. Les Figures 11 et 12 montrent les deux scalabilités avec les données obtenues.

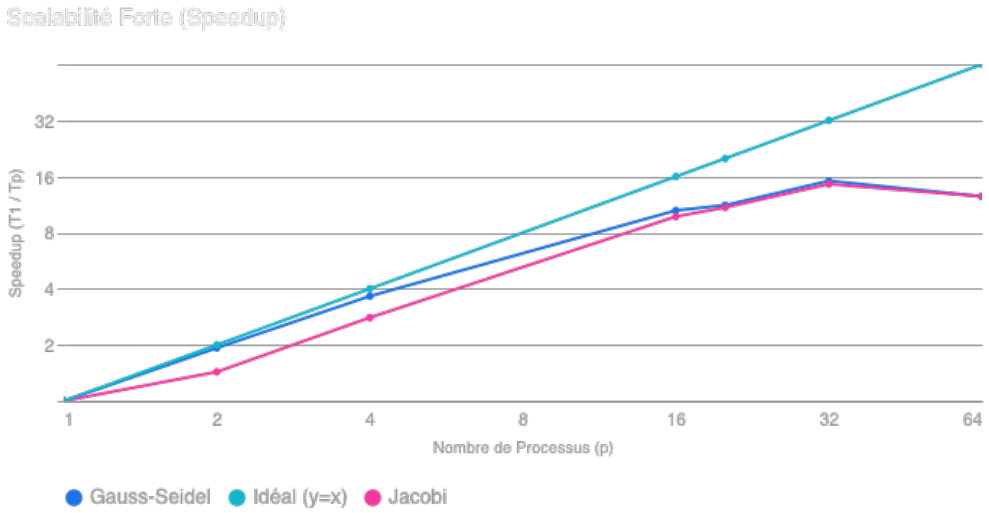

Figure 11: Scalabilité forte - Cholesky

Taille_Domaine	Methode	Métrique	p=1	p=4	p=16	p=20	p=32	p=64
64x256	GS	Speedup	1.0	3.53	8.48	7.59	6.56	4.10
		Efficacité	1.0	0.88	0.53	0.38	0.21	0.06
	Jacobi	Speedup	1.0	3.42	9.31	9.04	7.71	4.33
		Efficacité	1.0	0.86	0.58	0.45	0.24	0.07
128x256	GS	Speedup	1.0	3.56	10.47	10.89	10.98	7.88
		Efficacité	1.0	0.89	0.65	0.54	0.34	0.12
	Jacobi	Speedup	1.0	3.40	9.87	10.77	9.84	7.23
		Efficacité	1.0	0.85	0.62	0.54	0.31	0.11
256x256	GS	Speedup	1.0	3.87	13.07	14.33	16.73	14.36
		Efficacité	1.0	0.97	0.82	0.72	0.52	0.22
	Jacobi	Speedup	1.0	3.68	12.28	14.30	17.59	14.24
		Efficacité	1.0	0.92	0.77	0.72	0.55	0.22

Figure 12: Speedup et Efficacité sur petits domaines - Cholesky

Pour le speedup, on observe bien une augmentation quasi linéaire en fonction du nombre de processus (mais qui ne l'est pas parfaitement à cause du temps pris par les communications MPI). On note le déclin autour de 20 processus, lié au fait que chaque noeud utilisé sur Cholesky possède 20 coeurs. L'efficacité quant à elle, reste relativement proche de 1 jusqu'à 20 processus, sans l'être parfaitement à cause des communications, après quoi elle dégringole.

Dans un second temps, nous avons voulu mesurer la scalabilité faible, mais avec des domaines beaucoup plus grands: nous sommes parti d'un domaine 64×256, et sommes monté progressivement jusqu'à 4096×256. Malheureusement, les résultats étaient au départ très peu concluants en raison du nombre d'itérations qui explose, et qui rend l'efficacité caduque (cf Figure 13). Nous avons donc refait une poignée de tests, en fixant cette fois-ci le nombre d'itérations maximal à 50000. La méthode ne converge pas (et est même parfois assez loin de la tolérance fixée), mais l'efficacité est beaucoup plus proche de 1 (avec une décroissance attendue lorsque qu'on augmente le nombre de processus, et une chute notable passé les 20 processus, cf Figure 14).

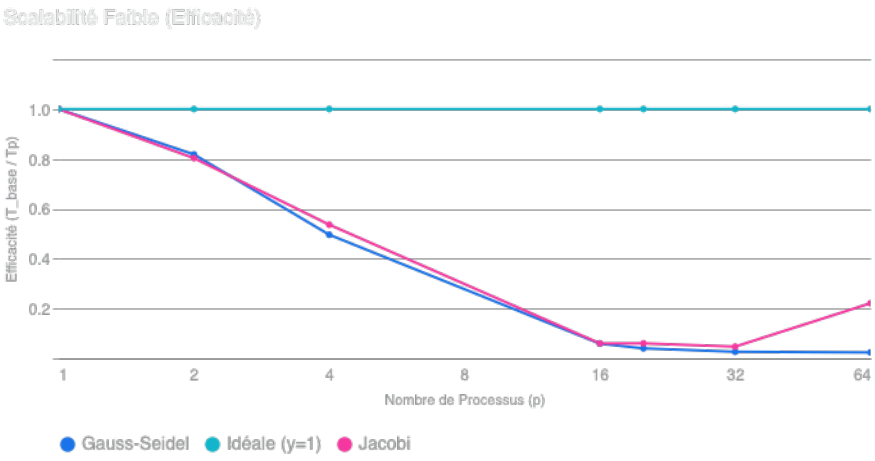

Figure 13: Scalabilité faible sur des grands domaines - Cholesky

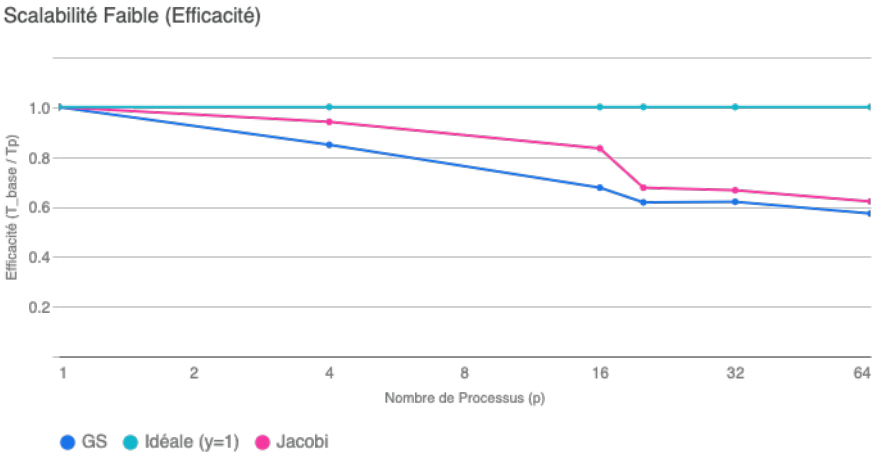

Figure 14: Scalabilité faible sur des grands domaines à itérations fixe - Cholesky

Conclusion

Ce rapport compare les méthodes de Jacobi et Gauss-Seidel (GS) pour résoudre l'équation de Laplace, en séquentiel puis en parallèle avec MPI sur le cluster Cholesky.

En séquentiel, bien que Gauss-Seidel converge en deux fois moins d'itérations, l'étude de la complexité montre que Jacobi (pente 1.15) a une meilleure scalabilité que GS (pente 1.28). Ceci pourrait être dû aux opérations effectuées dans chacune des boucles, avec dans la méthode de Jacobi l'utilisation de l'itération précédente déjà entièrement calculée, et dans celle de Gauss-Seidel l'utilisation de valeurs qui viennent d'être mises à jour.

En parallèle, Jacobi (découpage en bandes) montre une meilleure scalabilité forte que GS (méthode Rouge/Noir), puisque recquiert moins d'opérations de communications. Sur Cholesky, le speedup est quasi-linéaire jusqu'à 20 cœurs (limite du nœud), mais la scalabilité faible n'est validée qu'en fixant le nombre d'itérations (et donc en compromettant la convergence).

Des potentielles pistes à suivre pour améliorer la scalabilité de ces codes, et notamment celle de Gauss-Seidel, seraient d'utiliser d'autres schémas de parallélisation, comme par exemple un schéma 2D voire 3D.