TDP 4 - Factorisation LU et descente-remontée

Etienne THIERY - Youssef SEDRATI

8 janvier 2017

Dans ce projet nous développons une implémentation parallèle distribuée de la factorisation LU.

1 La factorisation LU

Étant donné une matrice générale A de taille $M \times N$. Nous souhaitons résoudre en parallèle le système suivant : $Ax = b$. Pour ce faire, nous avons mis notre matrice sous la forme suivante : $A = LU$.

Cette dernière est décomposée en :
- une matrice triangulaire inférieure L de taille $M \times min(M, N)$ dont les coefficients diagonaux valent 1.
- une matrice triangulaire supérieure U.

Dans la suite de ce rapport :

- `dgetrf2_nopiv` désigne une implémentation parallèle de la factorisation LU sans pivotage, utilisant les BLAS ≤ 2 `dscal` (multiplication scalaire vecteur) et `ger` (multiplication matrice vecteur), qui utilise les coeurs d'une même machine mais peu efficacement.

- `dgetrf_nopiv` désigne une implémentation parallèle en blocs de la factorisation LU sans pivotage utilisant `dgetrf2_nopiv` et les BLAS 3 `dtrsm` (résolution d'un système linéaire triangulaire) et `dgemm` (produit de matrices), qui utilise les coeurs d'une même machine beaucoup plus efficacement.

- `pdgetrf_nopiv` désigne une implémentation parallèle distribuée de la factorisation LU sans pivotage s'appuyant sur `dgetrf_nopiv` localement, qui utilisent les coeurs de plusieurs machines communiquant via MPI.

- `pdgetrf` désigne une implémentation parallèle distribuée de la factorisation LU avec pivotage, non seulement efficace comme `pdgetrf_nopiv` mais aussi plus stable numériquement. Nous n'avons malheureusement pas eu le temps de développer celle-ci.
- `dgetrs2_nopiv` désigne une implémentation parallèle de la résolution d'un système linéaire précédemment factorisé LU, utilisant le BLAS 1 `ddot`.

2 Utilisation du projet

2.1 Compilation

La compilation du projet nécessite la présence d'Intel MKL et d'une implémentation d'MPI sur la machine (modules `intel/mkl/64/11.2/2016.0.0`, `intel-tbb-oss/intel64/43_20150424oss` et `mpi/openmpi/gcc/1.10.0-tm` sur PlaFRIM).

Les deux éxécutables du projet peuvent être compilés via la commande :

```
make
```

2.2 Exécution

`correctionAndStability` ne prend aucun argument et teste la correction et la stabilité numérique de nos implémentations `dgetrf2_nopiv`, `dgetrf_nopiv` et `pdgetrf_nopiv`. Voir partie 4.1 pour plus de détails.

`performance` effectue la factorisation LU d'une matrice aléatoire, et affiche le temps nécessaire à ce calcul. Il prend en argument :

- Le nom de l'implémentation à utiliser parmi `dgetrf_MKL` (implémentation MKL, pour comparaison), `dgetrf2_nopiv`, `dgetrf_nopiv` et `pdgetrf_nopiv`.
- Les deux dimensions de la matrice M et N.
- Les valeurs minimales et maximales des coefficients de la matrice initiale.
- Le seed à utiliser pour la génération de la matrice initiale.
- La largeur en nombre d'éléments des blocs colonne distribués aux différents noeuds pour `pdgetrf_nopiv`. Celle-ci doit être un diviseur de N divisé par le nombre de noeuds.
- La largeur = hauteur des blocs utilisés par `dgetrf_nopiv`. Celle-ci doit également diviser N par le nombre de noeuds.

Exemple pour la factorisation LU d'une matrice de 20000×10000 coefficients dans $[-5, 5]$ générés à partir d'un seed 42, utilisant des blocs colonne de largeur 250 et des blocs fins de taille 50, distribuée sur 4 noeuds de 20 coeurs chacun :

```
MKL_NUM_THREADS=20 mpiexec -np 4 --bind-to none ./performance pdgetrf_nopi2 10000 10000
-5 5 42 250 50
```

2.3 Reproduction des expériences

Les études dont les résultats sont présentés en section 4 peuvent être reproduites à l'aide script SLURM, Bash et gnuplot.

Étude 1 :

```
sbatch Jobs/correctionAndStabilityJob.sh
```

Étude 2 :

```
sbatch funWithLapack/Jobs/tuningJob.sh
```

puis une fois ce job fini

```
gnuplot Plots/fineBlockSizeTuning.p
```

Étude 3 :

```
sbatch funWithLapack/Jobs/job1.sh
sbatch funWithLapack/Jobs/job3.sh
sbatch funWithLapack/Jobs/job6.sh
sbatch funWithLapack/Jobs/job9.sh
```

puis, une fois ces 4 jobs finis

```
bash Jobs/aggregate.sh
gnuplot Plots/pdgetrfScaling.p
```

3 Architecture du projet

Les fichiers `util.c|h` contiennent quelques fonctions utilitaires pour parser les arguments d'une ligne de commande, initialiser des matrices et systèmes linéaires de test / aléatoires, et calculer les erreurs commises sur les solutions.

Les fichiers `distributedData.c|h` définissent une structure de matrice distribuée par blocs colonne en "serpentin" (exemple : 1 2 3 3 2 1 1 2 3), et les méthodes de scatter et gather correspondantes.

Les fichiers `LU.c|h` contiennent les 3 implémentations de la factorisation `dgetrf2_nopiv`, `dgetrf_nopiv` et `pdgetrf_nopiv`, et une implémentation de `dgetrs2_nopiv`. Les fichiers `genericWrapper.c|h` définissent un wrapper commun aux 3 implémentations de la factorisation LU pour pouvoir les utiliser de façon interchangeable.

Enfin, les fichiers `correctionAndStability.c` et `performance.c` définissent les fonctions principales des deux exécutables correspondants.

4 Résultats

4.1 Expérience 1 : Correction et stabilité numérique

Les différentes implémentations de la factorisation sont de complexité croissante et s'utilisent les unes les autres. Pendant le développement de celles-ci, nous avons tout d'abord vérifié qu'elles étaient correctes avec une métrique très simple : la valeur absolue maximale des coefficients de la différence $A - LU$ (première colonne des résultats plus bas).

Nous avons appris dans un autre cours que cette métrique n'est pas la plus pertinente, mais en comparant les résultats avec ceux obtenus pour l'implémentation de la MKL, nous avons pu nous assurer que nos implémentations étaient globalement correctes.

Pour évaluer la précision de nos implémentations, nous avons utilisé deux métriques plus appropriées définies comme suit.

L'erreur inverse en norme sur les membres gauche et droit :
$\eta^N_{A,b}(x) = \min\{\epsilon : (A + \Delta A)x = b + \Delta b,\ ||\Delta A|| \leq \epsilon||A||, ||\Delta b|| \leq \epsilon||b||\}$
Qui peut être calculée comme : $\eta^N_{A,b}(x) = \frac{||Ax-b||}{||A||||x||+||b||}$

Et l'erreur inverse sur le second membre uniquement :
$\eta^N_b(x) = \min\{\epsilon : Ax = b + \Delta b,\ ||\Delta b|| \leq \epsilon||b||\}$
Qui peut être calculée comme :$\eta^N_b(x) = \frac{Ax-b}{b}$

Ces deux métriques représentent en quelque sorte une distance entre le système linéaire à résoudre, et le système linéaire dont la solution obtenue est la solution exacte. La première est la plus importante, et est à comparer avec l'epsilon machine, de l'ordre de 10^{-16} en double précision.

Les résultats obtenus montrent qu'on n'a aucune perte de précision avec l'implémentation de MKL, mais environ 3 décimales de perdus avec nos implémentations. C'est sans aucune doute lié à l'absence de pivotage, qui rend l'algorithme beaucoup moins stable numériquement. Nous n'avons cependant pas eu le temps d'implémenter un pivotage qui fonctionne.

	$\max(\|A - LU\|)$	Erreur inverse	Erreur inverse sur second membre seulement
`MKL`	2.486900e-13	3.254802e-16	7.143778e-12
`dgetrf2_nopiv`	3.410605e-11	1.156529e-13	3.084873e-09
`dgetrf_nopiv`	2.984279e-11	1.209783e-13	3.226921e-09
`pdgetrf2_nopiv`	5.303491e-11	1.156992e-13	3.086108e-09

FIGURE 1 – Factorisation LU d'une matrice aléatoire avec coefficients dans $[-5, 5]$ de taille 2000×1000 (seed = 42, blocs columnes de largeur 50, blocs fins de taille 10)

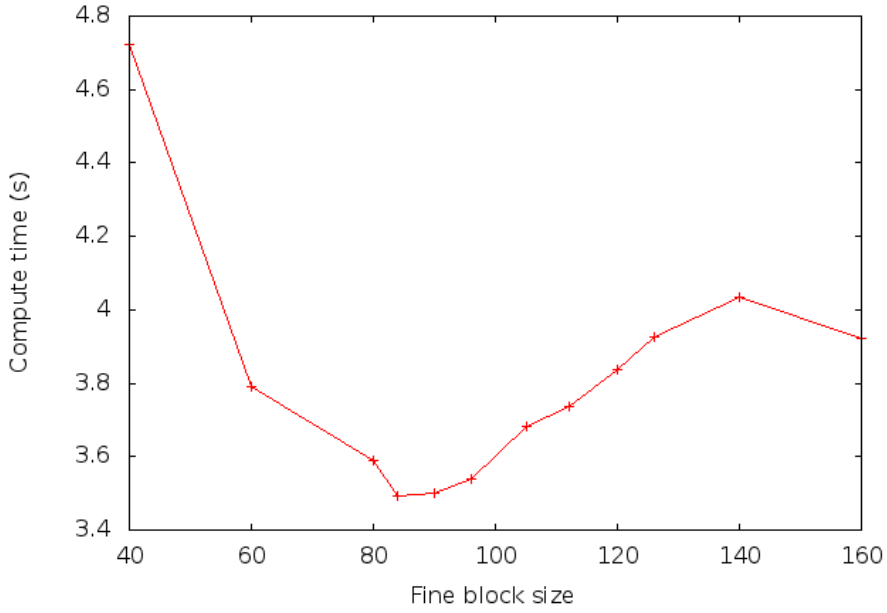
4.2 Expérience 2 : tuning des tailles de blocs

Une fois assuré que nos implémentations étaient correctes et avant d'effectuer une étude de scalabilité, nous avons voulu déterminer les paramètres optimaux pour les tailles de blocs utilisés.

La première taille est la taille des blocs utilisés dans `dgetrf_nopiv` pour améliorer la localité spatiale des calculs. Nous avons tracer le temps pris par `dgetrf_nopiv` pour effectuer la factorisation d'une matrice 10080×10080 en utilisant les 20 coeurs d'une machine, selon la taille des blocs. Pour la suite, nous avons utilisé une taille d'environ 85, qui semble optimale.

Noter qu'avec cette taille, on atteint : $2 * \frac{2}{3} * 10080^3/3.5 = 390 Gflop/s$.
Soit quasiment la capacité crête d'un noeud (20 coeurs, avx2, 2.5 Ghz) : $20 * 8 * 2.5 \times 10^9 = 400$

FIGURE 2 – Temps de calcul d'une factorisation LU bloquée selon la taille des blocs

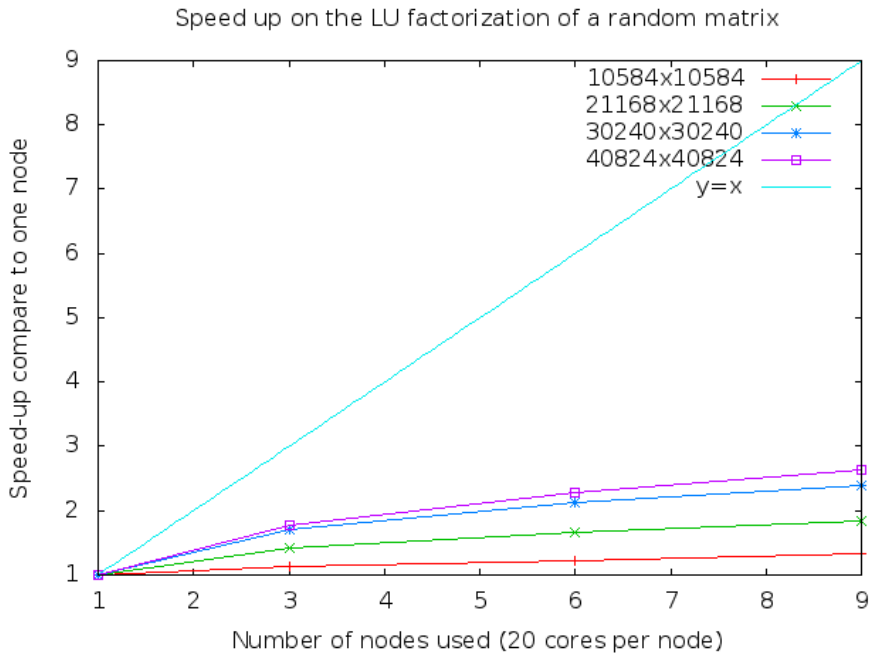


Nous avons ensuite mené la même expérience pour la largeur des blocs colonnes distribués entre les noeuds dans `pdgetrf_nopiv`, avant de nous rendre compte que la taille optimale était nécessairement la même que la taille précédente. En effet, les blocs colonnes d'un même noeud sont stockées de façon contigue de façon à pouvoir appliquer les blas d'un seul coup, donc avoir des blocs colonnes plus larges n'a aucun avantage. En revanche, plus les blocs colonnes sont larges, moins la distribution en serpentin est homogène, et donc moins les performances sont bonnes. Ce paramètre n'est donc pas très utile, et en pratique nous le fixons toujours à la même valeur que le premier.

4.3 Expérience 3 : scalabilité

Enfin, nous avons mesuré le speedup obtenu avec `pdgetrf_nopiv` sur un nombre de noeuds variant de 1 à 9, et ce pour des tailles de matrices différentes. Les temps mesurés n'incluent pas les temps de scatter/gather de la distribution en serpentin, important pour des matrices de grande taille.

FIGURE 3 – Speedup de notre implémentation distribuée, avec différentes taille de matrices



On constate que la scalabilité de notre implémentation est assez mauvaise, et que même si elle s'améliore lorsqu'on travaille sur des matrices plus grandes, cet effet se "tasse" rapidement (écart faible entre les 2 meilleures courbes). On dépasse à peine le téraflop/s, alors que la capacité crête du cluster utilisé et de plus de 3 téraflop/s.

Nous estimons que cette mauvaise scalabilité est du notamment au fait qu'à chaque étape de la factorisation, l'ensemble des noeuds doit attendre qu'un noeud broadcast ces résultats pour pouvoir effectuer leurs calculs. Nous n'avons pas eu le temps de régler ce problème.