Systèmes distribués et grilles : TP1

Thibaut EHLINGER, Benjamin HERB *Université de Strasbourg*

19 septembre 2016

Le but de ce TP est de mettre en évidence l'impact de l'utilisation de *clusters* de PC dans l'exécution d'un algorithme très gourmand en calcul, ici un produit de matrices. Dans un premier temps, nous présenterons rapidement notre résolution du problème pour le code manquant dans le projet. Puis nous présenterons les résultats des différents essais ; résultats que nous analyserons ensuite afin d'essayer de trouver quel est le choix le plus judicieux pour exécuter un produit de matrices de 4096×4096 lignes.

Résultats

Conditions d'expérimentation

Les mesures de l'expérimentation n'ont été effectuées qu'une seule fois. Il est donc claire que ces résultats sont qualitatifs et non quantitatifs. Ce rapport sert surtout à mettre en avant un raisonnement. Les conclusions sont à prendre avec précautions et à revérifier expérimentalement sur plus d'essais.
Nous nous sommes basés sur le temps d'exécution de la boucle pour les mesures de temps.

Kernel 1 contre kernel 0

Bien que cela ne présente qu'un aspect mineur des performances de la machine que nous devions évaluer, il est tout de même intéressant de comparer les différences de performances entre les deux kernels. Cette différence est mise en exergue figure 1, où l'on constate que le kernel 0 est bien plus lent que le kernel 1. Bien que l'échelle logarithmique estompe visuellement cette différence, elle permet de mettre en valeur la régularité de cette écart. Ainsi sur deux nœuds par exemple, on constate que le temps d'exécution sur le kernel 1 et 4 fois plus rapide environ que sur le kernel 0. A noter que le kernel 0 met même plus de **deux minutes** avec un seul nœud à faire le calcul.

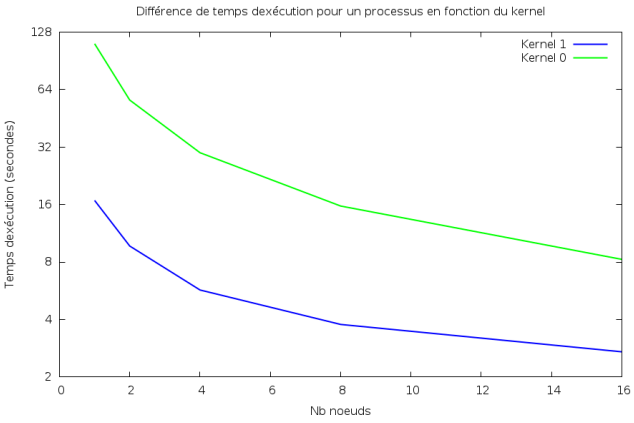

Figure 1 – *Comparaison du temps d'exécution avec un seul processus entre les kernels 0 et 1.*

Round robin contre une répartition voisine des tâches

Avec deux instance *MPI* par nœud, il est très important de prêter attention à la répartition des instances MPI. En d'autres mots, on constate que la manière de répartir les instances sur les nœuds par paire influe grandement sur les résultats.
Ici, nous avons choisi de nous intéresser à deux répartitions précises : *round robin* contre une répartition de proche en proche, aussi appelée "*par voisins*".
En effet, dans l'algorithme, chaque instance MPI calcule une tranche de la matrice résultat à partir d'une tranche de la matrice $\mathcal{A}$, puis il transfère cette tranche de $\mathcal{A}$ à son voisin. C'est la localisation de ce voisin qui différencie les deux modes de répartition des instances MPI : en mode "*par voisins*", on met les voisins sur un même nœud, tandis qu'en *round robin*, on répartit les instances tour à tour sur les nœuds et c'est seulement après avoir attribué une instance par nœud qu'on recommence. Ainsi, ce sont à chaque fois des instances avec des identifiants très éloignées qui se retrouvent sur un même nœud, ce qui fait que la circulation des tranches de $\mathcal{A}$ nécessite

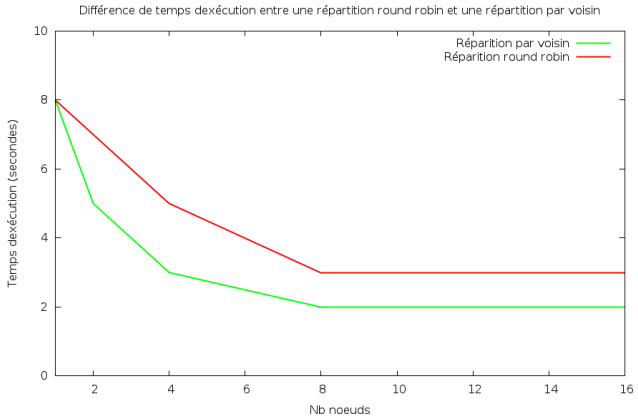

Figure 2 – *Comparaison du temps d'exécution en round robin contre voisins pour 1 et 2 threads. Les performances sont évidemment les mêmes pour un seul nœud.*

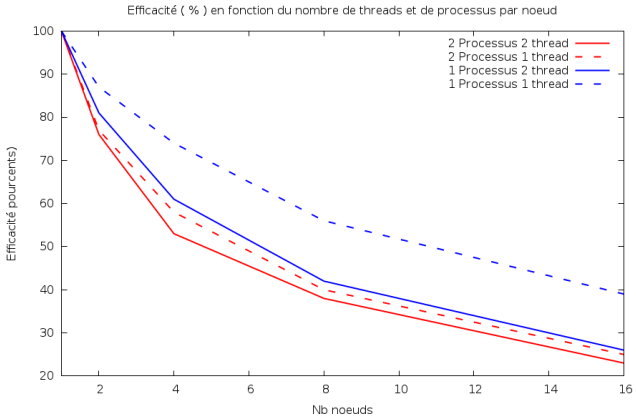

Figure 3 – *Comparaison du temps de l'efficacité, en pourcents, en fonction du nombre de threads/processus par nœud.*

bien plus de temps de communication.
La figure 2 met la différence de temps d'exécution en exergue. On constate que la répartition en *round robin* est plus lente que la répartition en voisins quelque soit le nombre de noeuds, sauf évidemment pour **un noeud**, où le mode de répartition donne évidemment le même résultat.

Comparaison de l'efficacité en fonction des différents paramètres

Pour cette problématique nous n'allons nous intéresser, pour les paires de processus par nœud, qu'au mode de répartition par voisins, celui-ci étant meilleur toutes choses égales par ailleurs au mode de répartition *round-robin*.
Est-il *interéssant* d'utiliser deux threads par processus et deux processus par noeud au lieu de juste un thread par noeud et un noeud par processus ?
De plus, n'ayant effectué les mesures qu'une seule fois, il est claire que ces données ne peuvent en aucun cas être représentatives. Néanmoins la démarche employée serait la bonne pour un plus grand jeu de données. Sur la figure 4 on voit très clairement que le mode de calcul 1 processus/1 thread est beaucoup plus lent que les autres, en revanche départager les autres n'est pas évident car les courbes de la figure 3 témoignent d'une efficacité très similaire en fonction du nombre de nœuds. Comme le temps de calcul ne permet pas de repérer de très grande différence, nous nous baserons sur la puissance de calcul. La nature même de l'unité (le gigaflop) donne lieu à de plus grands écarts mis en évidence figure 5. Sur celle-ci, on constate que la puissance est globalement similaire pour 1 processus/2 threads et 2 processus/2threads.

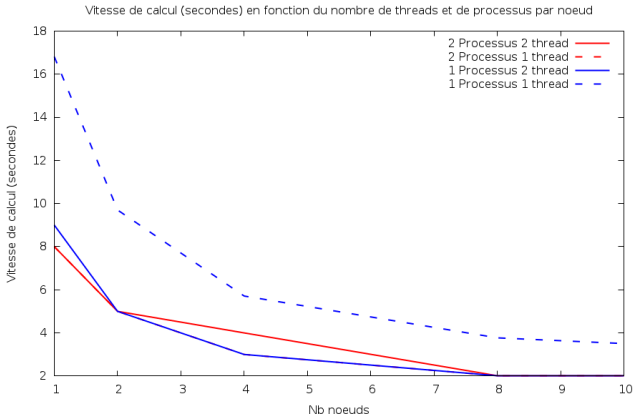

Figure 4 – *Comparaison du temps de calcul en fonction du nombre de threads/processus par nœud.(On s'arrête à 8 nœuds car difficile de voir une différence au-delà)*

En revanche il y a une grande "chute" de puissance chez 2 processus/1 thread à 8 nœuds plus. Il faudrait reproduire plusieurs fois l'expérience pour savoir s'il s'agit d'une anomalie.

Tendances déduites des résultats

Pour résumer, on peut conclure qu'il assez certain :
- que le kernel 0 est bien moins efficace que le kernel 1 ;
- que la répartition en round robin est bien moins efficace que la répartitions par voisins.

Les premiers résultats, qu'il faudrait compléter avec plus de données, laissent à penser, que :
- le mode 1 thread/1 processus est clairement moins bon que les autres. ;
- il est presque toujours rentable d'utiliser 2 threads au lieu de 1 thread ;
- il difficile de trouver le meilleur compromis

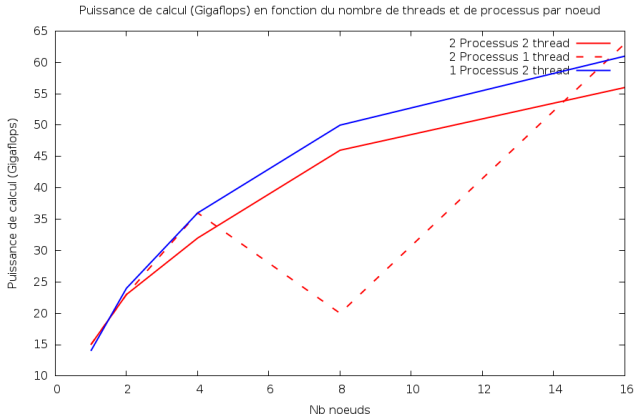

Figure 5 – *Comparaison de la puissance (gigaflops) en fonction du nombre de threads/processus par nœud.*

entre 2 threads/2 processus, 2 threads/ 1 processus et 2 processus / 1 thread. Dans ce contexte, j'aurai tendance à prendre le mode de calcul le **moins gourmand**, ou en tout cas à éliminer le plus gourmand, c'est à dire 2 threads 2 processus.

Résolution du problème

Dans un premier temps, nous devions compléter trois parties spécifiques du code, qui concernaient l'initialisation MPI, le calcul de l'*offset* de départ dans la matrice pour chacun de processus en fonction de leur identifiant MPI, et enfin les différentes étapes de la boucle principale, c'est à dire l'échange des données et le calcul des sous-parties de la matrice résultat par chaque processus. Cette partie se décompose en deux sous-fonctions : `ComputationAndCirculation` et `OneStepCirculation`.

Initialisation MPI

```
#include <stdio.h>
#define N 10

/* Initialisation of processor coordinates
    */

void ProcessorInit(void){
  MPI_Comm_size(MPI_COMM_WORLD,&NbPE);
    MPI_Comm_rank(MPI_COMM_WORLD,&Me);
}
```

Calcul de l'offset pour un processus `Me` donné

```
/* Compute the current step offset, in the
    MPI program, to access right C lines
   */ OffsetStepLigC = ((Me + step) *
   LOCAL_SIZE) % SIZE;
```

Boucle principale

```
void ComputationAndCirculation()
{
 unsigned long step;

 for(step=0;step<NbPE;step++) {
  OneLocalProduct(step);
  OneStepCirculation(step);
 }
}
/* Elementary circulation of A and B.

    */
void OneStepCirculation(unsigned long step
    )
{
 MPI_Status   status;

 MPI_Sendrecv_replace(A_Slice, SIZE*
    LOCAL_SIZE, MPI_DOUBLE, (Me-1+NbPE)%
    NbPE, 0, (Me+1)%NbPE, 0,
    MPI_COMM_WORLD, &status);
}
```