Programmation des Architectures Parallèles

Rapport Mini-Projet

DIALLO Abdoul & DIETRICH Brian

4 février 2019

Table des matières

1 Négatif d'une Image :

Pour obtenir le négatif d'une image nous avons créé la fonction **invert_compute_omp(unsigned nb_iter)**. On peut voir que chaque thread calcule une bande horizontale de l'image à l'aide de la ligne **printf()** passée en commentaire.

```
unsigned invert_compute_omp (unsigned nb_iter)
{

  for (unsigned it = 1; it <= nb_iter; it++) {
    #pragma omp parallel for
    for (int i = 0; i < DIM; i++){
      //printf("je suis le thread %d\n",omp_get_thread_num());
      for (int j = 0; j < DIM; j++)
      cur_img (i, j) = invert_pixel (cur_img (i, j));
    }
  }
  return 0;
}
```

```
adiallo004@monet:~/Bureau/OPENMP/mini-projet/fichiers$ ./2Dco
a.png -k invert -v omp -n -i 1000
Using kernel [invert], variant [omp]
Arrêt après 1000 itérations
339.414
adiallo004@monet:~/Bureau/OPENMP/mini-projet/fichiers$ ./2Dco
a.png -k invert -v seq -n -i 1000
Using kernel [invert], variant [seq]
Arrêt après 1000 itérations
1718.184
```

On observe que le rapport (temps séquentiel) / (temps parallèle) est environ égal à 5.0622. On en déduit donc que la version parallèle est beaucoup plus rapide(environ 5 fois plus rapide).

2 Décalage vers le haut

Pour procéder au décalage d'une image vers le haut, nous avons le code suivant :

```
unsigned scrollup_compute_omp (unsigned nb_iter)
{
  #pragma omp parallel for
  for (unsigned it = 1; it <= nb_iter; it++) {

    for (int j = 0; j < DIM; j++)
      next_img (DIM - 1, j) = cur_img (0, j);

    for (int i = 0; i < DIM - 1; i++)
      for (int j = 0; j < DIM; j++)
        next_img (i, j) = cur_img (i + 1, j);

    //printf("je suis le thread %d et l'itération %d\n",omp_get_thread_num(),it)
    swap_images ();
  }

  return 0;
}
```

Nous observons à l'aide du printf() que la fonction swap_images() s'exécute une seule fois par itération.

3 Mandelbrot

Nous avons réalisé trois parallélisations de la fonction **mandel_compute_seq** en utilisant diverses politiques de distribution.

Dans un premier temps nous avons effectué une parallélisation de nos fonctions en ajoutant une directive **pragma omp parallel for schedule(x)** au niveau de la boucle for, cependant nous avons observé que les performances n'étaient pas impactées par cet ajout de code.

```
unsigned mandel_compute_ompb (unsigned nb_iter)
{
  #pragma omp parallel for schedule(static)
   for (unsigned it = 1; it <= nb_iter; it++) {

    // On traite toute l'image en une seule fois
    traiter_tuile (0, 0, DIM - 1, DIM - 1);

    zoom ();
  }

  return 0;
}
```

```
unsigned mandel_compute_ompb (unsigned nb_iter)
{
  #pragma omp parallel for schedule(static,3)
   for (unsigned it = 1; it <= nb_iter; it++) {

    // On traite toute l'image en une seule fois
    traiter_tuile (0, 0, DIM - 1, DIM - 1);

    zoom ();
  }

  return 0;
}
```

```
unsigned mandel_compute_ompb (unsigned nb_iter)
{
  #pragma omp parallel for schedule(dynamic,3)
   for (unsigned it = 1; it <= nb_iter; it++) {

    // On traite toute l'image en une seule fois
    traiter_tuile (0, 0, DIM - 1, DIM - 1);

    zoom ();
  }

  return 0;
}
```

Nous avons donc décidé de remplacer l'appel de la fonction **static void traiter_tuile (int i_d, int j_d, int i_f, int j_f)** par son contenu parallélisé.

Dans notre premier cas nous avons utilisé une politique de distribution statique.

```
unsigned mandel_compute_ompb (unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++) {

    // On traite toute l'image en une seule fois
    //traiter_tuile (0, 0, DIM - 1, DIM - 1);

  #pragma omp parallel for schedule(static)
    for (int i = 0; i <= DIM-1; i++)
    for (int j = 0; j <= DIM-1; j++)
      cur_img (i, j) = iteration_to_color (compute_one_pixel (i, j));

    zoom ();
  }

  return 0;
}
```

Dans le deuxième,nous avons utilisé une politique de distribution cyclique.

```
unsigned mandel_compute_ompc_k (unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++) {
    // On traite toute l'image en une seule fois
    #pragma omp parallel for schedule(static,3)
    for (int i = 0; i <= DIM-1; i++)
    for (int j = 0; j <= DIM-1; j++)
      cur_img (i, j) = iteration_to_color (compute_one_pixel (i, j));
    zoom ();
  }
  return 0;
}
```

Nous avons observé que l'optimisation était plus efficace lorsque l'on utilise un paramètre k=3 lors de la parallélisation. Nous avons obtenu de meilleurs résultats en augmentant le paramètre k jusqu'à la valeur 3, au delà les performances chutent.

Dans notre troisième cas nous avons utilisé une politique de distribution dynamique.

```
unsigned mandel_compute_ompd_k (unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++) {
    #pragma omp parallel for schedule(dynamic,1)
    for (int i = 0; i <= DIM-1; i++)
    for (int j = 0; j <= DIM-1; j++)
      cur_img (i, j) = iteration_to_color (compute_one_pixel (i, j));
    zoom ();
  }
  return 0;
}
```

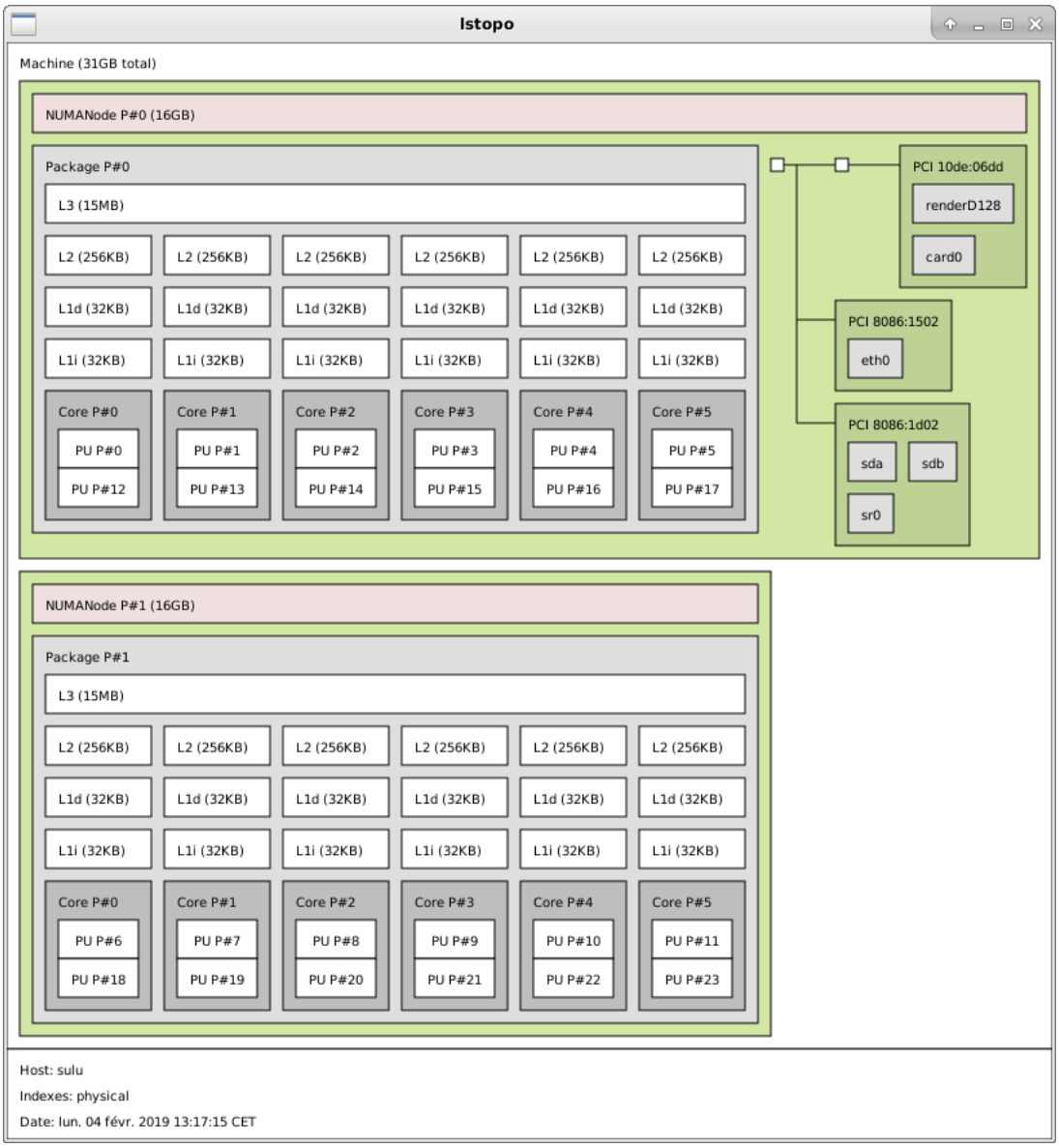
Nous avons observé que l'optimisation était plus efficace lorsque l'on utilise un paramètre k=1 lors de la parallélisation. Nous obtenons des résultats moins performants en augmentant la valeur de k.

Pour finir nous avons réalisé une parallélisation de la fonction **mandel_compute_tiled()** en utilisant une politique de distribution dynamique des tuiles à l'aide de la directive collapse. L'efficacité était optimale lorsque l'on effectue la parallélisation avec un paramètre k=2.

```
unsigned mandel_compute_omptiled(unsigned nb_iter)
{
  tranche = DIM / GRAIN;
  for (unsigned it = 1; it <= nb_iter; it++) {
    // On itére sur les coordonnées des tuiles
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < GRAIN; i++)
      for (int j = 0; j < GRAIN; j++)
        traiter_tuile (i * tranche /* i debut */, j * tranche /* j debut */
                       (i + 1) * tranche - 1 /* i fin */,
                       (j + 1) * tranche - 1 /* j fin */);
    zoom ();
  }
  return 0;
}
```

4 Expériences

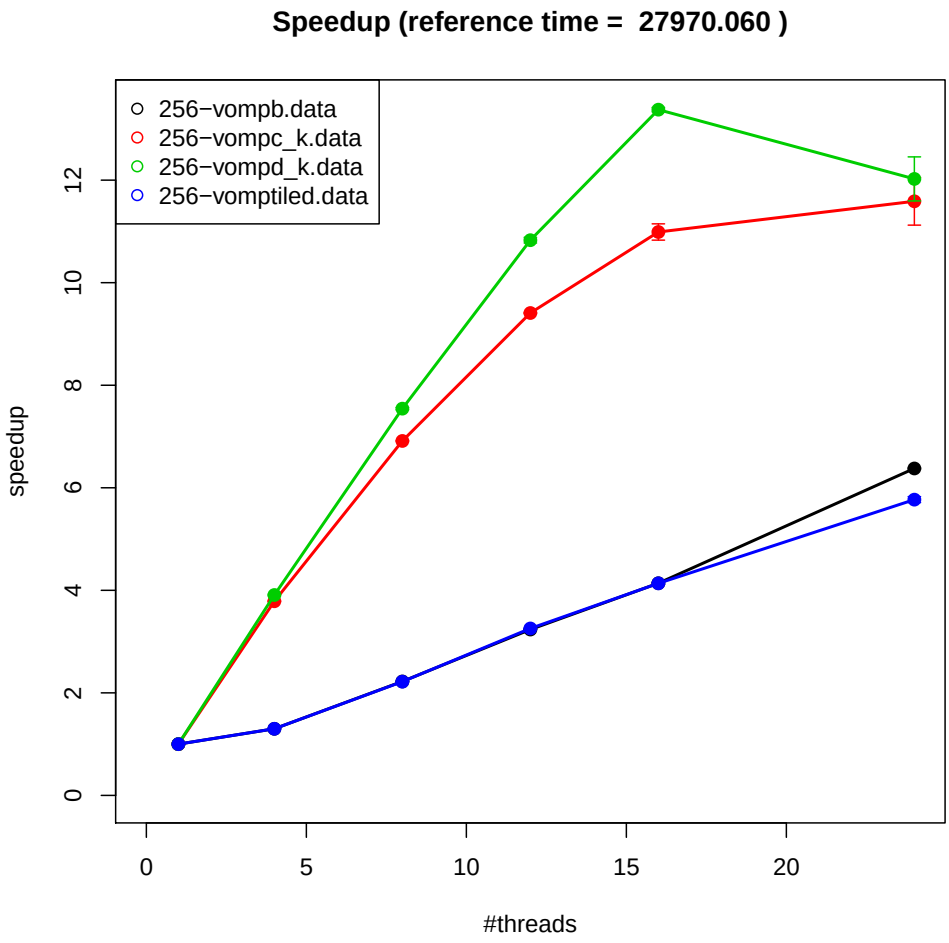
Nous avons réalisé nos expériences dans la salle 203 au CREMI sur la machine **sulu**, dont la configuration est la suivante :

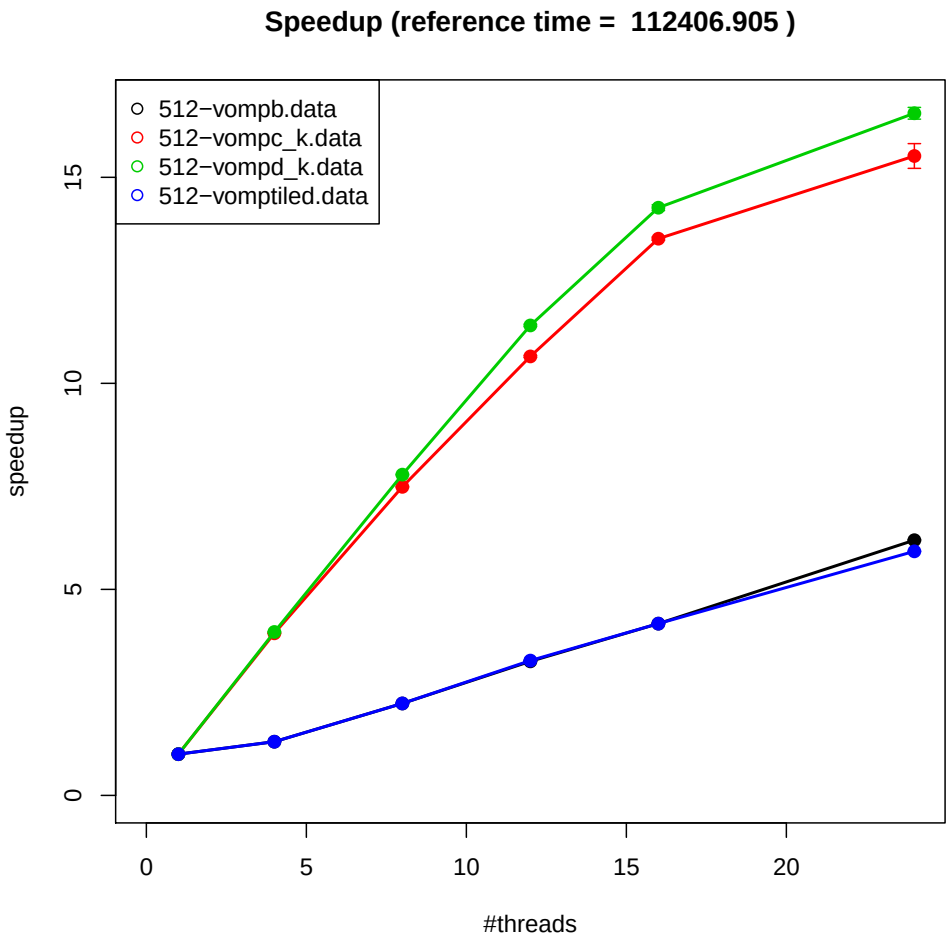


Nous avons effectué nos expériences avec les paramètres suivants :

```
export OMP_NUM_THREADS
export GOMP_CPU_AFFINITY

ITE=$(seq 3) # nombre de mesures
THREADS="1 4 8 12 16 24" # nombre de threads à utiliser pour les expés
GOMP_CPU_AFFINITY=$(seq 0 23) # on fixe les threads

PARAM="./2Dcomp -n -k mandel -i 100 -g 16 -s " # parametres commun à toutes les executions

execute (){
EXE="$PARAM $*"
OUTPUT="$(echo $* | tr -d ' ')".data
for nb in $ITE; do for OMP_NUM_THREADS in $THREADS ; do echo -n "$OMP_NUM_THREADS " >> $OUTPUT  ; $EXE 2>> $OUTPUT ; done; done
}

# on suppose avoir codé 2 fonctions :
#   mandel_compute_omp_dynamic()
#   mandel_compute_omp_static()

for i in 256 512 ;  # 2 tailles : -s 256 puis -s 512
do
        execute  $i -v ompb
        execute  $i -v ompc_k
        execute  $i -v ompd_k
        execute  $i -v omptiled
done
```

Nous avons effectué nos expériences avec des images de tailles 256 puis 512 avec un nombre d'itérations égal à 100.


Speedup (reference time = 27970.060)
256−vompb.data
256−vompc_k.data
256−vompd_k.data
256−vomptiled.data
speedup
0
2
4
6
8
10
12
0
5
10
15
20
#threads



5 Conclusion

On remarque dans nos deux résultats un "classement" des parallélisations. Les courbes données par les fonctions **omptiled** et **ompd_k** sont confondues. Elles ont donc presque les mêmes performances. Les deux autres courbes sont également relativement proche, la plus performante étant **ompd_k**. Les courbes obtenues semblent en adéquation avec les tests de temps effectués précédemment.