TP2 : Etudes de performances

Borne, Isnel

1 Multiplication matrice/vecteur

Nous étudions les gains de performances liés à la parallélisation d'un algorithme de multiplication matrice/vecteur avec `OpenMP`. Nous évaluons les performances du code suivant :

```
id mult_mat_vector_tri_inf (matrix M, vector b, vector c) {
register int i ;
register int j ;
#pragma omp parallel for schedule(politique_ordonnancement, chunksize) private(i,
 ↪  j) num_threads(nb_thread)
for(i=0; i<N; i++){
  for(j=N-1; j>=0; j--){
    c[i] += b[j] * M[i][j];
  }
}
```

Nous nous intéressons en particulier à l'influence du choix de trois politiques d'ordonnancement (Statique, Dynamique, Guidé) ainsi que la valeur du paramètre `chunksize` sur le temps d'exécution de l'algorithme pour un nombre de fils d'exécution donné.

1.1 Algorithme

On considère une matrice M, de taille N fixée à 1024. La directive `OpenMP` permet de répartir les itérations des boucles entre différents threads $T_1, T_2, ..., T_{nb\_threads}$. Un thread T_i, au cours de l'exécution du programme, se voit affecté successivement des "blocs" (`chunks`) d'itérations dont il aura la charge. La taille de ces "blocs" est spécifiée par le paramètre `chunksize`.

1.2 Performances

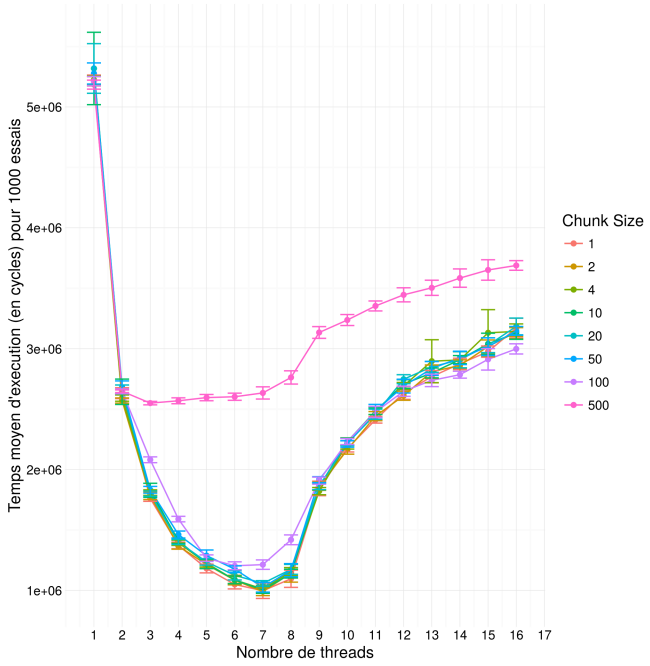
Notre machine possède 8 coeurs physiques et, grâce à l'hyperthreading, est capable d'exécuter deux threads par coeurs. Ainsi nous réalisons des mesures de

temps d'exécution pour des valeurs de `nb_thread` allant de 1 (exécution séquentielle) à 16. Pour chaque valeur de `nb_thread` nous faisons ensuite varier la valeur de `chunksize` parmi $\{1, 2, 4, 10, 20, 50, 100, 500\}$.

Remarque. Afin de pouvoir placer notre étude statistique dans le domaine d'application de la loi des grands nombres nous réalisons, pour chaque couple (`nb_thread`, `chunksize`), la mesure du temps d'exécution 1000 fois. La mesure du temps retournée par le programme est une moyenne sur 100 exécution de chaque fonction et nous lançons 10 fois le programme pour chaque mesure.

Ordonnancement statique

Avec la politique d'ordonnancement statique, la répartition des itérations entre les différents threads est figée avant l'exécution du programme.



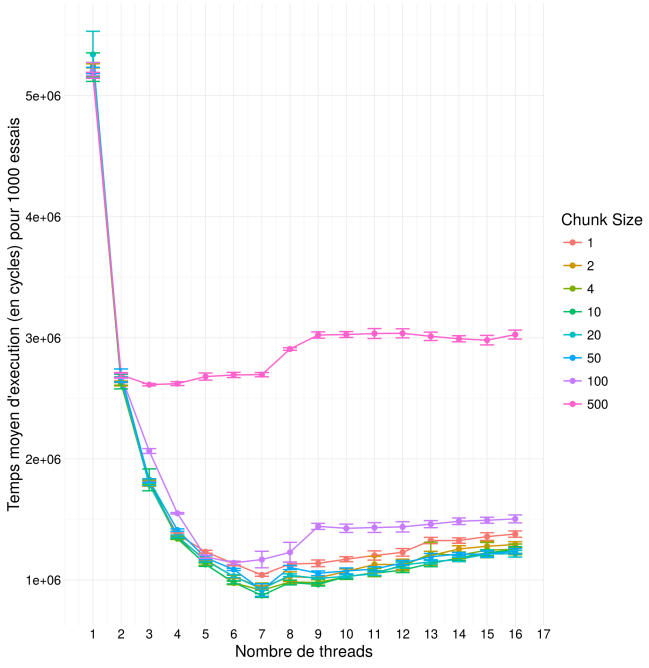
Remarque. Sur le graphique ci-dessus nous reportons pour chaque couple (`nb_thread`, `chunksize`) l'écart type calculé pour nos mesures sous forme de "bâtonnet" vertical.

Nous observons bien le gain de performance apporté par la parallélisation. Avec deux threads, on met deux fois moins de temps qu'avec un seul. Au delà de deux threads on observe le comportement pathologique pour `chunksize` égal à 500 avec des performances nettement dégradées. Pour les autres valeurs de `chunksize`, le meilleur temps moyen est obtenu avec 7 fil d'exécutions et on observe une

très nette dégradation des performances au delà de cette valeur. Avec 9 thread on est deux fois moins efficace qu'avec 7. Pour des valeurs situées en dessous d'une limite entre 100 et 500, `chunksize` n'a pas l'air d'affecter drastiquement les performances.

Ordonnancement dynamique

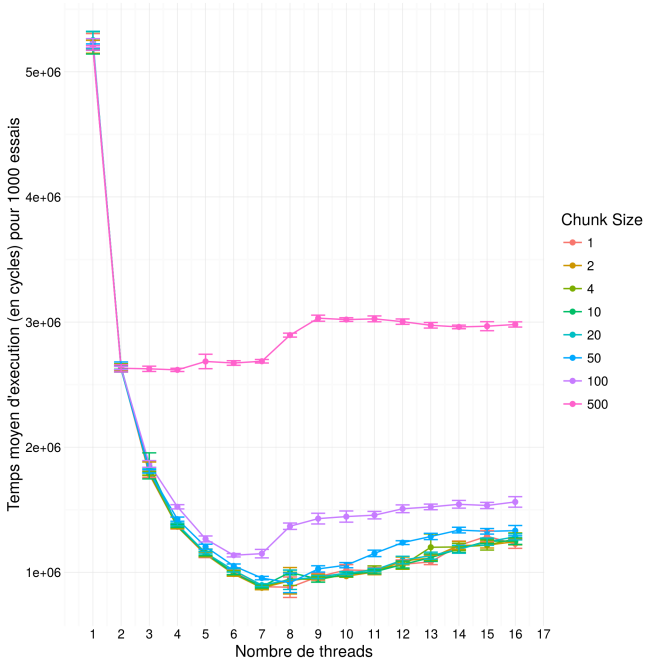
Contrairement à l'ordonnancement statique, la répartition des itérations entre les threads n'est pas connue avant l'exécution. Lorsqu'un thread finit d'exécuter sur son `chunk` d'itérations courant, il va chercher, parmi les itérations pas encore affectées, un nombre `chunksize`, de nouvelles itérations sur lesquelles travailler. L'intérêt de cette politique se comprend quand le temps d'exécution varie beaucoup entre les différentes itérations. Dans ces conditions, en mode statique, un thread pourrait se voir assigner tout le travail facile et attendre "les bras croisés" que les autres finissent les tâches plus compliquées. Dans notre application, il est raisonnable de supposer que les itérations sont de difficulté égale et qu'on ne gagnera pas forcément à utiliser un ordonnancement dynamique. En effet, pour un thread, le fait d'aller chercher de nouvelles itérations disponibles a un coût non négligeable. Le paramètre `chunksize` a précisément pour but de limiter l'overhead induit par cette recherche.



Nous observons ici aussi les gains de performances liés à la parallélisation et le même comportement pathologique pour `chunksize` égal à 500. Les meilleurs

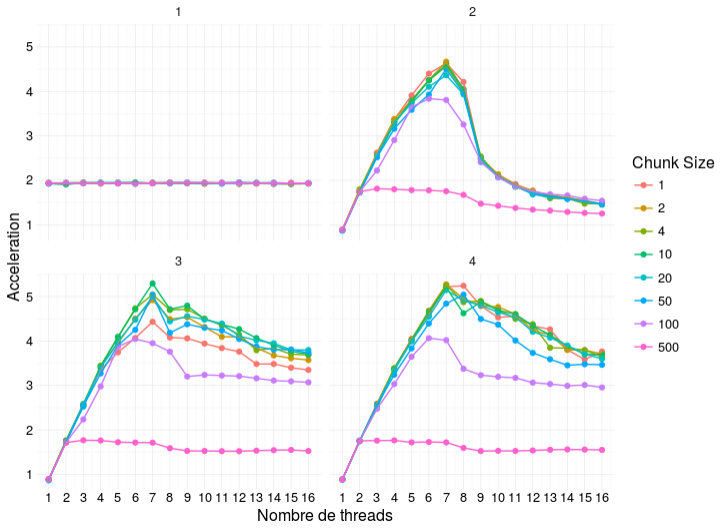
temps d'exécution sont obtenus pour 7 threads et, contrairement à l'ordonnancement statique, la dégradation des performances est peu marquée au delà de cette valeur. On observe aussi l'influence de la taille des `chunk`. Ici le temps moyen mesuré avec `chunksize` égal à 1 est moins bon que celui pour `chunksize` égal à 2, 4, 10, 20, 50. Aussi on met bien en évidence l'impact de l'opération de recherche des itérations disponibles par un thread et de l'avantage d'affecter des blocs d'itération. On remarque cependant que les gains observés avec des tailles de `chunk` > 1 sont d'un ordre de grandeur négligeable devant ceux observés par le passage d'un thread à deux thread par exemple. On voit aussi les effets négatifs que peut avoir une taille de `chunk` trop grande.

Ordonnancement guidé



Accélération

Une autre représentation des performances peut-être faite en terme de facteur d'accélération. l'accélération s'exprime comme le temps moyen de la version séquentielle d'un algorithme divisé par le temps moyen de la version parallèle. On prends pour référence de temps d'exécution séquentiel, celui de l'algorithme naïf de multiplication entre une matrice et un vecteur.



1 : Séquentiel "optimisé", 2 : Statique, 3 : Dynamique, 4 : Guidé

Remarque. Sur la figure ci-dessus, en haut à gauche est représentée la version "triangulaire inférieur" de l'algorithme séquentiel. Nous observons un facteur 2 pour l'accélération par rapport à la version naïve de l'algorithme. Ce résultat confirme bien le fait qu'on effectue deux fois moins d'opérations si l'on exclut les valeurs nulles au dessus de la diagonale de nos calculs.

Les graphiques 2, 3 et 4 représentent respectivement les accélérations pour les politiques d'ordonnancement statique, dynamiques et guidé. On observe que le meilleur gain est obtenu en ordonnancement dynamique avec un facteur 5.5 d'accélération pour 7 threads et une taille de `chunk` égale à 4. L'ordonnancement guidé, sur lequel nous ne nous sommes pas attardé faute de temps, donne sensiblement les mêmes performances que l'ordonnancement dynamique.

2 Tri à bulles

Nous étudions les gains de performances liés à la parallélisation d'un algorithme de tri à bulles avec `OpenMP`.

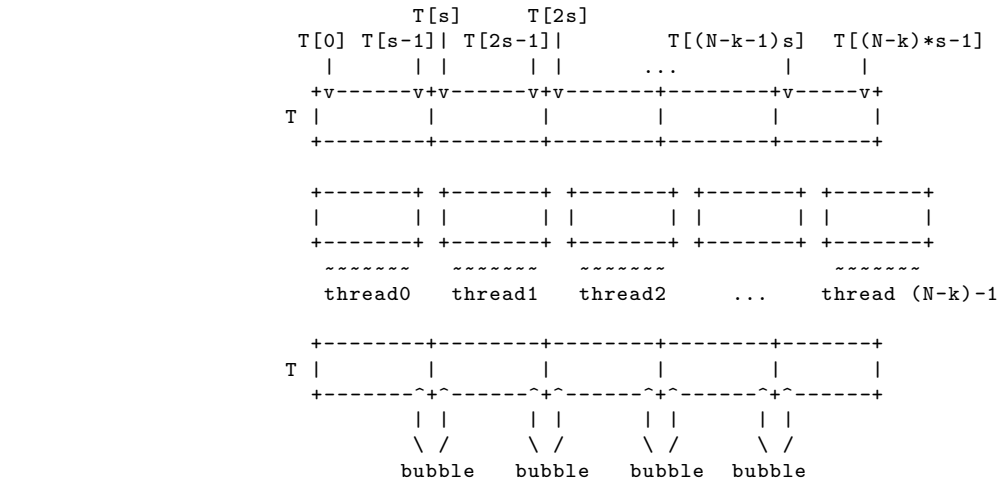
2.1 Algorithme

On considère le tri d'un tableau d'entiers `T` de taille 2^N. Pour notre expérience notre tableau sera de taille 4096, c'est à dire $N = 12$. Le principe de notre algorithme de tri par bulles parallèle est le suivant :

1. On divise `T` en sous blocs de taille $s = 2^k$ avec $k < N$.

2. Pour chacun des $(N-k)$ sous-blocs on fait une passe de bubble sort dans un thread séparé.
3. A l'issue des passes parallèles on effectue une passe bubble-sort sur les cases adjacentes des différents sous-blocs.
4. On répète l'opération tant que l'état du tableau `T` à été modifié lors d'une des étapes précédentes.

2.2 Illustrations



2.3 Performances

Nous évaluons les performances du code suivant :

```
id bloc_bubble_sort(int *T, const int size, const int blocsize)

  register int i;
  int swapped;
  int tmp;
  int j, k, l;
  int q = size / blocsize;
  do {
      swapped = 0;
      /* Une passe de Bubble dans chaque sous bloc du tableau T */
      #pragma omp parallel for schedule(politique_ordonancement, chunksize)
↪   private(i) num_threads(nb_thread)
      for (i = 0; i < q; i++) {
          if(bloc_bubble_pass(T+(i * blocsize), blocsize) == 1) swapped = 1;
      }
      /* Bubble sur les cases adjacentes des blocs */
      for (j = 0; j < q; j++) {
```

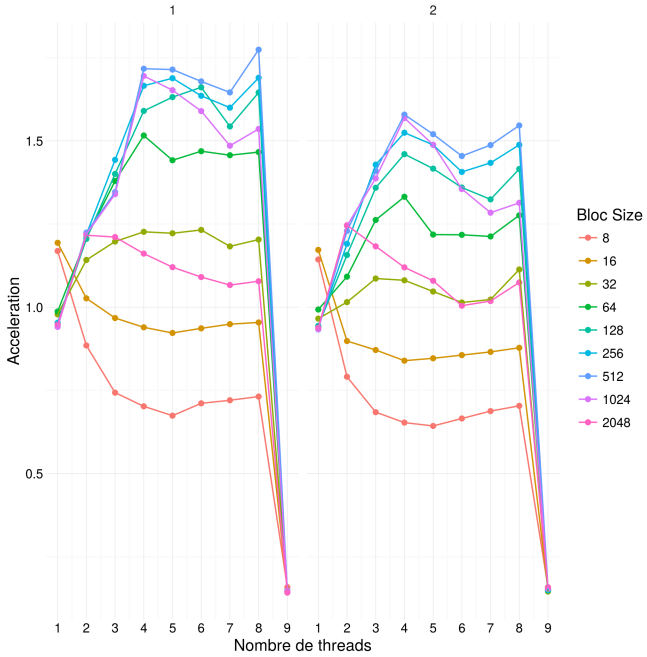
```
        k = j * blocsize - 1;
        l = k + 1;
        if (T[k] > T[l]) {
            tmp = T[k];
            T[k] = T[l];
            T[l] = tmp;
            swapped = 1;
        }
    }
} while (swapped);
return;
```

Nous nous intéressons en particulier à l'influence de la taille des blocs `blocsize` sur le temps d'exécution de l'algorithme pour un nombre de threads `nb_thread` donné. Nous mesurons donc l'accélération obtenue par rapport à la version séquentielle pour des tailles de blocs variant de 2^3 à 2^{11} pour un nombre de threads donné allant de 1 à 9. Nous comparons au passage les performances des politiques d'ordonnancement statique et dynamique.

Remarque. On calcule `chunksize` de la manière suivante :

```
if (nb_thread < q) {
    chunksize = q / nb_thread;
} else {
    chunksize = 1;
}
```

Accélération



1 : Ordonnancement statique, 2 : Dynamique

A gauche, l' accélération pour un ordonnancement statique et à droite, un ordonnancement dynamique. On observe que, globalement, pour des paramètres similaires l'ordonnancement dynamique donne de moins bonnes performances que l'ordonnancement statique avec des allures générales des courbes semblables. La meilleure performance est obtenue pour 8 threads et des blocs de taille 512. Intuitivement c'est bien le résultat attendu. Notre tableau est de taille 4096 et en le divisant en huit blocs (de taille 512) on se dit qu'on pourra tirer parti au mieux des huit coeurs de calcul de notre processeur en affectant le travail de chaque bloc sur un coeur. Les mesures ci-dessus confirment bien cette intuition. On voit aussi ici que l'hyperthreading ne fonctionne pas bien pour notre application. Au delà de huit threads les performances s'effondrent.

3 Tri fusion

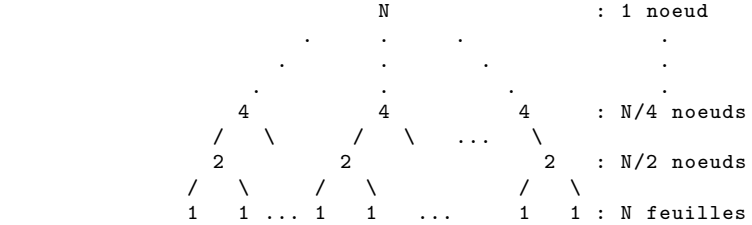
Nous nous intéressons aux gains de performances obtenues par parallélisation du tri par fusion. Notre étude porte sur le code suivant :

```
void sort(int *T, unsigned int debut, unsigned int fin){
  #pragma omp parallel num_threads(nb_thread)
```

```
  #pragma omp master
  {
    parallel_merge_sort(T, debut, fin);
  }
}

void parallel_merge_sort(int *T, unsigned int debut, unsigned int fin)
{
  if (debut < fin) {
    /* si le tableau à trier est petit on le trie
       avec une version séquentielle de merge_sort */
    if ((fin - debut) <= threshold) {
      merge_sort(T, debut, fin);
    } else {
      unsigned int milieu = (fin + debut) / 2;

      /* partie gauche */
      #pragma omp task
      parallel_merge_sort(T, debut, milieu);

      /* partie droite */
      #pragma omp task
      parallel_merge_sort(T, milieu + 1, fin);

      /* merge des deux parties du tableau */
      #pragma omp taskwait
      merge(T, debut, milieu, fin);
    }
  }

  return;
}
```

3.1 Nombre de tâches crées

Soit `T` un tableau de taille $N = 2^n$ à trier. Avec l'algorithme proposé dans l'énoncé, lors de la division du tableau en deux sous-parties de tailles égales, chaque partie est affectée à une nouvelle tâche. On peut représenter les opérations successives de divisions par un arbre binaire complet :



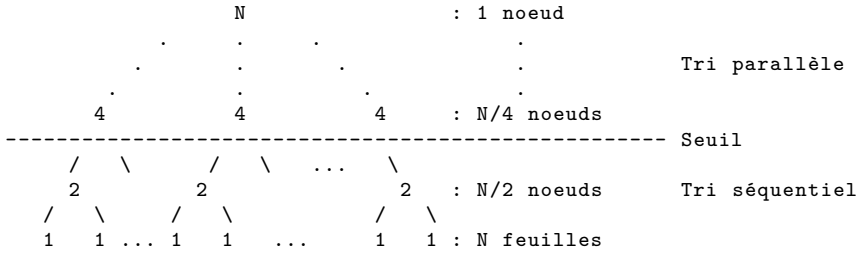
Chaque noeud de l'arbre représente une tâche affectée au tri d'un sous-tableau dont la taille est l'étiquette du noeud. Le nombre de noeuds de l'arbre est

$$\sum_{k=0}^{n} N * 2^{-k}$$

et sa hauteur est $log_2(N) = n$.

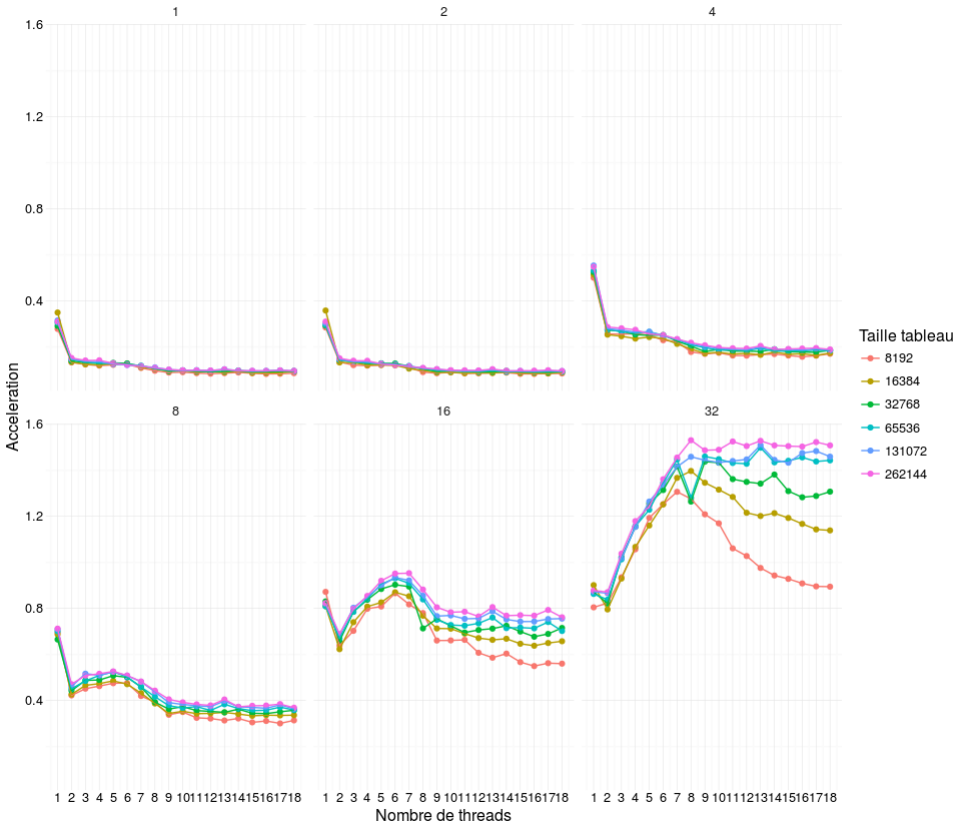
3.2 Seuil

Afin de limiter le nombre de tâches crées, nous proposons d'introduire un seuil (`threshold`) pour la taille des sous-tableaux. Lorsqu'un tableau à trier est de taille inférieure à ce seuil on effectue le tri avec une version séquentielle de l'algorithme. Ainsi on ne parallélise que le tri des noeuds en haut de l'arbre.



3.3 Accélération

Nous mesurons le temps de tri de tableaux de tailles allant de 2^{13} à 2^{18}. Pour calculer l'accélération, pour chaque triplet (`nb_thread`, `taille_tableau`, `threshold`) nous prenons pour référence le temps moyen d'exécution séquentielle du tri par fusion pris sur 100 expériences. Nous observons ici l'influence de la taille du seuil sur l'accélération. Au dessus de chaque graphique est indiqué le seuil utilisé pour les mesures.



En haut à gauche, on observe que pour la version naïve de l'algorithme, avec une tâche pour chaque sous-tableau, on obtient des performances beaucoup moins bonnes que la version séquentielle. Les premiers gains commencent à apparaître avec un seuil égal à 32 (en bas à droite). On poursuit notre expérimentation avec des seuils plus grands.

On observe que si l'on dimensionne correctement la valeur du seuil, on arrive à obtenir un facteur 2.5 d'accélération. Intuitivement on voit que prendre un seuil trop proche de la taille du tableau à trier revient à faire majoritairement du séquentiel, et prendre un seuil trop petit revient à paralléliser inutilement. On perçoit avec notre algorithme une nette amélioration des performances du tri fusion. Intuitivement nous ne pensions pas obtenir de si bon résultats en raison des opérations de fusion s'effectuant en séquentiel. Nous sommes donc très satisfait de notre choix.

4 Conclusion

Ce travail constituait une première expérience avec `OpenMP`. Nous avons pû confronter nos intuitions sur les gains de performances à des mesures et une étude statistique concrète. Nous avons apprécié la simplicité de mise en oeuvre du parallélisme avec `OpenMP` et l'efficacité du langage `R` pour l'analyse statistique de nos résultats expérimentaux.

5 Annexe

5.1 Conditions expérimentales

Infos Cpu

la commande `lscpu` permet d'afficher les informations liées aux processeurs.

```
Architecture :                          x86_64
Mode(s) opératoire(s) des processeurs :  32-bit, 64-bit
Boutisme :                              Little Endian
Processeur(s) :                         8
Liste de processeur(s) en ligne :       0-7
Thread(s) par œcur :                    2œ
Cur(s) par socket :                    4
Socket(s) :                             1œ
Nud(s) NUMA :                          1
Identifiant constructeur :              GenuineIntel
Famille de processeur :                 6
Modèle :                                94
Nom de modèle :                         Intel(R) Core(TM) i7-6700 CPU @
    3.40GHz
Révision :                              3
Vitesse du processeur en MHz :          799.987
Vitesse maximale du processeur en MHz :  4000,0000
Vitesse minimale du processeur en MHz :  800,0000
BogoMIPS :                              6818.00
Virtualisation :                        VT-x
Cache L1d :                             32K
Cache L1i :                             32K
Cache L2 :                              256K
Cache L3 :                              8192Kœ
Nud NUMA 0 de processeur(s) :          0-7
Drapaux :                               fpu vme de pse tsc msr pae mce cx8
    apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss
    ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs
    bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq
    pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma
cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes
    xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault intel_pt
    tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2
    smep bmi2 erms invpcid rtm mpx rdseed adx smap clflushopt xsaveopt xsavec
    xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window
    hwp_epp
```

5.2 Info Mémoire

Nous récupérons les informations concernant la mémoire avec la commande `sudo lshw`

```
 description: Mémoire Système
       identifiant matériel: 41
       emplacement: Carte mère
       taille: 32GiB
     *-bank:0
          description: DIMM DDR4 Synchrone 2133 MHz (0,5 ns)
```

```
     produit: CMK32GX4M2A2133C13
     fabriquant: AMI
     identifiant matériel: 0
     numéro de série: 00000000
     emplacement: ChannelA-DIMM0
     taille: 16GiB
     bits: 64 bits
     horloge: 2133MHz (0.5ns)
*-bank:1
     description: DIMM DDR4 Synchrone 2133 MHz (0,5 ns)
     produit: CMK32GX4M2A2133C13
     fabriquant: AMI
     identifiant matériel: 1
     numéro de série: 00000000
     emplacement: ChannelA-DIMM1
     taille: 16GiB
     bits: 64 bits
     horloge: 2133MHz (0.5ns)
```