Université Ibn Tofaïl

Faculté des Sciences
كلية العلوم

UNIVERSITÉ IBN TOFAIL

FACULTÉ DES SCIENCES - KENITRA

DÉPARTEMENT INFORMATIQUE

MASTER INFORMATIQUE ET INTELLIGENCE ARTIFICIELLE

COMPTE RENDU

TP 5 : Optimisation de calculs parallèles avec la réduction en OpenMP

Réalisé par :
Anas BOUKHLIJA

Encadré par :
Pr. Nada FAQIR

ANNÉE UNIVERSITAIRE 2024-2025

Table des matières

1 Phase 1 : Implémentation d'un calcul parallèle simple avec réduction 3

1.1 Implémentation de l'algorithme de Monte-Carlo 3

1.2 Utilisation de la réduction en OpenMP 4

2 Phase 2 : Optimisation des performances 5

2.1 Stratégies d'optimisation 5

2.2 Tests de performance 7

3 Phase 3 : Mesure et analyse des performances 7

3.1 Mesure du speedup et de l'efficacité 7

3.2 Analyse des points de contention 10

Table des figures

1 Phase 1 : Implémentation d'un calcul parallèle simple avec réduction

1.1 Implémentation de l'algorithme de Monte-Carlo

Implémentation de l'algorithme de Monte-Carlo pour estimer la valeur de pi en utilisant une approche séquentielle.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {

    // Nombre d'echantillons a generer
    int n     = 1000000000;
    int count = 0;

    // Initialiser le generateur de nombres aleatoires
    srand(time(NULL));

    // Boucle pour generer des points aleatoires
    for (int i = 0; i < n; i++) {
        // Generer des coordonnees aleatoires dans le carre [0, 1] x [0, 1]
        double x = (double)rand() / RAND_MAX;
        double y = (double)rand() / RAND_MAX;

        // Verifier si le point est a l'interieur du cercle de rayon 1
        if (x * x + y * y <= 1) {
            // Point a l'interieur du cercle
            count++;
        }
    }

    // Estimer la valeur de pi
```

```
    double pi = (double)count / n * 4;

    printf("Estimation de Pi = %f\n", pi);
    return 0;
}
```

1.2 Utilisation de la réduction en OpenMP

Parallélisation du calcul du nombre de points à l'intérieur du cercle grâce à la réduction avec OpenMP.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

int main() {

    // Nombre d'echantillons a generer
    int n     = 1000000000;
    int count = 0;

    // Initialiser le generateur de nombres aleatoires
    srand(time(NULL));

    // Boucle pour generer des points aleatoires
    #pragma omp parallel
    {
        unsigned int seed = time(NULL) ^ omp_get_thread_num();
        #pragma omp for reduction(+:count)
        for (int i = 0; i < n; i++) {
            // Generer des coordonnees aleatoires dans le carre
   [0, 1] x [0, 1]
            double x = (double)rand_r(&seed) / RAND_MAX;
            double y = (double)rand_r(&seed) / RAND_MAX;

            // Verifier si le point est a l'interieur du cercle de
    rayon 1
```

```
            if (x * x + y * y <= 1) {
                // Point a l'interieur du cercle
                count++;
            }
        }
    }


    // Estimer la valeur de pi
    double pi = (double)count / n * 4;

    printf("Estimation de Pi = %f\n", pi);
    return 0;
}
```

2 Phase 2 : Optimisation des performances

2.1 Stratégies d'optimisation

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

int main() {
    int n = 1000000000;
    int chunk_size = n / 32;

    srand(time(NULL));

    for (int num_threads = 2; num_threads <= 8; num_threads+=2) {
        int global_count = 0;
        double start_time, end_time;

        // Set the number of threads
        omp_set_num_threads(num_threads);

```

```
19          start_time = omp_get_wtime(); // Start timing
20
21          #pragma omp parallel
22          {
23              int local_count = 0;
24              unsigned int seed = time(NULL) ^ omp_get_thread_num();
25
26              #pragma omp for schedule(dynamic, chunk_size)
27              for (int i = 0; i < n; i++) {
28                  double x = (double)rand_r(&seed) / RAND_MAX;
29                  double y = (double)rand_r(&seed) / RAND_MAX;
30
31                  if (x * x + y * y <= 1) {
32                      local_count++;
33                  }
34              }
35
36              #pragma omp atomic
37              global_count += local_count;
38          }
39
40          end_time = omp_get_wtime();
41
42          double pi = (double)global_count / n * 4;
43
44          printf("Threads : %d, Estimation de Pi = %f, Temps d'
    ex cution = %f secondes\n", num_threads, pi, end_time -
    start_time);
45      }
46
47      return 0;
48  }
```

Nous avons optimisé l'algorithme Monte-Carlo pour estimer pi avec plusieurs améliorations :

- **Équilibrage des charges :** Utilisation de schedule(dynamic, chunk-size) pour une meilleure répartition des itérations entre les threads.
- **Réduction des conflits de cache :** Compteurs locaux par thread pour éviter les accès

concurrents à une variable partagée.

- **Minimisation des dépendances :** Variables locales pour chaque thread, réduisant les accès partagés et les blocages.

2.2 Tests de performance

Résultat de l'exécution

==========

Threads : 2, Estimation de Pi = 3.141573, Temps d'exécution = 11.731465 secondes
Threads : 4, Estimation de Pi = 3.141568, Temps d'exécution = 10.738926 secondes
Threads : 6, Estimation de Pi = 3.141603, Temps d'exécution = 9.323693 secondes
Threads : 8, Estimation de Pi = 3.141602, Temps d'exécution = 10.273215 secondes

3 Phase 3 : Mesure et analyse des performances

3.1 Mesure du speedup et de l'efficacité

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

int main() {

    int n = 1000000000;
    int chunk_size = n / 32;

    // ----- Version Sequentielle -----
    unsigned int seed_seq = time(NULL);
    double start_time_seq = omp_get_wtime();
    int global_count_seq = 0;

    for (int i = 0; i < n; i++) {
        double x = (double)rand_r(&seed_seq) / RAND_MAX;
        double y = (double)rand_r(&seed_seq) / RAND_MAX;

```

```
20          if (x * x + y * y <= 1) {
21              global_count_seq++;
22          }
23      }
24
25      double end_time_seq = omp_get_wtime();
26      double pi_seq = (double)global_count_seq / n * 4;
27      printf("Version sequentielle: Estimation de Pi = %f, Temps d'
    execution = %f secondes\n", pi_seq, end_time_seq -
    start_time_seq);
28
29      // ----- Version Parall le -----
30      for (int num_threads = 2; num_threads <= 8; num_threads += 2)
    {
31          int global_count = 0;
32          double start_time, end_time;
33
34          omp_set_num_threads(num_threads);
35
36          start_time = omp_get_wtime();
37
38          #pragma omp parallel
39          {
40              int local_count = 0;
41              unsigned int seed = time(NULL) ^ omp_get_thread_num();
42
43              #pragma omp for schedule(dynamic, chunk_size)
44              for (int i = 0; i < n; i++) {
45                  double x = (double)rand_r(&seed) / RAND_MAX;
46                  double y = (double)rand_r(&seed) / RAND_MAX;
47
48                  if (x * x + y * y <= 1) {
49                      local_count++;
50                  }
51              }
52
53              #pragma omp atomic
```

```
54              global_count += local_count;
55          }
56
57          end_time = omp_get_wtime();
58
59          double pi = (double)global_count / n * 4;
60
61          // Calcul du speedup et de l'efficacite
62          double speedup = (end_time_seq - start_time_seq) / (
    end_time - start_time);
63          double efficiency = speedup / num_threads;
64
65          printf("Threads : %d, Estimation de Pi = %f, Temps d'
    execution = %f secondes, Speedup = %f, Efficacite = %f\n",
66                 num_threads, pi, end_time - start_time, speedup,
    efficiency);
67      }
68
69      return 0;
70  }
```

3.2 Analyse des points de contention

Résultat de l'exécution

==========

Version séquentielle : Estimation de Pi = 3.141595, Temps d'exécution = 18.979354 secondes
Threads : 2, Estimation de Pi = 3.141587, Temps d'exécution = 11.272648 secondes, Speedup = 1.683664, Efficacité = 0.841832
Threads : 4, Estimation de Pi = 3.141576, Temps d'exécution = 8.085170 secondes, Speedup = 2.347428, Efficacité = 0.586857
Threads : 6, Estimation de Pi = 3.141597, Temps d'exécution = 7.302728 secondes, Speedup = 2.598940, Efficacité = 0.433157
Threads : 8, Estimation de Pi = 3.141608, Temps d'exécution = 6.353411 secondes, Speedup = 2.987270, Efficacité = 0.373409

Interprétation

- **Estimation de Pi** : Les résultats sont cohérents et proches de la valeur réelle de Pi.
- **Temps d'exécution** : Le temps diminue avec l'augmentation du nombre de threads, indiquant une amélioration des performances.
- **Speedup** : Modeste, atteignant 2.99 avec 8 threads, suggérant des limites à l'efficacité de la parallélisation.
- **Efficacité** : Excellente avec 2 threads (0.84) mais diminue à 0.37 avec 8 threads, indiquant une surcharge croissante.