HPC I

# Benchmark do cálculo de Pi

Pedro Henrique dos Santos Cunha

Universidade Federal Fluminense
Instituto de Ciências Exatas
Departamento de Física

# 1 Introdução

Nesse trabalho implementamos o algoritmo do cálculo de Pi em serial e em paralelo utilizando OpenMP. Para o código em paralelo, foram utilizadas 8 threads. O cálculo foi realizado na minha maquina pessoal, cujas especificações estão listadas abaixo:

- Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz Max: 4.10GHz
- 4 Core(s) per socket with 2 Thread(s) per core
- 16GB RAM

# 2 Calculo de Pi em SERIAL

O cálculo do valor de Pi em serial foi feito sem muitas complicações. O algoritmo foi dado em aula no slide 18, sendo assim, só fizemos a implementação do código em C, onde usamos a struct timespec para realizar o registro do tempo utilizado no cálculo. Utilizamos também o valor de $N = 100000$ iterações para o calculo. Repetimos o algoritmo 100 vezes de forma a tentar atingir uma média, uma vez que a função rand sorteia números aleatórios. Os valores encontrados na média estão listados na tabela abaixo.

| **Valor de Pi** | **Tempo em segundos** |
|---|---|
| 3.141520 | 0.002427 |

Tabela 1: Resultados do cálculo de Pi do algoritmo em SERIAL.

# 3 Calculo de Pi em paralelo utilizando OpenMP

Para realizar o cálculo de Pi em paralelo, primeiramente fixamos o número de iterações $N = 100000$, onde utilizamos o mesmo valor do cálculo em SERIAL. O algoritmo foi feito de forma a comparar o tempo do cálculo ao variarmos o número de chunks utilizados nas schedules. O registro do tempo foi feito utilizando novamente a struct timespec.

As chunks são regiões de memória de tamanho pré-definido que são distribuidas para as threads. A distribuição pode ser feita de forma dinâmica, estáticas ou guiadas. Neste trabalho, fizemos a distribuição de forma dinâmica e estática.

A schedule estática distribui de forma sequêncial e estática - por isso o nome - a memória para cada thread utilizada, caso essa divisão seja feita de maneira irregular (ocasião que pode ocorrer tanto pelo número de threads utilizado quanto pelo número do chunk utilizado), encontraremos quedas de perfomances na execução do código.

A schedule dinâmica distribui de forma mais flexível essa região de memória para evitar irregularidades nessa divisão. No entanto, caso ocorra da divisão também ser feita de forma irregular, quedas de perfomance maiores acontecerão.

Variamos o número de chunks de forma a analisar a perfomance para cada valor diferente de chunk e para cada thread a fim de analisar os resultados de maneira mais conclusiva.


Figura 1: Cálculo de Pi paralelizado com OpenMP com Schedules estáticas.

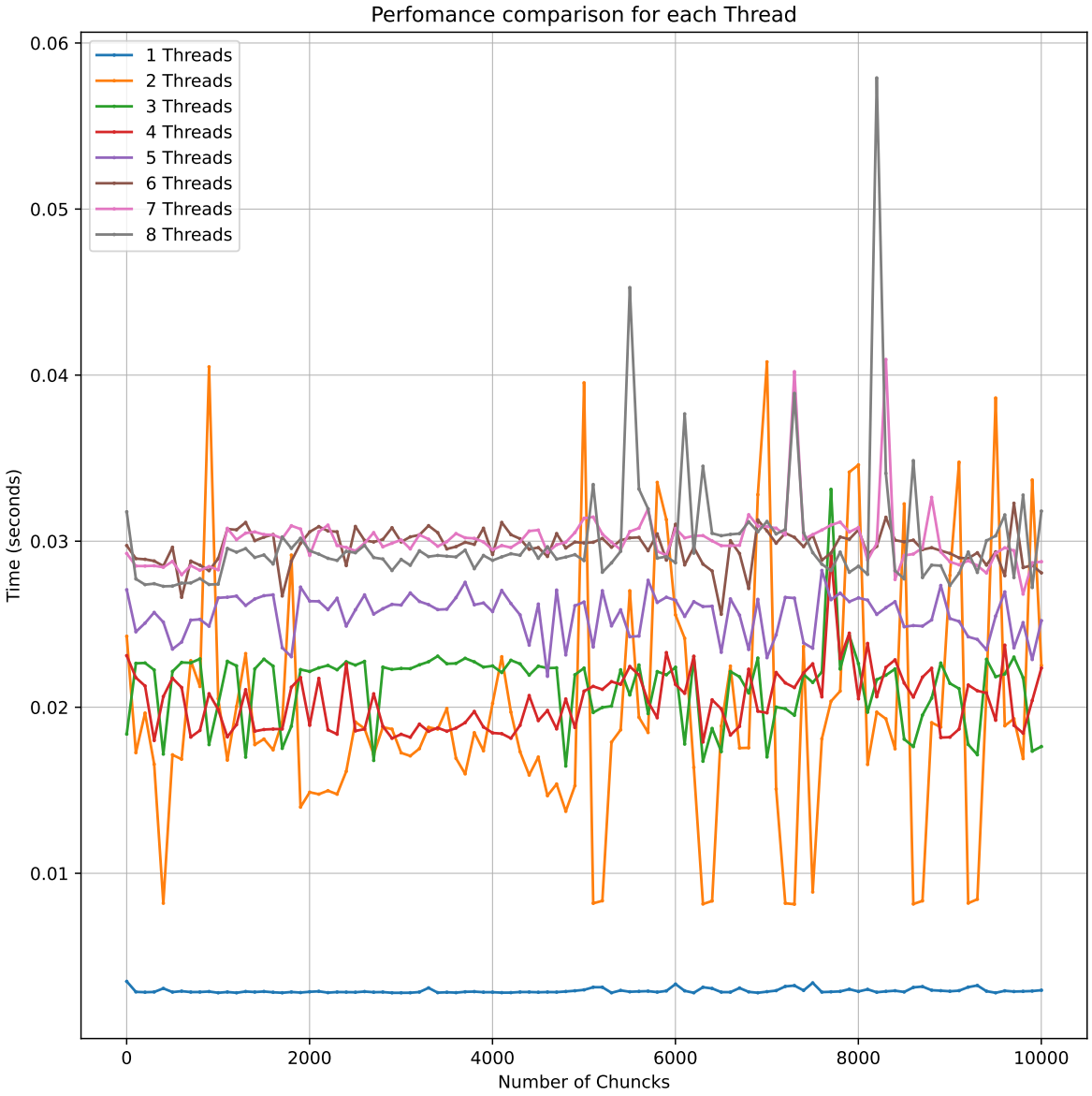


Figura 2: Cálculo de Pi paralelizado com OpenMP com Schedules dinâmicas.

Repare que os resultados encontrados confirmam a diferença anteriormente citada entre as schedules dinâmicas e estáticas. Para as schedules estáticas encontramos muito mais oscilações e quedas de perfomances, quando comparado as schedules dinâmicas. Em contrapartida, repare que para 8 threads a pico de oscilação para as schedules dinâmicas é muito maior, chegando a atingir 0.058 segundos em seu pico máximo (chunk = 8201).

O resultado com menor tempo se deu nas schedules dinâmicas para 1 thread, onde encontramos:

| **Número de threads** | **Valor de Pi** | **Chunk** | **Tempo em segundos** |
|---|---|---|---|
| 1 | 3.143440 | 1901 | 0.002819 |

Tabela 2: Resultado com menor tempo para o cálculo de Pi utilizando o código em paralelo.

No entanto, repare que o valor de Pi encontrado no menor tempo no código em paralelo difere bastante do valor de referência. O valor mais próximo da referência encontrado ocorre para 2 threads e está listado abaixo:

| **Número de threads** | **Valor de Pi** | **Chunk** | **Tempo em segundos** |
|---|---|---|---|
| 2 | 3.141560 | 4201 | 0.019706 |

Tabela 3: Resultado com o melhor valor de Pi utilizando o código em paralelo.

Embora o valor encontrado seja mais próximo da referência do que o valor listado na utilização do código em SERIAL, o tempo é bem maior.

# 4 Conclusões

Podemos concluir que o código implementado em paralelo possui uma maior exatidão quando comparamos o valor encontrado com o valor de referência. No entanto, o tempo para a execução se mostrou pior em relação ao código em SERIAL.

Além disso, com a análise feita, conseguimos mostrar a diferença entre as schedules dinâmicas e estáticas e como elas funcionam quando variamos o número de chunks. Tais características revelam a importância da escolha do tipo de schedule e do tamanho do chunk a ser utilizado em códigos mais sofisticados e nos mostram um caminho para tomarmos a melhor escolha.

# 5 Codigo do Cálculo de Pi em SERIAL

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

#define N 100000

int main(int argc, char *argv[]){

    struct timespec begin, end;
    FILE *data;

    data = fopen(argv[1], "w+");

    fprintf(data, "PI\tT\n");

  double pi, x_coordinate, y_coordinate, circle_count, realTime, realTimeNano,
    realTimeSec;
    int step, numberSteps;

    pi = 0.0;
  x_coordinate = 0.0;
  y_coordinate = 0.0;
    circle_count = 0.0;
    numberSteps = N;

    clock_gettime(CLOCK_REALTIME, &begin);

    for (step=0; step<numberSteps; step=step+1){

        x_coordinate=((float)rand())/RAND_MAX;
        y_coordinate=((float)rand())/RAND_MAX;

        if(x_coordinate*x_coordinate + y_coordinate*y_coordinate < 1){
            circle_count = circle_count + 1;
        }
    }

    pi=4.0*circle_count/numberSteps;
    clock_gettime(CLOCK_REALTIME, &end);

    realTimeSec = end.tv_sec - begin.tv_sec;
    realTimeNano = end.tv_nsec - begin.tv_nsec;
  realTime = realTimeSec + realTimeNano*1e-9;

    fprintf(data, "%lf %lf\n", pi, realTime);

    fclose(data);

    return 1;
}
```

Listing 1: Programa Serial

# 6 Codigo do Cálculo de Pi em paralelo com OpenMP

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

#define N 100000

int main(int argc, char *argv[]){

    struct timespec begin, end;
    FILE *data;

    data = fopen(argv[1], "w+");

    fprintf(data, "NT\tPI\tCH\tT\n");

  double pi, x_coordinate, y_coordinate, circle_count, realTime, realTimeSec,
    realTimeNano;
  int step, chunk, numberSteps, numberThreads, tid, n;

    pi = 0.0;
  x_coordinate = 0.0;
  y_coordinate = 0.0;
    circle_count = 0.0;

  numberSteps = N;
    tid = -1;
    srand(time(NULL));

    for(chunk = 1; chunk <= 10001; chunk+=100){

      numberThreads = 8;

      while(numberThreads != 0){

          circle_count = 0.0;
          omp_set_num_threads(numberThreads);

            clock_gettime(CLOCK_REALTIME,&begin);
            #pragma omp parallel private(step, x_coordinate, y_coordinate, tid)
            {
                    #pragma omp for reduction(+:circle_count) schedule(dynamic, chunk)

                    for (step=0; step<numberSteps; step=step+1){
                        x_coordinate=((float)rand())/RAND_MAX;
                        y_coordinate=((float)rand())/RAND_MAX;

                        if(x_coordinate*x_coordinate + y_coordinate*y_coordinate < 1){
                            circle_count = circle_count + 1;
                    }
                }
            }

```

```
53              pi=4.0*circle_count/numberSteps;
54
55          clock_gettime(CLOCK_REALTIME, &end);
56
57              realTimeSec = end.tv_sec - begin.tv_sec;
58              realTimeNano = end.tv_nsec - begin.tv_nsec;
59        realTime = realTimeSec + realTimeNano*1e-9;
60
61          fprintf(data, "%d %lf %d %lf\n", numberThreads, pi, chunk, realTime);
62          numberThreads--;
63
64        }
65      }
66
67      fclose(data);
68
69      return 1;
70  }
```

Listing 2: Programa paralelizado com OpenMP