COC 472 - Computação de Alto Desempenho: Trabalho 2

Professores: Jose Camata e Álvaro Coutinho

Lucas de Carvalho Gomes

DRE: 113080060

Conteúdo

1 Recursos Utilizados

1.1 *Hardware e Sistema Operacional*

Foi utilizado um *desktop* com o sistema operacional Fedora 23, uma distribuição do GNU/Linux. O computador possui o processador Intel Core i5 750, descrito na Tabela 1.

Modelo	Intel Core i5 750
Clock Máximo (1 ou 2 núcleos)	3,2 GHz
Clock Máximo (3 ou 4 núcleos)	2,66 GHz
Cache L1	32 KB/core
Cache L2	256 KB/core
Cache L3	8 MB
Tamanho dos Cache Lines	64 B

Tabela 1: Especificações do processador utilizado.

Quanto à memória, foram utilizados 2 pentes de 4 GB DDR3, com *clock* de 1600 MHz. Os programas foram instalados diretamente na máquina, sem *containers* ou máquinas virtuais.

1.2 Linguagem e Compilador

Para os códigos elaborados, a linguagem utilizada foi C++. No Exercício 2, o código foi compilado com os compiladores do GNU e da Intel, para realizar comparações: para C, são o gcc e o icc; e para C++, são o g++ e o icpc. Nos demais exercícios, o código foi compilado apenas com o compilador da Intel, uma vez que, como será visto mais adiante, ele apresentou os melhores resultados.

1.3 Repositório e Controle de Versão

Os códigos utilizados, os arquivos com os resultados e este relatório estão hospedados em um repositório público no *GitHub*. Ele pode ser acessado pela URL `https://github.com/Lucas-CG/HPC`.

2 Exercício 1: Perfilando códigos com OpenMP

2.1 Resultados

O código foi compilado com o script `tau_cc.sh` e o compilador gcc, usando a flag `-lm` para fazer o *link* de bibliotecas matemáticas do C (libm.so.6). O tamanho usado para o *array* foi 10^7.

A Figura 1 apresenta o gráfico tridimensional obtido com os resutlados da perfilagem pelo `paraprof`. Fica evidenciado que há alguns trechos de código (provavelmente laços) que consomem a grande maioria do tempo de CPU. O gráfico também indica uma quase uniformidade na distribuição do tempo de CPU entre as *threads*, com a exceção da última instrução (mais à direita), que apresenta uma quantidade um pouco maior para a *thread* 0.

Alterando a visão do gráfico tridimensional e usando o gráfico bidimensional (Figuras 2 e 3), podemos verificar quais são as partes do código que consomem o maior tempo de CPU, que são, em ordem decrescente de tempo de CPU: o laço das linhas 334 a 336 (operação SUM); o laço das linhas 344 a 346 (operação TRIAD); o laço das linhas 314 a 316 (COPY); e o laço das linhas 324 a 326 (SCALE); . Essa ordem é a mesma para todas as *threads*. A Tabela 2 mostra os tempos de CPU inclusivo (tempo de CPU dentro do

trecho somado ao tempo de CPU dentro de funções chamadas dentro do trecho) e exclusivo (apenas dentro do trecho) para essas 4 operações, por *thread*.

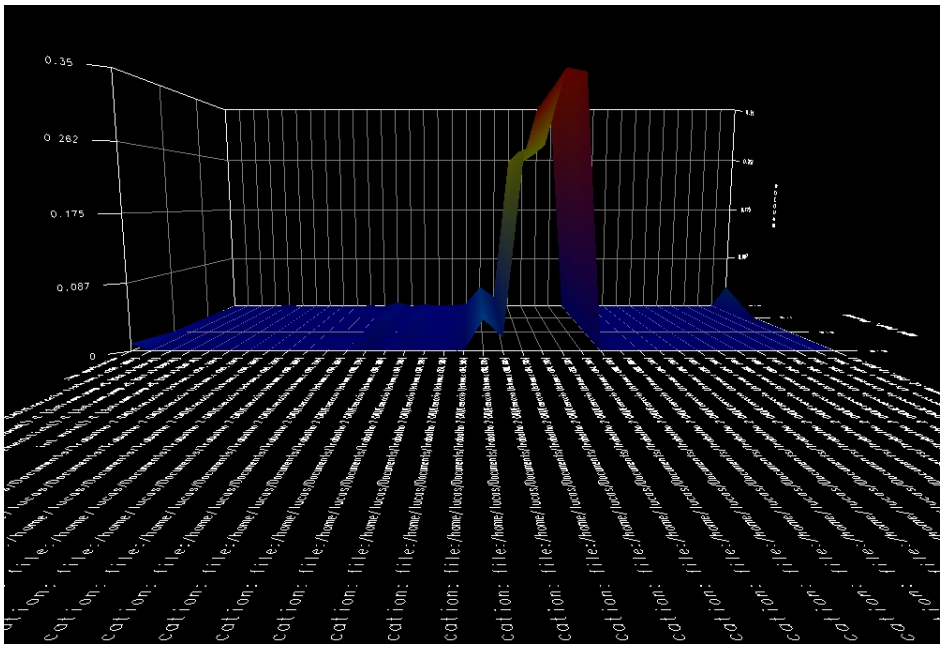

Figura 1: Gráfico tridimensional obtido com os resultados do TAU para o STREAM. Eixo x: trecho de código; eixo y: tempo; eixo z: *thread*.

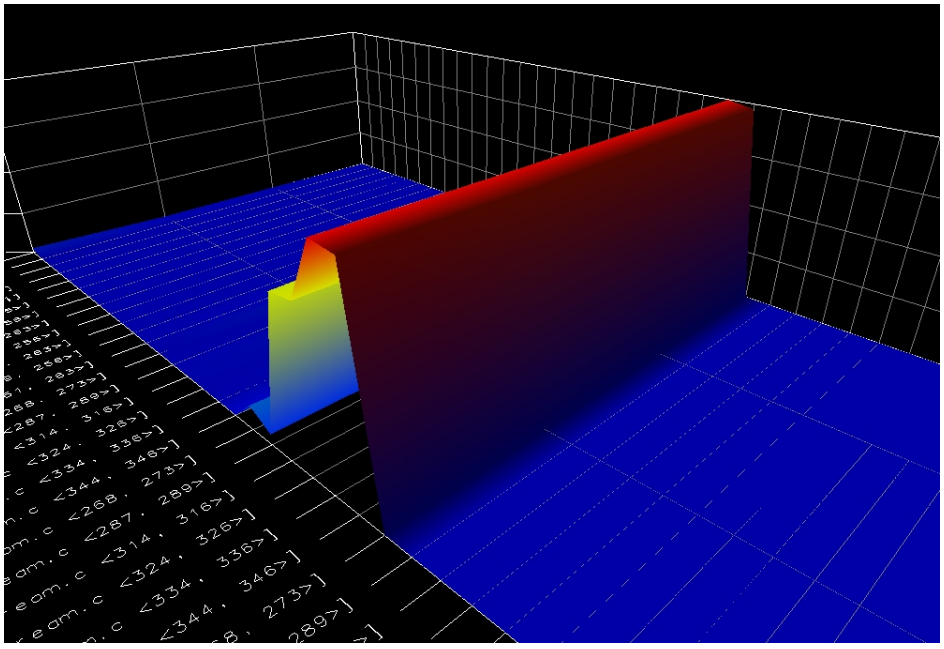

Figura 2: Gráfico tridimensional obtido com o TAU para o STREAm, com evidência no trecho onde há maior gasto de tempo de CPU.

Operação	*Thread*	*Tempo Exclusivo (s)*	*Tempo Inclusivo (s)*

SUM	0	0,349	0,35
	1	0,349	0,35
	2	0,349	0,35
	3	0,35	0,35
TRIAD	0	0,341	0,346
	1	0,342	0,346
	2	0,34	0,346
	3	0,345	0,346
COPY	0	0,26	0,263
	1	0,259	0,263
	2	0,258	0,263
	3	0,253	0,255
SCALE	0	0,258	0,259
	1	0,258	0,259
	2	0,257	0,259
	3	0,258	0,259

Tabela 2: Tempos de CPU consumidos por instrução por thread, para as 4 instruções que mais consomem tempo de CPU por ordem decrescente de consumo.

Dois resultados antiintuitivos que surgiram são os fatos de que o tempo de CPU da instrução COPY é maior que o da SCALE e de que o tempo de CPU da instrução TRIAD é menor que o da instrução SUM. Como todas as instruções envolvem uma cópia, o esperado era que a instrução COPY apresentasse os menores tempos, pois as outras instruções chamam outras subrotinas. Além disso, como a instrução TRIAD envolve uma multiplicação além da adição, o seu tempo de CPU ser menor que o da operação SUM também vai contra o esperado, mesmo se forem consideradas instruções FMA (*Fused Multiply-Add*), uma vez que uma FMA é executada em tempo igual ou um pouco maior que o tempo de uma soma convencional.

3 Exercício 2: Paralelizando a propagação da onda

3.1 Código Otimizado Serial

O código usado recebeu as seguintes modificações em relação ao original:

- Conversão das matrizes para *arrays* unidimensionais
  - Para isso, foram necessárias uma função que convertesse as três coordenadas em uma posição do vetor e outra que realizasse a conversão contrária.
- Ajustes no laço
  - O bloco condicional foi mesclado ao laço, reduzindo o número de iterações.
- Correção na deleção dos *arrays*, evitando vazamentos de memória
- Otmizações do Compilador (*flag* -O3)

A Tabela 3 apresenta os tempos reais de execução para a versão serial original e a otimizada, medidos com execuções dos códigos compilados com o gcc e o icc usando a ferramenta *time*.

Compilador	*Versão do Código*	*Tempo de Execução*
gcc	Original	35m 25.914s
gcc	Otimizado	16m32.392s
icc	Original	35m38.085s
icc	Otimizado	4m50.381s

Tabela 3: Tempos reais de execução para o WAVE serial: código original e otimizado.

Analisando esses resultados, vemos que as otimizações realizadas reduziram o tempo aproximadamente pela metade. Mais ainda, o uso de otimizações do compilador da Intel (logo, específicas para a arquitetura da Intel) reduziram ainda mais o tempo, obtendo uma diferença total de 30 minutos e 35 segundos.

3.2 Paralelização do Código

A paralelização do código ocorreu em 3 laços: os dois laços da função `initialize`, sendo que o último deles possui três *loops* aninhados, podendo ser colapsado, e o laço aninhado da função `iso_3dfd_it`, que também pôde ser colapsado. Nesse último laço, foram aplicados os tipos de escalonamento testados. Foram utilizadas 4 *threads*

A Tabela 4 apresenta os tempos de execução, obtidos com o *time* para cada tipo de escalonamento e colapso de laços.

Compilador	*Collapse*	*Escalonamento*	*Tempo de Execução*
gcc	1	Dinâmico	4m45.765s
gcc	1	Guiado	4m52.213s
gcc	1	Estático	6m28.242s
gcc	2	Dinâmico	4m43.207s
gcc	2	Guiado	4m48.780s
gcc	2	Estático	6m22.979s
gcc	3	Dinâmico	6m25.470s
gcc	3	Guiado	4m55.489s
gcc	3	Estático	6m34.805s
icc	1	Dinâmico	1m30.755s
icc	1	Guiado	1m32.441s
icc	1	Estático	1m23.029s
icc	2	Dinâmico	1m31.572s
icc	2	Guiado	1m33.349s
icc	2	Estático	1m35.497s
icc	3	Dinâmico	3m10.883s
icc	3	Guiado	1m48.547s
icc	3	Estático	1m49.963s

Tabela 4: Tempos reais de execução para o WAVE paralelo: teste de escalonamentos e colapso de laços.

Vemos que há uma diferença nos resultados a depender do compilador. Para o gcc, o melhor resultado foi colapsando 2 laços e usando escalonamento dinâmico, com um tempo de 4 minutos e 43.207 segundos. Já para o icc, o melhor resultado foi obtido usando-se escalonamento estático sem colapsar os laços, com um

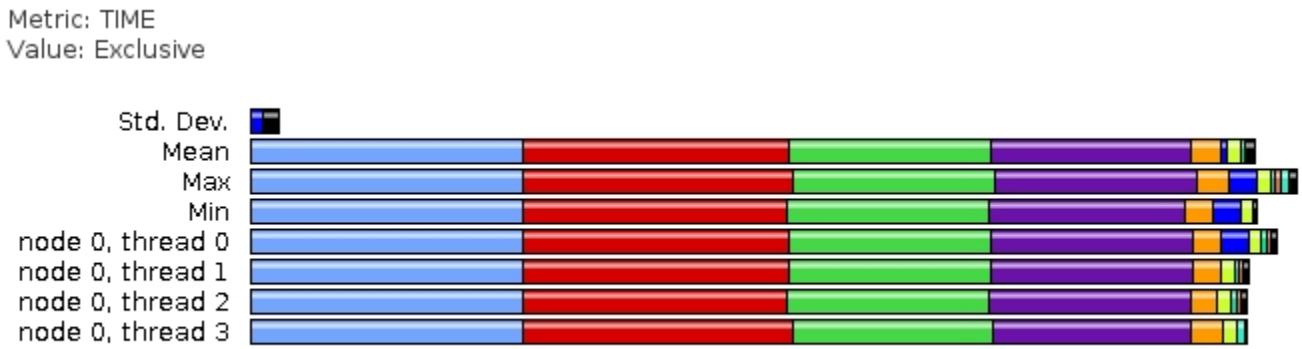

Figura 3: Gráfico bidimensional obtido pelo TAU. Cada cor corresponde a uma instrução, e a instrução mais à esquerda foi a que causou maior gasto de tempo de CPU.

tempo de 1 minuto e 23.029 segundos. Essa diferença provavelmente vem de divergências nas implementações do padrão OpenMP em cada compilador.

A melhor configuração obtida, então, foi usando-se escalonamento estático, sem colapso de laços e com o compilador icc.

Devido a problemas com o *parser* do TAU, para realizar a perfilagem foi utilizado o comando `tau_cxx.sh`, especificando o icc como compilador. Além disso, foi necessário comentar as linhas usadas para gerar os arquivos .plot. O comando final foi:

```
tau_cxx.sh -optTauCC="/home/intel/compilers_and_libraries/linux/bin/intel64/icc" -fopenmp -O3
main.cc -o wave.exe
```

As Figuras 4 e 5 apresentam os gráficos gerados pelo `paraprof`.

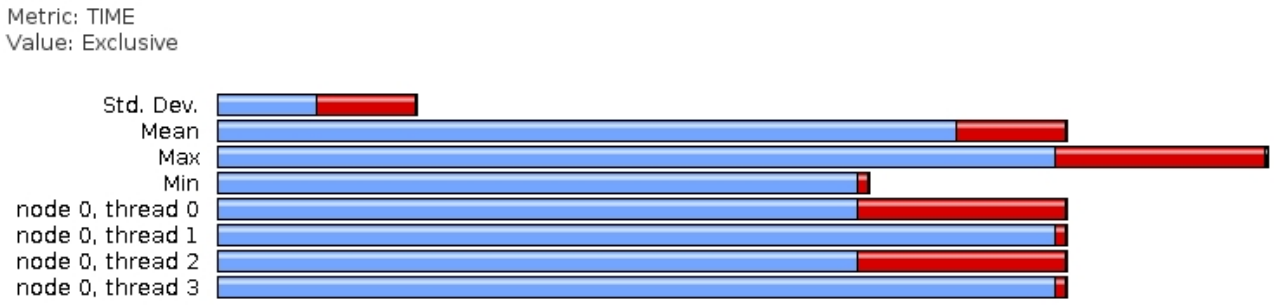

Figura 4: Gráfico bidimensional obtido pelo TAU para o WAVE. Cada cor corresponde a uma instrução, e a instrução mais à esquerda foi a que causou maior gasto de tempo de CPU.

Os gráficos apresentam um bom balanceamento de carga de trabalho entre as *threads*, de modo que todas usam aproximadamente o mesmo tempo de CPU. É possível notar uma diferença no tempo de CPU entre os pares de *threads* (0, 2) e (1, 3). No gráfico 2D, as barras azuis correspondem ao tempo de CPU gasto no laço entre as linhas 203 e 225, correspondente à função `iso_3dfd_it`, e as barras vermelhas correspondem

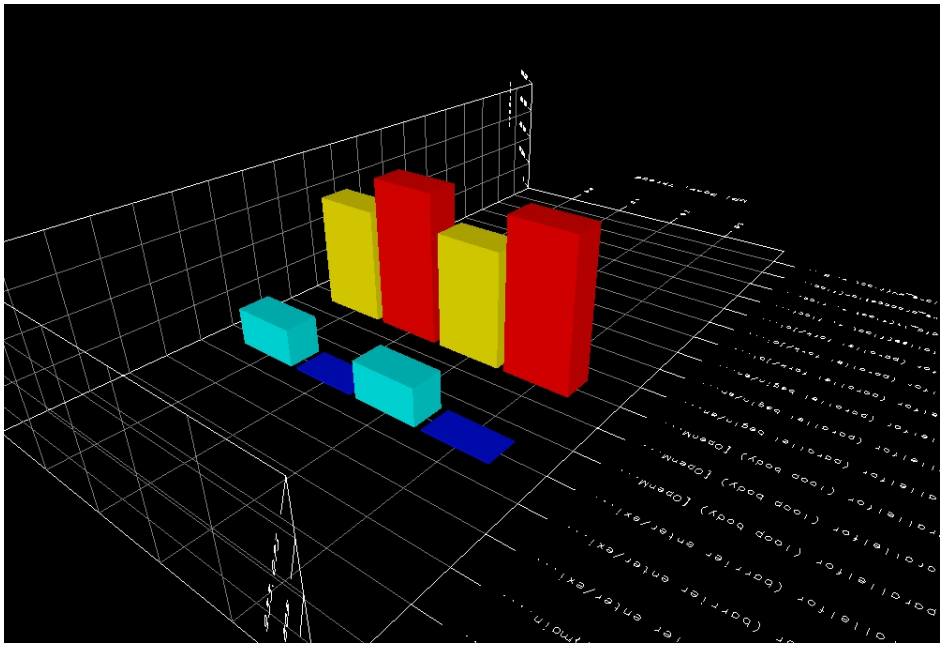

Figura 5: Gráfico tridimensional obtido pelo TAU para o WAVE.

ao tempo de sincronização (*barrier*) e saída do laço. No gráfico 3D, a operação no laço é representada pelas barras vermelhas e amarelas, e a sincronização e a saída são representadas pelas barras azuis. Isso mostra que as *threads* 0 e 2 gastam mais tempo de CPU aguardando a sincronização que as *threads* 1 e 3, pois executam suas partes do laço mais rápido, tendo que aguardar as duas últimas finalizarem a execução.

A Figura 6 indica o *speedup* dessa configuração com o número de *threads* variando entre 2 e 20. É possível constatar que o melhor *speedup* é obtido quando são utilizadas 4 *threads*, que é o número de núcleos do processador, o que portanto define quantas *threads* podem executar simultaneamente. O desempenho é melhor nesse caso porque há menos trocas de contexto entre as *threads*. Acima de 4 *threads*, ocorre uma pequena queda no desempenho, mas com um maior aumento, o desempenho aumenta novamente, aproximando-se do melhor caso.

4 Exercício 3: Cálculo do π usando Monte Carlo

4.1 Detalhes da implementação

O código foi escrito em C++, usando bibliotecas definidas no padrão C++11. Para gerar números aleatórios, a biblioteca `<random>`, que possui implementações de geradores de números pseudo-aleatórios e distribuições de probabilidade. Em particular, foram utilizadas as classes `std::mt19937`, que implementa o gerador *Mersenne Twister* com período de $2^{19937} - 1$, e a classe `std::uniform_real_distribution`, que implementa uma distribuição uniforme e foi usada como mapeamento dos resultados do gerador para valores no interior do quadrado. Para as sementes do gerador, foi usada a biblioteca `<chrono>` para obter o *unix time* em milissegundos, que foi multiplicado pelo número da *thread* e dividido pelo total de *threads* em execução. Para representar o vetor aleatório, foi usado o *container* `std::vector`

O OpenMP cria *threads* de maneira dinâmica, não necessariamente gerando o número de *threads* requisitado. Para forçar os números de *threads* usados, foi usada a rotina `omp_set_dynamic`, com argumento 0 para

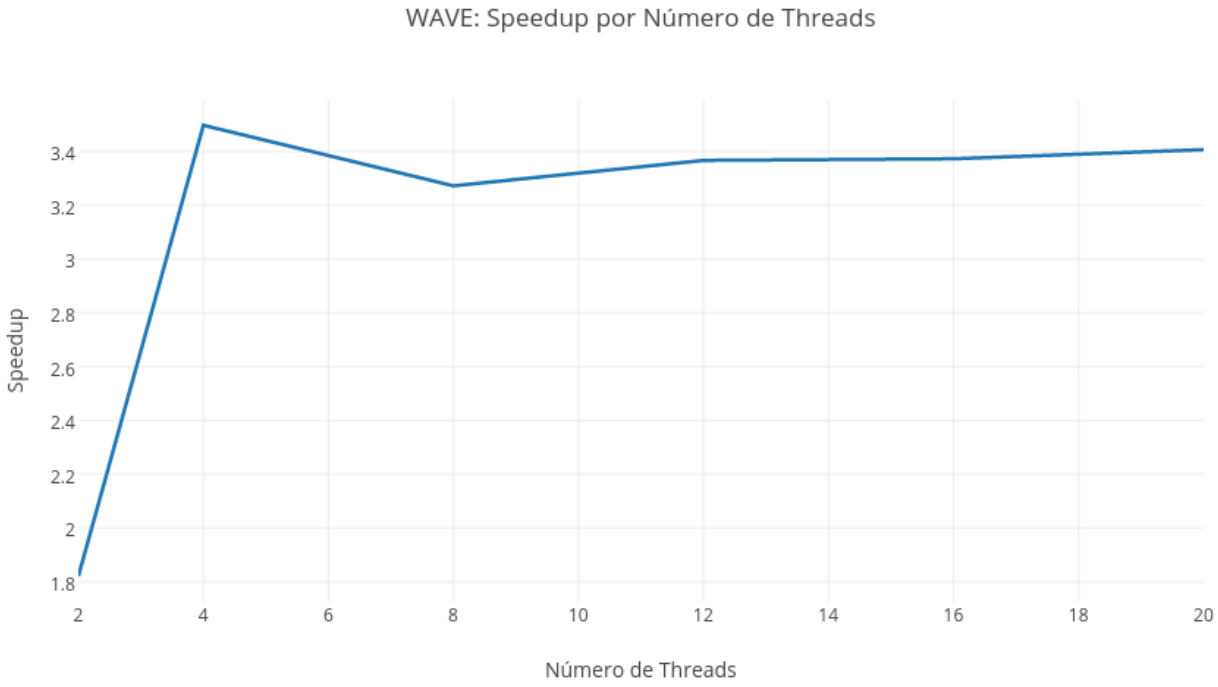

Figura 6: Gráfico indicando o speedup por número de *threads* no WAVE paralelizado.

desativar esse recurso.

4.2 Resultados

O programa foi executado em dois testes: no primeiro, o número de *threads* foi variado de 1 a 52, com o vetor assumindo o tamanho fixo de 10^7, e no segundo, o tamanho do vetor assumiu valores entre 10^2 e 10^7, com 4 *threads*. Em ambos os testes, o código foi executado 1000 vezes. Os gráficos das Figuras 7 e 8 abaixo apresentam as médias dos valores de π obtidos em cada caso. A linha laranja nos dois gráficos corresponde ao valor de π original, obtido em um site da Universidade de Illinois (`www.geom.uiuc.edu/~huberty/math5337/groupe/digits.html`) e truncado para 22 casas decimais, resultando em 3.141592653589793238462 6.

Os gráficos mostram que, tanto aumentando o número de *threads* quanto aumentando o tamanho do vetor, o resultado obtido se aproxima do valor original. No caso do tamanho do vetor variável, esse resultado pode ser explicado pelo consequente aumento do espaço amostral.

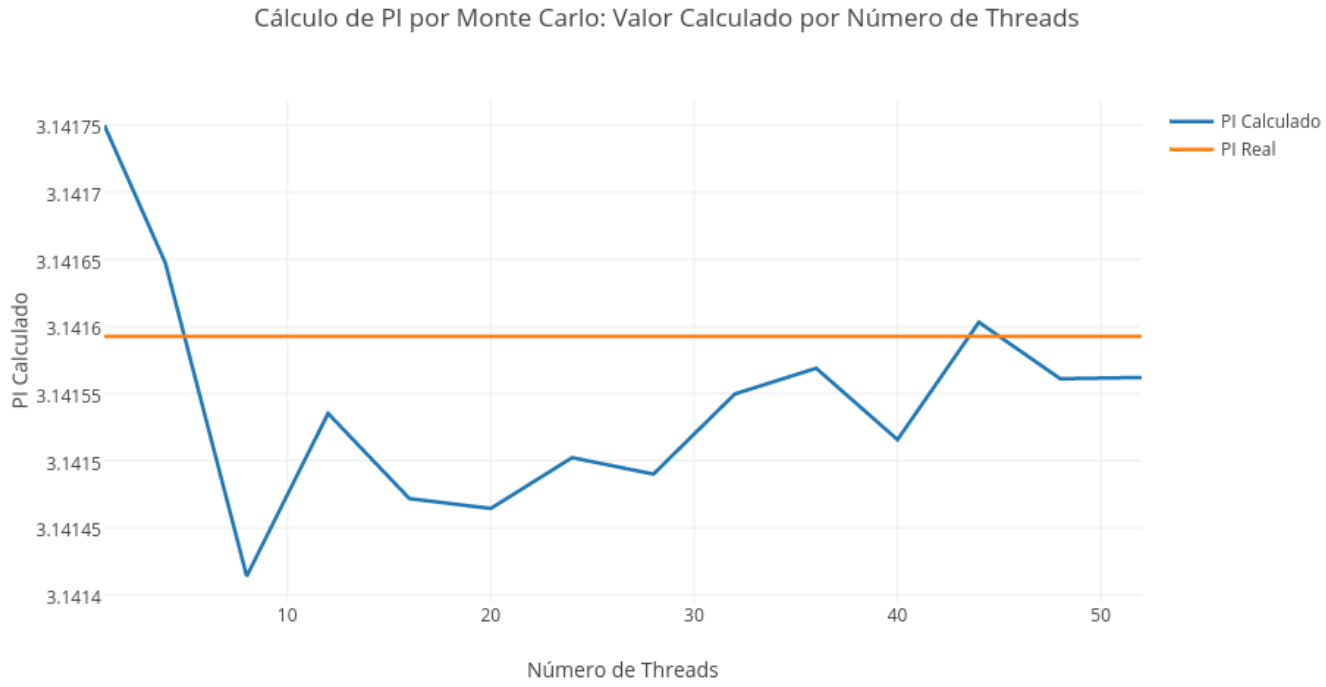

Figura 7: Gráfico indicando os valores de PI obtidos por número de *threads*.

5 Exercício 4: QuickSort

5.1 Detalhes da Implementação

Novamente, o código foi implementado usando-se C++11. Novamente, foi usado o gerador de números pseudo-aleatórios *Mersenne Twister* e o *container* `std::vector`. O vetor a ser ordenado foi gerado paralelamente, com 4 *threads* (forçado por meio da função `omp_set_num_threads`), para economizar tempo na execução dos testes. Aqui também foi desativado o gerenciamento dinâmico de *threads* do OpenMP. Para o *QuickSort*, cada chamada recursiva ao algoritmo é uma seção paralela.

5.2 Resultados

A Tabela 5 apresenta os tempos de execução e o *speedup* para cada versão do código.

Número de Threads	*Valor de k*	*Tempo de Execução*	*Speedup*
1	20	157.191515 ms	1
1	21	331.90503 ms	1
2	20	252.76196 ms	0.621895458
2	21	490.438355 ms	0.676751781
4	20	258.507405 ms	0.608073548
4	21	508.99794 ms	0.65207539
8	20	222.530105 ms	0.706383143
8	21	447.658015 ms	0.741425416

Tabela 5: Tempos de execução do para o *QuickSort* e *speedup* em relação ao código sequencial.

Podemos ver que a implementação paralela utilizada foi mais lenta que a serial em todos os casos. Uma provável causa para isso é o aumento de erros de *cache*. Uma vez que cada *thread* manipula uma seção diferente do vetor, a taxa de erros da *cache* compartilhada (L3) provavelmente aumenta, a depender da quantidade de números recebidos pela *cache* em uma consulta à memória. Como o acesso à memória é mais lento que o acesso à *cache*, isso pode explicar o aumento no tempo de execução.

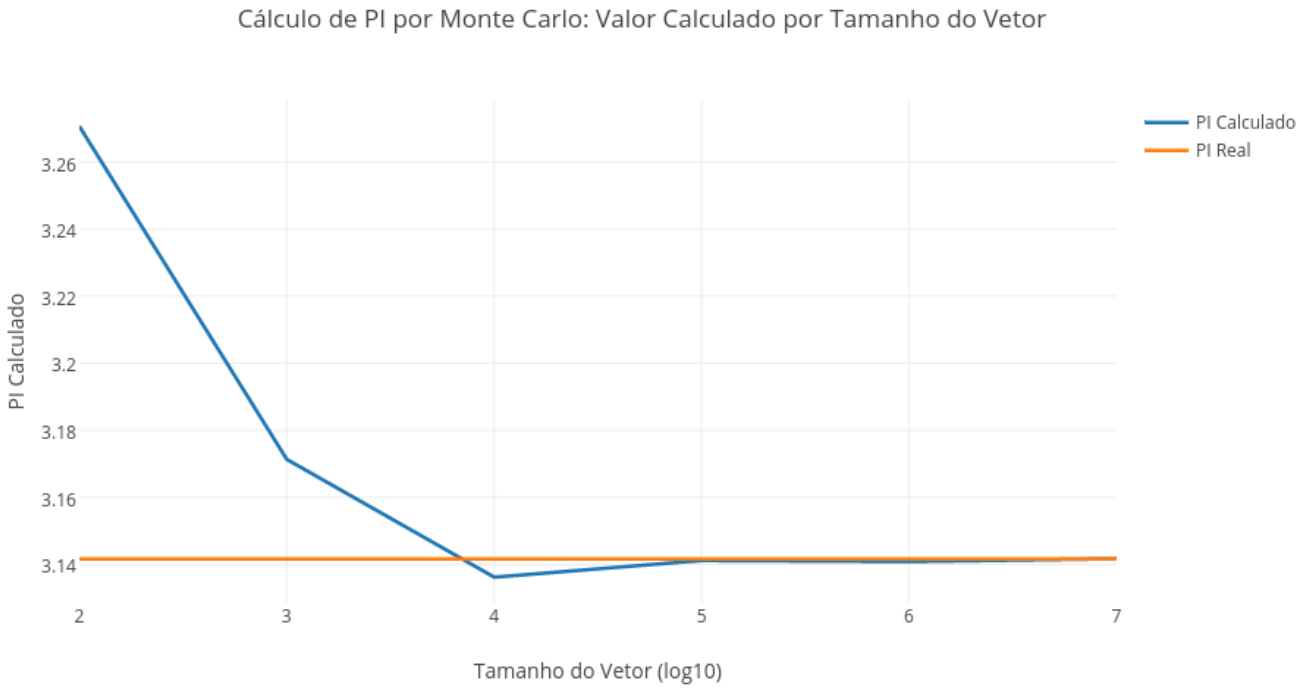

Figura 8: Gráfico indicando os valores de PI obtidos por tamanho do vetor.