Trabalho 2 de Sistemas Operacionais
Relatório de Execução

Letícia Souza de Souza, Marcos Paulo Vieira Pedrosa, Luiz Gabriel Antunes Sena, Paulo Victor Fernandes de Melo

Instituto de Computação (IComp) – Universidade Federal do Amazonas (UFAM)
Av. Gen. Rodrigo Octávio, 6200, Coroado I, Setor Norte
69080-900 – Manaus – AM – Brasil

leticia.souza@icomp.ufam.edu.br, marcos.pedrosa@icomp.ufam.edu.br,
luiz.sena@icomp.ufam.edu.br, paulo.fernandes@icomp.ufam.edu.br

1. Hierarquia do projeto

Abaixo está uma descrição dos diretórios e arquivos encontrados na raíz do projeto.

`include/`	Contém os arquivos de cabeçalho C dos programas;
`scripts/`	Contém os scripts *Python* que geram os gráficos e tabelas de execução;
`src/`	Contém os arquivos de código em C dos programas;
`Dockerfile`	Contém as instruções de criação da imagem *Docker*;
`docker-compose.yml`	Contém as instruções de execução dos serviços com *Docker*;
`CMakeLists.txt`	Contém as instruções de geração de configuração do *CMake*.

2. Implementações e entrada

Para o trabalho, foi desenvolvido 1 programa em C para cada parte solicitada, totalizando 3 programas completos. Cada programa recebe os mesmos padrões de entrada na linha de comando, simplificando o uso das implementações. Abaixo estão os formatos das linhas de comando para cada um dos tipos de execução.

- **Execução sequencial:** `programa --size TAMANHO --mode seq`
- **Execução paralela:** `programa --size TAMANHO --mode par --threads NUM_THREADS`

Para todos os programas, `NUM_THREADS` dita o número de threads a serem criadas em caso de execução paralela. `TAMANHO` pode ditar diferentes coisas dependendo do programa executado. Um parâmetro `--seed SEMENTE` ainda pode ser adicionado ao final da linha de execução, com um inteiro no lugar de `SEMENTE`, para tornar a geração de números aleatórios durante o programa reprodutível.

3. Compilação

Os programas podem ser compilados utilizando o CMake e o Make para produzir os binários finais do programa, ou utilizando o Docker para executar os testes e produzir apenas as tabelas e gráficos.

3.1. Usando o Docker

Para gerar os gráficos e tabelas usando o Docker, instale o Docker e o Docker Compose na máquina. Após isso, basta executar o comando

```
sudo docker compose up --build
```

no diretório raiz do projeto e aguardar o fim da execução. Os gráficos e tabelas estarão disponíveis no diretório `out/` relativo à raiz do projeto.

3.1.1. Extra: Executando os programas pelo Docker

É possível também executar os programas compilados usando o Docker através do seguinte prefixo, **substituindo `prog` pelo nome do programa**.

```
docker compose run --remove-orphans --build benchmark build/prog
```

Para executar o simulador, porém, é necessário rodar o seguinte comando para permitir o acesso ao servidor X11 do Linux pelo Docker.

```
xhost +local:docker
```

3.2. Usando CMake

Para compilar o projeto manualmente, instale o CMake na máquina, junto da biblioteca SDL3, necessária para a execução do simulador da Parte 3. Após isso, gere as configurações do projeto usando os seguintes comandos no diretório raiz do projeto.

```
mkdir build
cd build
cmake -G "Unix Makefiles" ..
```

Após gerar as configurações, execute

```
make
```

para compilar os programas. Eles estarão disponíveis no diretório `build/` relativo à raiz do projeto.

4. Ambientes

Os programas implementados foram executados em dois computadores diferentes, com especificações diferentes. O primeiro computador possui componentes mais antigos, com menos núcleos no processador. O segundo computador já é mais atual, com especificações melhores. Abaixo estão os processadores de cada um dos computadores.

- **Computador 1:** Intel® Core™ i5-7300HQ com 4 núcleos e 4 threads;
- **Computador 2:** Intel® Core™ i7-1255U com 10 núcleos e 12 threads.

5. Produto escalar (programa `scalar_product`)

O programa `scalar_product` foi desenvolvido para realizar o produto escalar entre dois vetores aleatórios de mesmo tamanho. O tamanho dos vetores é dado pelo argumento `--size TAMANHO`, onde `TAMANHO` é um inteiro que dita a quantidade de elementos nos vetores.

Para dividir a execução do produto escalar em threads, nossa implementação separa cada vetor em n subvetores, chamados no código de *segmentos*, onde n é o número

de threads a serem executadas. As threads calculam o produto escalar entre esses subvetores, cada uma calculando um resultado parcial. Os resultados parciais de todas as threads são coletadas e, então, somados para gerar um resultado final.

5.1. Execução

Executamos o `scalar_product` várias vezes variando diversos parâmetros, como o modo de execução (sequencial ou paralelo), número de threads usadas e tamanho do vetor. Cada combinação de parâmetros foi executada 3 vezes, e o tempo de execução médio das 3 execuções foi salvo.

Na Figura 1 e Figura 2, observamos que o uso de threads para vetores muito pequenos não deu um bom resultado quando comparado à execução sequencial, e isso se manteve para alguns dos outros testes também. Isso ocorre devido ao *overhead* de dividir os vetores, criar as *threads* e aguardar o fim de suas execuções, que no total demora mais tempo do que apenas calcular o resultado sequencialmente. Porém, para vetores maiores, ambos os gráficos mostram que há uma diferença no tempo de execução quando se usam mais *threads*.

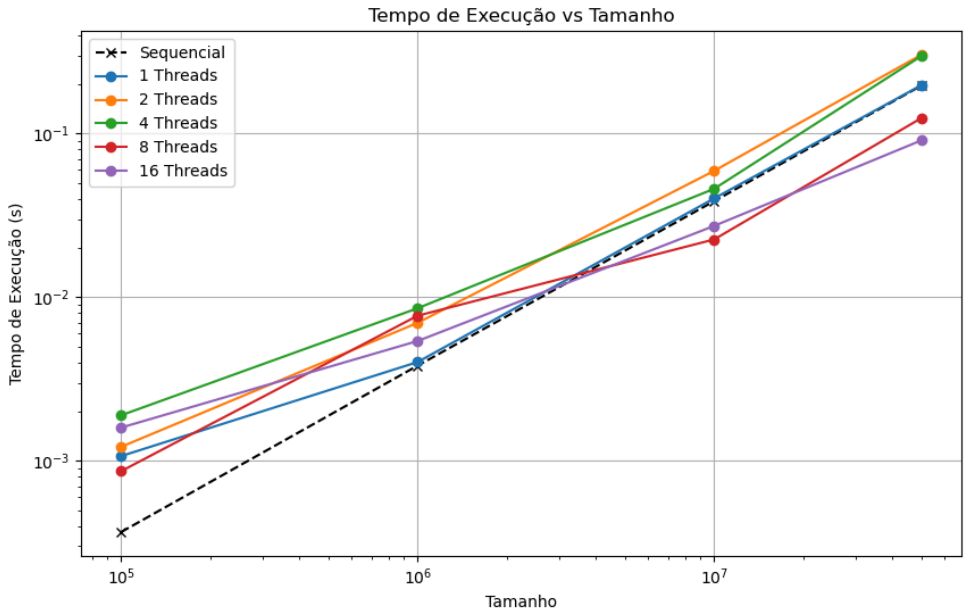

Figura 1. Gráfico de duração vs. tamanho dos vetores por número de threads no computador 1.

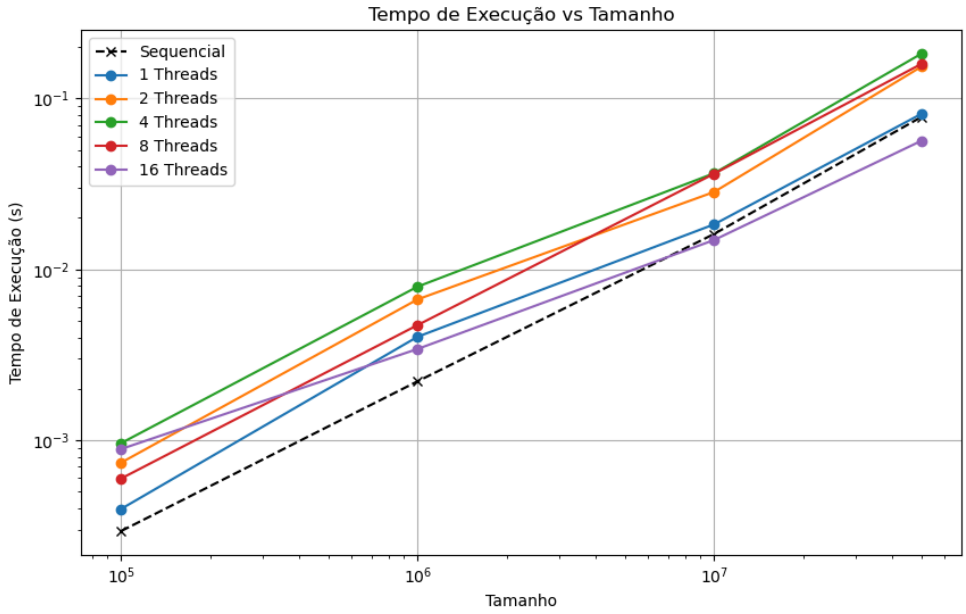

Figura 2. Gráfico de duração vs. tamanho dos vetores por número de threads no computador 2.

Na Figura 3 e Figura 4, observamos a *aceleração* medida pelos experimentos acima. Perceba que ambos os gráficos mostram uma aceleração quando muitas *threads* são usadas, porém, quando menos *threads* são usadas, a *aceleração* fica abaixo de 1, mostrando uma diminuição na performance em comparação com a execução sequencial.

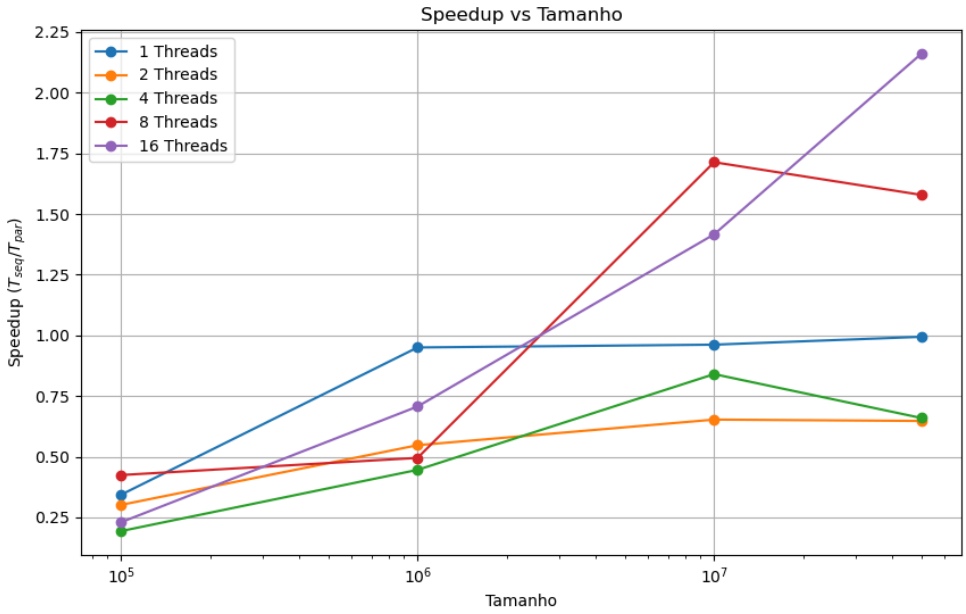

Figura 3. Gráfico de aceleração vs. tamanho dos vetores por número threads no computador 1.

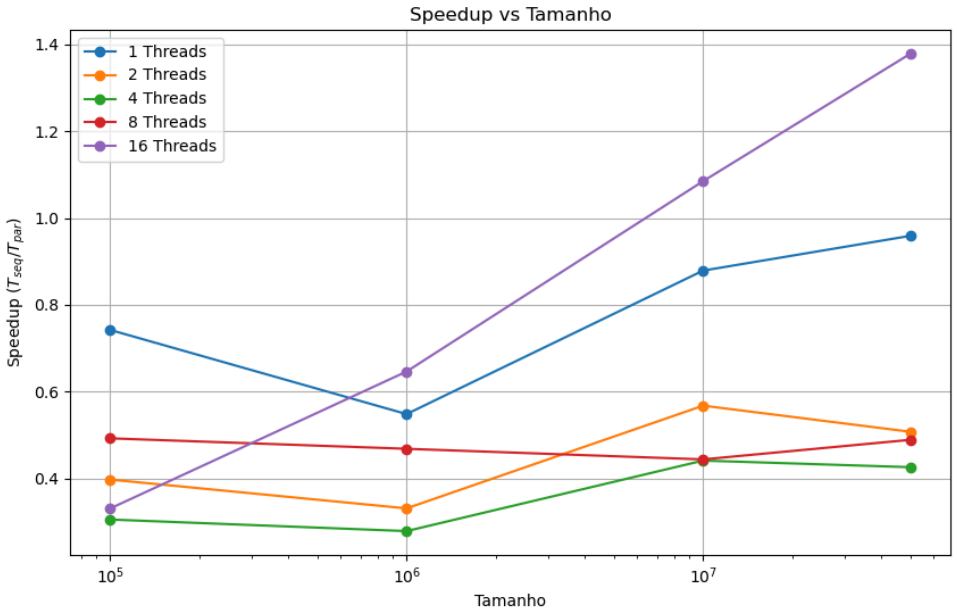

Figura 4. Gráfico de aceleração vs. tamanho dos vetores por número threads no computador 2.

Na Tabela 1, observamos os tempos de execução do produto escalar para cada combinação de parâmetros em ambos os computadores. Realizando alguns testes, percebemos que as vezes, apesar de possuir uma especificação pior, o computador 1 ganhava do computador 2, e outras vezes o oposto ocorria. Chegamos à conclusão que como o tempo de execução dos testes é tão pouco, pequenas variações no ambiente de execução podem mudar completamente os resultados, explicando essas variações.

Tabela 1. Tabela de execução do `scalar_product` nos computadores 1 e 2.

Modo	Tamanho	Threads	Duração (s) (Comp. 1)	Duração (s) (Comp. 2)
Seq.	100000	1	0.000366	0.000294
Seq.	1000000	1	0.003808	0.002212
Seq.	10000000	1	0.038545	0.016115
Seq.	50000000	1	0.195907	0.077940
Par.	100000	1	0.001067	0.000395
Par.	100000	2	0.001215	0.000739
Par.	100000	4	0.001894	0.000961
Par.	100000	8	0.000863	0.000596
Par.	100000	16	0.001593	0.000888
Par.	1000000	1	0.004007	0.004032
Par.	1000000	2	0.006957	0.006673
Par.	1000000	4	0.008542	0.007924

Par.	1000000	8	0.007690	0.004719
Par.	1000000	16	0.005387	0.003421
Par.	10000000	1	0.040079	0.018335
Par.	10000000	2	0.059032	0.028373
Par.	10000000	4	0.045867	0.036500
Par.	10000000	8	0.022492	0.036275
Par.	10000000	16	0.027216	0.014855
Par.	50000000	1	0.19707	0.081276
Par.	50000000	2	0.302606	0.153483
Par.	50000000	4	0.297035	0.182829
Par.	50000000	8	0.124077	0.159284
Par.	50000000	16	0.090667	0.056550

6. Produto matricial (programa `matrix_product`)

O programa `matrix_product` foi desenvolvido para realizar o produto entre duas matrizes quadradas aleatórias de mesmo tamanho. O tamanho das matrizes é dado pelo argumento `--size TAMANHO`, onde `TAMANHO` é um inteiro que dita a quantidade de linhas e colunas de cada matriz.

Para dividir a execução do produto matricial em threads, decidimos fragmentar o cálculo da matriz final em *subconjuntos de linhas*, ou seja, cada thread é responsável por calcular n linhas da matriz final. Desta forma, a primeira thread calcula as n primeiras linhas, a segunda thread calculas as próximas n, assim por diante. Desta forma, não há concorrência visto que as matrizes não modificam os mesmos endereços de memória.

6.1. Execução

Na Figura 5 e Figura 6 observamos que os benefícios no uso de *threads* beneficiam bem mais o computador 2, que possui mais *threads* que o computador 1. Percebemos também que o mesmo *overhead* visto na Parte 1 também é observável aqui, numa matriz 10x10, onde o tempo de manuseamento das *threads* é maior que o cálculo sequencial. Porém, por todo o resto dos experimentos, os benefícios do uso de *threads* são mais claros, e o tempo de execução sequencial fica próximo o tempo de execução "paralela" com apenas 1 *thread*, o que faz bastante sentido considerando que a escala do tempo de execução é maior.

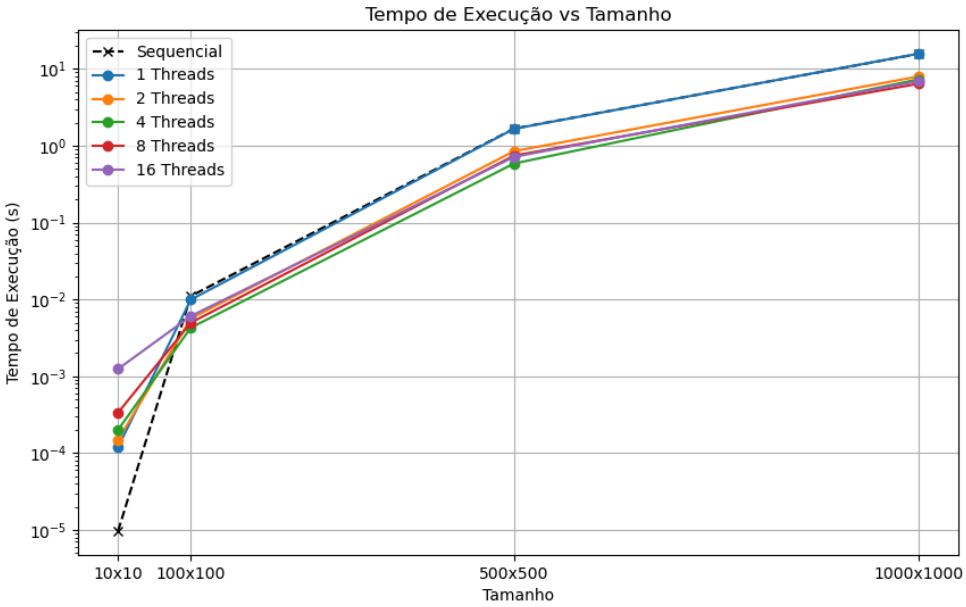

Figura 5. Gráfico de duração vs. tamanho das matrizes por número de threads no computador 1.

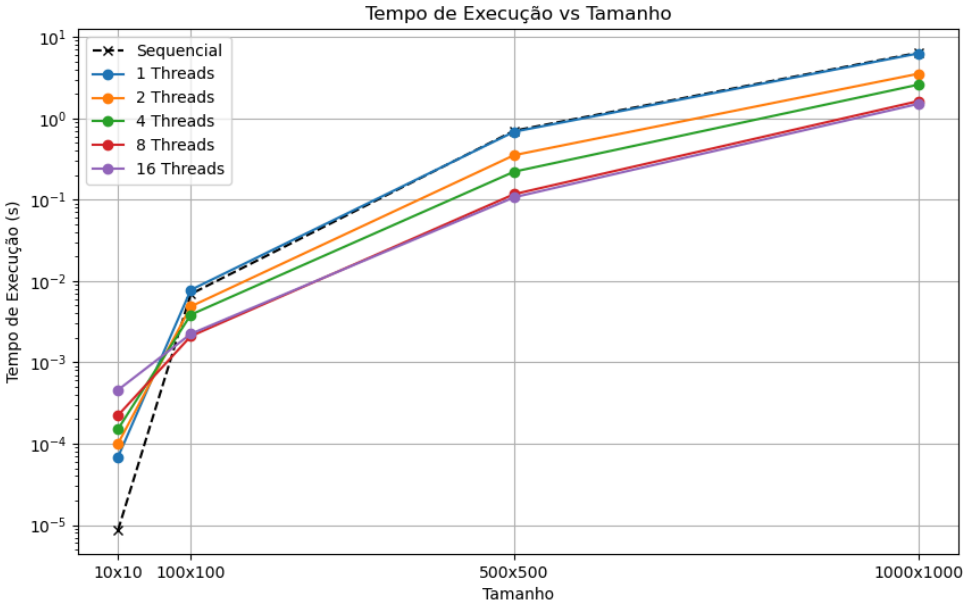

Figura 6. Gráfico de duração vs. tamanho das matrizes por número de threads no computador 2.

Na Figura 7 e Figura 8, observamos mais claramente a aceleração do uso de *threads* na multiplicação de matrizes, que diminuiu bastante o tempo de execução. No computador 2, houve uma aceleração de 6x ao usar 8 *threads*, enquanto o uso de 4 a 16 *threads* no computador 1 foram bem próximos. Isso se principalmente deve ao fato do computador 1

possuir poucas *threads*, assim como o fato de não possuir a tecnologia de *hyperthreading* presente no computador 2.

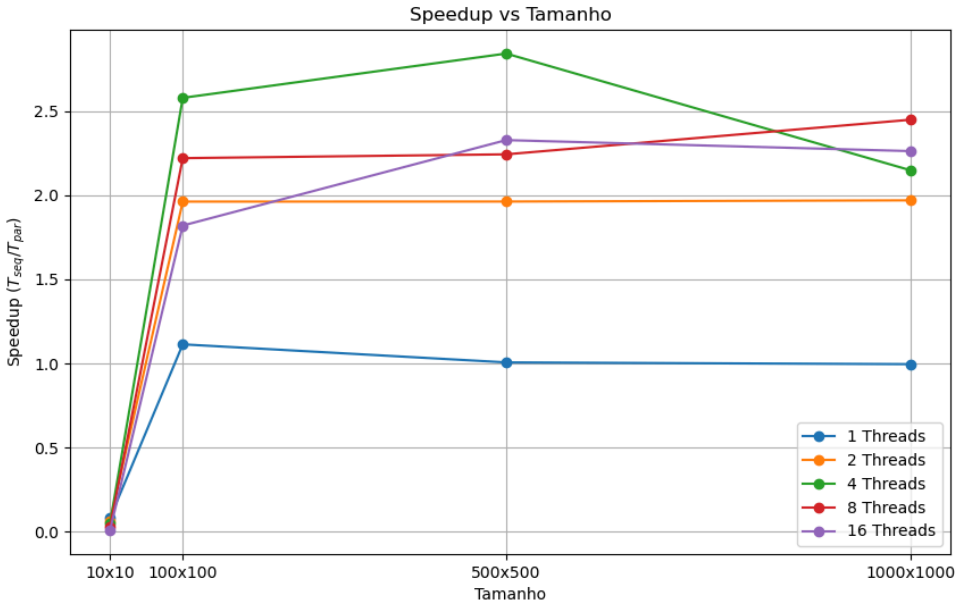

Figura 7. Gráfico de aceleração vs. tamanho das matrizes por número de threads no computador 1.

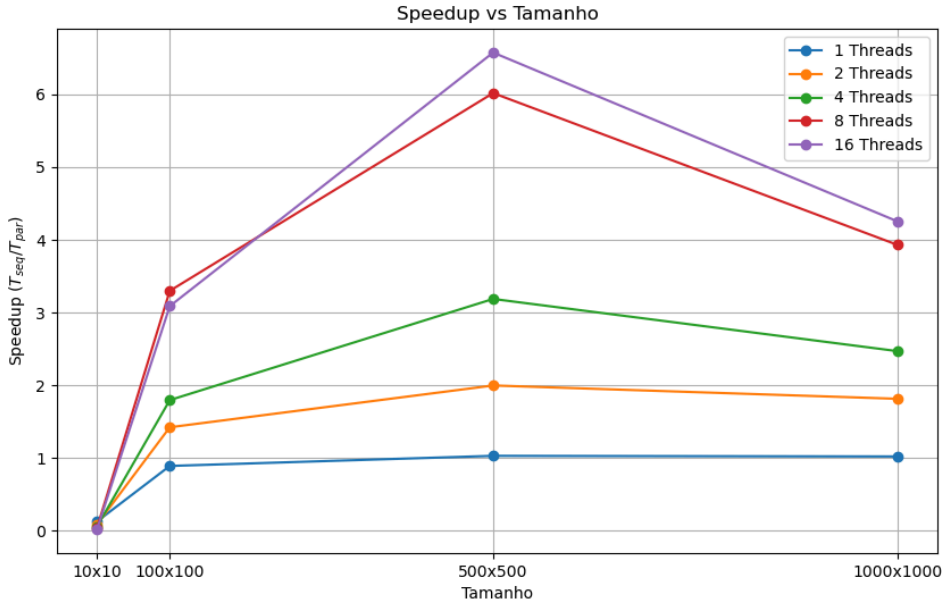

Figura 8. Gráfico de aceleração vs. tamanho das matrizes por número de threads no computador 2.

Na Tabela 2, observamos a grande diferença entre os tempos de execução do `matrix_product` no computador 1 e 2. No pior caso, o computador 1 demorou 15 segundos enquanto o computador 2 demorou apenas 6 segundos.

Tabela 2. Tabela de execução do `matrix_product` nos computadores 1 e 2.

Modo	Tamanho	Threads	Duração (s) (Comp. 1)	Duração (s) (Comp. 2)
Seq.	10	1	0.000009	0.000008
Seq.	100	1	0.011061	0.006915
Seq.	500	1	1.674091	0.703917
Seq.	1000	1	15.71786	6.405719
Par.	10	1	0.000119	0.000068
Par.	10	2	0.000148	0.000100
Par.	10	4	0.000200	0.000151
Par.	10	8	0.000336	0.000223
Par.	10	16	0.001244	0.000457
Par.	100	1	0.009932	0.007772
Par.	100	2	0.005635	0.004864
Par.	100	4	0.004287	0.003851
Par.	100	8	0.004979	0.002096
Par.	100	16	0.006076	0.002239
Par.	500	1	1.663615	0.684164
Par.	500	2	0.852799	0.35256
Par.	500	4	0.588767	0.22087
Par.	500	8	0.746049	0.116954
Par.	500	16	0.718946	0.107038
Par.	1000	1	15.78478	6.27796
Par.	1000	2	7.977884	3.532208
Par.	1000	4	7.313053	2.594120
Par.	1000	8	6.416334	1.629528
Par.	1000	16	6.945187	1.505668

7. Simulador (programa `simulation`)

Para a Parte 3 do trabalho, resolvemos desenvolver um simulador gráfico **renderizado na CPU**. O programa `simulation` desenvolvido é uma demonstração de uma simulação de várias bolinhas 2D coloridas pulando pela tela e colidindo entre si, porém a parte pesada da execução é a renderização via software. A cor de cada píxel da janela é calculado manualmente de acordo com o estado atual da simulação física, e o objetivo da simulação é demonstrar que a renderização da janela em paralelo usando *threads* melhora drasticamente a performance do programa. Para isso, utilizamos o **FPS** (quadros por segundo)

como medida de comparação, com cada teste executando por 5 segundos. O parâmetro `TAMANHO` na linha de comando determina a resolução da janela renderizada.

Similarmente à solução do produto de matrizes, a tela é dividida em n seções horizontais, onde cada *thread* é responsável por renderizar uma dessas seções. Para sinalizar que a tela toda foi renderizada, usamos uma barreira (implementada pelo `pthread_barrier_t`), que interrompe a execução de cada *thread* que a esperar até que $n + 1$ *threads* executem `pthread_barrier_wait(...)` nesta barreira. Não estudamos esta ferramenta durante as aulas de SO, porém descobrimos ela através do manual da bibioteca `pthread`[1].

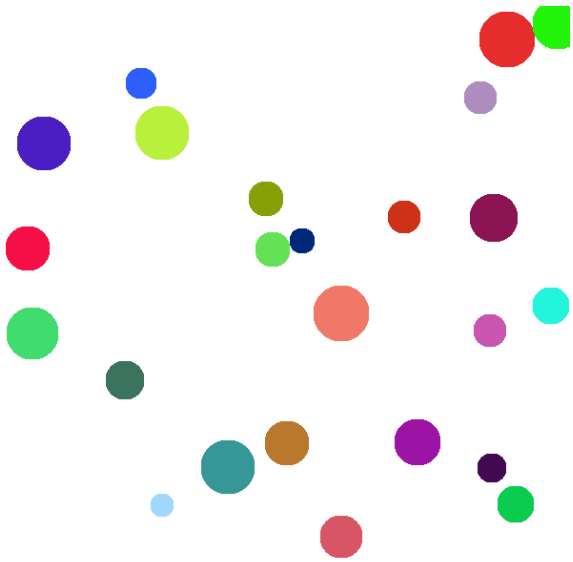

Figura 9. O simulador em execução no computador 1.

7.1. Execução

Na Figura 10 e Figura 11, percebemos novamente o benefício do uso de *threads* em todos os casos, visto que a resolução mínima usada foi 400x400, porém, o uso de mais *threads* beneficia principalmente o computador 2 como discutido préviamente.

Percebemos na Figura 10 que o computador 1 tem uma diferença maior entre a execução sequencial e a execução "paralela" com 1 *thread*. Assumimos que, neste caso, o sistema operacional esteja colocando menos processos para executar no núcleo onde a nova *thread* está em execução, causando menos trocas de contexto e acelerando a execução quando comparado com a execução sequencial. Isto provavelmente não ocorreu no computador 2 pois **menos programas estavam em execução** nele quando os experimentos foram realizados, diminuindo a quantidade de trocas de contexto no total.

[1] https://pubs.opengroup.org/onlinepubs/009696899/functions/pthread_barrier_wait.html

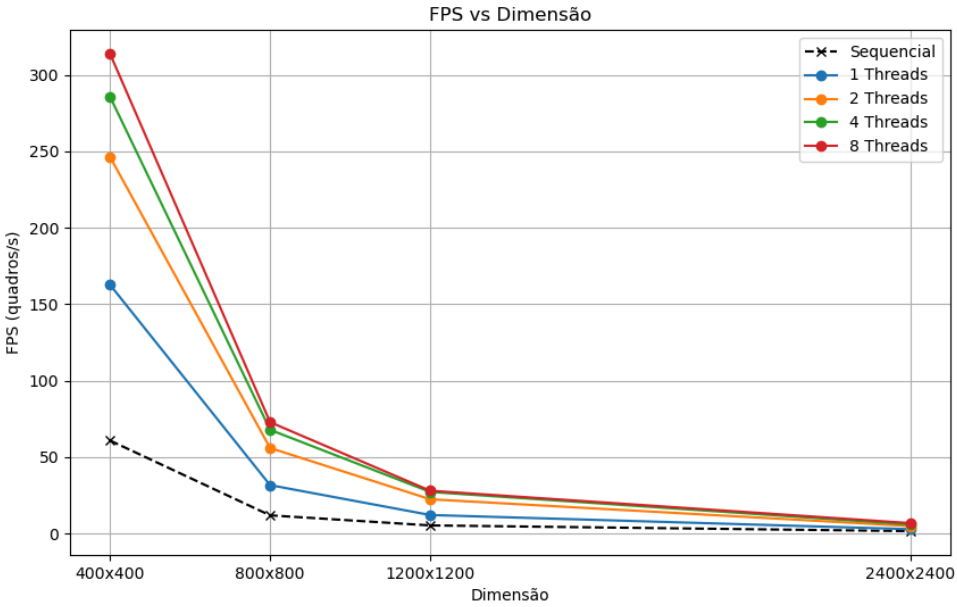

Figura 10. Gráfico de quadros por segundo vs. número de bolinhas por número de threads no computador 1.

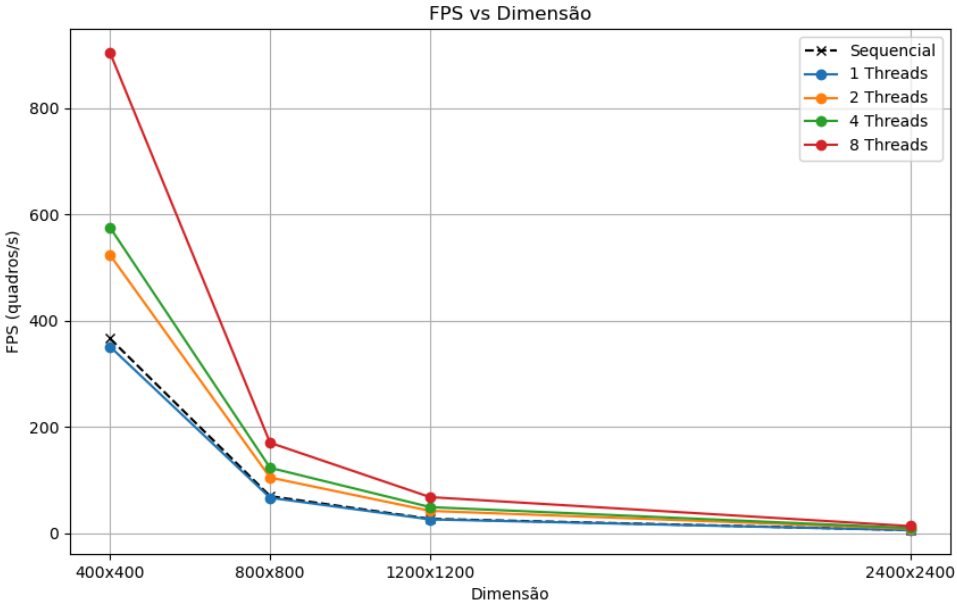

Figura 11. Gráfico de quadros por segundo vs. número de bolinhas por número de threads no computador 2.

Na Tabela 3, vemos a enorme diferença no **FPS** dos experimentos do computador 1 vs o computador 2, apesar da renderização via *software*. Percebe-se que diferentes fatores, como velocidade de *clock* e cache fazem uma grande diferença nos tempos de execução, mesmo quando a renderização é feita sequencialmente.

Tabela 3. Tabela de execução do `simulation` nos computadores 1 e 2.

Modo	Tamanho	Threads	FPS (Comp. 1)	FPS (Comp. 2)
Seq.	400	1	60.80170	366.4571
Seq.	800	1	11.85133	69.56315
Seq.	1200	1	5.259592	26.83759
Seq.	2400	1	1.535324	5.712808
Par.	400	1	162.7224	351.3087
Par.	400	2	246.3504	523.8721
Par.	400	4	285.7994	575.7229
Par.	400	8	313.8896	904.9141
Par.	800	1	31.61760	66.58581
Par.	800	2	55.92354	104.7159
Par.	800	4	67.81418	122.9963
Par.	800	8	72.80881	170.1680
Par.	1200	1	12.07161	25.78599
Par.	1200	2	22.36683	41.82994
Par.	1200	4	27.16609	48.94116
Par.	1200	8	27.96020	67.74737
Par.	2400	1	2.773623	5.62698
Par.	2400	2	4.838025	8.267345
Par.	2400	4	5.870032	9.511889
Par.	2400	8	6.754936	13.18011