Avaliação de Performance

Relatório Trabalho 1

Computação Paralela

Mestrado Integrado em Engenharia Informática e Computação

Elaborado por:
Pedro Filipe Agrela Faria - 201406992

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

13 de Março de 2018

Conteúdo

1- Descrição do problema e explicação dos algoritmos

Em resposta ao repto lançado pelo docente da unidade curricular de Computação Paralela, desenvolvi pequenas aplicações tendo como objetivo o estudo do desempenho de processadores relativamente à hierarquia de memória quando deparado com acesso a grandes volumes de dados.

Para resolver e analisar o problema, foi sugerido pelo docente a multiplicação de duas matrizes usando dois algoritmos diferentes, utilização de duas linguagens de programação diferentes, duas bibliotecas para leitura da performance (PAPI e Tiptop) e a utilização da biblioteca OpenMP com a finalidade de usar paralelismo nos dois algoritmos.

Algoritmo 1 - MulLineCol

Trata-se da tradicional multiplicação de matrizes. Considerando duas matrizes, A e B, no qual é feita a multiplicação consecutiva da linha (matriz A) e coluna (matriz B). A complexidade temporal é de $O(N^3)$ e o número de instruções aritméticas de vírgula flutuante (FLOP) é de $2N^3$. Como é usada uma soma e uma multiplicação, requer assim duas instruções, assim como 3 loops (1 loop para cada for). O seguinte excerto de código representa o algoritmo aplicado:

```
for(i = 0; i < n; i++){
  for(j = 0; j < n; j++){
    for(k = 0; k < n; k++){
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

Figura 1: Excerto de código do algoritmo 1.

Algoritmo 2 - MulEleCol

A multiplicação neste algoritmo é feita de um modo diferente, em vez de usar o método tradicional, foi realizada uma abordagem no qual a multiplicação é feita com um elemento da matriz A pela linha correspondente na matriz B, i.e., se escolhermos o segundo elemento da matriz A, este será multiplicado pela segunda linha da matriz B. A complexidade temporal e o número de instruções aritméticas de vírgula flutuante são iguais aos do algoritmo 1. O seguinte excerto de código representa o algoritmo aplicado:

```
for(i = 0; i < n; i++){
  for(k = 0; k < n; k++){
    for(j = 0; j < n; j++){
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

Figura 2: Excerto de código do algoritmo 2.

2- Métricas de desempenho e metodologia de avaliação

A biblioteca PAPI disponibiliza uma enorme lista de eventos que ajudam o utilizador a ter leituras e monitorização consistente entre a estrutura do código e a eficiência do mapeamento do código para a arquitetura envolvente. Um dos eventos mais importantes presentes no PAPI são na leitura dos *cache misses* nas *caches* L1 e L2 (necessidade de aceder ao nível seguinte de memória para a computação).

Outras métricas utilizadas foram o número de *threads*, tempo de execução, tamanho da matriz, número instruções e ciclos. Foram ainda calculadas algumas métricas derivadamente, sendo estas, as operações por vírgula flutuante (FLOPS), no qual a fórmula usada foi: $\frac{2*TamanhoMatriz^3}{TempoDeExecução}$, rácio de instruções por ciclo, no qual a fórmula usada foi: $\frac{NrDeInstruções}{NrDeCiclos}$ e as *cache misses* por FLOP em L1 e L2, onde a fórmula usada foi: $\frac{CacheMissesEmL_X}{2*TamanhoMatriz^3}$.

A metodologia de avaliação aplicada neste trabalho foi na comparação dos resultados obtidos. Através destas comparações de resultados entre os tempos de execução, aos acessos de memória aos seguintes níveis, a mudança do número de *threads*, as instruções por ciclo e as *cache misses* por FLOP em L1 e L2, foi possível chegar à conclusão de que cada uma destas métricas ajudam na obtenção de resultados finais distintos.

Para obtenção destes resultados foi utilizado um processador Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, com 32KB de cache L1, 256KB de cache L2 e 8MB de cache L3 e ainda, uma otimização O3 ao compilar o programa em C++ de forma a obter os melhores resultados possíveis.

3- Resultados e análise

Performance do algoritmo 1 em C++ e Java

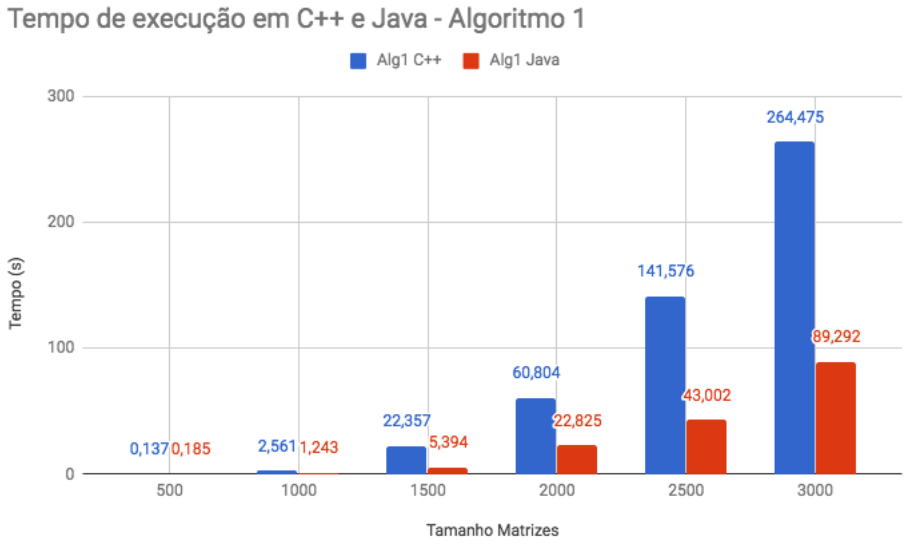

Figura 3: Tempo de execução do algoritmo 1 em C++ e em Java.

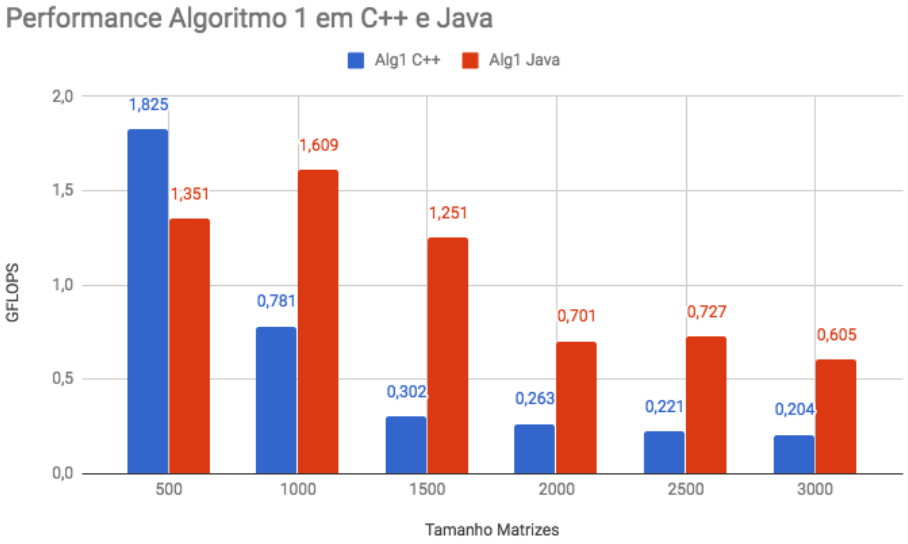

Figura 4: Número de GFLOPS do algoritmo 1 em C++ e em Java.

As figuras 3 e 4 representam, respetivamente, o tempo de execução e o número de FLOPS, do algoritmo 1 programados e executados em C++ e em java. Foram testados valores para os diferentes tamanhos das matrizes, entre 500 a 3000. Pelas figuras é possível verificar que a execução em java obteve melhores resultados que a em C++. No teste da matriz com tamanho de 500 a performance

em C++ foi um pouco melhor que nos restantes tamanhos realizados no teste, isto pode estar relacionado com erro significativo de leitura, pois neste teste o tempo de execução foi de 0,137 segundos para o código C++ em vez dos 0,185 segundos do código java. Nos tamanhos da matriz de 1000 a 3000 a performance em java foi melhor cerca de 2 a 3 vezes do que a em C++.

Performance do algoritmo 1 e 2 em C++

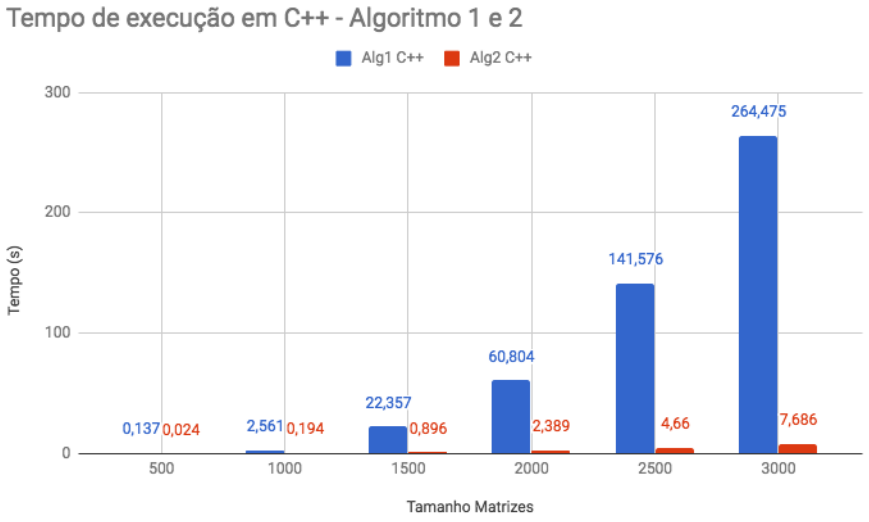

Figura 5: Tempo de execução do algoritmo 1 e 2 em C++.

Na figura 5 foi realizada uma comparação no tempo de execução do algoritmo 1 com o algoritmo 2. Como é possível verificar o algoritmo 2 obteve substancialmente um melhor tempo de execução que o algoritmo 1. Na figura seguinte é possível comparar a performance de ambos os algoritmos, no qual foi usada a unidade de Giga FLOPS.

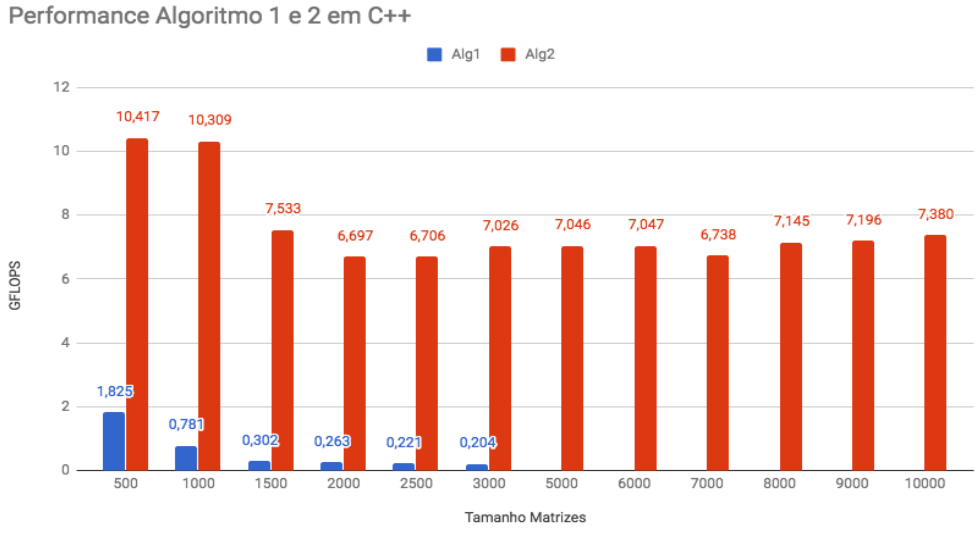

Figura 6: Performance do algoritmo 1 e 2 em C++.

Após a análise da figura 6, e como referido anteriormente, podemos comprovar a melhoria da performance entre os dois algoritmos, quando a matriz de teste teve um tamanho de 1000, a diferença foi superior em 10 vezes. É possível verificar a estabilização da performance dos dois algoritmos quando o primeiro obtém um valor aproximado de 0,3 GFLOPS e o segundo um valor aproximado de 7 GFLOPS.

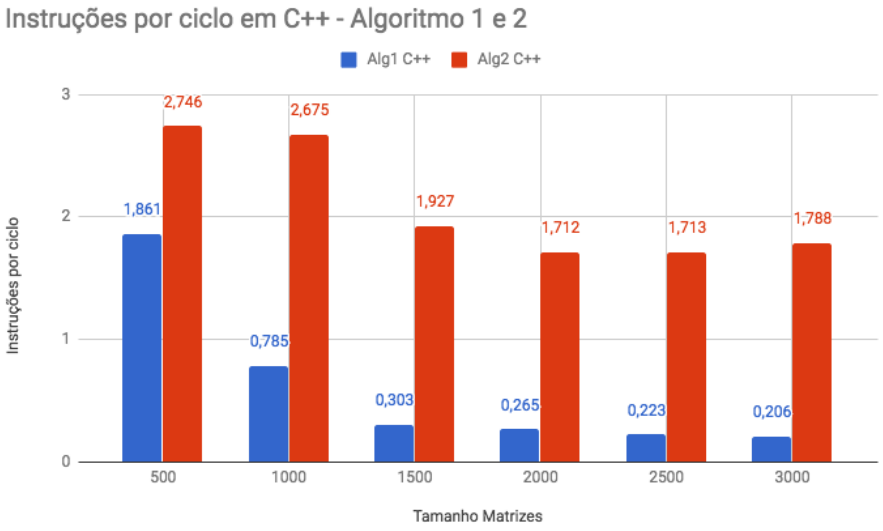

Figura 7: Instruções por ciclo do algoritmo 1 e 2 em C++.

Na figura 7 podemos constatar que o algoritmo 2 obteve um maior número de instruções por ciclo, em que ambos começaram a estabilizar quando as matrizes têm um tamanho de 1500.

Impacto da cache na performance do algoritmo 1 e 2 em C++

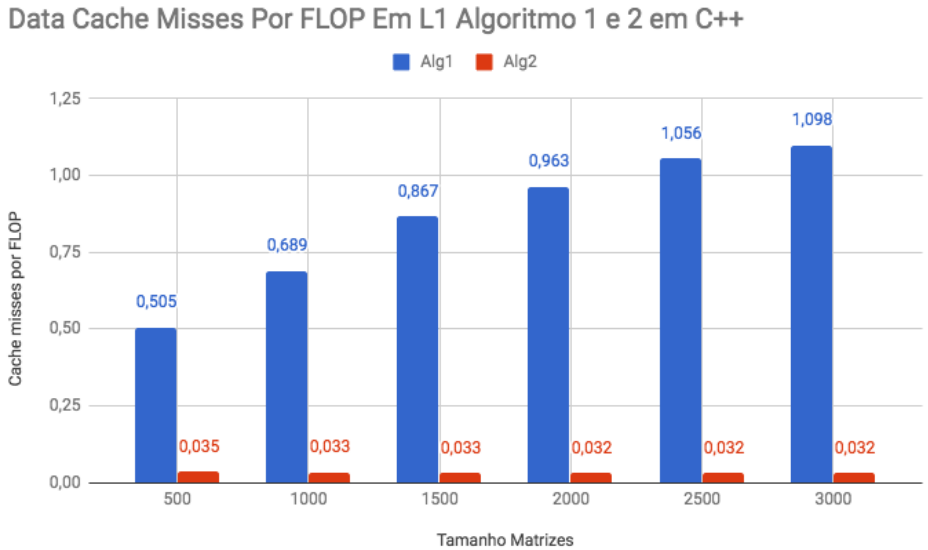

Figura 8: Data Cache Misses por FLOP em L1 do algoritmo 1 e 2 em C++.

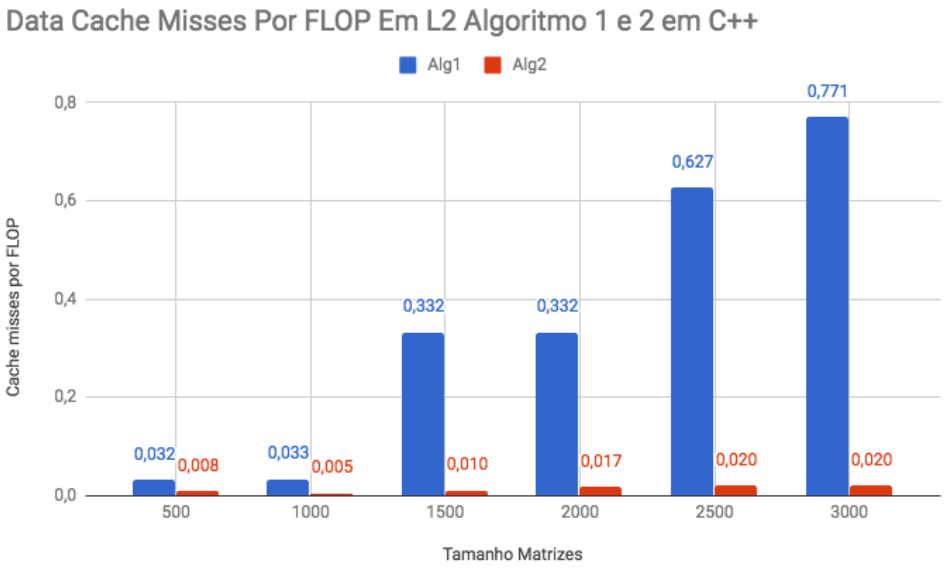

Figura 9: Data Cache Misses por FLOP em L2 do algoritmo 1 e 2 em C++.

Em cada processo, o processador procura primeiro na *cache* L1 pelos dados, caso não o encontre, passa para L2 e caso não encontre passa para L3 e assim sucessivamente até a RAM. Neste ponto foi apenas considerado as *cache misses* em L1 e L2. A partir das duas figuras anteriormente expostas, é possível comparar nos dois algoritmos, quantas *cache misses* ocorreram por FLOP em L1 e L2. Como era de esperar, o segundo algoritmo obteve um número muito inferior, pelo que consegue encontrar os dados à priori. Em ambos os algoritmos, os *cache misses* por FLOP em L1 e L2 começam a estabilizar quando as matrizes têm um tamanho de aproximadamente 2000.

Performance dos algoritmos com paralelismo

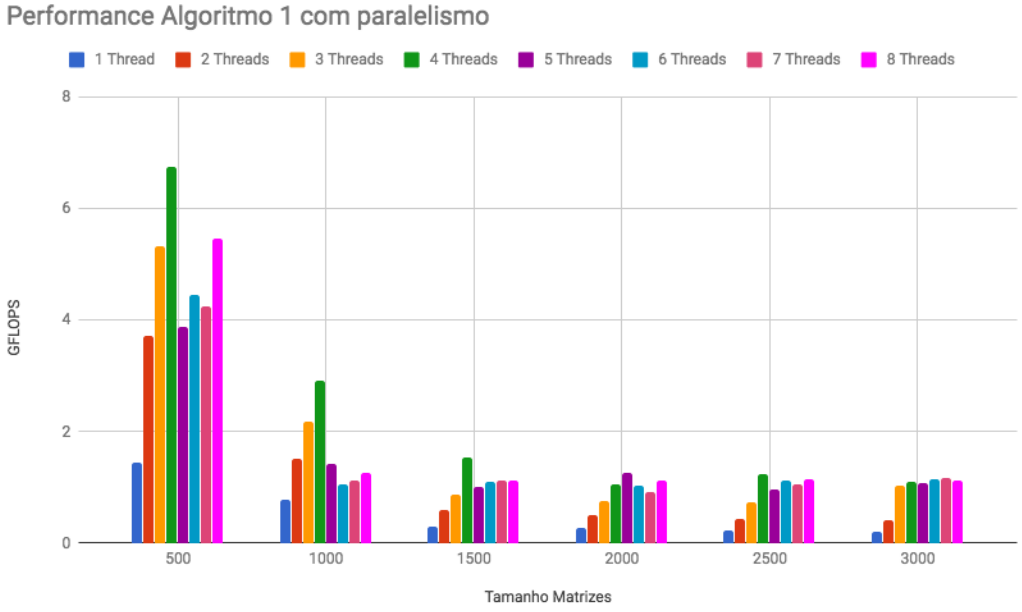

Figura 10: Performance do algoritmo 1 com paralelismo em C++.

Na análise da performance do algoritmo 1 usando paralelismo, é possível verificar pela figura anterior que entre 1 a 4 *threads* a performance aumenta com o aumento do número de *threads*, isto porque dividindo os processos entre *threads* facilita o tempo de execução. Contudo, a utilização de muitas *threads* pode não ser benéfico, pois pode haver uma sobrecarga dos recursos de hardware e isto pode ser observado quando utilizamos 5 a 8 *threads*.

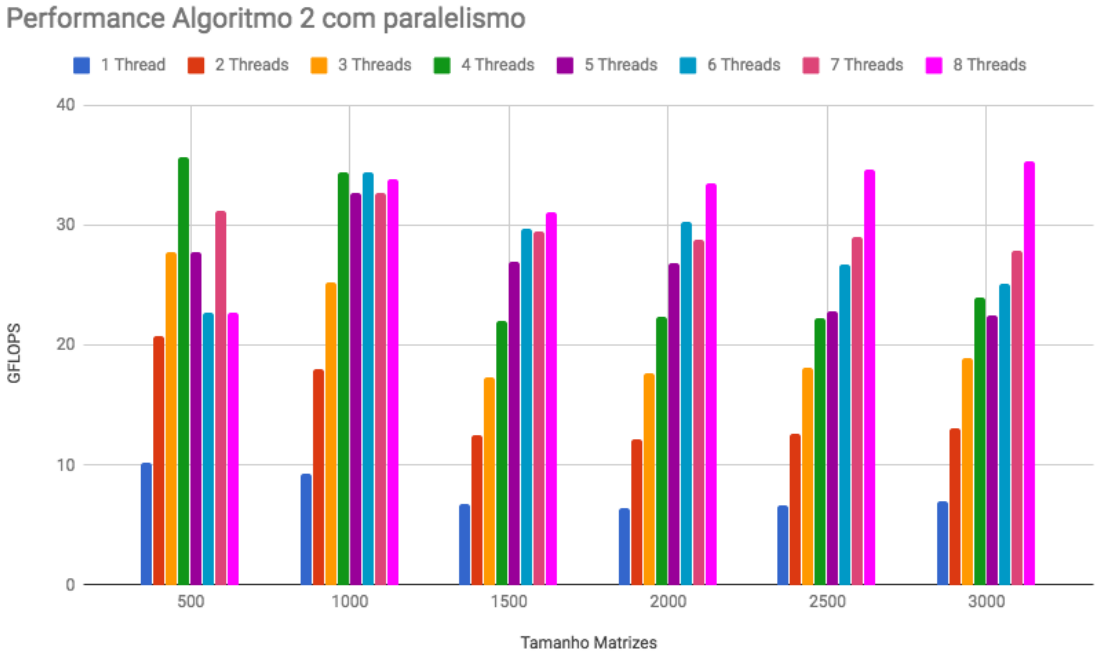

Figura 11: Performance do algoritmo 2 com paralelismo em C++.

No caso do algoritmo 2, podemos confirmar que o número de Giga FLOPS aumenta drasticamente em relação ao algoritmo 1 usando a versão paralelizada. Quanto à performance em relação uso de *threads*, quando as matrizes têm um tamanho entre 500 a 1000 e o uso de 1 a 4 *threads*, a performance aumenta com o aumento do número de threads e entre 5 a 8 *threads* a performance é desequilibrada. Considerando um tamanho de matrizes entre 1500 a 3000, a performance aumenta com o aumento do número de *threads* (entre 1 a 8), tendo um pico em aproximadamente 35 Giga FLOPS quando é utilizado 8 *threads*.

4- Conclusões

Este projeto teve como objetivo a análise de performance de duas linguagens de programação e uso de dois algoritmos com diferentes métodos de acesso aos dados, permitindo assim estudar os diferentes acessos aos níveis hierárquicos da memória.

É possível afirmar que a utilização da linguagem java obteve melhores tempos de execução do que a linguagem em C++. Na comparação dos dois

algoritmos aplicados, o segundo obteve uma melhor performance do que o primeiro, pois não necessita de fazer muitos acessos à memória, tendo mais tempo para processar instruções aritméticas de vírgula flutuante. Desta forma, obtém uma melhor performance e melhores tempos de execução.

Quando é aplicado o paralelismo, na comparação dos dois algoritmos, o segundo obteve resultados muito significativos. O primeiro algoritmo obteve melhores resultados na aplicação de 4 *threads*, já a utilização de mais *threads* neste algoritmo gerou instabilidade. No segundo algoritmo, foi concluído que na utilização de tamanhos pequenos de matrizes, a aplicação de 4 *threads* obteve melhores resultados. Contudo, na utilização de tamanhos maiores de matrizes, a aplicação de 8 *threads* obteve melhores resultados.