Universidade de São Paulo (USP)
Instituto de Ciências Matemáticas e de Computação (ICMC)

Trabalho 1 - Computação de Alto Desempenho

Beatriz Lomes da Silva

Gustavo Moura Scarenci de Carvalho Ferreira

Hugo Hiroyuki Nakamura

Matheus Henrique Dias Cirillo

São Carlos - São Paulo

16 de maio de 2024

Beatriz Lomes da Silva NUSP: 12548038
Gustavo Moura Scarenci de Carvalho Ferreira NUSP: 12547792
Hugo Hiroyuki Nakamura NUSP: 12732037
Matheus Henrique Dias Cirillo NUSP: 12547750

Trabalho 1 - Computação de Alto Desempenho

Resolução de sistemas lineares com Open MP

Relatório apresentado para a disciplina *Computação de Alto Desempenho (SSC0903)*, orientado pelo professor *Paulo Sérgio Lopes de Souza*, a fim de obter a nota do primeiro trabalho prático do semestre.

São Carlos - São Paulo

16 de maio de 2024

Sumário

1 Contextualização **2**
1.1 Método Jacobi-Richardson 2
1.2 Critério de convergência para o método Jacobi-Richardson 2

2 Implementação do Jacobi-Richardson **3**
2.1 Criação e normalização da matriz de equações lineares 3
2.2 Jacobi-Richardson sequencial 4
2.3 Jacobi-Richardson paralelo 5

3 Organização do projeto **6**

4 Compilação e execução **7**
4.1 Instruções básicas 7
4.2 Diretivas de compilação 7
4.3 Flags de otimização 8

5 Análises e resultados **9**
5.1 Testes 9
5.2 Teste principal 10

6 Conclusão **17**

Referências **18**

Apêndice **19**

1 Contextualização

O método Jacobi-Richardson é um algoritmo de resolução de sistemas lineares que consiste em um método iterativo para fornecer uma sequência de aproximantes da solução. Para este trabalho, o método Jacobi-Richardson é implementado na linguagem *C* utilizando programação sequencial e paralela com *OpenMP*, buscando a maximização do desemepenho por meio de um maior *speedup* com uma eficiência razoável.[1]

1.1 Método Jacobi-Richardson

Dado um sistema linear, o método Jacobi-Richardson [2][3] consiste em aplicar valores iniciais às variáveis a fim de encontrar uma solução aproximada. Para tanto, é preciso remodelar o sistema linear, isolando as variáveis da diagonal principal. Por exemplo, considere o sistema linear

$$\begin{cases} 3x_1 - x_2 - x_3 = 1 \\ -x_1 + 3x_2 + x_3 = 3 \\ 2x_1 + x_2 + 4x_3 = 4 \end{cases} \iff \begin{cases} x_1 = \frac{1}{3} \cdot (1 + x_2 + x_3) \\ x_2 = \frac{1}{3} \cdot (3 + x_1 - x_3) \\ x_3 = \frac{1}{4} \cdot (4 - 2x_1 - x_2) \end{cases}$$

Sistema 1: rearranjo do sistema linear para Jacobi-Richardson

Dessa forma, aplicando valores iniciais para uma iteração $\vec{x}^{(k)}$, é possível encontrar valores aproximados para a próxima iteração $\vec{x}^{(k+1)}$.

1.2 Critério de convergência para o método Jacobi-Richardson

Para garantir a convergência do sistema, é preciso que *a soma dos coeficientes de uma linha do sistema, sem conter o da diagonal principal, divido pelo coeficiente da diagonal principal seja menor que 1.* Isso equivale a dizer que, *em uma linha, a soma de todos os coeficientes fora da diagonal principal deve ser menor que o valor do coeficiente da diagonal principal.* Nesse caso, essa matriz é dita de diagonal dominante.

$$\begin{cases} 0 + \frac{a_{12}}{a_{11}} + ... + \frac{a_{1N}}{a_{11}} < 1 \\ \frac{a_{21}}{a_{22}} + 0 + ... + \frac{a_{2N}}{a_{22}} < 1 \\ \vdots \\ \frac{a_{N1}}{a_{NN}} + ... + \frac{a_{NN-1}}{a_{NN}} + 0 < 1 \end{cases} \iff \begin{cases} a_{12} + ... + a_{1N} < a_{11} \\ a_{21} + ... + a_{2N} < a_{22} \\ \vdots \\ a_{N1} + ... + a_{NN-1} < a_{NN} \end{cases}$$

Sistema 2: regras para o critério de convergência

2 Implementação do Jacobi-Richardson

Nessa seção, será descrito como efetivamente o método Jacobi-Richardson foi implementado, mostrando as partes mais importante dos códigos e comentando-as. O código em sua plenitude pode ser visto no repositório público do Github, disponível em: https://github.com/GuScarenci/jacobi-openmp-ssc0903

2.1 Criação e normalização da matriz de equações lineares

A função *initiateMatrixAndVectors()* é responsável por criar a matriz de equações lineares e o vetor de coeficientes independentes.

Primeiro, ela cria uma matriz de números aleatórios, limitadas por *LIMIT_RAND* (definido como 10), mas sem iniacilizar a diagonal principal. Depois, para garantir a convergência do sistema, instancia-se o valor da diagonal principal como a soma dos coeficientes da linha acrescido de 1, gerando uma *matriz de diagonal dominante*.

```
//Inicializa a matriz.
float sum = 0;
for(int j = 0;j<N;j++){
    if(j != i){
        a[i*N + j] = (float)(rand()%LIMIT_RAND);
        sum += fabsf(a[i*N + j]);
    }
}
a[i*N+i] = sum + 1;
```

Código 1: criação e instanciação da matriz de coeficientes dependentes.

A limitação dos valores aleatórios gerados e o modo como se garante uma matriz dominante pode alterar **massivamente** o número de iterações e tempo de resposta. Nesse trabalho escolheu-se valores padronizados, de certa forma, pelo professor da disciplina, que geraram tempos de resposta relativamente altos.

Em seguida, normaliza-se a matriz, dividindo todos os coeficientes das linhas pelo respectivo valor da diagonal principal, que será zerada.

```
//Armazena diagonal original.
diagonal[i] = a[i*N+i];

//Normaliza a matriz.
for(int j = 0;j<N;j++){
    if(j != i){
        a[i*N + j] = a[i*N + j]/diagonal[i];
    }
}
a[i*N + i] = 0;
```

Código 2: normalização da matriz de coeficientes dependentes.

Por fim, é criado e normalizado o vetor de coeficientes independentes.

```
//Inicializa constantes independentes
b[i] = (float)(rand()%LIMIT_RAND);
```

```
//Normaliza as constantes
b[i] = b[i]/diagonal[i];
```

Código 3: criação e normalização do vetor de coeficientes independentes.

2.2 Jacobi-Richardson sequencial

Com a matriz de coeficientes dependentes normalizada a e o vetor de coeficientes independentes normalizado b, o primeiro passo é definir os valores iniciais para as primeiras iterações de $\vec{x}^{(0)}$, que nesse caso são os mesmos valores dos coeficientes independentes.

```
//Chute inicial igual a b[]
    for(int i = 0;i<N;i++){
        x[i] = b[i];
    }
```

Código 4: inicialização dos valores iniciais para $\vec{x}^{(0)}$.

Em seguida, calcula-se a próxima iteração do método. Como a matriz a está normalizada, basta realizar uma multiplicação escalar entre a linha e o vetor x e subtrair o resultado do respectivo coeficiente independente.

```
//Calculo do vetor X_k+1
for(int i = 0;i <N;i++){
    float sum = 0;
    for(int j = 0; j < N; j++){
        sum += a[i * N + j] * x[j];
    }
    nextX[i] = (b[i]-sum);
}
```

Código 5: cálculo da próxima iteração

Por fim, é preciso obter os valores máximos da diferença de iterações ($maxDif$) e do vetor x ($maxX$) para obter Mr. Também é preciso atualizar os valores de x para $nextX$, para prosseguir à próxima iteração.

```
//Verificacao do criterio de parada
float maxDif = -1;
float maxX = -1;

for(int i = 0;i<N;i++){
    float currentDif = fabsf(nextX[i] - x[i]);
    if (currentDif > maxDif) {
        maxDif = currentDif;
    }
    float currentAbsX = fabsf(nextX[i]);
    if (currentAbsX > maxX) {
        maxX = currentAbsX;
    }
}
mr = maxDif/maxX;

//Atualiza o o vetor X_k com o vetor X_k+1
float* temp = x;
x = nextX;
```

```
nextX = temp;
```

Código 6: atualização de Mr e de x

2.3 Jacobi-Richardson paralelo

Similarmente à versão sequencial, o vetor x da primeira iteração, $\vec{x}^{(0)}$, é inicializado com os coeficientes de b, mas utiliza uma diretiva *parallel for* para separar as iterações em threads, e, também, a diretiva *simd* para utilizar instruções vetoriais em cada conjunto de iterações.

```
//Chute inicial igual a b[]
#pragma omp parallel for simd
for(int i = 0;i<N;i++){
    x[i] = b[i];
}
```

Código 7: definição dos valores iniciais para $\vec{x}^{(0)}$ com paralelismo e vetorização.

Para o cálculo de x da próxima iteração, $\vec{x}^{(k+1)}$, divide-se as linhas da matriz entre threads, com a diretiva *parallel for*. Cada uma delas calculará o produto escalar de suas linhas pelo vetor de variáveis da iteração atual x. Esse processo é *vetorizado* e *reduzido*, com a diretiva *simd reduction*, realizando instruções vetoriais e horizontais.

```
//Calculo do vetor X_k+1
#pragma omp parallel for shared(x, nextX, a)
for(int i = 0;i < N;i++){
    float sum = 0;

    #pragma omp simd reduction(+:sum)
    for(int j = 0; j < N; j++){
        sum += a[i*N + j] * x[j];
    }
    nextX[i] = b[i] - sum;
}
```

Código 8: cálculo da próxima iteração ($\vec{x}^{(k+1)}$).

A cada iteração, uma task é responsável pelo critério de parada, pois ela é uma verificação independente e outra iteração pode ser iniciada enquanto uma task espera para ser verificada.

Uma diretiva *simd reduction* obtém a máxima diferença entre duas iterações (*currentError*) e o máximo valor de variável da iteração atual (*maxVariable*). Esse resultado fornece Mr, que se for menor que *errorTolerance*, finaliza as iterações.

```
//Verificacao do criterio de parada.
#pragma omp task shared(stop) firstprivate(nextX, x)
{
    float mr = 1;
    float maxDif = -1;
    float maxX = -1;

    #pragma omp simd reduction(max:maxDif,maxX)
    for(int i = 0;i<N;i++){
        float currentDif = fabsf(nextX[i] - x[i]);
```

```
            if (currentDif > maxDif) {
                maxDif = currentDif;
            }
            float currentAbsX = fabsf(nextX[i]);
            if (currentAbsX > maxX) {
                maxX = currentAbsX;
            }
        }

        mr = maxDif/maxX;
        stop = !(mr > errorTolerance);
    }
```

Código 9: critério de parada das iterações.

Por fim, após a criação da task, os vetores de variáveis são atualizados e a proxima iteração acontece.

3 Organização do projeto

Dentro da pasta raíz do projeto existe a pasta *src*, onde estão presentes os códigos fonte e seus cabeçalhos, a pasta *scripts*, onde está presente o arquivo *daq.py* usado para rodar os códigos várias vezes e coletar dados dos tempos de resposta, o `Makefile`, usado para facilitar os comandos de compilação e execução do código, e o *README.md* com uma breve explicação do contexto do trabalho e instruções de execução. Essa organização é ilustrada na figura 1 abaixo.

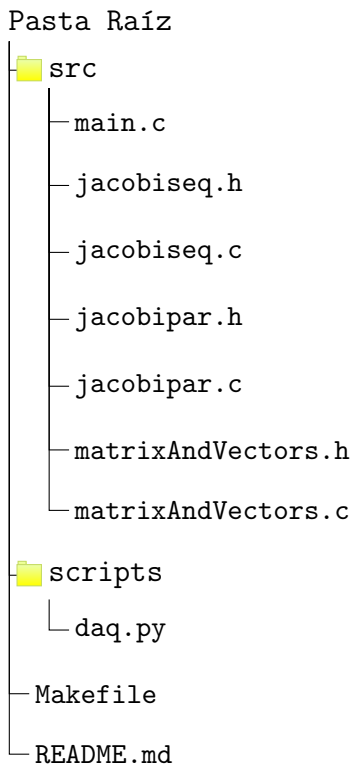

Figura 1: Estrutura do Projeto

4 Compilação e execução

4.1 Instruções básicas

Para compilar o código, execute:

```
$ make
```

Para executar o código do método jacobi sequencial, faça:

```
$ make run_seq ARGS = "<N> <T> <S> <E>"
```

Mas para executar o código do método jacobi sequencial, faça:

```
$ make run_par ARGS = "<N> <T> <S> <E>"
```

Por fim, para executar ambos os códigos sequencial e paralelo, faça:

```
$ make run ARGS = "<N> <T> <S> <E>"
```

Para todas essas execuções:

- $< N >$ é a ordem da matriz;
- $< T >$ é o número de threads a se utilizar;
- $< S >$ é o valor da seed para a geração aleatória de valores;
- $< E >$ equação do sistema que o usuário deseja escolher para associar os resultados obtidos.

4.2 Diretivas de compilação

As diretivas de compilação [4] em C, como #ifdef, #ifndef, #if, e #else, oferecem uma poderosa habilidade de incluir ou excluir funcionalidades do código de forma condicional. Isso traz várias vantagens, especialmente em termos de depuração, gerenciamento de versões e compatibilidade com diferentes plataformas.

No contexto do código fornecido, se utiliza diretivas de pré-processador para permitir a escolha entre duas versões da função de cálculo de Jacobi: uma paralela e outra sequencial. Isso é feito durante a fase de compilação, por meio da definição da flag -DJACOBIPAR.

- **Com a flag** `-DJACOBIPAR`: Ao compilar o código com a flag `-DJACOBIPAR`, a macro `JACOBIPAR` é definida, instruindo o compilador a utilizar a versão paralela da função de Jacobi. Neste caso, a chamada da função será:

```
float* results = jacobipar(a,b,N,TOLERANCE);
```

- **Sem a flag** `-DJACOBIPAR`: Caso a flag `JACOBIPAR` não seja definida durante a compilação, o código utilizará a versão sequencial da função de Jacobi. A chamada da função será:

```
float* results = jacobiseq(a,b,N,TOLERANCE);
```

4.3 Flags de otimização

As flags de otimização `-O1`, `-O2`, `-O3`, e `-Ofast` no compilador GCC (GNU Compiler Collection) são usadas para ajustar o nível de otimização aplicado ao código durante a compilação. Cada flag ativa um conjunto de otimizações que podem melhorar o desempenho e a eficiência do código gerado, embora com diferentes trade-offs em termos de tempo de compilação e possíveis impactos na depuração [5].

- `-O1`: Otimização básica, melhora o desempenho sem complicar a depuração.
- `-O2`: Otimização intermediária, oferece um bom equilíbrio entre desempenho e tempo de compilação.
- `-O3`: Otimização agressiva, melhora o desempenho, mas pode aumentar o tempo de compilação e o tamanho do código.
- `-Ofast`: Otimização máxima, inclui todas as otimizações de `-O3` e desconsidera conformidades estritas com o padrão, priorizando o desempenho máximo.

Escolha para o projeto

Inicialmente, ao usar a flag `-Ofast`, foi observado que o código sequencial apresentava um nível de otimização muito elevado, o que obscurecia os ganhos de desempenho obtidos pela paralelização.

Para contornar esse problema e visualizar de forma mais clara os efeitos reais da paralelização, optou-se por utilizar a flag `-O1`, que aplica um nível mínimo de otimização. Isso permitiu que a influência da paralelização no desempenho fosse mais evidente, sem que as otimizações agressivas do compilador interferissem significativamente.

A tabela 1 foi criada para demonstrar o impacto das diferentes flags de otimização no tempo de execução das instruções SIMD. Os resultados mostram que, tanto para `-Ofast` quanto para `none` (sem flag de otimização), não há aumento de desempenho com a adição do SIMD, o que elimina essas duas opções. Entre as restantes, a flag que apresenta o maior ganho de desempenho é `-O2`. No entanto, o grupo optou por

utilizar `-O1`, acreditando que, por otimizar menos o código [5], esta opção mantém o código em C o mais próximo possível do código de máquina, enquanto ainda permite o uso das instruções SIMD do processador. Dessa forma, é mais previsível e permite observar melhor o impacto da paralelização do código. Entretanto, o uso de uma flag de otimização tão baixa traz seus problemas, o tempo de resposta do código compilado com `-O1` pode ficar até quatro vezes mais lento do que usando a flag `-O3` e em alguns casos até uma ordem de magnitude mais lento que `-Ofast`. (Esses resultados podem ser observados na Tabela 2).

Carga	**Threads**	**none**	**-O1**	**-O2**	**-O3**	**-Ofast**
500	**4**	0.932	2.820	7.078	3.848	0.967
1000	**4**	0.958	4.066	6.434	3.552	1.010

Tabela 1: SIMD *speedup* para flags de otimização

Carga	**Threads**	**-O1** [s]	**-O3** [s]	**-Ofast**[s]
1000	**1**	60.364	14.484	2.542
	4	2.513	2.413	1.742
2000	**1**	162.222	157.333	40.678
	4	33.919	32.612	20.860

Tabela 2: Tempo de execução para diferentes flags de otimização

5 Análises e resultados

5.1 Testes

Para realizar os testes do projeto, foi desenvolvido um script em Python visando automatizar o processo de compilação e execução do código. O script utiliza o comando *make run* para compilar e executar o código, capturando o tempo de execução tanto pelo comando *time* do shell quanto pela função *omp_get_wtime*, que cerca a execução do algoritmo Jacobi no código.

O script é configurado para executar quantas vezes o usuário especificar, variando tanto a carga de trabalho quanto o número de threads utilizado. Para cada execução, o script registra o tempo bruto de todas as execuções (considerando diferentes cargas e contagens de threads) até aquele momento.

Além disso, o script calcula e armazena estatísticas relevantes para cada execução, incluindo a média, o desvio padrão, a mediana, o *speedup* e a eficiência. Essas métricas são essenciais para avaliar o desempenho do código paralelo em comparação com o código sequencial e para entender o impacto das diferentes condições de teste.

Máquina

Os testes foram realizados em uma máquina equipada com um processador Intel Core i7-7700HQ, operando a 2.80GHz (com capacidade de atingir 3.80GHz no modo turbo). O processador possui 64KB de cache L1 por núcleo, totalizando quatro núcleos físicos e oito threads [6]. O sistema conta ainda com 16GB de memória RAM DDR4 e roda o sistema operacional Fedora Workstation 37.

5.2 Teste principal

O script de Python levou um total de 107 horas e 42 minutos e 42 segundos para executar 30 vezes as cargas de 1000, 3000 e 5000, com números de threads variando entre 1 (sequencial), 2, 4, 8 e 12. Os resultados das 450 execuções podem ser encontradas na tabela 6.

Os *speedups* e eficiências apresentados nas tabelas 3 e 5 são calculados com base na média dada nas tabelas 4 e 6, respectivamente. O tempo de execução sequencial utilizado como referência para encontrar o *speedup*, que é definido a partir das execuções com uma única thread.

Ordem da Matriz	Threads	*speedup*	Eficiência
1000	1	1.000	1.000
	2	12.992	6.496
	4	24.020	6.005
	8	30.189	3.774
	12	13.862	1.115
3000	1	1.000	1.000
	2	9.720	4.860
	4	13.432	3.358
	8	13.645	1.706
	12	12.408	1.034
5000	1	1.000	1.000
	2	9.113	4.556
	4	12.788	3.197
	8	13.245	1.656
	12	12.475	1.040

Tabela 3: *speedup* e Eficiência do código completo

Matriz de ordem 1000

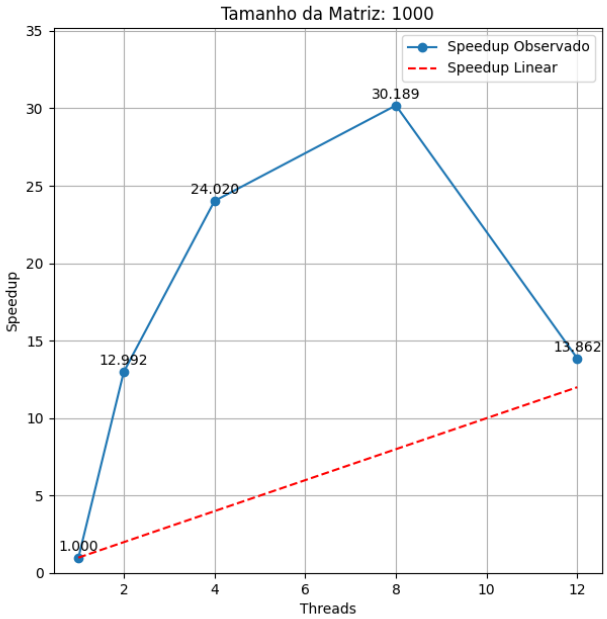

Figura 2: gráfico do *speedup* para carga de 1000

- **1 Thread**: como esperado, o *speedup* é 1, pois é o tempo base de execução sequencial.
- **2 Threads**: um *speedup* significativo de 12.992 foi observado. Isso pode ser porque a carga de trabalho inicial é pequena o suficiente para caber no cache L1, melhorando a localidade temporal e espacial dos dados.
- **4 Threads**: o *speedup* aumenta para 24.020, e a eficiência se mantém, mostrando uma excelente escalabilidade.
- **8 Threads**: um aumento no *speedup* para 30.189, mas a eficiência cai para 3.774. Isso sugere que a saturação do cache L1 e L2 começa a ocorrer, e a comunicação entre threads em núcleos diferentes gera overhead.
- **12 Threads**: o *speedup* diminui para 13.862 com uma eficiência de apenas 1.115, devido ao aumento do overhead de gerenciamento de threads e à limitação do número físico de núcleos (4 núcleos físicos).

Matriz de ordem 3000

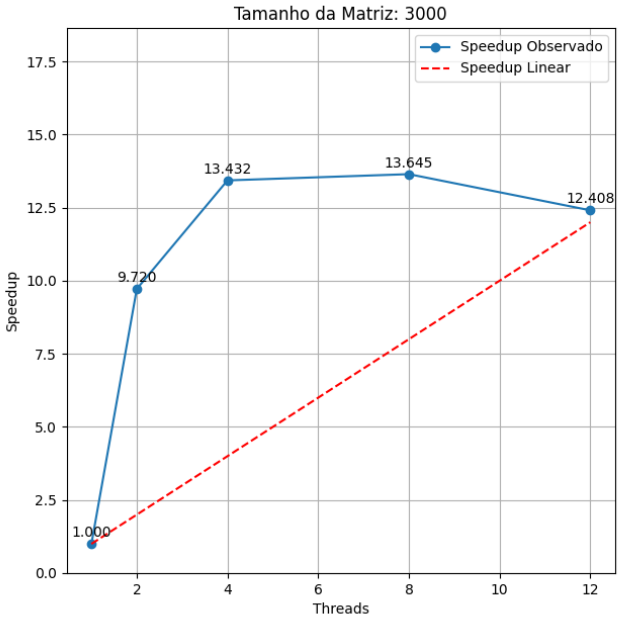

Figura 3: gráfico do *speedup* para carga de 3000

- **1 Thread**: o *speedup* é 1, servindo como tempo base.
- **2 Threads**: o *speedup* é 9.720 com uma eficiência de 4.860. A carga de trabalho maior começa a mostrar mais claramente a localidade espacial e temporal do cache.
- **4 Threads**: o *speedup* aumenta para 13.432, mas a eficiência é 3.358, mostrando uma melhor utilização dos caches L1 e L2, mas também maior overhead de comunicação.
- **8 Threads**: o *speedup* se estabiliza em 13.645, e a eficiência cai para 1.706, indicando que o ganho de desempenho está limitado pela capacidade do cache e pela sobrecarga de sincronização.
- **12 Threads**: a eficiência diminui ainda mais para 1.034, com um *speedup* de 12.408, refletindo o aumento significativo no overhead de gerenciamento de threads.

Matriz de ordem 5000

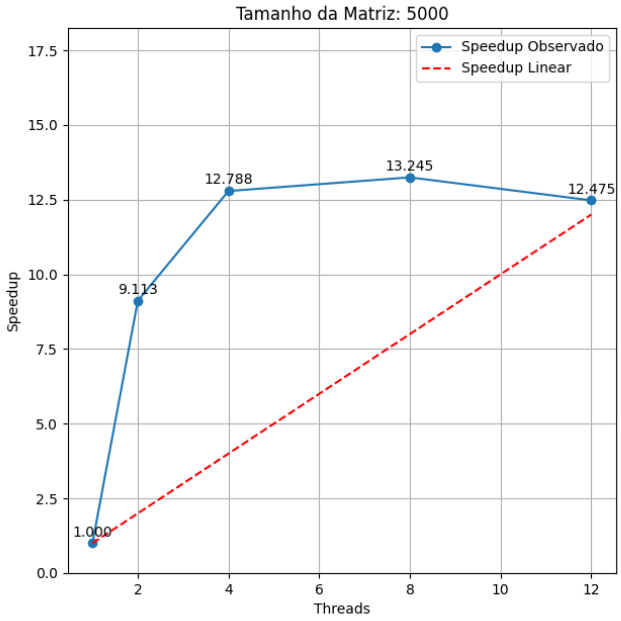

Figura 4: gráfico do *speedup* para carga de 5000

- **1 Thread**: serve como a linha base com um *speedup* de 1.
- **2 Threads**: um *speedup* de 9.113 e eficiência de 4.556 são observados. A maior carga de trabalho beneficia-se do paralelismo, mas a eficiência já mostra sinais de degradação.
- **4 Threads**: o *speedup* de 12.788 e eficiência de 3.197 indicam boa utilização dos recursos do processador, mas com uma redução na eficiência devido ao aumento do overhead de sincronização.
- **8 Threads**: o *speedup* é 13.245 com uma eficiência de 1.656, mostrando que o ganho de desempenho adicional é marginal devido a limitações de cache e comunicação entre threads.
- **12 Threads**: com um *speedup* de 12.475 e uma eficiência de 1.040, a sobrecarga de gerenciamento e a saturação do cache tornam-se os principais limitadores do desempenho.

Fatores que Impactam o Desempenho

Quando o número de threads excede os 4 núcleos físicos, o desempenho é prejudicado pela concorrência pelos recursos dos núcleos físicos, resultando em uma diminuição na

eficiência. Isso explica a queda significativa na eficiência observada na máquina usada ao executar com 8 e 12 threads.

Outro fator importante é o cache L1, que possui apenas 64KB por núcleo. Matrizes menores que cabem completamente no cache L1 ou L2 beneficiam-se significativamente da localidade espacial e temporal, como evidenciado pelo *speedup* considerável para a carga de 1000 (Figura 2). À medida que o tamanho da matriz aumenta, a vantagem do cache diminui, levando a mais acessos à memória RAM, que são mais lentos.

Além disso, a eficiência do código paralelo é impactada pela necessidade de sincronização entre threads. Overheads associados ao gerenciamento de threads, especialmente em configurações com muitas threads, podem anular os benefícios do paralelismo, como observado nos casos de 8 e 12 threads. O Turbo Boost pode aumentar temporariamente a frequência do processador para 3.80GHz, mas esse benefício é mais notável em workloads que não utilizam todos os núcleos simultaneamente. Em testes com múltiplas threads, a capacidade de atingir essas frequências mais altas pode ser limitada.

Outras tabelas e dados importantes

Os tempos de resposta do código completo (Tabela 4) e da parte correspondente apenas ao Método de Jacobi (Tabela 6) foram capturados para permitir uma análise detalhada do desempenho. Os dados apresentados indicam que os tempos de execução quase não diferem, uma vez que a maioria do tempo é consumida pelo Método de Jacobi. No entanto, para matrizes de ordem menor ou em casos onde o método converge rapidamente, a proporção do tempo total gasto na inicialização das matrizes e na saída torna-se mais significativa em relação ao tempo gasto no Método de Jacobi. Nesses casos, a diferença no tempo total do código se torna relevante, destacando a importância de considerar também o tempo de execução das operações auxiliares além do método principal.

Tamanho da Matriz	Threads	Média	Mediana	Desvio Padrão
1000	1	60.364	60.334	0.072
	2	4.646	4.642	0.035
	4	2.513	2.509	0.019
	8	2.000	1.997	0.021
	12	4.355	4.354	0.021
3000	1	1690.125	1690.527	0.778
	2	173.875	171.144	3.961
	4	125.831	125.819	0.166
	8	123.862	123.828	0.330
	12	136.213	136.215	0.063
5000	1	7890.420	7890.684	4.466
	2	865.851	846.925	31.329
	4	617.029	617.042	0.336
	8	595.712	595.665	0.672
	12	632.477	632.397	0.277

Tabela 4: estatísticas sobre o tempo de execução do código completo

Tamanho da Matriz	Threads	*speedup*	Eficiência
1000	1	1.000	1.000
	2	13.091	6.545
	4	24.371	6.093
	8	30.762	3.845
	12	13.976	1.165
3000	1	1.000	1.000
	2	9.737	4.868
	4	13.464	3.366
	8	13.679	1.710
	12	12.435	1.036
5000	1	1.000	1.000
	2	9.121	4.561
	4	12.805	3.201
	8	13.264	1.658
	12	12.492	1.041

Tabela 5: *speedup* e Eficiência da função Jacobi

Tamanho da Matriz	Threads	Média	Mediana	Desvio Padrão
1000	1	60.326	60.296	0.072
	2	4.608	4.605	0.035
	4	2.475	2.472	0.019
	8	1.961	1.959	0.021
	12	4.316	4.315	0.021
3000	1	1689.800	1690.200	0.778
	2	173.550	170.819	3.961
	4	125.507	125.494	0.166
	8	123.535	123.500	0.330
	12	135.889	135.890	0.063
5000	1	7889.523	7889.787	4.464
	2	864.952	846.021	31.329
	4	616.124	616.107	0.335
	8	594.803	594.764	0.676
	12	631.580	631.501	0.277

Tabela 6: estatísticas da função Jacobi

6 Conclusão

Este trabalho apresentou a implementação do método Jacobi-Richardson utilizando C e paralelismo com OpenMP, demonstrando a eficácia do paralelismo no diminuição do tempo de resposta de um código. Observou-se que a paralelização melhora significativamente o tempo de reposta, mas enfrenta limitações devido à sobrecarga de gerenciamento de threads e à saturação de cache em matrizes maiores. A escolha adequada das flags de otimização foi crucial para obter resultados equilibrados que mostrassem os diferentes impactos do paralelismo. Os testes realizados mostraram que a implementação paralela pode alcançar altos níveis de *speedup*, destacando a importância do paralelismo em aplicações científicas e de engenharia.

Referências

[1] P. Lopes de Souza, "Aulas da disciplina SSC-0903." Acessado em 14/05/2024.

[2] L. Abreu, "Solução de Sistemas Lineares - Método de Jacobi-Richardson," 2015. Accessado em 14/05/2024.

[3] Paiva, Afonso, "[SME0300/SME0892] Aula 07: Método de Gauss-Jacobi," 2020. Accessado em 14/05/2024.

[4] GNU Project, "GCC Online Documentation: Preprocessor Options." https://gcc.gnu.org/onlinedocs/gcc/Preprocessor-Options.html. Accessado em 14/05/2024.

[5] GNU Project, "GCC Online Documentation: Optimize Options." https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Accessado em 14/05/2024.

[6] TechPowerUp, "Intel Core i7-7700HQ." https://www.techpowerup.com/cpu-specs/core-i7-7700hq.c3098. Acessado em 14/05/2024.

Apêndice

Tabela 7: Dados brutos dos 30 testes principais

Execução	Tamanho da matriz	Threads	Tempo do código inteiro	Tempo do código separadamente
1	1000	1	60.373	60.336
1	1000	2	4.652	4.613
1	1000	4	2.504	2.466
1	1000	8	1.992	1.954
1	1000	12	4.338	4.300
1	3000	1	1690.722	1690.398
1	3000	2	178.785	178.461
1	3000	4	125.743	125.419
1	3000	8	124.044	123.720
1	3000	12	136.276	135.952
1	5000	1	7894.519	7893.624
1	5000	2	918.879	917.985
1	5000	4	616.967	616.071
1	5000	8	595.662	594.759
1	5000	12	632.207	631.310
2	1000	1	60.331	60.292
2	1000	2	4.641	4.603
2	1000	4	2.526	2.489
2	1000	8	1.995	1.956
2	1000	12	4.353	4.315
2	3000	1	1690.604	1690.280
2	3000	2	177.639	177.315
2	3000	4	125.570	125.246
2	3000	8	123.697	123.371
2	3000	12	136.125	135.802
2	5000	1	7895.257	7894.362
2	5000	2	844.722	843.827
2	5000	4	617.577	616.682
2	5000	8	595.342	594.439
2	5000	12	632.188	631.293
3	1000	1	60.308	60.268

Continua na próxima página...

Tabela 7 – Continuação

Execução	Tamanho da matriz	Threads	Tempo do código inteiro	Tempo do código separadamente
3	1000	2	4.649	4.612
3	1000	4	2.529	2.492
3	1000	8	2.019	1.979
3	1000	12	4.330	4.291
3	3000	1	1690.503	1690.177
3	3000	2	178.023	177.698
3	3000	4	125.978	125.652
3	3000	8	124.078	123.751
3	3000	12	136.100	135.777
3	5000	1	7895.119	7894.218
3	5000	2	849.114	848.213
3	5000	4	617.328	616.432
3	5000	8	595.387	594.488
3	5000	12	632.938	632.042
4	1000	1	60.409	60.371
4	1000	2	4.697	4.659
4	1000	4	2.539	2.502
4	1000	8	1.995	1.956
4	1000	12	4.364	4.326
4	3000	1	1690.790	1690.462
4	3000	2	171.213	170.887
4	3000	4	126.116	125.792
4	3000	8	123.807	123.480
4	3000	12	136.207	135.883
4	5000	1	7894.984	7894.088
4	5000	2	915.754	914.858
4	5000	4	617.152	616.247
4	5000	8	595.608	594.705
4	5000	12	632.253	631.356
5	1000	1	60.431	60.394
5	1000	2	4.673	4.635
5	1000	4	2.536	2.498
5	1000	8	1.944	1.906
5	1000	12	4.333	4.295
5	3000	1	1690.782	1690.458

Continua na próxima página...

Tabela 7 – Continuação

Execução	Tamanho da matriz	Threads	Tempo do código inteiro	Tempo do código separadamente
5	3000	2	169.996	169.672
5	3000	4	125.820	125.496
5	3000	8	123.927	123.600
5	3000	12	136.169	135.846
5	5000	1	7885.731	7884.835
5	5000	2	846.596	845.693
5	5000	4	616.822	615.925
5	5000	8	594.868	593.969
5	5000	12	632.457	631.562
6	1000	1	60.355	60.317
6	1000	2	4.675	4.628
6	1000	4	2.498	2.460
6	1000	8	2.004	1.966
6	1000	12	4.347	4.309
6	3000	1	1690.553	1690.228
6	3000	2	177.721	177.395
6	3000	4	125.990	125.666
6	3000	8	123.848	123.521
6	3000	12	136.163	135.839
6	5000	1	7895.077	7894.181
6	5000	2	919.787	918.888
6	5000	4	616.915	616.014
6	5000	8	594.705	593.806
6	5000	12	632.266	631.370
7	1000	1	60.332	60.294
7	1000	2	4.639	4.602
7	1000	4	2.481	2.442
7	1000	8	1.980	1.942
7	1000	12	4.361	4.315
7	3000	1	1690.210	1689.883
7	3000	2	170.386	170.060
7	3000	4	125.790	125.465
7	3000	8	123.412	123.084
7	3000	12	136.189	135.865
7	5000	1	7894.882	7893.986

Continua na próxima página...

Tabela 7 – Continuação

Execução	Tamanho da matriz	Threads	Tempo do código inteiro	Tempo do código separadamente
7	5000	2	848.398	847.497
7	5000	4	617.418	616.522
7	5000	8	595.844	594.941
7	5000	12	632.171	631.276
8	1000	1	60.482	60.443
8	1000	2	4.678	4.640
8	1000	4	2.483	2.444
8	1000	8	1.977	1.938
8	1000	12	4.331	4.292
8	3000	1	1690.823	1690.499
8	3000	2	177.798	177.471
8	3000	4	125.970	125.643
8	3000	8	123.536	123.209
8	3000	12	136.289	135.966
8	5000	1	7885.897	7885.001
8	5000	2	846.433	845.532
8	5000	4	617.078	616.181
8	5000	8	595.901	594.997
8	5000	12	632.341	631.446
9	1000	1	60.405	60.368
9	1000	2	4.677	4.638
9	1000	4	2.508	2.471
9	1000	8	2.016	1.978
9	1000	12	4.315	4.278
9	3000	1	1689.377	1689.053
9	3000	2	170.053	169.729
9	3000	4	125.403	125.079
9	3000	8	123.372	123.045
9	3000	12	136.238	135.914
9	5000	1	7886.046	7885.149
9	5000	2	896.431	895.529
9	5000	4	616.846	615.950
9	5000	8	595.509	594.385
9	5000	12	632.384	631.483

Continua na próxima página...

Tabela 7 – Continuação

Execução	Tamanho da matriz	Threads	Tempo do código inteiro	Tempo do código separadamente
10	1000	1	60.296	60.259
10	1000	2	4.578	4.541
10	1000	4	2.513	2.475
10	1000	8	1.973	1.934
10	1000	12	4.336	4.297
10	3000	1	1691.202	1690.879
10	3000	2	177.923	177.599
10	3000	4	125.795	125.471
10	3000	8	123.732	123.405
10	3000	12	136.271	135.945
10	5000	1	7885.396	7884.499
10	5000	2	919.859	918.962
10	5000	4	617.521	616.625
10	5000	8	596.593	595.695
10	5000	12	632.403	631.508
11	1000	1	60.305	60.266
11	1000	2	4.608	4.570
11	1000	4	2.497	2.459
11	1000	8	1.975	1.937
11	1000	12	4.335	4.297
11	3000	1	1689.238	1688.915
11	3000	2	178.495	178.169
11	3000	4	125.749	125.425
11	3000	8	123.520	123.188
11	3000	12	136.301	135.978
11	5000	1	7888.348	7887.452
11	5000	2	846.230	845.335
11	5000	4	616.711	615.811
11	5000	8	595.936	595.033
11	5000	12	632.618	631.723
12	1000	1	60.348	60.311
12	1000	2	4.617	4.580
12	1000	4	2.483	2.446
12	1000	8	2.023	1.985

Continua na próxima página...

Tabela 7 – Continuação

Execução	Tamanho da matriz	Threads	Tempo do código inteiro	Tempo do código separadamente
12	1000	12	4.355	4.318
12	3000	1	1689.931	1689.603
12	3000	2	170.727	170.403
12	3000	4	125.848	125.522
12	3000	8	123.801	123.477
12	3000	12	136.221	135.898
12	5000	1	7895.100	7894.197
12	5000	2	847.081	846.178
12	5000	4	616.421	615.524
12	5000	8	596.019	595.120
12	5000	12	633.141	632.241
13	1000	1	60.572	60.535
13	1000	2	4.739	4.701
13	1000	4	2.503	2.466
13	1000	8	2.023	1.985
13	1000	12	4.367	4.328
13	3000	1	1690.588	1690.264
13	3000	2	170.344	170.020
13	3000	4	125.741	125.416
13	3000	8	124.111	123.785
13	3000	12	136.131	135.807
13	5000	1	7890.557	7889.662
13	5000	2	844.950	844.055
13	5000	4	616.831	615.933
13	5000	8	595.378	594.479
13	5000	12	632.784	631.888
14	1000	1	60.335	60.297
14	1000	2	4.599	4.561
14	1000	4	2.508	2.471
14	1000	8	1.986	1.948
14	1000	12	4.342	4.304
14	3000	1	1689.626	1689.296
14	3000	2	178.591	178.266
14	3000	4	125.692	125.368
14	3000	8	123.894	123.566

Continua na próxima página...

Tabela 7 – Continuação

Execução	Tamanho da matriz	Threads	Tempo do código inteiro	Tempo do código separadamente
14	3000	12	136.221	135.898
14	5000	1	7884.566	7883.671
14	5000	2	845.399	844.503
14	5000	4	616.333	615.436
14	5000	8	594.984	594.081
14	5000	12	633.013	632.118
15	1000	1	60.319	60.282
15	1000	2	4.652	4.614
15	1000	4	2.527	2.490
15	1000	8	2.027	1.989
15	1000	12	4.342	4.303
15	3000	1	1688.889	1688.566
15	3000	2	170.292	169.966
15	3000	4	125.964	125.640
15	3000	8	124.454	124.121
15	3000	12	136.174	135.846
15	5000	1	7885.324	7884.429
15	5000	2	845.629	844.732
15	5000	4	617.006	616.093
15	5000	8	596.309	595.411
15	5000	12	632.242	631.346
16	1000	1	60.400	60.362
16	1000	2	4.616	4.579
16	1000	4	2.491	2.454
16	1000	8	2.030	1.992
16	1000	12	4.351	4.313
16	3000	1	1690.745	1690.416
16	3000	2	170.911	170.584
16	3000	4	125.988	125.664
16	3000	8	123.693	123.368
16	3000	12	136.132	135.808
16	5000	1	7885.744	7884.848
16	5000	2	851.010	850.109
16	5000	4	616.969	616.073
16	5000	8	594.836	593.933

Continua na próxima página...

Tabela 7 – Continuação

Execução	Tamanho da matriz	Threads	Tempo do código inteiro	Tempo do código separadamente
16	5000	12	632.216	631.316
17	1000	1	60.405	60.368
17	1000	2	4.628	4.590
17	1000	4	2.498	2.459
17	1000	8	1.978	1.940
17	1000	12	4.341	4.304
17	3000	1	1690.347	1690.024
17	3000	2	170.023	169.699
17	3000	4	125.719	125.395
17	3000	8	123.721	123.392
17	3000	12	136.170	135.846
17	5000	1	7895.107	7894.212
17	5000	2	844.669	843.774
17	5000	4	617.408	616.511
17	5000	8	597.500	596.597
17	5000	12	632.381	631.486
18	1000	1	60.279	60.241
18	1000	2	4.630	4.592
18	1000	4	2.510	2.473
18	1000	8	2.008	1.970
18	1000	12	4.366	4.328
18	3000	1	1690.550	1690.224
18	3000	2	169.935	169.609
18	3000	4	125.716	125.390
18	3000	8	123.896	123.571
18	3000	12	136.227	135.903
18	5000	1	7893.998	7893.101
18	5000	2	846.769	845.864
18	5000	4	616.820	615.925
18	5000	8	595.667	594.769
18	5000	12	632.483	631.584
19	1000	1	60.307	60.267
19	1000	2	4.601	4.564
19	1000	4	2.507	2.470

Continua na próxima página...

Tabela 7 – Continuação

Execução	Tamanho da matriz	Threads	Tempo do código inteiro	Tempo do código separadamente
19	1000	8	2.041	2.003
19	1000	12	4.397	4.360
19	3000	1	1690.487	1690.160
19	3000	2	170.369	170.043
19	3000	4	125.817	125.491
19	3000	8	124.104	123.776
19	3000	12	136.262	135.938
19	5000	1	7895.643	7894.743
19	5000	2	845.272	844.374
19	5000	4	617.292	616.392
19	5000	8	596.009	595.104
19	5000	12	632.320	631.421
20	1000	1	60.322	60.285
20	1000	2	4.618	4.580
20	1000	4	2.520	2.482
20	1000	8	1.992	1.954
20	1000	12	4.405	4.366
20	3000	1	1688.857	1688.533
20	3000	2	178.488	178.164
20	3000	4	125.724	125.396
20	3000	8	123.759	123.434
20	3000	12	136.218	135.893
20	5000	1	7895.147	7894.245
20	5000	2	919.599	918.697
20	5000	4	617.275	616.373
20	5000	8	594.586	593.670
20	5000	12	632.352	631.457
21	1000	1	60.327	60.290
21	1000	2	4.615	4.578
21	1000	4	2.527	2.489
21	1000	8	2.009	1.971
21	1000	12	4.375	4.337
21	3000	1	1690.627	1690.299
21	3000	2	171.664	171.340
21	3000	4	125.617	125.293

Continua na próxima página...

Tabela 7 – Continuação

Execução	Tamanho da matriz	Threads	Tempo do código inteiro	Tempo do código separadamente
21	3000	8	124.454	124.128
21	3000	12	136.243	135.920
21	5000	1	7885.784	7884.888
21	5000	2	844.455	843.560
21	5000	4	617.204	616.121
21	5000	8	596.294	595.391
21	5000	12	632.421	631.525
22	1000	1	60.301	60.263
22	1000	2	4.654	4.615
22	1000	4	2.542	2.505
22	1000	8	1.997	1.959
22	1000	12	4.355	4.317
22	3000	1	1688.878	1688.555
22	3000	2	170.116	169.790
22	3000	4	125.995	125.671
22	3000	8	123.503	123.177
22	3000	12	136.386	136.059
22	5000	1	7894.941	7894.035
22	5000	2	845.509	844.603
22	5000	4	616.666	615.764
22	5000	8	594.957	594.060
22	5000	12	632.390	631.494
23	1000	1	60.327	60.290
23	1000	2	4.657	4.619
23	1000	4	2.518	2.481
23	1000	8	1.997	1.959
23	1000	12	4.359	4.322
23	3000	1	1690.817	1690.489
23	3000	2	170.933	170.599
23	3000	4	125.847	125.524
23	3000	8	123.574	123.248
23	3000	12	136.134	135.808
23	5000	1	7894.724	7893.797
23	5000	2	844.935	844.033
23	5000	4	617.227	616.318

Continua na próxima página...

Tabela 7 – Continuação

Execução	Tamanho da matriz	Threads	Tempo do código inteiro	Tempo do código separadamente
23	5000	8	596.277	595.376
23	5000	12	632.820	631.924
24	1000	1	60.541	60.502
24	1000	2	4.642	4.605
24	1000	4	2.507	2.469
24	1000	8	1.986	1.948
24	1000	12	4.369	4.332
24	3000	1	1691.012	1690.688
24	3000	2	171.074	170.751
24	3000	4	125.813	125.489
24	3000	8	124.648	124.320
24	3000	12	136.195	135.869
24	5000	1	7893.988	7893.092
24	5000	2	868.149	867.254
24	5000	4	616.569	615.668
24	5000	8	594.852	593.949
24	5000	12	632.498	631.601
25	1000	1	60.386	60.347
25	1000	2	4.641	4.603
25	1000	4	2.512	2.473
25	1000	8	2.010	1.972
25	1000	12	4.364	4.326
25	3000	1	1688.952	1688.628
25	3000	2	178.446	178.121
25	3000	4	125.921	125.595
25	3000	8	123.246	122.921
25	3000	12	136.190	135.866
25	5000	1	7887.037	7886.142
25	5000	2	848.129	847.232
25	5000	4	617.268	616.370
25	5000	8	595.580	594.677
25	5000	12	632.405	631.509
26	1000	1	60.292	60.254
26	1000	2	4.696	4.659

Continua na próxima página...

Tabela 7 – Continuação

Execução	Tamanho da matriz	Threads	Tempo do código inteiro	Tempo do código separadamente
26	1000	4	2.489	2.452
26	1000	8	1.986	1.949
26	1000	12	4.340	4.302
26	3000	1	1688.982	1688.658
26	3000	2	178.559	178.236
26	3000	4	125.649	125.325
26	3000	8	123.757	123.430
26	3000	12	136.211	135.888
26	5000	1	7890.810	7889.913
26	5000	2	854.058	853.157
26	5000	4	617.513	616.617
26	5000	8	595.923	595.025
26	5000	12	632.941	632.045
27	1000	1	60.299	60.262
27	1000	2	4.657	4.619
27	1000	4	2.536	2.497
27	1000	8	2.015	1.977
27	1000	12	4.370	4.332
27	3000	1	1690.746	1690.419
27	3000	2	170.120	169.796
27	3000	4	126.055	125.729
27	3000	8	123.971	123.647
27	3000	12	136.191	135.868
27	5000	1	7885.635	7884.740
27	5000	2	919.581	918.676
27	5000	4	616.781	615.878
27	5000	8	596.645	595.742
27	5000	12	632.161	631.265
28	1000	1	60.323	60.285
28	1000	2	4.642	4.605
28	1000	4	2.544	2.506
28	1000	8	2.009	1.971
28	1000	12	4.368	4.330
28	3000	1	1689.226	1688.900
28	3000	2	178.511	178.187

Continua na próxima página...

Tabela 7 – Continuação

Execução	Tamanho da matriz	Threads	Tempo do código inteiro	Tempo do código separadamente
28	3000	4	125.955	125.630
28	3000	8	123.917	123.592
28	3000	12	136.219	135.892
28	5000	1	7886.652	7885.757
28	5000	2	845.272	844.377
28	5000	4	617.204	616.303
28	5000	8	596.131	595.226
28	5000	12	632.811	631.916
29	1000	1	60.433	60.393
29	1000	2	4.701	4.662
29	1000	4	2.507	2.469
29	1000	8	1.990	1.952
29	1000	12	4.397	4.359
29	3000	1	1689.044	1688.720
29	3000	2	178.898	178.571
29	3000	4	125.822	125.498
29	3000	8	124.092	123.768
29	3000	12	136.312	135.988
29	5000	1	7885.268	7884.373
29	5000	2	918.245	917.348
29	5000	4	616.537	615.640
29	5000	8	595.520	594.617
29	5000	12	632.444	631.546
30	1000	1	60.381	60.343
30	1000	2	4.619	4.582
30	1000	4	2.551	2.513
30	1000	8	2.010	1.963
30	1000	12	4.336	4.298
30	3000	1	1690.642	1690.317
30	3000	2	170.231	169.907
30	3000	4	126.136	125.812
30	3000	8	124.291	123.964
30	3000	12	136.234	135.910
30	5000	1	7885.333	7884.437
30	5000	2	844.617	843.712

Continua na próxima página...

Tabela 7 – Continuação

Execução	Tamanho da matriz	Threads	Tempo do código inteiro	Tempo do código separadamente
30	5000	4	617.209	616.313
30	5000	8	596.537	595.633
30	5000	12	632.246	631.350