Programação Paralela: Bitonic Sort

Giovanni Venâncio de Souza, Rafael Rocha de Carvalho
gvsouza@inf.ufpr.br, rrcarvalho@inf.ufpr.br

1 Análise Sequencial

Esta seção tem como objetivo apresentar o algoritmo *Bitonic Sort* e a análise sequencial, abordando a complexidade computacional e projeções de escalabilidade forte segundo a lei de Amdahl.

1.1 Bitonic Sort

O algoritmo *Bitonic Sort* é utilizado para ordenação de elementos. Entretanto, o vetor de elementos deve ser uma sequência bitônica. Uma sequência bitônica é uma sequência monotonicamente crescente seguida de uma sequência monotonicamente decrescente. Uma sequência bitônica pode ser definida da seguinte forma:

$$x_0 \leq x_1 \leq x_2 \leq ... \leq x_k \geq x_{k+1} \geq ... \geq x_{n-1}$$

A Figura 1 exemplifica a execução do algoritmo *Bitonic Sort*. A execução do algoritmo consiste em dividir o vetor em grupos, onde os grupos são as metades do vetor. Inicialmente no primeiro passo existe apenas um grupo de $\frac{n}{2}$ elementos, formado pela primeira metade do vetor. No segundo passo, o primeiro grupo é dividido em dois grupos de $\frac{n}{4}$ elementos cada. No próximo passo, irão existir quatro grupos de $\frac{n}{8}$ elementos. Essa divisão ocorre até existirem $\frac{n}{2}$ grupos de um único elemento. Em cada um dos passos são feitas $\frac{n}{2}$ comparações. Para cada elemento de posição i em um grupo, é verificado se o elemento $v[i]$ é maior do que o elemento $v[i + k]$, onde k é o tamanho do grupo. Se a expressão for verdadeira, os elementos são trocados. Ao final dos passos, o vetor de elementos está ordenado. A seção 1.3 faz uma análise da complexidade computacional do *Bitonic Sort*.

1.2 Implementação

Para esta implementação em específico, é considerado que $k = \frac{n}{2}$. Ou seja, a primeira metade do vetor é monotonicamente crescente e a segunda metade é monotonicamente decrescente. Além disso, consideramos que a sequência de elementos já é uma sequência bitônica. Por fim, consideramos que o tamanho do vetor é uma potência de dois.

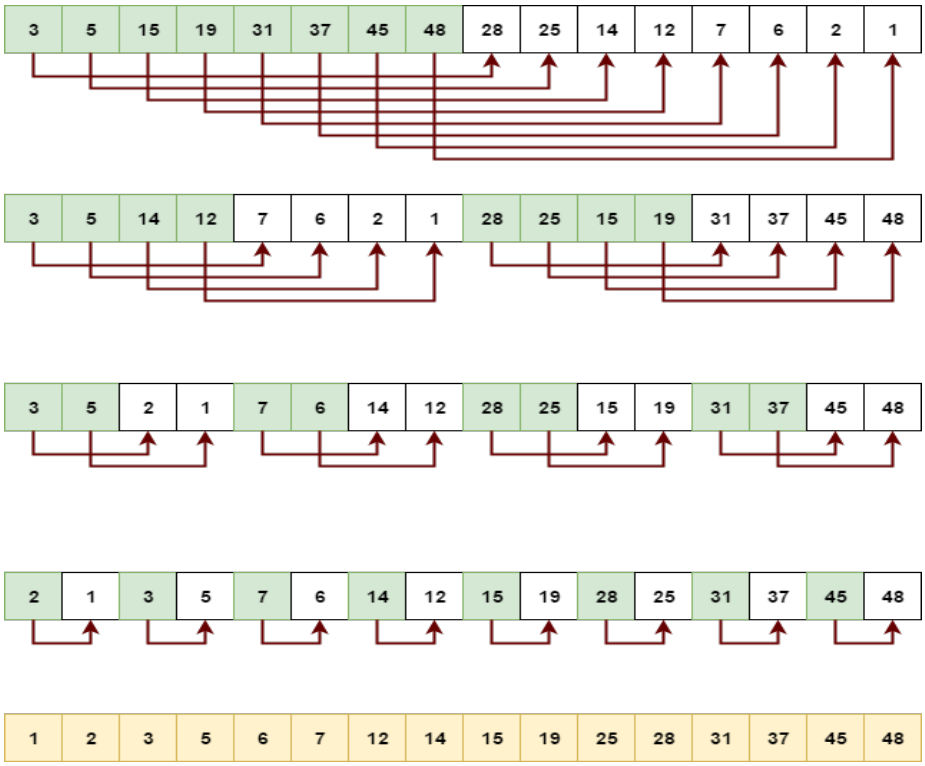

Figura 1: Execução do algoritmo *Bitonic Sort*.

A implementação consiste em, a cada passo do algoritmo, dividir o vetor em metades (primeiro laço). Após essa divisão, o algoritmo itera por cada uma das metades (segundo laço). Para cada uma das metades percorridas, os elementos de cada metade são percorridos e comparados com o elemento na posição $i + k$. Esses passos são repetidos $\log_2(n)$ vezes, onde o tamanho da metade varia de $\frac{n}{2}$ até 1. Por fim, o algoritmo percorre novamente o vetor verificando se ele está totalmente ordenado. O Código 1 apresenta a ordenação do vetor de elementos.

Código 1: Trecho do código de ordenação sequencial

```
for (k = n/2; k >= 1; k /= 2)
    for (i = 0; i < n; i += 2*k)
        for (j = 0; j < k; j++)
            swap(elem, i+j, k);
```

A fim de que o algoritmo possa ser executado com um grande tamanho de vetor, os elementos são do tipo *unsigned short int*. Isso significa que os números variam de 0 a 65535. Sendo assim, na hora de preencher o vetor em uma sequência bitônica, os elementos são repetidos $\frac{n}{65536}$ vezes.

1.3 Complexidade

Para a análise de complexidade computacional do algoritmo *Bitonic Sort*, considere um vetor de tamanho n. O tamanho dos grupos decresce pela metade a cada etapa do algoritmo $(\frac{n}{2}, \frac{n}{4}, ..., 1)$. Ou seja, cada grupo necessita de $\log_2(n)$ etapas para ordenação. Como dividimos o vetor sempre em metades, o número total de passos é dado pela seguinte recorrência:

$$T(N) = \log_2(n) + T(\frac{n}{2})$$
$$T(N) = \log_2(n) + \log_2(\frac{n}{2}) + T(\frac{n}{4})$$
$$T(N) = \log_2(n) + \log_2(\frac{n}{2}) + \log_2(\frac{n}{4}) + ... + T(\frac{n}{n})$$
$$T(N) = \log_2(n) + \log_2(n) - 1 + \log_2(n) - 2 + ... + 1$$
$$T(N) = \sum_{i=1}^{\log_2(n)} i$$
$$T(N) = \frac{(\log_2(n) + 1) \times \log_2(n)}{2}$$
$$T(N) = \frac{\log_2^2(n) + \log_2(n)}{2}$$

Como o número de comparações realizadas em cada etapa é $\frac{n}{2}$, temos que o número total de comparações é:

$$T(N) = \frac{n}{2} \times \frac{\log_2^2(n) + \log_2(n)}{2} = \mathcal{O}(n \times \log_2^2(n))$$

1.4 Escalabilidade Forte

Segundo a lei de Amdahl, temos que o *speedup* máximo teórico é dado por:

$$S(P) = \frac{1}{\beta + \frac{1-\beta}{p}}$$

Para isso, é necessário determinar qual o valor de β, que representa a fração do código que deve ser executada sequencialmente (i.e., fração do código que não pode ser paralelizado). Neste trabalho, o valor de β é definido pela inicialização da sequência bitônica. Sendo assim, como a lei de Amdahl considera que o tamanho do problema permanece fixo, para $n = 2^{26}$ temos que a execução do algoritmo ocorre em 6,2 segundos. Desse tempo, a parte puramente sequencial é a criação do vetor de elementos, que leva em média 510 ms para ser executada. Logo, temos que $\beta = 0.08$. A Tabela 1 mostra a projeção de escalabilidade forte a partir da lei de Amdahl.

Tabela 1: Projeção de escalabilidade forte segundo a lei de Amdahl.

$N = 2^{26}$	2 CPUs	4 CPUs	8 CPUs	16 CPUs	∞ CPUs
E(p)	0,925	0,8	0,64	0,45	0

2 Paralelização e Avaliação de Desempenho

Esta seção tem como objetivo apresentar os tópicos referentes a paralelização do algoritmo *Bitonic Sort*, apresentando o modelo PRAM (*Parallel random-access machine*), estratégias de paralelização, dependência de dados e resultados experimentais que avaliam o desempenho do algoritmo.

2.1 Modelo PRAM

Conforme visto na seção 1.3, o tempo de execução sequencial é dado por:

$$T_s(N) = \mathcal{O}(n \times \log_2^2(n))$$

O modelo PRAM é permite utilizar qualquer quantidade de processadores, dessa forma vamos assumir que $p = \frac{n}{2} = \mathcal{O}(n)$. Assim, para cada etapa no algoritmo *Bitonic Sort*, cada processador irá realizar uma comparação. Como temos $\frac{n}{2}$ comparações e $\frac{n}{2}$ processadores, as comparações podem ser realizadas em tempo $\mathcal{O}(1)$.

Neste caso, a complexidade computacional paralela é dada por:

$$T(N) = 1 \times \frac{\log_2^2(n) + \log_2(n)}{2} = \mathcal{O}(\log_2^2(n))$$

Para um algoritmo ser considerado ótimo no modelo PRAM, as seguintes afirmativas devem ser satisfeitas:

1. $T_p(n)$ deve ser o menor possível: a complexidade computacional diminuiu de $\mathcal{O}(n \times \log_2^2(n))$ para $\mathcal{O}(\log_2^2(n))$;

2. $P(N)$ polinomial em n: Número de processadores utilizados é $\mathcal{O}(n)$;

3. Número de instruções em paralelo deve ser semelhante ao sequencial: Cada processador executa uma comparação em $\log_2^2(n)$ passos. Como temos $\frac{n}{2}$ processadores, o trabalho é $\frac{n}{2} \times \log_2^2(n) = \mathcal{O}(n \times \log_2^2(n))$, que é semelhante ao sequencial;

4. Por fim, o custo deve ser próximo do trabalho: O custo é o produto do número de processadores pelo tempo paralelo, portanto: $C_p(N) = p \times T_p(N) = \frac{n}{2} \times \mathcal{O}(\log_2^2(n)) = \mathcal{O}(n \times \log_2^2(n))$, que possui a mesma complexidade computacional do trabalho.

Como todas as afirmativas são satisfeitas, este algoritmo é considerado ótimo.

2.2 Estratégia de Paralelização

Para definir qual região deveria ser paralelizada, foi analisado qual trecho do programa apresentava maior impacto sobre o tempo de execução e que poderia ser paralelizado. Os dois pontos principais são: a criação do vetor de elementos e a sua respectiva ordenação. Como a criação do vetor é a parte sequencial, foi definido que a estratégia de paralelização iria explorar a ordenação dos elementos.

Após a definição de qual trecho do código seria paralelizado passou-se a definir em qual momento da ordenação a paralelização ocorreria. O Código 2 é idêntico ao Código 1 apresentado na seção 1.2, porém, possui uma diretiva *OpenMP* responsável por paralelizar o código.

Código 2: Trecho do código de ordenação paralelizado

```
for (k = n/2; k >= 1; k /= 2)
    #pragma omp parallel for collapse(2) schedule(
       guided) num_threads(NTHREADS)
    for (i = 0; i < n; i += 2*k)
        for (j = 0; j < k; j++)
            swap(elem, i+j, k);
```

O primeiro laço não pode ser paralelizado devido a problemas de dependência de dados, uma vez que para dividir novamente os grupos é necessário que todas as $\frac{n}{2}$ comparações tenham sido realizadas. Como nenhuma garantia pode ser feita com relação ao tempo em que as *threads* irão terminar, é necessário um ponto de sincronização.

Dessa forma, o segundo e terceiro laço podem ser paralelizados, e utilizar o final do segundo laço como ponto de encontro das *threads*, garantindo que todas as comparações tenham sido realizadas. Os dois laços internos podem ser paralelizados já que não existem dependências entre as comparações evitando assim condições de corrida entre as *threads*.

Sendo assim, a diretiva *OpenMP* utiliza as seguintes opções:

- *pragma parallel for*: Cria um grupo de *threads* onde cada uma delas recebe uma parte das iterações do laço;
- *collapse(2)*: Especifica que o terceiro laço também deve ser paralelizado;
- *schedule(guided)*: As *threads* recebem dinamicamente pedaços da iteração;
- *num_threads(NTHREADS)*: Especifica a quantidade de *threads* que serão criadas.

Com uma única diretiva *OpenMP* e sem nenhuma alteração no código sequencial, o algoritmo *Bitonic Sort* é paralelizado. Os resultados de desempenho são apresentados na seção 2.3.

2.3 Avaliação do Desempenho

Os experimentos foram realizados em uma máquina com 16 processadores Intel(R) Xeon(R) 3.00GHz e sistema operacional *Linux Mint* 18.1 (servidora *mumm*). Todos os resultados apresentados são uma uma média de 3 execuções. As medidas de tempo foram obtidas através da ferramenta *time* do próprio *linux*.

Overhead

O *overhead* de uma aplicação paralela pode ser calculado através da seguinte maneira:

$$T_o = p \times T_p - T_s$$

Dessa forma, a Tabela 2 mostra o *overhead* da aplicação conforme aumentamos o número de processadores bem como o tamanho do problema.

Tabela 2: *Overhead* da aplicação.

	1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
$N = 2^{29}$	15	5	23	39	79
$N = 2^{30}$	9	3	41	69	149
$N = 2^{31}$	11	6	72	156	388

É possível observar que conforme o número de processadores e o tamanho do problema o *overhead* aumenta significativamente. Esse custo pode estar associado a pontos de sincronização, divisão de trabalho e gerenciamento das *threads*.

Speedup

O *speedup* é a razão entre o tempo sequencial e o tempo paralelo. A Figura 2 apresenta o *speedup* da aplicação paralela conforme variamos o número de processadores e o tamanho do problema.

É possível notar que existe uma pequena diferença no *speedup* conforme aumentamos o tamanho do problema. Ou seja, para esta aplicação, o grau de desempenho independe do tamanho do vetor de elementos. Por outro lado, existe um ganho significativo a partir do momento que mais processadores são utilizados.

Eficiência

A eficiência é a razão entre o *speedup* e o número de processadores. Esta medida permite avaliar o grau de aproveitamento dos recursos computacionais. A Figura 3 mostra a eficiência da aplicação paralela conforme variamos o número de processadores e o tamanho do problema.

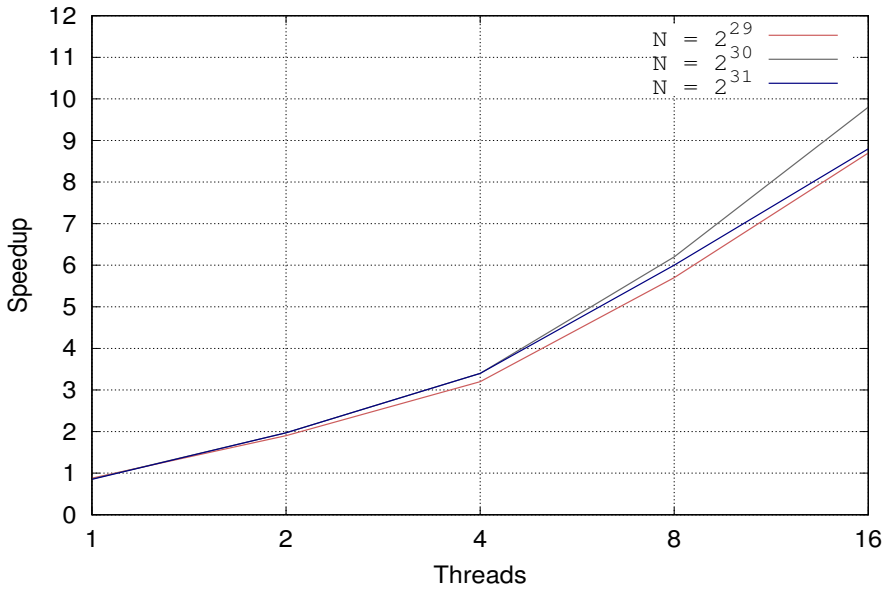

Figura 2: Speedup conforme varia o tamanho do vetor.

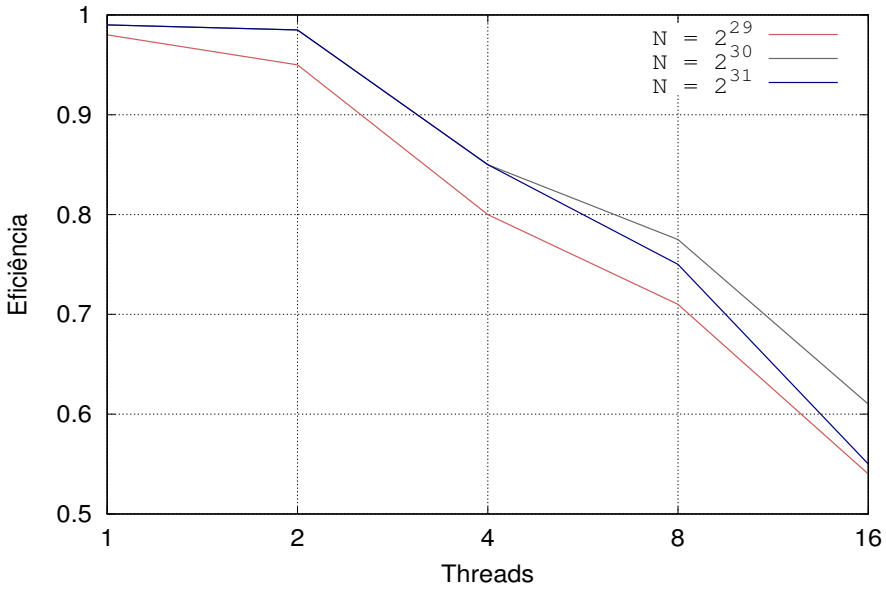

Figura 3: Eficiência conforme varia o tamanho do vetor.

É de se esperar que na maioria das aplicações a eficiência seja uma função decrescente conforme o número de processadores aumenta. Esse comportamento é reproduzido nesta aplicação, já que a eficiência decresce até 54%. Assim como para o *speedup*, a eficiência desta aplicação não apresenta grandes variações quando o tamanho do problema aumenta.

Tempo de execução

Por fim, foi comparado o tempo de execução entre o *Bitonic Sort* sequencial e paralelo utilizando 16 processadores. A Figura 4 apresenta a diferença no tempo de execução conforme aumentamos o tamanho do vetor.

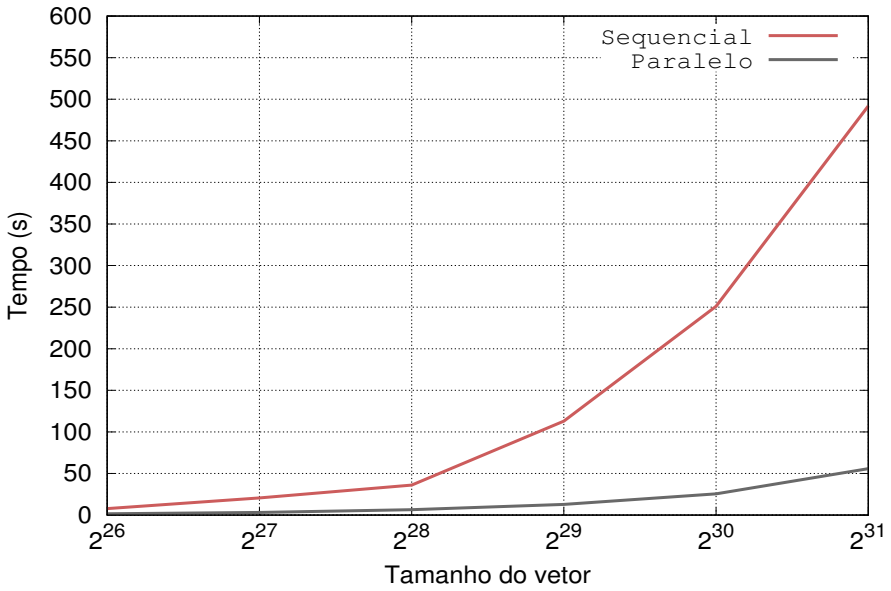

Figura 4: Comparação do tempo de execução sequencial e paralelo.

É possível observar que a aplicação obtém uma melhoria significativa no tempo de execução quando paralelizada, sendo em média 7.3 vezes mais rápida do que a implementação sequencial.

2.4 Comparação com o *Speedup* Máximo Teórico

Além das avaliações de desempenho apresentadas na seção 2.3, esta seção apresenta uma comparação dos resultados obtidos com as estimativas geradas pelas leis de Amdahl e Gustafson-Barsis.

A Figura 5 mostra uma comparação entre o *speedup* máximo teórico com o *speedup* obtido pela aplicação.

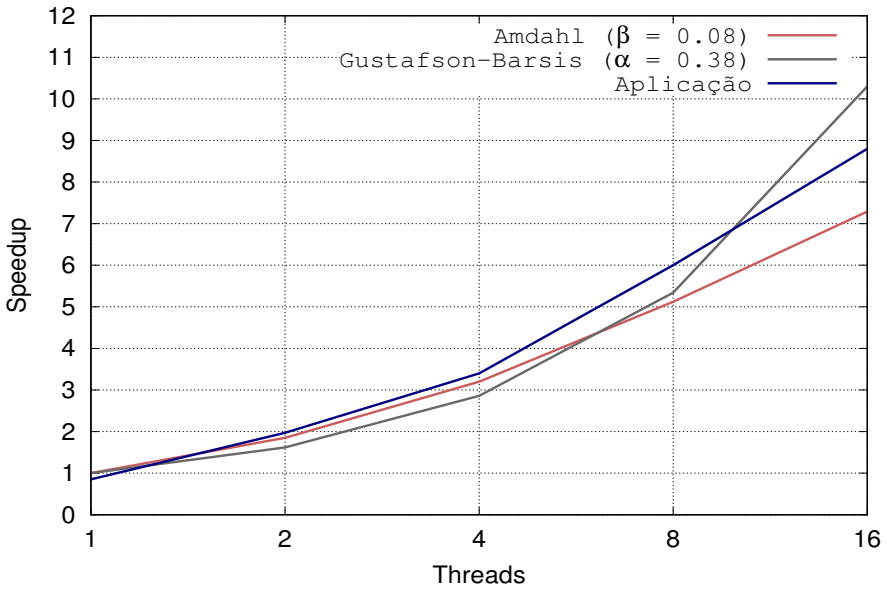

Figura 5: Comparação do *speedup* com a lei de Amdahl e Gustafson-Barsis.

É possível perceber que, como esperado, a lei de Gustafson-Barsis apresenta um *speedup* maior conforme aumenta o número de processadores. Isso acontece porque essa lei considera que o tamanho do problema irá aumentar. Além disso, o tempo sequencial é calculado a partir do tempo de execução paralelo, fazendo com que o *speedup* seja maior. Por outro lado, a lei de Amdahl possui o menor *speedup* conforme o número de processadores aumenta. Da mesma forma, a Figura 6 faz uma comparação da eficiência nos três casos.

A partir destes resultados, percebe-se que a lei de Gustafson-Barsis possui a menor queda de eficiência, como esperado.

2.5 Análise do Código

Como demonstrado na seção 2.2, a aplicação do paralelismo no código foi simples, sem a necessidade de qualquer alteração no código sequencial além da adição da diretiva *OpenMP*, demonstrado no Código 2. Este demonstra o funcionamento da ordenação bitônica de forma controlada conforme a implementação descrita na seção 1.2.

Sendo assim, através das diretivas do *OpenMP*, foi possível implementar a versão paralela do algoritmo *Bitonic Sort* de forma simples e que obteve uma melhoria significativa em comparação com a versão sequencial.

A partir dos resultados apresentados na seção 2.3, podemos perceber que a eficiência se mantém a mesma conforme aumentamos o tamanho do problema. Entretanto, a eficiência decresce conforme aumentamos o número de processa-

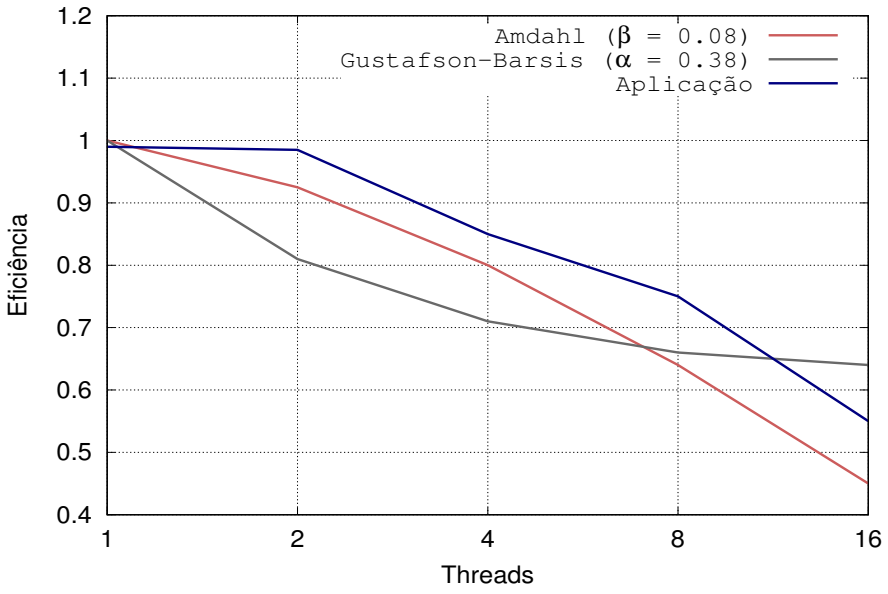

Figura 6: Comparação da eficiência com a lei de Amdahl e Gustafson-Barsis.

dores. Logo, podemos concluir que a aplicação não é escalável.

Por fim, a Figura 7 avalia os ganhos e quedas de desempenho na aplicação. Foi medido o tempo para realizar as $\frac{n}{2}$ comparações conforme aumentamos o tamanho do grupo. Como esperado, o tempo para realizar as comparações aumenta conforme aumenta o tamanho do vetor. Entretanto, o tempo para realizar as comparações para um mesmo tamanho de vetor é variável, já que existem diversos fatores que influenciam no tempo, como a sincronização, gerenciamento das *threads* e até mesmo o uso do sistema operacional.

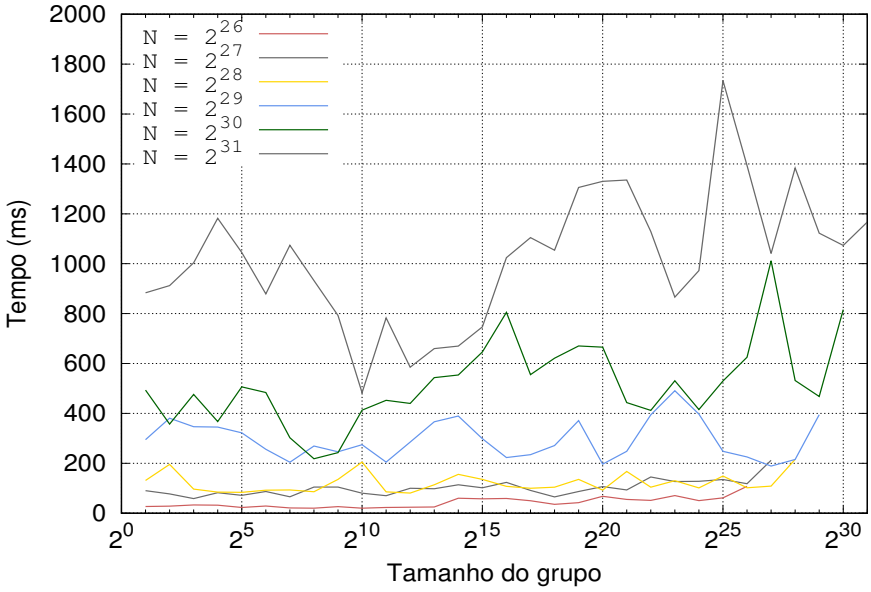

Figura 7: Tempo para comparação dos $\frac{n}{2}$ elementos conforme aumenta o tamanho do grupo.