



**Universidade de Brasília**

PROGRAMAÇÃO PARALELA

---

## Relatório do Exercício 3

---

*Autor:*

Alexandre Lucchesi Alencar

*Professor:*

George Luiz Medeiros Teodoro

20 de outubro de 2014

# 1 Introdução

Este relatório tem como objetivo apresentar os resultados obtidos a partir da execução do terceiro exercício de programação paralela [1], que consiste na implementação de uma árvore de redução de soma (Sum Tree) utilizando Message Passing Interface (MPI) <sup>1</sup>. Primeiramente, os aspectos principais do algoritmo desenvolvido são apresentados. Em seguida, apresenta-se detalhes dos artefatos de *software* desenvolvidos, o processo de geração do arquivo de entrada, dois tipos distintos de função de espalhamento implementadas e as técnicas utilizadas para medir os tempos de execução da aplicação, evitar *deadlocks* e evitar problemas relacionados a *estouros de pilha*. Por fim, é realizada uma análise de desempenho comparando os tempos de execução do algoritmo em diversas configurações, isto é, variando-se o número de processos e a quantidade de números de ponto-flutuante a serem somados. São apresentadas como métricas: o *speedup* obtido através da criação de múltiplos processos, a eficiência da aplicação e uma análise de escalabilidade. O código-fonte completo deste trabalho, incluindo os arquivos L<sup>A</sup>T<sub>E</sub>X que compõem este relatório, estão publicamente disponíveis no GitHub <sup>2</sup>.

## 2 O Algoritmo

Conforme acordado em sala de aula [2], a especificação original do projeto [1] foi alterada. A nova especificação determina que o programa deve funcionar apenas para um número de elementos e processos que sejam expoentes de 2 ( $2^i \mid i \in \{0, 1, 2, 3, \dots, N\}$ ). Além disso, deve-se assumir a possibilidade do tamanho do vetor de entrada ser maior que a quantidade de processos, fazendo com que mais de um número seja atribuído ao mesmo processo. Dessa forma, os processos devem fazer “reduções intermediárias”, isto é, os processos devem *obrigatoriamente* receber um ou mais números provenientes de outro processo (via `MPI_Recv()`) e realizar reduções locais até que fiquem com apenas um número. Nesse momento, no qual todos os processos possuem apenas um número, deve-se executar o algoritmo original (Sum Tree) para se obter o resultado final da soma.

O algoritmo que realiza as somas intermediárias foi encapsulado na função `reduce_sumtree()`, cujo um trecho de código é apresentado na Seção 2.5; e o

---

<sup>1</sup>Neste trabalho, utilizou-se durante o desenvolvimento a versão mais recente do OpenMPI para Mac OS X instalada a partir do utilitário Homebrew. Durante a execução dos testes de desempenho, utilizou-se a versão mais recente do OpenMPI para o Ubuntu instalada via `apt-get`.

<sup>2</sup><https://github.com/alexandreLucchesi/parallel-programming-ex03>

algoritmo Sum Tree foi encapsulado na função `reduce()`. A Figura 1 apresenta a sequência de etapas realizada em `reduce_sumtree()` para executar as reduções.

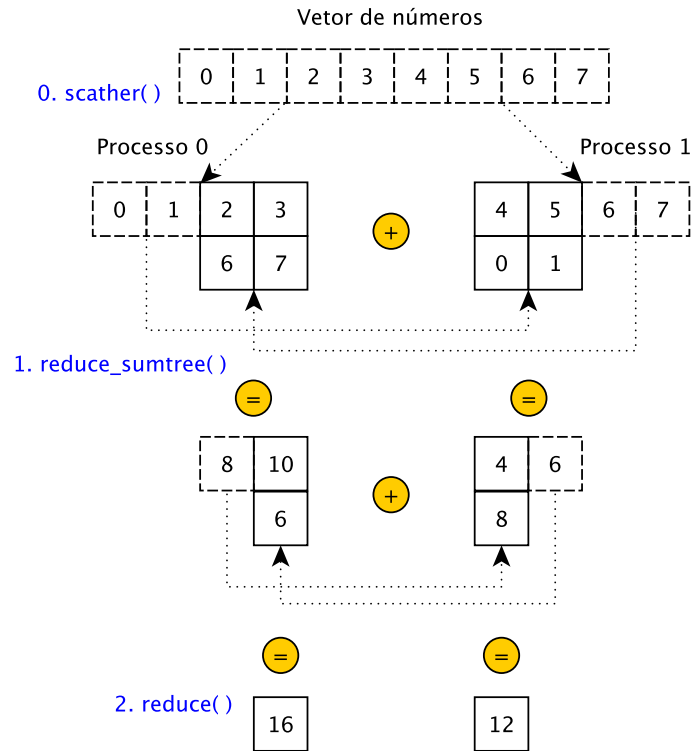


Figura 1: Exemplo de uso do algoritmo utilizado para fazer as reduções intermediárias usando 8 números e 2 processos.

O exemplo ilustra um vetor de entrada contendo 8 números <sup>3</sup>, cuja soma deve ser calculada por 2 processos: o Processo 0 (P0) e o Processo 1 (P1). Inicialmente, cada processo chama a função de espalhamento `scather()` (vide Seção 2.3) para receber os números que deve calcular. Internamente, na função `scather()`, o processo com *rank* 0 é o responsável por dividir o vetor de entrada entre os processos. No exemplo, os quatro primeiros números são atribuídos a P0 e os quatro últimos a P1. Então, todos os processos chamam a função `reduce_sumtree()`. Essa função verifica que a quantidade

<sup>3</sup>Por motivos de simplificação, utilizou-se números inteiros no exemplo, mas o código foi implementando usando números de ponto-flutuante.

de elementos é maior que um e decide, portanto, que os processos devem realizar reduções intermediárias. Assim, ambos P0 e P1 trocam metade de seus números entre si (via `MPI_Send()` e `MPI_Recv()`) e executam somas locais. Por fim, a função `reduce_sumtree()` é chamada recursivamente até que a quantidade de elementos seja igual um. Nesse caso, `reduce()` é chamada.

As seções a seguir descrevem em detalhes os artefatos de *software* que foram desenvolvidos durante este trabalho, o processo de geração do arquivo de dados, as duas versões de função de espalhamento que foram implementadas e as técnicas utilizadas para medir os tempos de execução da aplicação, evitar *deadlocks* e evitar problemas relacionados a *estouros de pilha*.

## 2.1 Artefatos Desenvolvidos

O programa C desenvolvido possui duas funcionalidades principais: (i) geração de um arquivo de dados contendo um número arbitrário de valores de ponto-flutuante; (ii) e processamento de um arquivo de dados retornando a soma dos elementos e o tempo de execução do algoritmo. Além do programa principal (`main.c`), foram desenvolvidos dois *scripts bash*: um para facilitar a execução do programa principal (`run.sh`), encapsulando a chamada ao `mpiexec` (ou `mpirun`), e outro para automatizar os testes da aplicação (`test.sh`). Esses artefatos são descritos a seguir.

- **main.c**: programa em C contendo o código-fonte da aplicação. Após compilado com o `mpicc` (vide `Makefile`), pode ser executado com o *script* `run.sh` passando-se o número de processos e o arquivo de dados. O `run.sh` executa o programa usando `mpiexec` e passando esses dois argumentos, que são recebidos via `scanf()`. A saída do programa é uma linha contendo dois números: o primeiro representa o resultado da soma dos números de ponto-flutuante e o segundo, o tempo de execução do algoritmo de redução em milissegundos (desconsiderando o tempo de entrada de dados).
- **test.sh**: *script* desenvolvido para automatizar os testes da aplicação. Recebe como entrada 2 argumentos, em ordem:
  - **max\_numbers**: número máximo de processos. O *script* varia o número de processos de  $2^{20}$  até  $2^{\text{max\_numbers}}$ .
  - **max\_runs**: número máximo de vezes em que o programa deve ser executado em uma mesma configuração.

## 2.2 Geração do Arquivo de Dados

Para a geração de quantidades configuráveis de números de ponto-flutuante em um formato apropriado para servir de entrada para o programa, pode-se executar o binário proveniente do processo de compilação diretamente (sem utilizar `mpiexec`). Por exemplo, para gerar um arquivo de dados `numbers.dat` contendo 64 elementos, basta executar `sumtree` passando-se a *flag* `-gen`, conforme descrito a seguir:

```
$ make          # Gera o binário com nome: 'sumtree'.
$ ./sumtree -gen numbers.dat 64
```

Uma outra opção disponível é a `--help`, que exhibe informações de uso da aplicação.

## 2.3 Função de “Espalhamento”

Para distribuir os dados entre os diferentes processos, foram criadas duas funções:

```
void scather(int my_rank, int comm_sz,
             unsigned int *my_count, float **my_nums);
void scather_intercalate(int my_rank, int comm_sz,
                        unsigned int *my_count, float **my_nums);
```

Uma sempre pode ser utilizada no lugar da outra preservando-se a corretude do código <sup>4</sup> (note que a assinatura é a mesma). A única diferença entre as duas está na política de atribuição dos números aos processos. Internamente, o processo com `my_rank` igual a zero é sempre o responsável por ler o arquivo de dados e dividir os números entre os `comm_sz` processos, retornando em `my_count` e `my_nums` a quantidade de elementos e os números, respectivamente.

No caso da `scather()`, a quantidade total de números (lida do arquivo de entrada) é dividida pelo número de processos (`comm_sz`) e o resultado (`res`) obtido é utilizado para atribuir sequencialmente os números aos processos, ou seja, o processo 0 recebe os números indexados pelo intervalo  $[0, res - 1]$ , o processo 1 recebe  $[res, 2 \times res - 1]$ , e assim por diante.

Por outro lado, a `scather_intercalate()` intercala os números entre os processos, varrendo o vetor e atribuindo o elemento no índice  $i$  ao processo

---

<sup>4</sup>O resultado final da redução pode ser diferente, dado que a operação de soma não é associativa para números de ponto-flutuante nos computadores.

$i \bmod comm\_sz$ .

## 2.4 Medindo o Tempo de Execução

Para medir o tempo máximo de execução da aplicação de forma precisa, utilizou-se *barreiras*. Com uma chamada à função `MPI_Barrier()` antes de `reduce_sumtree()`, garante-se que todos os processos começam a executar o algoritmo de redução “ao mesmo tempo”. Dessa forma, coletando-se os tempos do “relógio de parede” no processo 0 imediatamente após a barreira, é possível calcular o tempo total de execução da aplicação.

Utilizou-se a função `gettimeofday()` (disponível em “`sys/time.h`”) para se obter os tempos de início e término e calculou-se usando aritmética simples o intervalo de tempo de execução em milisegundos.

## 2.5 Evitando *Deadlocks*

Na implementação do MPI utilizada, ambas as primitivas `Send()` e `Recv()` são “blocantes”. Isso significa que cuidado adicional deve ser tomado para que não ocorram *deadlocks*. O trecho de código 1 apresenta como a função `reduce_sumtree()` foi projetada para evitar a ocorrência de *deadlocks*.

Código 1: Ordem das primitivas MPI\_Send() e MPI\_Recv() na função reduce\_sumtree().

```
1  ...
2  if (my_rank % 2 == 0) {
3      int dst = my_rank + 1;
4
5      // Copy second half of 'nums[]' to 'my_nums'.
6      memcpy(my_nums, nums + qty, qty * sizeof(float));
7
8      // Send first half of 'nums[]' to 'dst'.
9      MPI_Send(nums, qty, MPI_FLOAT, dst, 2, MPI_COMM_WORLD);
10
11     // Receive second half of his 'nums[]' into 'his_nums'.
12     MPI_Recv(his_nums, qty, MPI_FLOAT, dst, 2, MPI_COMM_WORLD,
13              MPI_STATUS_IGNORE);
14 } else {
15     int dst = my_rank - 1;
16
17     // Copy first half of 'nums[]' to 'my_nums'.
18     memcpy(my_nums, nums, qty * sizeof(float));
19
20     // Receive first half of his 'nums[]' into 'his_nums'.
21     MPI_Recv(his_nums, qty, MPI_FLOAT, dst, 2, MPI_COMM_WORLD,
22              MPI_STATUS_IGNORE);
23
24     // Send second half of 'nums[]' to 'dst'.
25     MPI_Send(nums + qty, qty, MPI_FLOAT, dst, 2, MPI_COMM_WORLD);
26 }
```

Se as primitivas MPI\_Recv() e MPI\_Send() aparecem na mesma ordem no if e no else, os processos entram em *deadlock*. Ao colocá-los de forma alternada, garante-se que para cada MPI\_Send() sempre existirá um MPI\_Recv() e vice-versa.

## 2.6 Evitando “Estouro de Pilha”

O padrão C99 permite a declaração de variáveis de forma intercalada com as instruções. Dessa forma, é possível, por exemplo, ler um número inteiro `n` via `scanf()` e logo em seguida declarar um vetor de tamanho `n`. Apesar desse recurso ser bastante útil, permitindo um controle mais granular do escopo das variáveis e aumentando consideravelmente a legibilidade do código, ele pode levar a comportamentos inesperados. Isso ocorre porque essas variáveis são alocadas no seguimento de pilha, cujo tamanho máximo pode variar entre diferentes sistemas operacionais. Diante disso, existem duas abordagens possíveis para contornar o problema:

1. Forçar um tamanho de pilha maior que o padrão. Existem duas formas de se fazer isso: (i) modificando o tamanho padrão alocado pelo sistema operacional — no Mac OS X isso pode ser feito de forma global (por sessão no *shell*) com o comando `ulimit -s hard`; ou (ii) instruindo o *linker* durante a compilação (`gcc -Wl,-stack_size,<stack_size> ...`). A opção (ii) é mais extensível.
2. Alocar a memória dinamicamente no *heap* (via `malloc()`), cujo limite é determinado pelo tamanho da memória virtual do sistema.

Apesar da alocação no *heap* ser um pouco mais lenta do que a alocação na pilha, optou-se por essa abordagem nas partes do programa que poderiam causar estouro de pilha, como por exemplo na função `read_data()`. Essa função, apresentada a seguir, é utilizada internamente nas funções `scather()` e `scather_intercalate()` para retornar um ponteiro para o vetor de `floats` que será atribuído a cada processo e sua respectiva quantidade de elementos:

```
void read_data(unsigned int *count, float **nums) {
    scanf("%u", count); // Get numbers count.
    (*nums) = (float *) malloc((*count) * sizeof(float));
    for (unsigned int i = 0; i < (*count); i++) {
        scanf("%f", (*nums) + i);
    }
}
```

## 3 Resultados

Executou-se o *script* de testes (`test.sh`) passando-se como argumentos: `25 max_numbers`, para executar testes variando-se a quantidade de elementos de



$2^{20}$  à  $2^{25}$ ; e 5 `max_runs`, para se realizar 5 execuções em cada configuração.

O `script test.sh` gerou como saída arquivos de dados contendo os números de ponto-flutuante para serem usados nos testes (extensão `.dat`) e arquivos no formato CSV contendo as tabelas que compõem este relatório. Cada tabela é indexada pela quantidade de elementos (no exemplo acima, de  $10^{20}$  à  $10^{25}$ ) e o número da execução (no exemplo acima, de 1 a 5). Metade possui o prefixo `sum_n`, onde `n` é o número de processos utilizado, representando os valores aproximados da soma calculados. A outra metade possui prefixo `time_n`, e contém os valores dos tempos de execução. Em poucas palavras, tem-se 2 arquivos para cada número de processos contendo as duas saídas do programa: o valor estimado da soma e o tempo de execução do algoritmo.

As seções a seguir apresentam o *hardware* utilizado e os resultados dos testes, apresentando métricas de *speedup*, eficiência e escalabilidade. Os resultados são apresentados sob a forma de gráficos. As tabelas com os dados exatos que deram origem a esses gráficos estão anexadas ao final do documento.

### 3.1 *Hardware* Utilizado

- Processador: AMD FX(tm)-8350 Eight-Core Processor
- Velocidade por *core*: 1406.2 MHz
- Número de processadores: 1
- Número de *cores*: 8 (máx. 8)
- Número de *threads*: 8 (máx. 8)
- L1 Data cache: 8 x 16 KBytes, 4-way set associative, 64-byte line size
- L1 Instruction cache: 4 x 64 KBytes, 2-way set associative, 64-byte line size
- L2 cache: 4 x 2048 KBytes, 16-way set associative, 64-byte line size
- L3 cache: 8 MBytes, 64-way set associative, 64-byte line size

### 3.2 *Speedup*

A Figura 2 apresenta um gráfico em que cada curva relaciona o *speedup* obtido, o número de processos criados e a quantidade de elementos somados

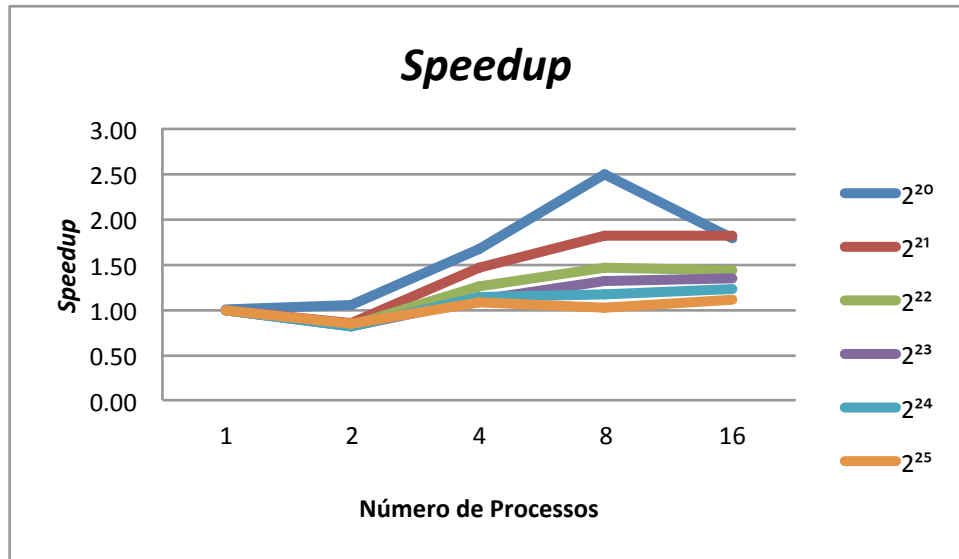


Figura 2: Gráfico apresentando a relação entre o *speedup*, o número de processos e a quantidade de elementos somados na redução.

na redução. De uma forma geral, observa-se que o *speedup* aumenta conforme a quantidade de processos cresce. A excessão ocorre apenas quando se utiliza dois processos, onde perde-se desempenho para quase todos os tamanhos de entrada <sup>5</sup>.

Isso ocorre porque o *overhead* envolvido na criação de dois processos e, principalmente, na comunicação entre os processos é maior do que os possíveis ganhos advindos da concorrência entre os mesmos.

### 3.3 Eficiência

A Figura 3 apresenta um gráfico em que cada curva relaciona a *eficiência* obtida, o número de processos criados e a quantidade de elementos somados na redução. Observa-se que a *eficiência* diminui drasticamente ao se utilizar dois processos ao invés de um. A partir daí, a eficiência continua diminuindo conforme a quantidade de processos cresce. A taxa de queda é um pouco mais sutil quando comparada à de um processo para dois, porém continua significativa, evidenciando que a eficiência da aplicação tende a ser muito ruim com um número de processos grande.

<sup>5</sup>Com  $2^{20}$  elementos houve um ganho de desempenho mínimo, com o *speedup* no valor de 1.04.

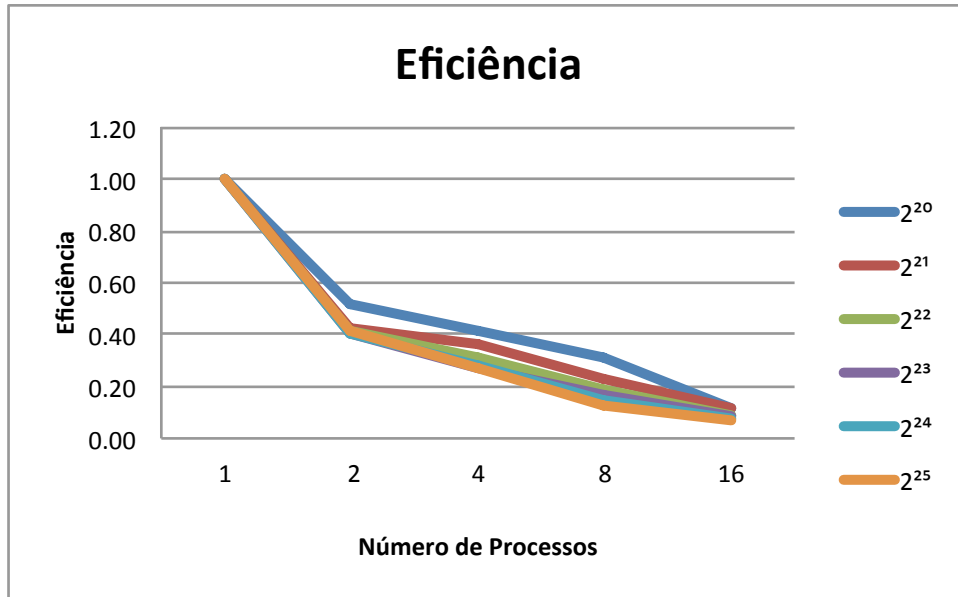


Figura 3: Gráfico apresentando a relação entre o *speedup*, o número de processos e a quantidade de elementos somados na redução.

### 3.4 Escalabilidade

Conforme ilustrado na Figura 3 e observando as tabelas de tempo de execução ao final deste documento, nota-se que a eficiência cai em uma taxa muito maior que os tempos de execução. Dessa forma, conclui-se que a aplicação não possui uma boa *escalabilidade forte*.

De forma similar, não se tem também uma boa *escalabilidade fraca*, conforme apresentado na Figura 4, que relaciona a eficiência, o número de processos, e a quantidade de elementos somados na redução. Não está explícito no gráfico, mas para cada número de processos obteve-se o valor da eficiência incrementando-se proporcionalmente o tamanho do problema (ou número de elementos a serem somados). Sendo assim, para 1 processo, utilizou-se  $2^{20}$  elementos; para 2 processos,  $2^{21}$  elementos; para 4 processos,  $2^{22}$  elementos; e assim por diante.

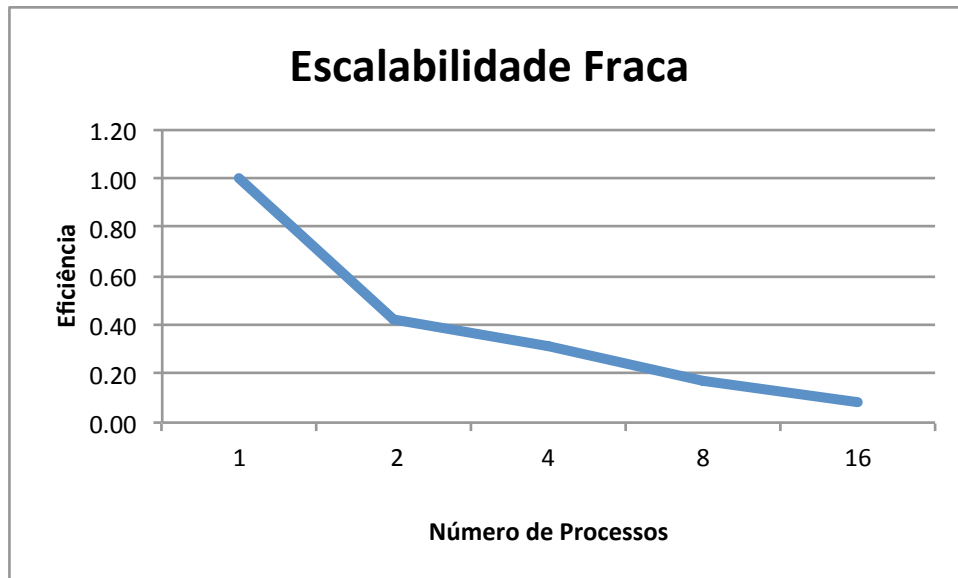


Figura 4: Gráfico apresentando a relação entre a eficiência e o número de processos, aumentando-se proporcionalmente o tamanho do problema.

## 4 Conclusão

Este trabalho possibilitou uma maior compreensão acerca de modelos de programação baseados em trocas de mensagem, em particular, que utilizam primitivas `Send()` e `Recv()`, através da implementação de uma árvore de redução de soma usando MPI.

O algoritmo desenvolvido possibilita a execução de reduções usando um número grande de elementos, utilizando as primitivas citadas anteriormente para realizar somas intermediárias até que cada processo tenha um elemento (vide Seção 2). Nesse momento, uma última redução é realizada utilizando-se o algoritmo Sum Tree. Uma análise de desempenho da aplicação evidenciou um crescimento no *speedup* na medida em que a quantidade de processos cresce, apresentando um valor máximo de 2.50 ao se utilizar 8 processos para somar  $2^{20}$  elementos. No entanto, a eficiência e a escalabilidade apresentaram taxas de decaimento significativas com o aumento da quantidade de processos, revelando limitações na aplicação.

Por fim, é válido ressaltar que o o *design* da aplicação (incluindo o *script test.sh*) permite variar de forma fácil as condições de teste, permitindo a reprodução dos experimentos apresentados neste trabalho e facilitando a experimentação com novas configurações.

## Referências

- [1] G. L. M. Teodoro. Programação paralela, exercício de programação 03, sum tree, September 2014.
- [2] G. L. M. Teodoro. Programação paralela, notas de aula, September 2014.

# Anexo I

**Sequencial (Nro. de Execuções x Nro. de Elementos)**

Tempos de Execução								
	1	2	3	4	5	Média	D. Padrão	Coef. de Variação
2 <sup>20</sup>	5	5	5	5	5	5	0	0.00%
2 <sup>21</sup>	9	9	9	9	9	9	0	0.00%
2 <sup>22</sup>	18	18	18	17	17	17.6	0.4898979	2.78%
2 <sup>23</sup>	35	35	35	35	35	35	0	0.00%
2 <sup>24</sup>	69	69	70	69	70	69.4	0.4898979	0.71%
2 <sup>25</sup>	138	138	139	138	138	138.2	0.4	0.29%

**2 Processos (Nro. de Execuções x Nro. de Elementos)**

Tempos de Execução								
	1	2	3	4	5	Média	D. Padrão	Coef. de Variação
2 <sup>20</sup>	5	5	4	5	5	4.8	0.4	8.33%
2 <sup>21</sup>	10	12	10	11	10	10.6	0.8	7.55%
2 <sup>22</sup>	21	21	21	22	22	21.4	0.4898979	2.29%
2 <sup>23</sup>	46	44	43	43	43	43.8	1.1661904	2.66%
2 <sup>24</sup>	86	86	86	85	85	85.6	0.4898979	0.57%
2 <sup>25</sup>	166	168	168	166	165	166.6	1.2	0.72%

**4 Processos (Nro. de Execuções x Nro. de Elementos)**

Tempos de Execução								
	1	2	3	4	5	Média	D. Padrão	Coef. de Variação
2 <sup>20</sup>	3	3	3	3	3	3	0	0.00%
2 <sup>21</sup>	6	7	6	6	6	6.2	0.4	6.45%
2 <sup>22</sup>	13	15	15	15	13	14.2	0.9797959	6.90%
2 <sup>23</sup>	32	30	30	33	34	31.8	1.6	5.03%
2 <sup>24</sup>	61	63	60	61	61	61.2	0.9797959	1.60%
2 <sup>25</sup>	129	128	137	126	128	129.6	3.8262253	2.95%

**8 Processos (Nro. de Execuções x Nro. de Elementos)**

Tempos de Execução								
	1	2	3	4	5	Média	D. Padrão	Coef. de Variação
2 <sup>20</sup>	2	2	2	2	2	2	0	0.00%
2 <sup>21</sup>	5	5	5	5	5	5	0	0.00%
2 <sup>22</sup>	12	12	12	12	12	12	0	0.00%
2 <sup>23</sup>	26	26	27	27	26	26.4	0.4898979	1.86%
2 <sup>24</sup>	59	59	57	58	62	59	1.6733201	2.84%
2 <sup>25</sup>	137	142	135	136	132	136.4	3.2619013	2.39%

**16 Processos (Nro. de Execuções x Nro. de Elementos)**

Tempos de Execução								
	1	2	3	4	5	Média	D. Padrão	Coef. de Variação
2 <sup>20</sup>	3	3	3	2	3	2.8	0.4	14.29%
2 <sup>21</sup>	5	5	5	5	5	5	0	0.00%
2 <sup>22</sup>	13	12	12	13	12	12.4	0.4898979	3.95%
2 <sup>23</sup>	26	26	26	26	27	26.2	0.4	1.53%
2 <sup>24</sup>	56	60	62	54	52	56.8	3.7094474	6.53%
2 <sup>25</sup>	124	127	123	124	123	124.2	1.4696938	1.18%

**Speedup (Nro. de Processos x Nro. de Elementos)**

	1	2	4	8	16
2 <sup>20</sup>	1.00	1.04	1.67	2.50	1.79
2 <sup>21</sup>	1.00	0.85	1.45	1.80	1.80
2 <sup>22</sup>	1.00	0.82	1.24	1.47	1.42
2 <sup>23</sup>	1.00	0.80	1.10	1.33	1.34
2 <sup>24</sup>	1.00	0.81	1.13	1.18	1.22
2 <sup>25</sup>	1.00	0.83	1.07	1.01	1.11

**Eficiência (Nro. de Processos x Nro. de Elementos)**

	1	2	4	8	16
2 <sup>20</sup>	1.00	0.52	0.42	0.31	0.11
2 <sup>21</sup>	1.00	0.42	0.36	0.23	0.11
2 <sup>22</sup>	1.00	0.41	0.31	0.18	0.09
2 <sup>23</sup>	1.00	0.40	0.28	0.17	0.08
2 <sup>24</sup>	1.00	0.41	0.28	0.15	0.08
2 <sup>25</sup>	1.00	0.41	0.27	0.13	0.07

**Escalabilidade Fraca**

1	2	4	8	16
1.00	0.42	0.31	0.17	0.08