**3﷽﷽﷽﷽﷽﷽﷽﷽3﷽﷽﷽﷽﷽﷽﷽﷽3﷽﷽﷽﷽﷽﷽﷽﷽3﷽﷽﷽﷽﷽﷽﷽﷽**

Disciplina: Programação Paralela, código 316342

Professor: George Luiz Medeiros Teodoro

Aluno: Jefferson Chaves Gomes – 14/0189581

**Exercício de Programação 03**

Índice

1. Introdução 3

2. Requisitos à implementação 3

3. Objetivos 4

4. Entrada esperada 4

5. Descrição do Algoritmo Desenvolvido 5

5.1. Estruturas criadas 5

5.2. Leitura e validação de dados de entrada 5

5.3. Estruturação de dados de entrada 7

5.4. Processamento dos dados 9

6. Implementação MPI 11

7. Instrução de Compilação 11

8. Instrução de Execução 11

9. Ambiente de Execução dos Testes 12

10. Descrição dos testes 12

11. Análise de Resultados 13

11.1. Tempos de Execução 13

11.2. Análise de *Speedup* 14

11.3. Análise de Eficiência 15

12. Código Completo 16

# Introdução

Neste exercício foram comparados dados estatísticos no que se refere ao tempo de processamento de um programa quando este é executado de maneira sequencial e paralela para calcular a soma de *n* números através da troca de mensagens entre *np* processos utilizando uma estrutura de árvore de redução. A ilustração abaixo mostra a estratégia de soma quando temos oito números a serem somados (*n* = *8*) e oito processos (*np* = 8) onde cada um é possuidor de um número para ser somado:

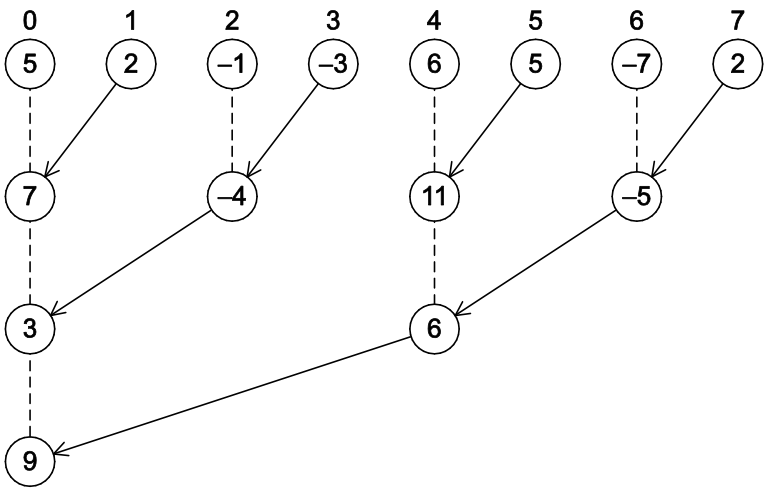


Ilustração 1

Na ilustração acima a comunicação entre os processos ocorre da seguinte forma:

* Processo 1 envia para 0, 3 envia para 2, 5 envia para 4, e 7 envia para 6;
* Processos 0, 2, 4, e 6 somam os valores recebidos no passo acima;
* Processos 2 e 6 enviam seus novos valores respectivamente para os processos 0 e 4;
* Processos 0 e 4 somam os valores nos seus novos valores;
* Processo 4 envia seu novo valor para o processo 0;
* Processo 0 soma o valor recebido no seu novo valor.

# Requisitos à implementação

Para a implementação deste exercício os itens abaixo foram dados como requisitos:

* A implementação deve fazer uso exclusivamente de C ou C++;
* O processamento distribuído deve ser aplicado através do uso da especificação MPI (Message Passing Interface);
* Para a troca de mensagens entre os processos apenas as funções *MPI\_Send()* e *MPI\_Recv()* deverão ser utilizadas;
* A função *MPI\_Reduce()* não deve ser usada;
* O desenvolvimento do algoritmo deve obedecer três etapas:

1. Distribuição de elementos de entrada: os elementos a serem somados devem ser recebidos através da entrada padrão *stdin*, para então serem distribuídos entre os processos;
2. Simulação da árvore de redução: Para configurar a árvore de redução, um dos operandos de cada soma sempre deverá ser originado de outro processo. Desta forma, os processos deverão realizar troca de mensagens entre si somando os valores recebidos àqueles já de posse de um determinado processo, até que cada processo possua somente um elemento;
3. Árvore de redução: Agora que cada processo possui somente um único elemento, os processos deverão comunicar-se entre si de forma a acumular a soma em um único processo final, conforme Ilustração 1.

* O resultado do programa deve ser impresso na saída padrão *stdout*.

# Objetivos

O objetivo geral deste exercício é aplicar os conhecimentos adquiridos em sala de aula a fim obter ganhos de performance com uso de processamento distribuído através do uso de *MPI,* gerando com a execução do programa dados necessários para avaliar a performance, *speedup* e eficiência.

# Entrada esperada

O programa deverá receber, após sua chamada, três parâmetros como entrada, sendo o primeiro, um texto que representara o tipo de saída que deve ser impressa, são elas:

* **time**: para exibir somente o tempo (em milisegundos) necessário ao processamento;
* **sum**: para que o programa exiba como saída apenas o valor da soma dos números, com duas casas decimais;
* **all**: para imprimir em duas linhas os valores de tempo e soma respectivamente.

Já o segundo, deve ser um número inteiro *n* que representa a quantidade de números desejados para executar o processo de soma.

O terceiro, consiste no conjunto dos números a serem distribuídos e somados.

# Descrição do Algoritmo Desenvolvido

De uma forma geral será descrito como a solução foi desenvolvida, apontando pontos importantes a fim de esclarecer detalhes do funcionamento do programa.

# Estruturas criadas

Foi desenvolvida uma estrutura visando armazenar o resultado do processamento dos dados. A estrutura foi chamada de *Result* e tem a função de armazenar o tempo gasto no processamento bem como o resultado obtido no cálculo da soma dos números informados. Segue abaixo o trecho do código referente a estrutura citada:

|  |
| --- |
| **struct** Result {  **double** processTime;  **long** **double** sum;  }; |

# Leitura e validação de dados de entrada

Uma função responsável pela validação da quantidade de parâmetros passados ao programa foi criada. Esta função foi chamadade *checkInputParams* e é exibida a seguir:

|  |
| --- |
| **void** **checkInputParams**(**int** argc, **char** \*\*argv) {  **if** (argc != 1) {  printUsage();  **exit**(EXIT\_FAILURE);  }  } |

A entrada de dados ao programa foi implementada de forma a permitir ao usuário informar novamente (5 tentativas) um dado de entrada se o mesmo for considerado inválido pelas rotinas de leitura e validação. A seguir, a função responsável permitir ou não que usuário continue tentado informar dados de entrada corretos:

|  |
| --- |
| **void** **allowAnotherAttempt**(**const** std::string msg, **int** &badAttempts) {  **if** (++badAttempts >= MAX\_ATTEMPTS) {  printUsage();  **exit**(EXIT\_FAILURE);  }  std::cout << msg << BREAK\_LINE;  std::cout << "Try again (you can try more " << MAX\_ATTEMPTS - badAttempts << " times)." << BREAK\_LINE;  } |

Para ler e validar o tipo de saída desejada pelo usuário, uma função chamada *readOutputType* foi implementada e seu código é ilustrado abaixo:

|  |
| --- |
| OutputType **readOutputType**() {  **int** badAttempts = 0;  std::string outputOption = "NONE";  std::cin >> outputOption;  **while** (stringToOutputType(outputOption) == *NONE*) {  allowAnotherAttempt(  "Error: The output type must be equal then [sum | time | all]."  , badAttempts);  std::cin.clear();  std::cin >> outputOption;  }  **return** stringToOutputType(outputOption);  } |

Já para ler e validar o dado de entrada referente a quantidade de números a serem somados, a função *readNumbersCount* foi criada e é exibida abaixo:

|  |
| --- |
| **int** **readNumbersCount**() {  **int** badAttempts = 0;  **int** numberCount;  std::cin >> numberCount;  **while** (std::cin.bad() || std::cin.fail() || numberCount < 2) {  allowAnotherAttempt(  "Error: The amount of numbers must be a valid integer greater then 1."  , badAttempts);  std::cin.clear();  std::cin.ignore(INT\_MAX, '\n');  std::cin >> numberCount;  }  **return** numberCount;  } |

Agora para ler o conjunto de números referente a quantidade de números obtida acima, foi criada uma função chamada *readNumbersArray*. Seu código é o que segue:

|  |
| --- |
| **void** **readNumbersArray**(**float**\* numbersArray, **const** **int** numbersCount) {  **int** badAttempts = 0;  **double** number = 0.0;  **for** (**long** i = 0; i < numbersCount; i++) {  std::cin >> number;  **while** (std::cin.bad() || std::cin.fail()) {  allowAnotherAttempt(  "Error: The number must be a integer or a float [5 | 5.0]."  , badAttempts);  std::cin.clear();  std::cin.ignore(INT\_MAX, '\n');  std::cin >> number;  }  numbersArray[i] = number;  }  } |

# Estruturação de dados de entrada

Caso o número de processos informado pelo usuário seja igual ao número 1 (um), a distribuição dos dados de entrada não é necessária, pois a solução proposta fará a soma do conjunto de números de maneira sequencial com uma iteração simples onde todos os números são somados. A avalição do tempo de processamento para casos sequenciais foi realizada através do uso da função *gettimeofday* como foi sugerido no exercício de programação EP02. O código referente ao processamento sequencial é exibido abaixo:

|  |
| --- |
| **void** **startSerialProcess**(**const** **float** \*numbersArray, **const** **int** numbersCount) {  **double** totalSum = 0;  timeval startTime, endTime;  **gettimeofday**(&startTime, NULL);  **for** (**int** i = 0; i < numbersCount; ++i) {  totalSum += numbersArray[i];  }  **gettimeofday**(&endTime, NULL);  **double** time = (  (endTime.tv\_sec \* 1000 + endTime.tv\_usec) –  (startTime.tv\_sec \* 1000 + startTime.tv\_usec  ));  result.sum = totalSum;  result.processTime = time;  } |

Aos casos os quais o número de processos é maior que 1 (um), é necessário que a solução distribua os dados entre os processos (esta fase da solução é referente a ETAPA-01 proposta pelo exercício), para isto foi utilizado o tipo de escalonamento de laços *schedule static*, ou seja, as iterações são atribuídas aos processos antes que o seu laço seja executado. A distribuição dos elementos foi executada de forma cíclica como ilustrado abaixo:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Número de processos | Distribuição Cíclica | | | |
| 1º | 1 | 4 | 7 | 10 |
| 2º | 2 | 5 | 8 | 11 |
| 3º | 3 | 6 | 9 | 12 |

A função chamada *startDispatchElements* tem o objetivo de enviar número a número aos processos existentes bem como a informação de quantos números foram enviados a cada um dos processos, ou seja, esta função faz duas chamadas à função *MPI\_Send()* fazendo uso de duas *TAGS*, uma para identificar o envio de números (GLOBAL\_LISTEN\_TAG) e outra para identificar o envio da informação de quantos números foram enviados a cada processo (BLOCKS\_LISTEN\_TAG).

A execução desta função é de responsabilidade do processo de *RANK* igual a zero (PROCESS\_MASTER) o qual é responsável pela leitura de todos os dados de entrada.

O trecho do código onde é realizada a distribuição dos números entre os processos é mostrado a seguir:

|  |
| --- |
| **void** **startDispatchElements**(**const** **int** numbersCount, **const** **float** \*numbersArray) {  **int** \*blockSizesArray = **new** **int**[processCount];  std::fill\_n(blockSizesArray, processCount, 0);  **for** (**int** i = 0, processDest = 0; i < numbersCount; i++) {  **if** (processDest == PROCESS\_MASTER) {  processPartNumbersVector.push\_back(numbersArray[i]);  } **else** {  **MPI\_Send**(&numbersArray[i], 1, MPI\_FLOAT, processDest, GLOBAL\_LISTEN\_TAG, MPI\_COMM\_WORLD);  }  blockSizesArray[processDest]++;  **if** (processDest == (processCount - 1)) {  processDest = 0;  } **else** {  processDest++;  }  }  **for** (**int** i = 1; i < processCount; i++) {  **MPI\_Send**(&blockSizesArray[i], 1, MPI\_INT, i, BLOCKS\_LISTEN\_TAG, MPI\_COMM\_WORLD);  }  blockSize = blockSizesArray[processRank];  **delete**[] blockSizesArray;  } |

Os números enviados a cada processo são armazenados em uma variável de tipo *vector* (*processPartNumbersVector*).

O trecho do código responsável por realizar o recebimento das informações enviadas acima é o que segue:

|  |
| --- |
| **void** **startReceivingElements**() {  **MPI\_Recv**(&blockSize, 1, MPI\_INT, 0, BLOCKS\_LISTEN\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  **float** receivedNumber;  **for** (**int** i = 0; i < blockSize; i++) {  **MPI\_Recv**(&receivedNumber, 1, MPI\_FLOAT, 0, GLOBAL\_LISTEN\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  processPartNumbersVector.push\_back(receivedNumber);  }  } |

O próximo trecho de código tem a responsabilidade de executar o que diz respeito a ETAPA-02 proposta pelo exercício, onde os operandos de cada soma sempre deverão ser originados de outro processo.

O objetivo desta etapa é fazer com que cada processo possua apenas um único elemento. Para isto foi utilizada a estratégia de envio em anel onde cada processo envia um número (um operando) para o processo seguinte o qual deve somar o número recebido ao primeiro elemento existente em seu vetor (*processPartNumbersVector*) conforme exemplificado na ilustração a seguir:

Ilustração 2

O trecho do código responsável por realizar a distribuição dos operandos conforme exemplificado acima e a soma dos números recebidos é o que segue:

|  |
| --- |
| **void** **startReductionTreeSimulation**(**long** **double** &partialSum) {  partialSum = processPartNumbersVector[0];  **int** numbersToSendCount = blockSize - 1;  **float** receivedNumber;  **if** (numbersToSendCount > 0) {  **int** processTarget = processRank == (processCount - 1) ? PROCESS\_MASTER : processRank + 1;  **MPI\_Send**(&numbersToSendCount, 1, MPI\_INT, processTarget, BLOCKS\_LISTEN\_TAG, MPI\_COMM\_WORLD);  **for** (**int** i = 1; i < processPartNumbersVector.size(); i++) {  **MPI\_Send**(&processPartNumbersVector[i], 1, MPI\_FLOAT, processTarget, GLOBAL\_LISTEN\_TAG, MPI\_COMM\_WORLD);  }  **int** processSource = processRank == PROCESS\_MASTER ? processCount - 1 : processRank - 1;  **int** numbersToReceiveCount;  **MPI\_Recv**(&numbersToReceiveCount, 1, MPI\_INT, processSource, BLOCKS\_LISTEN\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  **for** (**int** i = 0; i < numbersToReceiveCount; i++) {  **MPI\_Recv**(&receivedNumber, 1, MPI\_FLOAT, processSource, GLOBAL\_LISTEN\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  partialSum += receivedNumber;  }  }  } |

# Processamento dos dados

Agora que finalmente cada processo possui apenas um único número a solução inicia o processamento correspondente a EPATA-03 onde a árvore de redução é executada.

Para encontrar a altura da árvore de redução foi calculado o logaritmo do número total de processos na base dois (levelsCount = **log2**(processCount)), ou seja, para uma situação onde temos 8 processos (*np* = 8) teremos uma árvore de altura igual a 3. Com posse desta informação a solução executa uma iteração para varrer todos os níveis da árvore para cada processo.

O trecho do código responsável por realizar o processamento da árvore de redução é o que segue:

|  |
| --- |
| **void** **startReductionTree**(**long** **double** &partialSum) {  **int** processSource, processTarget;  **int** level, currentLevel, nextLevel;  **float** levelsCount = 0;  **long** **double** receivedNumber;  **long** **double** totalSum = partialSum;  levelsCount = **log2**(processCount);  **for** (currentLevel = 0; currentLevel < levelsCount; currentLevel++) {  level = (**int**) (std::pow(2, currentLevel));  **if** ((processRank % level) == 0) {  nextLevel = (**int**) (**pow**(2, (currentLevel + 1)));  **if** ((processRank % nextLevel) == 0) {  processSource = processRank + level;  **if** (processSource > processCount - 1) {  **continue**;  }  **MPI\_Recv**(&receivedNumber, 1, MPI\_LONG\_DOUBLE, processSource, GLOBAL\_LISTEN\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  totalSum += receivedNumber;  } **else** {  processTarget = processRank - level;  **MPI\_Send**(&totalSum, 1, MPI\_LONG\_DOUBLE, processTarget, GLOBAL\_LISTEN\_TAG, MPI\_COMM\_WORLD);  }  }  }  result.sum = totalSum;  } |

Para avaliar o tempo de execução necessário ao processamento, os passos referentes as ETAPAS 01 e 02 foram medidas através da utilização da função *MPI\_Wtime()*.

A seguir é mostrado o trecho do código responsável por chamar as funções que realizam as três ETAPAS propostas pelo exercício bem como a medição do tempo gasto com o processamento:

|  |
| --- |
| **void** **startParallelProcess**(**const** **int** numbersCount, **const** **float** \*numbersArray) {  **long** **double** partialSum = 0;  // STEP 1 - Elements Distribution  // ----------------------------  **if** (processRank == PROCESS\_MASTER) {  // SENDING  **if** (processCount > numbersCount) {  processCount = numbersCount;  }  startDispatchElements(numbersCount, numbersArray);  } **else** {  // RECEIVING  startReceivingElements();  }  **double** startTime, endTime;  startTime = **MPI\_Wtime**();  // STEP 2 - Reduction Tree Simulation  // ----------------------------  startReductionTreeSimulation(partialSum);  // STEP 3 - Reduction Tree  // ----------------------------  startReductionTree(partialSum);  endTime = **MPI\_Wtime**();  result.processTime = (endTime - startTime) \* 1000;  } |

# Implementação MPI

Ao desenvolvimento deste exercício foi utilizado a implementação OPEN-MPI versão 1.8.4 disponibilizada em 19 de dezembro de 2014.

# Instrução de Compilação

mpic++ -g -Wall main.cpp -o prog

# Instrução de Execução

MacOs/Linux:

mpirun -n 2 prog

Windows:

mpiexec -n 2 prog

Modo de uso:

mpirun [-n <número de processos desejados>] prog

tipo de saída -- após o comando acima, informe o tipo de saída [sum | time | all]

quantidade de números – quantidade números a serem somados

Exemplo:

mpirun -n 4 prog

all

8

1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0

Observação: se o valor informado ao número de processos for igual a 0 (zero), a implementação de *MPI* utilizada irá automaticamente alterar este valor para 2(dois).

# Ambiente de Execução dos Testes

|  |  |
| --- | --- |
| Nome do Modelo | iMac |
| Identificador do Modelo | iMac12,1 |
| Nome do Processador | Intel Core i5 |
| Velocidade do Processador | 2,5 GHz |
| Número de Processadores | 1 |
| Número Total de Núcleos | 4 |
| Cache L2 (por Núcleo) | 256 KB |
| Cache de L3 | 6 MB |
| Memória | 10 GB 1333 MHz DDR3 |

# Descrição dos testes

Os testes foram executados 5 vezes utilizando 1, 2, 4, 6, 8, 10 e 16 processos respectivamente, onde para cada um dos testes aplicados foram passados como parâmetros os arquivos listados na tabela abaixo contendo diferentes quantidades de números para serem somados.

A coluna “Valor da soma” apresenta o resultado final de cada soma calculado pelo programa desenvolvido.

|  |  |  |
| --- | --- | --- |
| **Quantidade de números** | **Valor da soma** | **Link para baixar** |
| 262.144 (218) | 34.359.974.972,24 | [input-file-01.txt](https://drive.google.com/file/d/0BzY0JDw3_l90a3ltbU5EbFE1UkU/view?usp=sharing#Jefferson%20Chaves%20Gomes) |
| 524.288 (219) | 137.439.477.760,00 | [input-file-02.txt](https://drive.google.com/file/d/0BzY0JDw3_l90UWlJZjRSRlJTa1k/view?usp=sharing#Jefferson%20Chaves%20Gomes) |
| 1.048.576 (220) | 549.756.338.176,00 | [input-file-03.txt](https://drive.google.com/file/d/0BzY0JDw3_l90N1ZUTjlSOXBKMzA/view?usp=sharing#Jefferson%20Chaves%20Gomes) |
| 1.594.323 (313) | 1.270.933.711.326,00 | [input-file-04.txt](https://drive.google.com/file/d/0BzY0JDw3_l90SDNQSUJEWjVJb0U/view?usp=sharing#Jefferson%20Chaves%20Gomes) |
| 2.097.152 (221) | 2.199.024.304.128,00 | [input-file-05.txt](https://drive.google.com/file/d/0BzY0JDw3_l90SHJpbjZIUGs2MWM/view?usp=sharing#Jefferson%20Chaves%20Gomes) |
| 4.194.304 (222) | 8.796.095.119.360,00 | [input-file-06.txt](https://drive.google.com/file/d/0BzY0JDw3_l90QTVEM0c3NlAtcnM/view?usp=sharing#Jefferson%20Chaves%20Gomes) |

# Análise de Resultados

# Tempos de Execução

Os resultados referentes a performance após a execução dos testes são ilustrados a seguir:

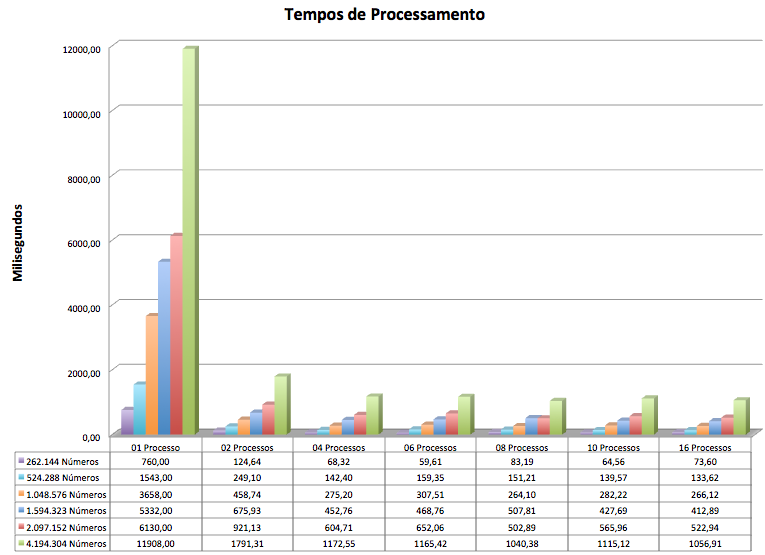


Ilustração 3

O gráfico ilustrado acima mostra que os ganhos de performance com a execução paralela em relação a execução sequencial são realmente muito significativos.

Analisando o gráfico como um todo é fácil perceber que o tempo de processamento atinge o melhor número quando quatro processos são utilizados para efetuar o calculo e que este fato está diretamente relacionado a quantidade de *cores* existentes no ambiente de execução dos testes que fornece apenas quatro *cores* ao programa. Mesmo o hardware de testes estando limitado a quatro *cores* é possível obter ganhos no tempo de processamento utilizando dez e dezesseis processos.

# Análise de *Speedup*

O *speedup* correspondente ao gráfico de tempos de execução é o que segue:

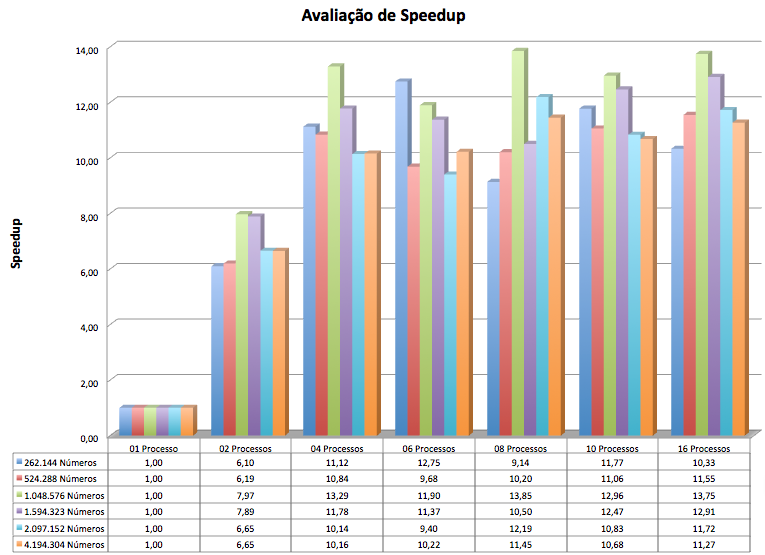


Ilustração 4

Analisando o *speedup* no gráfico acima, é possível notar que as relações entre o tempo de execução sequencial e o tempo de execução em paralelo tem valores significativos mostrando que a execução paralela chega a ser *6,1x* mais rápida no pior caso e *13,85x* mais rápida no melhor caso.

O pico alcançado nesta relação (tempo de execução sequencial X tempo de execução em paralelo) foi atingido com oito processos calculando a soma de 1.048.576 números.

# Análise de Eficiência

A eficiência correspondente ao gráfico de análise de *speedup* é ilustrado a seguir:

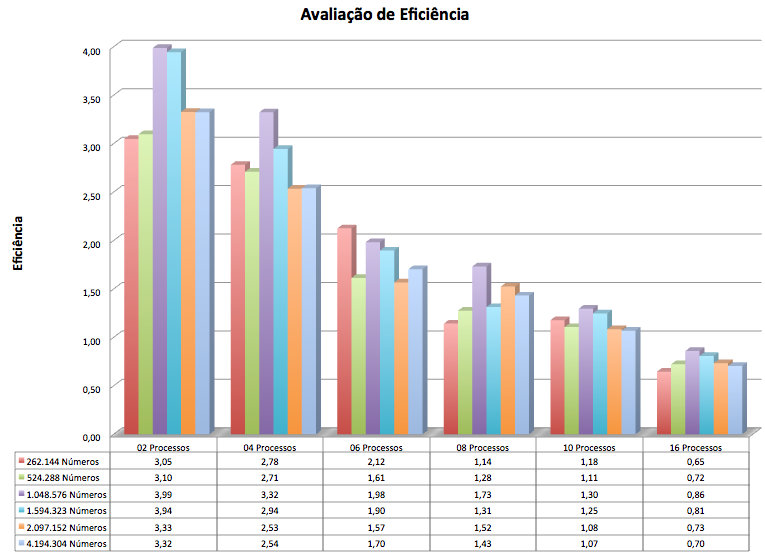


Ilustração 5

De acordo com o resultados de eficiência mostrados na ilustração acima é possível concluir que a solução é escalável.

Devido ao fato do grande ganho de performance mostrada nas ilustrações acima, os resultados referentes a eficiência na maioria dos casos se manteve acima de uma unidade.

Após análise de eficiência é possível determinar a configuração paralela que mais consumiu recursos do hardware foi a que utilizou dois processos.

Levando em consideração que a máquina de testes possui **apenas quatro *cores*** é possível concluir que a solução é fortemente escalável tendo visto nos gráficos ilustrados acima (ilustração 3, 4 e 5) que aumentando o número recursos (processos) é possível diminuir proporcionalmente o tempo de execução mantendo proporcionalmente a eficiência.

# Código Completo

|  |
| --- |
| //============================================================================  // Name : main.cpp  // Author : Jefferson Chaves Gomes  // Version : 1.0.0  // Copyright : Academic Program  // Description : EP 03 in C++  // Compiling : mpic++ -g -Wall main.cpp -o prog  // Executing : mpirun -n 1 prog  //============================================================================  // Libraries definitions  // -----------------------------------------------------  **#include** <iostream> // for std::cin, std::cout, etc.  **#include** <limits.h> // for INT\_MAX  **#include** <sys/time.h> // for gettimeofday  **#include** <vector> // for vector  **#include** <cmath> // for std::pow  **#include** <iomanip> // for std::setprecision  **#include** <mpi.h> // for MPI  // Constants definitions  // -----------------------------------------------------  **#define** BREAK\_LINE "\n"  **#define** TAB "\t"  **#define** MAX\_ATTEMPTS 5  **#define** PROCESS\_MASTER 0  **#define** GLOBAL\_LISTEN\_TAG 0  **#define** BLOCKS\_LISTEN\_TAG 1  **#define** MPI\_WTIME\_IS\_GLOBAL 1  // Enums definitions  // -----------------------------------------------------  **enum** OutputType {  *NONE* = 0,  *TIME* = 1,  *SUM* = 2,  *ALL* = 3  };  // Structs definitions  // -----------------------------------------------------  **struct** Result {  **double** processTime;  **long** **double** sum;  };  // Functions declarations  // -----------------------------------------------------  **void** **checkInputParams**(**int**, **char**\*\*);  **int** **readNumbersCount**();  OutputType **readOutputType**();  OutputType **stringToOutputType**(std::string);  **void** **readNumbersArray**(**float**\*, **const** **int**);  **void** **printUsage**();  **void** **checkProcessTime**();  **void** **printResult**(OutputType);  **void** **allowAnotherAttempt**(**const** std::string, **int**&);  **void** **startSerialProcess**(**const** **float**\*, **const** **int**);  **void** **startParallelProcess**(**const** **int**, **const** **float**\*);  **void** **startDispatchElements**(**const** **int**, **const** **float**\*);  **void** **startReceivingElements**();  **void** **startReductionTreeSimulation**(**long** **double**&);  **void** **startReductionTree**(**long** **double**&);  // Globals variable  // -----------------------------------------------------  Result result;  **int** processCount;  **int** processRank;  **int** blockSize = 0;  std::vector<**float**> processPartNumbersVector;  // -----------------------------------------------------  // main Function  // -----------------------------------------------------  **int** **main**(**int** argc, **char** \*\*argv) {  // Input variables  OutputType outputType = *NONE*;  **int** numbersCount = 0;  **float** \*numbersArray = NULL;  // MPI initialization  // ----------------------------  **MPI\_Init**(&argc, &argv);  **MPI\_Comm\_size**(MPI\_COMM\_WORLD, &processCount);  **MPI\_Comm\_rank**(MPI\_COMM\_WORLD, &processRank);  // Read input data  // ----------------------------  **if** (processRank == PROCESS\_MASTER) {  checkInputParams(argc, argv);  outputType = readOutputType();  numbersCount = readNumbersCount();  numbersArray = **new** **float**[numbersCount];  readNumbersArray(numbersArray, numbersCount);  }  // Checks if data processing will be serial  // ----------------------------  **if** (processCount == 1) {  startSerialProcess(numbersArray, numbersCount);  printResult(outputType);  **MPI\_Finalize**();  **return** EXIT\_SUCCESS;  }  // Start parallel process  // ----------------------------  startParallelProcess(numbersCount, numbersArray);  // Check the process time  // ----------------------------  checkProcessTime();  // Printing the result  // ----------------------------  **if** (processRank == PROCESS\_MASTER) {  printResult(outputType);  }  **delete**[] numbersArray;  // MPI finalization  // ----------------------------  **MPI\_Finalize**();  **return** EXIT\_SUCCESS;  }  // Function implementations  // -----------------------------------------------------  **void** **checkInputParams**(**int** argc, **char** \*\*argv) {  **if** (argc != 1) {  printUsage();  **exit** (EXIT\_FAILURE);  }  }  OutputType **readOutputType**() {  **int** badAttempts = 0;  std::string outputOption = "NONE";  std::cin >> outputOption;  **while** (stringToOutputType(outputOption) == *NONE*) {  allowAnotherAttempt("Error: The output type must be equal then [sum | time | all].", badAttempts);  std::cin.clear();  std::cin >> outputOption;  }  **return** stringToOutputType(outputOption);  }  **int** **readNumbersCount**() {  **int** badAttempts = 0;  **int** numberCount;  std::cin >> numberCount;  **while** (std::cin.bad() || std::cin.fail() || numberCount < 2) {  allowAnotherAttempt("Error: The amount of numbers must be a valid integer greater then 1.", badAttempts);  std::cin.clear();  std::cin.ignore(INT\_MAX, '\n');  std::cin >> numberCount;  }  **return** numberCount;  }  **void** **readNumbersArray**(**float**\* numbersArray, **const** **int** numbersCount) {  **int** badAttempts = 0;  **double** number = 0.0;  **for** (**long** i = 0; i < numbersCount; i++) {  std::cin >> number;  **while** (std::cin.bad() || std::cin.fail()) {  allowAnotherAttempt("Error: The number must be a integer or a float [5 | 5.0].", badAttempts);  std::cin.clear();  std::cin.ignore(INT\_MAX, '\n');  std::cin >> number;  }  numbersArray[i] = number;  }  }  **void** **allowAnotherAttempt**(**const** std::string msg, **int** &badAttempts) {  **if** (++badAttempts >= MAX\_ATTEMPTS) {  printUsage();  **exit** (EXIT\_FAILURE);  }  std::cout << msg << BREAK\_LINE;  std::cout << "Try again (you can try more " << MAX\_ATTEMPTS - badAttempts << " times)." << BREAK\_LINE;  }  **void** **printUsage**() {  std::cout << BREAK\_LINE << BREAK\_LINE;  std::cout << "Usage:" << BREAK\_LINE;  std::cout << TAB << "mpirun [-n <number of process>] pp-ep03-012015" << BREAK\_LINE << BREAK\_LINE;  std::cout << TAB << "output type -- after run, inform the output type [sum | time | all]" << BREAK\_LINE;  std::cout << TAB << "amount of numbers -- and after, inform the amount of numbers to be added" << BREAK\_LINE << BREAK\_LINE;  std::cout << "Sample:" << BREAK\_LINE;  std::cout << TAB << "mpirun -n 4 pp-ep03-012015" << BREAK\_LINE;  std::cout << TAB << "all" << BREAK\_LINE;  std::cout << TAB << "8" << BREAK\_LINE;  std::cout << TAB << "1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0" << BREAK\_LINE << BREAK\_LINE;  std::cout << "NOTE: if the given number for <number of process> is equal to 0, then the MPI library will set it to 2." << BREAK\_LINE << BREAK\_LINE;  }  OutputType **stringToOutputType**(std::string input) {  **if** (input == "time") {  **return** *TIME*;  } **else** **if** (input == "sum") {  **return** *SUM*;  } **else** **if** (input == "all") {  **return** *ALL*;  }  **return** *NONE*;  }  **void** **printResult**(OutputType outputType) {  **switch** (outputType) {  **case** *TIME*:  std::cout << result.processTime << BREAK\_LINE;  **break**;  **case** *SUM*:  std::cout << std::fixed;  std::cout << std::setprecision(2) << result.sum << BREAK\_LINE;  **break**;  **case** *ALL*:  std::cout << std::fixed;  std::cout << std::setprecision(2) << result.sum << BREAK\_LINE;  std::cout << result.processTime << BREAK\_LINE;  **break**;  **default**:  **break**;  }  }  **void** **checkProcessTime**() {  **if** (processRank != PROCESS\_MASTER) {  **MPI\_Send**(&result.processTime, 1, MPI\_DOUBLE, 0, GLOBAL\_LISTEN\_TAG, MPI\_COMM\_WORLD);  }  **if** (processRank == PROCESS\_MASTER) {  **double** time;  **for** (**int** i = 1; i < processCount; ++i) {  **MPI\_Recv**(&time, 1, MPI\_DOUBLE, i, GLOBAL\_LISTEN\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  **if** (result.processTime < time) {  result.processTime = time;  }  }  }  }  **void** **startSerialProcess**(**const** **float** \*numbersArray, **const** **int** numbersCount) {  **double** totalSum = 0;  timeval startTime, endTime;  **gettimeofday**(&startTime, NULL);  **for** (**int** i = 0; i < numbersCount; ++i) {  totalSum += numbersArray[i];  }  **gettimeofday**(&endTime, NULL);  **double** time = ((endTime.tv\_sec \* 1000 + endTime.tv\_usec) - (startTime.tv\_sec \* 1000 + startTime.tv\_usec));  result.sum = totalSum;  result.processTime = time;  }  **void** **startParallelProcess**(**const** **int** numbersCount, **const** **float** \*numbersArray) {  **long** **double** partialSum = 0;  // STEP 1 - Elements Distribution  // ----------------------------  **if** (processRank == PROCESS\_MASTER) {  // SENDING  **if** (processCount > numbersCount) {  processCount = numbersCount;  }  startDispatchElements(numbersCount, numbersArray);  } **else** {  // RECEIVING  startReceivingElements();  }  **double** startTime, endTime;  startTime = **MPI\_Wtime**();  // STEP 2 - Reduction Tree Simulation  // ----------------------------  startReductionTreeSimulation(partialSum);  // STEP 3 - Reduction Tree  // ----------------------------  startReductionTree(partialSum);  endTime = **MPI\_Wtime**();  result.processTime = (endTime - startTime) \* 1000;  }  **void** **startDispatchElements**(**const** **int** numbersCount, **const** **float** \*numbersArray) {  **int** \*blockSizesArray = **new** **int**[processCount];  std::fill\_n(blockSizesArray, processCount, 0);  **for** (**int** i = 0, processDest = 0; i < numbersCount; i++) {  **if** (processDest == PROCESS\_MASTER) {  processPartNumbersVector.push\_back(numbersArray[i]);  } **else** {  **MPI\_Send**(&numbersArray[i], 1, MPI\_FLOAT, processDest, GLOBAL\_LISTEN\_TAG, MPI\_COMM\_WORLD);  }  blockSizesArray[processDest]++;  **if** (processDest == (processCount - 1)) {  processDest = 0;  } **else** {  processDest++;  }  }  **for** (**int** i = 1; i < processCount; i++) {  **MPI\_Send**(&blockSizesArray[i], 1, MPI\_INT, i, BLOCKS\_LISTEN\_TAG, MPI\_COMM\_WORLD);  }  blockSize = blockSizesArray[processRank];  **delete**[] blockSizesArray;  }  **void** **startReceivingElements**() {  **MPI\_Recv**(&blockSize, 1, MPI\_INT, 0, BLOCKS\_LISTEN\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  **float** receivedNumber;  **for** (**int** i = 0; i < blockSize; i++) {  **MPI\_Recv**(&receivedNumber, 1, MPI\_FLOAT, 0, GLOBAL\_LISTEN\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  processPartNumbersVector.push\_back(receivedNumber);  }  }  **void** **startReductionTreeSimulation**(**long** **double** &partialSum) {  partialSum = processPartNumbersVector[0];  **int** numbersToSendCount = blockSize - 1;  **float** receivedNumber;  **if** (numbersToSendCount > 0) {  **int** processTarget = processRank == (processCount - 1) ? PROCESS\_MASTER : processRank + 1;  **MPI\_Send**(&numbersToSendCount, 1, MPI\_INT, processTarget, BLOCKS\_LISTEN\_TAG, MPI\_COMM\_WORLD);  **for** (**int** i = 1; i < processPartNumbersVector.size(); i++) {  **MPI\_Send**(&processPartNumbersVector[i], 1, MPI\_FLOAT, processTarget, GLOBAL\_LISTEN\_TAG, MPI\_COMM\_WORLD);  }  **int** processSource = processRank == PROCESS\_MASTER ? processCount - 1 : processRank - 1;  **int** numbersToReceiveCount;  **MPI\_Recv**(&numbersToReceiveCount, 1, MPI\_INT, processSource, BLOCKS\_LISTEN\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  **for** (**int** i = 0; i < numbersToReceiveCount; i++) {  **MPI\_Recv**(&receivedNumber, 1, MPI\_FLOAT, processSource, GLOBAL\_LISTEN\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  partialSum += receivedNumber;  }  }  }  **void** **startReductionTree**(**long** **double** &partialSum) {  **int** processSource, processTarget;  **int** level, currentLevel, nextLevel;  **float** levelsCount = 0;  **long** **double** receivedNumber;  **long** **double** totalSum = partialSum;  levelsCount = **log2**(processCount);  **for** (currentLevel = 0; currentLevel < levelsCount; currentLevel++) {  level = (**int**) (std::pow(2, currentLevel));  **if** ((processRank % level) == 0) {  nextLevel = (**int**) (**pow**(2, (currentLevel + 1)));  **if** ((processRank % nextLevel) == 0) {  processSource = processRank + level;  **if** (processSource > processCount - 1) {  **continue**;  }  **MPI\_Recv**(&receivedNumber, 1, MPI\_LONG\_DOUBLE, processSource, GLOBAL\_LISTEN\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  totalSum += receivedNumber;  } **else** {  processTarget = processRank - level;  **MPI\_Send**(&totalSum, 1, MPI\_LONG\_DOUBLE, processTarget, GLOBAL\_LISTEN\_TAG, MPI\_COMM\_WORLD);  }  }  }  result.sum = totalSum;  } |