Universidade de Brasília

Departamento de Ciência da Computação

Disciplina: Programação Paralela, código 316342

Professor: George Luiz Medeiros Teodoro

Aluno: Jefferson Chaves Gomes – 14/0189581

Exercício de Programação 03

Maio de 2015

Índice

1. 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:

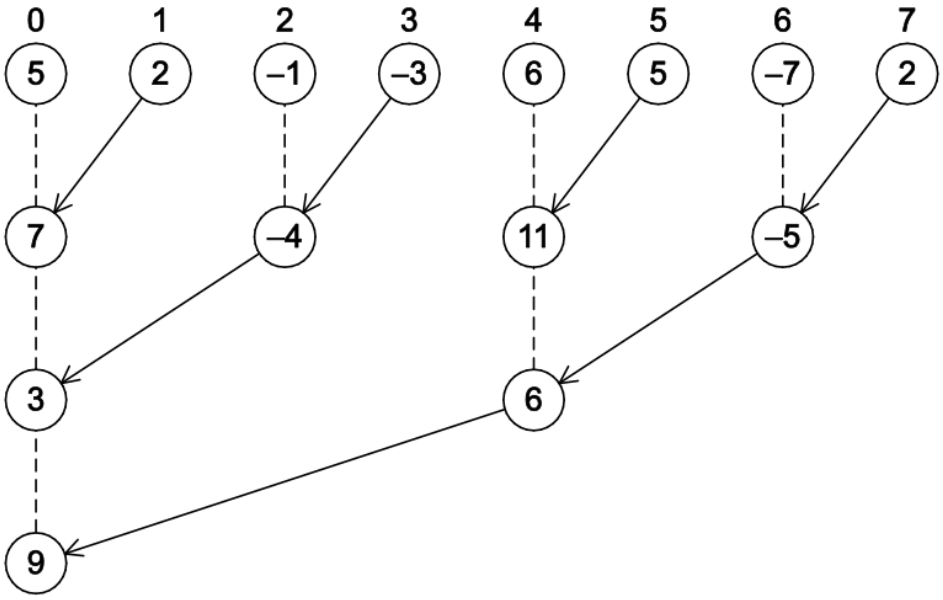


Ilustração 1

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.

2. 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:
    i. 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;
    ii. 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;
    iii. 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*.

3. 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.

4. 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 milissegundos) 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.

5. 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.

5.1. 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;
};
```

5.2. Leitura e validação de dados de entrada

Uma função responsável pela validação da quantidade de parâmetros passados ao programa foi criada. Esta função foi chamada de `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;
    }
}
```

5.3. 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 da utilização de apenas um único processo, ou seja, a aferição será de *speedup* relativo que é definida como o tempo decorrente de um algoritmo paralelo executando com um único processo e o tempo decorrente do mesmo algoritmo paralelo executando com *np* processos.

Tanto aos casos seriais quanto aos casos paralelos, 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:

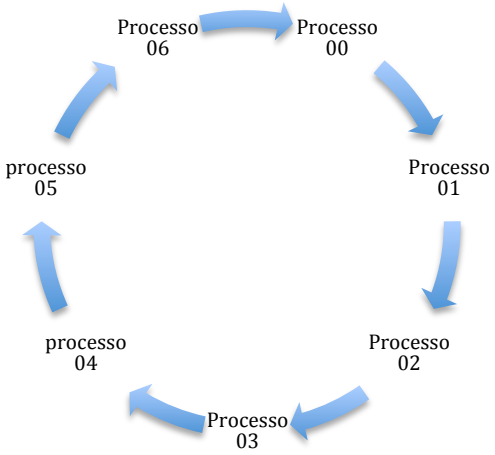


Ilustração 2

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;
        }
    }
}
```

5.4. 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 três. 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;
}
```

6. Implementação MPI

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

7. Instrução de Compilação

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

8. 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).

9. 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

10. 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.

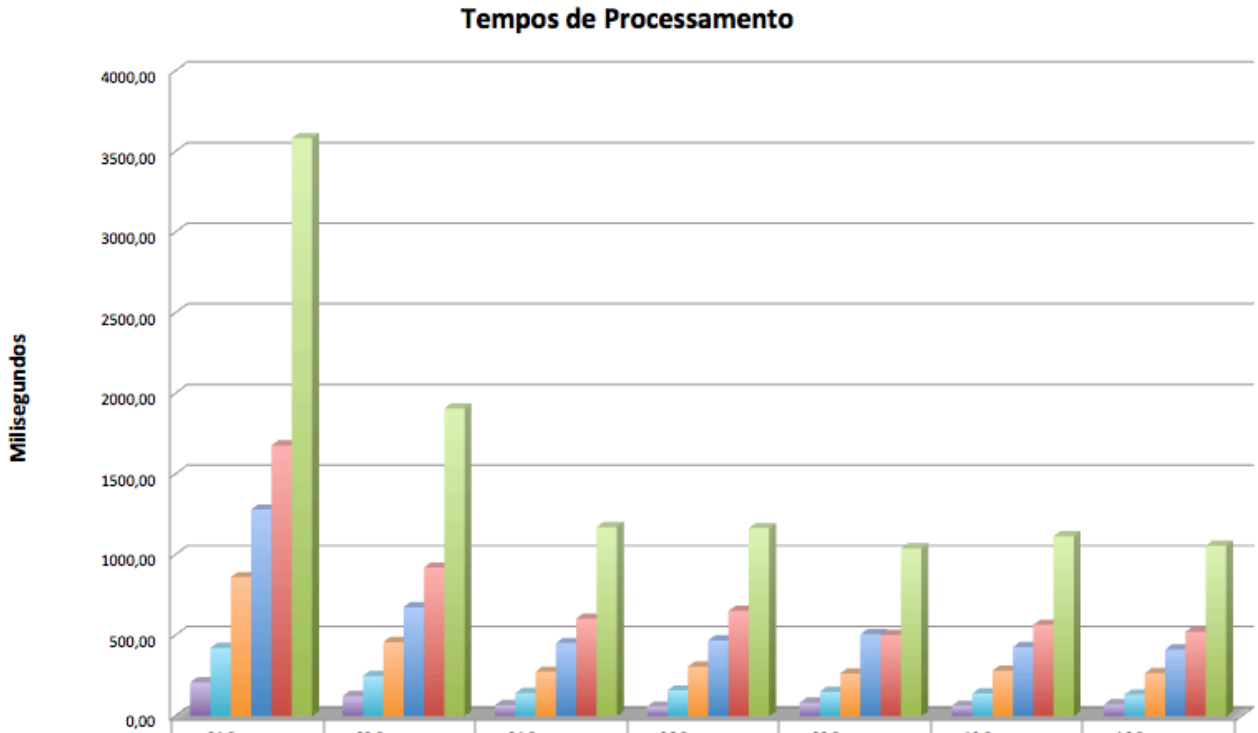
A coluna "Valor da soma" apresenta o resultado final de cada soma calculado pelo programa desenvolvido.

Quantidade de números	Valor da soma	Link para baixar
262.144 (2^{18})	34.359.974.972,24	input-file-01.txt
524.288 (2^{19})	137.439.477.760,00	input-file-02.txt
1.048.576 (2^{20})	549.756.338.176,00	input-file-03.txt
1.594.323 (3^{13})	1.270.933.711.326,00	input-file-04.txt
2.097.152 (2^{21})	2.199.024.304.128,00	input-file-05.txt
4.194.304 (2^{22})	8.796.095.119.360,00	input-file-06.txt

11. Análise de Resultados

11.1. Tempos de Execução

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



	01 Processo	02 Processos	04 Processos	06 Processos	08 Processos	10 Processos	16 Processos
262.144 Números	209,85	124,64	68,32	59,61	83,19	64,56	73,60
524.288 Números	423,70	249,10	142,40	159,35	151,21	139,57	133,62
1.048.576 Números	862,06	458,74	275,20	307,51	264,10	282,22	266,12
1.594.323 Números	1281,34	675,93	452,76	468,76	507,81	427,69	412,89
2.097.152 Números	1678,29	921,13	604,71	652,06	502,89	565,96	522,94
4.194.304 Números	3583,07	1907,58	1172,55	1165,42	1040,38	1115,12	1056,91

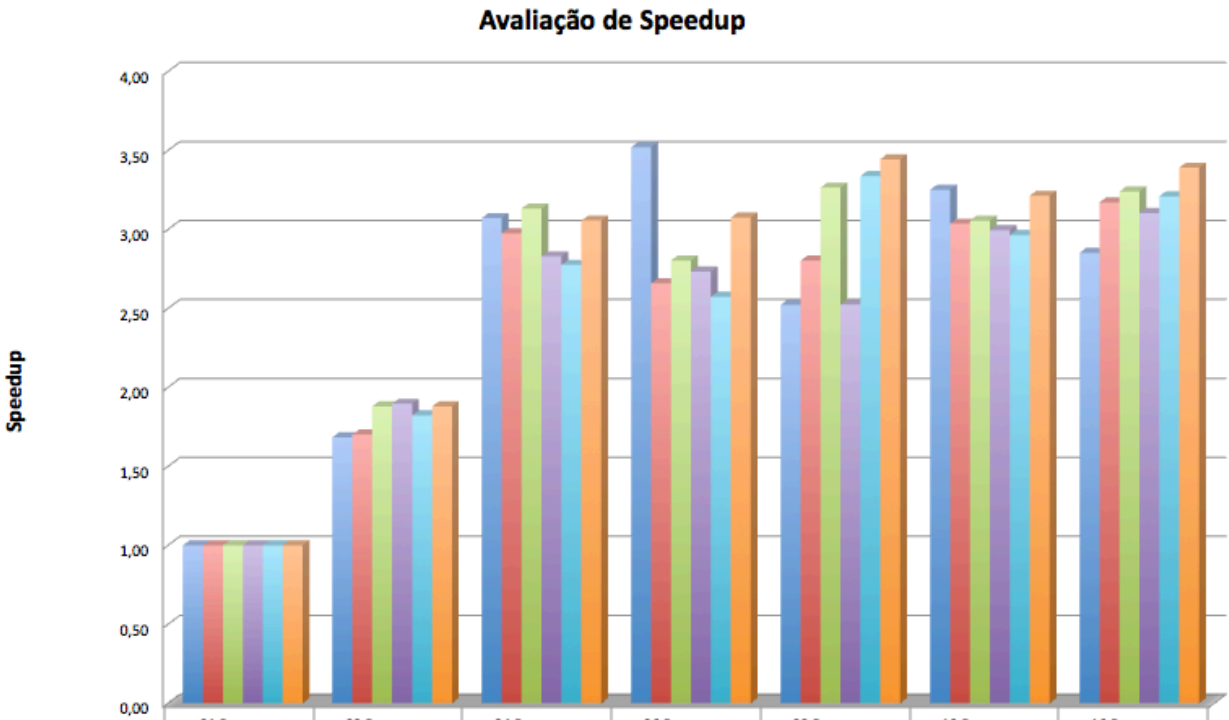
Ilustração 3

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* também foi possível obter ganhos no tempo de processamento utilizando seis, oito, dez e dezesseis processos.

11.2. Análise de *Speedup*

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



	01 Processo	02 Processos	04 Processos	06 Processos	08 Processos	10 Processos	16 Processos
262.144 Números	1,00	1,68	3,07	3,52	2,52	3,25	2,85
524.288 Números	1,00	1,70	2,98	2,66	2,80	3,04	3,17
1.048.576 Números	1,00	1,88	3,13	2,80	3,26	3,05	3,24
1.594.323 Números	1,00	1,90	2,83	2,73	2,52	3,00	3,10
2.097.152 Números	1,00	1,82	2,78	2,57	3,34	2,97	3,21
4.194.304 Números	1,00	1,88	3,06	3,07	3,44	3,21	3,39

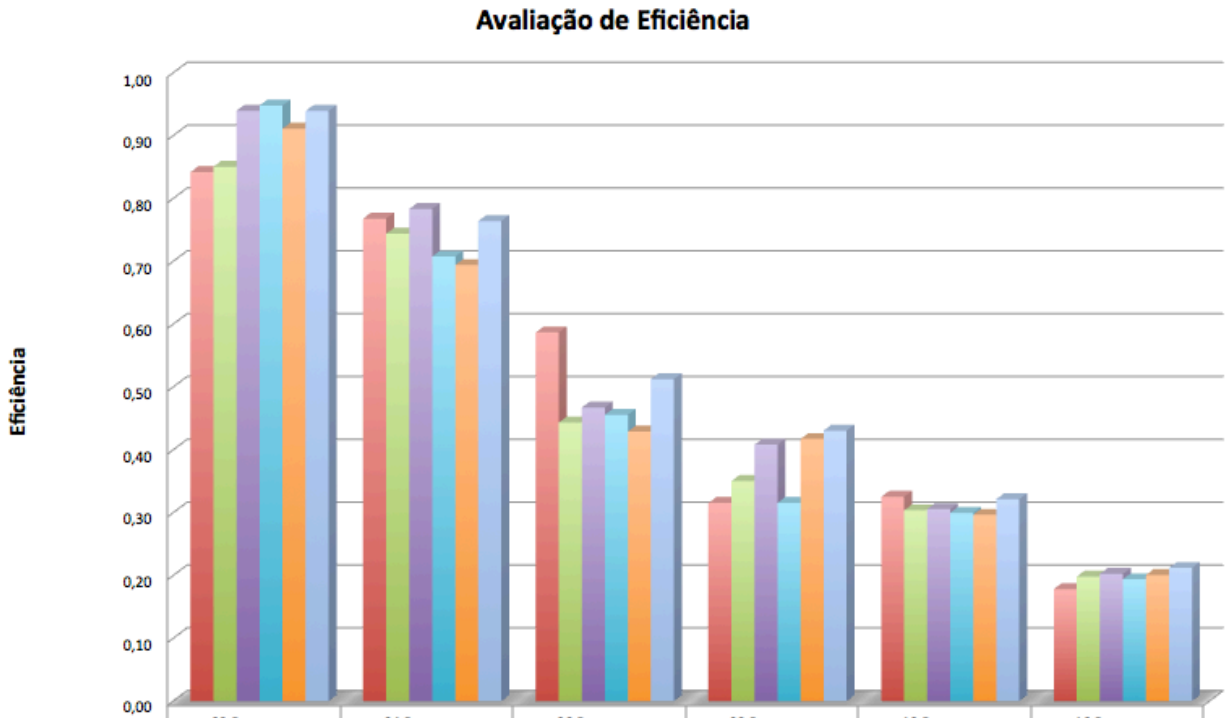
Ilustração 4

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 *1,68x* mais rápida no pior caso e *3,52x* mais rápida no melhor caso.

O pico alcançado nesta relação de razão entre o tempo de execução sequencial X tempo de execução em paralelo foi atingido com seis processos calculando a soma de 262.144 números.

11.3. Análise de Eficiência

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



	02 Processos	04 Processos	06 Processos	08 Processos	10 Processos	16 Processos
262.144 Números	0,84	0,77	0,59	0,32	0,33	0,18
524.288 Números	0,85	0,74	0,44	0,35	0,30	0,20
1.048.576 Números	0,94	0,78	0,47	0,41	0,31	0,20
1.594.323 Números	0,95	0,71	0,46	0,32	0,30	0,19
2.097.152 Números	0,91	0,69	0,43	0,42	0,30	0,20
4.194.304 Números	0,94	0,76	0,51	0,43	0,32	0,21

Ilustração 5

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.

De acordo com os resultados anteriores podemos concluir que a eficiência de uma aplicação é uma função decrescente do número de processadores e tipicamente uma função crescente do tamanho do problema.

12. 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 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);
    }

    // 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 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;
}
```