LABORATÓRIO III
BIBLIOTECA MPI

Grupo: **Not Found**

Julia De Souza Dos Santos - 238166
Matheus Sales Oliveira - 203577

SUMÁRIO

1. PROBLEMA

1.1. DEFINIÇÃO

Dadas três matrizes A_{yxw}, B_{wxv}, C_{vx1}, calcule a matriz D_{yx1} tal que D_{yx1}= (A_{yxw} X B_{wxv}) X C_{vx1}. Além disso, calcule a redução pela soma dos elementos na matriz D_{yx1}, isto é, a soma de todos os elementos em D_{yx1}.

1.2. REQUISITOS

- ✓ O programa deve ser escrito em linguagem C e no ambiente Linux.
- ✓ O código fonte deve estar disponível em um repositório no GitHub.
- ✓ O problema deverá ser resolvido utilizando a biblioteca MPI.
- ✓ As dimensões das matrizes são definidas pelas variáveis w, v e y (informadas pela linha de comando).
- ✓ Os dados das matrizes devem ser números reais, com até duas casas decimais, entre -10 e +10. Esses números devem ser gerados aleatoriamente.
- ✓ As matrizes devem ser alocadas dinamicamente em uma única etapa.
- ✓ Os valores de cada matriz devem estar armazenados sequencialmente em um arquivo do tipo “dat”.
- ✓ O programa deve ser executado da seguinte forma através do terminal:

```
./programa y w v arqA.dat arqB.dat arqC.dat arqD.dat
```

Onde:

- *./programa* → nome do programa que realiza a multiplicação das matrizes
- *y* → número de linhas da matriz A.
- *w* → número de colunas da matriz A e número de linhas da matriz B.
- *v* → número de colunas da matriz B e número de linhas da matriz C.
- *arqA.dat* → nome do arquivo que contém a matriz A.
- *arqB.dat* → nome do arquivo que contém a matriz B.
- *arqC.dat* → nome do arquivo que contém a matriz C.
- *arqD.dat* → nome do arquivo que contém a matriz resultante B.

2. OBJETIVO

O objetivo deste trabalho é avaliar a influência da paralelização de um programa sobre o seu tempo de execução. Nesse cenário, desenvolveu-se um programa na linguagem C que propõe uma solução para o problema apresentado acima utilizando a paralelização. Desse modo, o paralelismo foi implementado utilizando a biblioteca MPI, que possibilita a paralelização através da troca de mensagens em um sistema com memória distribuída.

3. PROGRAMA

O programa desenvolvido neste laboratório apresenta uma estrutura diferente quando comparado com aqueles desenvolvidos nos laboratórios anteriores. Dentre elas, a adição de algumas funções e uma *main* totalmente diferente, para que a biblioteca MPI funcionasse adequadamente. Dentre as funções que fazem uso das diretivas do MPI, temos: *main, enviar_para_workers* e *receber_das_workers.* Abaixo, detalhamos cada uma das funções presentes no programa:

a) `multiplicar_matrizes`: essa função é responsável por multiplicar duas matrizes e armazenar o resultado dessa multiplicação em uma matriz "*resultado"* passada como parâmetro. Em relação ao laboratório anterior, essa função sofreu algumas alterações: mudou-se o tipo de retorno de ***float** ** para ***void***, alterou-se os nomes das matrizes passadas como parâmetro e, por fim, retirou-se as diretivas relacionadas a biblioteca *OpenACC*.

```
void multiplicar_matrizes(float *mat1, float *mat2, float *matResult, unsigned int y, unsigned int w, unsigned int v){
    register unsigned int linha, coluna, i;
    register float soma=0;

    for(linha=0; linha<y; linha++){
        for(coluna=0; coluna < v; coluna++){
            soma=0;
            for(i=0; i < w; i++){
                soma += mat1[linha*w + i] * mat2[i*v + coluna];
            }
            matResult[linha*v + coluna] = soma;
        }
    }
}
```

b) `reducao_pela_soma`: essa função é responsável por calcular a redução pela soma dos elementos armazenados na matriz resultante D_{yx1}. Para isto, a função recebe como parâmetros a matriz D_{yx1}, seu número de linhas e seu número de colunas e utiliza estes dados em um laço de repetição do tipo *for* aninhado para somar cada elemento da matriz a uma variável auxiliar chamada *soma*. Por fim, o

programa retorna o valor armazenado nesta variável após o fim dos laços.

```
double reducao_pela_soma(float *matriz, unsigned int linM, unsigned int colM){
    register int linha, coluna;
    register double soma = 0;

    for(linha=0; linha < linM; linha++){
        for(coluna=0; coluna < colM; coluna++){
            soma += matriz[linha*colM + coluna];
        }
    }

    return soma;
}
```

c) `gerar_matriz`: essa função é responsável por ler o arquivo que contém todos os elementos da matriz e armazenar cada um dos dados escritos nele na matriz alocada dinamicamente. Essa função também sofreu uma alteração em relação ao laboratório 2: alterou-se o tipo de retorno de ***float** ** para ***void***.

```
void gerar_matriz (char *nomedoArquivo, float *matriz, unsigned int linM, unsigned int colM){
    FILE  *arquivo = fopen(nomedoArquivo,"r");

    if(arquivo == NULL){
        printf("O arquivo %s nao existe!\n", nomedoArquivo);
        fclose(arquivo);
        exit(0);
    }

    register unsigned int linha, coluna;

    for (linha=0; linha<linM;linha++){
        for (coluna=0; coluna<colM;coluna++){
            fscanf(arquivo,"%f",&matriz[linha*colM +coluna]);
        }
    }

    fclose(arquivo);
}
```

d) `gravar_matriz`: essa função é responsável por escrever o resultado da multiplicação das matrizes no arquivo de saída. Para isso, ela recebe como parâmetro o nome do arquivo que será criado, a matriz que será gravada no arquivo e o número de linhas e colunas desta matriz.

```c
void gravar_matriz(char *nomeArquivo, float *matriz, unsigned int linM, unsigned int colM){
    FILE *arquivo = fopen(nomeArquivo, "w");

    if(arquivo == NULL){
        printf("Erro ao criar arquivo: %s\n", nomeArquivo);
        fclose(arquivo);
        exit(0);
    }

    register unsigned int linha, coluna;

    for (linha=0; linha<linM; linha++){
        for (coluna=0; coluna<colM;coluna++){
            fprintf(arquivo,"%.2f\n", matriz[linha*colM +coluna]);
        }
    }

     fclose(arquivo);
}
```

e) `get_numLinhas`: a função get_numLinhas calcula quantas linhas serão enviadas para as workers. Para isto, essa função recebe como parâmetro o id da worker (*destination*), o resto da divisão do número de linhas da matriz e, por fim, a média de linhas da matriz.

```c
int get_numLinhas(int idWorker, int resto, int mediaLinhas){
  if(idWorker <= resto )
    return mediaLinhas+1;
  else
    return mediaLinhas;
}
```

f) `enviar_para_workers`: essa função é responsável por enviar as matrizes A, B e C para as workers onde serão manipuladas. Como as matrizes foram alocadas dinamicamente, seus dados não encontram-se na memória de forma contínua. Dessa forma, o primeiro passo dessa função é transformar as matrizes B e C em vetores. Feito isso, é calculado quantas linhas da matriz A serão enviadas para cada worker e essa matriz também é transformada em um vetor. Assim, todos esses dados (o número de linhas da matriz A e as 3 matrizes convertidas para vetor) são enviados para as workers.

```
void enviar_para_workers(float *matrizA, float *matrizB, float *matrizC, int y, int w, int v, int numWorkers){
    /* Transforma a matriz B em um array */
    float *bufferB = malloc(w*v*sizeof(float *));
    for(int i=0; i<w; i++){
        for(int j=0; j<v; j++){
            bufferB[i*v+j] = matrizB[i*v+j];
        }
    }

    /* Transforma a matriz C em um array */
    float *bufferC = malloc(v*1*sizeof(float *));
    for(int i=0; i < v; i++){
      for(int j=0; j < 1; j++){
        bufferC[i*1+j] = matrizC[i*1+j];
      }
    }

    int mediaLinhas = y/numWorkers;
    int resto = y%numWorkers;
    int offset = 0;

    /* Divide a matriz A entre as workers */
    for(int idWorker = 1; idWorker <= numWorkers; idWorker++){
        int numLinhas = get_numLinhas(idWorker, resto, mediaLinhas);

        /* Transforma a matriz A em um array */
        float *bufferA = malloc(numLinhas*w*sizeof(float *));
        int index = 0;
        for(int linha=offset; linha < offset + numLinhas; linha++){
          for(int coluna=0; coluna < w; coluna++){
            bufferA[index++] = matrizA[linha*w+coluna];
          }
        }
        offset += numLinhas;


        MPI_Send(&numLinhas, 1, MPI_INT, idWorker, TAG, MPI_COMM_WORLD);
        MPI_Send(bufferA, numLinhas*w, MPI_FLOAT, idWorker, TAG, MPI_COMM_WORLD);
        MPI_Send(bufferB, w*v, MPI_FLOAT, idWorker, TAG, MPI_COMM_WORLD);
        MPI_Send(bufferC, v*1, MPI_FLOAT, idWorker, TAG, MPI_COMM_WORLD);

        free(bufferA);
    }

    free(bufferB);
    free(bufferC);
}
```

g) `receber_das_workers`: essa função é responsável por receber as respectivas multiplicações de matrizes feitas nas workers e a soma dos elementos das matrizes resultantes de cada multiplicação. Assim, é alocado dinamicamente um vetor para “agrupar” todas as multiplicações recebidas e, posteriormente, esses valores são passados para a matriz final (que é a matriz D). Além disso, para cada soma parcial recebida, esse valor é adicionado à variável *soma* para obter a redução pela soma total da matriz D.

```c
void receber_das_workers(float *matrizResultado, int y, int v, int numWorkers, MPI_Status status, double *soma){
  int mediaLinhas = y/numWorkers;
  int resto = y%numWorkers;
  int offset = 0;
  double somaParcial = 0;

  for(int idWorker=1; idWorker<=numWorkers; idWorker++){
        int numLinhas = get_numLinhas(idWorker, resto, mediaLinhas);

        float *bufferResultado = malloc(numLinhas*v*sizeof(float *));

        MPI_Recv(bufferResultado, numLinhas*v, MPI_FLOAT, idWorker, TAG, MPI_COMM_WORLD, &status);
        MPI_Recv(&somaParcial,1, MPI_DOUBLE, idWorker, TAG, MPI_COMM_WORLD, &status);
        *soma +=somaParcial;

        int index = 0;
        for(int linha=offset; linha < offset + numLinhas; linha++){
          for(int coluna=0; coluna < v; coluna++){
            matrizResultado[linha*v+coluna] = bufferResultado[index++];
          }
        }
        offset += numLinhas;

        free(bufferResultado);
  }
}
```

h) `main`: essa função é responsável por coordenar todas as demais funções presentes no programa. Nela são feitas as alocações dinâmicas de todas as matrizes, inicializadas as operações envolvendo a biblioteca MPI e realizado as divisões de processos entre a Master e as Workers.

```c
/* Variáveis relacionadas ao MPI */
MPI_Status status;
int numTasks, rank, numWorkers, tag;

/* Variáveis relacionadas as matrizes */
int y = atoi(argv[1]), w = atoi(argv[2]), v = atoi(argv[3]);
float *matrizA, *matrizB, *matrizC, *matrizD, *matrizAB;

/* Variáveis relacionadas aos nomes dos arquivos */
char *nomeArqA = argv[4], *nomeArqB = argv[5], *nomeArqC = argv[6], *nomeArqD = argv[7];

/* Variáveis relacionadas ao tempo de execução */
struct timeval inicio, fim;
long segundos, microssegundos;
double tempo, soma;

/* Inicializa as matrizes */
matrizA   = (float *) malloc(y * w * sizeof(float));
matrizB   = (float *) malloc(w * v * sizeof(float));
matrizC   = (float *) malloc(v * 1 * sizeof(float));
matrizD   = (float *) malloc(y * 1 * sizeof(float));

/* Preenche as matrizes com os valores dos respectivos arquivos */
gerar_matriz(nomeArqA,matrizA,y,w);
gerar_matriz(nomeArqB,matrizB,w,v);
gerar_matriz(nomeArqC,matrizC,v,1);

/* Inicializa o MPI */
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numTasks);
numWorkers = numTasks-1;
```

Em relação à **Master**, nela são realizadas os seguintes processos: cálculo do tempo de execução (grifado em vermelho), chamadas das funções *enviar_para_workers* e *receber_das_workers* (detalhadas anteriormente e grifadas em verde) e, por fim, gravação da matriz D no arquivo de saída e print do resultado da soma na tela (grifado em azul).

```
if(rank == 0){
   gettimeofday(&inicio, 0);
   enviar_para_workers(matrizA, matrizB, matrizC, y, w, v, numWorkers);
   receber_das_workers(matrizD, y, 1, numWorkers, status, &soma);

   gettimeofday(&fim, 0);

   segundos = fim.tv_sec - inicio.tv_sec;
   microssegundos = fim.tv_usec - inicio.tv_usec;
   tempo = segundos + microssegundos*1e-6;
   printf("As operacoes com matrizes levaram %f segundos para executar\n", tempo);

   /* Grava a matrizD em um arquivo */
    gravar_matriz(nomeArqD,matrizD,y,1);

    printf("%lf\n", soma);
}
```

Em relação às **Workers**, nelas são realizadas os seguintes processos: recebimento do número de linhas da matriz A, alocação dos vetores que representam cada matriz que será enviada e/ou gerada, recebimento das respectivas matrizes e, uma vez recebidas, é calculada a multiplicação dessas matrizes de acordo com o número de linhas de A que serão processadas (ou seja, cada worker será responsável por calcular a multiplicação de apenas uma parte da matriz). Feito isso, é calculada a redução pela soma da respectiva matriz resultante da multiplicação (como é feita somente a multiplicação de uma parte da matriz, essa soma também corresponde apenas a essa respectiva parte, e não ao todo, por isso ela foi denominada de *somaParcial*). Por fim, a matriz resultante e a soma são enviadas para a Master.

```c
if(rank > 0){
    int numLinhasA;
    MPI_Recv(&numLinhasA, 1, MPI_INT, MASTER, TAG, MPI_COMM_WORLD, &status);

    float *matrizA = malloc(numLinhasA*w*sizeof(float*));
    float *matrizB = malloc(w*v*sizeof(float*));
    float *matrizC = malloc(v*1*sizeof(float*));
    float *matrizAB = malloc(numLinhasA*v*sizeof(int *));
    float *matrizD = malloc(numLinhasA*1*sizeof(int *));

    MPI_Recv(matrizA, numLinhasA*w, MPI_FLOAT, MASTER, TAG, MPI_COMM_WORLD, &status);
    MPI_Recv(matrizB, w*v, MPI_FLOAT, MASTER, TAG, MPI_COMM_WORLD, &status);
    MPI_Recv(matrizC, v*1, MPI_FLOAT, MASTER, TAG, MPI_COMM_WORLD, &status);

    /* --- Multiplicação das Matrizes --- */
    multiplicar_matrizes(matrizA, matrizB, matrizAB, numLinhasA, w, v);
    multiplicar_matrizes(matrizAB, matrizC, matrizD, numLinhasA, v, 1);

    /* --- Redução pela Soma --- */
    double somaParcial = reducao_pela_soma(matrizD, numLinhasA, 1);

    MPI_Send(matrizD, numLinhasA*1, MPI_FLOAT, MASTER, TAG, MPI_COMM_WORLD);
    MPI_Send(&somaParcial, 1, MPI_DOUBLE, MASTER, TAG, MPI_COMM_WORLD);

    free(matrizA);
    free(matrizB);
    free(matrizAB);
    free(matrizC);
    free(matrizD);
}
```

4. RESULTADOS E CONCLUSÃO

Para avaliar como a paralelização de um programa utilizando a biblioteca MPI influencia no seu desempenho, foram realizados 5 testes com o código desenvolvido. Em cada teste, foi atribuído o mesmo valor para as variáveis y, w, v que representam os números de linhas e colunas de cada matriz. Após definido os valores, o mesmo código foi executado porém, na primeira execução, não considerou-se o paralelismo (compilado com o comando: `gcc -o programa programa.c`). Já na segunda execução, a paralelização foi considerada (compilado com o comando: **`mpicc`** `programa.c -o programa`). Para cada execução, os respectivos valores do tempo de execução foram registrados na tabela abaixo.

y	w	v	Tempo de Execução (sem paralelização)	Tempo de Execução (com paralelização)	Diferença % dos Tempos de Execução
10	10	10	0,000004	0,000085	2025,00%
100	100	100	0,002190	0,001995	-8,90%
1000	1000	1000	2,581925	0,906817	-64,88%
1500	1500	1500	11,147819	4,654019	-58,25%

2000	2000	2000	29,854409	11,739553	-60,68%
					-48,18%

Figura 1 - Tabela de Análise

Após o registro dos tempos de execução, foi calculada a diferença percentual entre os tempos obtidos executando o programa sem paralelização e executando o programa com paralelização. Esse cálculo percentual foi feito utilizando-se a seguinte fórmula:

$$\left(\frac{Tempo\ de\ Execução\ com\ Paralelização - Tempo\ de\ Execução\ sem\ Paralelização}{Tempo\ de\ Execução\ sem\ Paralelização} \right) \times 100$$

Após as execuções, foi possível notar através da avaliação da *tabela de análise* e do cálculo apresentado acima que, exceto para o caso em que y=w=v=10, o programa paralelizado levou um tempo bem menor para executar quando comparado ao programa não paralelizado.

Realizando uma média entre a diferença percentual dos tempos de execução considerando os valores de y, w e v maiores que 10, percebe-se que o programa que fez uso da biblioteca MPI executou as mesmas operações **48,18% mais rápido** do que o programa não paralelizado. Ou seja, utilizando-se uma aplicação com memória distribuída, obteve-se um desempenho quase **2x** melhor do que utilizando-se uma aplicação serial.

Abaixo, encontram-se três gráficos que demonstram essa diferença entre os tempos de execução. Como há uma diferença considerável entre os tempos de execução para y=w=v=10 e y=w=v=100 em relação aos demais valores, foram criados gráficos exclusivos para esses valores de x, y e w e, posteriormente, foi criado um gráfico que inclui todos os valores testados.

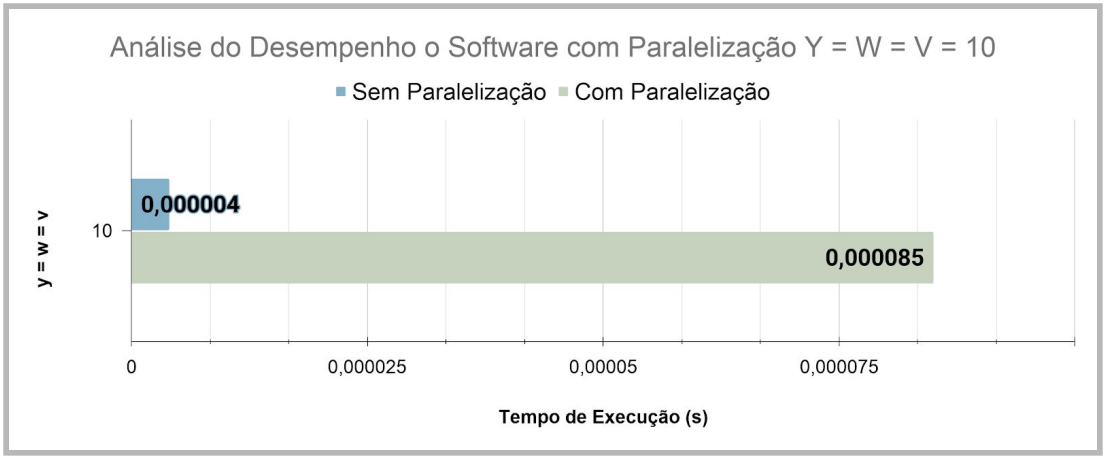

Figura 2 - Gráfico de desempenho para y = w = v =10

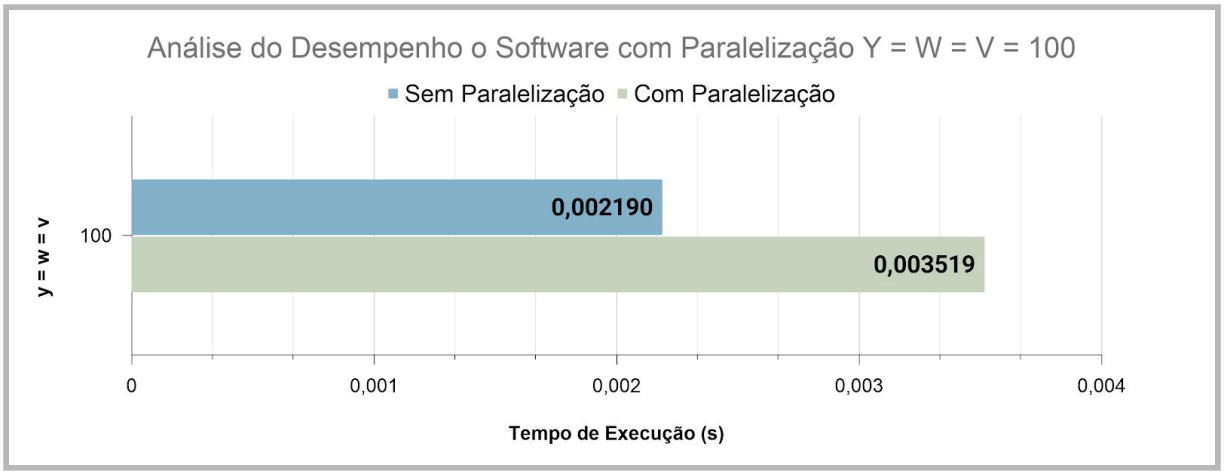

Figura 3 - Gráfico de desempenho para y = w = v =100

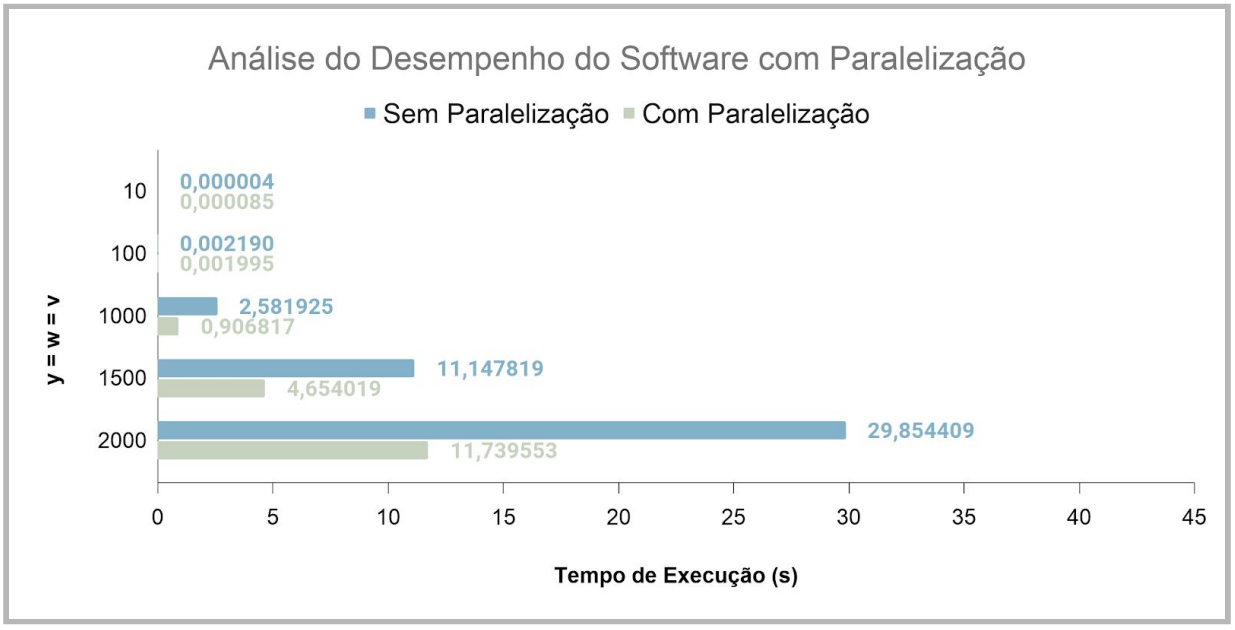

Figura 4 - Gráfico de desempenho para todos os valores de y, w e v

O gráfico abaixo compara as três bibliotecas utilizadas durante o semestre: openMP, openACC e MPI:

y	w	v	Tempo de Execução (com paralelização)		
			OpenMP	OpenACC	MPI
10	10	10	0,000141	0,000014	0,000085
100	100	100	0,001190	0,003519	0,001995
1000	1000	1000	1,044124	3,595931	0,906817
1500	1500	1500	4,536605	15,190705	4,654019
2000	2000	2000	11,202972	37,267058	11,739553
Diferença % em Relação ao Código Não-Paralelizado			**-57,47%**	**40,26%**	**-48,18%**

Figura 5- Gráfico de desempenho de todas as bibliotecas

Assim, baseando-se nos tempos de execução obtidos, nos dados apresentados na tabela, na análise dos gráficos e na comparação entre as diferentes bibliotecas, concluiu-se que a paralelização do programa desenvolvido utilizando a biblioteca MPI apresentou uma grande vantagem quando comparada à biblioteca OpenACC, uma vez que, além possuir um tempo de execução muito menor (ou seja, ser muito mais rápida), ela também obteve vantagem sobre o programa serial (caso que não ocorreu no openACC). Em relação ao OpenMP, vemos que há uma pequena diferença entre os tempos de execução, onde o OpenMP apresentou uma pequena vantagem quando comparadas as diferenças percentuais de cada biblioteca em relação ao programa serial.

Dessa forma, podemos dizer que dentre todas as bibliotecas testadas, a OpenMP gerou os melhores resultados, sendo seguida pela MPI e, por fim, pela OpenACC.

5. TUTORIAIS

5.1. COMPILANDO

Há várias formas de compilar um programa escrito em linguagem C, abaixo encontram-se algumas destas formas.

5.1.1 COMPILANDO VIA TERMINAL DO LINUX

Para compilar o programa via terminal do Linux usando a biblioteca MPI deve-se seguir os seguintes passos:

1. *Digite o comando abaixo no terminal do Linux para compilar o programa das matrizes:*

```
mpicc programa.c -o programa
```

Onde:

- `mpicc` → compila o programa escrito em linguagem C utilizando a biblioteca MPI.
- `programa.c` → nome do arquivo fonte escrito em linguagem C que será utilizado para gerar o arquivo executável.
- `-o programa` → define que o programa executável se chamará "programa".

5.1.2. COMPILANDO VIA SCRIPT

Para automatizar o processo de compilação foi criado um script do tipo SH. O arquivo de script recebeu o nome de `script.sh` e, antes de executá-lo, é preciso **garantir que ele esteja na mesma pasta que o programa escrito em linguagem C**. Após isso, é necessário alterar a permissão de administrador ao script para garantir que não haja nenhuma falha durante a fase de compilação. Para conferir as devidas permissões ao script basta digitar e executar o código abaixo no terminal do Linux:

```
chmod +x  script.sh
```

Por fim, para executar o programa basta digitar e executar o seguinte código:

```
./script.sh
```

5.1.2. COMPILANDO VIA MAKEFILE

Para automatizar o processo de compilação foi criado um arquivo makefile. O arquivo de makefile recebeu o nome padrão `makefile` e, antes de executá-lo, é preciso **garantir que ele esteja na mesma pasta que o programa escrito em linguagem C**. Após isso, é necessário digitar e executar o seguinte comando no terminal do Linux para compilar o programa:

```
make
```

5.2. EXECUTANDO O PROGRAMA

1. *Execute o programa das matrizes utilizando os comandos abaixo:*

```
mpirun -n 4 ./programa y w v arqA.dat arqB.dat arqC.dat arqD.dat
```

Onde:

- mpirun → comando que executa o programa gerado anteriormente utilizando a biblioteca MPI .
- -n 4 → define que o paralelismo será executado em 4 núcleos.
- `./programa` → o programa gerado anteriormente.
- `y` → número de linhas da matriz A.
- `w` → número de colunas da matriz A e número de linhas da matriz B.
- `v` → número de colunas da matriz B e número de linhas da matriz C.
- `arqA.dat` → nome do arquivo que contém os valores da matriz A_{yxw}.
- `arqB.dat` → nome do arquivo que contém os valores da matriz B_{wxv}.
- `arqC.dat` → nome do arquivo que contém os valores da matriz C_{vx1}.
- `arqD.dat` → nome do arquivo que contém os valores da matriz resultante D_{yx1}.

5.3. GERANDO ARQUIVOS ALEATÓRIOS

Para auxiliar na execução do programa principal, foi criado um segundo programa chamado “arquivos_aleatorios.c” que é responsável por gerar os arquivos com valores aleatórios. Para compilar o programa auxiliar, basta digitar os comandos abaixo no terminal do Linux:

```
gcc -o arquivos_aleatorios arquivos_aleatorios.c
```

Onde:

- `gcc` → executa o compilador GCC.
- `-o arquivos_aleatorios` → define que o programa executável se chamará *arquivos_aleatorios*.
- `arquivos_aleatorios.c` → nome do arquivo fonte escrito em linguagem C que será utilizado para gerar o arquivo executável.

Para executar o programa gerado anteriormente basta utilizar o código abaixo no terminal do Linux:

```
./arquivos_aleatorios y w v arqA.dat arqB.dat arqC.dat
```

Onde:

- `./arquivos_aleatorios` → o programa gerado anteriormente.
- `y` → número de linhas da matriz A.
- `w` → número de colunas da matriz A e número de linhas da matriz B.
- `v` → número de colunas da matriz B e número de linhas da matriz C.
- `arqA.dat` → nome do arquivo que contém os valores da matriz A_{yxw}.

- `arqB.dat` → nome do arquivo que contém os valores da matriz B_{wxv}.
- `arqC.dat` → nome do arquivo que contém os valores da matriz C_{vx1}.

O programa principal pode utilizar os arquivos com valores aleatórios gerados pelo programa auxiliar “arquivos_aleatorios.c” para preencher suas matrizes desde que:

- Os valores de `y,w,v` na execução do programa principal sejam iguais ao valores de `y,w,v` na execução do programa *arquivos_aleatorios.*
- O nome dos arquivos `arqA.dat, arqB.dat e arqC.dat` sejam iguais na execução do programa principal e na execução do programa auxiliar para que, dessa forma, o programa principal possa ler os dados armazenados nos arquivos gerado pelo programa auxiliar *arquivos_aleatorios.*

5.4. GITHUB

O código fonte escrito em Linguagem C assim como o arquivo makefile e o script SH encontram-se no seguinte repositório no GitHub na pasta(diretório) MPI:

https://github.com/matheusaleso/PAD

6. REFERÊNCIAS

GRADVOHL, André Leon Sampaio. Teste de alocação de matrizes. GitHub. Disponível em: <https://gradvohl.github.io/alocaMatrizes/>. Acesso em: 22 de out. de 2020