SSC0903
Computação de Alto Desempenho
(2023-2) BCC Turma B

2º Trabalho Prático – MPI

Membros:

- Eduardo Garcia de Gáspari Valdejão, 11795676
- Henrico Lazuroz Moura de Almeida, 12543502
- Luca Gomes Urssi, 10425396
- Victor Lucas de Almeida Fernandes, 12675399

Profs. Responsáveis:

- Paulo Sergio Lopes de Souza
- Sarita Mazzini Bruschi

Github:

- https://github.com/lucaurssi/SSC0903-CAD

Indice:

- **Implementação Sequencial 1**
- **Implementação Paralela com OpenMP 4**
- **Experimentação e análise dos resultados 7**

Implementação Sequencial:

Começamos a implementação sequencial criando uma matriz vazia (***calloc*** garante que é preenchida com 0's) de tamanho **MAT_SIZE** recebido na entrada do programa. Iniciamos o contador de tempo e criamos as variáveis onde colocaremos a saída do programa.

```
int MAT_SIZE = atoi(argv[1]);

// create matrix - initialized with zeros
int **mat = (int**)calloc (MAT_SIZE, sizeof (int*));
for (int i=0; i < MAT_SIZE; i++)
    mat[i] = (int*)calloc (MAT_SIZE, sizeof (int));

clock_t time = clock();
double total_time;

int max = 0;
int min = MAT_SIZE*2;
long int sum = 0;
int *sum_lin = (int*)calloc (MAT_SIZE, sizeof (int));
int *sum_col = (int*)calloc (MAT_SIZE, sizeof (int));
```

A seguir temos um loop que passa 100 vezes para achar uma média do tempo de execução do programa.

```
for(int k=0; k<100; k++){ // multiple runs

    max = 0;
    min = MAT_SIZE*2;
    sum = 0;
    memset(sum_lin, 0, MAT_SIZE * sizeof(int));
    memset(sum_col, 0, MAT_SIZE * sizeof(int));
```

No começo do loop iniciamos os valores de saída para não interferir com os resultados.

O primeiro **for** do loop enche a matriz com **i + j** para cada coluna **i** e linha **j** da matriz. Colocamos esse **for** dentro do loop para ser repetido apesar de 99 das vezes ele apenas reescrever o mesmo valor, pois esse mesmo **for** no código paralelo está sendo otimizado, então para levar em conta o tempo salvo estamos incluindo ele no loop.

O segundo loop na imagem abaixo faz as operações pedidas no enunciado.

```
    // populating 'mat' with i + j
    for(int i=0; i<MAT_SIZE; i++)
        for(int j=0; j<MAT_SIZE; j++)
            mat[i][j] = i + j;

    // actual loop
    for(int i=0; i<MAT_SIZE; i++)
        for(int j=0; j<MAT_SIZE; j++){
            if(mat[i][j] > max)
                max = mat[i][j];
            else
                if(mat[i][j] < min)
                    min = mat[i][j];

            sum += mat[i][j];

            sum_lin[j] += mat[i][j];
            sum_col[i] += mat[i][j];
        }
}
```

Terminamos o código imprimindo o tempo de 100 execuções e liberando a memória.

```
total_time = (double)(clock() - time)/CLOCKS_PER_SEC;
printf("time: %f\n", total_time);

if(PRINT){
    printf("max : %d\n", max);
    printf("min : %d\n", min);
    printf("sum : %ld\n", sum);
    printf("sum_lin :\n");
    for(int i=0; i<MAT_SIZE; i++)
        ;//printf("%d ", sum_lin[i]);
    printf("\nsum_col :\n");
    for(int i=0; i<MAT_SIZE; i++)
        ;//printf("%d ", sum_col[i]);
}



for(int i=0; i<MAT_SIZE; i++)
    free(mat[i]);
free(mat);
free(sum_col);
free(sum_lin);

return 0;
```

Implementação Paralela com MPI:

Iniciamos criando os processos logo antes do primeiro local que lida com memória local, no caso a criação da matriz, que todas as threads precisarão ter.

```
int MAT_SIZE = atoi(argv[1]);

MPI_Init(&argc, &argv);

// create matrix - initialized with zeros
int **mat = (int**)calloc (MAT_SIZE, sizeof (int*));
for (int i=0; i < MAT_SIZE; i++)
    mat[i] = (int*)calloc (MAT_SIZE, sizeof (int));
```

Diferente do código sequencial temos que criar muitas outras variáveis para efetuar a comunicação entre os processos. As variáveis mais notáveis são:

1. **myrank** (identifica o processo em relação aos outros dentro do mesmo comunicador)
2. **mat_size_por_processo** (que é o tamanho da matriz dividido pelo número de processos)
3. **global_***(5 variaveis que vão armazenar os valores finais a serão imprimidos)

```
clock_t time = clock();
double total_time;

int npes, myrank, ret;
char processor_name[MPI_MAX_PROCESSOR_NAME];
int name_len;

MPI_Comm_size(MPI_COMM_WORLD, &npes); // retorna o numero de processos
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  // retonaro rank do processo
MPI_Get_processor_name(processor_name, &name_len);

int mat_size_por_processo = MAT_SIZE  / npes;
int inicio = myrank * mat_size_por_processo;

int max = 0;
int min = MAT_SIZE*2;
long int sum = 0;
int *sum_lin = (int*)calloc (MAT_SIZE, sizeof (int));
int *sum_col = (int*)calloc (MAT_SIZE, sizeof (int));

int global_max=0;
int global_min=0;
long int global_sum=0;
int *global_lin = (int*)calloc (MAT_SIZE, sizeof (int));
int *global_col = (int*)calloc (MAT_SIZE, sizeof (int));
```

Assim como no sequencial iniciamos novamente as variáveis em cada iteração do loop.

```
for(int k=0; k<100; k++){ // multiple runs

    max = 0;
    min = MAT_SIZE*2;
    sum = 0;
    memset(sum_lin, 0, MAT_SIZE * sizeof(int));
    memset(sum_col, 0, MAT_SIZE * sizeof(int));

    global_max=0;
    global_min=0;
    global_sum=0;
    memset(global_lin, 0, MAT_SIZE * sizeof(int));
    memset(global_col, 0, MAT_SIZE * sizeof(int));
```

Como podemos ver abaixo, a única diferença do código sequencial é que cada processo faz apenas um trecho do loop, começando em **inicio** e andando **mat_size_por_processo**.

```
// inside loop because its improved with parallelism
for(int i=inicio; i<inicio + mat_size_por_processo; i++)
    for(int j=0; j<MAT_SIZE; j++)
        mat[i][j] = i + j;

// actual loop
for(int i=inicio; i<inicio + mat_size_por_processo; i++)
    for(int j=0; j<MAT_SIZE; j++){
        if(mat[i][j] > max)
            max = mat[i][j];
        else
            if(mat[i][j] < min)
                min = mat[i][j];

        sum += mat[i][j];

        sum_lin[j] += mat[i][j];
        sum_col[i] += mat[i][j];
    }
```

E antes de acabar o loop, passamos os valores encontrados para o processo de rank 0.

```
            sum_lin[j] += mat[i][j];
            sum_col[i] += mat[i][j];
        }

    MPI_Reduce(&max, &global_max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Reduce(&min, &global_min, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);
    MPI_Reduce(&sum, &global_sum, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(sum_lin, global_lin, MAT_SIZE, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(sum_col, global_col, MAT_SIZE, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
}
```

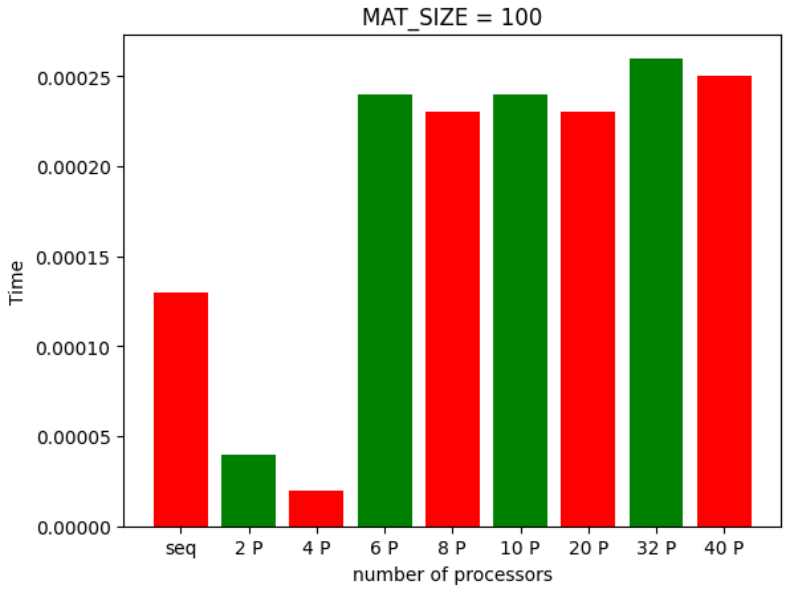
O final do código se diferencia do sequencial no fato de que temos que especificar apenas um dos processos para imprimir as respostas.

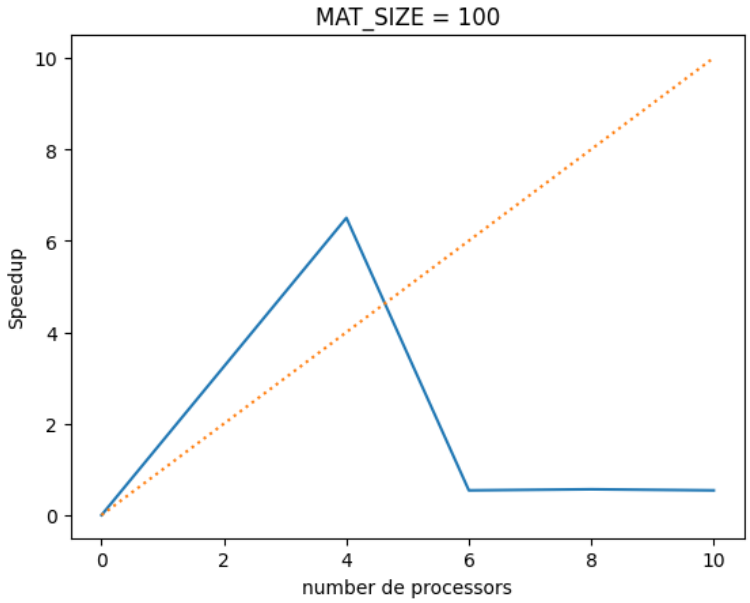
```
total_time = (double)(clock() - time)/CLOCKS_PER_SEC;
if(myrank == 0)
    printf("time: %f\n", total_time);

if(myrank == 0 && PRINT){

    printf("max : %d\n", global_max);
    printf("min : %d\n", global_min);
    printf("sum : %ld\n", global_sum);
    printf("global_lin :\n");
    for(int i=0; i<MAT_SIZE; i++)
        ;//printf("%d ", global_lin[i]);
    printf("\nglobal_col :\n");
    for(int i=0; i<MAT_SIZE; i++)
        ;//printf("%d ", global_col[i]);
}

ret = MPI_Finalize();

for(int i=0; i<MAT_SIZE; i++)
    free(mat[i]);
free(mat);
free(sum_col);
free(sum_lin);
free(global_lin);
free(global_col);


if (ret == MPI_SUCCESS && PRINT)
    printf("MPI_Finalize success from myrank %d from host %s\n", myrank, processor_name);

return 0;
```

Experimentação e análise dos resultados:

Utilizamos o Cluster para testar nosso código, com 10 nós e cada nó tem 4 núcleos.

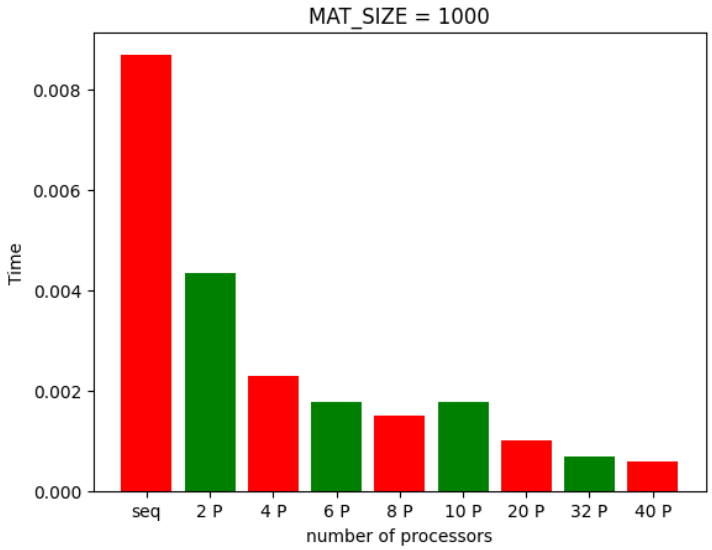
Decidimos testar o código mudando o tamanho da matriz e o número de processos disponibilizados. **Nota-se que em cada teste nosso programa roda 100 vezes.** O primeiro tamanho de matriz escolhido foi 100x100, onde podemos observar que o custo de paralelizar aumenta assim que há uma necessidade de conectar com outras máquinas.

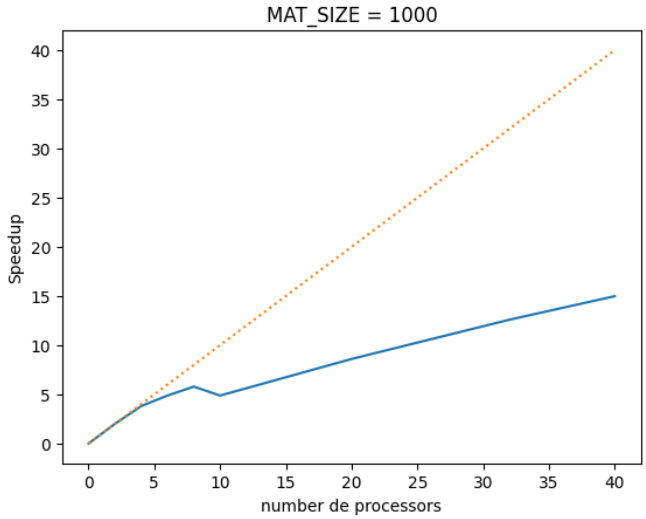




Podemos observar um comportamento superlinear em até 4 processos, onde o cluster não precisou ainda utilizar mais do que uma máquina.

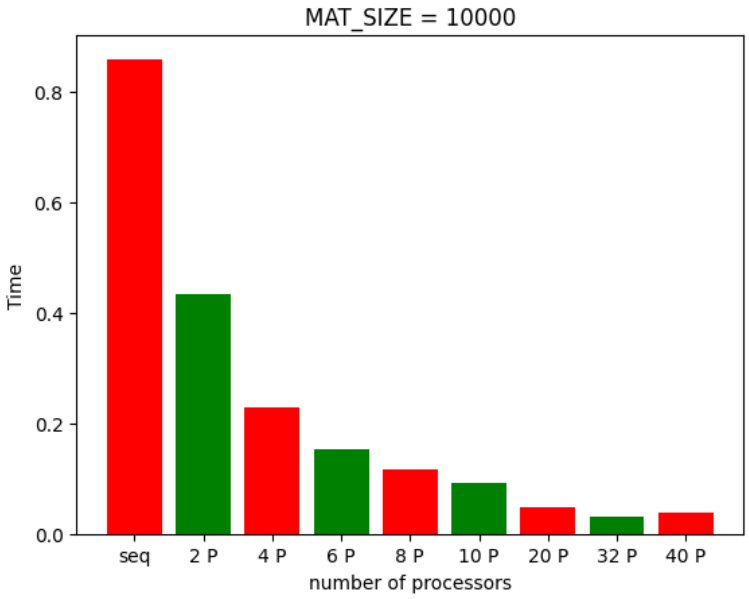
No segundo teste utilizamos o tamanho 1000x1000, onde tivemos os primeiros ganhos de desempenho com mais do que 4 processos.

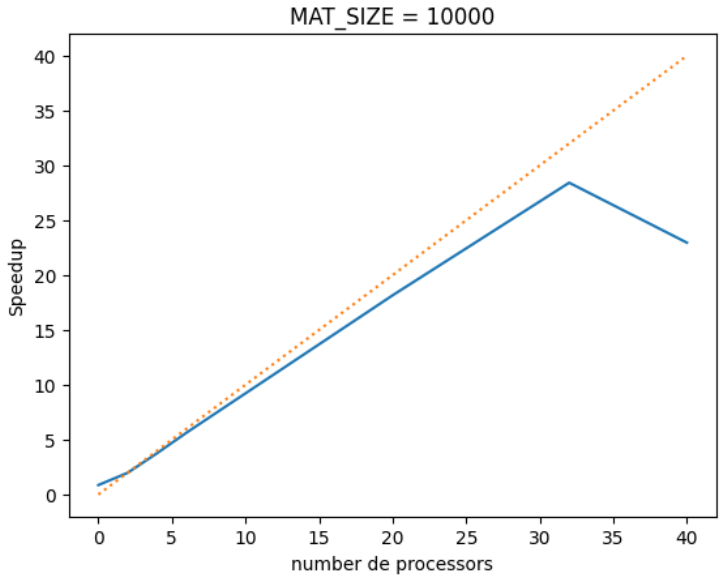




Podemos observar que o código é sublinear a partir de 5 processos em diante, onde a comunicação interna do cluster é necessária. Apesar disso, o desempenho se mantém muito melhor que o sequencial.

No terceiro teste utilizamos uma matriz de 10000x10000. Neste teste podemos observar um desempenho maior do código paralelo, se aproximando de um speedup linear, infelizmente com 40 processadores notamos uma queda no desempenho que supomos ser por causa do custo de comunicação elevado.





Dado o tempo de execução elevado, não prosseguimos com testes maiores.