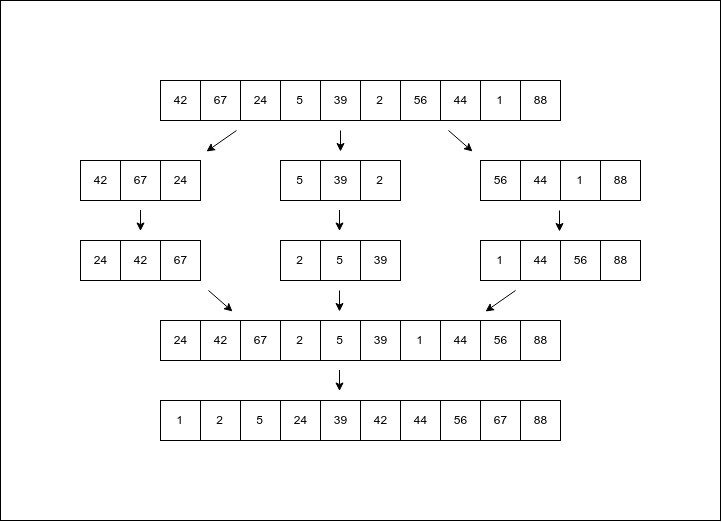
**1) Implementação**

Visando consolidar o aprendizado da paralelização de programas utilizando MPI, foi proposto executar a ordenação de um vetor no cluster Grad do LAD. Para executar o programa foi definido arbitrariamente um vetor de **250.000** posições, de forma que fosse possível verificar se há vantagem ou não em paralelizar o programa.

O processo de divisão do trabalho entre os nós foi realizado conforme o seguinte processo, no modelo mestre-escravo: O nó mestre **m** divide o tamanho total **t** do vetor pelo número **n-1** de processos (resultado **s**), e calcula também o resto **r** da divisão de **t** por **n**. Dessa forma, **n-2** nós vão ordenar um vetor de tamanho **s**, um dos nós **n** vai ordenar um vetor de tamanho **s**+**r** e o nó **m** não realiza nenhum trabalho. Depois os nós escravos enviam os vetores menores para o nó mestre, que junta os pedaços em um único vetor novamente e depois realiza o ordenamento desse vetor. Este processo está demonstrado na figura 1. O ordenamento dos vetores menores foi feito com o algoritmo Bubble Sort, e o vetor final com o algorito Heap Sort.

**Figura 1 Processo Ordenamento**

**2) Dificuldades encontradas**

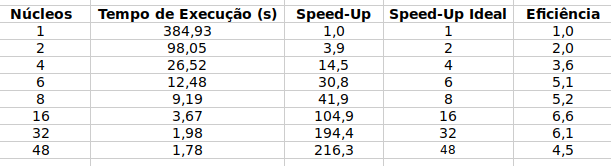
Compreender o funcionamento da paralelização com MPI e debug do código paralelizado.

**3) Testes**

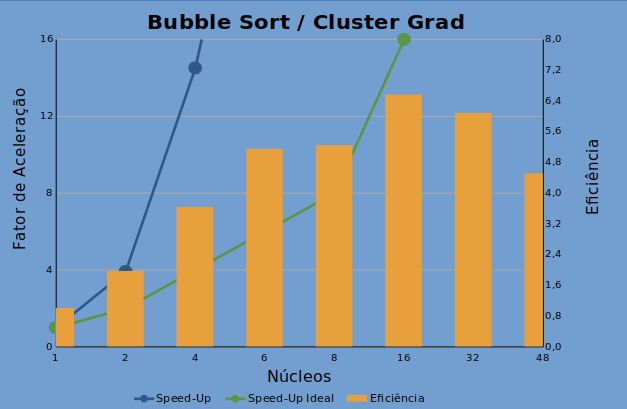
Inicialmente o programa foi executado de forma sequencial para obter o tempo de ordenamento do vetor. Em seguida o programa foi executado para 2, 4, 6, 8, 16, 32, e 48 processos, verificando-se o tempo de execução de cada um deles.

**4) Análise de Desempenho**

A versão paralela do ordenamento obteve um desempenho sigificantemente superior a sua versão sequencial, conforme demonstrado nas Figuras 2 e 3.

**Figura 2: Resultados observados**

Observa-se que a maior eficiência do programa é quando são executados 16 processos paralelos. Ainda que o *speed-up* para mais processos seja maior, o programa perde eficiência, o que pode não compensar para a execução do programa paralelo. Observa-se também que o *speed-up* real supera e muito o *speed-up* ideal, o que demonstra o real benefício de paralelizar as tarefas.

**Figura 3: Resultados observados**

**4) Observações finais**

Utilizar processamento paralelo em um algoritmo de ordenamento como o Bubble Sort mostrou-se extremamente vantajoso, diminuindo o tempo de ordenamento de um vetor de 250.000 elementos de 6 minutos para 9 segundos no auge da eficiência do programa. Dessa forma, percebe-se o quão benéfico é a paralelização das tarefas.

**Trecho paralelo do programa**

if(my\_Rank == 0) {

imprimeArquivo(vetor,tamanhoVetor, 'o',

imprimeSaida);

for(rank = 1 ; rank < proc\_size; rank++) {

if(rank == proc\_size-1){

divideVetor(vetor, tamanhoVetor,

ultPosVetSpl, tamUltPos, offset);

ret = MPI\_Send(ultPosVetSpl,

tamUltPos \* sizeof(int), MPI\_CHAR,rank, 0,

MPI\_COMM\_WORLD);

if(ret != MPI\_SUCCESS) {

mpi\_err(1,"MPI\_Send");

}

} else {

divideVetor(vetor, tamanhoVetor,

nVetSpl, tamVetSplit, offset);

offset += tamVetSplit;

ret = MPI\_Send(nVetSpl,

tamVetSplit \* sizeof(int),

MPI\_CHAR, rank, 0,

MPI\_COMM\_WORLD);

if(ret != MPI\_SUCCESS) {

mpi\_err(1,"MPI\_Send");

}

}

}

offset = 0;

for(rank = 1 ; rank < proc\_size ; rank++) {

if(rank == proc\_size-1){

ret = MPI\_Recv(ultPosVetSpl,

tamUltPos \* sizeof(int),

MPI\_CHAR,rank,0,

MPI\_COMM\_WORLD, &status);

if(ret != MPI\_SUCCESS){

mpi\_err(1,"MPI\_Recv");

}

juntaVetores(ultPosVetSpl,

tamUltPos, vetorFinal, offset);

} else {

ret = MPI\_Recv(nVetSpl,

tamVetSplit \* sizeof(int), MPI\_CHAR,rank,0,

MPI\_COMM\_WORLD,

&status);

if(ret != MPI\_SUCCESS) {

mpi\_err(1,"MPI\_Recv");

}

juntaVetores(nVetSpl, tamVetSplit,

vetorFinal, offset);

offset += tamVetSplit;

}

}

imprimeArquivo(vetorFinal,

tamanhoVetor, 'n', imprimeSaida);

sort\_after\_mpi(vetorFinal,

tamanhoVetor,

tamVetSplit,proc\_size);

if(imprimeSaida == 1) {

printf("Vetor ordenado impresso em saida.txt\t");

}

imprimeArquivo(vetorFinal,

tamanhoVetor, 'r', imprimeSaida);

} else {

ret = MPI\_Comm\_rank(

MPI\_COMM\_WORLD,

&my\_Rank);

if(ret != MPI\_SUCCESS){

mpi\_err(1,"MPI\_Comm\_rank");

}

if(my\_Rank == proc\_size-1) {

ret = MPI\_Recv(ultPosVetSpl,

tamUltPos \* sizeof(int), MPI\_CHAR,MPI\_ANY\_SOURCE,

0,MPI\_COMM\_WORLD, &status);

if(ret != MPI\_SUCCESS){

mpi\_err(1,"MPI\_Recv");

}

bubbleSort(ultPosVetSpl,

tamUltPos);

ret = MPI\_Send(ultPosVetSpl,

tamUltPos\*sizeof(int),MPI\_CHAR,

0, 0, MPI\_COMM\_WORLD);

if(ret != MPI\_SUCCESS){

mpi\_err(1,"MPI\_Send");

}

} else {

ret = MPI\_Recv(nVetSpl,

tamVetSplit \* sizeof(int), MPI\_CHAR,

MPI\_ANY\_SOURCE,0,

MPI\_COMM\_WORLD, &status);

if(ret != MPI\_SUCCESS){

mpi\_err(1,"MPI\_Recv");

}

bubbleSort(nVetSpl, tamVetSplit);

ret = MPI\_Send(nVetSpl,

tamVetSplit \* sizeof(int), MPI\_CHAR, 0, 0,

MPI\_COMM\_WORLD);

if(ret != MPI\_SUCCESS){

mpi\_err(1,"MPI\_Send");

}

}

}

O código completo está disponível em <https://github.com/brbmendes/sort_mpi>