Ordenamento com MPI

Relatório de entrega do trabalho 2

Disciplina: Programação Paralela **Professor:** Rodrigo Cataldo
Alunos: Bruno Bragança e Evandro Santos **Data:** 12/10/2019

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.

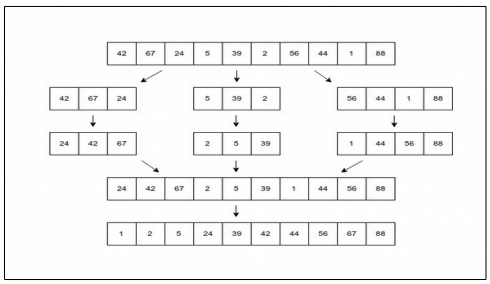

Figura 1 Processo Ordenamento

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.

Núcleos	Tempo de Execução (s)	Speed-Up	Speed-Up Ideal	Eficiência
1	384,93	1,0	1	1,0
2	98,05	3,9	2	2,0
4	26,52	14,5	4	3,6
6	12,48	30,8	6	5,1
8	9,19	41,9	8	5,2
16	3,67	104,9	16	6,6
32	1,98	194,4	32	6,1
48	1,78	216,3	48	4,5

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.

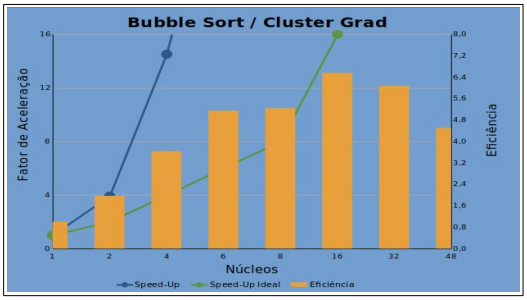

Figura 3: Resultados observados

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.

Ordenamento com MPI

Relatório de entrega do trabalho 2

Disciplina: Programação Paralela **Professor:** Rodrigo Cataldo
Alunos: Bruno Bragança e Evandro Santos **Data:** 12/10/2019

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