Relatório de Entrega de Trabalho
Disciplina de Programação Paralela (PP) – Prof. César De Rose

Alunos: João Langhanz – Paulo Jitsukawa
Usuários: pp12810 – pp12815
Exercicio: trabalho 1 de MPI (ME)
Entrega: 27/04/2017

1) Introdução

Com o objetivo de avaliar o desempenho de um algoritmo de ordenação em um ambiente de processamento distribuído de alto desempenho, foi implementado um algoritmo para realizar a distribuição de carga de trabalho em diversos processos, para que estes executem suas tarefas separadamente. O problema a ser resolvido baseia-se em um vetor de 1000 posições, chamado de saco de trabalho, e em cada posição outro vetor de 100.000 elementos. Cada vetor de 100.000 deve ser enviado a outro processo, ordenado, enviado de volta e colocado de volta em sua posição original no saco de trabalho.

2) Modelo de Programação

O programa desenvolvido utiliza o modelo de programação Mestre-Escravo, que consiste em um processo mestre ficar encarregado de distribuir trabalho para os processos escravos. No algoritmo implementado o mestre distribui uma rajada de trabalho do saco de trabalho para todos os escravos logo no início, já que eles, ao iniciarem, não tem tarefas executando. A seguir o mestre fica aguardando os escravos responderem com o trabalho feito, ao receber a resposta, o mestre coloca o trabalho feito de volta do saco de trabalho, e assume que o processo escravo que respondeu não tem mais tarefas executando, logo, o mestre envia o próximo vetor do saco de trabalho para este escravo. Quando o saco de trabalho acabar o mestre manda uma mensagem de suicídio aos escravos e o algoritmo termina. Dessa forma o processo mestre coordena a distribuição de trabalho e os processos escravos realizam a ordenação. Os vetores do saco de trabalho são inicializados com com números inteiros de maneira decrescente, a fim de termos o pior caso de uma ordenação.

3) Testes

Foram realizados testes com a alocação de 2 nós do cluster *grad*, que possui 8 processadores físicos por nó, executando o algoritmo com sequencial e com 2, 4, 8, 16, 32 e 64 processos paralelos, sendo medido o tempo que cada um levou para terminar. Para fins de comparação tambem foi executado o algoritmo com saco de trabalho de 1000 por 1.000.000 de elementos.

4) Análise de Desempenho

A versão sequencial do algoritmo teve um desempenho melhor que a com dois processos, isso porque ela não tem troca de mensagens. Com 4 processos obteve-se um bom ganho de desempenho se comparado com o sequencial e com 2 processos, assim como o de 8 processos em relação com 4 processos. Com 16 processos o tempo medido foi muito semelhante ao de 8, o que fez com que a eficiência caísse pela metade. Uma das possíveis razões que foram levadas em consideração para que isso ocorra é a grande quantidade de mensagens trocadas entre tantos processos, mas ao realizar uma conta rápida notou-se que são gastas duas mensagens para cada vetor do saco, logo, a quantidade de mensagens aumenta em função do tamanho do saco de trabalho e não da quantidade de processos. A quantidade de processos poderia influenciar nesse aspecto caso ocorra algum tipo de concorrência pelo meio de comunicação, onde mais mensagens ao mesmo tempo causaram um *delay*.

Outra possível, e mais provável, razão é o fato de que todos os processos precisam se comunicar com o mestre, que é apenas um processo, e que atua lendo e enviando mensagens sequencialmente, independente de quantos escravos paralelos estão sendo executados, gerando uma condição de corrida dos escravos pelo mestre. Então, quando aumentamos a quantidade de escravos as mensagens se acumulam na fila do processo mestre, que trata uma a uma, e enquanto isso os processos escravos estão ociosos esperando por mais trabalho. Se esta possibilidade for real, então há um grande desperdício de recursos, onde o mestre fica sobrecarregado lidando com mensagens e os escravos acabam ficando ociosos aguardando mais trabalho. Um gráfico de speed-up pode ser observado na figura 1.

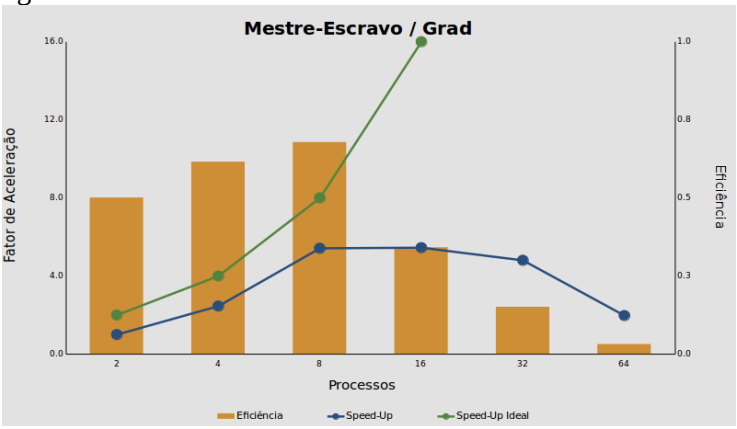

Figura 1 – resultados observados

Para 32 processos o cluster utiliza Hyper-Threading (HT), fazendo a eficiência cair ainda mais, por volta de 50%. A queda de desempenho se deve ao fato de que o HT aumenta o tempo de latência total, mas os efeitos negativos se tornam menores já que há mais *threads* rodando simultaneamente. Para 64 processos o tempo de execução piora significativamente e o desempenho cai ainda mais, com 4 processos rodando por nucleo, torna-se muito oneroso, tanto de ordenção nos escravos quando a troca de mensagens com o mestre.

Também realizamos testes com uma carga de trabalho maior, com 1.000.000 de elementos para serem ordenados nos escravos, que apresentaram tempos significativamente maiores, em média 10 vezes, mas levando em consideração que aumentamos a carga também em 10 vezes, poderíamos dizer que não houve uma queda de eficiência significativa.

5) Observações Finais

Com os testes realizados podemos chegar a conclusão de que a modelagem Mestre-Escravo escala bem até certo ponto, mas ao passar deste ponto o mestre se torna um gargalo, não conseguindo lidar com tantos escravos, deixando-os ociosos esperando por trabalho quando poderiam estar realizando alguma tarefa. Porém, quando aumentamos a carga não há uma queda de desempenho, pois os escravos ficam mais tempo ocupados.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#include <math.h>

#define COL 100000
#define LIN 1000

int cmpfunc (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}

int main(int argc, char** argv){
	int my_rank;       // Identificador deste processo
	int proc_n;        // Numero de processos disparados pelo usuario na linha de comando (np)
	int message[10];   // Buffer para as mensagens
	MPI_Status status; // estrutura que guarda o estado de retorno
	MPI_Status status2; // estrutura que guarda o estado de retorno
	MPI_Init(&argc , &argv); // funcao que inicializa o MPI, todo o codigo paralelo estah abaixo
	MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // pega pega o numero do processo atual (rank)
	MPI_Comm_size(MPI_COMM_WORLD, &proc_n);  // pega informacao do numero de processos (quantidade total)
	int resp[10];
	int f[proc_n];
	int *v =  (int*)malloc(COL*sizeof(int));

	double t1,t2;
	t1 = MPI_Wtime();  // inicia a contagem do tempo

	if( my_rank == 0 ){ //MESTRE
		printf("[M] sou o Mestre\n\n");

		int **saco_de_trabalho = (int**)malloc(LIN*sizeof(int*));
		int i,j;
		for (i=0 ; i<LIN; i++){
			saco_de_trabalho[i] = (int*)malloc(COL*sizeof(int));
			for (j=0 ; j<COL; j++){
				saco_de_trabalho[i][j] = COL-j;
			}
		}
		long order=1;
		int count = 0;
		//Envia todos
		for(order=1;order<proc_n;order++){
			MPI_Send(saco_de_trabalho[order-1], COL, MPI_INT, order, order, MPI_COMM_WORLD);
		}
		int tag = 0;
		int source;
		while(count<LIN && order <= LIN+proc_n-1){

			MPI_Probe(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD, &status2);
			tag = (int)(status2.MPI_TAG);
			if(tag > 0){

				MPI_Recv(saco_de_trabalho[tag-1], COL, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);
				source = status.MPI_SOURCE;
				if(count<LIN && order <= LIN) {
					MPI_Send(saco_de_trabalho[order-1], COL, MPI_INT, source, order, MPI_COMM_WORLD);

				}
				order++;
				count++;
		if(count % 50 == 0) printf(" ordenado %d\n",count);
			}
		}

		//Mata todos
		for(y=1;y<proc_n;y++){
			MPI_Send(saco_de_trabalho[0], 1, MPI_INT, y, 0, MPI_COMM_WORLD);
		}

		t2 = MPI_Wtime(); // termina a contagem do tempo
		printf("\n[M] Tempo de execucao: %f\n\n", t2-t1);
		MPI_Finalize();

	}else{ //ESCRAVO
		int tag=0;
		while(1){
			MPI_Recv(v, COL, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
			tag = status.MPI_TAG;
			// printf("[E] tag: %d\n",tag);
			if(tag >= 1){
				qsort(v,COL,sizeof(int),cmpfunc);
				MPI_Send(v, COL, MPI_INT, 0, tag, MPI_COMM_WORLD);

			}else{
				break;
			}
		}
		MPI_Finalize();
	}
return 0;
}
```