CENTRO UNIVERSITÁRIO FEI

VICTOR BIAZON
RA: 119.115-4

RELATÓRIO VII – PROGRAMAÇÃO CIENTIFICA
INTEGRAL DE MONTE CARLO POR PROGRAMAÇÃO PARALELA

SÃO BERNARDO DO CAMPO
2019

VICTOR BIAZON
RA: 119.115-4

RELATÓRIO VII – PROGRAMAÇÃO CIENTIFICA
INTEGRAL DE MONTE CARLO POR PROGRAMAÇÃO PARALELA

Relatório de desenvolvimento programação paralela para executar o algoritmo de integral de Monte Carlo, desenvolvido pelo aluno Victor Biazon, RA 119.115-4, para disciplina PEL216 – Programação Cientifica, ministrada pelo professor Reinaldo Bianchi.

São Bernardo do Campo
2019

Sumário:

Motivação

Implementação de programação paralela para melhorar desempenho do algoritmo de integral pelo método de Monte Carlo.

Objetivo:

Desenvolver algoritmo para execução do algoritmo de integral pelo método de Monte Carlo utilizando de programação paralela para que seja possível diminuir o tempo necessário para execução do mesmo. Verificar o uso do MPI no Linux como forma de utilizar a programação paralela.

Teoria

Programação Paralela

“Through the 1990s desktop PCs became faster and faster. By shrinking the physical size of the chips, Intel could make its chips faster whilst simultaneously reducing the cost of manufacture. There wasn't much call for parallel systems on the desktop - why bother with all the problems of running two processors when in 18 months’ time you could buy a serial processor that ran twice as fast?

Unfortunately, the miniaturization process began to hit fundamental physical barriers at about 3GHz. When transistor sizes become very tiny, electrons start to behave in quantum mechanical ways, and circuits no longer work properly. So the only way to make machines faster and faster is to add more chips, Hence parallel programming, and the importance of MPI.” (MCLEAN, 2006)

Como mencionado por McLean (2006) os chips de processamento estavam ficando cada vez mais rápidos não existindo a necessidade de utilizar programação paralela, no entanto com a limitação física encontrada no início do século se criou a necessidade de utilizar mais processadores realizando a mesma tarefa ou tarefas dependentes para que fosse possível aumentar ainda mais a velocidade de processamento.

Também definido pro Mclean (2006) problemas do “mundo real” como simulações de fenômenos físicos e interações entre eles podem ser simulados com processamento paralelo tendo em vista os comandos MPI_Bcast() e MPI_Reduce() que servem justamente para compartilhar informações entre tarefas distintas. Menciona também que programas realmente paralelos devem ser assíncronos, sendo assim os objetos simulados precisam agir com envio e recebimento de mensagens.

Implementação

Para a implementação do MPI aplicado ao método de integração por Monte Carlo foram utilizadas as seguintes funções (as funções especificas do MPI estão em negrito no código):

O código abaixo foi desenvolvido em C++ e compilado no Linux Ubuntu 18.04 com a biblioteca MPIC++ com os comandos:

Compilar: Mpic++ <nome do aquivo.extensao> -o <nome do arquivo.executavel>
Executar: Mpiexec ./<nome do arquivo.executavel>

```
void MonteCarloIntegral(int numeros){ // calcula taxa de acerto no toroide pelo metodo de montecarlo
	float x = 0;
	float y = 0;
	float z = 0;
	float resultado = 0;
	float contador = 0;
	for (; contador <= numeros; contador++) { //calcula n numeros de amostras com entradas x y e z randomicas e verifica se esta no toroide
		x = random(xUm, xD);
		y = random(yUm, yD);
		z = random(zUm, zD);

		resultado = funcao(x, y, z);

		if (isToroide(resultado)) // se estiver no toroide soma 1 no total
			totaltask += 1;
			f2task += pow(1, 2);
	}
}


int main(int argc, char *argv[]) {
	int tasks, taskid;

	MPI_Init(&argc, &argv); // inicializa o MPI
	MPI_Comm_size(MPI_COMM_WORLD, &tasks); //determina numero de tasks
	MPI_Comm_rank(MPI_COMM_WORLD, &taskid); //assume numeracao de tasks

	int numeros = 100000; //define variaveis auxiliares
	if(argc > 1)
		numeros = atoi(argv[1]);
	int contador = 0;
	float total = 0;
	float f2 = 0;
	float f = 0;

	MonteCarloIntegral(numeros / tasks); //chama a funcao e divide a tarefa entre as tasks
	MPI_Reduce(&totaltask, &total, 1, MPI_FLOAT, MPI_SUM, MASTER, MPI_COMM_WORLD); //recebe informacao das variaveis totaltask para a variavel total
	MPI_Reduce(&f2task, &f2, 1, MPI_FLOAT, MPI_SUM, MASTER, MPI_COMM_WORLD); //recebe informacao das variaveis f2task para a variavel f2
```

```
	if(taskid == MASTER) { //verifica se a task é o MASTER (taskid = 0) para
executar apenas uma vez o o calculo final e print na tela
		f2 = f2 / numeros; // calcula media das amostras quadradas
		f = total / numeros; // calcula media de acertos dentro do toroide das
amostras
		float V = (xD - xUm) * (yD - yUm) * (zD - zUm); //calcula volume de
integracao
		float erro = V * sqrt((f2 - (float)pow(f, 2)) / numeros); // calcula erro
estimado
		//mostra na tela taxa de acerto dentro do toroide, resultado da
integracao e erro estimado
		cout << "Taxa de acerto no toroide: "<< f << " Resultado: " << V * (total
/ numeros) << endl;
		cout << "Erro: " << erro << endl;
		cout << "Numero de amostras: " << numeros << endl;
		cout << "Tasks: " << tasks << endl;
		}
	MPI_Finalize(); //finaliza execucao do MPI
```

Experimentos e resultados

Para testar os algoritmos foram propostas três comparações para integral do toróide abaixo:

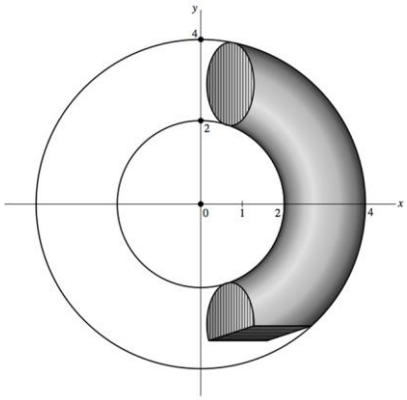

Fig. 1 Secção Toroidal.

Abaixo o exemplo de execução de uma task com 4 processares e com 1 processador:

Como pode-se notar o valor da integral e do erro foram muito próximos.

```
ubuntu@UBUNTU-ROS:~/Desktop$ mpiexec -n 4 ./Exec 100000
Taxa de acerto no toroide: 0.52494 Resultado: 22.0475
Erro: 0.113048
Numero de amostras: 100000
Tasks: 4
ubuntu@UBUNTU-ROS:~/Desktop$ mpiexec -n 1 ./Exec 100000
Taxa de acerto no toroide: 0.5282 Resultado: 22.1844
Erro: 0.112777
Numero de amostras: 100000
Tasks: 1
```

Abaixo segue uma tabela com a comparação entre o tempo de execução para diversas quantidades de amostras n para 1 e 4 processadores:

	1 Processador	4 Processadores
100000 amostras	5.64s	1.42s
1000000 amostras	54.32s	14.35s
10000000 amostras	555.70s	144.3s

Trabalhos correlatos

Distributed TensorFlow with MPI

Abhinav Vishnu Charles Siegel Jeffrey Daily

https://www.researchgate.net/publication/301842011_Distributed_TensorFlow_with_MPI

Efficient Large Message Broadcast using NCCL and CUDA-Aware MPI for Deep Learning ∗

A. A. Awan K. Hamidouche A. Venkatesh D. K. Panda

https://dl.acm.org/citation.cfm?id=2966912

Conclusão

Como pode-se notar o processamento paralelo diminui muito o tempo necessário para desempenhar uma tarefa quando a mesma é repetitiva e independente do resultado de outras iterações. Portanto para o método de Monte Carlo e outros que seu resultado é não dependente de resultados anteriores, o processamento paralelo é uma ferramenta importante para diminuir o tempo necessário para chegar ao resultado sem perder a qualidade do mesmo.

Referências bibliográficas

[1] McLean, Malcolm. **MPI Program models.** Disponível em: < https://web.archive.org/web/20080129015118/http://www.personal.leeds.ac.uk/~bgy1mm/MPITutorial/MPIModels.html > Acesso realizado: 03 Set 2019.