Universidade Federal de Minas Gerais

Programação Paralela

Relatório EP1

Professor: George Luiz Medeiros Teodoro
Turma: TH TN
Autores: Arthur Phillip F. Silva
Pedro Nascimento Costa

Introdução

O trabalho consiste na análise do uso de computação paralela na solução de prolemas computacionais. Para isso, foi utilizado o problema de achar todos os números primos através do crivo de *Eratóstene*, um algorítimo em que, dada uma lista de números não marcados de 2 a N, a partir do menor número, para cada elemento não marcado, marca-se todos os seus múltiplos na lista. No final, todos os números não marcados são primos.

Para este trabalho foram implementados dois algoritmos, o primeiro segue as instruções de *Eratóstene* de forma sequencial de acordo com o algorítimo original, no segundo foram feitas algumas alterações para que ele funcione de forma paralela. Ambos resolvem o problema dos números primos para qualquer valor entre 0 e N, sendo N um número especificado na entrada do programa podendo assumir valores entre $1 < N \leq 10^9$.

Especificações das máquinas

Máquina utilizada para os primeiros testes cujos resultados estão dispostos nas tabelas 1 e 2.

1. **Sistema Operacional**: Linux
2. **Processador**: Dois do tipo Intel(R) Xeon(R) CPU E5645 @ 2.40GHz
3. **Número de Cores**: 6 por processador, 12 no total.
4. **Número de Threads**:12 por processador, 24 no total.

5. **RAM**: 48GiB

Máquina utilizada para os testes de balanceamento cujos resultados estão dispostos nos gráficos 1a, 1b, 1c.

1. **Sistema Operacional**: Linux
2. **Processador**: vCPU Intel Haswell (instancia virtual no Google Cloud)
3. **Número de Cores**: 8 no total
4. **RAM**: 30GiB

Compilação do programa

Para a compilação do programa, basta rodar o comando **make** no diretório no terminal. Feito isso serão gerados três executáveis, **sequencial_primes** para o código sequencial e **parallel_primes** para o código paralelo, além desses dois principais, o terceiro é o código paralelo para impressão dos tempos das *threads*, **parallel_primes_print**[1].

Particionamento de Tarefas

Na solução paralela é fornecido por parâmetro a quantidade de *threads* a serem utilizadas, estas são chamadas pelo comando *OMP parallel for* que cria uma *pool* de *threads* e gerencia a alocação e execução destas. Neste trabalho foram testados dois tipos de escalonador *Static*, que aloca as *threads* de forma sequencial cíclica e o *Dynamic*, que usa uma política de *first-come first-served*. Para a segmentação do vetor entre as *threads* foram utilizados três opções de *ChunkSize* conforme mostra a tabela 1 e é melhor explicado na proxima seção. *ChunkSize* se refere ao numero de iterações alocado a cada *thread*.

No *omp parallel for*, o balanceamento da carga é feita de tal forma que iterações de um *loop* são distribuidas entre as *threads* seguindo o método do escalonador.

Escalonador e *ChunkSize*

Para análise dos escalonadores e *ChunkSize*, foram feitos os testes conforme a tabela 1.

Tempo do sequencial(s), $\mathbf{N} = 10^8$: 3,28

Tempo do sequencial(s), $\mathbf{N} = 10^9$: 36,98

[1] O **parallel_primes_print** é um Código modificado que imprime tempo total de execução por *thread*, por ter mais operações e gravação em memória ele é bem mais lento que os demais.

Análise do SpeedUp					
Escalonador	*ChunkSize*	Valor de **N**	Valor de **P**	Tempo de Execução(s)	*SpeedUp*
Static	1	100.000.000	2	2,94	1,1156
Static	1	1.000.000.000	2	34,79	1,0629
Dynamic	1	100.000.000	2	3,53	0,9292
Dynamic	1	1.000.000.000	2	36,01	1,0269
Static	1	100.000.000	4	2,04	1,6078
Static	1	1.000.000.000	4	23,61	1,5663
Dynamic	1	100.000.000	4	2,65	1,2377
Dynamic	1	1.000.000.000	4	27,53	1,3433
Static	1	100.000.000	8	1,25	2,6240
Static	1	1.000.000.000	8	13,95	2,6509
Dynamic	1	100.000.000	8	2,32	1,4138
Dynamic	1	1.000.000.000	8	47,31	0,7817
Static	256	100.000.000	2	2,42	1,3554
Static	256	1.000.000.000	2	27,96	1,3226
Dynamic	256	100.000.000	2	2,15	1,5256
Dynamic	256	1.000.000.000	2	23,99	1,5415
Static	256	100.000.000	4	2,14	1,5327
Static	256	1.000.000.000	4	21,81	1,6956
Dynamic	256	100.000.000	4	1,58	2,0759
Dynamic	256	1.000.000.000	4	16,48	2,2439
Static	256	100.000.000	8	1,70	1,9294
Static	256	1.000.000.000	8	17,98	2,0567
Dynamic	256	100.000.000	8	1,37	2,3942
Dynamic	256	1.000.000.000	8	14,72	2,5122
Static	512	100.000.000	2	2,85	1,1509
Static	512	1.000.000.000	2	29,52	1,2527
Dynamic	512	100.000.000	2	2,17	1,5115
Dynamic	512	1.000.000.000	2	23,93	1,5453
Static	512	100.000.000	4	2,19	1,4977
Static	512	1.000.000.000	4	22,78	1,6234
Dynamic	512	100.000.000	4	1,75	1,8743
Dynamic	512	1.000.000.000	4	16,93	2,1843
Static	512	100.000.000	8	1,69	1,9408
Static	512	1.000.000.000	8	18,74	1,9733
Dynamic	512	100.000.000	8	1,50	2,1867
Dynamic	512	1.000.000.000	8	16,48	2,2439

Tabela 1: Variação de escalonador e chunk size para casos extremos.

Análise dos resultados

Com base nos testes realizados e apresentados na tabela 1, observou-se que a melhor combinação de escalonador e *ChunkSize* foi de *Static* e 1 respectivamente. O **N** maior obteve um resultado melhor com relação ao **N** menor nesse caso e o número crescente de processadores melhorou o desempenho.

É interessante notar que para o escalonador *Dynamic* o melhor resultado foi com *ChunkSize* 256 e assim como o *Static*, nesse melhor caso o *SpeedUp* foi melhor em casos de mais *threads* e um **N** maior.

Análise da Implementação Feita

Utilizando como base os resultados da tabela 1 apresentadas na análise do Escalonador e *ChunkSize*, a tabela 2 analisa o escalonamento dos parâmetros de melhor desempenho(*SpeedUp*) ao se aumentar o tamanho de **N**.

Análise do Escalonamento	
Valor de **N**	Tempo de Execução (s)
10.000	0,0004905
100.000	0,0012765
1.000.000	0,0084855
10.000.000	0,0943515
100.000.000	1,286317
1.000.000.000	14,2322775

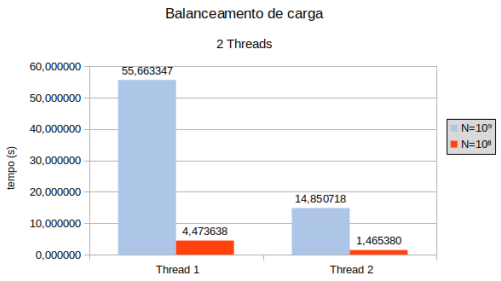
Tabela 2: Crescimento do tempo de execução.

Dado o observado em ambas tabelas 1 e 2, podemos concluir que a implementação é escalável, considerando que o crescimento do tempo de execução não acompanha linearmente o aumento de **N**, porém é importante perceber que apesar de escalar, escala de forma fraca porque o problema tem crescimento é exponencial.

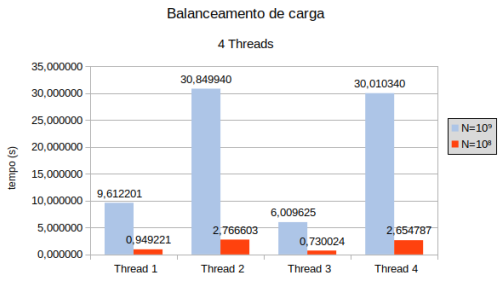
Balanceamento de Cargas

Os graficos 1a, 1b e 1c mostram o desempenho do balanceamento entre as *threads* de três situações, o programa executando com 2, 4 e 8 *threads* respectivamente, além disso se manteve o **N** constante em 10^8 e em 10^9.

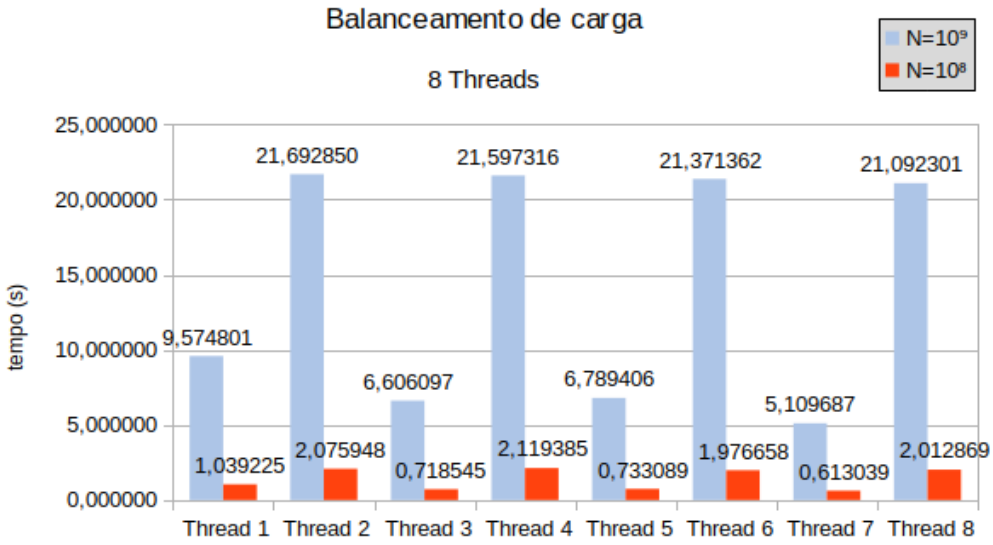
Observação: o código foi levemente alterado para contar o tempo de execução de cada *thread*, isso afetou o tempo execução geral do programa, notavelmente as modificações pesam mais o desempenho a medida que se usa mais *threads*.



(a) 2 threads



(b) 4 threads



(c) 8 threads

Figura 1: Tempo de execução por thread para **N** constante em 10^8 e 10^9

Observações sobre o balanceamento entre as *Threads*

Pelos graficos 1a, 1b e 1c, levando em consideração a máquina utilizada para execução dos testes, observou-se que algumas *threads* receberam maior parte do trabalho, levando em consideração o método de escalonamento estático, conclui-se que isso se deve ao modo como as *threads* mais lentas seriam justamente as que recebem iterações com os primeiros primos, que são os que mais geram operações nos *loop*.

Conclusão

O trabalho proporcionou uma boa experiência inicial com paralelismo. Os maiores desafios foram primeiramente entender o fluxo da parte paralela e a utilização do *OpenMp* como também a realização dos balanceamentos de carga e uso do escalonador da parte paralela.

Foi observado que não basta paralelizar para se obter um resultado(*SpeedUp*) favorável, deve-se analisar os parâmetros dessa paralelização realizada e entender o *Hardware* utilizado.