Relatório Projeto 1 - Programação Paralela

Vinicius A. T. Resende
RA: 21006116

1. Respostas às questões iniciais

- Entre 1 e 1.000.000.000 existem 50,847,534 números primos
- Os 20 últimos números primos menores que 1.000.000.000 são : 999999491, 999999503, 999999527, 999999541, 999999587, 999999599, 999999607, 999999613, 999999667, 999999677, 999999733, 999999739, 999999751 999999757, 999999761, 999999797, 999999883, 999999893, 999999929, 999999937

2. Introdução e explicação da implementação

Foi escolhido utilizar a versão do crivo de Eratóstenes segmentado, nessa versão dado a entrada *n* o programa calcula normalmente todos os primos até a raiz de n, após isso o resto do intervalo de 0 até n é subdividido em blocos(com tamanho de no mínimo raiz de n) e com os primos já encontrados checam-se os números nesse blocos por primos (para cada primo já encontrado, os números que são divididos por ele no bloco são marcados). Essa versão do crivo tem duas principais vantagens sobre o crivo normal. A primeira é que para um n muito grande preserva-se o princípio da localidade, pois os blocos são consideravelmente menores que n e se bem implementados cabem na cache - o que não ocorre na versão normal - em que dependendo do tamanho da entrada o vetor resultante não caberá na cache, além dele ser percorrido de ponta a ponta para cada primo. A segunda característica é a facilidade com que o crivo segmentado é paralelizado, a clara divisão em blocos já indica como dividir as tarefas entre diferentes processadores. Para efeitos de comparação, além da versão serial do crivo de Eratóstenes também foi feita uma versão normal do crivo (otimizada considerando só ímpares) , a versão serial possui performance parecida com a de Eratóstenes para entradas pequenas e médias, mas para entradas maiores ela rapidamente supera a normal. Como orientação e base para a implementação foi usado tanto a descrição na Wikipedia do crivo segmentado como a no site Geeks for Geeks (https://www.geeksforgeeks.org/segmented-sieve/) .

A versão paralelizada do crivo segmentado é de fácil particionamento de tarefas, o intervalo de 0-n já está subdividido em blocos de no mínimo raiz de n, basta subdividir estes blocos entre os p processadores a serem usados de forma que cada processador

receba a mesma quantidade de blocos a ser processada, com exceção do core com rank mais alto que terá que processar alguns números a mais caso *n/(limite*número_de_processadores)* seja um número quebrado. É importante notar que todos os processadores realizam o passo inicial de calcular os primos até raiz de n, essa tarefa poderia ser feita por somente um e depois o vetor resultante passado aos outros cores, isso não é feito pois os outros processadores ficariam sem trabalho nenhum durante esse passo inicial. Por último a soma do total de primos encontrados por cada core é unida através da função MPI_Reduce() no processador de rank 0. Dessa forma garante-se que todos os processadores recebem a mesma carga de trabalho ainda que haja uma redundância pequena nas tarefas realizadas.

Existem otimizações que não foram feitas, mas provavelmente teriam melhorado um pouco a performance do código, especificamente na fase inicial em que ocorre um crivo simples. Algumas otimizações seriam não verificar números pares e só considerar números da forma 6k -\+ 1, essas alterações não foram implementadas pois a tentativa de inseri-las no código geraram bugs inexplicáveis e que não foram possíveis de serem resolvidos. Por mais simples que deveria ser implementar essas melhorias, não foi possível identificar uma forma de fazê-las que não quebrasse o código.

Junto com os códigos fontes estão dois scripts em Python, um que compila todos os arquivos e outro que executa todos os testes de uma vez (das implementações de crivo segmentado, não o tradicional), quantas vezes for indicado de acordo com o parâmetro na linha de comando em que foi chamado com n variando de 10^6 a 10^{10} e p de 1 a 16. A implementação paralela é executada com a flag "--oversubscribe" para que rode em uma máquina que tiver menos cores e ainda execute os testes corretamente. Todos os programas recebem n como parâmetro de entrada na linha de comando retornando o número de primos encontrados e o tempo de execução em segundos. Para testar com valores acima de 10^{10} é preciso usar o comando da função paralela diretamente na linha de comando usando mpiexec, o programa é compilado com nome "parasegerast" e recebe *n* como argumento da linha de comando. É importante também notar que os programas são compilados com a flag -O3 do gcc, que força a maior otimização possível por parte do processador, o uso dessa flag aumentou consideravelmente a performance de todas as versões.

Para um teste em que cada variação é executada uma única vez (a execução é feita começando do menor n e com a versão sequencial, a versão paralela é testada com ordem crescente para o número de processadores) :

1. python compila_programas.py
2. Python executa_testes.py 1

Por motivos que serão discutidos depois no texto, talvez seja interessante/necessário compilar sem a flag -O3 ou --oversubscribe, para isso basta removê-las do script compila_programas.py.

3. Performance e análise de speedup

n/ comm_sz	10^6	10^7	10^8	10^9	10^{10}
1	1.0	1.0	1.0	1.0	1.0
2	1.95	1.94	1.95	1.95	1.94
4	2.77	2.83	2.80	2.66	2.82
8	2.66	2.82	2.76	2.64	2.77
16	2.69	2.54	2.70	2.56	2.75

Tabela 1: Speedup, tamanho da entrada (n) por número de processadores (comm_sz) sem a flag -O3 no gcc

Antes de analisar os valores na tabela, é preciso considerar que a máquina em qual os testes foram realizados tem um processador Intel i7-5500U que possui dois cores físicos, hyper-threading de 4 threads e 4Mb de cache, a máquina também possui 8GB de RAM.

Analisando primeiro a tabela em que foi compilado sem a flag -O3, os speedups mostram que o programa nessa máquina tem uma eficiência boa/razoável até 4 cores sendo utilizados, acima de 4 a eficiência decai drasticamente e não ocorrem ganhos significativos de performance mesmo aumentando o tamanho de n. A primeira vista isso indicaria que a implementação não é nem fortemente ou fracamente escalável, entretanto considerando que a máquina só possui 2 cores com 4 threads é compreensível não ocorrer melhoras muito significativas na eficiência com o aumento de comm_sz. Vale notar também que para 2 cores o programa atua no limite da eficiência, o que não permite analisar se existe escalabilidade fraca com o aumento de n em 2 cores, o que se altera com 4 cores cuja eficiência não está tão perto do limite e também não aumenta com n, isso indicaria que o programa não seria nem fracamente escalável. Considerando os dados da

tabela a implementação não seria escalável em nenhum nível, mas devido às limitações da máquina em que os testes foram executados esse não deve ser o veredito final sobre a escalabilidade da implementação, para tanto seria necessário testar em máquinas com mais do que 2 ou 4 cores.

n/ comm_sz	10^6	10^7	10^8	10^9	10^{10}
1	0.75	0.8	0.78	0.76	0.87
2	1.61	1.53	1.54	1.50	1.73
4	2.22	2.23	2.26	1.93	2.75
8	2.26	1.71	2.22	1.91	2.64
16	2.93	1.59	2.10	2.10	2.60

Tabela 2: Speedup, tamanho da entrada (n) por número de processadores (comm_sz) com a flag -O3 no gcc

A segunda tabela considera a performance usando a flag -O3 e apresenta um comportamento anômalo quanto a primeira. Para somente 1 core ocorre uma perda de performance considerável, não mais sendo 1.0 enquanto que os speedups com mais processadores também diminuíram consideravelmente em comparação com a primeira tabela. A explicação para esse comportamento é que a flag utilizada não otimiza tão bem programas paralelos quanto sequenciais, o que foi percebido ao comparar a melhora com e sem a flag, enquanto a versão sequencial melhorava em fatores de duas até quatro vezes dependendo do tamanho da entrada, a versão paralela ficava em torno de 1.7x e duas vezes. Não foi encontrado nenhuma outra explicação tanto nas documentações pesquisadas ou em outras fontes, mas ainda com esse comportamento desconhecido o aumento de performance foi grande o suficiente para justificar o uso dessa flag . Os dados da segunda tabela também não fornecem mais informações relevantes para mudar as conclusões da análise da primeira tabela.