UNIVERSIDADE DE BRASÍLIA
DISCIPLINA: PROGRAMAÇÃO PARALELA - TURMA A
PROFESSOR: GEORGE LUIZ MEDEIROS TEODORO

Cálculo Paralelo de Números Primos

Luís Felipe Rabello Taveira
15/0074662

April 2, 2015

1 INTRODUÇÃO

Implementou-se um algoritmo paralelo utilizando os pragmas de OpenMP para gerar uma lista ordenada de números primos de 2 até um inteiro n passado na entrada, onde $2 < n <$ 10^9. Foram avaliados e comparados o impacto no desempenho dos escalonadores e dos chunkSize utilizados.

O programa principal encontra-se no arquivo main.c, e os benchmarks podem ser repetidos através da chamada do programa benchmark.c. Para compilá-los deve-se utilizar o gcc com a flag -fopenmp -std=c99 -lm.

2 PROBLEMA

Calcular todos os números primos compreendidos entre 2 e n (entrada do usuário) de forma paralela, utilizando a biblioteca OpenMP. Além disso, avaliar o impacto do tipo de escalonador e chunkSize no desempenho.

3 SOLUÇÃO PROPOSTA

Para uma entrada de n posições foi criado um vetor de tamanho n/2, pois não é necessário testar os números pares. O Algoritmo 1 foi utilizado para testar se um número é primo ou não. Ao checar se um número X é primo, não é necessário testar se ele é divisivel por cada um dos números naturais menores que ele. É necessário testar apenas até a raiz quadrada de X.

Algoritmo 1: Código que verifica se um número é primo ou não. A complexidade desse algoritmo é sqrt(n).

```
int IsPrime(unsigned int number) {
        // zero e um nao sao primos
        if (number <= 1) return 0;
        //Se o numero for par nao eh necessario checar se eh primo
        if(number != 2 && number % 2 == 0) return 0;
        unsigned int i;
        for (i=3; i*i<=number; i+=2) {
                if (number % i == 0) return 0;
        }
        return 1;
}
```

4 ESTRATÉGIA DE PARALELIZAÇÃO

A estratégia utilizada para paralelizar a solução do problema foi subdividir os índices do laço *for* entre as threads, de modo que cada uma delas ficasse encarregada de testar a primalidade de um subconjunto da entrada.

Foram testados os três tipos de escalonadores de tarefas do OpenMP, o escalonador que obteve o melhor desempenho foi o Dynamic ou o Guided dependendo do tamanho do chunkSize e do número de threads utilizado. Mais detalhes sobre os testes podem ser conferidos na Seção 5. O Algoritmo 2 ilustra como foi realizada a paralelização do problema.

Algoritmo 2: Foram utilizados os pragmas do OpenMP para subdividir as tarefas entre as threads. A configuração do escalonador utilizado foi o *runtime* para que ele pudesse ser alterado em tempo de execução a fim de realizar os testes de desempenho com mais de um escalonador. Foi por esse motivo também que o número de threads a ser utilizado também não foi fixado. A complexidade desse algoritmo é (n * sqrt(n)).

```
//Check which numbers greater than 2 are prime (no need to check even numbers)
#pragma omp parallel num_threads (nthreads)
    {
#pragma omp for schedule (runtime)
        for(int i=3 ; i<=n; i+=2){
                if(IsPrime(i)) out[(int)floor(i/2)] = i;
                else out[(int)floor(i/2)]=0;
        }
    }
```

5 TESTES E ANÁLISE

Todos os testes descritos nesta seção foram realizados três vezes. A máquina utilizada nos testes foi um *Intel Core i7 2,3ghz com 16GB RAM DDR3* rodando a versão 14.0.1 do Ubuntu Linux.

Para testar a eficiência e o *speed-up* da solução proposta, foram realizadas diversas baterias de testes variando o tipo de escalonador (*static, dynamic, guided*), a quantidade de threads utilizada (*1, 2, 4 e 8*) e o tamanho do chunkSize (*64, 128, 256, 512, 2500, 5000, 10000*). O resultado de todos esses testes se encontra no arquivo ZIP enviado na plataforma Moodle da disciplina. Foram selecionados apenas um sub-conjunto desses testes para serem apresentados neste documento. As Figuras 5.1, 5.2, 5.3 ilustram esses testes. Vale notar que os testes com os chunkSizes variando de 64 a 512 foram realizados em momentos distintos dos testes que variavam o chunkSize de 2500 a 10000.

O ganho de performance ao aumentar o número de threads foi muito grande ao passar de 1 para 2 ou 4 threads. A partir de 8 threads a eficiência do algorimo decresceu, porém o ganho de *speed-up* foi notável até oito threads. Um dos motivos para isso é a existência de apenas 4 núcleos físicos no processador utilizado com suporte a *HyperThreading*. No geral, os escalonadores Guided e Dynamic obtiveram os melhores resultados. Dos três chunksizes apresentados neste documento o valor que apresentou a melhor performance foi o de 64. Nos testes utilizando este valor de chunksize, os escalonadores Guided e Dynamic apresentaram desempenhos muito similares, com uma leve vantagem para o Guided quando o tamanho da entrada era muito grande. Esperava-se que a variação do tipo de escalonador causasse uma variação muito maior de performance do que o observado, uma vez que o teste de primalidade para números grandes demanda muito mais processamento do que o teste para números pequenos, pois é necessário testar a divisibilidade por todos os números até a raiz quadrada do número desejado. O principal fator que justifica essa pouca diferença entre os escalonadores foi o pequeno tamanho dos chunkSizes utilizados em relação ao tamanho da entrada, eles impediram que algumas threads ficassem ociosas por muito tempo antes que as outras terminassem a computação.

5.1 ESCALABILIDADE

O algoritmo desenvolvido é *fortemente escalável* pois ao aumentarmos o número de threads, a eficiência permanece a mesma quando o tamanho do problema permanece o mesmo. Isso pode ser observado nos experimentos até o momento em que o número de threads utilizadas fosse igual ao número de núcleos do processador utilizado, que no caso do experimento foi 4. A partir de 4 threads a efiência do algoritmo decresceu devido a troca de contexto entre as threads do processador.

Em alguns casos ao aumentar o número de threads, a eficiência observada foi um pouco maior que 1, isso provavelmente foi causado por otimizações realizadas pelo compilador e pelo processador em tempo de execução. Vale notar que a versão serial utilizada para comparações foi o mesmo código da versão paralela porém utilizando o OpenMP com apenas uma thread. Esse fator também pode ter contribuído para a eficiência do programa ser maior do que 1 em alguns casos, pois dependendo da forma que a versão do OpenMP utilizada foi implementada, o overhead de criação de threads e a distribuição de trabalho ficou melhor diluído ao utilizar mais threads.

Input Size	Chunk Size	Scheduler	Threads	Time (s)	SpeedUp	Eficiency
1000000	64	Static	1	0.167473	1	1
1000000	64	Static	2	0.077651	2.15673977	1.07836989
1000000	64	Static	4	0.039542	4.23531941	1.05882985
1000000	64	Static	8	0.032111	5.21544019	0.65193002
1000000	64	Dynamic	1	0.149804	1	1
1000000	64	Dynamic	2	0.07303	2.0512666	1.0256333
1000000	64	Dynamic	4	0.038221	3.91941603	0.97985401
1000000	64	Dynamic	8	0.028709	5.21801526	0.65225191
1000000	64	Guided	1	0.156556	1	1
1000000	64	Guided	2	0.078654	1.99043914	0.99521957
1000000	64	Guided	4	0.03995	3.9187985	0.97969962
1000000	64	Guided	8	0.029933	5.23021414	0.65377677
10000000	64	Static	1	3.919488	1	1
10000000	64	Static	2	1.86047	2.10671927	1.05335963
10000000	64	Static	4	0.978823	4.00428678	1.0010717
10000000	64	Static	8	0.804199	4.87377875	0.60922234
10000000	64	Dynamic	1	3.890648	1	1
10000000	64	Dynamic	2	1.812121	2.14701336	1.07350668
10000000	64	Dynamic	4	0.967182	4.02266378	1.00566594
10000000	64	Dynamic	8	0.773959	5.02694329	0.62836791
10000000	64	Guided	1	3.85769	1	1
10000000	64	Guided	2	1.821983	2.11730296	1.05865148
10000000	64	Guided	4	0.978945	3.94066061	0.98516515
10000000	64	Guided	8	0.782531	4.92975997	0.61622
100000000	64	Static	1	98.066308	1	1
100000000	64	Static	2	47.125871	2.0809442	1.0404721
100000000	64	Static	4	25.697981	3.81610944	0.95402736
100000000	64	Static	8	20.496243	4.78459921	0.5980749
100000000	64	Dynamic	1	97.209591	1	1
100000000	64	Dynamic	2	46.855301	2.07467648	1.03733824
100000000	64	Dynamic	4	25.42582	3.82326277	0.95581569
100000000	64	Dynamic	8	20.218675	4.80791105	0.60098888
100000000	64	Guided	1	97.756966	1	1
100000000	64	Guided	2	46.819415	2.08795787	1.04397893
100000000	64	Guided	4	25.498912	3.83377008	0.95844252
100000000	64	Guided	8	20.141827	4.85343092	0.60667887

Figure 5.1: Testes realizados com o chunckSize igual a 64. Foram alterados o número de threads e o os escalonadores. Os resultados em verde são os que apresentaram a melhor performance da categoria em que estão incluídos.

Input Size	Chunk Size	Scheduler	Threads	Time (s)	SpeedUp	Eficiency
1000000	2500	Static	1	0.148127	1	1
1000000	2500	Static	2	0.07017	2.11097335	1.05548668
1000000	2500	Static	4	0.039079	3.79045011	0.94761253
1000000	2500	Static	8	0.035109	4.21906064	0.52738258
1000000	2500	Dynamic	1	0.139721	1	1
1000000	2500	Dynamic	2	0.07159	1.9516832	0.9758416
1000000	2500	Dynamic	4	0.03659	3.81855698	0.95463925
1000000	2500	Dynamic	8	0.031978	4.36928513	0.54616064
1000000	2500	Guided	1	0.141823	1	1
1000000	2500	Guided	2	0.069444	2.04226427	1.02113214
1000000	2500	Guided	4	0.037533	3.77862148	0.94465537
1000000	2500	Guided	8	0.031806	4.45900145	0.55737518
10000000	2500	Static	1	3.443363	1	1
10000000	2500	Static	2	1.749661	1.96801723	0.98400862
10000000	2500	Static	4	0.891758	3.86132	0.96533
10000000	2500	Static	8	0.779449	4.41768865	0.55221108
10000000	2500	Dynamic	1	3.464699	1	1
10000000	2500	Dynamic	2	1.719595	2.01483431	1.00741715
10000000	2500	Dynamic	4	0.896136	3.86626472	0.96656618
10000000	2500	Dynamic	8	0.80771	4.28953337	0.53619167
10000000	2500	Guided	1	3.449127	1	1
10000000	2500	Guided	2	1.721062	2.004069	1.0020345
10000000	2500	Guided	4	0.90352	3.81743293	0.95435823
10000000	2500	Guided	8	0.812797	4.24352821	0.53044103
100000000	2500	Static	1	92.313427	1	1
100000000	2500	Static	2	45.695423	2.02018979	1.0100949
100000000	2500	Static	4	26.543029	3.47787839	0.8694696
100000000	2500	Static	8	22.903326	4.03056862	0.50382108
100000000	2500	Dynamic	1	92.621388	1	1
100000000	2500	Dynamic	2	45.520333	2.03472562	1.01736281
100000000	2500	Dynamic	4	26.049252	3.55562563	0.88890641
100000000	2500	Dynamic	8	23.6357	3.91870721	0.4898384
100000000	2500	Guided	1	90.268129	1	1
100000000	2500	Guided	2	45.678998	1.97614074	0.98807037
100000000	2500	Guided	4	25.843267	3.49290703	0.87322676
100000000	2500	Guided	8	23.642407	3.81806002	0.4772575

Figure 5.2: Testes realizados com o chunckSize igual a 2500. Foram alterados o número de threads e o os escalonadores. Os resultados em verde são os que apresentaram a melhor performance da categoria em que estão incluídos.

Input Size	Chunk Size	Scheduler	Threads	Time (s)	SpeedUp	Eficiency
1000000	10000	Static	1	0.140056	1	1
1000000	10000	Static	2	0.069839	2.00541245	1.00270622
1000000	10000	Static	4	0.036855	3.80018993	0.95004748
1000000	10000	Static	8	0.032178	4.352539	0.54406738
1000000	10000	Dynamic	1	0.138118	1	1
1000000	10000	Dynamic	2	0.069452	1.98868283	0.99434142
1000000	10000	Dynamic	4	0.037588	3.67452378	0.91863095
1000000	10000	Dynamic	8	0.031631	4.36653915	0.54581739
1000000	10000	Guided	1	0.137414	1	1
1000000	10000	Guided	2	0.073633	1.8662013	0.93310065
1000000	10000	Guided	4	0.041784	3.28867509	0.82216877
1000000	10000	Guided	8	0.030786	4.46352238	0.5579403
10000000	10000	Static	1	3.598504	1	1
10000000	10000	Static	2	1.795531	2.00414473	1.00207237
10000000	10000	Static	4	0.948297	3.79470145	0.94867536
10000000	10000	Static	8	0.817164	4.4036497	0.55045621
10000000	10000	Dynamic	1	3.605837	1	1
10000000	10000	Dynamic	2	1.789114	2.01543166	1.00771583
10000000	10000	Dynamic	4	0.921769	3.91186621	0.97796655
10000000	10000	Dynamic	8	0.810189	4.45061214	0.55632652
10000000	10000	Guided	1	3.57039	1	1
10000000	10000	Guided	2	1.784177	2.00114114	1.00057057
10000000	10000	Guided	4	0.984342	3.62718445	0.90679611
10000000	10000	Guided	8	0.80586	4.43053384	0.55381673
100000000	10000	Static	1	90.71778	1	1
100000000	10000	Static	2	45.062772	2.01314247	1.00657123
100000000	10000	Static	4	25.207998	3.59876972	0.89969243
100000000	10000	Static	8	22.863404	3.96781599	0.495977
100000000	10000	Dynamic	1	90.391314	1	1
100000000	10000	Dynamic	2	45.945899	1.96734237	0.98367119
100000000	10000	Dynamic	4	25.482423	3.54720248	0.88680062
100000000	10000	Dynamic	8	22.891311	3.94871722	0.49358965
100000000	10000	Guided	1	92.393194	1	1
100000000	10000	Guided	2	45.93017	2.01160139	1.0058007
100000000	10000	Guided	4	25.176205	3.66986184	0.91746546
100000000	10000	Guided	8	22.809662	4.05061653	0.50632707

Figure 5.3: Testes realizados com o chunckSize igual a 10000. Foram alterados o número de threads e o os escalonadores. Os resultados em verde são os que apresentaram a melhor performance da categoria em que estão incluídos.

6 CONCLUSÃO

O ganho de performance obtido na solução paralela em relação a solução serial foi de mais de cinco vezes em alguns casos utilizando oito threads. Além disso, a eficiência foi maior do que 1 em vários casos quando foi utilizado duas ou quatro threads, o que comprova que a estratégia de paralelização utilizada foi adequada.

Outro fator relevante para a melhora do desempenho da aplicação é a escolha de um chunk-Size não muito grande, pois ele pode causar desbalanceamento de carga dependendo do escalonador utilizado. Isso pode ser observado claramente nas três figuras apresentadas.

O experimento realizado neste exercício ajudou a exemplificar o ganho de performance que pequenas alterações no código para transformar uma solução serial em uma solução paralela pode trazer para a aplicação.