Crivo de Eratóstenes Paralelo usando MPI

Relatório da Programação Paralela da Universidade Federal do ABC

Hiago Lucas Cardeal de Melo Silva - 11077315

April 21, 2019

1 Metodologia

O método utilizado aqui tem como ideia base aproveitar o máximo do cache dos processadores. Portanto, o objetivo central é evitar *cache-miss*. Para isso, utilizou-se uma implementação pouco intuitiva, onde o *array* de n números é dividido para os p processos e, cada *subarray* restante é novamente dividido dentro de cada processo de forma que cada partiçao ocupe o máximo de *cache* possível.

1. A primeira etapa de cada processo é calcular a porção de números que deverá ser processado por ele. Uma das otimizaçoes feitas aqui é considerar apenas números ímpares.

2. Após a divisao de todos os números que serão calculados pelo processo atual, este novo *array* de números é novamente dividido igualmente em outros *cache_length arrays*. Onde *cache_length* é o número de linhas de cache somando-se os caches L1, L2 e L3. O número *cache_length* é obtido através da função *get_best_cache_length*. Suponha como sendo C o conjunto dos *cache_length arrays* restantes.

3. Calcula-se então, de forma sequencial, os números primos entre 0 e $\sqrt(n)$. Suponha que P representa o conjunto de números primos entre 0 e $\sqrt(n)$.

4. Este é o principal passo do algoritmo. Se trata da organização de dois *loops*. O primeiro *loop* itera para cada *subarray* em C, armazenando este *subarray* no cache. O segundo loop itera por todos os números primos. Estes também sao armazenados no cache. Com esta estrutura,

o número de $cache - miss$ é diminuido drásticamente. O pseudocódigo desta estapa pode ser encontrada na figura 1.

5. Finalmente, realiza-se um *Reduce* para somar a quantidade de primos em cada processo.

   Note que inicialmente, cada processo realiza exatamente os mesmos procedimento dos demais. Apenas na ultima etapa é que ocorre uma comunicação para somar os primos encontrados. Sendo assim, a **distribuição de carga** entre os processadores é praticamente a mesma.

```
1  for subarray in C {
2      for i in P {
3          mark_all_multiples(c, i)
4      }
5  }
```

Figure 1: Pseudo código da etapa 4.

2 *Speedup* e Eficiencia

A máquina utilizada para os testes possui as seguintes especificações:

- 8 GB de RAM
- Processador Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
- 4 *cores*
- 2 *threads* por *core*
- 32K de cache L1d / L1i
- 256K de cache L2
- 6144K de cache L3

Como base para o algorítmo sequencial foi utilizado o crivo implementado em Stack Exchange (link) como resposta à um *code review*. Esta implementaçao foi uma das mais eficientes encontradas, e, portanto, a análise dos *speedups* terá melhor precisão.

A algorítmo sequencial utilizado nao possui boa escalabilidade de memória, portanto, nao foi possível usa-lo para valores maiores que $2 * 10^{10}$

Tempo de Execução em segundos									
	1,00E+03	1,00E+04	1,00E+05	1,00E+06	1,00E+07	1,00E+08	1,00E+09	2*1e9	1,00E+10
1	0,11553883	0,10609054	0,10237345	0,10946239	0,14464989	0,413687	3,95444003	7,914	42,123
2	0,11672177	0,1097214	0,10695619	0,1029941	0,1281874	0,31834923	2,12251761	4,225	21,7604026
4	0,10323522	0,11220818	0,10517937	0,11212196	0,12424955	0,22848752	1,36060551	2,202	13,5999666
8	0,154	0,154	0,156	0,15	0,158	0,249	0,981	1,865	11,539
16	0,22	0,231	0,214	0,206	0,224	0,343	1,244	2,214	14,111
sequencia	0,001	0,001	0,006	0,008	0,08	0,63	8,71	18,326	

Speed Up									
	1,00E+03	1,00E+04	1,00E+05	1,00E+06	1,00E+07	1,00E+08	1,00E+09	2*1e9	1,00E+10
1	0,0086551	0,00942591	0,05860895	0,07308447	0,55305952	1,52289049	2,20258745	2,31564316	
2	0,00856738	0,00911399	0,05609773	0,07767436	0,62408631	1,97895878	4,10361731	4,33751479	
4	0,00968662	0,00891201	0,05704541	0,07135087	0,64386549	2,75726224	6,40156162	8,32243415	
8	0,00649351	0,00649351	0,03846154	0,05333333	0,50632911	2,53012048	8,87869521	9,82627346	
16	0,00454545	0,004329	0,02803738	0,03883495	0,35714286	1,83673469	7,00160772	8,27732611	

Eficiencia									
	1,00E+03	1,00E+04	1,00E+05	1,00E+06	1,00E+07	1,00E+08	1,00E+09	2*1e9	1,00E+10
1	0,0086551	0,00942591	0,05860895	0,07308447	0,55305952	1,52289049	2,20258745	2,31564316	
2	0,00428369	0,004557	0,02804887	0,03883718	0,31204316	0,98947939	2,05180865	2,1687574	
4	0,00242165	0,002228	0,01426135	0,01783772	0,16096637	0,68931556	1,60039041	2,08060854	
8	0,00081169	0,00081169	0,00480769	0,00666667	0,06329114	0,31626506	1,1098369	1,22828418	
16	0,00028409	0,00027056	0,00175234	0,00242718	0,02232143	0,11479592	0,43760048	0,51733288	

Figure 2: Tabela de resultados para tempo de execução, speedup e eficiencia.

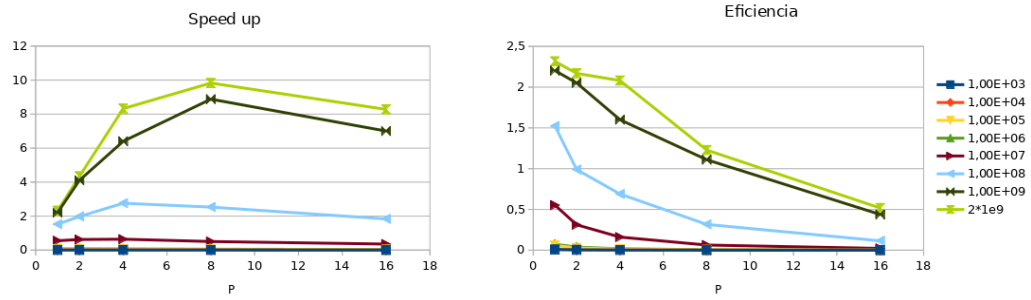

Figure 3: Speedup e eficiência.

2.1 Análise do *speedup*

A implementaçao paralela feita aqui calcula parte dos primos (de 0 até $\sqrt{n}$) de forma sequencial. Apenas a outra porção ($\sqrt{n}$ até n) é calculada paralelamente. Se n for pequeno uma grande porção dos números será processdo sequencialmente, o que diminui a eficiência do algoritmo.

Além disso, com uma entrada pequena de dados, os *caches* dos processadores podem não ser totalmente aproveitados.

É possível notar na figura 3 que até 10^7 a implementação apresenta *speedups* menores que 1 e, portanto, a utilizaçao deste método nao é justificável. No entanto para valores mais altos de o *speedup* aumenta drásticamente.

Uma outra análise que pode ser feita sobre os *speedups* é referente ao seu comportamento à partir de 4 processadores.

A máquina onde os testes foram feitos possui 4 *cores*, contendo cada um 2 *threads*. Embora existam apenas 4 *cores* disponíveis, o pico de *speedup* se

encontra no 8. Isto ocorre pois o sistema operacional é capaz de alternar entre as 2 *threads* durante os períodos de *stall*.

A partir de 16 processos o *speedup* começa a cair, pois o número de *cores* é ultrapassado e apenas o custo de *overhead* aumenta.

2.2 Eficiencia e escalabilidade

Como é possível observar na figura 3, o aumento do número de processos faz com que a eficiencia do algoritmo caia. Isto ocorre pois os *caches* dos processadores são menos aproveitados uma vez que os dados estao mais dividídos. Portanto, para aumentar a eficiência deve-se aumentar a entrada.

Este comportamento caracteriza um algoritmo com **escalabilidade fraca**.

3 Resultados

3.1 Quantidade de primos entre 0 e 1.000.000.000

50847534

3.2 Lista dos 20 ultimos primos entre 0 e 1.000.000.000

Em ordem decrescente.

1. 999999937	6. 999999761	11. 999999677	16. 999999587
2. 999999929	7. 999999757	12. 999999667	17. 999999541
3. 999999893	8. 999999751	13. 999999613	18. 999999527
4. 999999883	9. 999999739	14. 999999607	19. 999999503
5. 999999797	10. 999999733	15. 999999599	20. 999999491