Paralelização do Algoritmo "The Sieve of Eratosthenes"

Relatório

4º ano do Mestrado Integrado em Engenharia Informática e Computação

Computação Paralela

Elementos do grupo:
José Pedro Pereira Amorim - 201206111 - ei12190@fe.up.pt
Miguel Oliveira Sandim - 201201770 - miguel.sandim@fe.up.pt

22 de maio de 2016

1 Introdução

No âmbito da Unidade Curricular de Computação Paralela foi proposto um estudo da paralelização do algoritmo "The Sieve of Eratosthenes", fazendo uso das bibliotecas "OpenMP" (com memória partilhada) e "OpenMPI" (com memória distribuída).

No decorrer deste relatório será descrito o algoritmo proposto, bem como várias abordagens utilizando paralelismo. Serão também analisadas medições experimentais do desempenho das diferentes abordagens segundo algumas métricas.

2 Descrição do Problema

O problema em análise neste relatório é do cálculo de números primos num intervalo $[2, n] : n \in \mathbb{N} \setminus \{1\}$. Um número j é considerado primo se for inteiro, superior a 1 e se possuir dois únicos divisores naturais distintos: os números 1 e j.

3 Algoritmo "The Sieve of Eratosthenes"

Um dos algoritmos existentes para solucionar o problema descrito é o "The Sieve of Eratosthenes" [1]. Este algoritmo opera sobre uma lista que contém todos os números inteiros no intervalo $[2, n]$ e consiste na marcação de todos os múltiplos dos números primos que sejam inferiores ou iguais a $\sqrt{n}$. No final da sua execução, todos os números não marcados da lista correspondem aos números primos existentes no intervalo especificado (ver figura 1 e pseudocódigo 1).

O algoritmo apresenta complexidade temporal $\mathcal{O}(n\ log\ log\ n)$ e complexidade espacial $\mathcal{O}(n)$ [2].

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Primes:
2, 3, 5, 7,
11, 13, 17,
19, 23, 29,
31, 37, 41,
43, 47, 53,
59, 61, 67,
71, 73, 79,
83, 89, 97

Figura 1: Ilustração representativa do final da execução do algoritmo – a amarelo encontram-se os números primos no intervalo [2, 100] e a cor-de-rosa os múltiplos excluídos destes.

Pseudocódigo 1 Algoritmo "The Sieve of Eratosthenes".

```
1: // 1. Inicialização do vetor de inteiros no intervalo [2, n]
2: for i := 2 to n do
3:     numbers[i − 1] ← true
4: k ← 2
5: while k² ≤ n do
6:     // 2.1. Marcação dos múltiplos de k (seed) no intervalo [k², n]
7:     i ← k²
8:     while i ≤ n do
9:         numbers[i − 1] ← false
10:        i ← i + k
11:    // 2.2. k toma o valor do próximo número não marcado e superior a k
12:    k ← k + 1
13:    while numbers[k − 1] = false do
14:        k ← k + 1
```

4 Implementação do Algoritmo

Nesta secção encontra-se descrita as várias implementações do algoritmo, a serem estudadas na secção seguinte. Todas as implementações foram desenvolvidas com recurso à linguagem C++11.

4.1 Algoritmo Sequencial

A versão sequencial do algoritmo é uma implementação direta do pseudocódigo 1.

4.2 Paralelização do Algoritmo

De modo a estudar a performance e escalabilidade do algoritmo, foram implementadas diversas versões em diferentes modelos de memória de programação paralela, nomeadamente nos modelos de memória partilhada, de memória distribuída e híbrido (memória distribuída e partilhada). No caso do modelo de memória partilhada vários processos ou *threads* executam de forma paralela mas partilhando os mesmos recursos de memória, enquanto que no modelo de memória distribuída os processos lançados possuem a sua própria memória local que não é partilhada com outros [3]. Neste último caso, cabe ao programador decidir como a memória é repartida pelas diversas unidades de processamento, como os dados são comunicados (se um processo necessita de aceder a dados em memória de outro processo) e como estes são sincronizados [3]. Também é necessário uma rede que interligue as várias unidades de processamento, pelo que existirá latência na comunicação entre processos.

Atualmente o sistema mais usado pelos computadores com maior capacidade de processamento é o modelo de memória distribuída e partilhada, que combina características destes dois últimos modelos (em que em cada unidade de processamento pode existir partilha de memória pelos processos que nela executam) [3].

J. Quinn [1] aponta o passo 2.1. do pseudocódigo 1 como sendo a principal potencial fonte de paralelismo no algoritmo, pelo que se optou por explorar este

passo nas versões implementadas.

4.2.1 Paralelismo com Memória Partilhada

A versão do algoritmo usando o modelo de memória partilhada foi implementada usando a biblioteca "OpenMP", nomeadamente através da inclusão do "pragma" *opm parallel for* que divide sequencialmente as iterações do ciclo for de marcação dos múltiplos de k por t *threads* (ver código-fonte 1).

```
// Mark all multiples of k between k*k and n
#pragma omp parallel for num_threads(t)
for (long long i = k*k; i <= n; i += k)
    numbers[i-2] = false;
```

Código-fonte 1: Excerto de código que representa a paralelização da marcação dos múltiplos de k ente k^2 e n, utilizando a biblioteca "OpenMP".

4.2.2 Paralelismo com Memória Distribuída

A versão do algoritmo usando o modelo de memória distribuída foi implementada usando a biblioteca "Open MPI".

Nesta versão, o vetor "numbers" é dividido pelos p processos, sendo que cada processo i fica atribuído a um bloco de dados cujo o primeiro elemento é o elemento de índice:

$$first(i, n, p) = \left\lfloor \frac{i \cdot n}{p} \right\rfloor \tag{1}$$

e o último é o elemento de índice:

$$last(i, n, p) = \left\lfloor \frac{(i+1) \cdot n}{p} \right\rfloor - 1 \tag{2}$$

Dado que só é necessário marcar os múltiplos de k entre k^2 e n, cada processo i necessita de avaliar se precisa de efetuar marcações no bloco atribuído e, caso esta condição se verifique, de calcular o primeiro múltiplo no bloco atribuído. Neste sentido, podem ocorrer os casos contemplados na figura 2, que foram implementados de acordo com o pseudocódigo 2.

O processo $i = 0$ será sempre o único que, além de marcar os múltiplos de k no seu bloco de dados, é responsável verificar qual o próximo valor de k (vendo qual o menor número não marcado no seu bloco) e fazer *broadcast* para os restantes processos. De notar que o último valor k será sempre $\sqrt{n}$ [1], logo o primeiro processo conterá todos os possíveis valores de k se $\frac{n}{p} > \sqrt{n}$, o que é equivalente a $p < \sqrt{n}$. Dado que o maior valor que p tomará é 16 e o menor valor de n na gama de testes é 2^{25}, esta condição verificar-se-á sempre.

4.2.3 Paralelismo com Memória Partilhada e Distribuída)

A versão do algoritmo usando o modelo de memória distribuída foi implementada usando as bibliotecas "OpenMP" e "Open MPI". Esta implementação difere da implementação com apenas memória distribuída na adição do "pragma" de modo semelhante ao indicado no código-fonte 1.

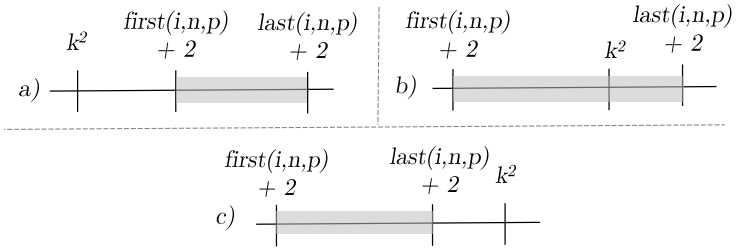

Figura 2: Esquema ilustrativo das situações que podem ocorrer na procura do primeiro múltiplo de k no bloco do processo, sendo $first(i, n, p) + 2$ o primeiro número do bloco alocado e $last(i, n, p) + 2$ o último e o retângulo cinzento o intervalo de números presentes no bloco.

Pseudocódigo 2 Algoritmo para procura do múltiplo de k no intervalo de números atribuido ao bloco ([$firstNumber$, $lastNumber$]).

```
1:  // Primeiro e último números do bloco
2:  firstNumber ← first(i, n, p) + 2
3:  lastNumber ← last(i, n, p) + 2
4:
5:  if k² < firstNumber then                                   ▷ Caso a)
6:      if (firstNumber + 2) mod k = 0 then
7:          firstMult ← firstNumber
8:      else
9:          firstMult ← firstNumber + (k − (firstNumber mod k))
10: else if k² ≥ firstNumber and k² ≤ lastNumber then          ▷ Caso b)
11:     firstMult ← k²
12: else                                                       ▷ Caso c)
13:     Calcular/esperar próximo valor de k
```

5 Experiências e Análise dos Resultados

5.1 Descrição das Experiências e Metodologia de Avaliação

Para avaliar as implementações do algoritmo descritas na secção anterior foram realizadas várias execuções experimentais, tendo sido medido o tempo de execução. Em todas as versões implementadas foram feitas execuções com $n = 2^i$, variando i de forma unitária entre 25 e 32. Adicionalmente, no caso da versão com memória partilhada foram feitas experiências usando 2 a 8 *threads* e na versão com memória distribuída foram lançados entre 2 a 16 processos (utilizando 4 computadores do laboratório ligados em rede, cada computador limitado no máximo a 4 processos – dado que cada computador possui um processador *quad-core*). De notar que o "Open MPI" lança os processos de forma sequencial nos computadores disponíveis (i.e. só lançará processos no computador Y depois de lançar o número máximo de processos no computador X, e assim sucessivamente).

No caso da versão com memória partilhada e distribuída foram feitas execu-

ções com as seguintes combinações de configurações (usando 4 computadores):

- 4 processos (1 em cada computador), cada um lançando de 2 a 8 *threads*;
- 8 processos (2 em cada computador), cada um lançando de 2 a 4 *threads*;
- 12 processos (3 em cada computador), cada um lançando 2 *threads*;
- 16 processos (4 em cada computador), cada um lançando 2 *threads*.

Estas combinações procuraram tentar maximizar o número de *threads* a executar em cada computador, dado que cada CPU permite a execução de até 8 *threads* simultâneas.

Como medidas de avaliação de desempenho das implementações, foi o usado o *Speedup* (equação 3) e o número de instruções por segundo (equação 4).

$$Speedup(i, v) = \frac{T_{i,seq}}{T_{i,v}} \tag{3}$$

$$GOP/s(i, v) = \frac{2^i \ log \ log \ 2^i}{T_{i,v} \cdot 10^9} \tag{4}$$

onde:

$T_{i,seq}$: Tempo de execução na versão sequencial para $n = 2^i$
$T_{i,v}$: Tempo de execução numa versão v do algoritmo para $n = 2^i$

As experiências foram realizadas em *Desktops* com um CPU Intel Core^{TM} i7-4790 de frequência 3.60GHz (com "Turbo Boost" até 4.00GHz) com 4 *cores*. O CPU apresenta 4 *caches* de dados L1 de 32KB, 4 *caches* de dados/instruções L2 de 256KB (uma por cada *core*), bem como uma *cache* de dados/instruções L3 de 8MB.

5.2 Análise dos Resultados

5.2.1 Algoritmo Sequencial

Na figura 3 estão presentes os diferentes valores de *performance* do algoritmo sequencial para vários tamanhos do problema, permitindo verificar uma descida geral da *performance* com o tamanho do problema (mais visível entre $n = 2^{25}$ e $i = 2^{26}$) que poderá estar relacionado com algum tipo de ineficiência ao nível da gestão de memória (que talvez pudesse ser melhorada com a reorganização dos ciclos do programa).

5.2.2 Paralelismo com Memória Partilhada

Relativamente à versão paralelizada usando memória partilhada (através de *threads*), constatou-se (através da figura 4 e tabela 1) que não existe melhoria de desempenho ao utilizar-se mais do que 4 *threads*, obtendo-se tempos de execução e *speedups* ligeiramente piores do que usando 2 *threads*, sendo esta distinção mais notória para valores de n mais pequenos.

Observa-se também que para $n = 2^{26}$ obtêm-se uma súbita subida e descida posterior do *Speedup*, sendo que para $n > 2^{26}$ existe um ligeiro aumento continuo

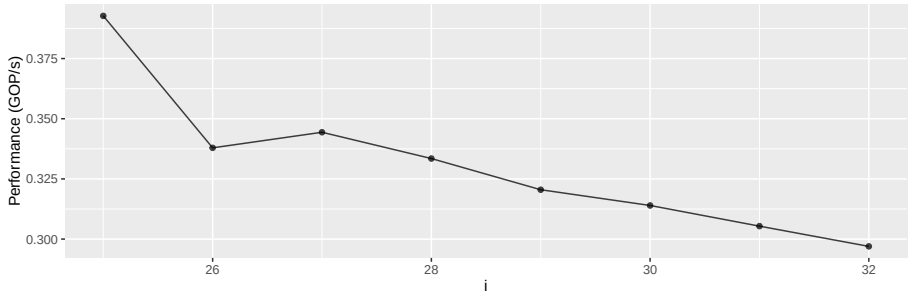

Figura 3: Desempenho da versão sequencial do algoritmo.

do mesmo e uma aproximação dos valores deste para os programas que usaram 2, 6 e 8 *threads*. Dividindo a tarefa de marcação de primos por 4 *threads* consegue-se um *Speedup* máximo de 1.35 para $n = 2^{26}$, que estabiliza perto dos 1.25 quando n se aproxima de 2^{32}.

i	Sequencial	Paralelo (Mem. Partilhada)			
		2	4	6	8
25	0.244	0.195	0.194	0.206	0.210
26	0.574	0.435	0.430	0.448	0.457
27	1.142	0.942	0.919	0.949	0.983
28	2.387	1.966	2.016	2.015	2.031
29	5.026	4.135	4.046	4.169	4.207
30	10.378	8.558	8.369	8.584	8.614
31	21.571	17.587	17.254	17.682	17.747
32	44.822	36.401	35.939	36.544	36.467

Tabela 1: Tempos de execução (em seg.) das versões sequenciais e paralelas (utilizando o modelo de memória partilhada) para 2 a 8 *threads*, variando $n = 2^i$.

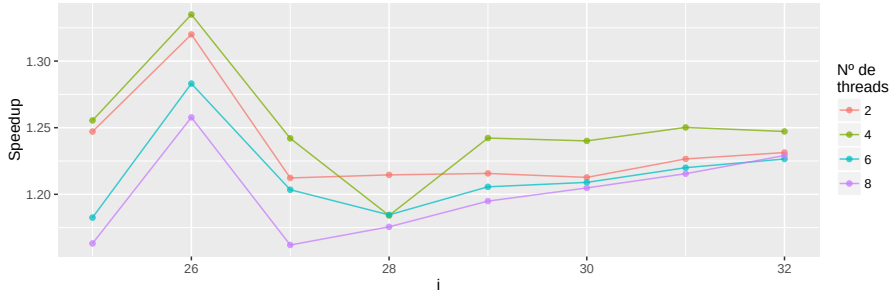

Figura 4: Desempenho da versão com memória partilhada com um número de *threads* variável.

5.2.3 Paralelismo com Memória Distribuída

Utilizando paralelismo com memória distribuída verificaram-se melhorias de desempenho bastante superiores à versão com memória partilhada (ver figura 5 e tabela 2), sendo que para o mesmo n o *Speedup* é sempre maior quando o número de processos é maior. Conclui-se também que o valor do *Speedup* estabiliza perto 4.8 para a versão com 16 processos e $n = 2^{32}$.

De notar que, o paralelismo com memória distribuída obtêm sempre melhores resultados que o paralelismo com memória partilhada, mesmo quando o número de processos num é igual ao número de *threads* no outro.

i	Paralelo (Memória Distribuída)							
	2	4	6	8	10	12	14	16
25	0.195	0.213	0.134	0.096	0.043	0.041	0.040	0.046
26	0.433	0.446	0.293	0.203	0.167	0.140	0.127	0.104
27	0.919	0.944	0.629	0.489	0.372	0.312	0.283	0.219
28	1.946	2.026	1.329	1.023	0.801	0.687	0.614	0.512
29	4.034	3.882	2.566	2.149	1.729	1.459	1.278	1.084
30	8.365	8.510	5.324	4.501	3.463	2.983	2.566	2.271
31	17.250	17.520	11.120	8.962	6.830	6.198	5.315	4.579
32	35.413	35.251	23.465	18.838	14.354	11.808	10.120	9.320

Tabela 2: Tempos de execução (em seg.) da versão distribuída (utilizando o modelo de memória distribuída) para 2 a 16 processos, variando i (tamanho do problema).

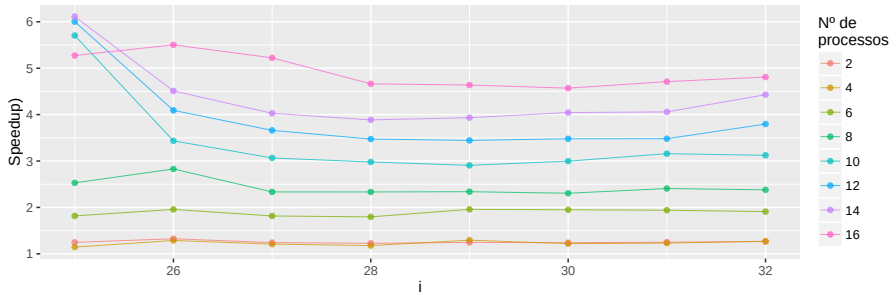

Figura 5: Desempenho da versão com memória distribuída com um número de processos variável.

5.2.4 Paralelismo com Memória Partilhada e Distribuída

Utilizando um misto de memória partilhada e distribuída obtiveram-se resultados ligeiramente melhores aos obtidos utilizando apenas memória distribuída, apesar de se ter verificado um *Speedup* bastante mais alto na maioria das execuções para $n = 2^{25}$ (ver figura 6 e tabela 3). Contudo, à medida que n aumenta o valor do *Speedup* estabiliza entre os 4.5 e os 5 (para $n = 2^{32}$).

		Paralelo (Memória Partilhada e Distribuída)							
Processos		4				8		12	16
Threads		2	4	6	8	2	4	2	2
i	25	0.022	0.017	0.018	0.029	0.059	0.026	0.047	0.028
	26	0.094	0.098	0.098	0.117	0.094	0.109	0.094	0.081
	27	0.235	0.230	0.244	0.288	0.250	0.239	0.222	0.233
	28	0.510	0.501	0.518	0.568	0.499	0.504	0.498	0.498
	29	1.075	1.055	1.085	1.180	1.033	1.236	1.153	1.036
	30	2.237	2.304	2.250	2.540	2.225	2.599	2.206	2.162
	31	4.621	4.516	4.667	5.007	4.464	4.964	4.452	4.488
	32	9.450	9.925	9.500	9.803	9.122	9.202	9.776	9.146

Tabela 3: Tempos de execução (em seg.) da versão híbrida (utilizando o modelo de memória partilhada e distribuída) para 4 a 16 processos e 2 a 8 *threads*, variando i (tamanho do problema).

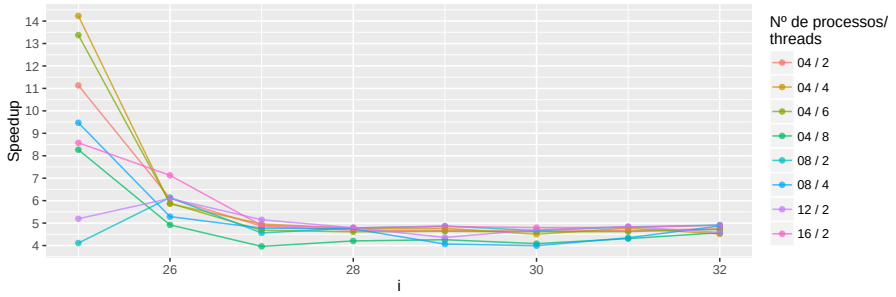

Figura 6: Desempenho da versão híbrida com um número de processos e *threads* variável.

5.2.5 Resultados Globais

De forma a avaliar qual a melhor abordagem das 3 diferentes apresentadas anteriormente, selecionaram-se as 2 melhores versões de cada (sendo que cada versão é caracterizada pelo uso de um certo número de processos e/ou número de *threads*). Considera-se que uma versão do programa obteve um desempenho melhor do que outra caso o valor da métrica $\widetilde{Speedup}$ seja superior (ver equação 5). Optou-se pela mediana no cálculo desta métrica por existirem valores de *Speedup* dispares para alguns valores mais pequenos de i nas figuras 4, 5 e 6.

$$\widetilde{Speedup}(v) = median(Speedup(i, v) : i \in [25, 32]) \tag{5}$$

Os resultados da aplicação desta metodologia de avaliação estão presentes na tabela 4 e permitem verificar que a abordagens com base em memória distribuída conseguem superar largamente as baseadas em memória partilhada (*Speedup* de 4.760 v.s. 1.244). As abordagens híbridas (usando memória partilhada e distribuída) conseguem melhorar ainda mais este resultado em aproximadamente 0.1.

Algoritmo	Nº Processos	Nº Threads	$\widetilde{Speedup}$
MPD	16	2	4.873
MPD	12	1	4.845
MD	16	1	4.760
MD	14	1	4.051
MP	1	4	1.244
MP	1	2	1.221

Tabela 4: Comparação das 2 melhores versões do programa para cada uma das 3 abordagens (Memória Partilhada e Distribuída, Memória Distribuída e Memória Partilhada.

6 Conclusões

Neste relatório foram analisadas 3 possíveis abordagens de paralelização do algoritmo "The Sieve of Eratosthenes", permitindo explorar os conceitos dos modelos de memória partilhada, memória distribuída e memória partilhada e distribuída e fazendo uso das bibliotecas "OpenMP" e "Open MPI". A análise dos resultados da secção anterior permitiu verificar que as abordagens baseadas em memória distribuída obtiveram resultados de *performance* que superaram em 400% as abordagens em memória partilhada, constatando-se que a divisão do problema em processos distribuídos por diversas unidades de processamento como *clusters* (que têm acesso a recursos locais não-partilhados, como memória e disco) é muito mais vantajosa do que a paralelização apenas numa única unidade de processamento.

Pode ainda ser mais vantajoso conjugar estas duas modalidades, usando bibliotecas como o "Open MPI" como um mecanismo de distribuição (para dividir o problema e reparti-lo por processos remotos) e usar paralelismo ao nível local (usando "OpenMP" ou "CUDA") para maximizar a *performance*.

Referências

[1] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. 2003.

[2] Chris K. Caldwell. *The Prime Glossary: Sieve of Eratosthenes*. URL: `http://primes.utm.edu/glossary/xpage/sieveoferatosthenes.html` (acedido em 16/05/2016).

[3] Blaise Barney et al. «Introduction to parallel computing». Em: *Lawrence Livermore National Laboratory* 6.13 (2010), p. 10.