Universidade do Porto

FEUP Faculdade de Engenharia

Mestrado Integrado em Engenharia Informática e Computação

Computação Paralela (CPAR)

Crivo de Eratóstenes

Relatório

Daniel Reis - up201308586 (up201308586@fe.up.pt)

Miguel Botelho - up201304828 (up201304828@fe.up.pt)

Faculdade de Engenharia da Universidade do Porto Rua Roberto Frias, s/n,

4200-465 Porto, Portugal

30 de Abril de 2017

Índice

1 Descrição do Problema

De acordo com a unidade curricular, foi proposto estudar o impacto de diferentes versões de paralelismo ao executar o algoritmo *The Sieve Of Eratosthenes* para números de uma elevada dimensão (2^25 a 2^32). Este algoritmo calcula os números primos até um certo número, definido pelo utilizador.

Assim, devido ao facto do algoritmo ser tão computacionalmente elevado, a *performance* do mesmo será avaliada de três diferentes formas: sequencialmente, paralelamente com memória partilhada, paralelamente com memória distribuída e ainda, uma versão com memória distribuída.

Para alcançar estas avaliações, foi utilizada a biblioteca do OpenMP (*Open Multi Processing*) e o MPI (*Message Passing Interface*).

2 Explicação do Algoritmo

O algoritmo é bastante simples e conciso. Calcula todos os números primos de [2, **n**] com uma complexidade temporal de $O(sqrt(n) * log(log(n))$. No entanto, devido a restrições de memória nos computadores testados, o mesmo foi alterado um pouco, em relação à sua forma inicial. O algoritmo consiste em:

1. Criar uma lista de booleanos, não marcados, com tamanho **n - 1**, de modo a que se possa guardar resultados de [2, **n**].
2. Determinar um número **k** como o menor número da lista não marcado.
3. Depois, marcar todos os seus múltiplos, até **n**, deixando **k** não marcado.
4. Voltar ao passo 2, até que **k** > raíz quadrada de **b**.

Pseudo-Código:

```
BEGIN
    FOR k:= 2 to SQRT(n) DO
        IF list[k] is not marked THEN
            FOR j:= k * k to n STEP j + k
                list[j]:=marked
END;
```

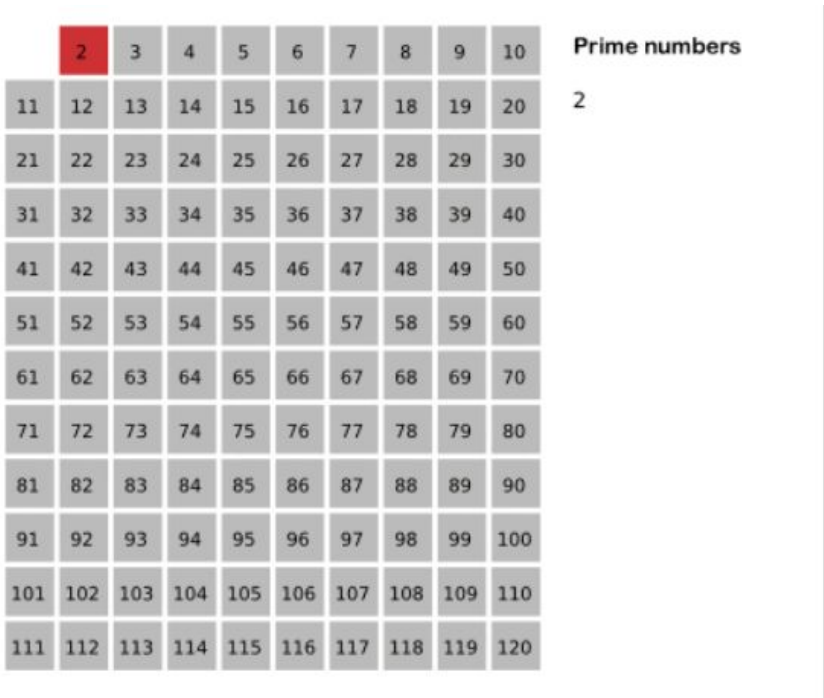

Figura 1: 1ª iteração do algoritmo, selecionando k = 2

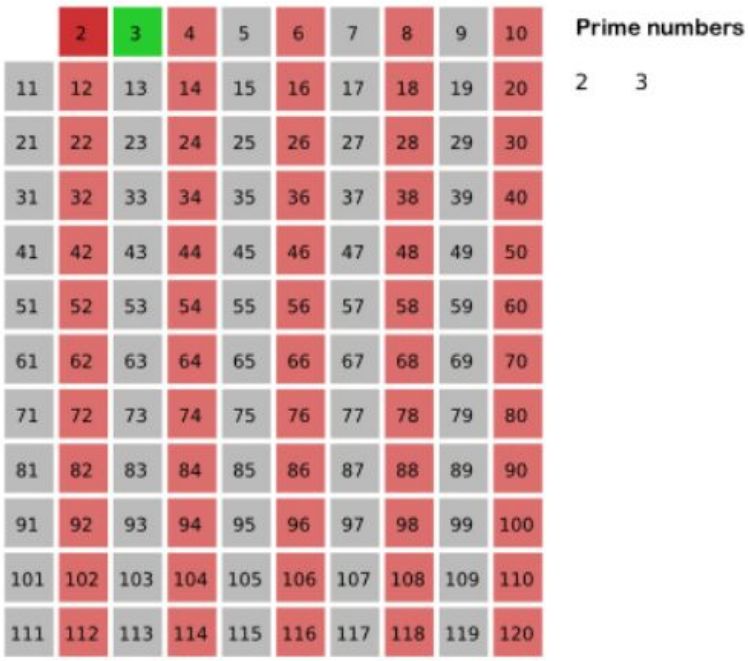

Figura 2: 2ª iteração do algoritmo, após a seleção de todos os múltiplos de k = 2 e escolhendo k = 3, o próximo número não marcado

3 Métricas de Desempenho

As métricas usadas para comparar o desempenho das diferentes execuções do algoritmo foram as seguintes:

- Tempo de Execução
- Número de *Data Cache Misses* (DCM) da cache L1 e L2
- Número de Instruções por Ciclo (IPC)
- Percentagem de falhas de cache
- GFlops

A forma de avaliação dos diferentes desempenhos consistiu, maioritariamente, na comparação entre as métricas dos mesmos.

Características do computador usado para a análise:

- Processador: Intel® Core™ i7-4790 Processor @ 3.60 GHz
- RAM: 16 GB
- Número de Cores: 4
- Número de Threads: 8 (via *hyper-threading*)
- 8 MB Cache

Além disso, foram apenas usados 2 computadores (ambos com as mesmas especificações, a descrita em cima) em todos os testes de programação com memória distribuída (MPI).

4 Resultados e Análise

4.1 Sequencial vs Paralelismo com Memória Partilhada

Ao comparar os tempos de execução do algoritmo sequencial e do paralelo (apenas com OpenMP), é notável a melhoria na redução do tempo, como era expectável. Calculando os números primos até 2^{32}, o algoritmo sequencial registou um tempo de, aproximadamente, 31 segundos, ao passo que o paralelo com OpenMP, com 4 threads, registou quase metade desse tempo, com 16 segundos. Além disso, é também possível verificar que a diferença entre usar 3 ou 4 threads é quase impercetível, como visto no gráfico em baixo.

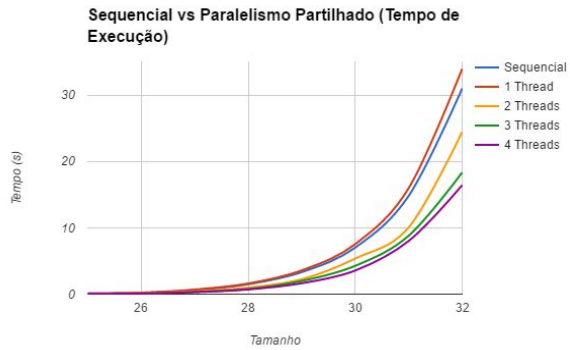

Figura 3: Comparação do tempo de execução

No que toca às falhas de cache L1 e L2, as curvas geradas são aproximadamente iguais às do gráfico anterior, que comparava os tempos de execução. No entanto, nota-se um considerável menor número de *misses* quando são usadas 4 threads, ao invés de apenas 3, o que não era verificado no gráfico anterior.

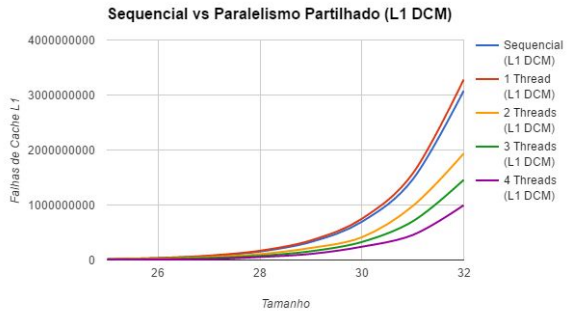

Figura 4: Comparação das falhas de cache L1

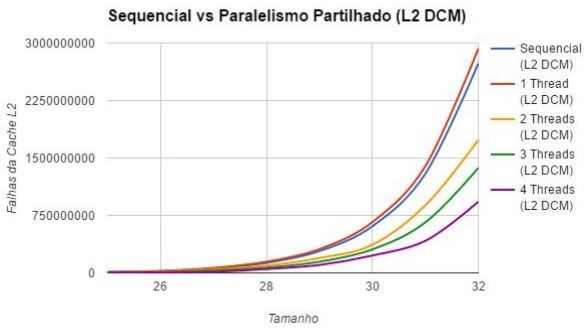

Figura 5: Comparação das falhas de cache L2

4.2 Paralelismo com Memória Distribuída e Paralelismo com Memória Distribuída e Partilhada

No que toca ao paralelismo com memória distribuída, sem o recurso ao OpenMP, é notada uma melhoria quantos mais processos são criados. No entanto, existe um claro limite onde a criação de mais processos não melhora o desempenho do algoritmo, em tempo de execução, que é quando são criados 8 processos, como visto no gráfico em baixo:

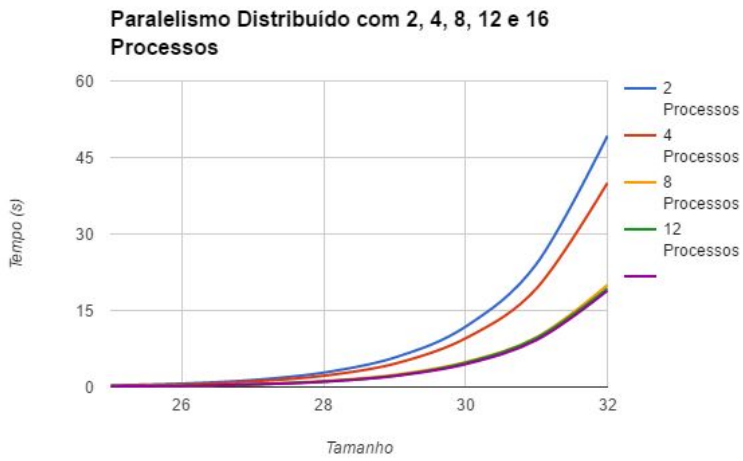

Figura 6: Comparação do tempo de execução com a criação de 2, 4, 8 e 12 processos

Além disso, no paralelismo com memória distribuída e partilhada verifica-se a mesma situação descrita em cima, o número ideal de processos a criar são 8. Contudo, a criação de threads extra, com recurso ao OpenMP, não se mostrou significativamente importante, visto que mantém os tempos de execução como visto nos gráficos em baixo:

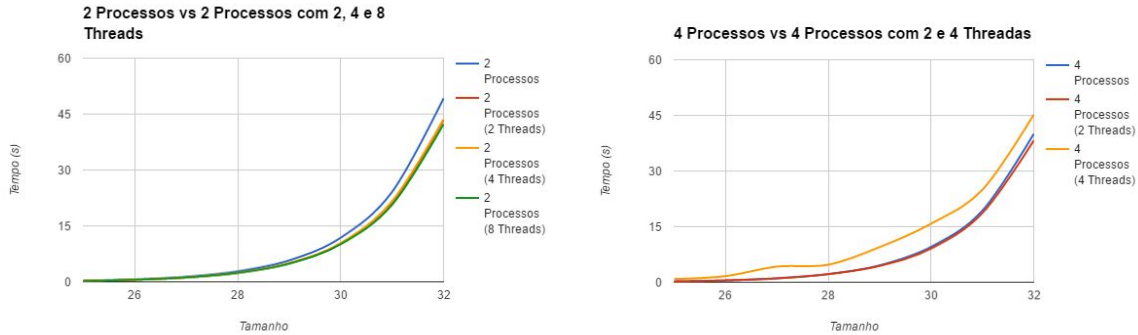

Figura 7 e 8: Comparação do Tempo de Execução com 2 e 4 processos, com e sem threads

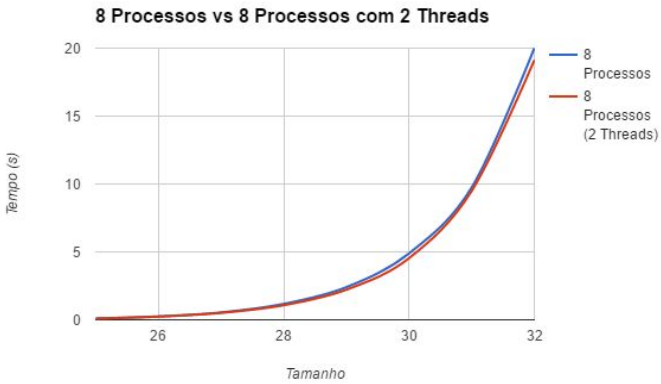

Figura 9: Comparação do Tempo de Execução com 8 processos, com e sem threads

4.3 Paralelismo com Memória Partilhada vs Paralelismo com Memória Distribuída

Após a execução de ambos os algoritmos, a versão com memória partilhada (OpenMP) teve um tempo de execução ligeiramente menor do que a versão com memória distribuída (OpenMPI). Infelizmente, não foi possível medir as *misses* da cache L1 e L2 na versão com memória distribuída, devido ao facto do programa estar a correr em computadores diferentes.

Como visto no gráfico em baixo, a criação de mais processos melhora, substancialmente, o tempo de execução. No entanto, nota-se claramente que existe um limite de tempo de execução ao criar mais processos, visto que a diferença de tempos entre 8 e 16 processos é quase insignificante.

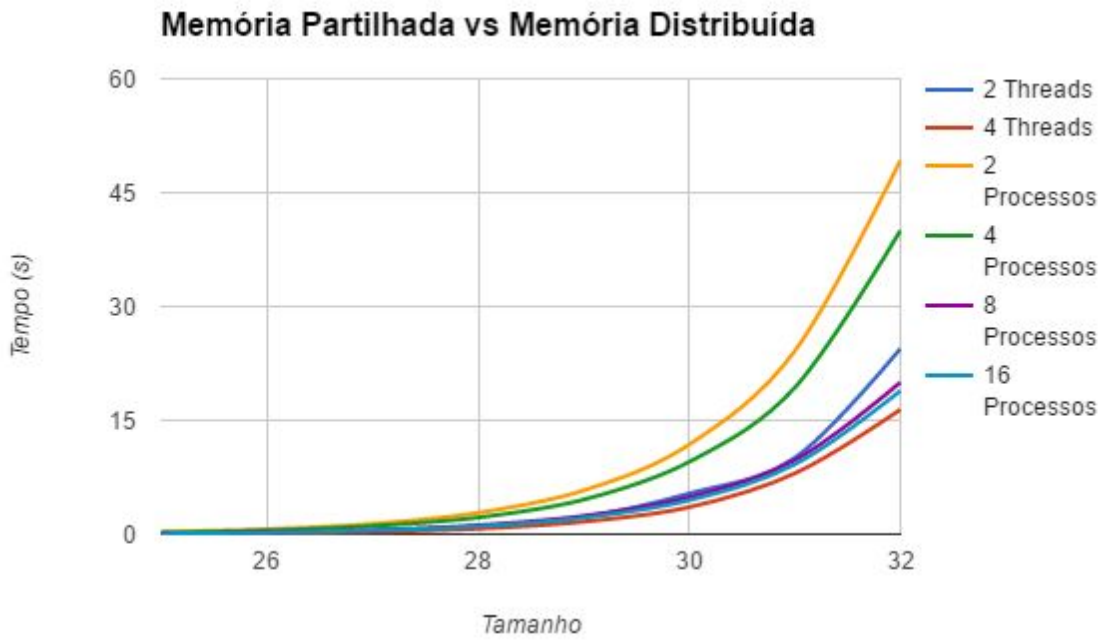

Figura 10: Comparação do tempo de execução do paralelismo com memória partilhada e memória distribuída

4.4 Comparação de Performances e Análise de Escalabilidade

A paralelização do algoritmo usando OpenMP e OpenMPI revelou ser bastante vantajosa. Ambas as implementações revelaram ser mais rápidas que a implementação sequencial. O OpenMP demonstrou ser uma boa solução quando se pretende fazer uma implementação local usando vários cores mas é mais difícil de programar. Já o OpenMPI revelou ser bastante eficiente quando se pretende escalar o algoritmo, mas a intercomunicação de processos pela rede demora mais que o OpenMP. No entanto, caso

usássemos mais computadores aquando da execução do OpenMPI, seria expectável que os tempos registados reduzissem ainda mais, sendo até mais rápido do que a implementação do paralelismo com memória partilhada.

É também importante salientar que o processador onde foram feitos todos os testes suporta *hyper-threading*, uma tecnologia que permite que cada *core* do processador possa executar mais de um processo de uma única vez, sendo então compreensível que os melhores resultados tenham sido alcançados aquando da criação de 8 threads / processos por computador, apesar do processador apenas ter quatro *cores*.

Implementando o algoritmo usando OpenMP e OpenMPI, notamos que existe uma melhoria significativa que depende inteiramente do range dos dados fornecidos pelo input.

Na tabela em baixo encontram-se os melhores resultados para cada algoritmo.

Algoritmo	Processos	Threads	Tempo (s)	Speedup
Partilhado e Distribuído (2 computadores)	8	2	19.1199	1.62
Distribuído (2 computadores)	16	1	18.8806	1.64
Partilhado	1	8	13.8912	2.23
Sequencial	1	1	30.9780	1

Figura 11: Tabela indicativa onde estão registados os melhores tempos de cada algoritmo

5 Conclusões

A realização deste estudo foi bastante importante para a compreender as vantagens e quando recorrer quer seja a programação paralela, quer seja a programação distribuída, assim como os seus limites.

Para paralelizar algoritmos de forma distribuída devemos partir sempre de uma solução sequencial otimizada, de forma a obter resultados significativos, mas é também necessário inferir primeiramente se é realmente vantajoso implementar este tipo de programação. O *overhead* na troca de mensagens e a divisão da carga de trabalho para cada processo pode mesmo tornar a execução do algoritmo mais lenta caso o tamanho de dados seja demasiado pequeno. Sendo assim, a quantidade de dados a processar e forma como estes se processam é determinante para saber de que forma se pode aumentar a performance.

6 Referências

- Tiptop - http://tiptop.gforge.inria.fr/
- Documentação do Tiptop - https://hal.inria.fr/hal-00639173/document
- Documentação do OpenMP - http://www.openmp.org/
- Tutorial de OpenMP - http://bisqwit.iki.fi/story/howto/openmp/
- Open MPI - https://www.open-mpi.org/

7 Anexos

Sequencial:

Tamanho	Sequencial	L1 DCM	L2 DCM
25	0.125	15169455	11131897
26	0.203	33 194 737	22 758 401
27	0.609	72016514	57954352
28	1.493	153944504	131144292
29	3.318	327804939	284318352
30	7.008	694953444	608426178
31	14.803	1465697868	1295161919
32	30.978	3076384332	2734795233

Paralelo (1 thread):

Tamanho	1 Thread	L1 DCM	L2 DCM
25	0.0945	17211601	12952906
26	0.2258	37156864	25820355
27	0.6930	79489707	65417327
28	1.6032	168606104	146092581
29	3.5857	356448215	310349296
30	7.5372	750002605	663032860
31	15.9649	1572674608	1394385354
32	33.9521	3280818175	2928092734

Paralelo (2 threads):

Tamanho	2 Threads	L1 DCM	L2 DCM
25	0.0603	10638231	7401322
26	0.1321	23624565	16493462
27	0.4211	52942420	46668821
28	0.9837	107422852	94429374
29	2.2395	219099520	194442533
30	5.3566	413183178	366389723
31	10.0545	982008491	882282962
32	24.4380	1935971309	1735043095

Paralelo (3 threads):

Tamanho	3 Threads	L1 DCM	L2 DCM
25	0.0540	8408798	6497350
26	0.0937	14180842	11097001
27	0.3329	36921839	33298400
28	0.7720	76157925	68940275
29	1.9852	159550117	146094149
30	4.2620	327236987	306499517
31	8.7992	703752068	659466857
32	18.3265	1457262455	1370300103

Paralelo (4 threads):

Tamanho	4 Threads	L1 DCM	L2 DCM
25	0.0453	3634954	2782953
26	0.0945	8038907	6396632
27	0.2993	15551704	14317568
28	0.7091	54501964	50529622
29	1.6455	110766834	103729763
30	3.5555	241030818	227060949
31	8.0115	454750006	417932025
32	16.4174	995801696	929459559

Paralelo (8 threads):

Tamanho	4 Threads	4 Threads (L1 DCM)	4 Threads (L2 DCM)
25			
26			
27			
28			
29			
30			
31			
32	13.8912	653607321	585231475

MPI (2 processos):

Tamanho	MPI (2 processos)
25	0.3154
26	0.6648
27	1.3637
28	2.8170
29	5.7662
30	11.8171
31	24.1236
32	49.2358

MPI (4 processos):

Tamanho	MPI (4 processos)
25	0.2298
26	0.4988
27	1.0841
28	2.2014
29	4.5557
30	9.5300
31	19.3086
32	39.9936

MPI (8 processos):

Tamanho	MPI (8 processos)
25	0.0986
26	0.2507
27	0.5430
28	1.1711
29	2.4045
30	4.8916
31	9.7430
32	19.9982

MPI (16 processos):

Tamanho	16 processos
25	0.0869
26	0.2206
27	0.4959
28	1.0338
29	2.1609
30	4.4731
31	9.1979
32	18.8806

MPI + OpenMP (2 processos e 2 threads):

Tamanho	2 Processos (2 Threads)
25	0.2539
26	0.5455
27	1.17046
28	2.4183
29	4.9624
30	10.1819
31	20.9528
32	43.4161

MPI + OpenMP (2 processos e 3 threads):

Tamanho	2 Processos (3 Threads)
25	0.2777
26	0.5784
27	1.2195
28	2.6012
29	5.1137
30	10.5052
31	21.5483
32	43.8846

MPI + OpenMP (2 processos e 4 threads):

Tamanho	2 Processos (4 Threads)
25	0.2638
26	0.5848
27	1.1819
28	2.4425
29	5.0766
30	10.3990
31	21.6522
32	43.6244

MPI + OpenMP (2 processos e 8 threads):

Tamanho	2 Processos (8 Threads)
25	0.2629
26	0.5525
27	1.1736
28	2.4062
29	4.8937
30	10.0879
31	20.6715
32	42.2813

MPI + OpenMP (4 processos e 2 threads):

Tamanho	4 Processos (2 Threads)
25	0.2218
26	0.4941
27	1.0293
28	2.18153
29	4.4009
30	9.0725
31	18.6522
32	38.2091

MPI + OpenMP (4 processos e 4 threads):

Tamanho	4 Processos (4 Threads)
25	0.8790
26	1.6579
27	4.2197
28	4.7192
29	9.4678
30	15.7842
31	24.9344
32	45.199

MPI + OpenMP (8 processos e 2 threads):

Tamanho	8 Processos (2 Threads)
25	0.0918
26	0.2327
27	0.5098
28	1.0740
29	2.2155
30	4.5361
31	9.4794
32	19.1199