U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Computação Paralela
EIC0089 | MIEIC 4º Ano - 2º Semestre

Ano Letivo 2016/2017

Relatório

"The Sieve of Eratosthenes"

Paralelização do algoritmo com OpenMP/OpenMPI

Grupo

Diogo Belarmino Coelho Marques

up201305642@fe.up.pt

José Alexandre Barreira Santos Teixeira

up201303930@fe.up.pt

12 de Maio de 2017

Índice

Resumo

No âmbito da unidade curricular optativa *Computação Paralela* integrada no Mestrado Integrado em Engenharia Informática e Computação (MIEIC) da Faculdade de Engenharia da Universidade do Porto (FEUP) foi proposto um estudo da paralelização do algoritmo do crivo de Eratóstenes recorrendo ao OpenMP (para implementação deste algoritmo segundo um modelo de memória partilhada) e a uma implementação open-source do protocolo MPI (*message passing interface*) designada OpenMPI, de forma a facilitar as trocas de mensagens entre computadores na implementação deste algoritmo segundo um modelo de memória distribuída.

1 Descrição do problema

The Sieve of Eratosthenes (crivo de Eratóstenes em português) é um algoritmo simples para encontrar todos os números primos num intervalo ***[2, n] : n ∈ N \ { 1 }***. Um número é considerado primo se e só se for divisível por ele próprio e por 1. O principal objetivo do trabalho será paralelizar este algoritmo de modo a obter um melhor desempenho e escalabilidade para valores de ***n*** de grandeza elevada, ao mesmo tempo que procuramos reduzir a utilização de recursos (nomeadamente memória RAM) implementando uma versão otimizada para correr paralelamente em memória distribuída.

2 Detalhes da implementação

Este algoritmo actua sobre uma lista de ***n*** números inteiros e consiste em marcar todos os múltiplos dos números primos inferiores ou iguais a $\sqrt{n}$. Os números que se encontram marcados correspondem aos números primos encontrados nesse intervalo. Apresenta uma complexidade temporal ***O(n log log n)*** e uma complexidade espacial ***O(n)***.

2.1 Algoritmos sequenciais

Foram implementadas quatro versões sequenciais do Crivo de Eratóstenes de forma a estudar as possíveis melhorias (ou combinações de melhorias) que deveriam ser implementadas no algoritmo paralelo.

1. **Bool Array** (versão *naive* do algoritmo)

    a. os números são representados por um *array* com ***n*** valores booleanos, em que ***n*** representa o número até ao qual pretendemos encontrar todos os números primos;
    b. inicialmente assume-se que todos os números são primos (todos os elementos do *array* são inicializados a *true*);
    c. um elemento ***k*** do array marcado com *true* significa que ***k*** é um número primo; se estiver marcado com false, ***k*** não é um número primo;
    d. executado para todos os números naturais de ***2*** até ***n***;

```
while (k * k <= maximumValue)
{
    for (uint64_t i = k * k; i <= maximumValue; i += k)
    {
        v[i - 2] = false;
    }

    for (uint64_t i = k + 1; i <= maximumValue; ++i)
    {
        if (v[i - 2])
        {
            smallest = i;
            break;
        }
    }

    k = smallest;
}
```

2. **Bitwise** (representação e cálculos da versão *naive* convertidos em operações *bitwise*)

    a. os números são representados por um *array* com ***n*** números inteiros, em que ***n*** representa o número até ao qual pretendemos encontrar todos os números primos;

    b. cada inteiro ***k*** comporta-se como uma máscara de **32 *bits***, sendo que cada *bit* dessa máscara representa um número no intervalo $[2^k, 2^k + 32[$;

    c. sempre que um *bit* ***b*** de um inteiro n estiver a ***"1"***, então $2^k + b$ é um número primo; se esse bit estiver a ***"0"***, $2^k + b$ não é um número primo;

    d. executado para todos os números naturais de ***2*** até raiz quadrada de ***n***;

```
for (uint64_t i = 2; i <= sqrtMaximum; ++i)
{
    if (v[i >> 5] & (1 << (i & 31)))
    {
        for (uint64_t j = i << 1; j < maximumValue; j += i)
        {
            v[j >> 5] &= ~(1 << (j & 31));
        }
    }
}
```

3. **Fast Marking** (marcar todos os múltiplos de *k* que se encontram no mesmo bloco)

    a. os números são representados por um *array* com ***n*** valores booleanos, em que ***n*** representa o número até ao qual pretendemos encontrar todos os números primos;

    b. inicialmente assume-se que todos os números são primos (todos os elementos do *array* são inicializados a *true*);

c. um elemento ***k*** do *array* marcado com *true* significa que ***k*** é um número primo; se estiver marcado com *false*, ***k*** não é um número primo;
d. executado para todos os números naturais *k* a partir de 2, até que a condição ***k * k > n*** se verifique,

```
do
{
    if (k > lowValue)
    {
        firstIndex = k - lowValue + k;
    }
    else if (k * k > lowValue)
    {
        firstIndex = k * k - lowValue;
    }
    else if (lowValue % k == 0)
    {
        firstIndex = 0;
    }
    else
    {
        firstIndex = k - (lowValue % k);
    }

    for (uint64_t i = firstIndex; i <= maximumValue; i += k)
    {
        v[i] = false;
    }

    while (v[++primeIndex] == false)
    {
    }

    k = primeIndex + 2;
} while (k * k <= maximumValue);
```

4. **Skip Even** (semelhante à versão *naive*, mas não considera números pares)

a. baseado na observação de que todos os números primos (à exceção do 2) são ímpares;
b. os números são representados por um *array* com ***(n - 1) / 2*** valores booleanos, em que ***n*** representa o número até ao qual pretendemos encontrar todos os números primos;
c. inicialmente assume-se que todos os números ímpares são primos (todos os elementos do *array* são inicializados a *true*);
d. um elemento ***k*** do *array* marcado com *true* significa que ***2 * k + 1*** é um número primo; se estiver marcado com *false*, ***2 * k + 1*** não é um número primo;

e. executado para os números naturais de ***3*** até raiz quadrada de ***n*** (se ***n*** for ímpar) ou até ***n* - 1** (se ***n*** for par), com incrementos de 2;

```
for (uint64_t i = 3; i * i <= maximumValue; i += 2)
{
    if (v[i / 2])
    {
        for (uint64_t j = i * i; j <= maximumValue; j += 2 * i)
        {
            v[j / 2] = false;
        }
    }
}
```

2.2 Paralelismo com memória partilhada (OpenMP)

Foram implementados dois algoritmos em memória partilhada com recurso ao *OpenMP*. Estes foram adaptados dos algoritmos sequenciais com o mesmo nome, no entanto utilizamos diferentes abordagens na sua paralelização: *FastMarking*, no qual a divisão em blocos é realizada explicitamente no código e paralelizado posteriormente com diretivas *#pragma omp*; por outro lado, no algoritmo *SkipEven*, o *OpenMP* é responsável por distribuir das iterações do ciclo *for* mais exterior pelas *threads*. Para ambos os casos, o paralelismo teve de ser aplicado em blocos específicos do código do algoritmo, pois nem sempre era vantajoso paralelizar tudo.

Melhoria implementada: "*Delete even integers"* (apagar números pares)

- Permite reduzir número de cálculos para metade;
- Liberta espaço, aumentando os recursos de memória disponíveis para computar os números primos (relevante para valores de *n* elevados);
- No caso de *n* = 2^32 (o maior valor de n testado), espera-se que a memória utilizada pelo processo diminua de 4 *gigabytes* (1 *byte* para representar um booleano * 2^{32} elementos no *array*) para 2 *gigabytes* (1 *byte* para representar um booleano * 2^{16} elementos no array, aproximadamente).

2.3 Paralelismo com memória distribuída (MPI)

As implementações em MPI (*message passing interface*) baseiam-se na distribuição da memória por diferentes processos, que podem estar a correr tanto na mesma máquina como em máquinas diferentes numa rede (ou *nodes*), constituindo um *cluster*. Este paralelismo foi aplicado ao nível da estrutura de dados utilizada no algoritmo: um *array* contendo todos os números de ***[3, n]***, que depois é dividido (distribuído) de forma equitativa pelo número de processos a executar, ficando cada processo com a sua respetiva fatia.

A diferença dos algoritmos paralelos desenvolvidos em *OpenMP* para este reside no facto de cada processo guardar o seu bloco em memória local que não é partilhada, realizando a marcação dos múltiplos dos números primos no mesmo. O número primo em cada iteração do algoritmo é "anunciado" pelo processo *root* (com *clusterRank = 0*), sendo partilhado através de *broadcast* aos restantes processos para que estes calculem os seus múltiplos. No final de cada iteração, o processo *root* descobre o próximo primo a analisar e volta a comunicá-lo para que os outros processos possam executar a iteração seguinte.

Foram implementadas duas versões com MPI, ambas utilizando uma versão adaptada do algoritmo sequencial *FastMarking*: uma versão sequencial, que executa este algoritmo sequencialmente mas de forma distribuída, ou seja, utiliza no máximo uma *thread* por processo, bem como uma versão híbrida, que combina *MPI* com *OpenMP*. A única alteração feita nesta versão foi a adição da directiva *#pragma omp parallel for* no ciclo mais interior que permitirá aos processos executarem este algoritmo simultaneamente em mais do que uma *thread*.

3 Resultados

3.1 Métricas de desempenho

As métricas de desempenho utilizadas foram: tempo de execução (em segundos), *speedup* em função da dimensão dos dados **n**, em função do número de *threads* utilizadas (no caso da versão em memória partilhada), e ainda do número de processos utilizados (no caso da versão em memória distribuída com MPI), bem como a eficiência. A métrica do ***speedup*** é calculada em função do tempo de execução da versão sequencial, em que $\mathbf{T_{sequencial}}$ representa o tempo obtido no algoritmo sequencial com melhor tempo de execução, $\mathbf{T_{paralelo}}$ representa os tempos obtidos em cada uma das versões paralelas testadas.

$$Speedup = \frac{T_{sequencial}}{T_{paralelo}}$$

A **eficiência** traduz a taxa de utilização dos processadores das máquinas na execução do programa em paralelo. **P** representa o número de processadores utilizados, **S** o *speedup* obtido ao paralelizar este algoritmo. Esta métrica permitirá tirar conclusões posteriormente quanto à escalabilidade das diferentes versões do algoritmo.

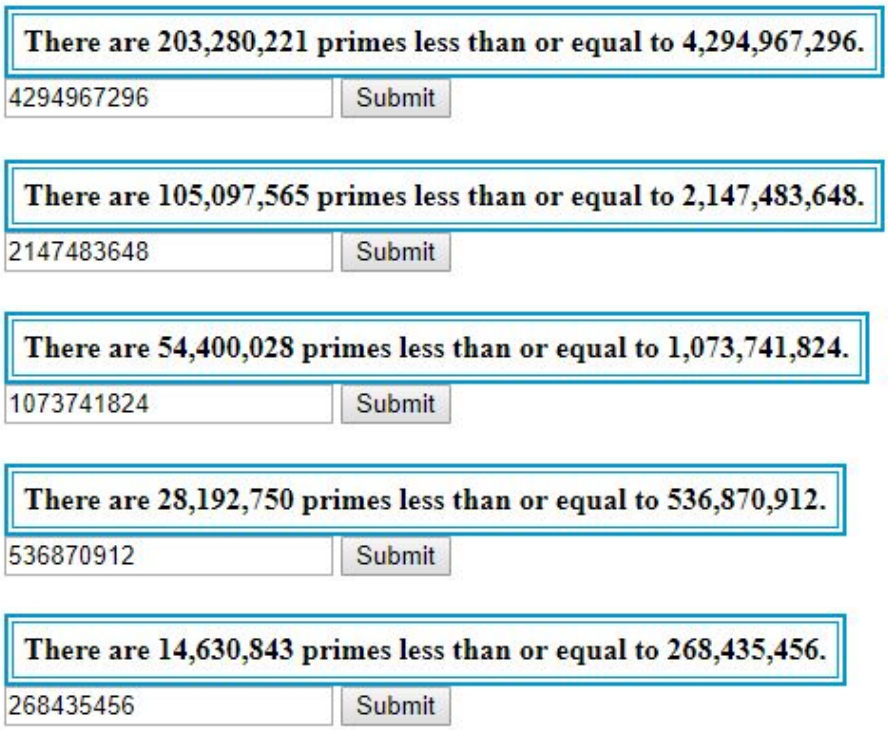
$$E = \frac{Speedup}{P}$$

Outras métricas utilizadas para analisar os resultados obtidos foram: memória RAM utilizada durante (*resident set size*, em *kilobytes*) e memória virtual utilizada (*virtual memory,* em *kilobytes*). Do conjunto de *counters* disponibilizados pela biblioteca *PAPI* foram utilizados três: número de *data cache misses* de nível L1, número de *data cache misses* de nível L2 e número de instruções executadas por cada *thread* do processador.

3.2 Metodologia de avaliação

Como metodologia de avaliação foi utilizado um método de comparação entre os valores obtidos para a execução sequencial e para a execução *multi-threaded*, executando tanto localmente numa máquina como em várias máquinas de forma distribuída recorrendo a MPI (*message passing interface*). Estabeleceram-se algumas comparações tais como a comparação dos tempos de execução, da eficiência, da utilização de memória física/virtual em relação ao valor teórico esperado, do melhoramento dos algoritmos paralelos face à execução sequencial, bem como dos rácios de *data cache misses* pelo número de instruções executadas.

Para validar os resultados obtidos nos diferentes algoritmos e ambientes de execução recorremos a uma ferramenta *online* com funcionalidade semelhante à nossa aplicação. Dado um valor de ***n***, esta ferramenta permite determinar quantos números primos existem até ***n***, embora utilizando um algoritmo diferente. Em baixo apresentamos os resultados obtidos nesta ferramenta para os valores de ***n*** testados (2^{32}, 2^{31}, 2^{30}, 2^{29}, 2^{28} respetivamente).



3.3 Especificações

As experiências foram realizadas num computador *desktop* equipado com um processador Intel Core™ i5-6600K (4 *cores*, 4 *threads*) que apresenta uma frequência base de 3.50GHz. No entanto, realizou-se um *overlock* no qual se conseguiu aumentar a frequência de relógio do processador para 4.40GHz nos quatro *cores*. Este CPU apresenta quatro *caches* de dados L1 de 32 *kilobytes* por *core*, quatro *caches* para dados/instruções de nível L2 com 256 *kilobytes* por *core*, bem como uma *cache* partilhada de nível L3 com 8 *megabytes*. O sistema dispõe ainda de 16GB de memória principal.

No caso da versão distribuída do algoritmo, foi utilizado um computador portátil equipado com um processador Intel Core™ i7-4510U (2 *cores*, 4 *threads*) que apresenta uma frequência base de 2.6 GHz, com 3.1 GHz de *turbo boost*. O sistema dispunha ainda de 4GB de memória principal. O sistema operativo utilizado em ambas as máquinas foi Ubuntu 17.04, atualizado com os *packages* mais recentes à data de realização deste relatório.

As experiências desta versão do algoritmo foram realizadas nos dois computadores acima descritos, variando o número de processos disponíveis em cada computador a partir do ficheiro de configuração *hostfile*. O número de processos permitidos nos computadores variou entre 1, 2 e 4 processos.

3.4 Tempos de execução

Os tempos de execução apresentados nas tabelas seguintes foram obtidos com precisão até aos nanosegundos através da função **int** clock_gettime(**clockid_t** clk_id, **struct timespec*** tp) existente na biblioteca <time.h> do compilador GCC/G++. Apenas se considerou o tempo de execução do ciclo que executa o algoritmo em si, os tempos de inicialização e construção das estruturas de dados auxiliares foram ignorados.

- algoritmos sequenciais

Sequencial	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}
BoolArray	1.386 s	2.872 s	6.013 s	12.457 s	25.547 s
Speedup	1.000	1.000	1.000	1.000	1.000
Média	1.000		**Desvio**	0.000	
Bitwise	1.483 s	3.385 s	6.710 s	14.908 s	31.301 s
Speedup	0.935	0.848	0.896	0.836	0.816
Média	0.727		**Desvio**	0.048	
SkipEven	0.670 s	1.417 s	2.946 s	6.046 s	12.496 s
Speedup	2.069	2.027	2.041	2.060	2.044
Média	2.048		**Desvio**	0.016	
FastMarking	1.461 s	3.043 s	6.249 s	12.986 s	26.900 s
Speedup	0.949	0.944	0.962	0.959	0.950
Média	0.953		**Desvio**	0.008	

- algoritmos paralelos (1 *thread*)

	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}
(Referência)	0.670 s	1.417 s	2.946 s	6.046 s	12.496 s
FastMarking	0.780 s	1.581 s	3.293 s	6.724 s	13.863 s
Speedup	0.859	0.896	0.895	0.899	0.901
Média	0.890		**Desvio**	0.018	
SkipEven	0.673 s	1.405 s	2.934 s	6.049 s	12.553 s
Speedup	0.996	1.009	1.004	1.000	0..995
Média	1.001		**Desvio**	0.006	

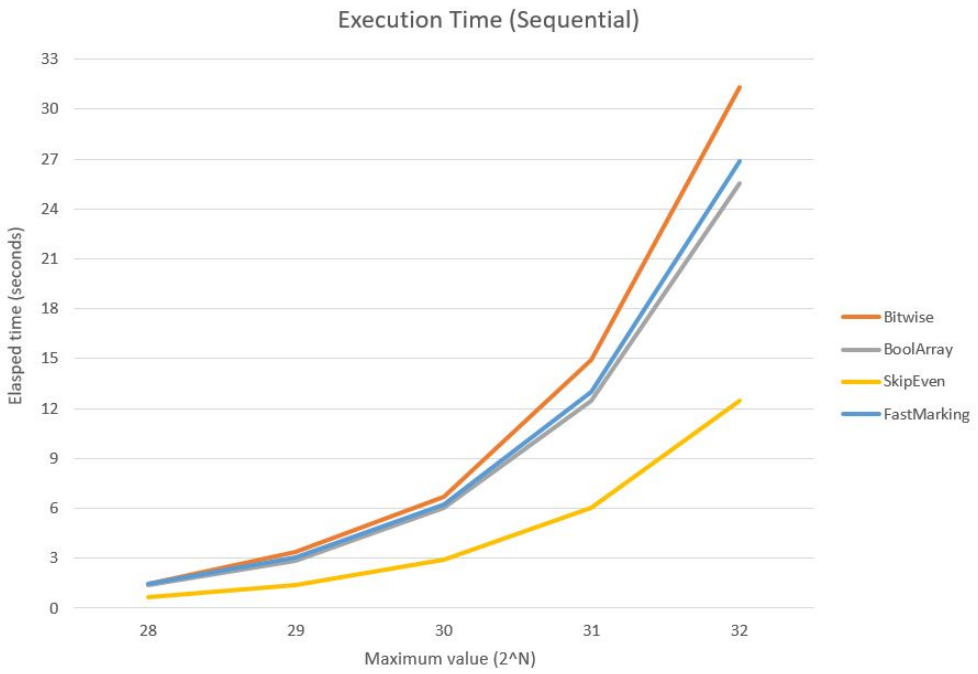
- algoritmos paralelos (2 *threads*)

	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}
(Referência)	0.670 s	1.417 s	2.946 s	6.046 s	12.496 s
FastMarking	0.595 s	1.232 s	2.566 s	5.432 s	11.137 s
Speedup	1.126	1.150	1.148	1.113	1.122
Média	1.132		**Desvio**	0.016	
Eficiência	0.563	0.575	0.574	0.557	0.561
Média	0.566		**Desvio**	0.008	
SkipEven	0.520 s	1.095 s	2.286 s	4.731 s	9.737 s
Speedup	1.288	1.294	1.289	1.278	1.283
Média	1.287		**Desvio**	0.006	
Eficiência	0.644	0.647	0.644	0.639	0.642
Média	0.643		**Desvio**	0.003	

- algoritmos paralelos (4 *threads*)

	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}
(Referência)	0.670 s	1.417 s	2.946 s	6.046 s	12.496 s
FastMarking	0.588 s	1.216 s	2.606 s	5.342 s	10.584 s
Speedup	1.139	1.165	1.130	1.132	1.181
Média	1.150		**Desvio**	0.022	
Eficiência	0.285	0.291	0.283	0.283	0.295
Média	0.287		**Desvio**	0.00006	
SkipEven	0.443 s	0.846 s	2.028 s	3.756 s	7.928 s
Speedup	1.512	1.675	1.453	1.160	1.576
Média	1.565		**Desvio**	0.086	
Eficiência	0.378	0.419	0.363	0.402	0.394
Média	0.391		**Desvio**	0.022	

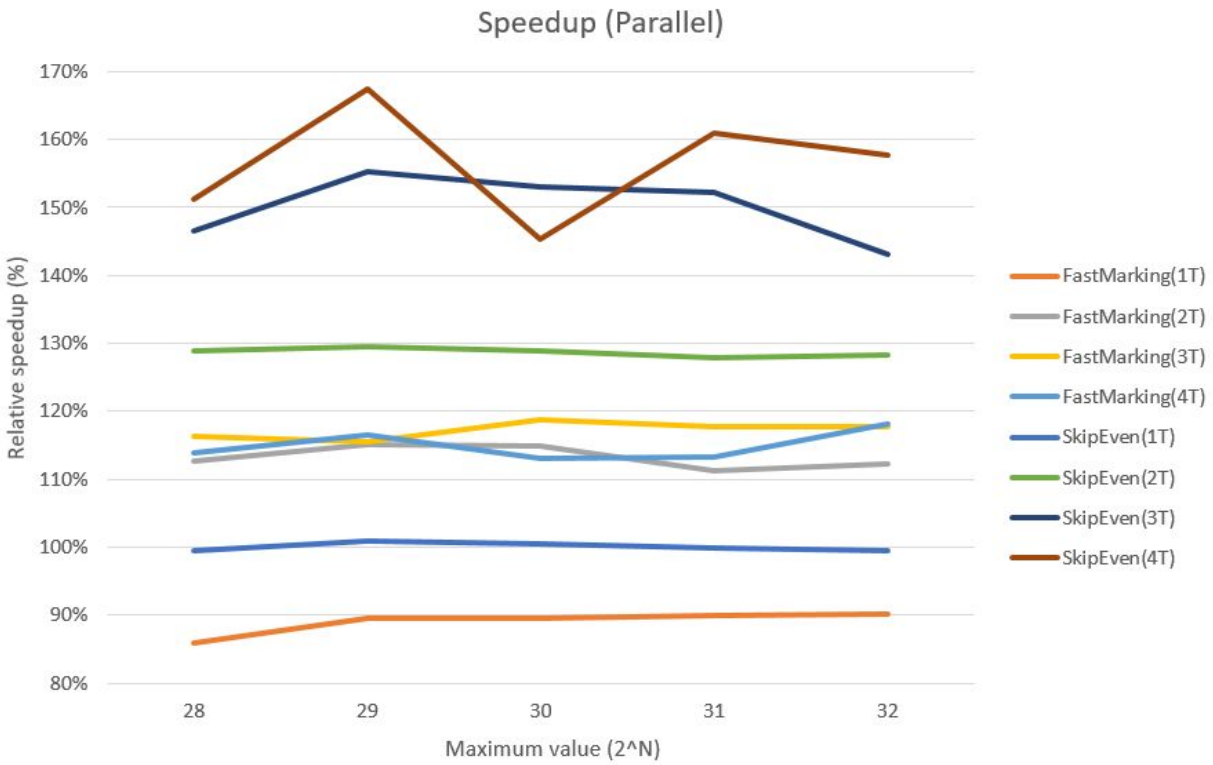
3.5 Análise dos resultados



No teste dos algoritmos sequenciais, ***SkipEven*** apresentou melhores resultados para todos os valores de ***n*** testados, sendo que se conseguiu obter um tempo de execução de 12.496 segundos para ***n = 32*** e um *speedup* médio de ***2.048*** em relação ao algoritmo ***BoolArray***, que por sua vez terminou a *benchmark* em 25.547 segundos. Por outro lado, ***Bitwise*** apresentou os piores resultados dos quatro, com um tempo de execução de 31.301 segundos e um *speedup* médio de ***0.866***, ao qual corresponde uma perda aproximada de ***13%*** no desempenho do programa.

	Bitwise	**BoolArray**	**SkipEven**	**FastMarking**
Physical	530 156 KB	4 200 300 KB	2 102 956 KB	4 200 220 KB
Virtual	549 184 KB	4 219 200 KB	2 122 048 KB	4 219 200 KB

Quanto ao consumo de memória física, ***Bitwise*** foi aquele que apresentou uma menor utilização de recursos (aproximadamente 518 *megabytes* para ***n = 32***), seguido do **SkipEven**, que utilizou 2 *gigabytes* nas mesmas condições. O ***BoolArray*** e ***FastMarking*** foram os algoritmos que apresentaram um pior aproveitamento dos recursos, utilizando 4 *gigabytes*.



No caso dos algoritmos paralelos, conseguiu-se algum paralelismo, no entanto a escalabilidade destes foi pouco satisfatória. Quatro *threads* no caso do algoritmo ***FastMarking*** introduziram um *speedup* médio de 1.15 relativamente ao algoritmo sequencial, enquanto a mesma configuração de *threads* no ***SkipEven*** conduziram a um modesto *speedup* de 1.576 em média. Consideramos também relevante referir que, para **n = 30**, o tempo de execução de três *threads* (1.926s) foi ligeiramente inferior ao tempo de quatro *threads* (2.028s). A mesma situação verifica-se no ***FastMarking*** para valores iguais ou superiores a **n = 30**. Com três *threads* conseguiu-se um desempenho melhor em quatro das cinco situações testadas, com duas *threads* conseguiu-se melhor resultado em **n = 30** mais uma vez do que quatro *threads*.

	FastMarking(2T)	**SkipEven(2T)**	**FastMarking(4T)**	**SkipEven(4T)**
Physical	2 103 208 KB	2 102 996 KB	2 103 268 KB	2 103 004 KB
Virtual	2 195 916 KB	2 130 244 KB	2 343 388 KB	2 146 636 KB

Utilização da memória nos algoritmos paralelos, n = 32

Não foram encontradas grandes diferenças nos consumos de memória física ao passar dos algoritmos sequenciais para os algoritmos paralelos equivalentes. No entanto, detectamos um pequeno *overhead* na memória virtual (um pequeno acréscimo relativamente aos valores obtidos anteriormente para a memória física) que aumentava proporcionalmente ao número de *threads* escolhidos. Este *overhead* estará associado à criação, bem como à gestão das *threads* por parte da biblioteca *OpenMP*. No caso do algoritmo **FastMarking**, devido à existência de dois vetores auxiliares, verifica-se um *overhead* ainda maior, chegando perto dos **50%** da memória total utilizada pelo processo para dimensões pequenas, por exemplo **n = 28**.

4 Conclusão

Com este trabalho foi possível conhecer e aplicar conceitos associados a modelos de programação em memória partilhada e de memória distribuída, *message passing* numa rede de computadores e paralelização de algoritmos. Este trabalho serviu também para aprofundar os conhecimentos de OpenMP adquiridos aquando a realização do primeiro trabalho, bem como introdução à programação em MPI.

Após a análise dos resultados obtidos em cada modelo foi possível concluir que o modelo de memória distribuída por vários computadores é mais vantajoso que o modelo de memória partilhada localmente num único computador. Para além de ser mais vantajoso, é também mais económico, pois a criação de *clusters* a partir de computadores comuns pode revelar-se menos dispendiosa do que adquirir um único computador “topo de gama” que proporcione um alto desempenho com muitos *cores* e dezenas de *gigabytes* de memória, ultrapassando assim uma das barreiras mais frequentes na computação (falta de recursos).

Finalmente, conclui-se também que a combinação dos modelos de memória partilhada e distribuída apresenta ainda mais vantagens do que a utilização de um desses modelos isolada, permitindo um melhor aproveitamento dos recursos computacionais existentes em cada uma das máquinas envolvidas no *cluster*, o que se traduz num aumento da eficiência.

5 Recursos

Segue-se uma lista de todas as referências bibliográficas consultadas, bem como do *software* utilizado ao longo do desenvolvimento do trabalho:

5.1 Referências bibliográficas

- "Optimized Sieve - Codeforces"
  (http://codeforces.com/blog/entry/17411)
- "Parallel Sieve of Eratosthenes - Imagine. Create. Enjoy! @ stephan-brumme.com"
  (http://create.stephan-brumme.com/eratosthenes)
- "Sieve of Eratosthenes - Wikipedia, The Free Encyclopedia"
  (https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes)
- "The Prime Database: The Nth Prime Page"
  (https://primes.utm.edu/nthprime/index.php#piofx)
- "Tutorials · MPI Tutorial"
  (http://mpitutorial.com/tutorials)
- Cordeiro, M. M. F. "Parallelization of the Sieve of Eratosthenes". Faculdade de Engenharia da Universidade do Porto. 2012.
  (https://mmfcordeiro.files.wordpress.com/2012/10/mmfcordeiro-parallelization-of-the-sieve-of-eratosthenes.pdf)
- Weeden, A. "Parallelization: Sieve of Eratosthenes". Shodor Education Foundation, Inc.
  (http://http://www.shodor.org/media/content//petascale/materials/UPModules/sieveOfEratosthenes/module_document_pdf.pdf)

5.2 *Software* utilizado

- **"Eclipse C/C++ Development Tooling"**

  (https://eclipse.org/cdt)

  The CDT Project provides a fully functional C and C++ Integrated Development Environment based on the Eclipse platform. Features include: support for project creation and managed build for various toolchains, standard make build, source navigation, various source knowledge tools, such as type hierarchy, call graph, include browser, macro definition browser, code editor with syntax highlighting, folding and hyperlink navigation, source code refactoring and code generation, visual debugging tools, including memory, registers, and disassembly viewers.

- **"OpenMPI: Open Source High Performance Computing"**

  (https://www.open-mpi.org)

  The OpenMPI Project is an open source Message Passing Interface implementation that is developed and maintained by a consortium of academic, research, and industry partners. OpenMPI is therefore able to combine the expertise, technologies, and resources from all across the High Performance Computing community in order to build the best MPI library available.

- **"PAPI (Performance Application Programming Interface)"**

  (http://http://icl.utk.edu/papi)

  PAPI provides the tool designer and application engineer with a consistent interface and methodology for use of the performance counter hardware found in most major microprocessors. PAPI enables software engineers to see, in near real time, the relation between software performance and processor events.

- **"Syrupy: System Resource Usage Profile"**

  (https://github.com/jeetsukumaran/Syrupy)

  Syrupy is a Python script that regularly takes snapshots of the memory and CPU load of one or more running processes, so as to dynamically build up a profile of their usage of system resources.