Paralelização do Algoritmo *The Sieve of Eratosthenes*

Relatório

4º ano do Mestrado Integrado em Engenharia Informática e Computação

Computação Paralela

Elementos do grupo:
Henrique Manuel Martins Ferrolho - 201202772 - ei12079@fe.up.pt
João Filipe Figueiredo Pereira - 201104203 - ei12079@fe.up.pt
Leonel Jorge Nogueira Peixoto - 201204919 - ei12079@fe.up.pt

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

25 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* através do uso de APIs como OpenMP para memória partilhada, e de bibliotecas como OpenMPI para memória distribuída.
Neste relatório é descrito o algoritmo proposto, e são discutidas algumas abordagens do mesmo utilizando paralelismo. São ainda apresentadas experiências do desempenho do algoritmo, bem como as suas avaliações e conclusões segundo algumas métricas.

2 Descrição do Problema

The Sieve of Eratosthenes é um algoritmo simples para encontrar todos os números primos num intervalo $[2,\ n]\ :\ n \in N \setminus \{1\}$. Um número é considerado primo se e só se for divisível por ele próprio e por 1.
O objetivo principal deste problema será paralelizar o algoritmo de modo a obter melhor *performance* e escalabilidade para intervalos de grandeza elevada.

3 Algoritmo *The Sieve of Eratosthenes*

O algoritmo *The Sieve of Eratosthenes* atua 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 não se encontram marcados representam o conjunto de números primos encontrados nesse intervalo.

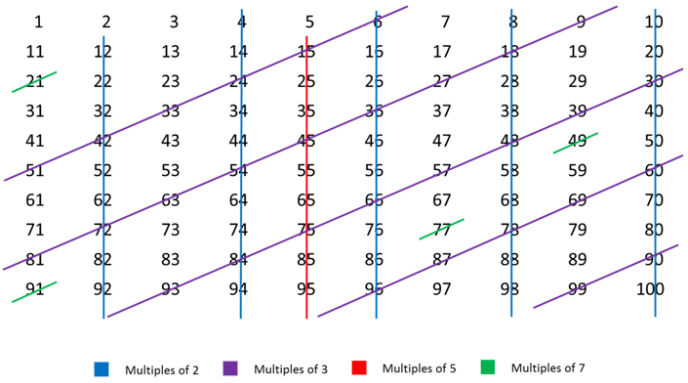

Figura 1: Exemplo do *The Sieve of Eratosthenes*.

As suas complexidades temporal e espacial são $\mathcal{O}(n\ log\ log\ n)$ e $\mathcal{O}(n)$, respetivamente. O pseudo-código é apresentado no **Algorithm 1**:

Algorithm 1 *The Sieve of Eratosthenes*

input an integer n > 1
let A be an **array of boolean values**, indexed by **integers** 2 to n, initially **all set to true**

for $i = 2$, not exceeding $\sqrt{n}$ **do**
    for $j = i^2$, $i^2 + i$, $i^2 + 2i$, $i^2 + 3i$, ..., not exceeding n **do**
        A[j] = **false**
    end for
    do
        $i = i + 1$
    while A[i] is **false and** $i^2 < n$
end for

4 Implementação do Algoritmo

Como foi dito na **Introdução**, foram desenvolvidas algumas implementações do algoritmo para o estudo da *performance* e escalabilidade neste problema: uma versão de memória partilhada, com a *API* **OpenMP**; e outra versão de memória distribuída, com a bilbioteca **OpenMPI**. Foi utilizada a linguagem de programação **C++** para o desenvolvimento de todas as versões.

4.1 Algoritmo Sequencial

Para a versão **sequencial** do algoritmo, apenas foi necessário implementar o seguinte pseudo-código.

Algorithm 2 Versão **Sequencial**

```
for (unsigned long p = 2; pow(p, 2) < size;) {
    for (unsigned long i = pow(p, 2); i < size; i += p)
        list[i] = false;

    do {
        p++;
    } while (!list[p] && pow(p, 2) < size);
}
```

O código apresentado em **Algorithm 2** pode ser encontrado no ficheiro ***SequentialSieve.cpp***, que se encontra na pasta ***src*** do projeto.

4.2 Paralelização do Algoritmo

Uma das soluções para melhorar a *perfomance* do algoritmo é paralelizar o mesmo. O paralelismo pode ser efetuado de duas formas: partilhando a memória pelo número de *threads* disponíveis no processador (**OpenMP**); ou dividindo a estrutura de dados utilizada por diferentes processadores, o que possibilita, neste caso, que cada processo tenha memória local não partilhada com os restantes

(**OpenMPI**). Para ambos os casos, o paralelismo tem de ser aplicado em blocos de código específicos do algoritmo. Paralelizar tudo não é concebível.

Pode ainda utilizar-se um modelo híbrido com partilha e distribuição de memória.

Algorithm 3 Bloco a paralelizar

```
for (unsigned long i = p * p; i < size; i += p)
        list[i] = false;
```

A marcação dos múltiplos do primo p no intervalo $[p^2, size]$ foi o bloco escolhido - como é mostrado no **Algorithm 3**.

4.2.1 Paralelismo com Memória Partilhada - OpenMP

A versão paralela utilizando a API **OpenMP** baseia-se na partilha de memória de um processo em execução, pelas diferentes *threads*, concorrentemente.

O **OpenMP** possui diversas formas de paralelizar um processo. Neste caso, a inserção da linha *pragma omp parallel for* é o necessário para as várias *threads* executarem paralelamente o ciclo pretendido, como se pode verificar no excerto de código apresentado no **Algorithm 4**.

Algorithm 4 Modelo Paralelo com **OpenMP**

```
omp_set_num_threads(threads); // Aplicar o numero de threads ↩
    especificado

for (unsigned long p = 2; p * p < size;) {

   #pragma omp parallel for // Bloco a paralelizar
    for (unsigned long long i = p * p; i < size; i += p)
       list[i] = false;

    do {
       p++;
    } while (!list[p] && p * p < size);
}
```

4.2.2 Versão Paralelizada com Memória Distribuída - OpenMPI

A versão paralela utilizando a biblioteca **OpenMPI** baseia-se na distribuição de memória por diferentes processos. Este método é aplicado ao nível da estrutura de dados que é usada no algoritmo: a lista contendo todos os números de $[2, n]$ é dividida pelo número de processos a executar. Cada processo terá memória local não partilhada para o seu bloco, e efetuará a marcação dos múltiplos dos números primos no mesmo. O número primo de cada ciclo é dado pelo processo *root* (rank $= 0$) e é partilhado, através de *broadcast*, aos processos restantes para que estes calculem os seus múltiplos. Após cada ciclo de cálculo, o processo *root*

descobre o próximo primo a analisar e volta a partilhá-lo.

Para o cálculo do intervalo em cada bloco (**BLOCK_SIZE**), e dos seus valores mínimo (**BLOCK_LOW**) e máximo (**BLOCK_HIGH**), foram definidas *macros* que recebem como parâmetros: o índice do processo, o número de elementos da lista (à exceção do número 1), e o número de processos que irão executar o algoritmo.

Algorithm 5 *Macros* para definir os valores de cada bloco (processo)

```
#define BLOCK_LOW(i, n, p) ((i) * (n) / (p))
#define BLOCK_HIGH(i, n, p) (BLOCK_LOW((i) + 1, n, p) - 1)
#define BLOCK_SIZE(i, n, p) (BLOCK_LOW((i) + 1, n, p) - BLOCK_LOW↩
    (i, n, p))
```

Nesta versão, é necessário calcular um *start value* para cada bloco no início de cada ciclo. Este valor terá obrigatoriamente de ser múltiplo do primo atual.

Algorithm 6 Modelo Paralelo com **Open MPI**

```
for (unsigned long p = 2; p * p <= n;) {
   // calcular o valor de inicio do bloco para cada processo
   if (p * p < lowValue) {
      lowValue % p == 0 ?
            startBlockValue = lowValue :
            startBlockValue = lowValue + (p - (lowValue % p));
   } else {
      startBlockValue = p * p;
   }

   // marcar os multiplos de cada primo
   for (unsigned long i = startBlockValue; i <= highValue; i += p↩
       )
      list[i - lowValue] = false;

   // obter o proximo primo e fazer broadcast para os outros ↩
       processos
   if (rank == 0) {
      do {
         p++;
      } while (!list[p - lowValue] && p * p < highValue);
   }

   MPI_Bcast(&p, 1, MPI_LONG, 0, MPI_COMM_WORLD);
}
```

Como se pode verificar no **Algorithm 6**, se o valor mínimo do bloco for múltiplo do primo atual, então esse valor será o *start value*; caso contrário, é necessário achar o múltiplo mais próximo e atribuí-lo ao valor inicial.

4.2.3 Versão Paralelizada Híbrida

A versão de paralelismo híbrida resulta da junção das implementações do **Algorithm 4** (OpenMP) com o **Algorithm 6** (OpenMPI).

A única adição feita neste modelo foi o *pragma omp parallel for* que, assim como no modelo paralelo com **OpenMP**, servirá para as várias *threads* executarem paralelamente o ciclo pretendido nos diferentes processos.

Algorithm 7 Modelo Híbrido com **OpenMPI** e **OpenMP** - Adição de *pragma*

```
1    omp_set_num_threads(threads); // Alocar numero de threads ↩
         pretendidas
2
3    ...
4
5    #pragma omp parallel for // ciclo a paralelizar
6    for (unsigned long i = startBlockValue; i <= highValue; i += p↩
         )
7      list[i - lowValue] = false;
```

5 Experiências e Análise dos Resultados

5.1 Descrição das Experiências

Nesta secção são apresentados os resultados obtidos para as diversas experiências realizadas com as diferentes versões do algoritmo.

Para as experiências utilizaram-se intervalos de grande escala, nomeadamente, potências de 2 com expoente a variar de 25 a 32: 2^i, $i \in [25, 32]$. Nos modelos paralelos foram adicionados outros fatores como: o número de processos a utilizar, no caso de modelo de memória distribuída (**OpenMPI**); o número de *threads*, no caso de modelo de memória partilhada (**OpenMP**); ambos os anteriores, como no caso do modelo híbrido.

Cada experiência foi realizada em quatro computadores, variando o número de processos disponíveis em cada computador a partir do ficheiro *hostfile*.

Exemplo do ficheiro *hosfile*
192.168.32.151 cpu=1 —— 2 —— 4 —— 8
192.168.32.152 cpu=1 —— 2 —— 4 —— 8
192.168.32.153 cpu=1 —— 2 —— 4 —— 8
192.168.32.150 cpu=1 —— 2 —— 4 —— 8

Os processos permitidos nos computadores variaram entre 1, 2, 4, e 8 processos, permitindo um número total de processos quatro vezes superior aos permitidos (**Equação 1**), e o número de *threads* entre 1 e 8, inclusive.

$$N^oTotalProcessos = 4Computadores * N^oProcessosPermitidos \quad (1)$$

As características dos computadores utilizados nas experiências são as seguintes:

1. **Modelo Processador:** Intel(R) Core^{TM} i7-4790 CPU @ 3.60GHz
2. **Nº de Processadores:** 8
3. ***Cache***:
   (a) 4 *caches* de dados L1 de 32KB
   (b) 4 *caches* de dados/instruções L2 de 256KB
   (c) 1 *cache* de dados/intruções L3 de 8MB
4. **Memória RAM total:** 16342672 KB

5.2 Metodologia de Avaliação

Além das variáveis referidas na secção anterior, foi ainda medido o tempo de execução de cada experiência. O tempo de execução é usado para o cálculo das métricas de *performance* e escalabilidade.

As **métricas de *performance*** utilizadas são: o ***speedup*** em função da dimensão de dados (n), o número de instruções por segundo (***OP/s***), e a **eficiência** em função do número de processadores utilizados.

O ***speedup*** é calculado em função do tempo de execução obtido na versão sequencial, como é apresentado na **Equação 2**. $T_{paralelo}$ representa os tempos obtidos em qualquer das versões paralelas.

$$Speedup = \frac{T_{sequancial}}{T_{paralelo}} \tag{2}$$

O número de instruções por segundo, ***OP/s***, é obtido através da complexidade temporal do algoritmo, como é possível observar na **Equação 3**. T_i corresponde ao tempo de execução obtido em cada experiência.

$$OP/s(i) = \frac{n\ log\ log\ n}{T_i} \tag{3}$$

A **eficiência** é o rácio de utilização dos processadores na execução do programa em paralelo - **Equação 4**. P representa o número de processadores utilizados.

$$E = \frac{Speedup}{P} \tag{4}$$

Esta métrica servirá para concluir a escalabilidade das diferentes versões do algoritmo, sendo que um modelo é escalável se $E \in [0, 1]$ com P e n a aumentarem.

5.3 Análise dos Resultados

5.3.1 Versão Sequencial

Os resultados obtidos para a versão sequencial do algoritmo são apresentados na coluna **Seq.** da **Tabela 1**.

Pode concluir-se que o tempo decorrido aumenta em função do intervalo dado. Note-se ainda que esse aumento é próximo do factor 2 (um dado tempo decorrido é, aproximadamente, o dobro do tempo decorrido anterior), que corresponde também ao aumento da dimensão do intervalo de primos a serem gerados.

Expoente	Seq.	Paralelo - Memória Partilhada			
		2	4	6	8
25	0.292	0.219	0.173	0.207	0.373
26	0.508	0.436	0.401	0.434	0.440
27	1.107	0.945	0.873	0.910	0.999
28	2.320	1.891	1.855	1.915	2.024
29	4.881	3.956	4.196	4.007	4.082
30	10.162	8.156	8.069	8.325	8.442
31	21.160	16.980	16.932	17.101	17.306
32	43.412	35.099	34.872	35.583	35.769

Tabela 1: Tempos de execução (segundos) da versão sequencial e da versão paralela com memória partilhada.

No que diz respeito à *performance* desta versão, podemos verificar um decréscimo no número de operações por segundo - **Figura 2**. Esse decréscimo deve-se a uma gestão de memória mais intensiva, consequente do aumento exponencial do tamanho da lista de primos a processar.

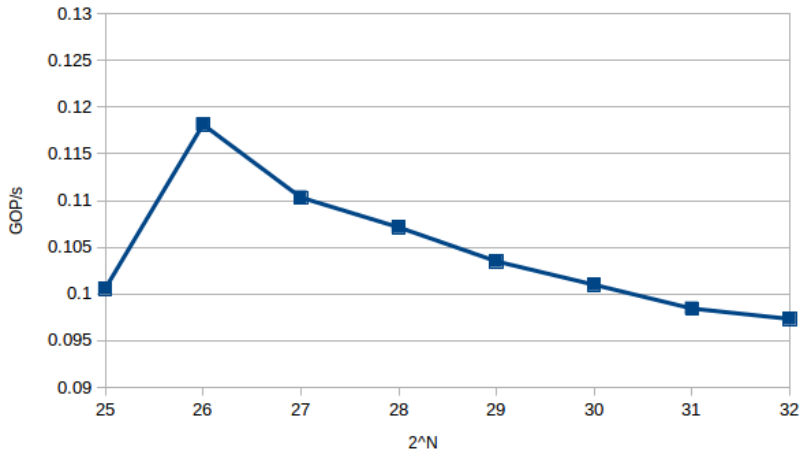

Figura 2: Performance da versão *sequencial* do algoritmo.

5.3.2 Versão de Memória Partilhada - OpenMP

Os resultados obtidos para o modelo de memória partilhada são apresentados na **Tabela 1**, juntamente com os resultados da versão sequencial. Os valores apresentados dizem respeito a um número par de *threads*.
A experiência que originou melhores resultados foi a que usou quatro *threads*. Verificou-se ainda que um número de *threads* superior ao número de *cores* físicos não melhorou os resultados.

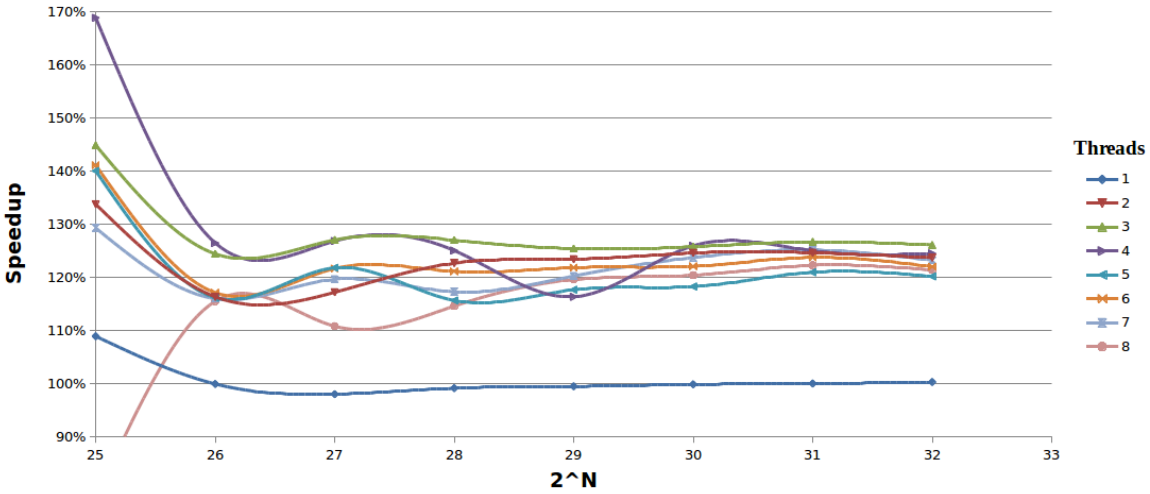

Figura 3: *Speedup* da versão paralela com memória partilhada em relação à versão sequencial do algoritmo.

Com base na **Figura 3**, conclui-se que o *speedup* calculado, em comparação com a versão sequencial, apresenta melhores resultados com o uso de 4 *threads* (tal como nos tempos de execução).

Novamente, o uso de um número de *threads* superior a 4 piora os tempos de execução. Para esses tempos de execução, o *speedup* calculado até chega a ser pior que o uso de 2 *threads*.

Note-se ainda que, quando o número de threads utilizadas é superior a 1, o *speedup* tende para um valor entre [120%, 130%].

5.3.3 Modelo de Memória Distribuída

Os resultados para o modelo paralelo de memória distribuída foram obtidos a partir de 4 experiências: 1 processo em cada computador (4 no total), 2 processos em cada computador (8 no total), 4 processos em cada computador (16 no total) e 8 processos em cada computador (32 no total).

Os *speedups* calculados são apresentados na **Figura 4**.

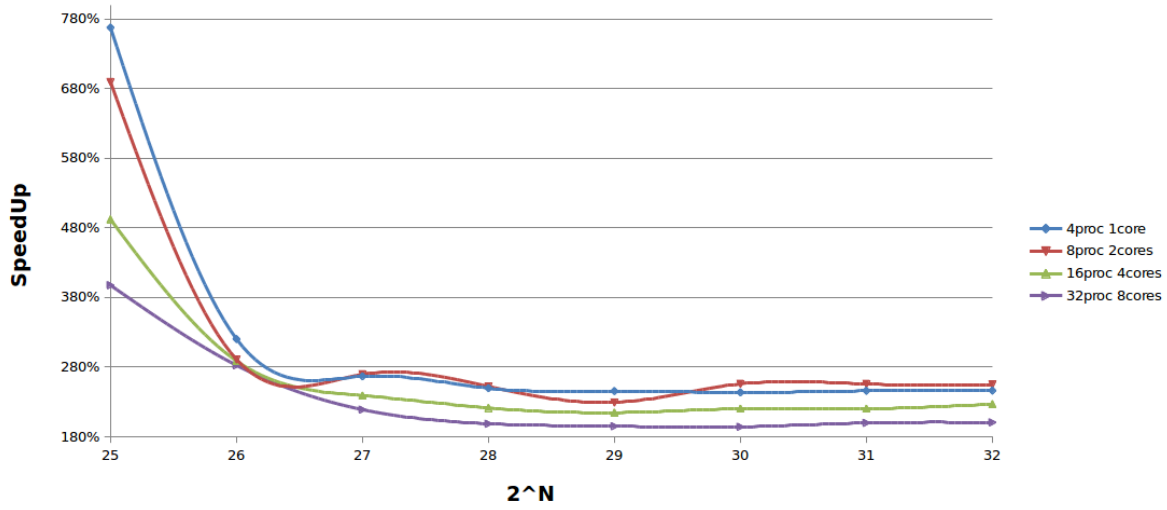

Figura 4: *Speedup* da versão paralela com memória distribuída em relação à versão sequencial do algoritmo

Analisando os valores do *speedup* obtidos, é possível verificar que a paralelização do processo de cálculo com o modelo de memória distribuída apresenta valores bastante mais elevados do que aqueles obtidos com o modelo de memória partilhada. De facto, na **Figura 3**, em média, o *speedup* para a versão de memória partilhada é de 125%, enquanto que, para a versão de memória distribuída, é de 240% (apesar de se verificar um maior *speedup* para $n = 25$).

Conclui-se ainda que o aumento do número de processos não leva ao aumento do *speedup*. Os valores ótimos foram obtidos com 2 processos em cada computador, levando a um total de 8 processos sobre os 32 disponíveis.

GOP/s (2 processos em cada computador)							
N	**2**	**3**	**4**	**5**	**6**	**7**	**8**
25	0.145	0.235	0.232	0.429	0.205	0.499	0.694
26	0.130	0.179	0.197	0.249	0.292	0.353	0.344
27	0.121	0.161	0.176	0.219	0.241	0.280	0.297
28	0.116	0.130	0.168	0.183	0.225	0.256	0.270
29	0.113	0.147	0.163	0.196	0.214	0.243	0.237
30	0.111	0.146	0.160	0.170	0.175	0.190	0.258
31	0.109	0.143	0.157	0.189	0.184	0.232	0.252
32	0.106	0.142	0.155	0.187	0.190	0.222	0.248

Tabela 2: GOP/s para o modelo de memória distribuída.

Com base na **Tabela 2**, podemos observar um grande aumento no número de operações por segundo relativamente à versão sequencial. Esse aumento é um dos fatores para a melhoria dos tempos de execução.

Relativamente à **eficiência** desta versão pode-se verificar que os valores se

encontram todos no intervalo $[0, 1]$ (à exceção de 2^{25} para um processo em cada computador).

Com esta análise, podemos concluir que esta versão é **escalável** para problemas de maiores dimensões.

Eficiência no modelo de memória distribuída					
		P			
N		4	8	16	32
	25	1.325	0.595	0.212	0.086
	26	0.644	0.292	0.145	0.071
	27	0.525	0.265	0.118	0.054
	28	0.491	0.248	0.109	0.049
	29	0.490	0.228	0.107	0.049
	30	0.484	0.254	0.109	0.048
	31	0.485	0.252	0.109	0.049
	32	0.488	0.253	0.112	0.050

Tabela 3: Eficiência obtida para o modelo de memória distribuída.

5.3.4 Modelo híbrido

Os resultados deste modelo foram obtidos a partir de vários testes, em que se fez variar o número de processos e *threads* em cada computador. É possível visualizar os *speedups* na **Figura 5**.

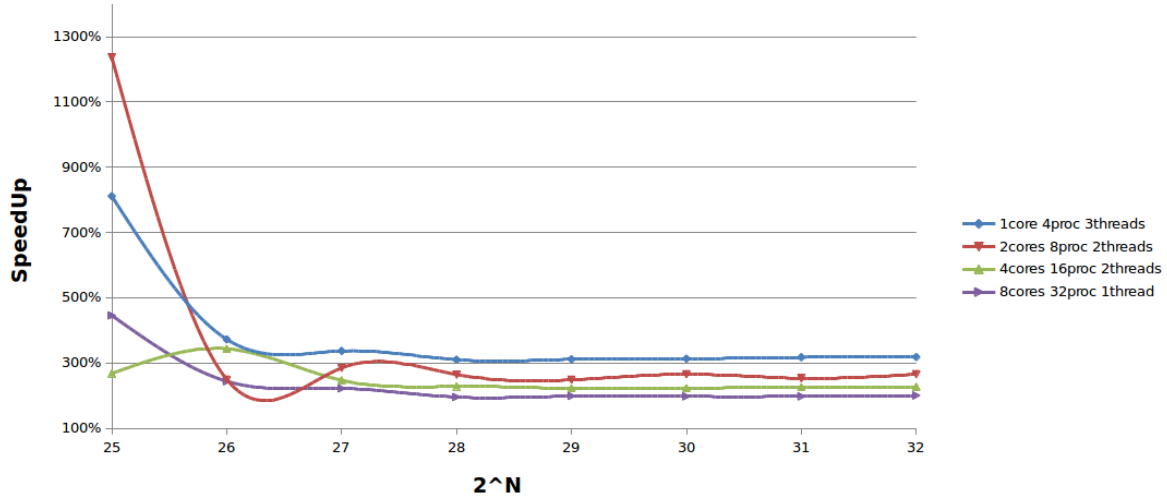

Figura 5: *Speedup* da versão híbrida com memória distribuída e memória partilhada em relação à versão sequencial do algoritmo.

Conjugando os modelos de memória partilhada e memória distribuída, obtiveram-se valores de *speedup* relativamente superiores aos do modelo de memória partilhada (300% vs 240% em média).

Tal como no modelo de memória partilhada, o aumento do número de processos e *threads* não leva ao aumento do *speedup*, pois neste modelo, os valores ótimos foram obtidos com 1 processo de 3 *threads* em cada computador - totalizando 12 *threads* sobre os 32 disponíveis.

Eficiência no modelo de híbrido					
		P			
N		4	8	16	32
	25	2.028	1.545	0.168	0.139
	26	0.931	0.312	0.215	0.076
	27	0.842	0.356	0.154	0.069
	28	0.775	0.331	0.143	0.061
	29	0.779	0.311	0.139	0.062
	30	0.783	0.332	0.139	0.062
	31	0.794	0.317	0.141	0.062
	32	0.796	0.333	0.142	0.063

Tabela 4: Eficiência obtida para o modelo híbrido.

Assim como no modelo de memória distribuída, a análise da eficiência através da **Tabela 4** permite-nos concluir que esta versão é **escalável** a problemas maiores, uma vez que os valores apresentados estão, maioritariamente, entre o intervalo $[0, 1]$.

6 Conclusões

Com este trabalho foi possível conhecer e aplicar conhecimentos sobre a *API* de paralelização *OpenMP* e a biblioteca *OpenMPI*.

Foram implementados três modelos diferentes de forma a paralelizar a geração de números primos pelo algoritmo *The Sieve of Eratosthenes*.

Após a análise de cada modelo, é possível concluir que o uso de memória distribuída por vários computadores é mais vantajosa que o uso de memória partilhada num único computador. Para além de ser mais vantajosa, é também mais económica, pois a criação de *clusters* a partir de computadores comuns pode revelar-se menos dispendiosa do que a aquisição de um único computador de processamento de alta performance com muitos *cores*.

Finalmente, conclui-se também que a combinação dos modelos de memória partilhada e distribuída apresenta ainda mais vantagens do que o uso de um desses modelos isolado.