Relatório de Experiência Laboratorial

Universidade do Porto

Faculdade de Engenharia

FEUP

Mestrado Integrado em Engenharia Informática e Computação

Unidade Curricular de Computação Paralela

João Manuel Ferreira Trindade – 201109221 – ei11118@fe.up.pt

Sérgio Manuel Soares Esteves – 201100784 – ei11162@fe.up.pt

22 de Maio de 2015

Índice

Introdução

Este projeto foi realizado com vista a analisar a influência de diferentes técnicas de paralelização de tarefas na solução de um problema. No decorrer da unidade curricular foram-nos instruídos os dois principais modelos de programação paralela: programação paralela utilizado memória partilhada, e utilizando memória distribuída. Neste trabalho dedicamo-nos, por um lado, a aperfeiçoar o nosso domínio das tecnologias que permitem aplicar estes modelos de paralelização, e por outro lado, a avaliar quantitativamente o desempenho de cada uma das técnicas.

Com vista a determinar o impacto de cada um dos modelos, as técnicas foram aplicadas á computação de números primos de ordem elevada segundo o Crivo de Eratóstenes.

Objetivos

Este projeto teve como objetivos principais:

- Desenvolvimento de versão sequencial para o cálculo de números primos até 2^32.
- Identificação de tarefas passiveis de serem paralelizadas utilizando um modelo de memória partilhada.
- Aplicação de Mecanismos do OpenMP para a paralelização de tarefas.
- Aplicação da metodologia de desenvolvimento de aplicações em memória distribuída (Particionamento do Problema – Padrões de Comunicação – Aglomeração - Mapeamento).
- Aplicação de Mecanismos do MPI para a paralelização de tarefas.
- Aplicação de mecanismos do MPI para a paralelização do problema em várias máquinas.
- Análise de novas métricas de desempenho como *Speedup* Escalável e Eficiência.

Descrição do Problema

O problema computacional escolhido para servir de base á análise dos processos de paralelização foi a geração de números primos. Os números primos têm inúmeras aplicações práticas, sendo usados em áreas como a criptografia (geração de números pseudoaleatórios ou em funções de hashing) ou teoria dos números.

Este problema oferece um nível de complexidade suficientemente grande para que as melhorias da utilização de vários "*cores*" de processamento sejam percetíveis.

Para a realização de testes e medição de tempos de execução e comparação de resultados foram utilizados limites para o valor de entrada, N, de 2^{25} a 2^{32}.

Explicação do Algoritmo

Como já foi referido anteriormente, para calcular os números primos até um dado limite N foi utilizado o Crivo de Eratóstenes.

Um número primo natural é aquele número que tem exatamente dois divisores naturais distintos: o número um e ele mesmo, o próprio número.

O Crivo de Eratóstenes permite-nos calcular os números primos até um dado N de uma forma muito simples e que pode ser descrita em quatro curtos passos:

1. Criação de uma lista de números naturais não marcados: 2, 3, …, n
2. Atribuir o valor de 0 a uma variável k
3. Estrutura Repetitiva
    a. Marcar todos os números naturais, múltiplos de k entre k^2 e n
    b. Atribuir o valor do menor número não marcado à variável k

    Condição de paragem: $k^2 > n$

4. Todos os números que não estiverem marcados na lista são números primos.

Figura 1 - Progresso de Marcação de Numeros Não-Primos

A Figura 1. Demonstra visualmente como progride o algoritmo ao longo do tempo para o intervalo de números de 1 a 120. Inicialmente encontra um número que seja primo, chamado de “semente” de seguida, marca todos os números entre a semente e o fim do intervalo que sejam múltiplos dessa semente como não primos. Todas as sementes do intervalo encontrar-se-ão no intervalo entre 2 e $\sqrt{N}$. Pelo que assim que sejam avaliados todos os números desse intervalo como possíveis sementes, todos os números não-primos, já foram marcados como tal.

Finalmente, faz-se um último varrimento da nossa lista de números, todos aqueles que restarem são números primos.

Implementação do Algoritmo

As implementações do Crivo de Eratóstenes desenvolvidas pelo grupo encontram-se nos anexos do relatório, algumas considerações relevantes sobre as mesmas são:

Versão OpenMP

- É alocado um array de booleanos de dimensão N
  Processo principal computa todas as seeds (entre 2 e raiz de N)
- Processo principal divide o trabalho restante (entre raiz de N e N) pelo número total de processos
- Cada processo processa o seu "CHUNK" do domínio, atualizando diretamente no array inicial

Versão MPI

- Processo principal aloca um array de booleanos de dimensão raiz de N
- Processo principal divide o trabalho restante (entre raiz de N e N) pelo número total de processos.
- Processo principal envia para cada um dos processos o tamanho do problema (N), e os indicies entre os quais deve trabalhar.
- Cada processo, incluído o processo principal, calcula as seeds para o problema de dimensão N.
- Cada processo, incluído o processo principal, aloca localmente um array de booleanos de dimensão do trabalho que recebe, e efetua a marcação dos números primos localmente.
- No fim, cada processo retorna uma mensagem ao processo principal contendo o número total de números primos

Metodologia

Validação de Resultados

Dada a complexidade da validação dos resultados obtidos dada a sua extensão, o grupo realizou comparação automática entre os ficheiros de output das diferentes versões, e com listas obtidas online (https://primes.utm.edu/, s.d.).

Recolha de dados

Com o objetivo de obter valores o mais fidedignos possíveis, as recolhas dos tempos de execução foram executados antes da escrita da lista de primos resultante em disco. Pelo mesmo motivo, na versão em MPI optamos por não retornar a lista concreta dos números primos, mas apenas um numero indicando a quantidade. Isto permite obtermos o tempo

aproximado da conclusão de todos os processos, sem a elevada demora que teríamos da comunicação de um elevado número de dados.

A recolha de dados foi executada desde computadores com as seguintes especificações: Intel 4790 3.6 GHz 8MB Cache & 16GB RAM DDR3 1600MHz, o sistema operativo utilizado foi o Ubuntu 14.04

Sequencial

Foram recolhidos cinco tempos de execução do processo para dimensões do problema (2^N) , com N a variar entre 25 e 32. Num total de 40 medições.

OpenMP

Foram recolhidos cinco tempos de execução do processo para dimensões do problema (2^N) , com N a variar entre 25 e 32. Sendo depois este processo repetido para diferente número de "*cores*", a variar entre 1 e 4. Num total de 160 medições

MPI

Na execução em quarto "*cores*" locais foram recolhidos cinco tempos de execução, com N a variar entre 25 e 32.

De seguida foram recolhidas cinco medições para a execução do problema apenas na dimensão N=32 para diferentes números de "*cores*": 6, 8, 10, 12, 14, 16, 18 e 20.

Num total acumulado de 70 medições.

Para dimensões do problema de tamanho N inferior a 32 não realizamos recolha de tempos de execução entre 6 e 16 "*cores*". Isto porque, nestas condições o tempo despendido nas comunicações tornar-se-ia demasiado significativo relativamente ao tempo despendido na execução dos cálculos.

Resultados

Capacidade

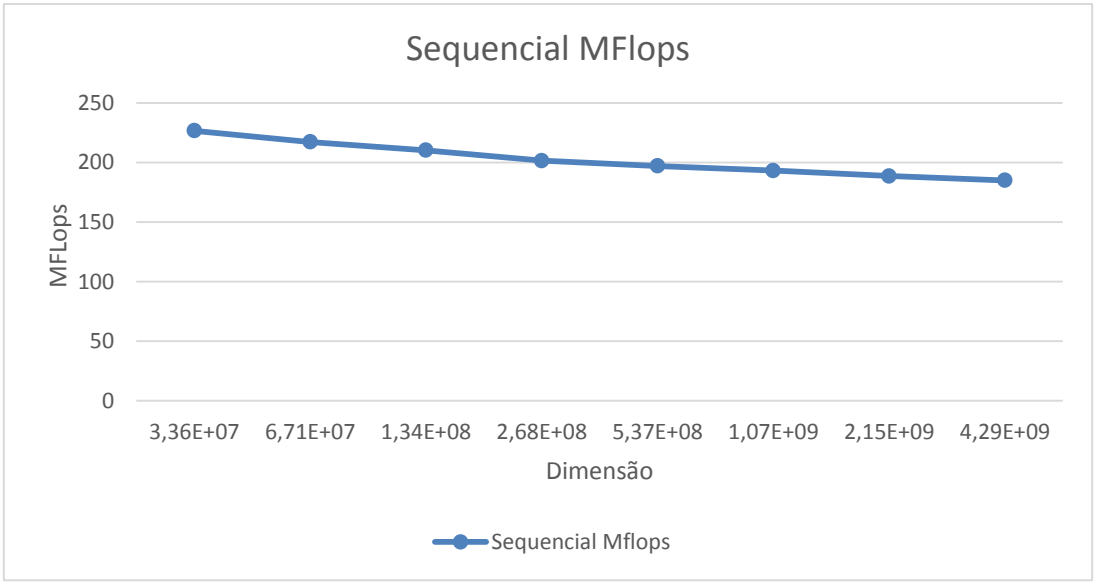

Gráfico 1 - Capacidade Alg. Sequencial

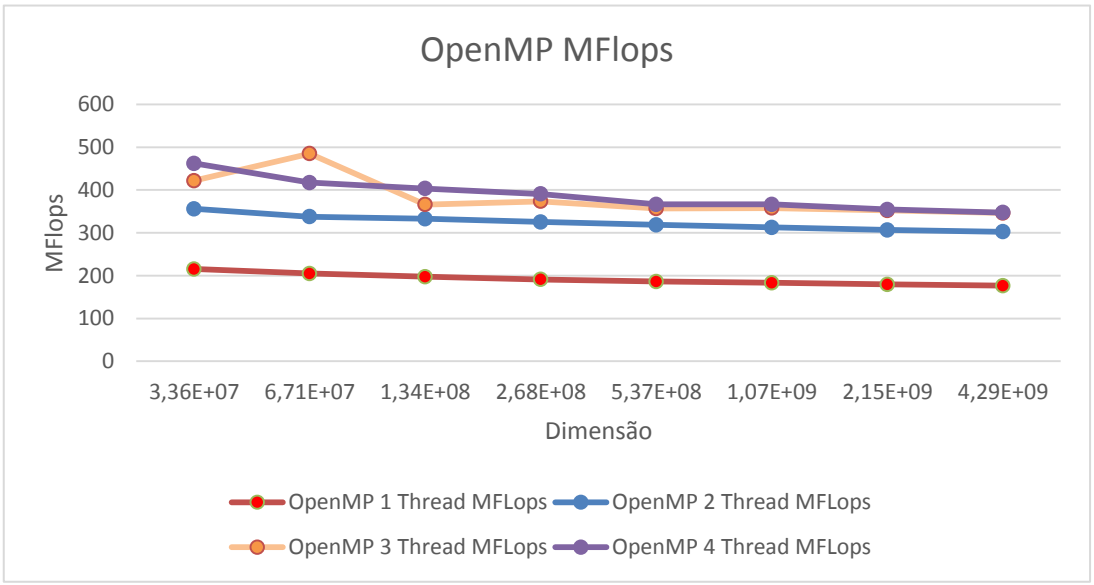

Gráfico 2 - Capacidade Alg. OpenMP

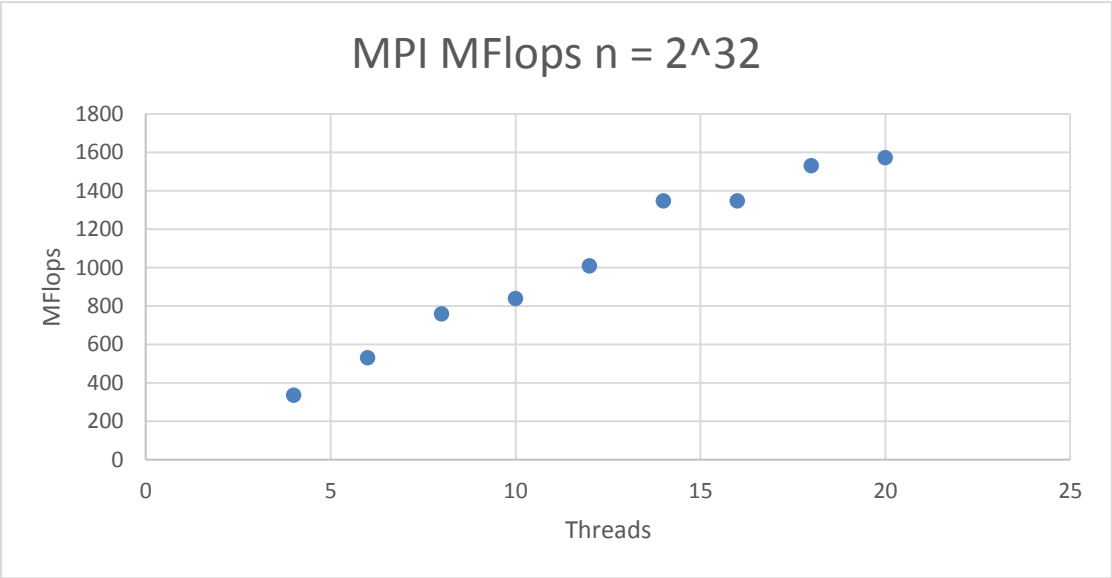

Gráfico 3 - Capacidade Alg. MPI (n=2^32)

Capacidade Entre Versões

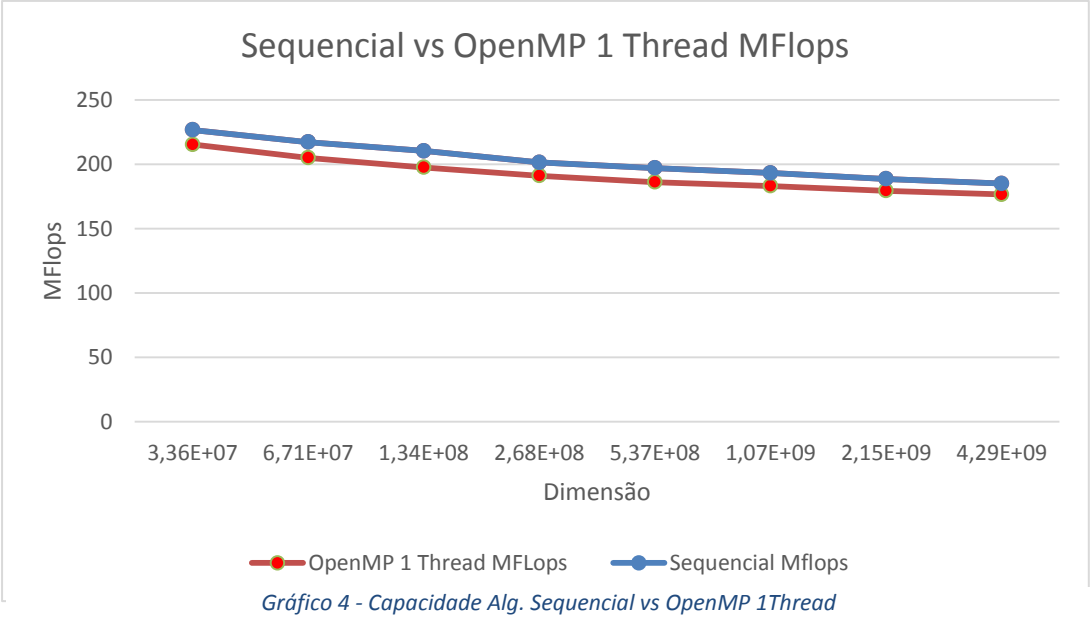

Gráfico 4 - Capacidade Alg. Sequencial vs OpenMP 1Thread

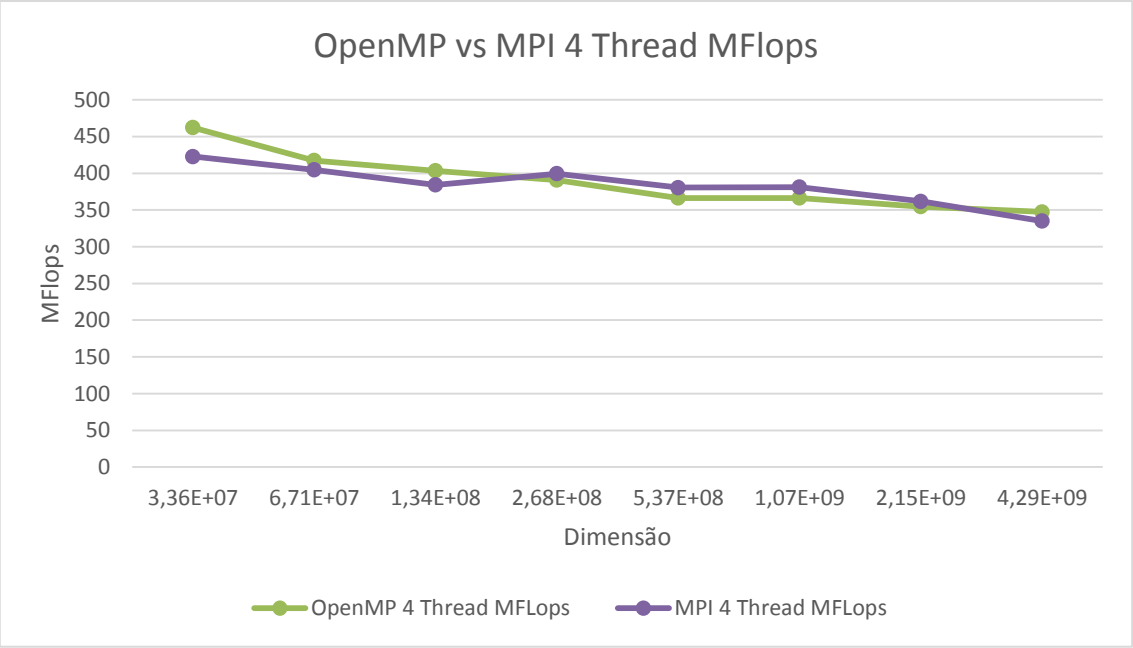

Gráfico 5 - Capacidade Alg. OpenMP vs MPI 4 Threads

SpeedUp

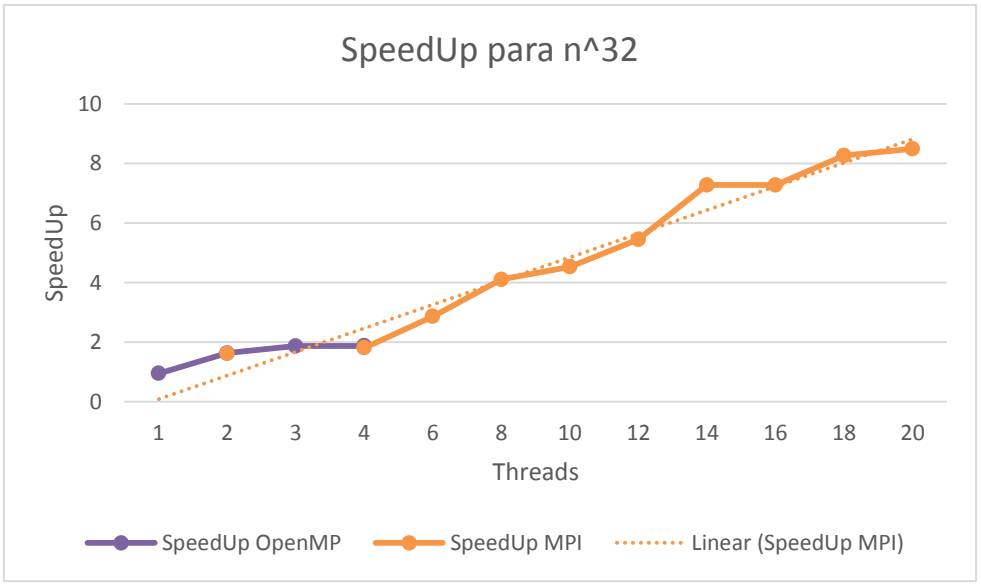

Gráfico 6 - Speedup OpenMP & MPI vs Speedup Linear

Eficiencia

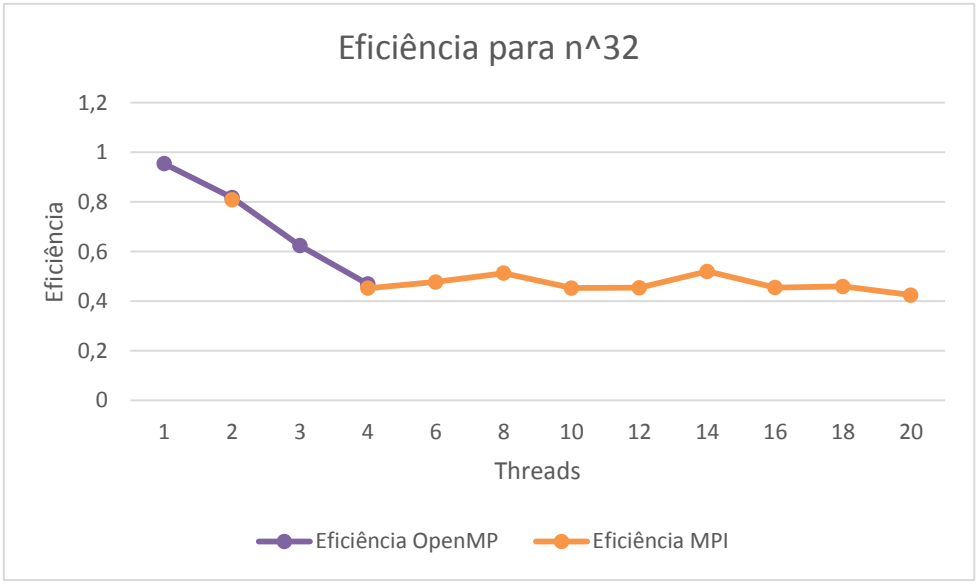

Gráfico 7 - Eficiência OpenMP & MPI

Speedup Escalável

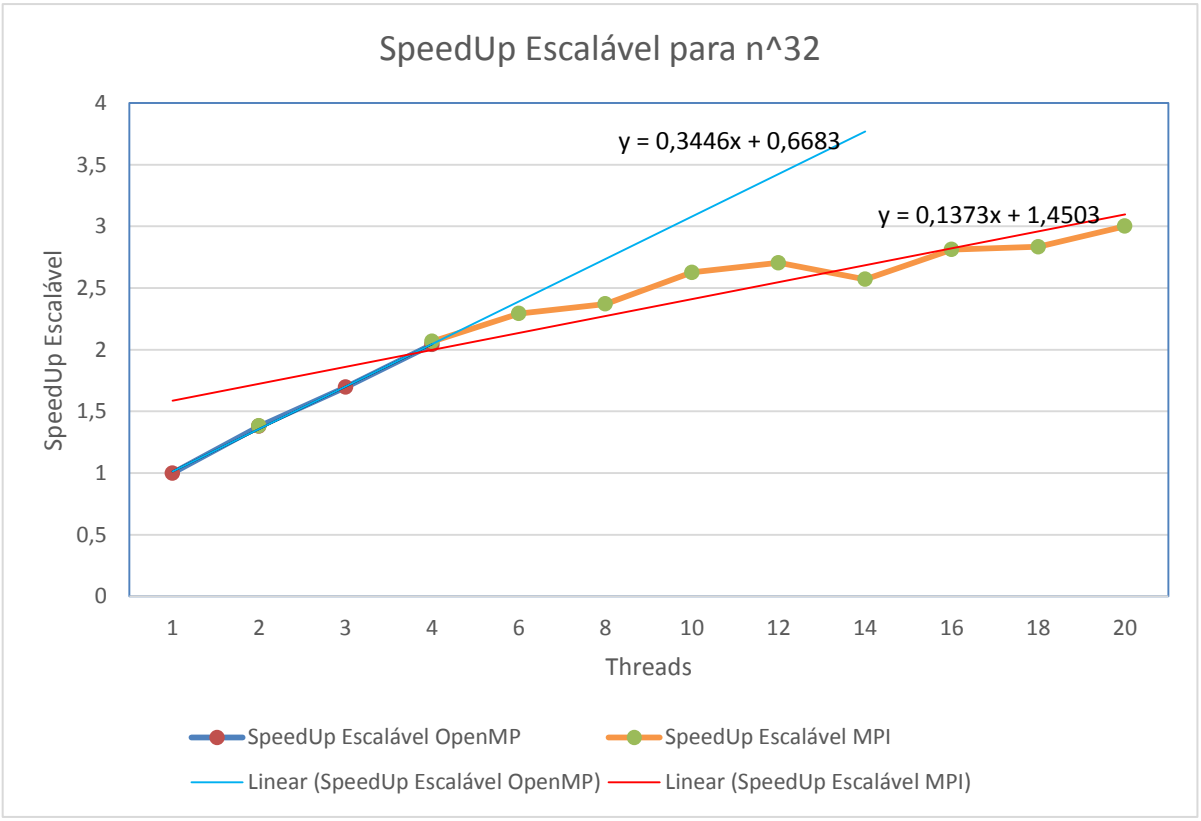

Gráfico 8 - Speedup Escalável OpenMP & MPI e Respectivas linhas de Tendencia

Perfil Execução

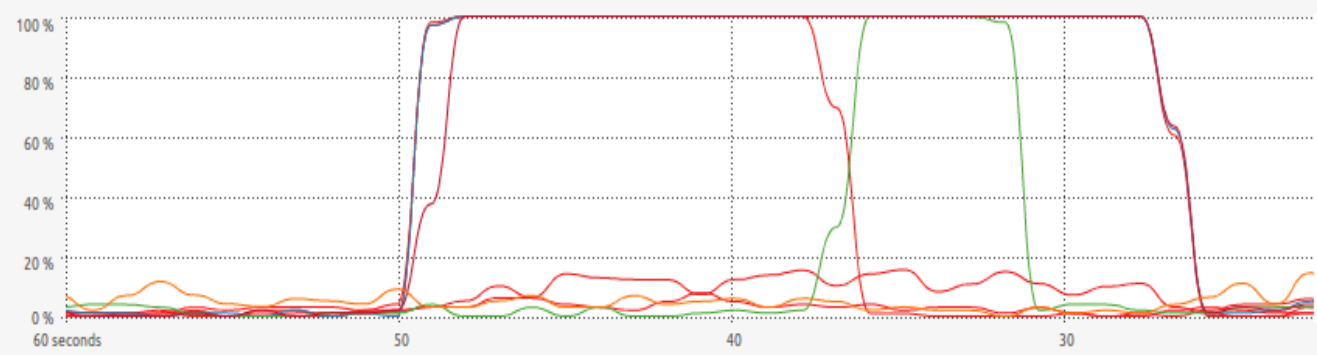

Gráfico 9 - Perfil Utilização CPU MPI 4 core

Análises

Os gráficos 1,2 e 3 representam a capacidade (MFLOPS/S) das três implementações, a sequencial, OpenMP e MPI, respetivamente. Nos três algoritmos podemos verificar uma diminuição progressiva da capacidade com o aumento da dimensão do problema. Tal como seria de esperar o Algoritmo em OpenMP, quando executado com apenas um core, tem o valor de capacidade mais baixo, sendo batido pelo algoritmo sequencial, (Gráfico 4). Isto deve-se aos *overheads* introduzidos pela paralelização, o mesmo que aconteceu no primeiro trabalho laboratorial.

Ao contrário do "Sequencial vs OpenMP em 1 Core", o desempenho do "OpenMP em 4 *Cores*" vs. o "MPI em 4 *Cores*" era uma incógnita para o grupo. Questionamo-nos sobre o que traria maiores *overheads*, se uma paralelização em memoria partilhada realizada pelo OpenMP, se a estratégia adotada pelo MPI de alocar memória para cada processo.

Os resultados são inconclusivos, variam periodicamente com o crescimento do tamanho do problema, seriam necessários mais testes para chegar uma conclusão relevante sobre este tópico. É de salientar também a grande incidência que a nossa implementação pode ter no apuramento deste resultado, e na sua variação, é possível que estes resultados se devam a um subaproveitamento de recursos do lado do OpenMP. No entanto, uma maior capacidade do OpenMP para dimensões mais pequenas do problema era algo que esperávamos, e que foi verificado, mesmo que de forma tangencial. No comparativo de capacidade entre OpenMP e MPI, apartir de N=2^28 (2,68E+8), nota-se que o MPI começa a superar o OpenMP, no entanto, essa superioridade volta a ser perdida em N=2^32 (4,29E9). Este facto foi dos mais complicados de justificar, podendo dever-se a inúmeros factores desde pior desempenho da versão MPI para alocação de maiores blocos em memória até um possível maior stresse na rede (originando maior latência na comunicação) no momento da recolha dos dados.

A métrica de *speedup* indica-nos a melhoria de um algoritmo relativamente à execução da sua versão sequencial. No caso de 2 *cores*, o OpenMP consegui um *speedup* de 1.63, nas mesmas condições o MPI foi marginalmente inferior, obtendo um *Speedup* de 1.61. No caso de 4 *cores*, o ultimo caso partilhado pelas duas soluções, o OpenMP continuou a superar o MPI, 1.87 face a 1.8. Apenas quando executado em 8 "*cores*", o MPI consegui um *Speedup* superior ao que o OpenMP conseguiu com 4 *cores*.

O *Speedup* da versão MPI alcançou os 8.49 quando executada para 20 *cores*, um valor que demostra bem a qualidade da solução alcançada. O *Speedup* desta versão, revelava ainda uma tendência crescente, conforme podemos analisar no Gráfico 6, de facto, o Speedup desta versão é ainda aproximada por uma função linear.

A medida da eficiência (Gráfico 7) analisa o rácio do *Speedup* alcançado face ao número de *cores* utilizados. Os valores obtidos, nomeadamente para o MPI foram bastante interessantes. De facto, tal como esperado, o MPI torna-se bastante ineficiente a partir dos 4 *cores*. Ou seja, apesar de estarmos a conseguir um aumento de performance, o custo em termos de poder

computacional (numero de *cores*) está a ser mais alto, baixando a eficiência do sistema como todo.

O Gráfico 8 demostra o *SpeedUp* Escalavel por numero de *cores* tanto para o OpenMP como para o MPI. As linhas de tendência mostram superioridade por parte do OpenMP, isto é em grande parte devido ao reduzido número de dados (apenas entre 1 e 4 *cores*) de execução do OpenMP. O que este Gráfico 8 representa, é que a nossa solução em OpenMP seria mais facilmente escalável para um problema de dimensão maior, mantendo um número inferior de processadores. Contudo essa informação só seria confirmada com um maior caso de testes. Porque, de facto, o MPI tem o mesmo *SpeedUp* escalável até 4 *cores*, baixando nos restantes 16 pontos analisados, o que contribui muito significativamente para a diminuição do declive da função tendencial.

Como análise extra decidimos incluir um trecho do perfil de execução do processo MPI numa das várias máquinas. Enquanto realizávamos a recolha de dados, observamos no monitor de recursos do sistema, uma tendência que era recorrente, por vezes, um ou dois *cores* terminavam o processamento alguns segundos (para dimensões n>30) antes dos restantes. Isto revelou que a distribuição de trabalho por cada core tinha ainda espaço para otimização.

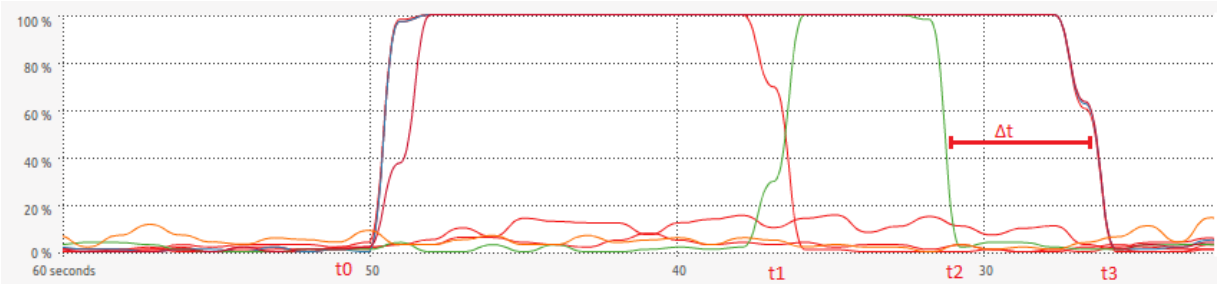

Gráfico 10 - Perfil Utilização CPU MPI 4 core – Analise de Melhoria

Neste exemplo, o instante T0 representa o momento em que o trabalho foi atribuído a este nó de computação. T1 é um momento em que por algum motivo desconhecido, o sistema decidiu alterar o processamento entre dois "*cores*", algo irrelevante para o caso em estudo, mas de interesse assinalar. T2 é o instante em que um dos "*cores*" terminou o seu trabalho e regressou ao estado de "*idle*". Apenas em T3, os restantes 3 processos terminaram a sua atividade. O período de tempo Δt, é o período em que os recursos foram subaproveitados. Em Δt ainda existia trabalho que, se melhor distribuído, poderia estar a ser computado por um maior numero de "*cores*". Uma estratégia de alocamento de *Chunks* de tamanho variável, consoante a sua posição relativa no vetor da solução deveria, em teoria, ser uma boa solução para diminuir este Δt.

Conclusões

Após concluídas as analises, algumas das conclusões que retiramos foram

- Um algoritmo sequencial tem melhor desempenho que um algoritmo em paralelizado, quando executado em apenas 1 core. Isto deve-se essencialmente aos *overheads* introduzidos, por parte do OpenMP, para a criação de tarefas, lançamento de *threads*, etc. (Gráfico 4)
- A partir de dois "*cores"*, inclusive, o desempenho em OpenMP foi sempre superior á execução sequencial (Gráfico 1 e Gráfico 2)
- O desempenho dos algoritmos em OpenMP e MPI, quando ambos são executados em 4 "cores" é muito similar e não existe uma vantagem definitiva para nenhuma das técnicas de paralelização. (Gráfico 5)
- Para dimensões mais pequenas do problema, o OpenMP leva vantagem sobre o MPI quando são ambos executados em 4 *cores*. A conclusão retirada é que o MPI introduz maiores *overheads* temporais, devido á comunicação.
- Foram obtidos resultados bastante positivos da paralelização deste problema, confirmado pelos grandes *Speedups* obtidos: 1.87 para OpenMP 4 Cores até 8.49 para MPI em 20 *Cores*.
- Para o problema em análise, a utilização de mais de 3 *cores* resulta numa eficiência inferior a 50%.
- A partir de 4 *cores*, a eficiência mantem-se aproximadamente constante, pelo que podemos concluir uma boa escalabilidade da nossa solução. Ou seja, quando aumentamos a dimensão do problema, e o número de *cores* utilizados, obtemos sempre uma eficiência aproximadamente igual. Ou seja, não há, por exemplo, uma necessidade de crescer exponencialmente o número de "*cores*" para acompanhar um crescimento do tamanho do problema.
- O *Speedup* Escalável do OpenMP é tendencialmente superior ao do MPI, contudo isto deve-se em parte, ao limitado número de *cores* com que o OpenMP foi executado. (Gráfico 8)
- Era ainda possível optimizar o funcionamento do MPI, alterando a estratégia de alocação de "*Chunks"* (Gráfico 9)

Bibliografia

- Apontamentos das Aulas
- "Using OpenMP", Barbara Chapman
- https://primes.utm.edu/
- http://mpitutorial.com/
- http://www.mpi-forum.org/docs/mpi-1.1/
- https://www.open-mpi.org/doc/v1.8/

Anexos

Anexo A – Algoritmo Sequencial

```
    for(i=2 ; i<sqrt(n)+1; i++)
    {
          if(isprime[i])
          {

            for(unsigned long j = i*i ; j < n; j+=i)
            {
              isprime[j] = false;
            }
          }
    }
```

Anexo B – Algoritmo em OpenMP

```
for (unsigned long i = 2 ; i < sqrt(n) ; i++)
{
          if (isprime[i] == true)
          {
                    seeds[number_seeds++] = i ;
          }
}
```

```
void computeChunk(unsigned long seeds[], int number_seeds, unsigned long startIndex,
unsigned long endIndex, unsigned long CHUNKSIZE)
{
          unsigned long maxLookupIndex = ceil(sqrt(endIndex)) + startIndex ;
          bool foundFirst ;

          for(unsigned long l = 0 ; l < number_seeds ; l++ )
          {
                    unsigned long temp_seed = seeds[l];
                    if (temp_seed  > maxLookupIndex ) break ;
                    foundFirst = false;
                    for(unsigned long i = startIndex ; i <= maxLookupIndex; i++)
                    {
                              if (foundFirst) break;
                              if (i % temp_seed == 0) // encontrou um para marcar
                              {
                                        isprime[i] = false;
                                        foundFirst = true;
                                        for(unsigned long j = i ; j <= endIndex; j+=
temp_seed)
                                        {
                                                  isprime[j] = false;
                                        }
                              }
                    }
          }

}
```

Anexo C – Algoritmo em MPI

```
int computeSeeds(unsigned long n, int seeds[])
```

```
{
    int maxIndex = 2*(int)sqrt(n);
    bool* isprime = new bool[maxIndex];
    memset(isprime, true, maxIndex);
    int number_seeds = 0;
    isprime[0] = isprime[1] = false;
    for(unsigned long i=2 ; i<=sqrt(n)+1; i++)
    {
             if(isprime[i])
             {

                      for(unsigned long j = i*i ; j <= sqrt(n); j+=i)
                      {
                               isprime[j] = false;
                      }
             }
    }

    for (unsigned long i = 2 ; i < sqrt(n) ; i++)
    {
         if (isprime[i] == true)
         {
                  seeds[number_seeds++] = i ;
         }
    }
    return number_seeds ;

}
```

```
void computeChunk(bool primeList[], int seeds[], int number_seeds, unsigned long
startIndex, unsigned long endIndex, unsigned long CHUNKSIZE)
{
         unsigned long maxLookupIndex = ceil(sqrt(endIndex)) + startIndex ;
         bool foundFirst ;
         for(unsigned long l = 0 ; l < number_seeds ; l++ )
         {
                  int temp_seed = seeds[l];
                  if (temp_seed  > maxLookupIndex ) break ;
                  foundFirst = false;
                  for(unsigned long i = 0 ; i <= CHUNKSIZE; i++)
                  {
                           if (foundFirst) break;
                           if ((i + startIndex)  % temp_seed == 0)
                           {
                                    primeList[i] = false;
                                    foundFirst = true;
                                    for(unsigned long j = i ; j <= endIndex -
startIndex ; j+= temp_seed)
                                    {
                                             primeList[j] = false;
                                    }
                           }
                  }
         }
}
```