Faculdade de Engenharia da Universidade do Porto

Parallelization of the Sieve of Erastosthenes

Relatório

Mestrado Integrado em Engenharia Informática e Computação

Computação Paralela, 4º ano

Pedro Dias Faria

ei11167@fe.up.pt

22 de Maio de 2016

Descrição do Problema

O Crivo de Eratosthenes é um algoritmo que tem como finalidade identificar números primos até um certo limite **n**. O algoritmo chega ao seu objectivo marcando todos os múltiplos de um número primo como números compostos, começando com os múltiplos de 2.
Os múltiplos de um dado número primo são gerados como uma sequência começando nesse primo com uma diferença constante entre eles igual a esse número primo.
Este projeto consiste na implementação do Crivo de Eratosthenes para contar a quantidade de números primos até um dado limite **n**, considerando as implementações seguintes:

- sequencial, utilizando apenas um *core* do CPU;
- paralelo, num sistema de memória partilhada, utilizando o OpenMP
- paralelo, num sistema de memória distribuída
    - utilizando apenas MPI
    - utilizando um híbrido de MPI com memória partilhada;

O intervalo de números **n** a considerar é de $\mathbf{2^{25}}$ a $\mathbf{2^{32}}$. Os principais objetivos deste projeto são a implementação do algoritmo e a posterior análise de eficiência, performance e escalabilidade.

Especificações da máquina

O estudo foi feito com recurso a uma máquina com as seguintes especificações:

- **Processador:** Intel(R) Core(TM) Intel(R) Core i7-4790 CPU @ 3.60GHz
- **Memórias Ram:** 15GB
- **Cache:**
    - **L1:** 128KB
    - **L2:** 1024KB

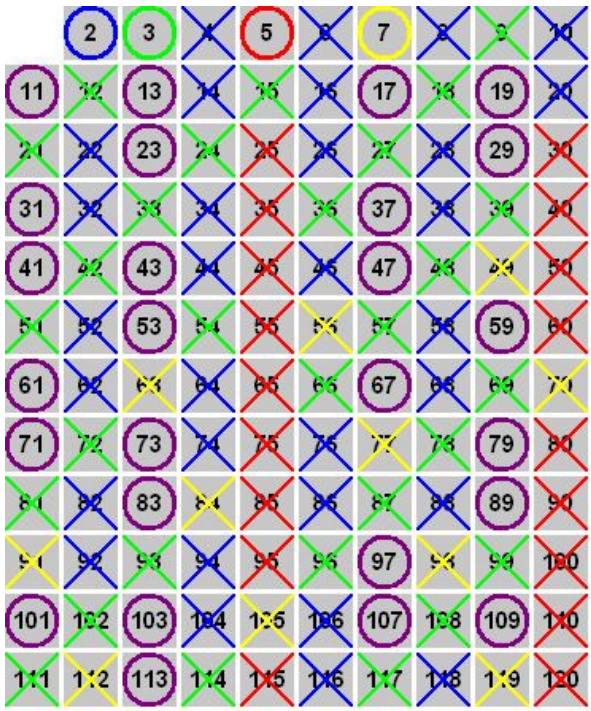
Algoritmos

Algoritmo *naïve* 'básico'

Um número primo, um número natural com 2 divisores distintos: o número 1 e ele mesmo. Para encontrar todos os números primos menores ou iguais ao número inteiro **n** dado pelo método de Eratosthenes:

- Cria-se uma lista de números inteiros consecutivos de 2 até **n**: (2, 3, 4 …, *n*)
- Inicia-se **k**=2, o menor número primo
- Enumera-se todos os múltiplos de **k** seguintes até **n** desde **2k** em incrementos de **k**, e marcá-los numa lista como números compostos (sendo estes **2k, 3k, 4k,** ....; **k** não é marcado)
- Encontrar o primeiro número maior que **k** na lista que ainda não foi marcado. Se não houver esse número, acaba algoritmo. Se houver, **k** será este novo número (que será o próximo primo), e repete-se o 3º passo.

Quando o algoritmo terminar, todos os números não marcados são números primos.
Tem como ordem espacial O(n) e ordem temporal O(n log log n).



Prime numbers

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	101	103	107
109	113		

Resultado final do algoritmo com n=120

Algoritmo por segmentos otimizado

Esta implementação do algoritmo, baseado no algoritmo anterior, altera a organização dos ciclos de cálculo dos números primos. Permite procurar várias sementes em simultâneo no mesmo intervalo de dados. A lista dos números até **n** é dividida em segmentos, sendo estes processados em paralelo. A implementação foi conseguida com ajuda das bibliotecas OpenMP e MPI.

Otimizações

- **Eliminação dos números pares:**
  Sabendo que todos os números pares, com excepção do número 2, são compostos, pode-se eliminar estes da lista a testar, conseguindo assim:
    - Reduzir a metade o número de cálculos;
    - Libertar memória para maior **n**

- **Raiz de n:**
  Sabendo que a maior semente múltipla de **n** será sempre a sua raiz quadrada, não é necessário utilizar sementes de número maior que $\sqrt{n}$. Com isto diminui-se o tempo de processamento substancialmente, limitando o número de cálculos realizados.

- **Fast Marking**
  Marcando apenas os múltiplos de **k**, ao contrário de testar a divisão do número a testar por **k**, salta-se o teste dos números não múltiplos de **k.**

- **Wheel factorization**
  Sabendo que números múltiplos dos primos 3, 5, 7, …, já foram previamente removidos, é possível saltar cálculos de múltiplos destes números, reduzindo um número significativo de cálculos desnecessários.
    - Na implementação realizada salta-se os cálculos dos números múltiplos de 3, 5, 7, 11, 13 e 17

- **Tamanho da Cache**
  Sabendo que o tamanho da cache L1 é 128KB, o tamanho dos segmentos analisados por cada *thread* terão este tamanho a fim de obter menor *cache miss* e melhor performance.

Implementação em OpenMP

Para aproveitar o poder de processamento em paralelo em computadores com vários *cores*, o algoritmo foi implementado da seguinte maneira utilizando a biblioteca OpenMP:

- Inicialização do contador de primos a 0
- Cálculo do número de primos de um segmento com tamanho 128*1024 (Tamano da *cache* L1*L2)
- Soma ao contador, com *reduction* para não existir conflitos entre cada *thread*, o resultado do cálculo e voltar ao passo anterior calculado o segmento seguinte com o limite inferior igual ao limite superior do segmento anterior +1, com o mesmo tamanho.
- Realizar o ciclo até o limite superior do último segmento ser igual ao limite **n**

Assim podemos calcular **c** blocos em paralelo simultaneamente (sendo **c** o número de *cores* do processador) e aproveitar a *cache* de cada um.

Implementação em MPI

O algoritmo implementado em MPI é uma adaptação do algoritmo anterior, com a vantagem de ser possível criar um *cluster* de computadores que dividem as tarefas pelos seus *cores*. Com isto, é possível um maior poder de processamento, ou seja, tempos de computação mais rápidos.
A implementação é muito semelhante ao anterior:

- **Thread 0 -** Cálculo do dos limites inferiores e superiores de cada segmento a ser enviado para as outras *threads*, começando depois do seu segmento inicial
- **Thread 0 -** Envio de mensagens para cada *thread* com o início e fim do seu segmento
- **Thread 0 -** Cálculo dos primos do seu segmento através do algoritmo por segmentos otimizado
- **Restantes threads -** Receber os limites superior e inferior do seu segmento e calcular os números primos do segmento através do algoritmo por segmentos otimizado
- **Restantes threads -** Envio da contagem para a **Thread 0**
- **Thread 0 -** Recolha dos resultados das restantes *threads* e somar à sua contagem

Implementação Híbrida em MPI com OpenMP

A mais potente das implementações pois combina os vários nós do *cluster* com a implementação por segmentos com OpenMP, a trabalhar para o mesmo problema. A implementação é baseada num híbrido das implementações anteriores:

- **Thread 0 -** Cálculo do dos limites inferiores e superiores de cada segmento a ser enviado para as outras *threads*, começando depois do seu segmento inicial
- **Thread 0 -** Envio de mensagens para cada *thread* com o início e fim do seu segmento
- **Thread 0 -** Cálculo dos primos do seu segmento em paralelo através da implementação OpenMP
- **Restantes threads -** Receber os limites superior e inferior do seu segmento e calcular os números primos do segmento em paralelo através da implementação OpenMP

- **Restantes threads -** Envio da contagem para a **Thread 0**
- **Thread 0 -** Recolha dos resultados das restantes *threads* e somar à sua contagem

Demonstração e Análise de Resultados

Testes realizados

Foram realizados testes, com limites de 10^6 a 2^{32}, em quatro categorias,:

- **Testes a execução sequencial:**
  Estes testes foram executados na implementação do algoritmo *naïve* básico, utilizando apenas um *core*, e na implementação em OpenMP utilizando apenas um core*;*

- **Testes à implmentação em OpenMP:**
  Estes testes foram executados na implementação em OpenMP, utilizando apenas um nó, com configurações diferentes de 2 a 8 *cores*, com finalidade de testar a sua escalabilidade*;*

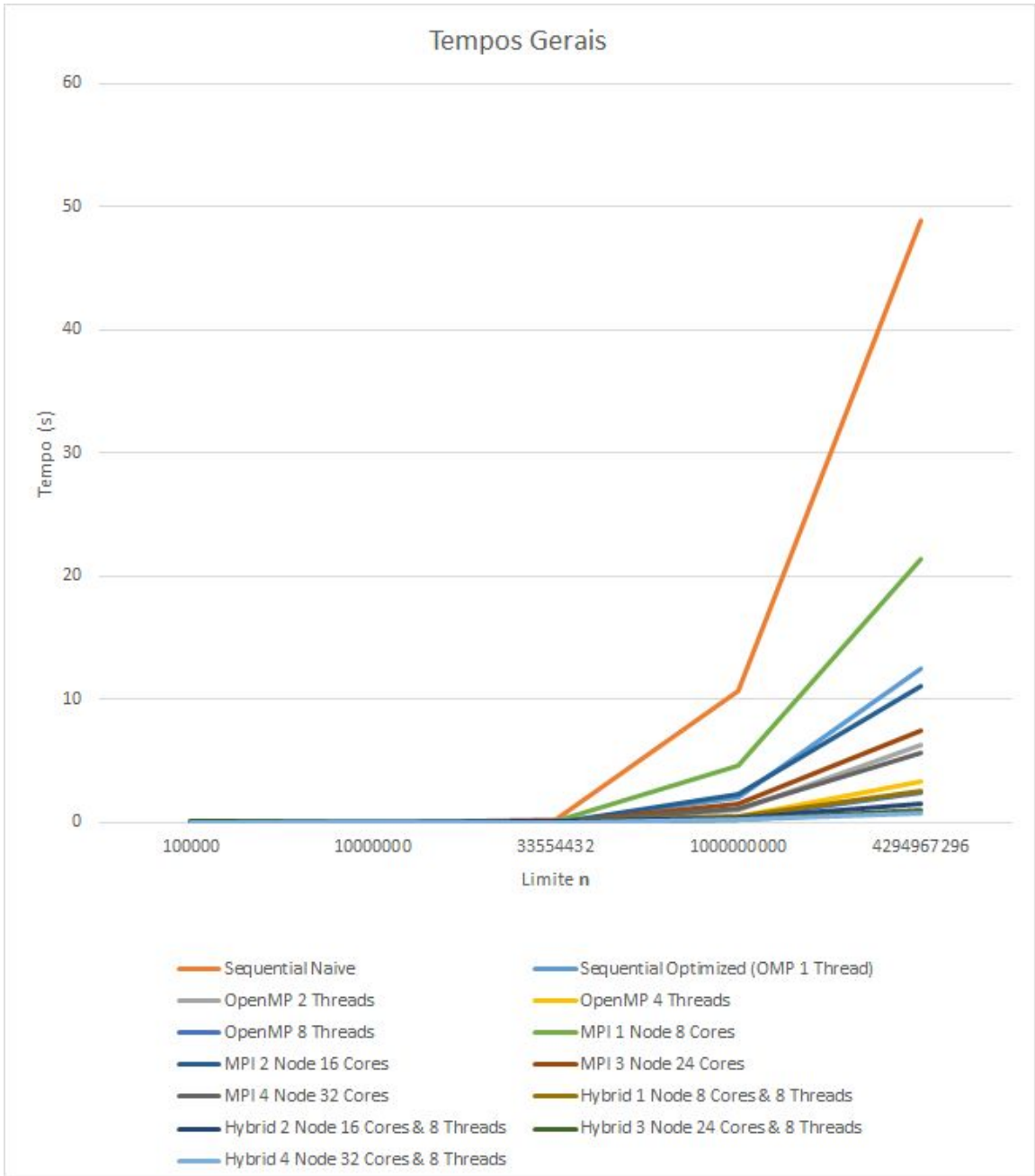
- **Testes à implementação em MPI:**
  Estes testes foram executados na implementação em MPI, utilizando 1 a 4 nós, sempre com 8 *cores,* com finalidade de testar a sua escalabilidade e se as comunicações no *cluster* tinha um *overhead* significativo*;*

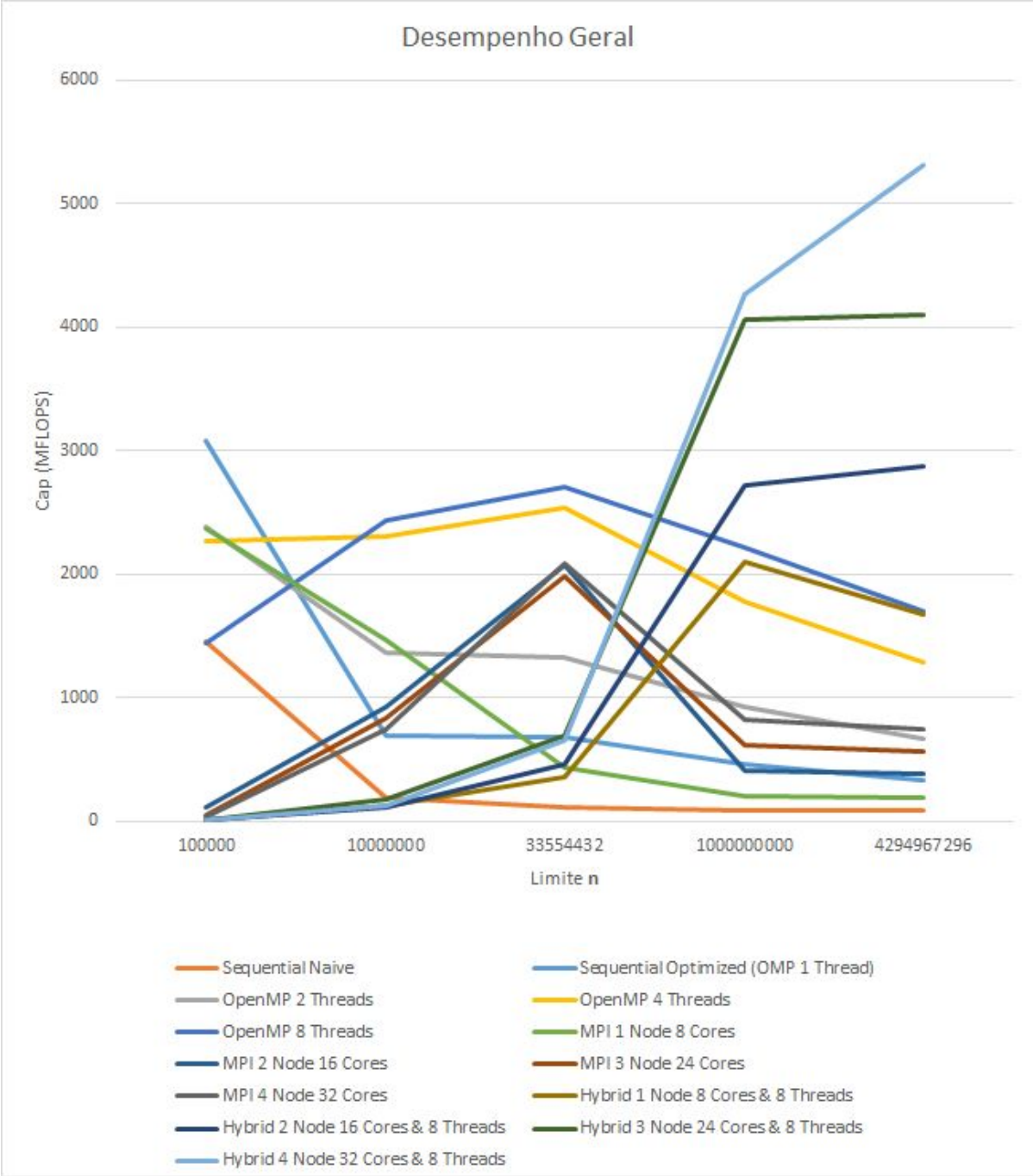
- **Testes à implementação híbrida em MPI com OpenMP:**
  Estes testes foram executados na implementação híbrida em MPI com OpenMP, utilizando 1 a 4 nós, sempre com 8 *cores* e 8 *threads,* com finalidade de testar a sua escalabilidade e se as comunicações no *cluster* tinha um *overhead* significativo*;*

Resultados

Análise dos Resultados

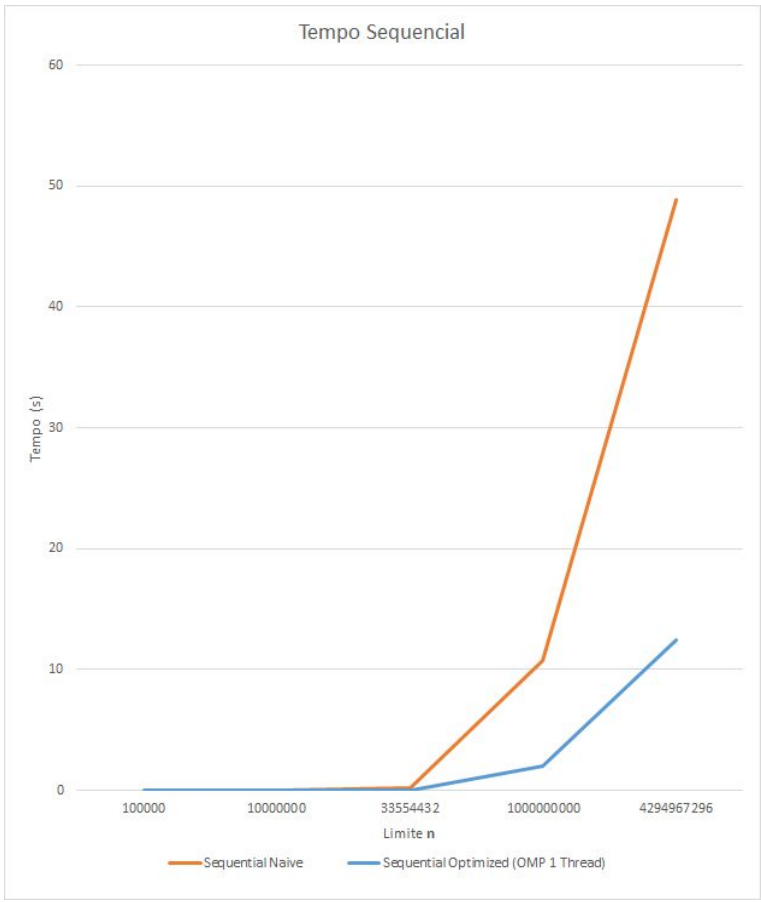
Esta análise foi feita considerando a performance dos algoritmos tendo em conta o tempo e execução destes, capacidade computacional (em MFLOPS) calculado através da fórmula $Cap = (n \log \log n) / t(FLOPS)$, e o *Speedup* ao acrescentar mais *cores* a correr o programa, calculado através da Lei de Amdahl: $Speedup = Tseq/Tnew$, sendo $Tnew$ o tempo de execução do algoritmo testado e $Tseq$ o tempo de execução do algoritmo *naïve* básico.

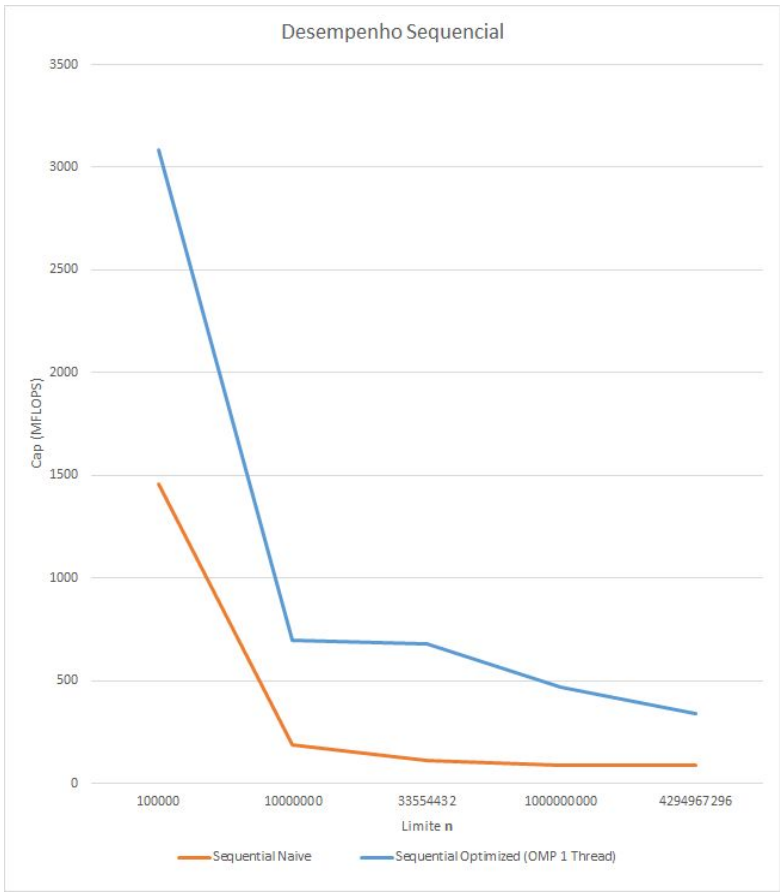




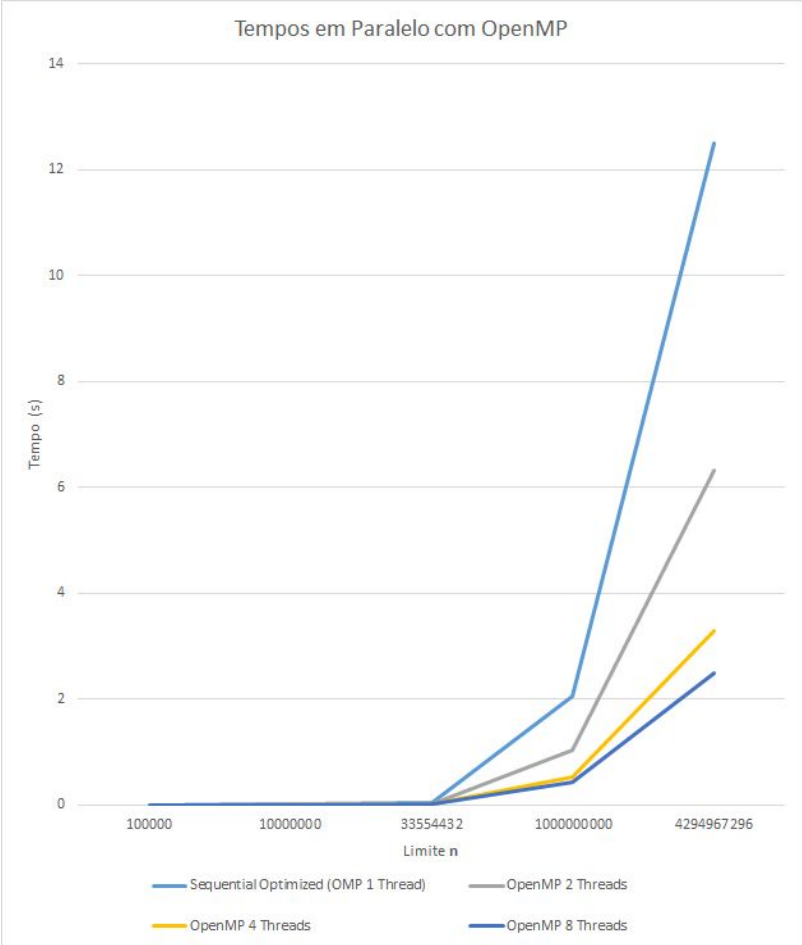
- **Execução sequencial:**

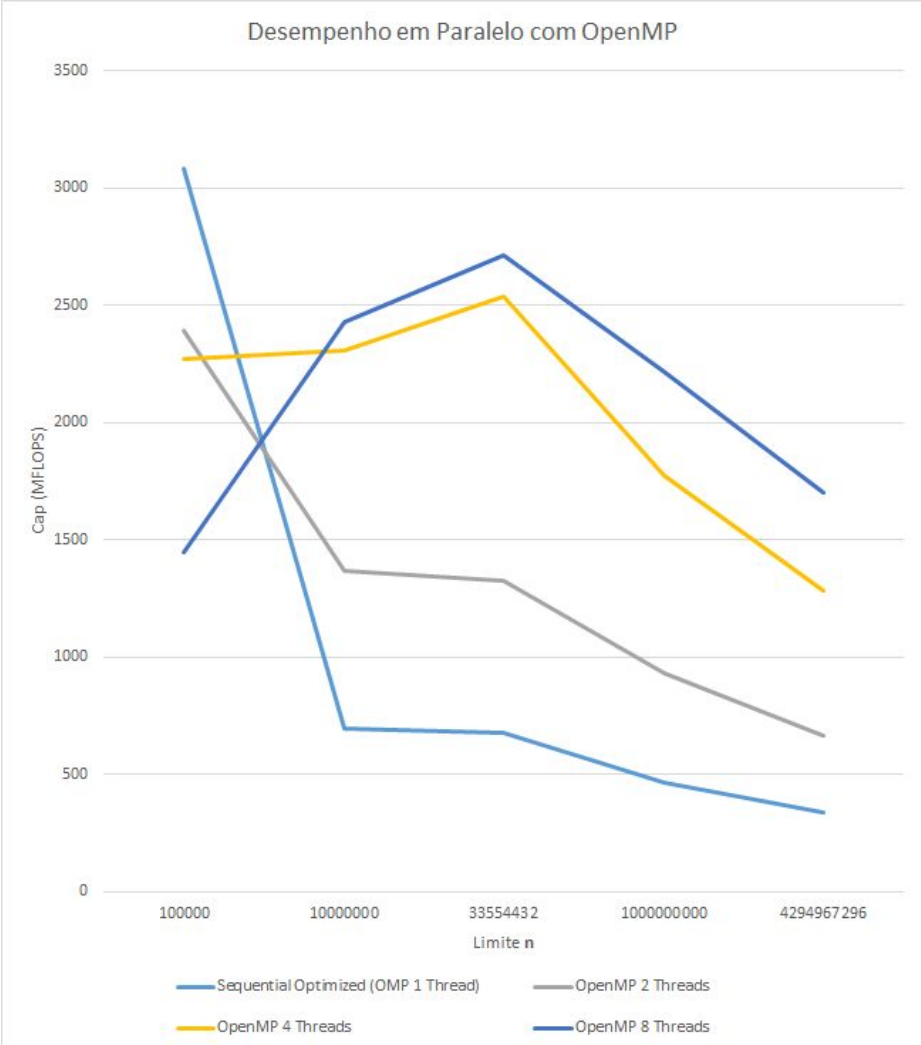
  Em termos de desempenho, o algoritmo otimizado demostra um grande aumento em relação ao algoritmo *naïve* básico. Pode-se observar através da diferença da Capacidade e *speedups* de 2.11, 3.75, 6.07, 5.24 e 3.9, respetivamente no processamento dos limites **n** de 10^6, 10^7, 2^{25}, 10^9 e 2^{33}.


Tempo Sequencial
60
50
40
30
20
10
0
Tempo (s)
100000
10000000
33554432
1000000000
4294967296
Limite n
Sequential Naive
Sequential Optimized (OMP 1 Thread)

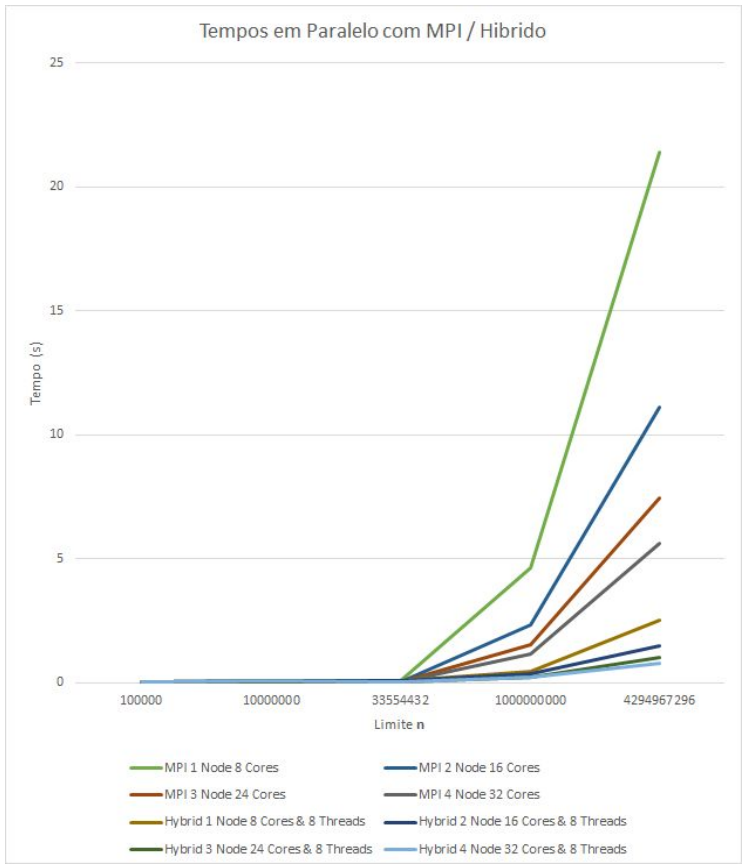

Desempenho Sequencial
3500
3000
2500
2000
1500
1000
500
0
Cap (MFLOPS)
100000
10000000
33554432
1000000000
4294967296
Limite n
Sequential Naive
Sequential Optimized (OMP 1 Thread)

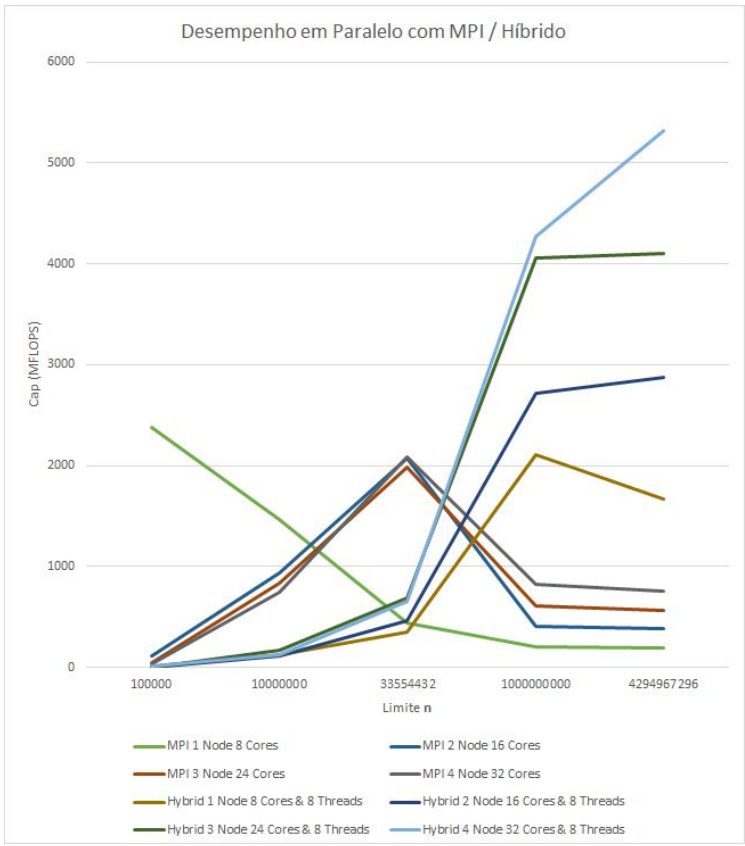
- **Execução em Paralelo com OpenMP:**
  Em termos de desempenho, a implementação OpenMP apenas compensa o *overhead* para **n** maior do que 100000, por para números inferiores os tempos são muito semelhantes. No entanto para os limites superiores são visíveis as melhorias de *performance* e *speedup* com o aumento de *threads* de processamento.





- **Execução em Paralelo com MPI e Híbrido:**
  Em termos de desempenho, a implementação com MPI apenas é notável uma degradação, resultado de um *overhead* significativo na comunicação, principalmente para **n** mais baixos.
  Quanto à implementação Híbrida nota-se a superioridade do desempenho a partir de **n** maiores do que 2^{25}, comparada às restantes implementações.
  Para reforçar o poder da computação paralela em relação ao algoritmo *naïve* básico, note-se o *speedup* da implementação híbrida em 4 nós e 32 *cores* é de 61.5.


Tempos em Paralelo com MPI / Hibrido
25
20
15
10
5
0
Tempo (s)
100000
10000000
33554432
1000000000
4294967296
Limite n
MPI 1 Node 8 Cores
MPI 2 Node 16 Cores
MPI 3 Node 24 Cores
MPI 4 Node 32 Cores
Hybrid 1 Node 8 Cores & 8 Threads
Hybrid 2 Node 16 Cores & 8 Threads
Hybrid 3 Node 24 Cores & 8 Threads
Hybrid 4 Node 32 Cores & 8 Threads


Desempenho em Paralelo com MPI / Híbrido
6000
5000
4000
3000
2000
1000
0
Cap (MFLOPS)
100000
10000000
33554432
1000000000
4294967296
Limite n
MPI 1 Node 8 Cores
MPI 2 Node 16 Cores
MPI 3 Node 24 Cores
MPI 4 Node 32 Cores
Hybrid 1 Node 8 Cores & 8 Threads
Hybrid 2 Node 16 Cores & 8 Threads
Hybrid 3 Node 24 Cores & 8 Threads
Hybrid 4 Node 32 Cores & 8 Threads

Observações

- Anexado a este documento está o código-fonte do programa utilizado para este estudo. Para execução dos testes a cada implementação apenas é necessário correr os *scripts* respectivo a cada. Estes *scripts* compilam o programa com otimizações e correm os testes (com os devidos incrementos, algoritmo, número de *threads* e *hostfiles* em caso da implementação com MPI;
- Anexado a este documento estão as folhas de cálculo com todos os dados relativos aos gráficos apresentados.

Conclusão

Analisando os resultados da aplicação de paralelismo ao Crivo de Eratosthenes, com as vantagens e desvantagens das diferentes implementações pode-se concluir que a aplicação de paralelismo pode resultar numa diminuição da performance. Isto acontece devido a problemas de sincronização das *threads* ou a *overheads* da comunicação entre os vários nós da rede. É no entanto bastante perceptível o ganho do paralelismo quando aplicado ao cálculo de um grande número de dados.
Durante o desenvolvimento do projeto, pude adquirir uma maior familiaridade sobre os conceitos de paralelismo computacional em *clusters* de processamento, e em especial, com as as ferramentas OpenMP e MPI.