Disciplina: Programação Paralela

Professor: George Luiz Medeiros Teodoro

Aluno: Jefferson Chaves Gomes – 14/0189581

**Exercício de Programação 01**

1. **Introdução**

Neste exercício foram comparados dados estatísticos no que se refere ao tempo de processamento de um programa quando este é executado de forma sequencial e paralela. Também foram avaliados os impactos causados no desempenho pelo tipo de escalonador e *chunkSize* utilizados no processamento. O paralelismo foi aplicado através do uso da API OpemMP. O programa gera uma lista ordenada de números primos de *2* até um inteiro *n* passado como parâmetro, onde *1 < n ≤ 1000000000* .

1. **Objetivos**

O objetivo geral deste exercício é aplicar os conhecimentos adquirídos em sala de aula a fim obter ganhos com uso da paralelização através do OpenMP. Os objetivos específicos são:

* Imprimir a lista de primos com um espaço após cada número e o tempo de execução em segundos com até seis casas decimais. O resultado será impresso na saída padrao (stdout) no seguinte formato, de acordo com a string *s* de entrada:
  + “time”: Uma linha contendo o tempo o tempo de execução do programa.
  + “list”: Uma linha contendo a lista de números primos.
  + “all”: A primeira linha contendo a lista de números primos e a segunda linha com o tempo de execução.
* Desenvolver um algoritmo que execute estas tarefas de forma sequencial.
* Aplicar o Paralelismo neste mesmo algoritmo utilizando OpenMP.
* Avaliar o desempenho na execução em cada cenário de implementação das aplicações desenvolvidas.

1. **Hardware de Execução**

Foi utilizado um desktop iMac com OS X Yosemite, processador 2,5 GHz Intel Core i5, 8 GB 1333 MHz DDR3,

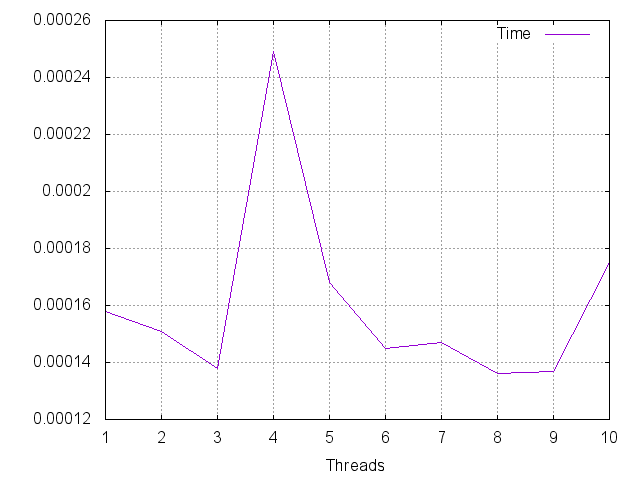
1. **Gráficos de avaliações**

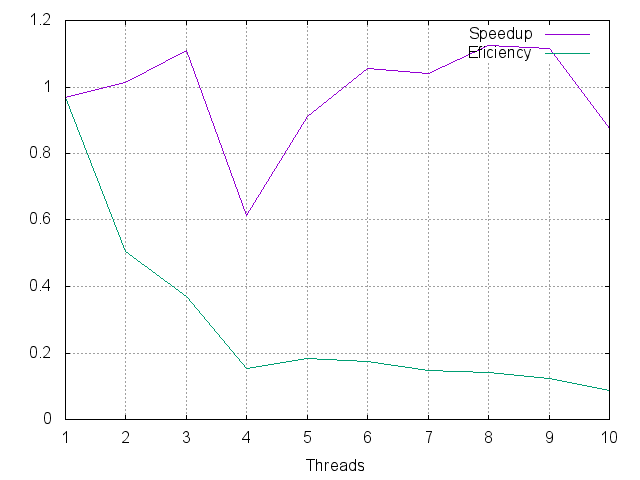
As avaliações exemplificadas a seguir mostram os resultados obtidos ao processar N números naturais tal que N = 1000, a fim de encontrar a lista de números primos onde 1 < primos <= N.

Foram usadas duas variações de *ChunkSize* com um determinado número de *Threads*, onde ao *ChunkSize* temos uma avaliação para 20% de N e outra para 80% de N, ambas variando o número de *Threads* de 1 até 10.

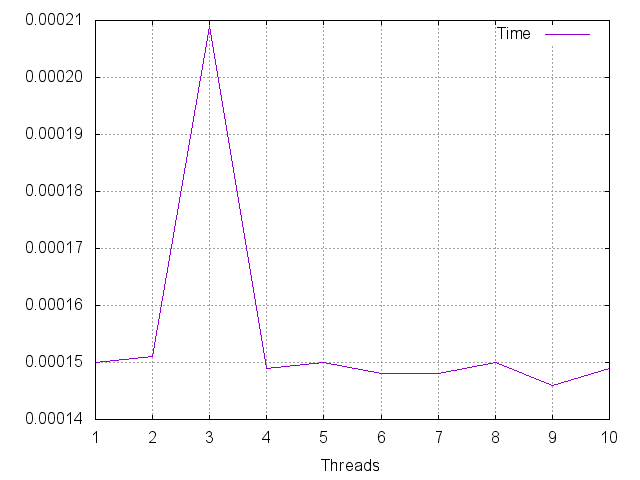
Os gráficos dos resultados estão separados por tipo de escalonamento, o desempenho e o *speedup/efficiency* para cada um dos dois valores utilizados para o *ChunkSize.*

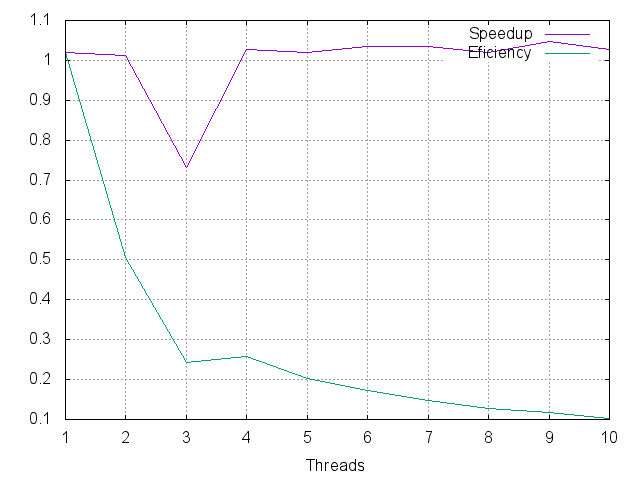
*Static Schedule - ChunkSize* = 80% de N



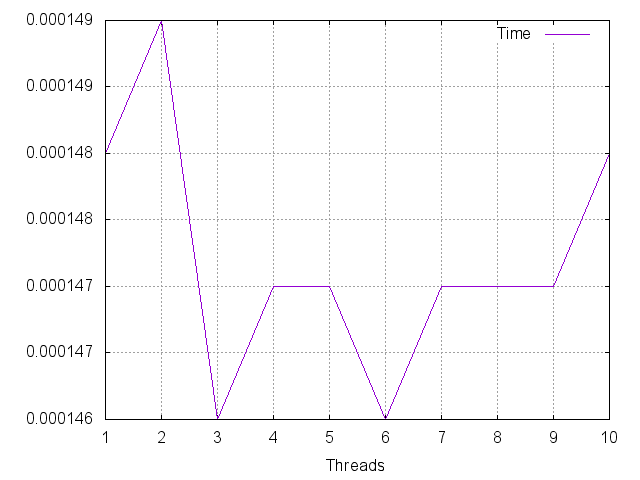


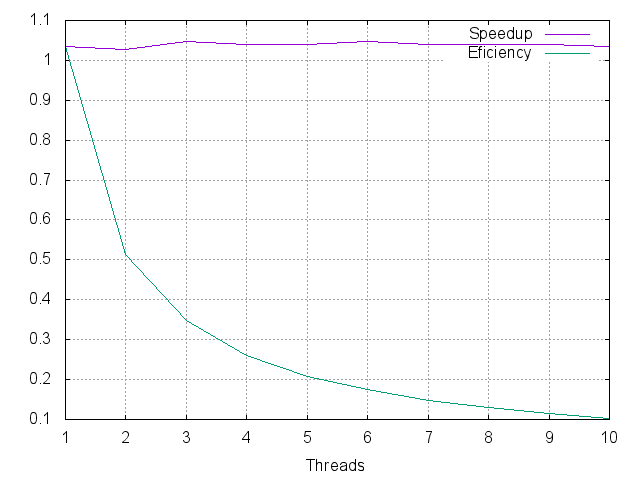
*Static Schedule - ChunkSize* = 20% de N



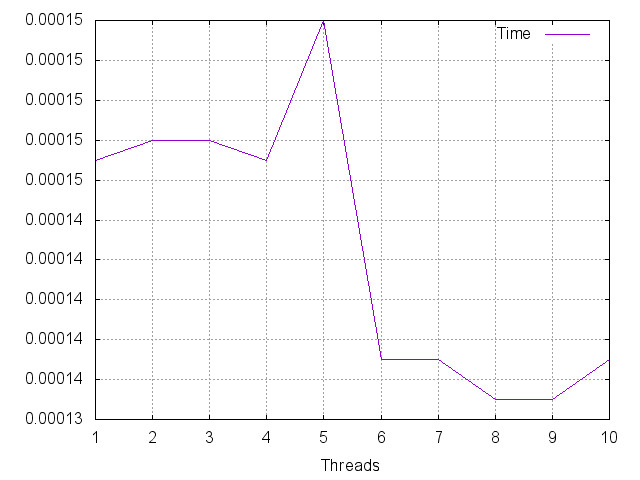


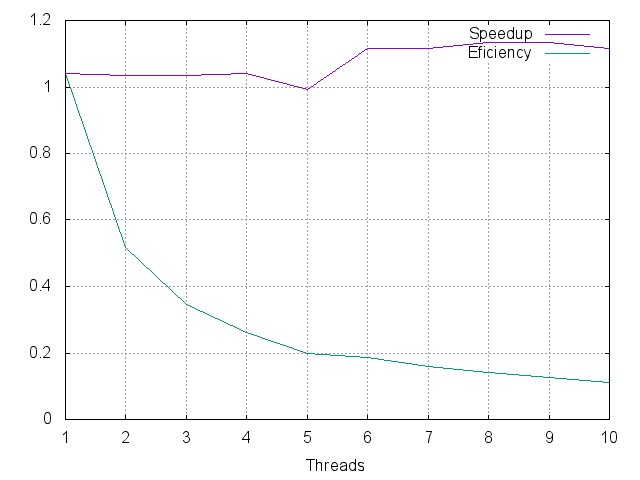
*Dynamic Schedule - ChunkSize* = 80% de N



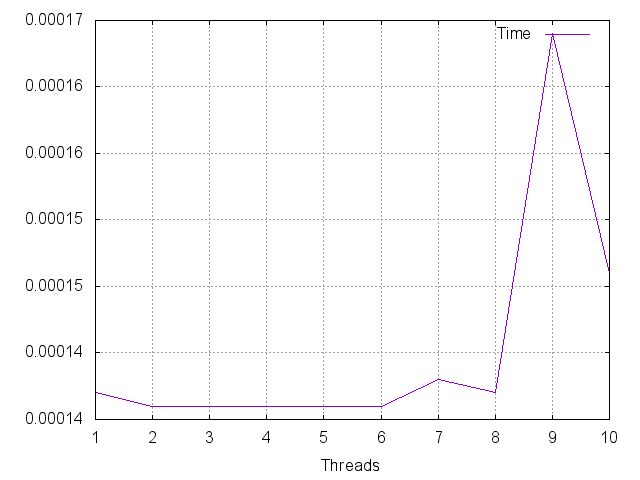


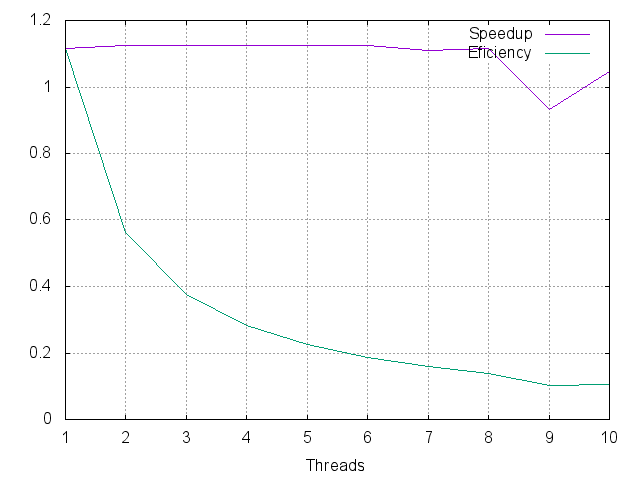
*Dynamic Schedule - ChunkSize* = 20% de N



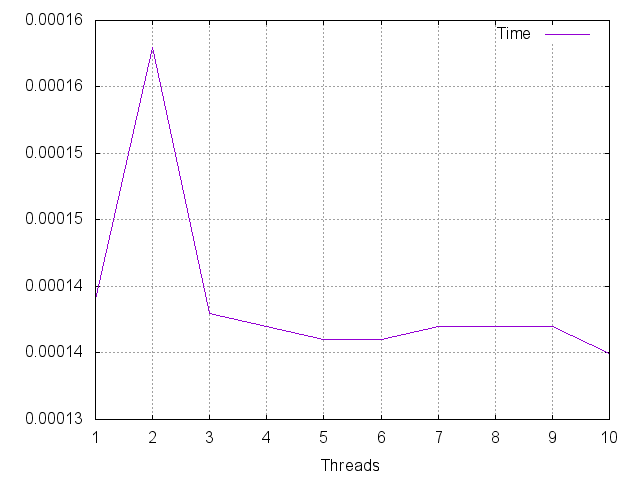


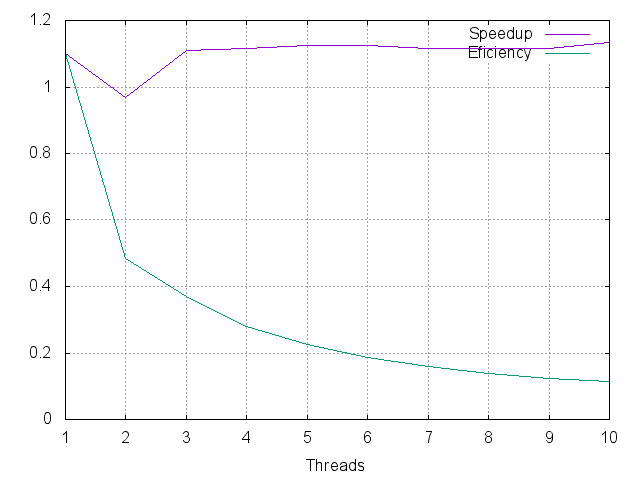
*Guided Schedule - ChunkSize* = 80% de N





*Guided Schedule - ChunkSize* = 20% de N





1. **Algoritmo Desenvolvido**

O algoritmo desenvolvido é bastante simples e os trechos de maior relevância do mesmo são os que seguem:



1. **Conclusão**

Após as avaliações de impacto no desempenho conforme o tipo de escalonador, *ChunKSize* e número de *Threads* é possível perceber que a paralelização trás ganhos significativos para o tempo de execução de um programa. Em relação ao número de *Threads* é sensível para o cenário apresentado afirmar que sempre será possível obter melhores resultados com um maior número de *Threads*, visto que, a este mesmo cenário há momentos em que um número menor de *Threads* se mostra mais eficiente. Já para o *ChunkSize* é notório o grande impacto que o mesmo representa ao desempenho do programa, sendo que o tempo de execução muitas das vezes aparentar ocorrer de forma mais linear quando temos *ChunKSizes* menores. O melhor resultado obtido em termos de velocidade de execução foi adquirido fazendo o uso do tipo de escalonador *Dynamic* com *ChunkSize* igual a 20% do valor iterado para buscar os números primos.