Disciplina: Programação Paralela, código 316342

Professor: George Luiz Medeiros Teodoro

Aluno: Jefferson Chaves Gomes – 14/0189581

**Exercício de Programação 02**

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 para calcular a área de polígonos irregulares a partir das coordenadas de seus vértices fazendo o uso do teorema de *Shoelace* dado pela fórmula a seguir:



Onde *A* representa a área do polígono, *n* o número de lados no polígono e (*xi, yi* ) representa o vértice *i* do polígono (com *i* variando de *1 . . . n*).

1. **Requisitos à implementação**

À implementação deste exercício os itens abaixo foram dados como requisitos:

* A implementação deve fazer uso exclusivamente de C ou C++;
* O paralelismo de ser aplicado através do uso de *PThreads*;
* Os vértices do Polígono devem estar ordenados no sentido horário;
* O polígono deverá ser fechado e o último vértice deve ser igual ao primeiro;
* O polígono não deve atravessar, sobrepor ou interceder (gerar área negativa por sobreposição) consigo mesmo.

1. **Objetivos**

O objetivo geral deste exercício é aplicar os conhecimentos adquiridos em sala de aula a fim obter ganhos de performance com uso da paralelização através do uso de *PThreads,* gerando com a execução do programa dados necessários para mensurar os ganhos de performance, *speedup* e eficiência.

1. **Descrição do Algoritmo Desenvolvido**

Para a implementação do teorema de *Shoelace* foi produzida a seguinte função chamada *shoelaceTheorem*:

|  |
| --- |
| **void** \*shoelaceTheorem(**void** \*arg0) {  // Cast the received parameter.  RlThreadRangeVertices \*pThreadArgs = (RlThreadRangeVertices \*) arg0;  **long** **double** sum1 = 0.0;  **long** **double** sum2 = 0.0;  **for** (**long** i = pThreadArgs->getBeginIndex(); i <= pThreadArgs->getEndIndex(); i++) {  sum1 += vecVertices[i].getX() \* vecVertices[i + 1].getY();  sum2 += vecVertices[i + 1].getX() \* vecVertices[i].getY();  }  pThreadArgs->setArea(sum1 - sum2);  pthread\_exit(arg0);  } |

Esta função é chamada por cada Thread criada para executar o processamento dos dados podendo variar de 1 até *INT\_MAX(2147483647)*.

A função tem como objetivo realizar o somatório dos vértices do polígono, informados como entrada do programa.

Uma classe chamada *RLThreadRangeVertices* foi criada para armazenar os argumentos necessários ao processamentos dos dados e é passada como parâmetro para a função *shoelaceTheorem*. As informações armazenadas nesta classe são as que seguem:

* **long pThreadId**: atributo responsável por armazenar o ID da thread em questão;
* **long beginIndex**;
* **long endIndex**;
* **Result result**: ;

1. **Hardware de Execução**

|  |  |
| --- | --- |
| Nome do Modelo: | iMac |
| Identificador do Modelo: | iMac12,1 |
| Nome do Processador: | Intel Core i5 |
| Velocidade do Processador: | 2,5 GHz |
| Número de Processadores: | 1 |
| Número Total de Núcleos: | 4 |
| Cache L2 (por Núcleo): | 256 KB |
| Cache de L3: | 6 MB |
| Memória: | 10 GB 1333 MHz DDR3 |

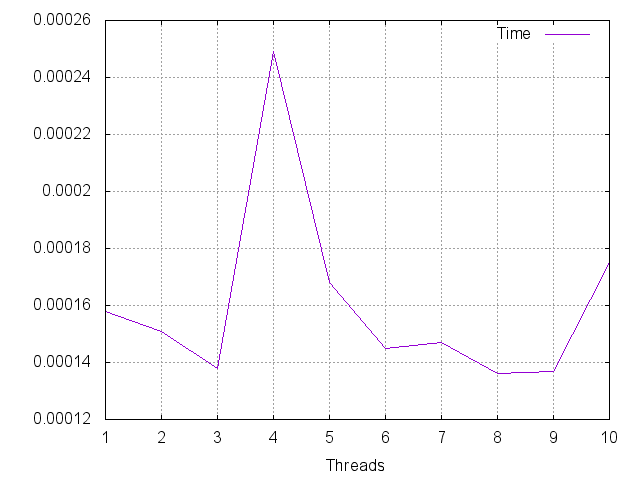
1. **Gráficos de avaliações**

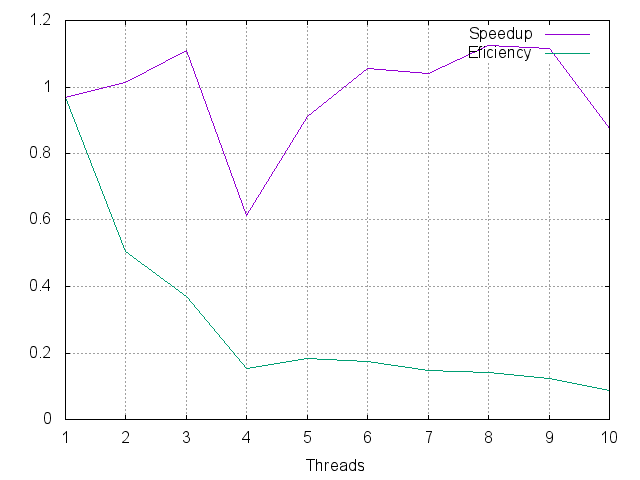
As avaliações exemplificadas a seguir mostram os resultados obtidos ao processar N números naturais tal que N = 1000, a fim de encontrar a lista de números primos onde 1 < primos <= N.

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.

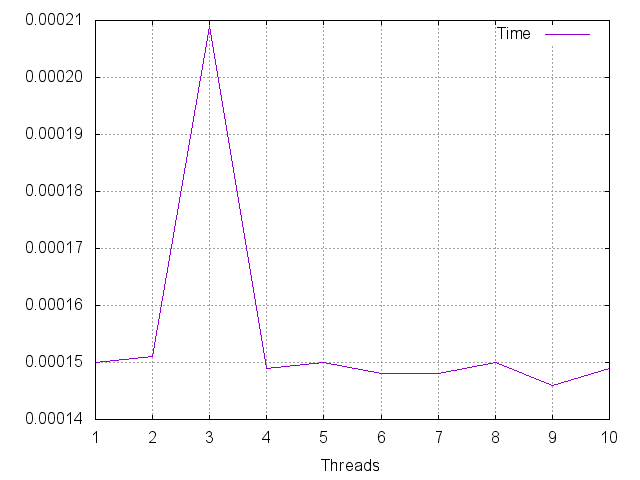
Os gráficos dos resultados estão separados por tipo de escalonamento, o desempenho e o *speedup/efficiency* para cada um dos dois valores utilizados para o *ChunkSize.*

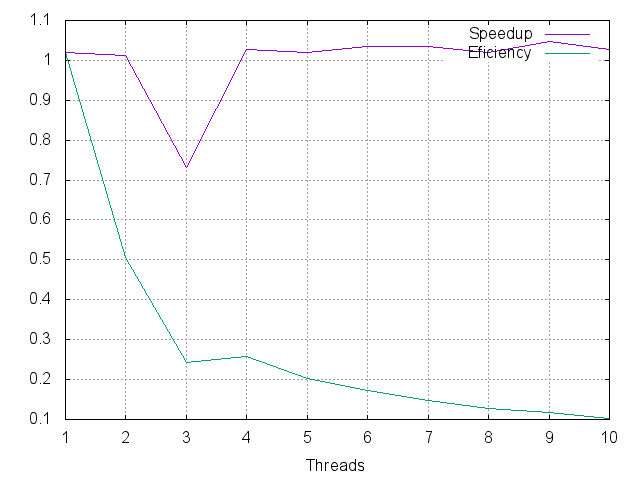
*Static Schedule - ChunkSize* = 80% de N



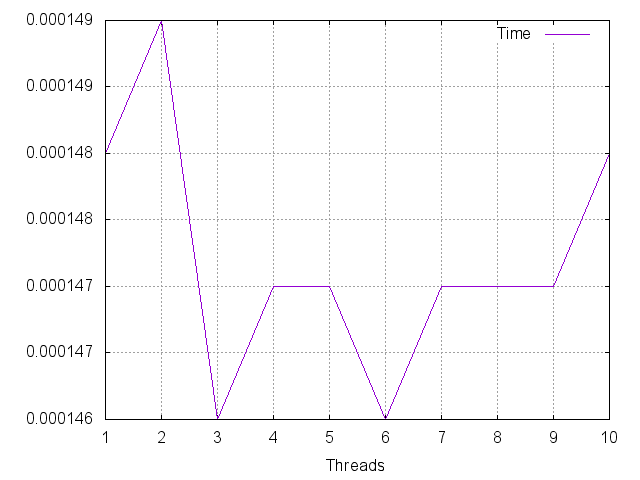


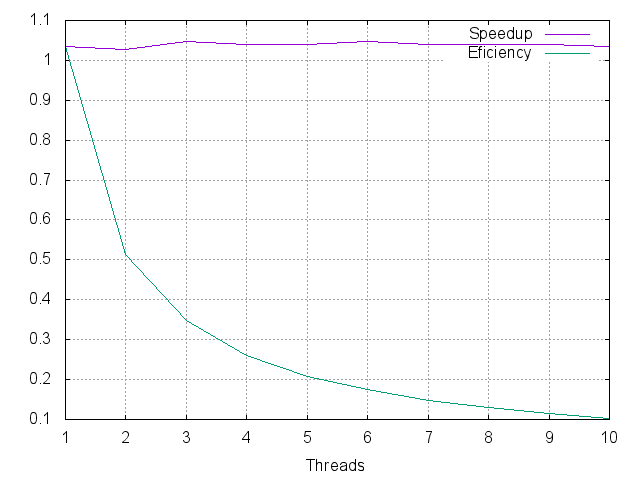
*Static Schedule - ChunkSize* = 20% de N



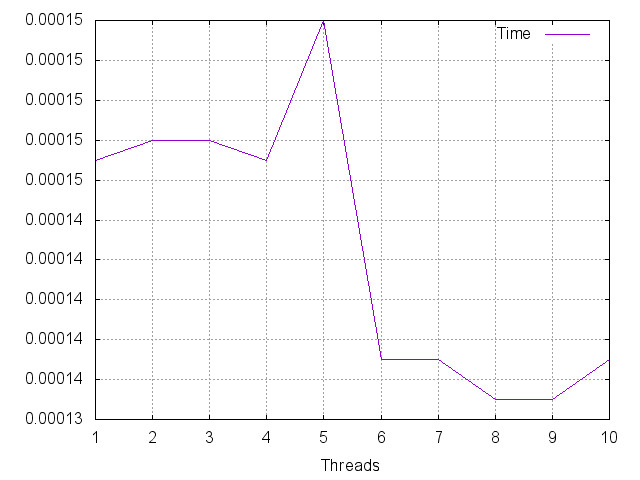


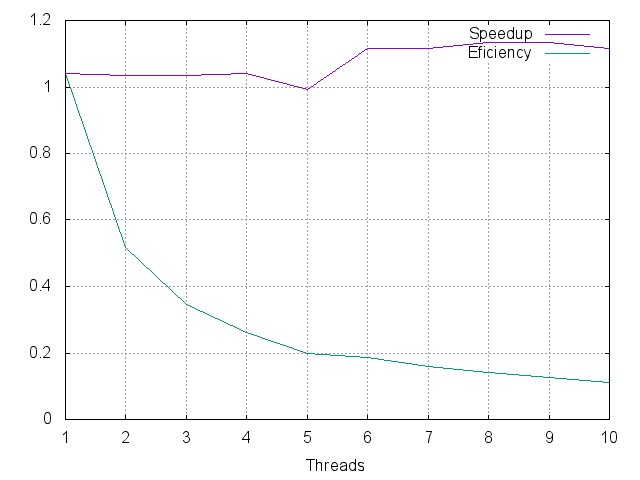
*Dynamic Schedule - ChunkSize* = 80% de N



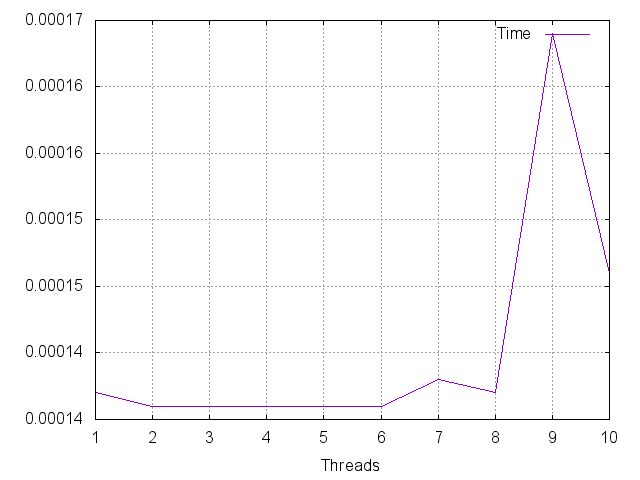


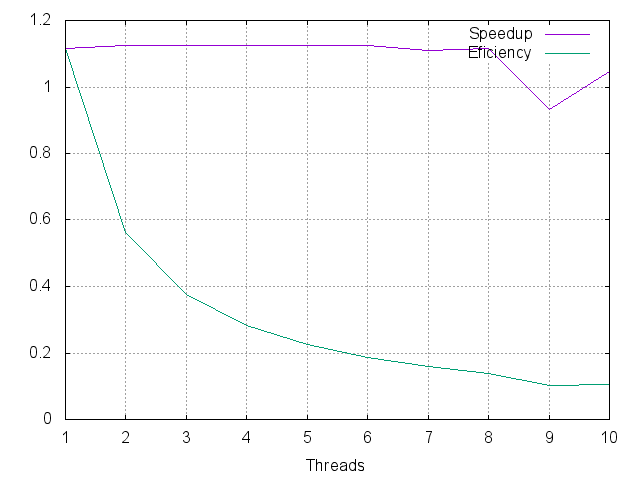
*Dynamic Schedule - ChunkSize* = 20% de N



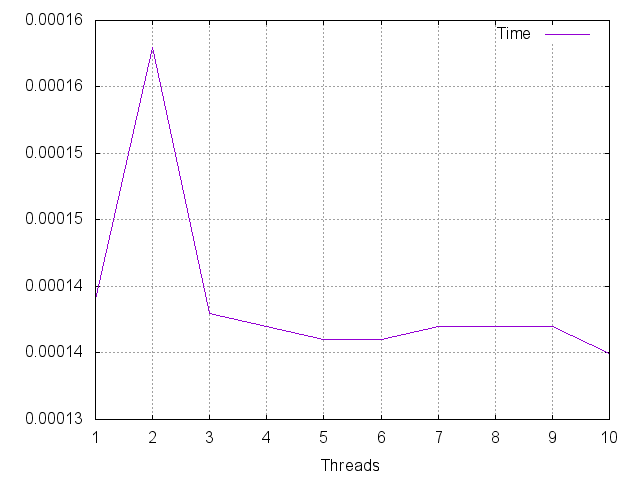


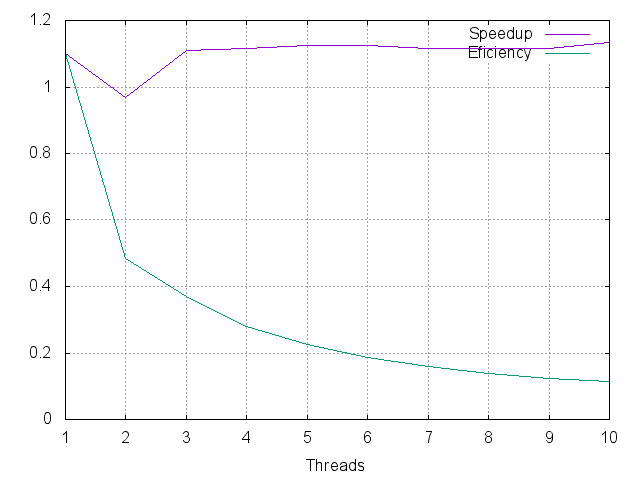
*Guided Schedule - ChunkSize* = 80% de N





*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 senário há momentos em que um número menos *Threads* se mostra mais eficiente. Já para o *ChunkSize* é notório o grande impacto que o mesmo representa ao desempenho do programa, sendo que a 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 execucão foi adquirido fazendo o uso do tipo de escalonador *Dynamic* com *ChunkSize* igual a 20% do valor iterado para buscar os números primos.