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. **Entrada esperada**

O programa deverá receber, em sua chamada, dois parâmetros como entrada, sendo o primeiro, um texto que representara o tipo de saída que deve ser impressa, são elas:

* **time**: para exibir somente o tempo (microsegundos) necessário ao processamento;
* **area**: para que o programa exiba como saída apenas o valor da área do polígono calculado, com duas casas decimais;
* **all**: para imprimir em duas linhas os valores de tempo e área respectivamente.

E o segundo, o número de threads desejados para executar o processo.

Após a chamada do programa, será necessário informar, no início da execução, um inteiro *n* com a quantidade de vértices do polígono bem como seus *n* vértices *x* e *y*.

1. **Descrição do Algoritmo Desenvolvido**

De uma forma geral será descrito como a solução foi desenvolvida, apontando pontos importantes a fim de esclarecer detalhes do funcionamento do programa.

* 1. **Estruturas e classes**

Foram desenvolvidas uma estrutura e duas classes visando facilitar a manipulação das informações necessárias ao processamento dos dados.

A estrutura foi chamada de *Result* e tem a função de armazenar o tempo gasto no processamento bem como o resultado obtido no cálculo da área do polígono informado. Segue o trecho do código referente a estrutura citada:

|  |
| --- |
| **struct** Result {  **long** processTime;  **long** **double** area;  }; |

Já a primeira classe criada foi chamada de *Vertice* e tem como objetivo armazenar as informações referentes as coordenadas de vértices ( *xi, yi* )*.* Segue o trecho do código referente a classe mencionada:

|  |
| --- |
| **class** Vertice {  **private**:  **double** x;  **double** y;  **public**:  **Vertice**(**double** arg0, **double** arg1) :  x(arg0), y(arg1) {  // Nothing to do.  }  **double** **getX**() **const** {  **return** x;  }  **double** **getY**() **const** {  **return** y;  }  }; |

A segunda classe criada foi chamada de *RlThreadRangeVertices* e tem como objetivo relacionar informações que correspondem ao:

[ identificador da *Thread* Responsável]

X

[ intervalo de vértices a ser processado pela *Thread* ]

X

[ resultado obtido no processamento do intervalo de vértices pela *Thread* ]

Segue o trecho do código referente a classe mencionada:

|  |
| --- |
| **class** RlThreadRangeVertices {  **private**:  **long** pThreadId;  **long** beginIndex;  **long** endIndex;  Result result;  **public**:  **RlThreadRangeVertices**(**long** arg0, **long** arg1, **long** arg2, Result arg3) :  pThreadId(arg0), beginIndex(arg1), endIndex(arg2), result(arg3) {  // Nothing to do.  }  **long** **getThreadId**() **const** {  **return** pThreadId;  }  **long** **getBeginIndex**() **const** {  **return** beginIndex;  }  **long** **getEndIndex**() **const** {  **return** endIndex;  }  **const** Result& **getResult**() **const** {  **return** result;  }  **void** **setArea**(**const** **long** **double** area) {  **this**->result.area = area;  }  }; |

* 1. **Leitura e validação de dados de entrada**

Uma função responsável pela validação e leitura de parâmetros de entrada foi criada para receber o tipo de saída esperada pelo usuário e o número de *Threads* desejadas para executar o processamento, este número pode variar de 1 até *INT\_MAX(2147483647)*. Esta função foi chamadade *readInputParams* e é exemplificada a seguir:

|  |
| --- |
| **bool** **readInputParams**(**int** argc, **char** \*\*argv, OutputType &outputType, **int** &numThreads) {  **if** (argc != 3) {  printUsage();  **return** **false**;  } **else** {  std::string input(argv[1]);  numThreads = **atoi**(argv[2]);  OutputType outputTypel = stringToOutputType(input);  **if** (numThreads <= 0 || numThreads > INT\_MAX || outputTypel == *NONE*) {  printUsage();  **return** **false**;  }  outputType = outputTypel;  }  **return** **true**;  } |

Para ler e armazenar os vértices do polígono em um vetor declarado de forma global, foi criada uma função chamada *readVertices*, seu código é ilustrada abaixo:

|  |
| --- |
| **void** **readVertices**(**const** **long** numVertices, std::vector<Vertice> &vecVertices) {  **double** x = 0.0;  **double** y = 0.0;  **for** (**long** i = 0; i < numVertices; i++) {  std::cin >> x;  std::cin >> y;  vecVertices.push\_back(Vertice(x, y));  }  } |

* 1. **Estruturação de dados de entrada**

Após a leitura de todos os dados, a solução estrutura os dados a fim gerar um escalonamento de laços de *schedule* *static*, ou seja, as iterações são atribuídas às threads antes que o seu laço seja executado. Para isto é definido um *chunksize* de tamanho = número de vertices / número de *Threads.* Com a informação de *chunksize,* a função preenche um vetor de relacionamentos recebido como parâmetro onde cada *Thread* será associada ao seu *chunksize*. A seguir, é exibido o código que representa a função, *buildRelations*, responsável por estruturar os dados conforme descrito acima:

|  |
| --- |
| **Void** **buildRelations**(  **long** numVertices,  **int** numThreads,  std::vector<RlThreadRangeVertices>& vecRelations) {  **long** chunkSize = std::**ceil**((**double**) (numVertices) / numThreads);  **long** beginIndex = 0;  **long** endIndex = chunkSize - 1;  **for** (**long** i = 0; i < numThreads; i++) {  Result result;  vecRelations.push\_back(RlThreadRangeVertices(i, beginIndex, endIndex, result));  **if** (endIndex + 1 == numVertices) {  **break**;  }  beginIndex = endIndex + 1;  endIndex = beginIndex + chunkSize - 1;  **if** (numVertices <= endIndex) {  endIndex = numVertices - 1;  }  }  } |

* 1. **Processando os dados**

Finalmente com os dados estruturado, a solução inicia o processamento sequencial ou paralelo, dependendo do número de *Threads* informado, do teorema de *Shoelace*. Para isto uma primeira iteração é executada a fim de criar as *Threads* necessárias, informando via passagem de parâmetros qual é o intervalo dentro do vetor de vértices que cada *Thread* deve processar. Após isto, uma segunda iteração é executada para parar todas as *Threads* criadas anteriormente e computar o resultado da soma calculado por cada uma das mesmas. Feita a iteração a função ainda aplica no resultado computado uma divisão por 2 e recuperar seu valor absoluto.

Para avaliar o tempo de execução necessário ao processamento das *Threads* foi utilizada função *gettimeofday()* exatamente conforme sugerido no exercício EP2. A função, startParallelProcess, que inicia o processamento do teorema de *Shoelace* é exibida a seguir:

|  |
| --- |
| Result **startParallelProcess**(**const** std::vector<RlThreadRangeVertices>& vecRelations) {  Result result;  timeval startTime;  timeval endTime;  **gettimeofday**(&startTime, NULL);  pthread\_t\* arrayThreads = **new** pthread\_t[vecRelations.size()];  **for** (**unsigned** **int** i = 0; i < vecRelations.size(); i++) {  **pthread\_create**(&arrayThreads[i], NULL, shoelaceTheorem, (**void** \*) &vecRelations[i]);  }  **long** **double** sum = 0.0;  **for** (**unsigned** **int** i = 0; i < vecRelations.size(); i++) {  **pthread\_join**(arrayThreads[i], NULL);  sum += vecRelations[i].getResult().area;  }  result.area = std::abs((sum) / 2.0);  **gettimeofday**(&endTime, NULL);  **long** time = ((endTime.tv\_sec \* 1000000 + endTime.tv\_usec) - (startTime.tv\_sec \* 1000000 + startTime.tv\_usec));  result.processTime = time;  **return** result;  } |

Por fim uma função chamada *shoelaceTheorem* foi criada com o objetivo de realizar o somatório dos vértices do polígono. Esta função é chamada por cada *Thread* criada anteriormente e o número de iterações executado por cada *Thread* vai depender do *chunksize* também defino anteriormente.

|  |
| --- |
| **void** \***shoelaceTheorem**(**void** \*arg0) {  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);  } |

1. **Instruções de Compilação**

g++ -lpthread

1. **Instruções de Execução**

Modo de uso:

./pp-ep02-012015 [output type] [number of threads]

output type -- Define the output type [time | area | all]

number of threads -- Define the number of threads to use in processing

Sample:

./pp-ep02-012015 all 4 < exemplo.txt

1. **Ambiente de Execução dos Testes**

|  |  |
| --- | --- |
| 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. **Descrição dos testes**

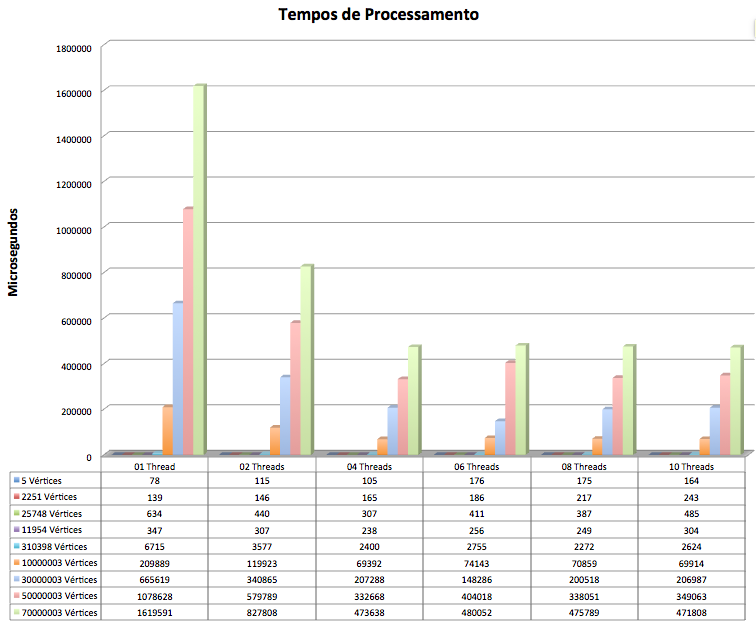
Os testes foram executados 3 vezes utilizando 1, 2, 4, 6, 8 e 10 *Thread(s)* respectivamente, onde para cada um dos testes aplicados foram passados como parâmetros os arquivos listados na tabela abaixo contendo dados de diferentes tamanhos de polígonos irregulares.

A coluna “Área” apresenta o resultado da área de cada polígono calculado pelo programa desenvolvido.

|  |  |  |  |
| --- | --- | --- | --- |
| **Quantidade de Vértices** | **Área** | **Link para baixar** | **Tamanho** |
| 5 | 64.00 | [Exemplo 00](http://164.41.209.62/exemplo-00.txt) | 22 |
| 2251 | 21999.00 | [Exemplo 01](http://164.41.209.62/exemplo-01.txt) | 22K |
| 25748 | 1725880.50 | [Exemplo 02](http://164.41.209.62/exemplo-02.txt) | 343K |
| 11954 | 7668714.50 | [Exemplo 03](http://164.41.209.62/exemplo-03.txt) | 124K |
| 310398 | 328170058.95 | [Exemplo 04](http://164.41.209.62/exemplo-04.txt) | 4,1M |
| 10000003 | 16495624.27 | [in-01.txt](http://164.41.209.62/in-01.txt) | 160M |
| 30000003 | 2039436.39 | [in-02.txt](http://164.41.209.62/in-02.txt) | 371M |
| 50000003 | 7398337.15 | [in-03.txt](http://164.41.209.62/in-03.txt) | 616M |
| 70000003 | 1749621.16 | [in-04.txt](http://164.41.209.62/in-04.txt) | 876M |

1. **Tempo de Execução**

Os resultados referente a performance após a execução dos testes são ilustrados a seguir:



1. **Speedup X Eficiência**

Já a primeira classe criada foi chamada de *Vertice* e tem como objetivo armazenar as informações referentes as coordenadas de vértices ( *xi, yi* )*.* Segue o trecho do código referente a classe mencionada

1. **Análise de Resultados**

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.

1. **Código Completo**

|  |
| --- |
| /\*  ============================================================================  Name : main.cpp  Author : Jefferson Chaves Gomes  Version : 1.0.0  Copyright : Academic Program  Description : EP 02 in C++  ============================================================================  \*/  // Libraries definitions  // -----------------------------------------------------  **#include** <iostream> // for std::cin, std::cout, etc.  **#include** <vector> // for vector  **#include** <sys/time.h> // for gettimeofday  **#include** <cmath> // for std::abs, std::ceil  **#include** <iomanip> // for std::setprecision  **#include** <pthread.h> // for pthreads  **#include** <stdlib.h> // for std:atoi  **#include** <climits> // for INT\_MAX  // Enums definitions  // -----------------------------------------------------  **enum** OutputType {  *NONE* = 0,  *TIME* = 1,  *AREA* = 2,  *ALL* = 3  };  // Structs definitions  // -----------------------------------------------------  **struct** Result {  **long** processTime;  **long** **double** area;  };  // Classes definitions  // -----------------------------------------------------  **class** Vertice {  **private**:  **double** x;  **double** y;  **public**:  **Vertice**(**double** arg0, **double** arg1) :  x(arg0), y(arg1) {  // Nothing to do.  }  **double** **getX**() **const** {  **return** x;  }  **double** **getY**() **const** {  **return** y;  }  };  **class** RlThreadRangeVertices {  **private**:  **long** pThreadId;  **long** beginIndex;  **long** endIndex;  Result result;  **public**:  **RlThreadRangeVertices**(**long** arg0, **long** arg1, **long** arg2, Result arg3) :  pThreadId(arg0), beginIndex(arg1), endIndex(arg2), result(arg3) {  // Nothing to do.  }  **long** **getThreadId**() **const** {  **return** pThreadId;  }  **long** **getBeginIndex**() **const** {  **return** beginIndex;  }  **long** **getEndIndex**() **const** {  **return** endIndex;  }  **const** Result& **getResult**() **const** {  **return** result;  }  **void** **setArea**(**const** **long** **double** area) {  **this**->result.area = area;  }  };  // Functions declarations  // -----------------------------------------------------  **bool** **readInputParams**(**int**, **char**\*\*, OutputType&, **int**&);  **void** **printUsage**();  OutputType **stringToOutputType**(std::string);  **void** **readVertices**(**long**, std::vector<Vertice>&);  **void** **buildRelations**(**long**, **int**, std::vector<RlThreadRangeVertices>&);  Result **startParallelProcess**(**const** std::vector<RlThreadRangeVertices>&);  **void** \***shoelaceTheorem**(**void**\*);  **void** **printResult**(OutputType, Result);  // Globals declarations  // -----------------------------------------------------  std::vector<Vertice> vecVertices;  // main Function  // -----------------------------------------------------  **int** **main**(**int** argc, **char** \*\*argv) {  **int** numThreads = 0;  OutputType outputType = *NONE*;  **if** (!readInputParams(argc, argv, outputType, numThreads)) {  **return** EXIT\_SUCCESS;  }  **long** numVertices = 0;  std::cin >> numVertices;  readVertices(numVertices, vecVertices);  std::vector<RlThreadRangeVertices> vecRelations;  buildRelations(numVertices, numThreads, vecRelations);  Result result = startParallelProcess(vecRelations);  printResult(outputType, result);  **return** EXIT\_SUCCESS;  }  // Function implementations  // -----------------------------------------------------  **bool** **readInputParams**(**int** argc, **char** \*\*argv, OutputType &outputType, **int** &numThreads) {  **if** (argc != 3) {  printUsage();  **return** **false**;  } **else** {  std::string input(argv[1]);  numThreads = **atoi**(argv[2]);  OutputType outputTypel = stringToOutputType(input);  **if** (numThreads <= 0 || numThreads > INT\_MAX || outputTypel == *NONE*) {  printUsage();  **return** **false**;  }  outputType = outputTypel;  }  **return** **true**;  }  **void** **printUsage**() {  std::cout << "Usage:\n";  std::cout << "\t./pp-ep02-012015 [output type] [number of threads]\n\n";  std::cout << "\toutput type -- Define the output type [time | area | all]\n";  std::cout << "\tnumber of threads -- Define the number of threads to use in processing [a valid interger greater than 0 and less or equal than " << INT\_MAX << "]\n";  std::cout << "Sample:\n";  std::cout << "\t./pp-ep02-012015 all 4\n";  }  OutputType **stringToOutputType**(std::string input) {  **if** (input == "time") {  **return** *TIME*;  } **else** **if** (input == "area") {  **return** *AREA*;  } **else** **if** (input == "all") {  **return** *ALL*;  }  **return** *NONE*;  }  **void** **readVertices**(**const** **long** numVertices, std::vector<Vertice> &vecVertices) {  **double** x = 0.0;  **double** y = 0.0;  **for** (**long** i = 0; i < numVertices; i++) {  std::cin >> x;  std::cin >> y;  vecVertices.push\_back(Vertice(x, y));  }  }  **void** **buildRelations**(**long** numVertices, **int** numThreads, std::vector<RlThreadRangeVertices>& vecRelations) {  **long** chunkSize = std::**ceil**((**double**) (numVertices) / numThreads);  **long** beginIndex = 0;  **long** endIndex = chunkSize - 1;  **for** (**long** i = 0; i < numThreads; i++) {  Result result;  vecRelations.push\_back(RlThreadRangeVertices(i, beginIndex, endIndex, result));  **if** (endIndex + 1 == numVertices) {  **break**;  }  beginIndex = endIndex + 1;  endIndex = beginIndex + chunkSize - 1;  **if** (numVertices <= endIndex) {  endIndex = numVertices - 1;  }  }  }  Result **startParallelProcess**(**const** std::vector<RlThreadRangeVertices>& vecRelations) {  Result result;  timeval startTime;  timeval endTime;  **gettimeofday**(&startTime, NULL);  pthread\_t\* arrayThreads = **new** pthread\_t[vecRelations.size()];  **for** (**unsigned** **int** i = 0; i < vecRelations.size(); i++) {  **pthread\_create**(&arrayThreads[i], NULL, shoelaceTheorem, (**void** \*) &vecRelations[i]);  }  **long** **double** sum = 0.0;  **for** (**unsigned** **int** i = 0; i < vecRelations.size(); i++) {  **pthread\_join**(arrayThreads[i], NULL);  sum += vecRelations[i].getResult().area;  }  result.area = std::abs((sum) / 2.0);  **gettimeofday**(&endTime, NULL);  **long** time = ((endTime.tv\_sec \* 1000000 + endTime.tv\_usec) - (startTime.tv\_sec \* 1000000 + startTime.tv\_usec));  **free**(arrayThreads);  result.processTime = time;  **return** result;  }  **void** \***shoelaceTheorem**(**void** \*arg0) {  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);  }  **void** **printResult**(OutputType outputType, Result result) {  std::cout << std::endl;  **switch** (outputType) {  **case** *TIME*:  std::cout << result.processTime << "\n";  **break**;  **case** *AREA*:  std::cout << std::fixed;  std::cout << std::setprecision(2) << result.area << "\n";  **break**;  **case** *ALL*:  std::cout << std::fixed;  std::cout << std::setprecision(2) << result.area << "\n";  std::cout << result.processTime << "\n";  **break**;  **default**:  **break**;  }  } |