Universidade Federal do Rio de Janeiro

Bacharelado em Ciência da Computação

Computação Concorrente



**Computação Concorrente - 2019.2**

**Relatório de Trabalho Prático**

João Ricardo Campbell Maia - 116111909

Thiago de Oliveira Silva - 111466197

2019

**Introdução:**

A vertente da computação dita Computação Concorrente busca definir um conceito onde atividades são executadas de forma simultânea (concorrentemente), sendo utilizada atualmente em diversas aplicações e em diferentes áreas.

No contexto da disciplina, se desenvolve entre os conceitos de *threads* e *processos*, estudando os impactos de programas que possuem mais de um fluxo de execução e comparando sua eficiência com versões sequenciais dos mesmos. Tem por objetivo também explorar ferramentas que possibilitem a implementação dos mesmos, otimizando assim o uso do processador e melhor aproveitamento dos recursos da máquina.

**Objetivo:**

A atividade consiste na implementação algorítmica de uma programa que calcule áreas através de **integrais definidas** utilizando o método de **integração numérica retangular com ponto médio.** São propostas versões **sequenciais** e **concorrentes** *(com balanceamento de carga entre as threads)* do mesmo programa e a comparação entre os programas revelando o ganho de desempenho obtido.

**Desenvolvimento:**

Foram utilizadas as ferramentas Geany (IDE), Sublime Text (Editor) e linguagem de programação C++ em ambiente Linux.

A solução apresentada consiste em construir um retângulo com largura entre os intervalos de integração da função e altura correspondente ao ponto de máximo. Definir um ponto médio na função e a partir daí gerar dois novos retângulos. Um do intervalo inicial até o ponto médio, outro do ponto médio ao intervalo final. Se a soma desses retângulos for muito maior do que o retângulo inicial, o processo se repete recursivamente nos dois retângulos. Foi criada também uma versão concorrente sem balanceamento para que pudesse ser comparada com a balanceada.

A seguir, uma breve descrição das versões com as soluções encontradas:

* **Versão sequencial:**

Para tal, foram utilizados os seguintes mecanismos:

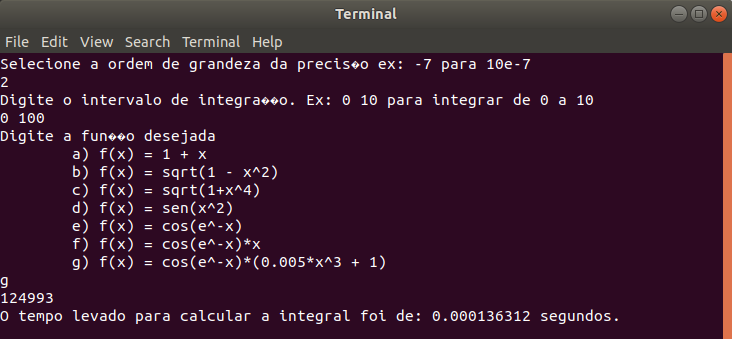
* Função que calcula o ponto médio
* Função que retorna o módulo
* Struct para representar os retângulos
* Função para o cálculo da integral
* Eventuais variáveis auxiliares como para definir a margem de erro

*Fluxo de execução do programa*

*Ao executar, o programa fornece ao usuário a opção de escolher:*

1. *A ordem de grandeza da precisão*
2. *O intervalo de integração*
3. *A função de integração*

*O programa chama a função Integral que cria os retângulos e compara, até que a diferença entre o maior e a soma dos menores seja menor ou igual a margem de erro estipulada. A função executa recursivamente. O valor é acumulado em uma variável soma que é retornado para o usuário como resultado esperado. O programa mede também o tempo gasto na execução da tarefa*



* **Versão concorrente (desbalanceada):**

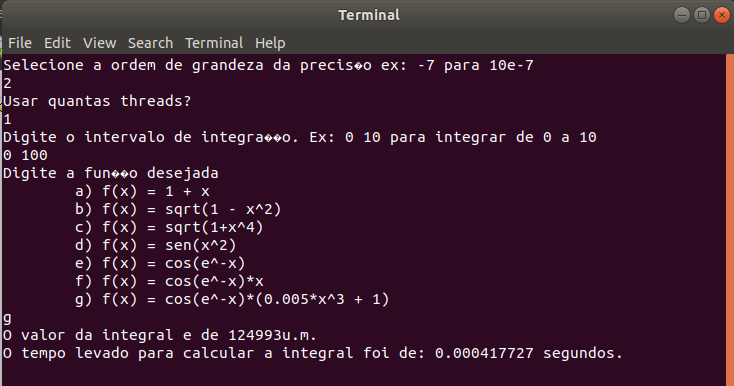
Nessa versão, foram adotadas as seguintes estratégias

* Um mutex da biblioteca Pthread para exclusão mútua
* Função que calcula o ponto médio
* Função que retorna o módulo
* Struct para representar os retângulos
* Função para o cálculo da integral
* Função IntegralConc (cria threads e divide a tarefa em blocos)
* Função IntegralAux (função para as threads)
* Struct para ser passada como argumento para a função IntegralAux
* Eventuais variáveis auxiliares

*Fluxo de execução do programa*

*O programa inicia realizando os mesmos procedimentos da forma sequencial solicitando entradas do usuário.*

*A função* ***IntegralConc*** *agora cria as threads e divide a tarefa em blocos. uma vez criadas, as threads executam a função* ***IntegralAux*** *de forma concorrente, onde cada thread, executa a função* ***Integral****, que antes era realizada por um único fluxo de execução, de forma sequencial. A função trata das devidas exclusões mútuas para que não hajam acessos indevidos simultâneos na variável soma a ser incrementada. Para tal utiliza-se de locks. Ao final, a função IntegralConc utiliza-se de um “join” para aguardar todas as threads terminarem e imprimir o resultado.*



* **Versão concorrente (balanceada)**

Para solucionar o problema do balanceamento de carga entre as threads, foi criado um **buffer**, que funciona como uma pilha de retângulos, implementada usando uma Struct. Temos agora também **dois mutex** (umpara **soma** e outro para o **buffer**)

*Fluxo de execução do programa*

*O programa inicia realizando os mesmos procedimentos da forma sequencial e concorrente desbalanceada solicitando entradas do usuário. A função* ***IntegralConc*** *também realiza os mesmos procedimentos. A principal diferença está no fato de que a função* ***IntegralAux*** *(a função realizada pelas threads) agora (...)*

*[ DESCREVER COMO FUNCIONA O BUFFER, COMO O PROGRAMA FUNCIONA AGORA ETC… ]*



**Resultados:**

*(Resultados obtidos, gráficos, desempenho, uso do processador etc )*

**Considerações finais e conclusão:**

*(O que se conclui do trabalho, o resultado foi o esperado? satisfatório? comentários relevantes? O que aprendemos com isso? Qual o sentido da vida?)*

**Códigos:**

*Versão Sequencial:*

|  |
| --- |
| #include <iostream>  #include <functional>  #include <cmath>  #include <chrono>    using namespace std;    static double Epsilon; // Margem de erro    template <typename A , typename B>  double ptMed (A a, B b) { // Função que calcula o pt médio  return (a+b)/2;  }    template<typename A>  double Modulo( A x) { // Função que retorna o módulo de um número  if ( x < 0 ) return -x;  else return x;  }    struct Retangulo { // Struct usada para representar os retangulos usados para aproximar a integral  Retangulo (double x0, double x1, double y) {  this->x0 = x0;  this->x1 = x1;  this->y = y;  area = (x1-x0)\*y;  }  Retangulo (double x0, double x1, function <double (double)> y) {  this->x0 = x0;  this->x1 = x1;  this->y = y(ptMed(x0,x1));  area = (x1-x0)\*this->y;  }    double getArea(){  return this->area;  }    double x0;  double x1;  double y;  double area;  };    // Redefinição de operadores + e - para somar a area dos retangulos, ou seja,  // Se somarmamos um Retangulo A com um Retangulo B o resultado será a soma de suas áreas    double operator -(Retangulo a, Retangulo b){  return a.getArea()-b.getArea();  }  double operator +(Retangulo a, Retangulo b){  return a.getArea()+b.getArea();  }  double operator - (double a, Retangulo b) {  return a-b.getArea();  }  double operator + (double a, Retangulo b) {  return a+b.getArea();  }  double operator - (Retangulo a, double b) {  return a.getArea()-b;  }  double operator + (Retangulo a, double b) {  return a.getArea()+b;  }      template <typename F>  double Integral(double x0, double x1, F func) {  double soma;  Retangulo Maior{x0,x1,func};  Retangulo MenorEsq{x0,ptMed(x0,x1),func};  Retangulo MenorDir{ptMed(x0,x1),x1,func};    if (Modulo((Maior-MenorEsq-MenorDir)) > Epsilon) {  soma = Integral(MenorEsq.x0,MenorEsq.x1,func) + Integral(MenorDir.x0,MenorDir.x1,func);  }  else {  soma = MenorEsq+MenorDir;  // cout << "FIM " << soma <<endl;  }      return soma;  }  void printMenu(char &input) {  cout << "Digite a função desejada" << endl  << "\ta) f(x) = 1 + x" << endl  << "\tb) f(x) = sqrt(1 - x^2)" << endl  << "\tc) f(x) = sqrt(1+x^4)" << endl  << "\td) f(x) = sen(x^2)" << endl  << "\te) f(x) = cos(e^-x)" << endl  << "\tf) f(x) = cos(e^-x)\*x" << endl  << "\tg) f(x) = cos(e^-x)\*(0.005\*x^3 + 1)" << endl;  cin >> input;  }    void setIntervalo(double& x0, double &x1) {  cout << "Digite o intervalo de integração. Ex: 0 10 para integrar de 0 a 10" << endl;  cin >> x0 >> x1;  }    void setPrecisao() {  cout << "Selecione a ordem de grandeza da precisão ex: -7 para 10e-7" << endl;  cin >> Epsilon;  Epsilon = pow(10,Epsilon);  }    int main () {    char input;  double intervalo1,intervalo2;  setPrecisao();  setIntervalo(intervalo1,intervalo2);  printMenu(input);  auto ini = chrono::high\_resolution\_clock::now(); //Mede o tempo de inicio  switch (input){  case 'a': // f(x) = 1 + x  cout << Integral(intervalo1,intervalo2,[](double x){return x+1;}) << endl;  break;    case 'b': // f(x) = sqrt(1 - x^2)  cout << Integral(intervalo1,intervalo2,[](double x){return sqrt(1-pow(x,2));}) << endl;  break;    case 'c': // f(x) = sqrt(1+x^4)  cout << Integral(intervalo1,intervalo2,[](double x){return sqrt(1+pow(x,4));}) << endl;  break;    case 'd': // f(x) = sen(x^2)  cout << Integral(intervalo1,intervalo2,[](double x){return sin(pow(x,2));}) << endl;  break;    case 'e': // f(x) = cos(e^-x)  cout << Integral(intervalo1,intervalo2,[](double x){return cos(exp(-x));}) << endl;  break;    case 'f': // f(x) = cos(e^-x)\*x  cout << Integral(intervalo1,intervalo2,[](double x){return cos(exp(-x))\*x;}) << endl;  break;    case 'g': // f(x) = cos(e^-x)\*(0.005\*x^3 + 1)  cout << Integral(intervalo1,intervalo2,[](double x){return cos(exp(-x))\*(0.005\*pow(x,3)+1);}) << endl ;  break;    default:  cout << "Opção inválida, tente novamente" << endl << endl << "Voce escolheu: " << input << endl;    }  auto fim = chrono::high\_resolution\_clock::now(); // Mede o Tempo de Fim  chrono::duration<double> tempo = fim - ini; // Calcula o tempo de execução do programa  cout << "O tempo levado para calcular a integral foi de: " << tempo.count() <<" segundos." << endl;  return 0;  } |

*Versão concorrente (desbalanceada):*

|  |
| --- |
| #include <iostream>  #include <functional>  #include <pthread.h>  #include <cmath>  #include <chrono>    using namespace std;    static double Epsilon; // Margem de erro  static int N\_THREADS; // Número de threads    pthread\_mutex\_t mutex;  double somathreads;    // Função que retorna o ponto médio de dois pontos    template <typename A , typename B>  double ptMed (A a, B b){  return (a+b)/2;  }    // Função que retorna o modulo de um número    template<typename A>  double Modulo( A x){  if ( x < 0 ) return -x;  else return x;  }    // Struct usada para representar os retangulos usados para aproximar a integral    struct Retangulo{  Retangulo(double x0, double x1, double y){  this->x0 = x0;  this->x1 = x1;  this->y = y;  area = (x1-x0)\*y;  }  Retangulo(double x0, double x1, function<double(double)> y){  this->x0 = x0;  this->x1 = x1;  this->y = y(ptMed(x0,x1));  area = (x1-x0)\*this->y;  }    double getArea(){  return this->area;  }    double x0;  double x1;  double y;  double area;  };    // Struct usada para ser passada como argumento para a função IntegralAux    struct Arg{    Arg() {  this->x0 = 0;  this->x1 = 0;  this->func = 0;  }  template <typename F>  Arg(double x0,double x1, F func){  this->x0 = x0;  this->x1 = x1;  this->func = func;  }  double x0;  double x1;  function<double(double)> func;  };    // Redefinição de operadores + e - para somar a area dos retangulos, ou seja,  // Se somarmamos um Retangulo A com um Retangulo B o resultado será a soma de suas áreas    double operator -(Retangulo a, Retangulo b){  return a.getArea()-b.getArea();  }  double operator +(Retangulo a, Retangulo b){  return a.getArea()+b.getArea();  }  double operator -(double a, Retangulo b){  return a-b.getArea();  }  double operator +(double a, Retangulo b){  return a+b.getArea();  }  double operator -(Retangulo a, double b){  return a.getArea()-b;  }  double operator +(Retangulo a, double b){  return a.getArea()+b;  }    // Função que realiza integral    template <typename F>  double Integral(double x0, double x1, F func){  double soma;    Retangulo Maior{x0,x1,func};  Retangulo MenorEsq{x0,ptMed(x0,x1),func};  Retangulo MenorDir{ptMed(x0,x1),x1,func};    if (Modulo((Maior-MenorEsq-MenorDir)) > Epsilon){  soma = Integral(MenorEsq.x0,MenorEsq.x1,func) + Integral(MenorDir.x0,MenorDir.x1,func);  }  else {  soma = MenorEsq+MenorDir;  }  return soma;  }    // Função que cria as threads com os blocos criados pela IntegralConc    void\* IntegralAux( void\* arg){  Arg aux = \*(Arg\*) arg;  double x0 = aux.x0;  double x1 = aux.x1;  function<double(double)> func = aux.func;  double soma;    Retangulo Maior{x0,x1,func};  Retangulo MenorEsq{x0,ptMed(x0,x1),func};  Retangulo MenorDir{ptMed(x0,x1),x1,func};    if (Modulo((Maior-MenorEsq-MenorDir)) > Epsilon){  soma = Integral(MenorEsq.x0,MenorEsq.x1,func) + Integral(MenorDir.x0,MenorDir.x1,func);  }  else {  soma = MenorEsq+MenorDir;  }    pthread\_mutex\_lock(&mutex);  somathreads += soma;  pthread\_mutex\_unlock(&mutex);  pthread\_exit(0);  return 0;  }      // Função que divide a tarefa em blocos    template<typename F>  double IntegralConc(double x0, double x1, F func){  Arg arg[N\_THREADS];  int i;  somathreads = 0;  pthread\_t tid[N\_THREADS];  pthread\_mutex\_init(&mutex,0);  double incremento = (x1-x0)/N\_THREADS;  for ( i = 0 ; i < N\_THREADS ; i++){  arg[i] = Arg(x0+incremento\*i,x0+incremento\*(i+1),func);  pthread\_create(&tid[i],0,IntegralAux,(void\*)&arg[i]);  }  for ( int i = 0 ; i < N\_THREADS ; i++){  pthread\_join(tid[i],0);  }  cout << "O valor da integral e de " << somathreads << "u.m." << endl;  pthread\_mutex\_destroy(&mutex);  return somathreads;  }  void printMenu(char &input) {  cout << "Digite a função desejada" << endl  << "\ta) f(x) = 1 + x" << endl  << "\tb) f(x) = sqrt(1 - x^2)" << endl  << "\tc) f(x) = sqrt(1+x^4)" << endl  << "\td) f(x) = sen(x^2)" << endl  << "\te) f(x) = cos(e^-x)" << endl  << "\tf) f(x) = cos(e^-x)\*x" << endl  << "\tg) f(x) = cos(e^-x)\*(0.005\*x^3 + 1)" << endl;  cin >> input;  }    void getIntervalo(double& x0, double &x1) {  cout << "Digite o intervalo de integração. Ex: 0 10 para integrar de 0 a 10" << endl;  cin >> x0 >> x1;  }    void selecionaPrecisao() {  cout << "Selecione a ordem de grandeza da precisão ex: -7 para 10e-7" << endl;  cin >> Epsilon;  Epsilon = pow(10,Epsilon);  }    void selNThreads() {  cout << "Usar quantas threads?" << endl;  cin >> N\_THREADS;  }      int main () {    char input;  double intervalo1,intervalo2;  selecionaPrecisao();  selNThreads();  getIntervalo(intervalo1,intervalo2);  printMenu(input);  auto ini = chrono::high\_resolution\_clock::now();  switch (input){  case 'a': // f(x) = 1 + x  IntegralConc(intervalo1,intervalo2,[](double x){return x+1;}) ;  break;    case 'b': // f(x) = sqrt(1 - x^2)  IntegralConc(intervalo1,intervalo2,[](double x){return sqrt(1-pow(x,2));}) ;  break;    case 'c': // f(x) = sqrt(1+x^4)  IntegralConc(intervalo1,intervalo2,[](double x){return sqrt(1+pow(x,4));}) ;  break;    case 'd': // f(x) = sen(x^2)  IntegralConc(intervalo1,intervalo2,[](double x){return sin(pow(x,2));}) ;  break;    case 'e': // f(x) = cos(e^-x)  IntegralConc(intervalo1,intervalo2,[](double x){return cos(exp(-x));}) ;  break;    case 'f': // f(x) = cos(e^-x)\*x  IntegralConc(intervalo1,intervalo2,[](double x){return cos(exp(-x))\*x;}) ;  break;    case 'g': // f(x) = cos(e^-x)\*(0.005\*x^3 + 1)  IntegralConc(intervalo1,intervalo2,[](double x){return cos(exp(-x))\*(0.005\*pow(x,3)+1);}) ;  break;    default:  cout << "Opção inválida, tente novamente" << endl << endl << "Voce escolheu: " << input << endl;    }  auto fim = chrono::high\_resolution\_clock::now();  chrono::duration<double> tempo = fim - ini;  cout << "O tempo levado para calcular a integral foi de: " << tempo.count() <<" segundos." << endl;  return 0;  } |

*Versão concorrente (balanceada):*

|  |
| --- |
| #include <iostream>  #include <functional>  #include <pthread.h>  #include <cmath>  #include <chrono>  #include <stack>  #include <utility>    using namespace std;    static double Epsilon; // Margem de erro  static int N\_THREADS; // Número de threads    struct Retangulo;  template<typename T>  struct Pilha;    pthread\_mutex\_t mutex\_soma;  pthread\_mutex\_t mutex\_buffer;  double somathreads; // Variavel global que guarda o resultado da integral    template <typename A , typename B> // Função que retorna o ponto médio de dois pontos  double ptMed (A a, B b){  return (a+b)/2;  }    // Função que retorna o modulo de um número    template<typename A>  double Modulo( A x){  if ( x < 0 ) return -x;  else return x;  }    template <typename T>  struct Pilha {  Pilha() {  topo = 0;  }  void push( T valor ) {  tab[topo++] = valor;  }  T pop() {  return tab[--topo];  }  bool empty(){  return topo ? false : true;  }    T tab[1024];  int topo;  };    // Struct usada para representar os retangulos usados para aproximar a integral    struct Retangulo{    Retangulo(){ }  Retangulo(double x0, double x1, function<double(double)> y){  this->x0 = x0;  this->x1 = x1;  this->func = y;  area = (x1-x0)\*y(ptMed(x0,x1));  }    pair<Retangulo,Retangulo> split(){  Retangulo esq{this->x0,ptMed(this->x0,this->x1),func};  Retangulo dir{ptMed(this->x0,this->x1),this->x1,func};  auto rets = make\_pair(esq,dir);  return rets;  }  double getArea(){  return this->area;  }    double x0;  double x1;  function<double(double)> func;  double area;  };    // Redefinição de operadores + e - para somar a area dos retangulos, ou seja,  // Se somarmamos um Retangulo A com um Retangulo B o resultado será a soma de suas áreas    double operator -(Retangulo a, Retangulo b){  return a.getArea()-b.getArea();  }  double operator +(Retangulo a, Retangulo b){  return a.getArea()+b.getArea();  }  double operator -(double a, Retangulo b){  return a-b.getArea();  }  double operator +(double a, Retangulo b){  return a+b.getArea();  }  double operator -(Retangulo a, double b){  return a.getArea()-b;  }  double operator +(Retangulo a, double b){  return a.getArea()+b;  }    Pilha<Retangulo> buffer; // Declaração da variável global buffer, que é uma Pilha de Retangulos.    // Função que processa os retangulos e realiza de fato a integral    void\* IntegralAux( void\* nada){  Retangulo Maior;  pair<Retangulo,Retangulo> Dividido;  Retangulo MenorEsq;  Retangulo MenorDir;  double soma = 0;  while (1){  pthread\_mutex\_lock(&mutex\_buffer);  if (buffer.empty()){ // Se o buffer estiver vazio, adiciona a soma da thread ao total e retorna  pthread\_mutex\_unlock(&mutex\_buffer);  pthread\_mutex\_lock(&mutex\_soma);  somathreads += soma;  pthread\_mutex\_unlock(&mutex\_soma);  pthread\_exit(0);  }  Maior = Retangulo{buffer.pop()}; // Cria um retangulo com o topo da pilha  pthread\_mutex\_unlock(&mutex\_buffer);  MenorEsq = Retangulo(Maior.x0,ptMed(Maior.x0,Maior.x1),Maior.func); // Divide o maior retangulo em dois  MenorDir = Retangulo(ptMed(Maior.x0,Maior.x1),Maior.x1,Maior.func);    if (Modulo((Maior-MenorEsq-MenorDir)) > Epsilon){ // Caso a diferença seja significativa, os dois retangulos são enviados ao buffer  pthread\_mutex\_lock(&mutex\_buffer); // Para serem divididos mais ainda  buffer.push(MenorEsq);  buffer.push(MenorDir);  pthread\_mutex\_unlock(&mutex\_buffer);  }  else {  soma += MenorEsq+MenorDir;  }  }  return 0;  }    template<typename F> // Função que divide a tarefa em blocos, passando os retangulos iniciais para o buffer  double IntegralConc(double x0, double x1, F func){  int i;  somathreads = 0;  pthread\_t tid[N\_THREADS];  pthread\_mutex\_init(&mutex\_soma,0);  pthread\_mutex\_init(&mutex\_buffer,0);  double incremento = (x1-x0)/N\_THREADS;  for ( i = 0 ; i < N\_THREADS ; i++){  cout << "Thread " << i+1 << " irá integrar de " << x0+incremento\*i << " a " << x0+incremento\*(i+1) << endl;  buffer.push(Retangulo{x0+incremento\*i,x0+incremento\*(i+1),func});  }  for ( i = 0 ; i < N\_THREADS ; i++){  pthread\_create(&tid[i],0,IntegralAux,0);  }  for ( int i = 0 ; i < N\_THREADS ; i++){  pthread\_join(tid[i],0);  }  cout << "O valor da integral e de " << somathreads << " u.m." << endl;  pthread\_mutex\_destroy(&mutex\_soma);  return somathreads;  }  void printMenu(char &input) {  cout << "Digite a função desejada" << endl  << "\ta) f(x) = 1 + x" << endl  << "\tb) f(x) = sqrt(1 - x^2)" << endl  << "\tc) f(x) = sqrt(1+x^4)" << endl  << "\td) f(x) = sen(x^2)" << endl  << "\te) f(x) = cos(e^-x)" << endl  << "\tf) f(x) = cos(e^-x)\*x" << endl  << "\tg) f(x) = cos(e^-x)\*(0.005\*x^3 + 1)" << endl;  cin >> input;  }    void getIntervalo(double& x0, double &x1) {  cout << "Digite o intervalo de integração. Ex: 0 10 para integrar de 0 a 10" << endl;  cin >> x0 >> x1;  }    void selecionaPrecisao() {  cout << "Selecione a ordem de grandeza da precisão ex: -7 para 10e-7" << endl;  cin >> Epsilon;  Epsilon = pow(10,Epsilon);  }    void selNThreads() {    cout << "Usar quantas threads?" << endl;  cin >> N\_THREADS;  }      int main () {    char input;  double intervalo1,intervalo2;  selecionaPrecisao();  selNThreads();  getIntervalo(intervalo1,intervalo2);  printMenu(input);  auto ini = chrono::high\_resolution\_clock::now(); //Mede o tempo de inicio  switch (input){  case 'a': // f(x) = 1 + x  IntegralConc(intervalo1,intervalo2,[](double x){return x+1;}) ;  break;    case 'b': // f(x) = sqrt(1 - x^2)  IntegralConc(intervalo1,intervalo2,[](double x){return sqrt(1-pow(x,2));}) ;  break;    case 'c': // f(x) = sqrt(1+x^4)  IntegralConc(intervalo1,intervalo2,[](double x){return sqrt(1+pow(x,4));}) ;  break;    case 'd': // f(x) = sen(x^2)  IntegralConc(intervalo1,intervalo2,[](double x){return sin(pow(x,2));}) ;  break;    case 'e': // f(x) = cos(e^-x)  IntegralConc(intervalo1,intervalo2,[](double x){return cos(exp(-x));}) ;  break;    case 'f': // f(x) = cos(e^-x)\*x  IntegralConc(intervalo1,intervalo2,[](double x){return cos(exp(-x))\*x;}) ;  break;    case 'g': // f(x) = cos(e^-x)\*(0.005\*x^3 + 1)  IntegralConc(intervalo1,intervalo2,[](double x){return cos(exp(-x))\*(0.005\*pow(x,3)+1);}) ;  break;    default:  cout << "Opção inválida, tente novamente" << endl << endl << "Voce escolheu: " << input << endl;  }  auto fim = chrono::high\_resolution\_clock::now(); // Mede o Tempo de Fim  chrono::duration<double> tempo = fim - ini; // Calcula o tempo de execução do programa  cout << "O tempo levado para calcular a integral foi de: " << tempo.count() <<" segundos." << endl;  return 0;  } |