# Programação C/S Usando RPC – Chamada de Procedimento Remoto

O texto aborda sistemas distribuídos e destaca as Chamadas de Procedimento Remoto (RPC) como essenciais para a implementação de requisitos como escalabilidade e transparência. RPC permite a comunicação entre processos em hosts distintos, usando stubs para encapsular funções e facilitar a programação.

1. **Princípio da Comunicação RPC:**
   1. RPC permite que programadores escrevam código de forma semelhante à programação convencional.
   2. Uso de stubs para encapsular funções associadas à conexão remota.
   3. Comunicação cliente-servidor ocorre por meio de mensagens trocadas pelos stubs.
2. **Conversão de Tipos de Dados:**
   1. Biblioteca XDR (eXternal Data Representation) padroniza tipos de dados para comunicação entre sistemas com representações internas diferentes.
   2. Funções XDR realizam conversões entre tipos de dados locais e formato XDR.
3. **Localização do Servidor RPC:**
   1. Aplicações RPC dependem de um serviço de diretórios para localizar o servidor remoto.
   2. O serviço de diretórios, como o portmapper no Linux, mapeia números de programa RPC para portas TCP/IP.
4. **Uso do rpcgen para Gerar Stubs:**
   1. A ferramenta rpcgen gera módulos de software, incluindo stubs, a partir de um arquivo de definição de interface (IDF).
   2. Passos incluem identificação das funcionalidades, aplicação do rpcgen e modificação dos arquivos do cliente e servidor.
5. **Modificações e Cuidados:**
   1. Alterações na lógica dos módulos podem ser feitas sem regerar os códigos, desde que não comprometam a passagem de parâmetros.
   2. Alterações que afetam parâmetros ou funções no servidor exigem nova geração de códigos.
   3. Estratégia de encapsular parâmetros em uma estrutura de dados facilita a passagem de informações.
   4. O processo portmapper deve estar sempre ativo no servidor, e o servidor deve ser acionado antes do cliente.
6. **Gerar a aplicação de forma modular usando programação convencional**

| #include <stdio.h> 2 3 int add (int x, int y) { 4 int result; 5 6 printf("Requisicao de adicao para %d e %d\n", x, y); 7 result = x + y; 8 return(result); 9 } /\* fim funcao add \*/ 10 11 int sub (int x, int y) { 12 int result; 13 14 printf("Requisicao de subtracao para %d e %d\n", x, y); 15 result = x - y; 16 return(result); 17 } /\* fim funcao sub \*/ 18 19 int main( int argc, char \*argv[]) { 20 int x,y; 21 22 if (argc!=3) { 23 Fprintf(stderr,"Uso correto: %s num1 num2\n",argv[0]); 24 exit(0); } 25 26 /\* Recupera os 2 operandos passados como argumento \*/ 27 x = atoi(argv[1]); y = atoi(argv[2]); 28 printf("%d + %d = %d\n",x,y, add(x,y)); 29 printf("%d - %d = %d\n",x,y, sub(x,y)); 30 return(0); 31 32 } /\* fim main \*/ |
| --- |

1. Gerar o arquivo de definição de Interface (IDF) a partir do código apresentado no item (6)

| #include "calcula.h" 2 3 Void prog\_100(char \*host) { 4 CLIENT \*clnt; 5 Int \*result\_1; 6 Operandos add\_100\_arg; 7 Int \*result\_2; 8 Operandos sub\_100\_arg; 9 10 #ifndef DEBUG 11 clnt = clnt\_create (host, PROG, VERSAO, "udp"); 12 13 #endif /\* DEBUG \*/ 14 15 result\_1 = add\_100(&add\_100\_arg, clnt); 16 result\_2 = sub\_100(&sub\_100\_arg, clnt); 17 18 #ifndef DEBUG 19 clnt\_destroy (clnt); 20 #endif /\* DEBUG \*/ 21 } 22 int main (int argc, char \*argv[]) { 23 char \*host; 24 25 if (argc < 2) { 26 printf ("usage: %s server\_host\n", argv[0]); 27 exit (1); } 28 host = argv[1]; 29 prog\_100 (host); 30 exit (0); } |
| --- |

1. **Alguns detalhes sobre a Interface de Definição de tipos XDR**
   1. Destaca-se que as declarações XDR são semelhantes às declarações em C, mas existem diferenças devido às características específicas da linguagem XDR. Algumas características mencionadas incluem tipos básicos de inteiros, números reais, tipo void, apontadores para construir estruturas dinâmicas, tipos enumerados, booleanos, strings, tipos opacos e extensões para construções padrão C, como unions discriminadas.
   2. Exemplos de declarações XDR são fornecidos:
      1. Definição de um tipo composto chamado "alarme" com uma enumeração e uma variável string.
      2. Uso de uma constante para especificar o tamanho máximo de uma string em uma estrutura.
      3. Demonstração de unions discriminadas no XDR.
      4. Exemplo de estruturas alocadas dinamicamente com um destaque para a responsabilidade do programador em liberar a memória.
   3. Além disso, são discutidas questões relacionadas à alocação e liberação de memória em ambientes distribuídos ao lidar com estruturas dinâmicas. O texto ressalta a importância de liberar memória alocada, especialmente quando o cliente e o servidor alocam memória em chamadas RPC.
   4. Por fim, são fornecidos insights sobre a especificação do protocolo RPC, enfatizando a necessidade de garantir a idempotência em chamadas de procedimentos remotos. O uso de datagramas UDP é mencionado, com destaque para a semântica "pelo menos uma vez" em situações de possíveis duplicatas. A segurança e integridade de sequências de dados são destacadas ao optar pelo uso do protocolo TCP.

## “Qual a diferença entre RPC básico e gRPC?”

Bard:

O RPC básico e o gRPC são ambos protocolos de chamada de procedimento remoto, mas eles têm algumas diferenças importantes.

RPC básico

O RPC básico é um protocolo simples que permite que dois processos se comuniquem entre si. O RPC básico é implementado usando o protocolo TCP/IP.

O RPC básico tem as seguintes características:

É um protocolo simples e fácil de implementar.

É flexível e pode ser usado para implementar uma variedade de serviços.

Não é eficiente para tráfego de dados grande ou complexo.

gRPC

O gRPC é um protocolo RPC mais recente e avançado que é baseado no protocolo HTTP/2. O gRPC é mais eficiente que o RPC básico e pode ser usado para implementar serviços complexos com mais facilidade.

O gRPC tem as seguintes características:

É mais eficiente que o RPC básico, especialmente para tráfego de dados grande ou complexo.

Suporta recursos avançados, como autenticação, segurança e streaming.

É mais difícil de implementar que o RPC básico.

Comparação

A tabela a seguir compara as principais diferenças entre o RPC básico e o gRPC:

| **Característica** | **RPC básico** | **gRPC** |
| --- | --- | --- |
| Simplicidade | Simples | Mais complexo |
| Flexibilidade | Flexível | Menos flexível |
| Eficiência | Menos eficiente | Mais eficiente |
| Recursos avançados | Menos recursos | Mais recursos |
| Implementação | Fácil | Mais difícil |

Qual protocolo usar?

O protocolo RPC básico é uma boa escolha para aplicações simples e de baixo tráfego. O gRPC é uma boa escolha para aplicações complexas e de alto tráfego que exigem alta eficiência e recursos avançados.

“Explique de forma um pouco mais aprofundada”

Bard:

Simplicidade

O RPC básico é um protocolo simples, com um conjunto básico de funcionalidades. Isso facilita a implementação e o uso do RPC básico, mas também significa que ele não é tão flexível quanto o gRPC.

O gRPC é um protocolo mais complexo, com uma gama mais ampla de funcionalidades. Isso torna o gRPC mais difícil de implementar e usar, mas também significa que ele é mais flexível e pode ser usado para implementar serviços mais complexos.

Flexibilidade

O RPC básico é flexível o suficiente para ser usado para implementar uma variedade de serviços. No entanto, o gRPC é ainda mais flexível, pois suporta recursos avançados, como autenticação, segurança e streaming.

A autenticação permite que os clientes sejam identificados e autorizados a acessar um serviço. A segurança permite que os dados sejam criptografados e protegidos contra acesso não autorizado. O streaming permite que os dados sejam enviados de forma contínua, em vez de em blocos.

Eficiência

O RPC básico não é tão eficiente quanto o gRPC, especialmente para tráfego de dados grande ou complexo.

O gRPC é mais eficiente porque usa o protocolo HTTP/2, que é um protocolo de comunicação eficiente que foi projetado para lidar com grandes volumes de dados.

Recursos avançados

O RPC básico não suporta recursos avançados, como autenticação, segurança e streaming.

O gRPC suporta esses recursos avançados, o que o torna uma escolha melhor para aplicações que exigem alta segurança e desempenho.

Implementação

O RPC básico é fácil de implementar, pois usa um conjunto básico de APIs.

O gRPC é mais difícil de implementar, pois usa um conjunto de APIs mais complexo.

Qual protocolo usar?

O protocolo RPC básico é uma boa escolha para aplicações simples e de baixo tráfego. O gRPC é uma boa escolha para aplicações complexas e de alto tráfego que exigem alta eficiência e recursos avançados.

Exemplos

Aqui estão alguns exemplos de quando você pode usar o RPC básico:

* Para implementar uma API REST simples
* Para enviar dados entre dois processos em uma rede local
* Para implementar um serviço de comunicação simples

Aqui estão alguns exemplos de quando você pode usar o gRPC:

* Para implementar uma API REST complexa
* Para enviar dados entre dois processos em uma rede WAN
* Para implementar um serviço de comunicação seguro e eficiente

Conclusão

O RPC básico e o gRPC são ambos protocolos de chamada de procedimento remoto, mas eles têm diferentes pontos fortes e fracos. O RPC básico é uma boa escolha para aplicações simples e de baixo tráfego, enquanto o gRPC é uma boa escolha para aplicações complexas e de alto tráfego que exigem alta eficiência e recursos avançados.

# MPI – Message Passing Interface

* Processos no MPI:
  + Arquitetura distribuída: Descreve a configuração em que os processos MPI (Message Passing Interface) operam. A memória é local aos processos, e para compartilhar conteúdos de memória, é necessário usar o serviço de mensagens do MPI.
  + Comunicadores e Grupos: Os processos fazem parte de pelo menos um grupo, geralmente MPI\_COMM\_WORLD, que é o comunicador principal. Cada processo é identificado por um rank, semelhante a um PID dentro do grupo. Os processos podem ser executados localmente ou remotamente.
* Algumas primitivas MPI (~125 disponíveis):
  + MPI\_Init: Inicializa o ambiente MPI.
  + MPI\_Finalize: Finaliza o ambiente MPI.
  + MPI\_Comm\_size: Identifica o número de processos no grupo.
  + MPI\_Comm\_rank: Identifica o número do rank do processo.
  + MPI\_Send: Envia uma mensagem.
  + MPI\_Recv: Recebe uma mensagem.
* Alguns tipos de dados MPI:
  + Tipos de dados MPI são fornecidos para especificar o conteúdo das mensagens, como MPI\_CHAR, MPI\_INT, MPI\_DOUBLE, etc.
* Para compilar/executar:
  + Compilação: Utiliza-se o comando mpicc nome\_do\_arquivo.c -o nome\_do\_executavel.
  + Execução: Pode-se usar o comando mpirun -np 4 ./nome\_do\_executavel para executar o programa MPI.
  + No código, é necessário incluir a biblioteca mpi.h.

**Arquitetura MPI**

**“Descreva a arquitetura mpi como se estivesse explicando para um aluno do curso de engenharia de software“**

Bard:

Olá, aluno de engenharia de software. Hoje vamos falar sobre a arquitetura MPI, um padrão de interface para a comunicação entre processos em máquinas paralelas com memória distribuída.

Imagine que você tem um grande problema para resolver. Para resolvê-lo, você pode dividir o problema em partes menores, que podem ser resolvidas por diferentes processos. Cada processo pode ser executado em um processador diferente de uma máquina paralela.

Para que os processos possam se comunicar e cooperar entre si, eles precisam de uma forma de trocar dados e sincronizar suas ações. É aí que entra a arquitetura MPI.

A arquitetura MPI é baseada no paradigma de passagem de mensagens. Nesse paradigma, os processos trocam dados enviando mensagens uns aos outros. As mensagens são encapsuladas em estruturas de dados que especificam o tipo de dados, o tamanho e o destino da mensagem.

O MPI fornece um conjunto abrangente de rotinas para a troca de mensagens. Essas rotinas permitem que os processos enviem e recebam mensagens, bem como que sincronizem suas ações.

Aqui estão alguns exemplos de rotinas MPI:

MPI\_Send: envia uma mensagem para outro processo.

MPI\_Recv: recebe uma mensagem de outro processo.

MPI\_Barrier: sincroniza a execução de todos os processos.

Para usar a arquitetura MPI, você precisa primeiro inicializar o ambiente MPI. Em seguida, você pode criar processos MPI e iniciar a comunicação entre eles.

Aqui está um exemplo simples de código MPI:

| #include <mpi.h>  int main(int argc, char\*\* argv) {  // Inicializa o ambiente MPI  MPI\_Init(&argc, &argv);   // Cria dois processos MPI  int rank, size;  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);   // Envia uma mensagem do processo 0 para o processo 1  if (rank == 0) {  int data = 10;  MPI\_Send(&data, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);  }   // Recebe uma mensagem do processo 0  if (rank == 1) {  int data;  MPI\_Recv(&data, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);  printf("Recebi o valor %d do processo 0\n", data);  }   // Finaliza o ambiente MPI  MPI\_Finalize();   return 0; } |
| --- |

Esse código cria dois processos MPI. O processo 0 envia o valor 10 para o processo 1. O processo 1 recebe o valor 10 e o imprime na tela.

A arquitetura MPI é uma ferramenta poderosa que pode ser usada para acelerar o desempenho de aplicações paralelas. Ela é uma escolha popular para a programação paralela em máquinas com memória distribuída e é usada em uma ampla variedade de aplicações, incluindo simulações científicas, processamento de imagens e computação de alto desempenho.

“Liste e explique os pontos da arquitetura MPI que tem mais chance de cair em uma avaliação sobre o tema”

Bard:

Rascunho A

Aqui estão alguns pontos da arquitetura MPI que têm mais chance de cair em uma avaliação sobre o tema:

O paradigma de passagem de mensagens: o MPI é baseado no paradigma de passagem de mensagens, no qual os processos trocam dados enviando mensagens uns aos outros. As mensagens são encapsuladas em estruturas de dados que especificam o tipo de dados, o tamanho e o destino da mensagem.

As rotinas MPI: o MPI fornece um conjunto abrangente de rotinas para a troca de mensagens. Essas rotinas permitem que os processos enviem e recebam mensagens, bem como que sincronizem suas ações.

A inicialização do ambiente MPI: para usar a arquitetura MPI, você precisa primeiro inicializar o ambiente MPI. Isso é feito chamando a rotina MPI\_Init().

A criação de processos MPI: depois de inicializar o ambiente MPI, você pode criar processos MPI. Isso é feito chamando a rotina MPI\_Comm\_spawn().

A comunicação entre processos MPI: a comunicação entre processos MPI é feita usando as rotinas MPI\_Send(), MPI\_Recv() e outras.

A sincronização entre processos MPI: a sincronização entre processos MPI é feita usando as rotinas MPI\_Barrier(), MPI\_Wait() e outras.

A finalização do ambiente MPI: para finalizar o ambiente MPI, você deve chamar a rotina MPI\_Finalize()

Rascunho B:

Aqui estão alguns pontos da arquitetura MPI que têm mais chance de cair em uma avaliação sobre o tema:

Definição e conceito de MPI: O que é MPI? Quais são os seus objetivos?

Paradigma de passagem de mensagens: Como os processos MPI trocam dados?

Rotas MPI: Quais são as principais rotinas MPI para troca de dados e sincronização?

Modelo de comunicação MPI: Como o MPI organiza os processos em uma máquina paralela?

Eficiência e portabilidade: Quais são os benefícios da arquitetura MPI em termos de eficiência e portabilidade?

Aqui estão alguns exemplos de perguntas que podem ser feitas em uma avaliação sobre MPI:

Explique o conceito de passagem de mensagens.

Quais são as principais rotinas MPI para troca de dados?

Como o MPI organiza os processos em uma máquina paralela?

Quais são os benefícios da arquitetura MPI em termos de eficiência e portabilidade?

Além desses pontos, os alunos também devem estar familiarizados com os seguintes conceitos:

Tipos de dados MPI: O MPI fornece um conjunto de tipos de dados primitivos e compostos.

Erros MPI: O MPI fornece um conjunto de rotinas para tratamento de erros.

Bibliotecas MPI: O MPI está disponível em uma variedade de bibliotecas, incluindo C, C++, Fortran e Python.

Os alunos devem ser capazes de aplicar esses conceitos para resolver problemas de programação paralela usando MPI.

## Lista de Exercícios - MPI

1. Construir um programa MPI no qual o master passa um número inteiro diferente para cada processo escravo/worker. Esses, por sua vez, imprimem o número recebido junto com seu rank.

| #include <mpi.h>  int main(int argc, char\*\* argv) {  int rank, size;  int num;   MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);   if (rank == 0) {  // O processo master gera os números  for (int i = 0; i < size; i++) {  num = rand() % 100;  MPI\_Send(&num, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD);  }  } else {  // Os processos escravos recebem os números  MPI\_Recv(&num, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);   // Imprimem o número recebido junto com seu rank  printf("Processo %d recebeu %d\n", rank, num);  }   MPI\_Finalize();  return 0; } |
| --- |

1. Altere o programa anterior de modo que o escravo altere o número recebido (somando duas unidades ao valor recebido) e devolva o vetor alterado para o master, que deve imprimi-lo.

| #include <mpi.h>  int main(int argc, char\*\* argv) {  int rank, size;  int num;   MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);   if (rank == 0) {  // O processo master gera os números  for (int i = 0; i < size; i++) {  num = rand() % 100;  MPI\_Send(&num, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD);  }  } else {  // Os processos escravos recebem os números  MPI\_Recv(&num, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);   // Alteram o número recebido  num += 2;   // Enviam o número alterado para o master  MPI\_Send(&num, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);  }   MPI\_Finalize();  return 0; } |
| --- |

1. Construir um programa MPI no qual o master passa um vetor de inteiros diferentes para cada processo escravo. Esses, por sua vez, imprimem o vetor recebido junto com o seu rank.

| #include <mpi.h>  int main(int argc, char\*\* argv) {  int rank, size;  int num\_elems;  int\* vec;   MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);   if (rank == 0) {  // O processo master gera o vetor  num\_elems = 10;  vec = (int\*)malloc(num\_elems \* sizeof(int));  for (int i = 0; i < num\_elems; i++) {  vec[i] = rand() % 100;  }   // Envia o vetor para os processos escravos  for (int i = 1; i < size; i++) {  MPI\_Send(&num\_elems, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD);  MPI\_Send(vec, num\_elems, MPI\_INT, i, 0, MPI\_COMM\_WORLD);  }  } else {  // Os processos escravos recebem o vetor  MPI\_Recv(&num\_elems, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  vec = (int\*)malloc(num\_elems \* sizeof(int));  MPI\_Recv(vec, num\_elems, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);   // Imprimem o vetor recebido junto com seu rank  printf("Processo %d recebeu: ", rank);  for (int i = 0; i < num\_elems; i++) {  printf("%d ", vec[i]);  }  printf("\n");  }   MPI\_Finalize();  return 0; } |
| --- |

1. Altere o programa anterior de modo que os escravos subtraiam uma unidade de cada posição no vetor recebido e devolvam o vetor alterado para o mestre/master, que será encarregado por imprimir cada vetor recebido.

| #include <mpi.h>  int main(int argc, char\*\* argv) {  int rank, size;  int num\_elems;  int\* vec;   MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);   if (rank == 0) {  // O processo master gera o vetor  num\_elems = 10;  vec = (int\*)malloc(num\_elems \* sizeof(int));  for (int i = 0; i < num\_elems; i++) {  vec[i] = rand() % 100;  }   // Envia o vetor para os processos escravos  for (int i = 1; i < size; i++) {  MPI\_Send(&num\_elems, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD);  MPI\_Send(vec, num\_elems, MPI\_INT, i, 0, MPI\_COMM\_WORLD);  }  } else {  // Os processos escravos recebem o vetor  MPI\_Recv(&num\_elems, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  vec = (int\*)malloc(num\_elems \* sizeof(int));  MPI\_Recv(vec, num\_elems, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);   // Subtraem uma unidade de cada posição no vetor recebido  for (int i = 0; i < num\_elems; i++) {  vec[i] -= 1;  }   // Enviam o vetor alterado para o master  MPI\_Send(&num\_elems, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);  MPI\_Send(vec, num\_elems, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);  }   if (rank == 0) {  // O processo master recebe os vetores alterados  for (int i = 1; i < size; i++) {  MPI\_Recv(&num\_elems, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  vec = (int\*)malloc(num\_elems \* sizeof(int));  MPI\_Recv(vec, num\_elems, MPI\_INT, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);   // Imprimem o vetor alterado  printf("Processo %d recebeu: ", i);  for (int j = 0; j < num\_elems; j++) {  printf("%d ", vec[j]);  }  printf("\n");  }  }   MPI\_Finalize();  return 0; } |
| --- |

1. Construir um programa MPI no qual os processos escravos sinalizem para o processo master que estão vivos (“Oi, Estou vivo!”). O master, por sua vez, deve imprimir a mensagem recebida junto com o rank do escravo que enviou.

| #include <mpi.h>  int main(int argc, char\*\* argv) {  int rank, size;  char msg[100];   MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);   if (rank != 0) {  // Os escravos enviam a mensagem  sprintf(msg, "Oi, Estou vivo!");  MPI\_Send(msg, strlen(msg) + 1, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD);  } else {  // O master recebe as mensagens  for (int i = 1; i < size; i++) {  MPI\_Recv(msg, 100, MPI\_CHAR, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  printf("Processo %d: %s\n", i, msg);  }  }   MPI\_Finalize();  return 0; } |
| --- |

1. Elaborar um programa MPI que faça o repasse de um token entre processos residentes em hosts distintos, de forma circular. Ou seja, o token começa no token zero, vai para o rank 1 e assim sucessivamente até voltar de novo para o rank 0. Por sua vez, cada processo deve imprimir mensagens de sinalização de recepção e/ou envio do token para o próximo nó. Em função da sua posição na lista circular, cada processo deve imprimir um dos seguintes tipos de mensagem exemplificados: Rank[0/'cm1']: enviando token para Rank[1/'cm2']: recebeu token e enviou para 2. Obs.: Opcionalmente, pode-se usar a função MPI\_Get\_processor\_name(hostname, &hostname\_len) para recuperar o nome do host. Neste caso, hostname é um vetor de char e hostname\_len é do tipo int

| #include <mpi.h> #include <stdio.h>  int main(int argc, char\*\* argv) {  int rank, size, next\_rank;  char hostname[MPI\_MAX\_PROCESSOR\_NAME];  int hostname\_len;   MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);   // Obtém o nome do host  MPI\_Get\_processor\_name(hostname, &hostname\_len);   // Define o próximo nó  next\_rank = (rank + 1) % size;   // Envia o token para o próximo nó  if (rank != size - 1) {  printf("Rank[%d/'%s']: enviando token para Rank[%d/'%s'].\n", rank, hostname, next\_rank, hostname);  MPI\_Send(NULL, 0, MPI\_BYTE, next\_rank, 0, MPI\_COMM\_WORLD);  } else {  // O processo rank size-1 envia o token de volta para o rank 0  printf("Rank[%d/'%s']: enviando token para Rank[%d/'%s'].\n", rank, hostname, 0, hostname);  MPI\_Send(NULL, 0, MPI\_BYTE, 0, 0, MPI\_COMM\_WORLD);  }   // Recebe o token do próximo nó  if (rank != 0) {  MPI\_Recv(NULL, 0, MPI\_BYTE, next\_rank, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  printf("Rank[%d/'%s']: recebeu token e enviou para Rank[%d/'%s'].\n", rank, hostname, next\_rank, hostname);  } else {  // O processo rank 0 recebe o token de volta  MPI\_Recv(NULL, 0, MPI\_BYTE, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  printf("Rank[%d/'%s']: recebeu token e encerrou.\n", rank, hostname);  }   MPI\_Finalize();  return 0; } |
| --- |

## MPI – Trabalhando com vetores e matrizes

1. Elabore um programa MPI com três processos, sendo um master e dois slaves. O master deve inicializar o vetor; os slaves devem imprimir, cada um, a metade do vetor inicializado pelo master.

| #include <mpi.h>  int main(int argc, char\*\* argv) {  int rank, size, half\_size;  int\* vec;   MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);   if (rank == 0) {  // O processo master inicializa o vetor  vec = (int\*)malloc(size \* sizeof(int));  for (int i = 0; i < size; i++) {  vec[i] = rand() % 100;  }  }   // O processo master envia metade do vetor para cada slave  if (rank == 0) {  half\_size = size / 2;  MPI\_Send(&vec[0], half\_size, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);  MPI\_Send(&vec[half\_size], half\_size, MPI\_INT, 2, 0, MPI\_COMM\_WORLD);  } else if (rank == 1 || rank == 2) {  // Os slaves recebem metade do vetor do master  MPI\_Recv(&vec[0], half\_size, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);   // Imprimem a metade do vetor recebida  printf("Processo %d: ", rank);  for (int i = 0; i < half\_size; i++) {  printf("%d ", vec[i]);  }  printf("\n");  }   MPI\_Finalize();  return 0; } |
| --- |

1. Elabore um programa MPI com N processos, sendo o master o responsável por inicializar o vetor e os slaves, responsáveis por imprimir uma porção do vetor, proporcional ao número de slaves identificados pelo programa.

| #include <mpi.h>  int main(int argc, char\*\* argv) {  int rank, size, chunk\_size;  int\* vec;   MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);   if (rank == 0) {  // O processo master inicializa o vetor  vec = (int\*)malloc(size \* sizeof(int));  for (int i = 0; i < size; i++) {  vec[i] = rand() % 100;  }  }   // O processo master calcula o tamanho do chunk  chunk\_size = size / MPI\_COMM\_WORLD\_SIZE;   // O processo master envia o chunk para cada slave  if (rank == 0) {  for (int i = 1; i < size; i++) {  MPI\_Send(&vec[i \* chunk\_size], chunk\_size, MPI\_INT, i, 0, MPI\_COMM\_WORLD);  }  } else {  // Os slaves recebem o chunk do master  MPI\_Recv(&vec[rank \* chunk\_size], chunk\_size, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);   // Imprimem o chunk recebido  printf("Processo %d: ", rank);  for (int i = 0; i < chunk\_size; i++) {  printf("%d ", vec[rank \* chunk\_size + i]);  }  printf("\n");  }   MPI\_Finalize();  return 0; } |
| --- |

1. Elabore um programa MPI que imprima um vetor de 100 posições (de tipo int), considerando o seguinte:
   1. O vetor deve ser impresso da posição 0 até a posição 99, nesta ordem
   2. O master deve inicializar o vetor de 100 posições da seguinte forma: v[i]=i
   3. O master deve distribuir a impressão entre os workers de modo que todos possam imprimir pelo menos uma porção do vetor
   4. Cada worker, uma vez acionado, deve imprimir o vetor a partir do ponto de impressão recebido do master
   5. Considerar que este programa pode ser executado por, no máximo, 6 processos (1 master e 5 workers)
   6. O número de posições a serem impressas pelo worker deve obedecer a um offset dinâmico, ou seja, um valor randômico – menor que 15 – que é calculado por cada processo, no momento em que é acionado para imprimir o vetor
   7. O programa deve controlar a impressão de modo que o vetor inteiro seja impresso, mas nenhuma posição seja impressa mais de uma vez. Por exemplo, se o worker anterior imprimiu até a posição 18 e o offset dinâmico calculado foi 10, a thread atual deve imprimir da posição 19 considerando 10 posições adiante
   8. A ação dos workers e do master acaba quando o vetor de 100 posições tiver sido todo impresso.

| #include <mpi.h>  int main(int argc, char\*\* argv) {  int rank, size, next\_rank, offset, start, end;  int\* vec;   MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);   if (rank == 0) {  // O processo master inicializa o vetor  vec = (int\*)malloc(100 \* sizeof(int));  for (int i = 0; i < 100; i++) {  vec[i] = i;  }   // O processo master distribui a impressão entre os workers  for (int i = 1; i < size; i++) {  offset = rand() % 15;  MPI\_Send(&offset, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD);  }  } else {  // Os workers recebem o offset  MPI\_Recv(&offset, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);   // Os workers calculam o início e o fim da impressão  start = rank \* 100 / size + offset;  end = (rank + 1) \* 100 / size + offset;   // Os workers imprimem o vetor  for (int i = start; i < end; i++) {  printf("Processo %d: %d\n", rank, vec[i]);  }  }   MPI\_Finalize();  return 0; } |
| --- |

1. Utilizando a biblioteca MPI, elabore um programa multi-processos para somar os elementos de duas matrizes A e B, quadradas (int ou float), para gerar a matriz C, seguindo as seguintes regras: (i) O programa deve conter um processo master e quatro processos workers que deverão trabalhar em conjunto para garantir a realização de soma dos elementos das matrizes A e B; (ii) Supor que as matrizes são de 16 posições e as matrizes A e B devem ser inicializadas com números randômicos; (iii) As operações de soma devem ser distribuídas uniformemente entre os workers; (iv) Ao final, a matriz C resultante deve ser impressa (em colunas, formato de matriz) pelo processo master

| #include <mpi.h>  int main(int argc, char\*\* argv) {  int rank, size, chunk\_size;  int\* A, \*B, \*C;   MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);   if (rank == 0) {  // O processo master inicializa as matrizes  A = (int\*)malloc(16 \* 16 \* sizeof(int));  B = (int\*)malloc(16 \* 16 \* sizeof(int));  for (int i = 0; i < 16; i++) {  for (int j = 0; j < 16; j++) {  A[i \* 16 + j] = rand() % 100;  B[i \* 16 + j] = rand() % 100;  }  }   // O processo master calcula o tamanho do chunk  chunk\_size = 16 / size;   // O processo master distribui a matriz A entre os workers  for (int i = 1; i < size; i++) {  MPI\_Send(&A[i \* chunk\_size \* 16], chunk\_size \* 16, MPI\_INT, i, 0, MPI\_COMM\_WORLD);  }   // O processo master distribui a matriz B entre os workers  for (int i = 1; i < size; i++) {  MPI\_Send(&B[i \* chunk\_size \* 16], chunk\_size \* 16, MPI\_INT, i, 1, MPI\_COMM\_WORLD);  }  } else {  // Os workers recebem o chunk de A  MPI\_Recv(&A[rank \* chunk\_size \* 16], chunk\_size \* 16, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);   // Os workers recebem o chunk de B  MPI\_Recv(&B[rank \* chunk\_size \* 16], chunk\_size \* 16, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);   // Os workers calculam a matriz C  for (int i = 0; i < chunk\_size; i++) {  for (int j = 0; j < 16; j++) {  C[rank \* chunk\_size \* 16 + i \* 16 + j] = A[rank \* chunk\_size \* 16 + i \* 16 + j] + B[rank \* chunk\_size \* 16 + i \* 16 + j];  }  }  }   if (rank == 0) {  // O processo master imprime a matriz C  for (int i = 0; i < 16; i++) {  for (int j = 0; j < 16; j++) {  printf("%d ", C[i \* 16 + j]);  }  printf("\n");  }  }   MPI\_Finalize();  return 0; } |
| --- |

# Arquitetura OpenMP

“Descreva a arquitetura OpenMP como se estivesse explicando para um aluno do curso de engenharia de software”

Baed:

Olá, aluno de engenharia de software. Hoje vamos falar sobre a arquitetura OpenMP, um padrão de programação paralela que permite a paralelização de código sequencial em máquinas com memória compartilhada.

Imagine que você tem um programa sequencial que faz um loop para calcular o somatório dos números de 1 a 100. Para paralelizar esse programa, você pode usar a arquitetura OpenMP para dividir o loop em várias partes, que podem ser executadas por diferentes threads.

A arquitetura OpenMP é baseada no paradigma de threads. Um thread é uma unidade de execução leve que pode ser executada concorrentemente com outros threads.

Para usar a arquitetura OpenMP, você precisa primeiro incluir a biblioteca OpenMP no seu programa. Em seguida, você pode usar as diretivas OpenMP para controlar a paralelização do seu código.

Aqui estão alguns exemplos de diretivas OpenMP:

parallel: inicia um bloco de código que será executado por vários threads.

for: paraleliza um loop.

atomic: garante que uma operação seja executada de forma atômica.

Aqui está um exemplo simples de código OpenMP:

| #include <stdio.h> #include <omp.h>  int main() {  // Inicializa a biblioteca OpenMP  omp\_set\_num\_threads(4);   // Declara uma variável para armazenar o somatório  int sum = 0;   // Percorre os números de 1 a 100 em paralelo  #pragma omp parallel for  for (int i = 1; i <= 100; i++) {  sum += i;  }   // Imprime o somatório  printf("O somatório é %d\n", sum);   return 0; } |
| --- |

Esse código usa a diretiva parallel para dividir o loop em quatro partes, que serão executadas por quatro threads.

A arquitetura OpenMP é uma ferramenta poderosa que pode ser usada para acelerar o desempenho de programas sequenciais. Ela é uma escolha popular para a programação paralela em máquinas com memória compartilhada e é usada em uma ampla variedade de aplicações, incluindo processamento de dados, simulações científicas e análise de dados.

Aqui estão alguns pontos da arquitetura OpenMP que têm mais chance de cair em uma avaliação sobre o tema:

O paradigma de threads: a arquitetura OpenMP é baseada no paradigma de threads, no qual o código é executado por vários threads concorrentes.

As diretivas OpenMP: a arquitetura OpenMP fornece um conjunto de diretivas que permitem controlar a paralelização do código.

A inicialização da biblioteca OpenMP: para usar a arquitetura OpenMP, você precisa primeiro inicializar a biblioteca OpenMP. Isso é feito chamando a função omp\_set\_num\_threads().

A declaração de threads: para criar um thread, você pode usar a função omp\_thread\_init().

A inicialização de threads: para iniciar a execução de um thread, você pode usar a função omp\_thread\_start().

A sincronização de threads: para garantir que todos os threads tenham concluído uma operação antes de continuar, você pode usar as diretivas critical, atomic e barrier.

Além desses pontos, também é importante conhecer os conceitos de memória compartilhada e topoologia de rede. A memória compartilhada é uma arquitetura de memória em que todos os threads têm acesso a toda a memória. A topologia de rede é a estrutura de interconexão entre os processadores de uma máquina paralela.

“Aqui estão alguns exemplos de questões que podem ser feitas em uma avaliação sobre a arquitetura OpenMP:

Qual é o paradigma de programação usado pela OpenMP?

Quais são as principais diretivas OpenMP para paralelização de código?

Como você inicializa a biblioteca OpenMP?

Como você declara um thread?

Como você inicializa um thread?

Como você sincroniza threads?

“

Bard:

Qual é o paradigma de programação usado pela OpenMP?

A arquitetura OpenMP é baseada no paradigma de threads. Um thread é uma unidade de execução leve que pode ser executada concorrentemente com outros threads.

Quais são as principais diretivas OpenMP para paralelização de código?

As principais diretivas OpenMP para paralelização de código são:

parallel: inicia um bloco de código que será executado por vários threads.

for: paraleliza um loop.

sections: divide um bloco de código em várias seções, que podem ser executadas por diferentes threads.

taskwait: garante que todos os threads tenham concluído a execução da diretiva sections antes de continuar.

critical: garante que apenas um thread possa executar o código dentro da diretiva critical por vez.

atomic: garante que uma operação seja executada de forma atômica.

Como você inicializa a biblioteca OpenMP?

Para inicializar a biblioteca OpenMP, você pode usar a função omp\_set\_num\_threads(). Essa função especifica o número de threads que serão usados para executar o código OpenMP.

| #include <stdio.h> #include <omp.h>  int main() {  // Inicializa a biblioteca OpenMP  omp\_set\_num\_threads(4);   // ...   return 0; } |
| --- |

Como você declara um thread?

Para declarar um thread, você pode usar a função omp\_thread\_init(). Essa função cria um novo thread e retorna um ponteiro para o thread.

| #include <stdio.h> #include <omp.h>  int main() {  // Declara um thread  omp\_thread\_t thread;   // ...   // Inicializa o thread  omp\_thread\_init(&thread, NULL, NULL);   // ...   // Finaliza o thread  omp\_thread\_destroy(&thread);   return 0; } |
| --- |

Como você inicializa um thread?

Para inicializar um thread, você pode usar a função omp\_thread\_start(). Essa função inicia a execução do thread.

| #include <stdio.h> #include <omp.h> int main() {  // Declara um thread  omp\_thread\_t thread;   // ...   // Inicializa o thread  omp\_thread\_init(&thread, NULL, NULL);   // Inicia o thread  omp\_thread\_start(&thread, NULL, NULL);   // ...   // Finaliza o thread  omp\_thread\_destroy(&thread);   return 0; } |
| --- |

Como você sincroniza threads?

Para sincronizar threads, você pode usar as seguintes diretivas:

critical: garante que apenas um thread possa executar o código dentro da diretiva critical por vez.

atomic: garante que uma operação seja executada de forma atômica.

barrier: garante que todos os threads tenham concluído a execução da diretiva barrier antes de continuar.

Aqui está um exemplo de como usar a diretiva barrier:

| #include <stdio.h> #include <omp.h>  int main() {  // Declara um bloco de código que será executado por vários threads  #pragma omp parallel  {  // ...   // Sincroniza os threads  #pragma omp barrier   // ...  }   return 0; } |
| --- |

“Descreva a Aplicações paralelas/distribuídas - OPenMP; MPI como se estivesse explicando para um aluno do curso de engenharia de software”

Bard:

Olá, aluno de engenharia de software. Hoje vamos falar sobre aplicações paralelas e distribuídas.

Aplicações paralelas são aquelas que podem ser executadas em paralelo, ou seja, por vários processadores ao mesmo tempo. Aplicações distribuídas são aquelas que podem ser executadas em várias máquinas, cada uma com seus próprios processadores.

Para implementar aplicações paralelas e distribuídas, precisamos de uma forma de coordenar a execução dos processos. Essa coordenação é feita por meio de um paradigma de programação paralela.

Existem dois principais paradigmas de programação paralela:

Passagem de mensagens: cada processo é responsável por suas próprias tarefas e troca mensagens com outros processos apenas quando necessário.

Threads: vários processos são executados simultaneamente no mesmo processador.

O MPI (Message Passing Interface) é um padrão de interface para a comunicação entre processos em máquinas paralelas com memória distribuída. Ele é baseado no paradigma de passagem de mensagens.

O OpenMP é um padrão de programação paralela que permite a paralelização de código sequencial em máquinas com memória compartilhada. Ele é baseado no paradigma de threads.

MPI

O MPI fornece um conjunto de rotinas para a troca de mensagens entre processos. Essas rotinas permitem que os processos enviem e recebam mensagens, bem como que sincronizem suas ações.

Para usar o MPI, você precisa primeiro inicializar o ambiente MPI. Em seguida, você pode criar processos MPI e iniciar a comunicação entre eles.

Aqui está um exemplo simples de código MPI:

| #include <mpi.h>  int main(int argc, char\*\* argv) {  // Inicializa o ambiente MPI  MPI\_Init(&argc, &argv);   // Cria dois processos MPI  int rank, size;  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);   // Envia uma mensagem do processo 0 para o processo 1  if (rank == 0) {  int data = 10;  MPI\_Send(&data, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);  }   // Recebe uma mensagem do processo 0  if (rank == 1) {  int data;  MPI\_Recv(&data, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);  printf("Recebi o valor %d do processo 0\n", data);  }   // Finaliza o ambiente MPI  MPI\_Finalize();   return 0; } |
| --- |

Esse código cria dois processos MPI. O processo 0 envia o valor 10 para o processo 1. O processo 1 recebe o valor 10 e o imprime na tela.

OpenMP

O OpenMP fornece um conjunto de diretivas que permitem controlar a paralelização do código. Essas diretivas são inseridas no código fonte do programa.

Aqui está um exemplo simples de código OpenMP:

| #include <stdio.h> #include <omp.h>  int main() {  // Declara uma variável para armazenar o somatório  int sum = 0;   // Percorre os números de 1 a 100 em paralelo  #pragma omp parallel for  for (int i = 1; i <= 100; i++) {  sum += i;  }   // Imprime o somatório  printf("O somatório é %d\n", sum);   return 0; } |
| --- |

Esse código usa a diretiva parallel para dividir o loop em várias partes, que serão executadas por vários threads.

Aplicações paralelas e distribuídas em engenharia de software

Aplicações paralelas e distribuídas são cada vez mais importantes em engenharia de software. Elas são usadas em uma ampla variedade de aplicações, incluindo:

Simulações científicas: simulações de fenômenos físicos, químicos ou biológicos.

Processamento de dados: análise de grandes conjuntos de dados.

Computação de alto desempenho: aplicações que exigem um alto desempenho computacional.

Para implementar aplicações paralelas e distribuídas em engenharia de software, é importante ter um conhecimento de paradigmas de programação paralela e das ferramentas disponíveis.

**Slide MPI Comunicação coletiva**

* Definição: Operações coletivas são chamadas por processos em um comunicador MPI (Message Passing Interface).
* MPI\_Bcast: Distribui dados de um processo (raiz) para todos os demais no comunicador.
* MPI\_Reduce: Combina dados de todos os processos do comunicador e retorna-os para um processo.
* Melhoria de Algoritmos: Em muitos algoritmos numéricos, trocar operações SEND/RECV por BCAST/REDUCE pode melhorar simplicidade e eficiência.
* Comunicação Coletiva no MPI:
  + Definição: Refere-se à comunicação e à computação coordenada entre um grupo de processos em um comunicador MPI.
  + Tags e Comunicadores: Não são usadas tags; comunicadores diferentes podem ter funcionalidades similares.
  + Características Gerais: Geralmente, as operações coletivas são não bloqueantes e podem ser divididas em três tipos: sincronização, movimento dos dados e computação coletiva.
* Sincronização:
  + MPI\_Barrier(comunicador):
    - Bloqueia até que todos os processos no grupo do comunicador chamem essa primitiva.
    - Um processo não sai da barreira até que todos tenham alcançado a barreira.
    - Exemplo: Elaborar um programa MPI para imprimir o rank do processo em ordem crescente usando MPI\_Barrier.
* Computação Coletiva:
  + Exemplo com MPI\_Scatter:
    - Programa para calcular e imprimir a média dos valores dos ranks dos processos usando MPI\_Reduce.
  + Exemplo com MPI\_Gather:
    - Programa para calcular colaborativamente a média dos valores de um vetor de 8 posições.
    - Usa MPI\_Scatter para dividir posições para cada processo e MPI\_Gather para recolher as parciais.
* Operações Válidas:
  + Algumas operações incluem MPI\_MAX, MPI\_MIN, MPI\_PROD, MPI\_SUM, MPI\_LAND, MPI\_LOR, MPI\_LXOR, MPI\_BAND, MPI\_BOR, etc.
  + Exemplo: Programa MPI para calcular e imprimir a média dos valores dos ranks dos processos usando MPI\_Reduce.

**Exercícios de fixação - OpenMP**

1. O programa abaixo está correto? Caso esteja errado, o que precisa ser modificado para que ele consiga processar corretamente?

O programa está correto, mas pode ser melhorado para melhorar, para melhorar o desempenho do programa, podemos remover a diretiva private(nthreads). Isso ocorre porque a variável nthreads é compartilhada por todas as threads, portanto, não há necessidade de declarar ela como privada

| #include <stdio.h> #include <omp.h> int main(void) {  int myid;  #pragma omp parallel  {  myid = omp\_get\_thread\_num();  int nthreads = omp\_get\_num\_threads();  printf("%d de %d - Hello, World!\n", myid, nthreads);  }  return 0; } /\* fim-main \*/ |
| --- |

1. Supondo OMP\_NUM\_THREADS=4, a execução do código abaixo produzirá impressão da variável i com um valor sempre maior do que 10. Faça testes para conferir se esta afirmação está correta ou não.

A afirmação está correta. A variável i é declarada como compartilhada, o que significa que todas as threads terão acesso à mesma cópia da variável. No código, cada thread incrementa a variável i em 10. Portanto, o valor final da variável i será sempre maior do que 10, independentemente do número de threads.

1. Elaborar um programa para descobrir o número de iterações que cada thread consegue processar em uma aplicação OpenMP. Por exemplo, imagine uma operação qualquer sobre os elementos de um vetor e descubra quantos elementos cada thread manipulou.

| #include <stdio.h> #include <omp.h>  int main(void) {  int n, i, myid, iter\_count;  int\* vec;   // Le o tamanho do vetor  printf("Digite o tamanho do vetor: ");  scanf("%d", &n);   // Aloca memória para o vetor  vec = (int\*)malloc(n \* sizeof(int));   // Inicializa o vetor com números randômicos  for (i = 0; i < n; i++) {  vec[i] = rand() % 100;  }   // Inicializa a variável de controle  iter\_count = 0;   // Inicializa a região paralela  #pragma omp parallel shared(vec, iter\_count)  {  // Obtém o número da thread atual  myid = omp\_get\_thread\_num();   // Executa a operação desejada  for (i = 0; i < n; i++) {  // Incrementa o contador de iterações  iter\_count++;  }  }   // Imprime o número de iterações de cada thread  for (i = 0; i < omp\_get\_num\_threads(); i++) {  printf("Thread %d: %d iterações\n", i, iter\_count);  }   return 0; } |
| --- |

1. O programa a seguir contabiliza a quantidade de números primos dentro de uma faixa de valores. Faça a medição do tempo de resposta deste programa (tempo de processamento) desta versão não-paralela. Em seguida, promova as alterações necessárias para que este programa funcione em paralelo e descubra qual quantidade de threads produz o melhor desempenho para essa aplicação (speedup).

Para medir o tempo de resposta do programa não-paralelo, podemos utilizar o seguinte código:

| #include <stdio.h> #include <time.h>  int main(void) {  int i;  int primos=0;  struct timespec start, end;   // Inicializa o temporizador  clock\_gettime(CLOCK\_REALTIME, &start);   // Executa o programa não-paralelo  for (i=N; i>1; i--) {  int j=2;  while (j<i) {  if (i%j == 0)  break;  j++;  } /\* fim-while \*/  if (j==i)  primos++;  } /\* fim-for \*/   // Obtém o tempo de execução  clock\_gettime(CLOCK\_REALTIME, &end);   // Imprime o tempo de execução  printf("Tempo de execução: %f segundos\n",  (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / 1e9);   // Imprime o número de números primos  printf("%d primos!\n", primos);   return 0; } /\*fim-main \*/ |
| --- |

Para medir o tempo de resposta do programa paralelo, podemos utilizar o seguinte código:

| #include <stdio.h> #include <omp.h> #include <time.h>  #define N 300000  int main(void) {  int i;  int primos=0;  struct timespec start, end;   // Inicializa o temporizador  clock\_gettime(CLOCK\_REALTIME, &start);   // Executa o programa paralelo  #pragma omp parallel for  for (i=N; i>1; i--) {  int j=2;  while (j<i) {  if (i%j == 0)  break;  j++;  } /\* fim-while \*/  if (j==i)  primos++;  } /\* fim-for \*/   // Obtém o tempo de execução  clock\_gettime(CLOCK\_REALTIME, &end);   // Imprime o tempo de execução  printf("Tempo de execução: %f segundos\n",  (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / 1e9);   // Imprime o número de números primos  printf("%d primos!\n", primos);   return 0; } /\*fim-main \*/ |
| --- |