Matemática Numérica Paralela

Pedro H A Konzen

4 de maio de 2021

Licença

Prefácio

Nestas notas de aula são abordados tópicos sobre computação paralela aplicada a métodos numéricos. Como ferramentas computacionais de apoio, exploramos exemplos de códigos em C/C++ usando as interfaces de programação de aplicações OpenMP, OpenMPI e o pacote de computação científica GSL.

Agradeço a todos e todas que de modo assíduo ou esporádico contribuem com correções, sugestões e críticas. :)

Pedro H A Konzen

Sumário

Capa i

Licença ii

Prefácio iii

Sumário v

1 Introdução 1

2 Multiprocessamento (MP) 4
2.1 Olá, Mundo! . . . 4
2.2 Construtores básicos . . . 9
2.2.1 Variáveis privadas e variáveis compartilhadas . . . 9
2.2.2 Laço e Redução . . . 10
2.2.3 Sincronização . . . 15
2.3 Resolução de Sistema Linear Triangular . . . 24
2.4 Decomposição LU . . . 30
2.5 Métodos iterativos para Sistemas Lineares . . . 36
2.5.1 Método de Jacobi . . . 37
2.5.2 Método *tipo* Gauss-Seidel . . . 40
2.5.3 Método do Gradiente Conjugado . . . 44
2.6 Métodos iterativos para problemas não lineares . . . 53
2.6.1 Método de Newton . . . 54
2.6.2 Método do acorde . . . 54
2.6.3 Métodos *quasi*-Newton . . . 55

3 Computação paralela e distribuída (MPI) 58

3.1 Olá, Mundo! 58
3.2 Rotinas de comunicação ponto-a-ponto 63
3.2.1 Envio e recebimento síncronos 63
3.2.2 Envio e recebimento assíncrono 67
3.3 Comunicações coletivas 71
3.3.1 Barreira de sincronização 72
3.3.2 Transmissão coletiva 74
3.3.3 Distribuição coletiva de dados 76
3.3.4 Recebimento coletivo de dados distribuídos 78
3.4 Reduções 82
3.5 Aplicação: Equação do calor 87

Respostas dos Exercícios **95**

Referências Bibliográficas **96**

Capítulo 1

Introdução

A computação paralela e distribuída é uma realidade em todas as áreas de pesquisa aplicadas. À primeira vista, pode-se esperar que as aplicações se beneficiam diretamente do ganho em poder computacional. Afinal, se a carga (processo) computacional de uma aplicação for repartida e distribuída em $n_p > 1$ processadores (**instâncias de processamentos**, *threads* ou *cores*), a computação paralela deve ocorrer em um tempo menor do que se a aplicação fosse computada em um único processador (em serial). Entretanto, a tarefa de repartir e distribuir (**alocação de tarefas**) o processo computacional de uma aplicação é, em muitos casos, bastante desafiadora e pode, em vários casos, levar a códigos computacionais menos eficientes que suas versões seriais.

Repartir e distribuir o processo computacional de uma aplicação sempre é possível, mas nem sempre é possível a computação paralela de cada uma das partes. Por exemplo, vamos considerar a iteração de ponto fixo

$$x(n) = f(x(n-1)), \quad n \geq 1, \tag{1.1}$$

$$x(0) = x_0, \tag{1.2}$$

onde $f : x \mapsto f(x)$ é uma função dada e x_0 é o ponto inicial da iteração. Para computar $x(100)$ devemos processar 100 vezes a iteração (1.1). Se tivéssemos a disposição $n_P = 2$ processadores, poderíamos repartir a carga de processamento em dois, distribuindo o processamento das 50 primeiras iterações para o primeiro processador (o processador 0) e as demais 50 para

o segundo processador (o processador 1). Entretanto, pela característica do processo iterativa, o processador 1 ficaria ocioso, aguardando o processador 0 computar $x(50)$. Se ambas instâncias de processamento compartilharem a mesma memória computacional (**memória compartilhada**), então, logo que o processador 0 computar $x(50)$ ele ficará ocioso, enquanto que o processador 1 computará as últimas 50 iterações. Ou seja, esta abordagem não permite a computação em paralelo, mesmo que reparta e distribua o processo computacional entre duas instâncias de processamento.

Ainda sobre a abordagem acima, caso as instâncias de processamento sejam de **memória distribuída** (não compartilhem a mesma memória), então o processador 0 e o processador 1 terão de se comunicar, isto é, o processador 0 deverá enviar $x(50)$ para a instância de processamento 1 e esta instância deverá receber $x(50)$ para, então, iniciar suas computações. A **comunicação** entre as instâncias de processamento levantam outro desafio que é necessidade ou não da **sincronização** () eventual entre elas. No caso de nosso exemplo, é a necessidade de sincronização na computação de $x(50)$ que está minando a computação paralela.

Em resumo, o design de métodos numéricos paralelos deve levar em consideração a **alocação de tarefas**, a **comunicação** e a **sincronização** entre as instâncias de processamentos. Vamos voltar ao caso da iteração (1.1). Agora, vamos supor que $x = (x_0, x_1)$, $f : x \mapsto (f_0(x), f_1(x))$ e a condição inicial $x(0) = (x_0(0), x_1(0))$ é dada. No caso de termos duas instâncias de processamentos disponíveis, podemos computar as iterações em paralelo da seguinte forma. Iniciamos distribuindo x às duas instâncias de processamento 0 e 1. Em paralelo, a instância 0 computa $x_0(1) = f_0(x)$ e a instância 1 computa $x_1(1) = f_1(x)$. Para computar a nova iterada $x(2)$, a instância 0 precisa ter acesso a $x_1(1)$ e a instância 1 necessita de $x_0(1)$. Isto implica na sincronização das instâncias de processamentos, pois uma instância só consegui seguir a computação após a outra instância ter terminado a computação da mesma iteração. Agora, a comunicação entre as instâncias de processamento, depende da arquitetura do máquina. Se as instâncias de processamento compartilham a mesma memória (memória compartilhada), cada uma tem acesso direto ao resultado da outra. No caso de uma arquitetura de memória distribuída, ainda há a necessidade de instruções de comunicação entre as instância, i.e. a instância 0 precisa enviar $x_0(1)$ à instância 1, a qual precisa receber o valor enviado. A instância 1 precisa enviar $x_1(1)$ à instância 0, a qual precisa receber o valor enviado. O processo segue análogo para cada

iteração até a computação de $x(100)$.

A primeira parte destas notas de aula, restringe-se a implementação de métodos numéricos paralelos em uma arquitetura de memória compartilhada. Os exemplos computacionais são apresentados em linguagem C/C++ com a interface de programação de aplicações (API, *Application Programming Interface*) OpenMP. A segunda parte, dedica-se a implementação paralela em arquitetura de memória distribuída. Os códigos C/C++ são, então, construídos com a API OpenMPI.

Capítulo 2

Multiprocessamento (MP)

Neste capítulo, vamos estudar aplicações da computação paralela em arquitetura de memória compartilhada. Para tanto, vamos discutir código C/C++ com a API OpenMP.

2.1 Olá, Mundo!

A computação paralela com MP inicia-se por uma instância de processamento **thread master**. Todas as instâncias de processamento disponíveis (**threads**) leem e escrevem variáveis compartilhadas. A ramificação (*fork*) do processo entre os *threads* disponíveis é feita por instrução explícita no início de uma região paralela do código. Ao final da região paralela, todos os *threads* sincronizam-se (*join*) e o processo segue apenas com o *thread master*. Veja a Figura 2.1.

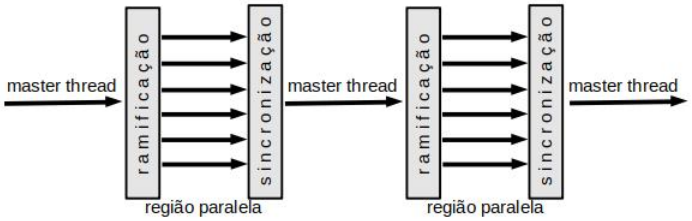

Figura 2.1: Fluxograma de um processo MP.

Vamos escrever nosso primeiro programa MP. O Código `ola.cc` inicia uma região paralela e cada instância de processamento escreve "Olá" e identifica-se.

Código: ola.cc

```
1  #include <stdio.h>
2
3  // OpenMP API
4  #include <omp.h>
5
6  using namespace std;
7
8  int main(int argc, char *argv[]) {
9
10   // região paralela
11   #pragma omp parallel
12   {
13     // id da instância de processamento
14     int id = omp_get_thread_num();
15
16     printf("Processo %d, olá!\n", id);
17   }
18
19   return 0;
20 }
```

Na linha 4, o API OpenMP é incluído no código. A região paralela vale dentro do escopo iniciado pela instrução

```
# pragma omp parallel
```

i.e., entre as linhas 12 e 17. Em paralelo, cada *thread* registra seu número de identificação na variável `id`, veja a linha 14. Na linha 16, escrevem a saudação, identificando-se.

Para compilar este código, digite no terminal

```
$ g++ -fopenmp ola.cc
```

Ao compilar, um executável `a.out` será criado. Para executá-lo, basta

digitar no terminal:

```
$ a.out
```

Ao executar, devemos ver a saída do terminal como algo parecido com[1]

```
Processo 0, olá!
Processo 3, olá!
Processo 1, olá!
Processo 2, olá!
```

A saída irá depender do número de *threads* disponíveis na máquina e a ordem dos *threads* pode variar a cada execução. Execute o código várias vezes e analise as saídas!

Observação 2.1.1. As variáveis declaradas dentro de uma região paralela são privadas de cada *threads*. As variáveis declaradas fora de uma região paralela são globais, sendo acessíveis por todos os *threads*.

Exercícios resolvidos

ER 2.1.1. O número de instâncias de processamento pode ser alterado pela variável do sistema `OMP_NUM_THREADS`. Altere o número de *threads* para 2 e execute o Código ola.cc.

Solução. Para alterar o número de *threads*, pode-se digitar no terminal

```
$ export OMP_NUM_THREADS=2
```

Caso já tenha compilado o código, não é necessário recompilá-lo. Basta executá-lo com

```
$ ./a.out
```

A saída deve ser algo do tipo

```
Olá, processo 0
Olá, processo 1
```

◊

ER 2.1.2. Escreva um código MP para ser executado com 2 *threads*. O *master thread* deve ler dois números em ponto flutuante. Então, em paralelo,

[1] O código foi rodado em uma máquina Quadcore com 4 *threads*.

um dos *threads* deve calcular a soma dos dois números e o outro thread deve calcular o produto.

Solução.

Código: sp.cc

```
1  #include <iostream>
2
3  // OpenMP API
4  #include <omp.h>
5
6  using namespace std;
7
8  int main(int argc, char *argv[]) {
9
10   double a,b;
11   printf("Digite o primeiro número: ");
12   scanf("%lf", &a);
13
14   printf("Digite o segundo número: ");
15   scanf("%lf", &b);
16
17   // região paralela
18 #pragma omp parallel
19   {
20     // id do processo
21     int id = omp_get_thread_num();
22
23     if (id == 0) {
24       printf("Soma: %f\n", (a+b));
25     }
26     else if (id == 1) {
27       printf("Produto: %f\n", (a*b));
28     }
29   }
30
31   return 0;
32 }
```

$\Diamond$

Exercícios

E 2.1.1. Defina um número de *threads* maior do que o disponível em sua máquina. Então, rode o código `ola.cc` e analise a saída. O que você observa?

E 2.1.2. Modifique o código `ola.cc` de forma que cada *thread* escreva na tela "Processo ID de NP, olá!", onde ID é a identificação do *thread* e NP é o número total de *threads* disponíveis. O número total de *threads* pode ser obtido com a função OpenMP

```
omp_get_num_threads();
```

E 2.1.3. Faça um código MP para ser executado com 2 threads. O *master thread* deve ler dois números a e b não nulos em ponto flutuante. Em paralelo, um dos *thread* de computar $a - b$ e o outro deve computar a/b. Por fim, o *master thread* deve escrever $(a - b) + (a/b)$.

E 2.1.4. Escreva um código MP para computar a multiplicação de uma matriz $n \times n$ com um vetor de n elementos. Inicialize todos os elementos com números randômicos em ponto flutuante. Ainda, o código deve ser escrito para um número arbitrário $m > 1$ de instâncias de processamento. Por fim, compare o desempenho do código MP com uma versão serial do código.

E 2.1.5. Escreva um código MP para computar o produto de uma matriz $n \times m$ com uma matriz de $m \times n$ elementos, com $n \geq m$. Inicialize todos os elementos com números randômicos em ponto flutuante. Ainda, o código deve ser escrito para um número arbitrário $m > 1$ de instâncias de processamento. Por fim, compare o desempenho do código MP com uma versão serial do código.

2.2 Construtores básicos

2.2.1 Variáveis privadas e variáveis compartilhadas

Vamos analisar o seguinte código.

Código: vpc.cc

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main(int argc, char *argv[]) {
5
6    int tid, nt;
7
8    // região paralela
9  #pragma omp parallel
10   {
11     tid = omp_get_thread_num();
12     nt = omp_get_num_threads();
13
14     printf("Processo %d/%d\n", tid, nt);
15   }
16   printf("%d\n",nt);
17   return 0;
18 }
```

Qual seria a saída esperada? Ao rodarmos este código, veremos uma saída da forma

```
Processo 0/4
Processo 2/4
Processo 3/4
Processo 3/4
```

Isto ocorre por uma situação de **condição de corrida** (**race condition**) entre os *threads*. As variáveis `tid` e `nt` foram declaradas antes da região paralela e, desta forma, são **variáveis compartilhadas** (**shared variables**) entre todos os *threads* na região paralela. Os locais na memória em que estas as variáveis estão alocadas é o mesmo para todos os *threads*.

A condição de corrida ocorre na linha 11. No caso da saída acima, as instâncias de processamento 1 e 3 entraram em uma condição de corrida no registro da variável `tid`.

Observação 2.2.1. Devemos estar sempre atentos a uma possível condição de corrida. Este é um erro comum no desenvolvimento de códigos em paralelo.

Para evitarmos a condição de corrida, precisamos tornar a variável `tid` privada na região paralela. I.e., cada *thread* precisa ter uma variável `tid` privada. Podemos fazer isso alterando a linha 9 do código para

```
#pragma omp parallel private(tid)
```

Com essa alteração, a saída terá o formato esperado, como por exemplo

```
Processo 0/4
Processo 3/4
Processo 2/4
Processo 1/4
```

Faça a alteração e verifique!

Observação 2.2.2. A diretiva `#pragma omp parallel` também aceita as instruções:

- `default(private|shared|none)`: o padrão é `shared`;
- `shared(var1, var2, ..., varn)`: para especificar explicitamente as variáveis que devem ser compartilhadas.

2.2.2 Laço e Redução

Vamos considerar o problema de computar

$$s = \sum_{i=0}^{99999999} 1 \tag{2.1}$$

em paralelo com *np threads*. Começamos analisando o seguinte código

Código: soma0.cc

```
1 #include <omp.h>
2 #include <stdio.h>
3 #include <math.h>
```

```
4
5  int main(int argc, char *argv[]) {
6
7    int n = 99999999;
8
9    int s = 0;
10   #pragma omp parallel
11   {
12     int tid = omp_get_thread_num();
13     int nt = omp_get_num_threads();
14
15     int ini = n/nt*tid;
16     int fin = n/nt*(tid+1);
17     if (tid == nt-1)
18       fin = n;
19     for (int i=ini; i<fin; i++)
20       s += 1;
21   }
22   printf("%d\n",s);
23   return 0;
24 }
```

Ao executarmos este código com $nt > 1$, vamos ter saídas erradas. Verifique! Qual o valor esperado?

O erro do código está na **condição de corrida** (*race condition*) na linha 20. Esta é uma operação, ao ser iniciada por um *thread*, precisa ser terminada pelo *thread* antes que outro possa iniciá-la. Podemos fazer adicionando o construtor

```
#pragma omp critical
```

imediatamente antes da linha de código `s += i;`. O código fica como segue, verifique!

Código: soma1.cc

```
1 #include <omp.h>
2 #include <stdio.h>
3 #include <math.h>
```

```
4
5  int main(int argc, char *argv[]) {
6
7    int n = 99999999;
8
9    int s = 0;
10   #pragma omp parallel
11   {
12     int tid = omp_get_thread_num();
13     int nt = omp_get_num_threads();
14
15     int ini = n/nt*tid;
16     int fin = n/nt*(tid+1);
17     if (tid == nt-1)
18       fin = n;
19     for (int i=ini; i<fin; i++)
20       #pragma omp critical
21       s += 1;
22   }
23   printf("%d\n",s);
24   return 0;
25 }
```

Esta abordagem evita a condição de corrida e fornece a resposta esperada. No entanto, ela acaba serializando o código, o qual é será muito mais lento que o código serial. Verifique!

Observação 2.2.3. A utilização do construtor

`#pragma omp critical`

reduz a performance do código e só deve ser usada quando realmente necessária.

Uma alternativa é alocar as somas parciais de cada *thread* em uma variável privada e, ao final, somar as partes computadas. Isto pode ser feito com o seguinte código. Verifique!

Código: soma2.cc

```
1  #include <omp.h>
```

```
2  #include <stdio.h>
3  #include <math.h>
4
5  int main(int argc, char *argv[]) {
6
7    int n = 99999999;
8
9    int s = 0;
10   #pragma omp parallel
11   {
12     int tid = omp_get_thread_num();
13     int nt = omp_get_num_threads();
14
15     int ini = n/nt*tid;
16     int fin = n/nt*(tid+1);
17     if (tid == nt-1)
18       fin = n;
19
20     int st = 0;
21     for (int i=ini; i<fin; i++)
22       st += 1;
23
24     #pragma omp critical
25     s += st;
26   }
27   printf("%d\n",s);
28   return 0;
29 }
```

Este último código pode ser simplificado usando o construtor

`#pragma omp for`

Com este construtor, o laço do somatório pode ser automaticamente distribuindo entre os *threads*. Verifique o seguinte código!

Código: somafor.cc

```
1  #include <omp.h>
2  #include <stdio.h>
```

```
3  #include <math.h>
4
5  int main(int argc, char *argv[]) {
6
7    int n = 99999999;
8
9    int s = 0;
10   #pragma omp parallel
11   {
12     int st = 0;
13
14     #pragma omp for
15     for (int i=0; i<n; i++)
16       st += 1;
17
18     #pragma omp critical
19     s += st;
20   }
21   printf("%d\n",s);
22   return 0;
23 }
```

Mais simples e otimizado, é automatizar a operação de redução (no caso, a soma das somas parciais) adicionado

```
reduction(+: s)
```

ao construtor que inicializa a região paralela. Verifique o seguinte código!

Código: soma.cc

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <math.h>
4
5  int main(int argc, char *argv[]) {
6
7    int n = 99999999;
8    int s = 0;
9
```

```
10    #pragma omp parallel for reduction(+: s)
11    for (int i=0; i<n; i++)
12      s += 1;
13
14    printf("%d\n",s);
15    return 0;
16  }
```

Observação 2.2.4. A instrução de redução pode ser usada com qualquer operação binária aritmética (`+`, `-`, `/`, `*`), lógica (`&`, `|`) ou procedimentos intrínsecos (`max`, `min`).

2.2.3 Sincronização

A sincronização dos *threads* deve ser evitada sempre que possível, devido a perda de performance em códigos paralelos. Atenção, ela ocorre implicitamente no término da região paralela!

Barreira

No seguinte código, o *thread* 1 é atrasado em 1 segundo, de forma que ele é o último a imprimir. Verifique!

Código: sinc0.cc

```
1  #include <stdio.h>
2  #include <ctime>
3  #include <omp.h>
4
5  int main(int argc, char *argv[]) {
6
7    // master thread id
8    int tid = 0;
9    int nt;
10
11   #pragma omp parallel private(tid)
12   {
13     tid = omp_get_thread_num();
14     nt = omp_get_num_threads();
```

```
15
16      if (tid == 1) {
17        // delay 1s
18        time_t t0 = time(NULL);
19        while (time(NULL) - t0 < 1) {
20        }
21      }
22
23      printf("Processo %d/%d.\n", tid, nt);
24    }
25    return 0;
26  }
```

Agora, podemos forçar a sincronização dos *threads* usando o construtor

```
#pragma omp barrier
```

em uma determinada linha do código. Por exemplo, podemos fazer todos os *threads* esperarem pelo *thread* 1 no código acima. Veja a seguir o código modificado. Teste!

Código: sinc1.cc

```
1  #include <stdio.h>
2  #include <ctime>
3  #include <omp.h>
4
5  int main(int argc, char *argv[]) {
6
7    // master thread id
8    int tid = 0;
9    int nt;
10
11   #pragma omp parallel private(tid)
12   {
13     tid = omp_get_thread_num();
14     nt = omp_get_num_threads();
15
16     if (tid == 1) {
17       // delay 1s
```

```
18        time_t t0 = time(NULL);
19        while (time(NULL) - t0 < 1) {
20        }
21      }
22
23      #pragma omp barrier
24
25      printf("Processo %d/%d.\n", tid, nt);
26    }
27    return 0;
28  }
```

Seção

O construtor `sections` pode ser usado para determinar seções do código que deve ser executada de forma serial apenas uma vez por um único *thread*. Verifique o seguinte código.

Código: secao.cc

```
1  #include <stdio.h>
2  #include <ctime>
3  #include <omp.h>
4
5  int main(int argc, char *argv[]) {
6
7    // master thread id
8    int tid = 0;
9    int nt;
10
11   #pragma omp parallel private(tid)
12   {
13     tid = omp_get_thread_num();
14     nt = omp_get_num_threads();
15
16     #pragma omp sections
17     {
18       // seção 1
19       #pragma omp section
```

```
20        {
21          printf("%d/%d exec seção 1\n", \
22                 tid, nt);
23        }
24
25        // seção 2
26        #pragma omp section
27        {
28          // delay 1s
29          time_t t0 = time(NULL);
30          while (time(NULL) - t0 < 1) {
31          }
32          printf("%d/%d exec a seção 2\n", \
33                 tid, nt);
34        }
35      }
36
37      printf("%d/%d terminou\n", tid, nt);
38    }
39
40    return 0;
41  }
```

No código acima, o primeiro *thread* que alcançar a linha 19 é o único a executar a seção 1 e, o primeiro que alcançar a linha 25 é o único a executar a seção 2.

Observe que ocorre a sincronização implícita de todos os *threads* ao final do escopo `sections`. Isso pode ser evitado usando a cláusula `nowait`, i.e. alterando a linha 16 para

```
# pragma omp sections nowait
```

Teste!

Observação 2.2.5. A clausula `nowait` também pode ser usada com o construtor `for`, i.e.

```
#pragma omp for nowait
```

Para uma região contendo apenas uma seção, pode-se usar o construtor

```
#pragma omp single
```

Isto é equivalente a escrever

```
#pragma omp sections
  #pragma omp section
```

Exercícios Resolvidos

ER 2.2.1. Escreva um código MP para computar o produto escalar entre dois vetores de n pontos flutuantes randômicos.

Solução. Aqui, vamos usar o suporte a vetores e números randômicos do pacote de computação cientifica GSL. A solução é dada no código a seguir.

Código: prodesc.cc

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <ctime>
4
5  // GSL vector suport
6  #include <gsl/gsl_vector.h>
7  #include <gsl/gsl_rng.h>
8
9  int main(int argc, char *argv[]) {
10
11   int n = 99999999;
12
13   // vetores
14   gsl_vector *a = gsl_vector_alloc(n);
15   gsl_vector *b = gsl_vector_alloc(n);
16
17   // gerador randômico
18   gsl_rng *rng = gsl_rng_alloc(gsl_rng_default);
19   gsl_rng_set(rng, time(NULL));
20
21   // inicializa os vetores
22   #pragma omp parallel for
```

```
23    for (int i=0; i<n; i++) {
24      gsl_vector_set(a, i, gsl_rng_uniform(rng));
25      gsl_vector_set(b, i, gsl_rng_uniform(rng));
26    }
27
28    // produto escalar
29    double dot = 0;
30    #pragma omp parallel for reduction(+: dot)
31    for (int i=0; i<n; i++)
32      dot += gsl_vector_get(a, i) * \
33        gsl_vector_get(b, i);
34
35    printf("%f\n",dot);
36
37    gsl_vector_free(a);
38    gsl_vector_free(b);
39    gsl_rng_free(rng);
40
41    return 0;
42  }
```

Para compilar o código acima, digite

```
$ g++ -fopenmp prodesc.cc -lgsl -lgslcblas
```

◊

ER 2.2.2. Faça um código MP para computar a multiplicação de uma matriz A $n \times n$ por um vetor de n elementos (pontos flutuantes randômicos). Utilize o construtor `omp sections` para distribuir a computação entre somente dois *threads*.

Solução. Vamos usar o suporte a matrizes, vetores, BLAS e números randômicos do pacote de computação científica GSL. A solução é dada no código a seguir.

Código: AxSecoes.cc

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <ctime>
```

```
4
5  #include <gsl/gsl_matrix.h>
6  #include <gsl/gsl_vector.h>
7  #include <gsl/gsl_rng.h>
8  #include <gsl/gsl_blas.h>
9
10 int main(int argc, char *argv[]) {
11
12   int n = 9999;
13
14   // vetores
15   gsl_matrix *a = gsl_matrix_alloc(n,n);
16   gsl_vector *x = gsl_vector_alloc(n);
17   gsl_vector *y = gsl_vector_alloc(n);
18
19   // gerador randômico
20   gsl_rng *rng = gsl_rng_alloc(gsl_rng_default);
21   gsl_rng_set(rng, time(NULL));
22
23   // inicialização
24   for (int i=0; i<n; i++) {
25     for (int j=0; j<n; j++) {
26       gsl_matrix_set(a, i, j, gsl_rng_uniform(rng));
27     }
28     gsl_vector_set(x, i, gsl_rng_uniform(rng));
29   }
30
31   //gsl_blas_dgemv(CblasNoTrans, 1.0, a, x, 0.0, y);
32
33   // y = A*x
34   #pragma omp parallel sections
35   {
36     #pragma omp section
37     {
38       gsl_matrix_const_view as1
39         = gsl_matrix_const_submatrix(a,
40                                      0,0,
41                                      n/2,n);
```

```
42        gsl_vector_view ys1
43          = gsl_vector_subvector(y,0,n/2);
44        gsl_blas_dgemv(CblasNoTrans,
45                       1.0, &as1.matrix, x,
46                       0.0, &ys1.vector);
47      }
48
49      #pragma omp section
50      {
51        gsl_matrix_const_view as2
52          = gsl_matrix_const_submatrix(a,
53                                       n/2,0,
54                                       (n-n/2),n);
55        gsl_vector_view ys2
56          = gsl_vector_subvector(y,n/2,(n-n/2));
57        gsl_blas_dgemv(CblasNoTrans,
58                       1.0, &as2.matrix, x,
59                       0.0, &ys2.vector);
60      }
61    }
62
63    //for (int i=0; i<n; i++)
64    //printf("%f\n", gsl_vector_get(y,i));
65
66    gsl_matrix_free(a);
67    gsl_vector_free(x);
68    gsl_vector_free(y);
69    gsl_rng_free(rng);
70
71    return 0;
72  }
```

◊

Exercícios

E 2.2.1. Considere o seguinte código

```
1     int tid = 10;
2     #pragma omp parallel private(tid)
3     {
4       tid = omp_get_thread_num();
5     }
6     printf("%d\n", tid);
```

Qual o valor impresso?

E 2.2.2. Escreva um código MP para computar uma aproximação para

$$I = \int_{-1}^{1} e^{-x^2}\, dx \tag{2.2}$$

usando a regra composta do trapézio com n subintervalos uniformes.

E 2.2.3. Escreva um código MP para computar uma aproximação para

$$I = \int_{-1}^{1} e^{-x^2}\, dx \tag{2.3}$$

usando a regra composta de Simpson com n subintervalos uniformes. Dica: evite sincronizações desnecessárias!

E 2.2.4. Escreva um código MP para computar a multiplicação de uma matriz A $n \times n$ por um vetor x de n elementos (pontos flutuantes randômicos). Faça o código de forma a suportar uma arquitetura com $n_p \geq 1$ *threads*.

E 2.2.5. Escreva um código MP para computar o produto de duas matrizes $n \times n$ de pontos flutuantes randômicos. Utilize o construtor `omp sections` para distribuir a computação entre somente dois *threads*.

E 2.2.6. Escreva um código MP para computar o produto de duas matrizes $n \times n$ de pontos flutuantes randômicos. Faça o código de forma a suportar uma arquitetura com $n_p \geq 1$ *threads*.

2.3 Resolução de Sistema Linear Triangular

Nesta seção, vamos discutir sobre a uma implementação em paralelo do método da substituição para a resolução de sistemas triangulares. Primeiramente, vamos considerar A uma matriz triangular inferior quadrada de dimensões $n \times n$, i.e. $A = [a_{i,j}]_{i,j=0}^{n-1}$ com $a_{i,j} = 0$ para $i < j$. Ainda, vamos considerar que A é invertível.

Neste caso, um sistema linear $Ax = b$ pode ser escrito na seguinte forma algébrica

$$a_{1,1}x_1 = b_1 \tag{2.4}$$
$$\vdots \tag{2.5}$$
$$a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,i-1}x_{i-1} + a_{i,i}x_i = b_i \tag{2.6}$$
$$\vdots \tag{2.7}$$
$$a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{i,i}x_i + \cdots + a_{n,n}x_n = b_n \tag{2.8}$$

O algoritmo serial do método da substituição (para frente) resolve o sistema começando pelo cálculo de x_1 na primeira equação, então o cálculo de x_2 pela segunda equação e assim por diante até o cálculo de x_n pela última equação. Segue o pseudocódigo serial.

1. Para $i = 0, \ldots, n-1$:
   (a) Para $j = 0, \ldots, i-1$:
      i. $b_i = b_i - A_{i,j}x_j$
   (b) $x_i = \dfrac{b_i}{A_{i,i}}$

Implemente!

Para o algoritmo paralelo, vamos considerar uma arquitetura MP com $n_p \geq 1$ instâncias de processamento. Para cada instância de processamento

$1 \leq p_{id} < n_p - 1$ vamos alocar as seguintes colunas da matriz A

$$t_{ini} = p_{id} \left\lfloor \frac{n}{n_p} \right\rfloor \tag{2.9}$$

$$t_{fim} = (p_{id} + 1) \left\lfloor \frac{n}{n_p} \right\rfloor - 1 \tag{2.10}$$

e, para $p_{id} = n_p - 1$ vamos alocar as últimas colunas, i.e.

$$t_{ini} = p_{id} \left\lfloor \frac{n}{n_p} \right\rfloor \tag{2.11}$$

$$t_{fim} = n - 1 \tag{2.12}$$

Segue o pseudocódigo em paralelo.

1. Para $i = 0, \ldots, n-1$
   (a) $s = 0$
   (b) Região paralela
      i. Para $j \in \{t_{ini}, \ldots, t_{fim}\} \wedge \{0, \ldots, i-1\}$
         A. $s = s + a_{i,j} x_j$
   (c) $x_i = \dfrac{b_i - s}{a_{i,i}}$

O código MP C/C++ que apresentaremos a seguir, faz uso do construtor `threadprivate`

```
#pragma omp threadprivate(list)
```

Este construtor permite que a lista de variáveis (estáticas) `list` seja privada para cada *thread* e seja compartilhada entre as regiões paralelas. Por exemplo:

```
x = 0
#pragma omp parallel private(x)
  x = 1
#pragma omp parallel private(x)
  x vale 0
```

Agora, com o construtor `threadprivate`:

```
static x = 0
#pragma omp threadprivate(x)
#pragma omp parallel
  x = 1
#pragma omp parallel private(x)
  x vale 1
```

Ainda, apenas para efeito de exemplo, vamos considerar que $a_{i,j} = (-1)^{i+j}(i+j)/(ij+1)$ para $i < j$, $a_{i,i} = 2[(i-n/2)^2+1]/n$ e $b_i = (-1)^i/(i+1)$ para $i = 0, \ldots, n-1$.

Segue o código paralelo para a resolução direta do sistema triangular inferior. Verifique!

Código: sistria1dcol.cc

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <ctime>
4  #include <algorithm>
5
6  #include <gsl/gsl_spmatrix.h>
7  #include <gsl/gsl_vector.h>
8  #include <gsl/gsl_rng.h>
9
10 int np, pid;
11 int ini, fim;
12 #pragma omp threadprivate(np,pid,ini,fim)
13
14 int main(int argc, char *argv[]) {
15
16   int n = 9999;
17
18   // vetores
19   gsl_spmatrix *a = gsl_spmatrix_alloc(n,n);
20   gsl_vector *b = gsl_vector_alloc(n);
21   gsl_vector *x = gsl_vector_alloc(n);
22
23   // inicialização
24   printf("Inicializando ... \n");
```

```
25
26    for (int i=0; i<n; i++) {
27      for (int j=0; j<i; j++) {
28        gsl_spmatrix_set(a, i, j,
29                         pow(-1.0,i+j)*(i+j)/(i*j+1));
30      }
31      gsl_spmatrix_set(a, i, i,
32                       (pow(i-n/2,2)+1)*2/n);
33      gsl_vector_set(b, i,
34                     pow(-1.0,i)/(i+1));
35    }
36
37    printf("feito.\n");
38
39    printf("Executando em paralelo ... \n");
40
41    time_t t = time(NULL);
42    #pragma omp parallel
43    {
44      np = omp_get_num_threads();
45      pid = omp_get_thread_num();
46
47      ini = pid*n/np;
48      fim = (pid+1)*n/np;
49      if (pid == np-1)
50        fim = n;
51    }
52
53    for (int i=0; i<n; i++) {
54      double s = 0;
55      #pragma omp parallel reduction(+: s)
56      {
57        for (int j=std::max(0,ini); j<i and j<fim; j++)
58          s += gsl_spmatrix_get(a,i,j) *
59               gsl_vector_get(x,j);
60      }
61      gsl_vector_set(x, i,
62                     (gsl_vector_get(b,i) - s) /
```

```
63                      gsl_spmatrix_get(a,i,i));
64    }
65
66    t = time(NULL)-t;
67
68    printf("feito. %ld s\n", t);
69
70
71    gsl_spmatrix_free(a);
72    gsl_vector_free(b);
73    gsl_vector_free(x);
74
75    return 0;
76  }
```

Exercícios resolvidos

ER 2.3.1. Seja $Ax = b$ um sistema triangular inferior de dimensões $n \times n$. O seguinte pseudocódigo paralelo é uma alternativa ao apresentado acima. Por que este pseudocódigo é mais lento que o anterior?

1. Região paralela
   (a) Para $j = 0, \ldots, n-1$
      i. Se $j \in \{t_{ini}, \ldots, t_{fim}\}$
         A. $x_j = \dfrac{b_j}{a_{j,j}}$
      ii. Para $i \in \{t_{ini}, \ldots, t_{fim}\} \wedge \{j+1, \ldots, n-1\}$
         A. $b_i = b_i - a_{i,j}x_j$

Solução. Este código tem um número de operações semelhante ao anterior, seu desempenho é afetado pelo chamado compartilhamento falso (*false sharing*). Este é um fenômeno relacionado ao uso ineficiente da memória *cache* de cada *thread*. O último laço deste pseudocódigo faz sucessivas atualizações do vetor b, o que causa sucessivos recarregamentos de partes do vetor b da memória RAM para a memória *cache* de cada um dos *threads*. Verifique!

$\Diamond$

ER 2.3.2. Seja A uma matriz triangular inferior e invertível de dimensões $n \times n$. Escreva um pseudocódigo MP para calcular a matriz inversa A^{-1} usando o método de substituição direta.

Solução. Vamos denotar $A = [a_{i,j}]_{i,j=1}^{n-1}$ e $A^{-1} = [x_{i,j}]_{i,j=1}^{n-1}$. Note que $x's$ são as incógnitas. Por definição, $AA^{-1} = I$, logo

$$a_{1,1}x_{1,k} = \delta_{1,k} \tag{2.13}$$
$$\cdots \tag{2.14}$$
$$a_{i,1}x_{1,k} + \cdots + a_{i,i-1}x_{i-1,k} + a_{i,i}x_{i,k} = \delta_{i,k} \tag{2.15}$$
$$\cdots \tag{2.16}$$
$$a_{n-1,1}x_{1,k} + \cdots + a_{n-1,n-1}x_{n-1,k} = \delta_{n-1,k} \tag{2.17}$$

onde, $k = 0, \ldots, n-1$ e $\delta_{i,j}$ denota o Delta de Kronecker. Ou seja, o cálculo de A^{-1} pode ser feito pela resolução de n sistemas triangulares inferiores tendo A como matriz de seus coeficientes.

Para construirmos um pseudocódigo MP, podemos distribuir os sistemas lineares a entre os *threads* disponíveis. Então, cada *thread* resolve em serial seus sistemas. Segue o pseudocódigo, sendo $x_k = (x_{1,k}, \ldots, x_{n-1,k})$ e $b_k = (\delta_{1,k}, \ldots, \delta_{n-1,k})$.

1. Região paralela
   (a) Para $k \in \{t_{ini}, \ldots, t_{fim}\}$
      i. resolve $Ax_k = b_k$

$\Diamond$

Exercícios

E 2.3.1. Implemente um código MP do pseudocódigo discutido no ER 2.3.1. Compare o tempo computacional com o do código `sistria1dcol.cc`.

E 2.3.2. Implemente um código MP para computar a inversa de uma matriz triangular inferior de dimensões $n \times n$.

E 2.3.3. Implemente um código MP para computar a solução de um sistema linear triangular superior de dimensões $n \times n$.

E 2.3.4. Implemente um código MP para computar a inversa de uma matriz triangular superior de dimensões $n \times n$.

2.4 Decomposição LU

Nesta seção, vamos discutir sobre a paralelização da decomposição LU para matrizes. A decomposição LU de uma matriz A com dimensões $n \times n$ é

$$A = LU \tag{2.18}$$

onde L é uma matriz triangular inferior e U é uma matriz triangular superior, ambas com dimensões $n \times n$.

Para fixar as ideais, vamos denotar $A = [a_{i,j}]_{i,j=0}^{n-1}$, $L = [l_{i,j}]_{i,j=0}^{n-1}$ sendo $l_{i,i} = 1$ e $l_{i,j} = 0$ para $i > j$, e $U = [u_{i,j}]_{i,j=0}^{n}$ sendo $u_{i,j} = 0$ para $i < j$. O pseudoalgoritmo serial para computar a decomposição LU é

1. U = A, L = I
2. Para $k = 0, \ldots, n-2$
   (a) Para $i = k+1, \ldots, n-1$
      i. $l_{i,k} = u_{i,k}/u_{k,k}$
      ii. Para $j = k, \ldots, n-1$
         A. $u_{i,j} = u_{i,j} - l_{i,k}u_{k,j}$

A forma mais fácil de paralelizar este algoritmo em uma arquitetura MP é paralelizando um de seus laços (itens 2., 2.(a) ou 2.(a)ii.). O laço do item 2. não é paralelizável, pois a iteração seguinte depende do resultado da iteração imediatamente anterior. Agora, os dois laços seguintes são paralelizáveis. Desta forma, o mais eficiente é paralelizarmos o segundo laço 2.(a).

O seguinte código é uma versão paralela da decomposição LU. A matriz A é inicializada como uma matriz simétrica de elementos randômicos (linhas 19-41), sendo que a decomposição é computada nas linhas 43-61.

Código: parallelLU.cc

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <ctime>
4  #include <algorithm>
5
6  #include <gsl/gsl_matrix.h>
7  #include <gsl/gsl_vector.h>
8  #include <gsl/gsl_rng.h>
9  #include <gsl/gsl_blas.h>
10
11 int main(int argc, char *argv[]) {
12
13   int n = 5;
14
15   gsl_matrix *a = gsl_matrix_alloc(n,n);
16   gsl_matrix *u = gsl_matrix_alloc(n,n);
17   gsl_matrix *l = gsl_matrix_alloc(n,n);
18
19   // gerador randômico
20   gsl_rng *rng = gsl_rng_alloc(gsl_rng_default);
21   gsl_rng_set(rng, time(NULL));
22
23   // inicialização
24   printf("Inicializando ... \n");
25   for (int i=0; i<n; i++) {
26     for (int j=0; j<i; j++) {
27       int sig = 1;
28       if (gsl_rng_uniform(rng) >= 0.5)
29         sig = -1;
30       gsl_matrix_set(a, i, j,
31                      sig*gsl_rng_uniform(rng));
32       gsl_matrix_set(a, j, i,
33                      gsl_matrix_get(a, i, j));
34     }
35     int sig = 1;
36     if (gsl_rng_uniform(rng) >= 0.5)
```

```
37        sig = -1;
38      gsl_matrix_set(a, i, i,
39                     sig*gsl_rng_uniform_pos(rng));
40    }
41    printf("feito.\n");
42
43    // U = A
44    gsl_matrix_memcpy(u,a);
45    // L = I
46    gsl_matrix_set_identity(l);
47
48    for (int k=0; k<n-1; k++) {
49      #pragma omp parallel for
50      for (int i=k+1; i<n; i++) {
51        gsl_matrix_set(l, i, k,
52                       gsl_matrix_get(u, i, k)/
53                       gsl_matrix_get(u, k, k));
54        for (int j=k; j<n; j++) {
55          gsl_matrix_set(u, i, j,
56                         gsl_matrix_get(u, i, j) -
57                         gsl_matrix_get(l, i, k) *
58                         gsl_matrix_get(u, k, j));
59        }
60      }
61    }
62
63    gsl_matrix_free(a);
64    gsl_matrix_free(u);
65    gsl_matrix_free(l);
66    gsl_rng_free(rng);
67
68    return 0;
69  }
```

Exercícios Resolvidos

ER 2.4.1. Faça um código MP para computar a solução de um sistema linear $Ax = b$ usando a decomposição LU. Assuma A uma matriz simétrica $n \times n$ de elementos randômicos, assim como os elementos do vetor b.

Solução. A decomposição LU da matriz A nos fornece as matrizes L (matriz triangular inferior) e U (matriz triangular superior), com

$$A = LU \tag{2.19}$$

Logo, temos

$$Ax = b \tag{2.20}$$
$$\Rightarrow (LU)x = b \tag{2.21}$$
$$\Rightarrow L(Ux) = b \tag{2.22}$$

Denotando $Ux = y$, temos que y é solução do sistema triangular inferior

$$Ly = b \tag{2.23}$$

e, por conseguinte, x é solução do sistema triangular superior

$$Ux = y. \tag{2.24}$$

Em síntese, o sistema $Ax = b$ pode ser resolvido com o seguinte pseudocódigo:

1. Computar a decomposição LU, A=LU.
2. Resolver $Ly = b$.
3. Resolver $Ux = b$.

Cada passo acima pode ser paralelizado. O código MP fica de exercício, veja E 2.4.1.

$\Diamond$

ER 2.4.2. Considere a decomposição LU de uma matriz $A\, n \times n$. Em muitas aplicações, não há necessidade de guardar a matriz A em memória após a decomposição. Além disso, fixando-se que a diagonal da matriz L tem todos os elementos iguais a 1, podemos alocar seus elementos não nulos na parte

triangular inferior (abaixo da diagonal) da própria matriz A. E, a matriz U pode ser alocada na parte triangular superior da matriz A. Faça um código MP para computar a decomposição LU de uma matriz A, alocando o resultado na própria matriz A.

Solução. O seguinte código faz a implementação pedida. Neste código, é necessário alocar apenas a matriz A, sem necessidade de locar as matrizes L e U. Da linha 17 à 39, apenas é gerada a matriz randômica A. A decomposição é computada da linha 41 a 54.

Código: parallelLU2.cc

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <ctime>
4  #include <algorithm>
5
6  #include <gsl/gsl_matrix.h>
7  #include <gsl/gsl_vector.h>
8  #include <gsl/gsl_rng.h>
9  #include <gsl/gsl_blas.h>
10
11 int main(int argc, char *argv[]) {
12
13   int n = 5;
14
15   gsl_matrix *a = gsl_matrix_alloc(n,n);
16
17   // gerador randômico
18   gsl_rng *rng = gsl_rng_alloc(gsl_rng_default);
19   gsl_rng_set(rng, time(NULL));
20
21   // inicialização
22   printf("Inicializando ... \n");
23   for (int i=0; i<n; i++) {
24     for (int j=0; j<i; j++) {
25       int sig = 1;
26       if (gsl_rng_uniform(rng) >= 0.5)
27         sig = -1;
```

```
28        gsl_matrix_set(a, i, j,
29                       sig*gsl_rng_uniform(rng));
30        gsl_matrix_set(a, j, i,
31                       gsl_matrix_get(a, i, j));
32      }
33      int sig = 1;
34      if (gsl_rng_uniform(rng) >= 0.5)
35        sig = -1;
36      gsl_matrix_set(a, i, i,
37                       sig*gsl_rng_uniform_pos(rng));
38    }
39    printf("feito.\n");
40
41    for (int k=0; k<n-1; k++) {
42      #pragma omp parallel for
43      for (int i=k+1; i<n; i++) {
44        gsl_matrix_set(a, i, k,
45                       gsl_matrix_get(a, i, k)/
46                       gsl_matrix_get(a, k, k));
47        for (int j=k+1; j<n; j++) {
48          gsl_matrix_set(a, i, j,
49                         gsl_matrix_get(a, i, j) -
50                         gsl_matrix_get(a, i, k) *
51                         gsl_matrix_get(a, k, j));
52        }
53      }
54    }
55    gsl_matrix_free(a);
56    gsl_rng_free(rng);
57
58    return 0;
59  }
```

Este algoritmo demanda substancialmente menos memória computacional que o código `parallelLU.cc` visto acima. Por outro lado, ele é substancialmente mais lento, podendo demandar até o dobro de tempo. Verifique!

O aumento no tempo computacional se deve ao mau uso da memória *cache*

dos processadores. A leitura de um elemento da matriz, aloca no *cache* uma sequência de elementos próximos na mesma linha. Ao escrever em um destes elementos, a alocação do *cache* é desperdiçada, forçando o *cache* a ser atualizado. Note que o código `parallelLU.cc` requer menos atualizações do *cache* que o código `parallelLU2.cc`.

$\Diamond$

Exercícios

E 2.4.1. Implemente o código MP discutido no ER 2.4.1.

E 2.4.2. Implemente um código MP para computar a inversa de uma matriz simétrica de elementos randômicos usando decomposição LU.

E 2.4.3. Considere o pseudoalgoritmo serial da composição LU apresentado acima. Por que é melhor paralelizar o laço 2.(a) do que o laço o 2.(a)ii.?

E 2.4.4. Use o código MP discutido no ER 2.4.2 para resolver um sistema $Ax = b$ de n equações e n incógnitas. Assuma que a matriz A seja simétrica.

E 2.4.5. Um algoritmo paralelo mais eficiente para computar a decomposição LU pode ser obtido usando-se a decomposição LU por blocos. Veja o vídeo https://youtu.be/E8aBJsC0bY8 e implemente um código MP para computar a decomposição LU por blocos.

2.5 Métodos iterativos para Sistemas Lineares

Nesta seção, vamos discutir sobre a paralelização MP de alguns métodos iterativos para sistemas lineares

$$Ax = b \tag{2.25}$$

com $A = [a_{i,j}]_{i,j=0}^{n-1}$, $x = (x_i)_{i=0}^{n-1}$ e $b = (b_i)_{i=0}^{n-1}$.

2.5.1 Método de Jacobi

Nós podemos escrever a i-ésima equação do sistema $Ax = b$ como

$$\sum_{j=1}^{n} a_{i,j}x_j = b_i. \tag{2.26}$$

Isolando x_i, obtemos

$$x_i = -\frac{1}{a_{i,i}}\left[\sum_{j\neq i} a_{i,j}x_j - b_i\right]. \tag{2.27}$$

Nesta última equação, temos que x_i pode ser diretamente calculado se todos os elementos x_j, $j \neq i$, forem conhecidos. Isso motiva o chamado método de Jacobi que é dado pela seguinte iteração

$$x_i(0) = \text{aprox. inicial}, \tag{2.28}$$

$$x_i(t+1) = -\frac{1}{a_{i,i}}\left[\sum_{j\neq i} a_{i,j}x_j(t) - b_i\right], \tag{2.29}$$

para cada $i = 0,1,\ldots,n-1$ e $t = 0,1,2,\ldots$. O número máximo de iterações $t_{\max}$ e o critério de parada podem ser escolhidos de forma adequada.

O pseudocódigo serial para o método de Jacobi pode ser escrito como segue

1. Alocar a aproximação inicial x^0.
2. Para $t = 0,1,2,\cdots,t_{\max}$:
   (a) Para $i = 0,1,2,\cdots,n$:
      i. $x_i = -\frac{1}{a_{i,i}}\left[\sum_{j\neq i} a_{i,j}x_j^0 - b_i\right]$.
   (b) Verificar o critério de parada.
   (c) $x0 = x$.

A paralelização MP no método de Jacobi pode ser feita de forma direta e eficaz pela distribuição do laço 2.(a) do pseudocódigo acima. O seguinte código é uma implementação MP do método de Jacobi. Os vetores b e $x0$ são

inicializados com elementos randômicos $(0, 1)$. A matriz A é inicializada como uma matriz estritamente diagonal dominante[2] com elementos randômicos $(-1, 1)$. O critério de parada é

$$\|x - x^0\|_2 < \text{tol}, \tag{2.30}$$

onde tol é a tolerância.

Código: pJacobi.cc

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <time.h>
4
5  #include <gsl/gsl_matrix.h>
6  #include <gsl/gsl_vector.h>
7  #include <gsl/gsl_blas.h>
8  #include <gsl/gsl_rng.h>
9
10 // random +/- 1
11 double randsig(gsl_rng *rng);
12
13 int main(int argc, char *argv[]) {
14
15   int n = 999;
16   int tmax = 50;
17   double tol = 1e-8;
18
19   gsl_matrix *a = gsl_matrix_alloc(n,n);
20   gsl_vector *b = gsl_vector_alloc(n);
21
22   gsl_vector *x = gsl_vector_alloc(n);
23   gsl_vector *x0 = gsl_vector_alloc(n);
24
25   // gerador randômico
26   gsl_rng *rng = gsl_rng_alloc(gsl_rng_default);
27   gsl_rng_set(rng, time(NULL));
28
```

[2] O método de Jacobi é convergente para matriz estritamente diagonal dominante.

```
29    // Inicializacao
30    // Matriz estritamente diagonal dominante
31    printf("Inicializacao ... \n");
32    double sig;
33    for (int i=0; i<n; i++) {
34      double s = 0;
35      for (int j=0; j<n; j++) {
36        double aux = gsl_rng_uniform(rng);
37        gsl_matrix_set(a, i, j,
38                       randsig(rng)*aux);
39        s += aux;
40      }
41      gsl_matrix_set(a, i, i,
42                     randsig(rng) * s);
43      gsl_vector_set(b, i,
44                     randsig(rng) *
45                     gsl_rng_uniform(rng));
46      gsl_vector_set(x0, i,
47                     randsig(rng) *
48                     gsl_rng_uniform(rng));
49    }
50    printf("feito.\n");
51
52    // Jacobi
53    for (int t=0; t<tmax; t++) {
54      #pragma omp parallel for
55      for (int i=0; i<n; i++) {
56        double s = 0;
57        for (int j=0; j<i; j++)
58          s += gsl_matrix_get(a, i, j) *
59            gsl_vector_get(x0, j);
60        for (int j=i+1; j<n; j++)
61          s += gsl_matrix_get(a, i, j) *
62            gsl_vector_get(x0, j);
63        gsl_vector_set(x, i,
64                       (gsl_vector_get(b, i) - s) /
65                       gsl_matrix_get(a, i, i));
66      }
```

```
67      // criterio de parada
68      // ||x-x0||_2 < tol
69      gsl_blas_daxpy(-1.0, x, x0);
70      double e = gsl_blas_dnrm2(x0);
71      printf("Iter. %d: %1.0e\n", t, e);
72      if (e < tol)
73        break;
74      gsl_vector_memcpy(x0, x);
75    }
76
77    gsl_matrix_free(a);
78    gsl_vector_free(b);
79    gsl_vector_free(x);
80    gsl_vector_free(x0);
81    gsl_rng_free(rng);
82
83    return 0;
84  }
85
86  double randsig(gsl_rng *rng)
87  {
88    double signal = 1.0;
89    if (gsl_rng_uniform(rng) >= 0.5)
90          signal = -1.0;
91    return signal;
92  }
```

2.5.2 Método *tipo* Gauss-Seidel

No algoritmo serial, observamos que ao calcularmos x_i pela iteração de Jacobi(2.27), as incógnitas x_j, $j < i$, já foram atualizadas. Isto motivo o método de Gauss-Seidel, cujo algoritmo é descrito no seguinte pseudocódigo:

1. Alocar a aproximação inicial x^0.
2. Para $t = 0,1,2,\cdots,t_{\max}$:
   (a) Para $i = 0,1,2,\cdots,n$:

i. $x_i = -\frac{1}{a_{i,i}} \left[\sum_{j<i} a_{i,j} x_j + \sum_{j>i} a_{i,j} x_j^0 - b_i \right]$.

(b) Verificar o critério de parada.

(c) $x0 = x$.

Embora este método seja normalmente muito mais rápido que o método de Jacobi, ele não é paralelizável. Isto se deve ao fato de que o cálculo da incógnita x_i depende dos cálculos precedentes das incógnitas x_j, $j < i$.

No entanto, a paralelização do método de Gauss-Seidel pode ser viável no caso de matrizes esparsas. Isto ocorre quando o acoplamento entre as equações não é total, podendo-se reagrupar as equações em blocos com base nos seus acoplamentos. Com isso, os blocos podem ser distribuídos entre as instâncias de processamento e, em cada uma, o método de Gauss-Seidel é aplicado de forma serial.

Uma alternativa baseada no Método de Gauss-Seidel, é utilizar o dado atualizado x_j loco que possível, independentemente da ordem a cada iteração. A iteração do tipo Gauss-Seidel pode-se ser escrita da seguinte forma

$$x_i = -\frac{1}{a_{i,i}} \left[\sum_{\hat{j} \neq i} a_{i,\hat{j}} x_{\hat{j}} + \sum_{j \neq i} a_{i,j} x_j^0 - b_i \right], \tag{2.31}$$

onde arbitrariamente $\hat{j}$ correspondem aos índices para os quais $x_{\hat{j}}$ já tenham sido atualizados na iteração corrente e j corresponde aos índices ainda não atualizados. O pseudocódigo MP deste método pode ser descrito como segue:

1. Alocar a aproximação inicial x.
2. Para $t = 0,1,2,\cdots,t_{\max}$:
   (a) $x^0 = x$.
   (b) *distribuição de laço em paralelo*:
      i. Para $i = 0,1,2,\cdots,n$:
         A. $x_i = -\frac{1}{a_{i,i}} \left[\sum_{j \neq i} a_{i,j} x_j - b_i \right]$.
   (c) Verificar o critério de parada.

Este método tipo Gauss-Seidel converge mais rápido que o método de Jacobi em muitos casos. Veja [1, p. 151–153], para alguns resultados sobre convergência.

A implementação MP do pseudocódigo acima é apresentada no código abaixo. Os elementos dos vetores b, x^0 e da matriz A são inicializados da mesma forma que no código `pJacobi.cc` acima.

Código: pGSL.cc

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <time.h>
4
5  #include <gsl/gsl_matrix.h>
6  #include <gsl/gsl_vector.h>
7  #include <gsl/gsl_blas.h>
8  #include <gsl/gsl_rng.h>
9
10 // random +/- 1
11 double randsig(gsl_rng *rng);
12
13 int main(int argc, char *argv[]) {
14
15   int n = 999;
16   int tmax = 50;
17   double tol = 1e-8;
18
19   gsl_matrix *a = gsl_matrix_alloc(n,n);
20   gsl_vector *b = gsl_vector_alloc(n);
21
22   gsl_vector *x = gsl_vector_alloc(n);
23   gsl_vector *x0 = gsl_vector_alloc(n);
24
25   // gerador randômico
26   gsl_rng *rng = gsl_rng_alloc(gsl_rng_default);
27   gsl_rng_set(rng, time(NULL));
28
29   // Inicializacao
```

```
30    // Matriz estritamente diagonal dominante
31    printf("Inicializacao ... \n");
32    double sig;
33    for (int i=0; i<n; i++) {
34      double s = 0;
35      for (int j=0; j<n; j++) {
36        double aux = gsl_rng_uniform(rng);
37        gsl_matrix_set(a, i, j,
38                       randsig(rng)*aux);
39        s += aux;
40      }
41      gsl_matrix_set(a, i, i,
42                     randsig(rng) * s);
43      gsl_vector_set(b, i,
44                     randsig(rng) *
45                     gsl_rng_uniform(rng));
46      gsl_vector_set(x, i,
47                     randsig(rng) *
48                     gsl_rng_uniform(rng));
49    }
50    printf("feito.\n");
51
52    // Random Gauss-Seidel
53    for (int t=0; t<tmax; t++) {
54      gsl_vector_memcpy(x0, x);
55      #pragma omp parallel for
56      for (int i=0; i<n; i++) {
57        double s = 0;
58        for (int j=0; j<i; j++)
59          s += gsl_matrix_get(a, i, j) *
60            gsl_vector_get(x, j);
61        for (int j=i+1; j<n; j++)
62          s += gsl_matrix_get(a, i, j) *
63            gsl_vector_get(x, j);
64        gsl_vector_set(x, i,
65                       (gsl_vector_get(b, i) - s) /
66                       gsl_matrix_get(a, i, i));
67      }
```

```
68      // criterio de parada
69      // ||x-x0||_2 < tol
70      gsl_blas_daxpy(-1.0, x, x0);
71      double e = gsl_blas_dnrm2(x0);
72      printf("Iter. %d: %1.0e\n", t, e);
73      if (e < tol)
74        break;
75    }
76
77    gsl_matrix_free(a);
78    gsl_vector_free(b);
79    gsl_vector_free(x);
80    gsl_vector_free(x0);
81    gsl_rng_free(rng);
82
83    return 0;
84  }
85
86  double randsig(gsl_rng *rng)
87  {
88    double signal = 1.0;
89    if (gsl_rng_uniform(rng) >= 0.5)
90          signal = -1.0;
91    return signal;
92  }
```

2.5.3 Método do Gradiente Conjugado

O Método do Gradiente Conjugado pode ser utilizado na resolução de sistemas lineares $Ax = b$, onde A é uma matriz simétrica e positiva definida. No caso de sistemas em gerais, o método pode ser utilizado para resolver o sistema equivalente $A'Ax = A'b$, onde A é uma matriz inversível, com A' denotando a transposta de A.

O pseudocódigo deste método é apresentado como segue:

1. Alocar a aproximação inicial x.
2. Calcular o resíduo $r = Ax - b$.

3. Alocar a direção $d = r$.
4. Para $t = 0,1,\cdots,t_{\max}$:
   (a) $\alpha = -\dfrac{r \cdot d}{d \cdot Ad}$.
   (b) $x = x + \alpha d$.
   (c) $r = Ax - b$.
   (d) $\beta = \dfrac{r \cdot Ad}{d \cdot Ad}$.
   (e) $d = -r + \beta d$

Uma versão MP deste método pode ser implementada pela distribuição em paralelo de cada uma das operações de produto escalar, multiplicação matriz-vetor e soma vetor-vetor. O seguinte código é uma implementação MP do Método do Gradiente Conjugado. Os elementos do vetor b e da matriz A são inicializados de forma randômica e é garantida que matriz é simétrica positiva definida.

Código: pGC.cc

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <time.h>
4  #include <math.h>
5
6  #include <gsl/gsl_matrix.h>
7  #include <gsl/gsl_vector.h>
8  #include <gsl/gsl_blas.h>
9  #include <gsl/gsl_rng.h>
10
11 int n = 999;
12 int tmax = 50;
13 double tol = 1e-8;
14
15 // inicializacao
16 void init(gsl_matrix *a,
17           gsl_vector *b);
18
```

```
19 // random +/- 1
20 double randsig(gsl_rng *rng);
21
22 // residuo
23 void residuo(const gsl_matrix *a,
24              const gsl_vector *x,
25              const gsl_vector *b,
26              gsl_vector *r);
27
28 // Metodo do Gradiente Conjugado
29 void pGC(const gsl_matrix *a,
30          const gsl_vector *b,
31          gsl_vector *x);
32
33 int main(int argc, char *argv[]) {
34
35   // sistema
36   gsl_matrix *a = gsl_matrix_alloc(n,n);
37   gsl_vector *b = gsl_vector_alloc(n);
38
39   // incognita
40   gsl_vector *x = gsl_vector_alloc(n);
41
42   // inicializacao
43   init(a, b);
44
45   // Método do Gradiente Conjugado
46   pGC(a, b, x);
47
48   gsl_matrix_free(a);
49   gsl_vector_free(b);
50   gsl_vector_free(x);
51
52   return 0;
53 }
54
55 /*****************************************
56 Inicializacao
```

```
57 ****************************************/
58 void init(gsl_matrix *a,
59           gsl_vector *b)
60 {
61   printf("Inicializacao ... \n");
62   // gerador randômico
63   gsl_rng *rng = gsl_rng_alloc(gsl_rng_default);
64   gsl_rng_set(rng, time(NULL));
65
66   // C - Matriz estritamente diagonal positiva
67   double sig;
68   gsl_matrix *c = gsl_matrix_alloc(n,n);
69   #pragma omp parallel for
70   for (int i=0; i<n; i++) {
71     double aux;
72     double s = 0;
73     for (int j=0; j<n; j++) {
74       aux = gsl_rng_uniform(rng);
75       gsl_matrix_set(c, i, j,
76                      randsig(rng) * aux);
77       s += aux;
78     }
79     gsl_matrix_set(c, i, i,
80                    randsig(rng) * s);
81     gsl_vector_set(b, i,
82                    randsig(rng) *
83                    gsl_rng_uniform(rng));
84   }
85   // A = C'C: Simétrica positiva definida
86   #pragma omp parallel for
87   for (int i=0; i<n; i++)
88     for (int j=0; j<n; j++) {
89       double s;
90       gsl_vector_const_view ci =
91         gsl_matrix_const_column(c, i);
92       gsl_vector_const_view cj =
93         gsl_matrix_const_column(c, j);
94       gsl_blas_ddot(&ci.vector, &cj.vector, &s);
```

```
95        gsl_matrix_set(a, i, j, s);
96      }
97
98    gsl_rng_free(rng);
99    gsl_matrix_free(c);
100
101   printf("feito.\n");
102 }
103 /*****************************************/
104
105 /*****************************************
106 Sinal randomico
107 *****************************************/
108 double randsig(gsl_rng *rng)
109 {
110   double signal = 1.0;
111   if (gsl_rng_uniform(rng) >= 0.5)
112         signal = -1.0;
113   return signal;
114 }
115 /*****************************************/
116
117 /*****************************************
118 residuo
119 *****************************************/
120 void residuo(const gsl_matrix *a,
121              const gsl_vector *x,
122              const gsl_vector *b,
123              gsl_vector *r)
124 {
125   #pragma omp parallel for
126   for (int i=0; i<n; i++) {
127     double s = 0;
128     for (int j=0; j<n; j++)
129       s += gsl_matrix_get(a, i, j) *
130         gsl_vector_get(x, j);
131     gsl_vector_set(r, i,
132                    s - gsl_vector_get(b, i));
```

```
133   }
134 }
135 /*************************************/
136
137 /*************************************
138 Metodo do Gradiente Conjugado
139 *************************************/
140 void pGC(const gsl_matrix *a,
141          const gsl_vector *b,
142          gsl_vector *x)
143 {
144   gsl_vector *r = gsl_vector_alloc(n);
145   gsl_vector *d = gsl_vector_alloc(n);
146   gsl_vector *ad = gsl_vector_alloc(n);
147
148   // x = 0
149   gsl_vector_set_zero(x);
150
151   // r = Ax - b
152   residuo(a, x, b, r);
153
154   // d = r
155   gsl_vector_memcpy(d, r);
156
157   for (int t=0; t<tmax; t++) {
158     // r.d, Ad, dAd
159     double rd = 0;
160     double dAd = 0;
161     #pragma omp parallel for reduction(+:rd,dAd)
162     for (int i=0; i<n; i++) {
163       rd += gsl_vector_get(r, i) *
164         gsl_vector_get(d, i);
165       double adi = 0;
166       for (int j=0; j<n; j++)
167         adi += gsl_matrix_get(a, i, j) *
168           gsl_vector_get(d, j);
169       gsl_vector_set(ad, i, adi);
170       dAd += gsl_vector_get(d, i) * adi;
```

```
171     }
172
173     // alpha
174     double alpha = rd/dAd;
175
176     // x = x - alpha*d
177     #pragma omp parallel for
178     for (int i=0; i<n; i++)
179       gsl_vector_set(x, i,
180                      gsl_vector_get(x, i) -
181                      alpha *
182                      gsl_vector_get(d, i));
183
184     // residuo
185     residuo(a, x, b, r);
186
187     // rAd
188     double rAd = 0;
189     #pragma omp parallel for reduction(+:rAd)
190     for (int i=0; i<n; i++)
191       rAd += gsl_vector_get(r, i) *
192         gsl_vector_get(ad, i);
193
194     // beta
195     double beta = rAd/dAd;
196
197     // d
198     #pragma omp parallel for
199     for (int i=0; i<n; i++)
200       gsl_vector_set(d, i,
201                      beta *
202                      gsl_vector_get(d, i) -
203                      gsl_vector_get(r, i));
204
205     // criterio de parada
206     // ||r||_2 < tol
207     double crt = 0;
208     #pragma omp parallel for reduction(+: crt)
```

```
209     for (int i=0; i<n; i++)
210       crt += gsl_vector_get(r, i) *
211         gsl_vector_get(r, i);
212     crt = sqrt(crt);
213     printf("Iter. %d: %1.1e\n", t, crt);
214     if (crt < tol)
215       break;
216   }
217
218   gsl_vector_free(r);
219   gsl_vector_free(d);
220   gsl_vector_free(ad);
221
222 }
223 /************************************/
```

Exercícios Resolvidos

ER 2.5.1. Faça uma implementação MP para computar a inversa de uma matriz A usando o Método de Gauss-Seidel. Assuma que A seja uma matriz estritamente diagonal dominante de dimensões $n \times n$ (n grande).

Solução. A inversa da matriz A é a matriz B de dimensões $n \times n$ tal que

$$AB = I \tag{2.32}$$

Denotando por b_k, $k = 0,1,\ldots,n$, as colunas da matriz B, temos que o problema de calcular B é equivalente a resolver os seguintes n sistemas lineares

$$Ab_k = i_k, \quad ,k = 0,1,\ldots,n, \tag{2.33}$$

onde i_k é a j-ésima coluna da matriz identidade I. Podemos usar o método de Gauss-Seidel para computar a solução de cada um destes sistemas lineares. Embora o método não seja paralelizável, os sistemas são independentes um dos outros e podem ser computados em paralelo. O pseudocódigo pode ser escrito como segue:

1. Alocar a matriz A.
2. *(início da região paralela)*

(a) Para $k = 0,1,\ldots,n$ *(laço em paralelo)*:

i. Alocar i_k.

ii. Inicializar b_k.

iii. Resolver pelo Método de Gauss-Seidel

$$Ab_k = i_k \tag{2.34}$$

A implementação fica como Exercício E 2.5.2.

$\Diamond$

ER 2.5.2. Faça uma implementação MP do método de sobre-relaxação de Jacobi (método JOR) para computar a solução de um sistema linear $Ax = b$, com A matriz estritamente diagonal dominante de dimensões $n \times n$ (n grande).

Solução. O método JOR é uma variante do método de Jacobi. A iteração JOR é

$$x_i(0) = \text{aprox. inicial}, \tag{2.35}$$

$$x_i(t+1) = (1-\gamma)x_i(t) - \frac{\gamma}{a_{i,i}}\left[\sum_{j\neq i} a_{i,j}x_j(t) - b_i\right], \tag{2.36}$$

para cada $i = 0,1,\ldots,n-1$ e $t = 0,1,2,\ldots$, com $0 < \gamma < 1$. Note que se $\gamma = 1$, então temos o Método de Jacobi.

A implementação MP do Método JOR pode ser feita de forma análoga a do Método de Jacobi (veja o código `pJacobi.cc` na Subseção 2.5.1). A implementação fica como exercício E 2.5.1.

$\Diamond$

Exercícios

E 2.5.1. Complete o ER 2.5.2.

E 2.5.2. Complete o ER 2.5.1.

E 2.5.3. O Método de Richardson para o cálculo da solução de um sistema linear $Ax = b$ de dimensões $n \times n$ tem a seguinte iteração

$$x(0) = \text{aprox. inicial}, \tag{2.37}$$
$$x(t+1) = x(t) - \gamma\left[Ax(t) - b\right], \tag{2.38}$$

onde γ é uma parâmetro escalar de relaxação e $t = 0,1,2,\ldots$. Faça uma implementação MP deste método.

E 2.5.4. O Método das Sucessivas Sobre-relaxações (SOR) é uma variante do Método de Gauss-Seidel. A iteração SOR é

$$x_i(0) = \text{aprox. inicial}, \tag{2.39}$$
$$x_i(t+1) = (1-\gamma)x_i(t) - \frac{\gamma}{a_{i,i}}\left[\sum_{j<i} a_{i,j}x_j(t+1) + \sum_{j>i} a_{i,j}x_j(t) - b_i\right], \tag{2.40}$$

onde $0 < \gamma < 1$, $i = 0,1,\ldots,n-1$ e $t = 0,1,2,\ldots$.

Este método não é paralelizável, mas ele pode ser adaptado pela distribuição paralela do cálculo das incógnitas a cada iteração conforme o Método tipo Gauss-Seidel apresentado na Subseção 2.5.2. Faça a adaptação do Método SOR e implemente em MP.

E 2.5.5. Faça a implementação do método do Gradiente Conjugado para computar a inversa de uma matriz A simétrica positiva definida de dimensões $n \times n$ (n grande).

2.6 Métodos iterativos para problemas não lineares

Vamos considerar um sistema de equações não lineares

$$F(x) = 0, \tag{2.41}$$

onde $x = (x_1,x_2,\ldots,x_n) \in \mathbb{R}^n$ e $F : \mathbb{R}^n \mapsto \mathbb{R}^n$, $n \geq 1$.

2.6.1 Método de Newton

Supondo que F é duas vezes diferenciável, a solução de (2.41) pode ser computada pela iteração de Newton:

$$x(0) = \text{aprox. inicial}, \tag{2.42}$$

$$x(t+1) = x(t) - \gamma J_F^{-1}\left(x(t)\right) F\left(x(t)\right), \tag{2.43}$$

onde $\gamma > 0$ é o tamanho do passo escolhido,

$$J_F\left(\cdot\right) = \left[\frac{\partial F_i}{\partial x_j}\left(\cdot\right)\right]_{i,j=1}^{n,n} \tag{2.44}$$

denota a jacobiana de F, e $t = 1,2,3,\ldots$.

Observamos que, em geral, a iterada de Newton (2.43) não é trivialmente paralelizável, devido a acoplamentos entre as n equações. Por outro lado, podemos reescrever (2.43) como segue:

$$x(t+1) = x(t) + \gamma s(t), \tag{2.45}$$

onde $s(t)$ é o passo de Newton, dado como a solução do seguinte sistema linear

$$J_F\left(x(t)\right) s(t) = -F\left(x(t)\right). \tag{2.46}$$

Desta forma, a cada iteração de Newton t, devemos computar a solução do sistema linear (2.46). A aplicação da paralelização no método de Newton dá-se pela utilização de métodos paralelizáveis para a resolução de sistemas lineares. Na Seção 2.4 e, principalmente, na Seção 2.5, discutimos sobre a paralelização de métodos para sistemas lineares.

No caso de um sistema de grande porte e uma vez computada $s(t)$, a atualização (2.45) também pode ser trivialmente paralelizada. Ainda, as computações da função objetivo F e de sua jacobiana J_F também são paralelizáveis.

2.6.2 Método do acorde

Em problemas de grande porte, o cálculo da jacobina J_F é, em muitos casos, o passo computacionalmente mais custoso na aplicação do método de

Newton. Uma alternativa é o chamado método do acorde, no qual a jacobiana é computada apenas na iteração inicial. Segue a iteração deste método

$$x(0) = \text{aprox. inicial}, \tag{2.47}$$
$$J_{F,0} = F_F\left(x(0)\right), \tag{2.48}$$

$$x(t+1) = x(t) + \gamma s(t), \tag{2.49}$$
$$J_{F,0}s(t) = -F(x(t)), \tag{2.50}$$

onde $t = 0, 1, 2, \ldots$.

Enquanto a taxa de convergência do método de Newton é quadrática, o método do acorde tem convergência linear. Portanto, este só é vantajoso quando o custo de computar a jacobiana é maior que o custo de se computar várias iterações a mais.

Além das paralelizações triviais na computação de (2.48) e (2.54), vamos observar a computação da direção $s(t)$ (2.55). Como a jacobiana $J_{F,0}$ é fixada constante, a utilização de métodos iterativos para computar $s(t)$ pode não ser o mais adequado. Aqui, a utilização de método direto, como a decomposição LU torna-se uma opção a ser considerada. Neste caso, a iteração ficaria como segue

$$x(0) = \text{aprox. inicial}, \tag{2.51}$$
$$J_{F,0} = J_F\left(x(0)\right), \tag{2.52}$$
$$LU = J_{F,0}, \tag{2.53}$$

$$x(t+1) = x(t) + \gamma s(t), \tag{2.54}$$
$$LUs(t) = -F(x(t)), \tag{2.55}$$

onde $t = 0, 1, 2, \ldots$.

2.6.3 Métodos *quasi*-Newton

Baseados em aproximações na computação do passo de Newton

$$J_F\left(x(t)\right)s(t) = -F\left(x(t)\right), \tag{2.56}$$

uma série de métodos *quasi*-Newton são derivados. A aplicação de cada uma dessas tais variantes precisa ser avaliada caso a caso. Em todas elas, busca-se abrir mão da convergência quadrática em troca de um grande ganho no tempo computacional em se computar $s(t)$.

Uma das alternativas é uma variante do método do acorde. A ideia é estimar a taxa de convergência p entre as iterações e atualizar a jacobiana quando a taxa estimada é menor que um certo limiar p_l considerado adequado (este limiar pode ser escolhido com base nos custos computacionais de se recomputar a jacobiana *versus* o de se computar várias iterações a mais). A convergência é da ordem p quando

$$\|F\left(x(t+1)\right)\| \approx K\|F\left(F(x(t))\right)\|^p, \tag{2.57}$$

com $K > 0$, $\|F\left(x(t)\right)\| \to 0$ quando $t \to \infty$. Assim sendo, é razoável esperar que

$$p \approx \frac{\log\left(\|F\left(x(t+1)\right)\|\right)}{\log\left(\|F\left(x(t)\right)\|\right)} \tag{2.58}$$

. Com isso, o pseudocódigo segue

1. Aproximação inicial: $x(0)$, $t = 0$.
2. Jacobiana: $J_F = J_F\left(x(t)\right)$.
3. Enquanto $\|F\left(x(t)\right)\| > tol$:
   (a) $J_F s(t) = -F\left(x(t)\right)$.
   (b) $x(t+1) = x(t) + \gamma s(t)$.
   (c) $p = \log\left(\|F\left(x(t+1)\right)\|\right) / \log\left(\|F\left(x(t)\right)\|\right)$
   (d) Se $p < p_l$, então:
      i. $J_F = J_F\left(x(t+1)\right)$.
   (e) $t = t + 1$.

Outra alternativa que pode ser considerada em determinados casos, é a de se computar $s(t)$ por

$$J_F\left(x(t)\right) s(t) = -F\left(x(t)\right) \tag{2.59}$$

de forma aproximada. No contexto de métodos iterativos para sistemas lineares, pode-se truncar a resolução do sistema acima fixando um número

pequeno de iterações. Desta forma, $s(t)$ não seria computada de forma precisa, mas a aproximação computada pode ser suficientemente adequada.

Do ponto de vista de paralelização em MP, estas variantes do método de Newton apresentam potenciais e requerem cuidados similares ao método original.

Exercícios

E 2.6.1. Implemente um código MP para computar a solução de

$$\operatorname{sen}(x_1 x_2) - 2x_2 - x_1 = -4.2 \tag{2.60}$$
$$3e^{2x_1} - 6ex_2^2 - 2x_1 = -1 \tag{2.61}$$

usando o método de Newton. Use a inversa da jacobiana exata e aproximação inicial $x(0) = (2{,}2)$.

E 2.6.2. Considere o seguinte problema de Poisson não-linear

$$-\frac{\partial}{\partial x}\left[\left(1+u^2\right)\frac{\partial u}{\partial x}\right] = \cos(\pi x), \quad x \in (-1,1), \tag{2.62}$$
$$u(0) = 1, \quad \left.\frac{\partial u}{\partial x}\right|_{x=1} = 0. \tag{2.63}$$

Use o método de diferenças finitas para discretizar este problema de forma a aproximá-lo como um sistema algébrico de equações não lineares. Implemente um código MP para computar a solução do sistema resultante aplicando o método de Newton.

E 2.6.3. No Exercício 2.6.2, faça uma implementação MP do método do acorde e compare com o método de Newton clássico.

E 2.6.4. No Exercício 2.6.2, faça uma implementação MP da variante do método do acorde com atualização da jacobiana com base na estimativa da taxa de convergência.

Capítulo 3

Computação paralela e distribuída (MPI)

Neste capítulo, vamos estudar aplicações da computação paralela em arquitetura de memória distribuída. Para tanto, vamos utilizar códigos C/C++ com a API Open MPI.

3.1 Olá, Mundo!

A computação paralela com MPI inicia-se simultaneamente com múltiplos processadores (instâncias de processamento), cada um utilizando seu próprio endereço de memória (memória distribuída). Cada processo lê e escreve em seu próprio endereço de memória privada. Observamos que o processamento já inicia-se ramificado e distribuído, sendo possível a comunicação entre os processos por instruções explícitas (instruções MPI, *Message Passing Interface*). A sincronização entre os processos também requer instruções específicas.

Vamos escrever nosso primeiro código MPI. O Código `ola.cc` é paralelamente executado por diferentes processadores, cada processo escreve "Olá" e identifica-se.

Código: ola.cc

```
1 #include <stdio.h>
2
```

```
3  // API MPI
4  #include <mpi.h>
5
6  int main(int argc, char** argv) {
7    // Inicializa o MPI
8    MPI_Init(NULL, NULL);
9
10   // número total de processos
11   int world_size;
12   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
13
14   // ID (rank) do processo
15   int world_rank;
16   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
17
18   // Escreve mensagem
19   printf("Olá! Eu sou o processo %d/%d.\n",
20          world_rank, world_size);
21
22   // Finaliza o MPI
23   MPI_Finalize();
24
25   return 0;
26 }
```

Na linha 3, o API MPI é incluído no código. O ambiente MPI é inicializado na linha 8 com a rotina MPI_Init inicializa o ambiente MPI. Na inicialização, o comunicador `MPI_COMM_WORLD` é construído entre todos os processos inicializados e um identificador (*rank*) é atribuído a cada processo. O número total de processos é obtido com a rotina MPI_Comm_size. Cada processo é identificado por um número natural sequencial 0, 1, ..., `world_size`-1. O id (*rank*) de um processo é obtido com a rotina MPI_Comm_rank (veja a linha 16). A rotina MPI_Finalize finaliza o ambiente MPI.

Para compilar este código, digite no terminal

```
$ mpic++ ola.cc
```

Esta instrução de compilação é análoga a

```
g++ ola.cc -I/usr/lib/x86_64-linux-gnu/openmpi/include/openmpi
   -I/usr/lib/x86_64-linux-gnu/openmpi/include
   -pthread -L/usr/lib/x86_64-linux-gnu/openmpi/lib
   -lmpi_cxx -lmpi
```

ou semelhante dependendo da instalação. Para ver a sua configuração, digite

```
$ mpic++ ola.cc --showme
```

Ao compilar, um executável `a.out` será criado. Para executá-lo, basta digitar no terminal:

```
$ mpirun -np 2 a.out
```

Esta instrução inicializa simultaneamente duas cópias (`-np 2`, dois processos) do código `ola.cc` (do executável `a.out`). Cada processo é executado de forma idependente (em paralelo e não sincronizados).

Ao executar, devemos ver a saída do terminal como algo parecido com

```
Olá! Eu sou o processo 1/2.
Olá! Eu sou o processo 0/2.
```

A saída irá variar conforme o processo que primeiro enviar a mensagem para o dispositivo de saída. Execute o código várias vezes e analise as saídas!

Exercícios resolvidos

ER 3.1.1. O número de instâncias de processamento pode ser alterado diretamente na instrução `mpirun` pela opção `-np`. Altere o número de instâncias de processamento para 4 e execute o Código ola.cc.

Solução. Para alterar o número de instâncias de processamento não é necessário recompilar o código[1]. Basta executá-lo com o comando

```
$ mpirun -np 4 ./a.out
```

A saída deve ser algo do tipo

```
Olá! Eu sou o processo 1/4.
Olá! Eu sou o processo 3/4.
Olá! Eu sou o processo 2/4.
```

[1] Caso ainda não tenha compilado o código, copile-o.

```
Olá! Eu sou o processo 0/4.
```

Execute o código várias vezes e analise as saídas!

$\Diamond$

ER 3.1.2. Escreva um código MPI para ser executado com 2 instâncias de processamento. Cada processo recebe os números inteiros

```
int n = 2;
int m = 3;
```

Então, um dos processos deve escrever a soma $n + m$ e o outro deve escrever o produto.

Solução. O código abaixo contém uma implementação deste exercício. Veja os comentários abaixo.

Código: sp.cc

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  // API MPI
5  #include <mpi.h>
6
7  int main(int argc, char** argv) {
8    // Inicializa o MPI
9    MPI_Init(NULL, NULL);
10
11   // número total de processos
12   int world_size;
13   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
14
15   // verifica o num. de processos
16   if (world_size != 2) {
17     printf("ERRO! Número de processos "
18            "deve ser igual 2.\n");
19     int errorcode=-1;
20     MPI_Abort(MPI_COMM_WORLD, errorcode);
21   }
22
```

```
23    // ID (rank) do processo
24    int world_rank;
25    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
26
27    int n = 2;
28    int m = 3;
29
30    if (world_rank == 0)
31      printf("n+m = %d\n", n+m);
32    else if (world_rank == 1)
33      printf("n*m = %d\n", n*m);
34
35    // Finaliza o MPI
36    MPI_Finalize();
37
38    return 0;
39  }
```

Neste código, os processos são abortados caso o usuário tente executá-lo com um número de processos diferente de 2. Para abortar todos os processos ativos, utiliza-se a rotina MPI_Abort (veja as linhas 15-21). O argumento de entrada `errorcode` é arbitrário e serve para informar o usuário de uma categoria de erros conforme a política de programação utilizada.

Observamos que o controle do que cada processo deve fazer, é feito através de sua identificação `world_rank` (veja as linhas 30-33).

♢

Exercícios

E 3.1.1. Rode o Código `ola.cc` com um número de processadores (*core*) maior do que o disponível em sua máquina. O que você observa? Modifique a instrução `mpirun` para aceitar a execução confirme o número de *threads* disponível na máquina. Por fim, modifique a instrução de forma a aceitar um número arbitrário de instâncias de processamento.

E 3.1.2. Faça um código MPI para ser executado com 2 instâncias de

processamento. Uma das instâncias de processamento deve alocar

```
int a = 2;
int b = 3;
```

e escrever a diferença $a - b$. A outra instância deve alocar

```
int a = 4;
int b = 5;
```

e escrever o quociente b/a.

3.2 Rotinas de comunicação ponto-a-ponto

Em computação distribuída, rotinas de comunicação entre as instâncias de processamento são utilizadas para o compartilhamento de dados. Neste capítulo, vamos discutir sobre as rotinas de comunicação ponto-a-ponto, i.e. comunicações entre uma instância de processamento com outra.

3.2.1 Envio e recebimento síncronos

O envio e recebimento de dados entre duas instâncias de processamento pode ser feita com as rotinas MPI_Send e MPI_Recv. A primeira é utilizada para o envio de um dado a partir de uma instância de processamento e a segunda é utilizada para o recebimento de um dado em uma instância de processamento.

A sintaxe da MPI_Send é

```
int MPI_Send(
  const void *buf,
  int count,
  MPI_Datatype datatype,
  int dest,
  int tag,
  MPI_Comm comm)
```

e a sintaxe da MPI_Recv é

```
int MPI_Recv(
  void *buf,
```

```
  int count,
  MPI_Datatype datatype,
  int source,
  int tag,
  MPI_Comm comm,
  MPI_Status *status)
```

O primeiro argumento é o ponteiro do *buffer* de dados. No caso do `MPI_Send` é o ponteiro para a posição da memória do dado a ser enviado. No caso do `MPI_Recv` é o ponteiro para a posição da memória do dado a ser recebido. O segundo argunto `count` é o número de dados sequenciais a serem enviados. O argundo `datatype` é o tipo de dado. O MPI suporta os seguintes tipos de dados

`MPI_SHORT`	`short int`
`MPI_INT`	`int`
`MPI_LONG`	`long int`
`MPI_LONG_LONG`	`long long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_UNSIGNED_LONG_LONG`	`unsigned long long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_BYTE`	`char`

Ainda sobre as sintaxes acima, o argumento `source` é o identificador *rank* da instância de processamento. O argunmento `tag` é um número arbitrário para identificar a operação de envio e recebimento. O argumento `Comm` especifica o comunicador (`MPI_COMM_WORLD` para aplicações básicas) e o último (somente para o `MPI_Recv`) fornece informação sobre o *status* do recebimento do dado.

Vamos estudar o seguinte código abaixo.

Código: sendRecv.cc

```
1 #include <stdio.h>
2
```

```
3  // API MPI
4  #include <mpi.h>
5
6  int main (int argc, char** argv) {
7    // Inicializa o MPI
8    MPI_Init(NULL, NULL);
9
10   // número total de processos
11   int world_size;
12   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
13
14   // ID (rank) do processo
15   int world_rank;
16   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
17
18   if (world_rank == 0) {
19     double x = 3.1416;
20     MPI_Send (&x, 1, MPI_DOUBLE, 1,
21               0, MPI_COMM_WORLD);
22   } else {
23     double y;
24     MPI_Recv (&y, 1, MPI_DOUBLE, 0,
25               0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26     printf ("Processo 1 recebeu o "\
27             "número %f do processo 0.\n", y);
28   }
29
30
31   // Finaliza o MPI
32   MPI_Finalize ();
33
34   return 0;
35 }
```

O código acima pode rodado com pelo menos duas instâncias de processamento (veja as linhas 14-19). Nas linhas 28-29, o processo 0 envia o número `3.1416` (alocado na variável `x`) para o processo 1. Nas linhas 32-33, o pro-

cesso 1 recebe o número enviado pelo processo 0 e o aloca na variável `y`.

Importante! As rotinas `MPI_Send` e `MPI_Recv` provocam a sincronização entre os processos envolvidos. Por exemplo, no código acima, no que o processo 0 atinge a rotina `MPI_Send` ele ficará aguardando o processo 1 receber todos os dados enviados e só, então, irá seguir adiante no código. Analogamento, no que o processo 1 atingir a rotina `MPI_Recv`, ele ficará aguardando o processo 0 enviar todos os dados e só, então, irá seguir adiante no código.

Envio e recebimento de *array*

As rotinas `MPI_Send` e `MPI_Recv` podem ser utilizadas para o envio e recebimento de *arrays*. A sintaxe é a mesma vista acima, sendo que o primeiro argumento `*buf` deve apontar para o início do *array* e o segundo argumento `count` corresponde ao tamanho da *array*.

Vamos estudar o seguinte código. Nele, o processo 0 aloca $v = (0,1,2,3,4)$ e o processo 1 aloca $w = (4,3,2,1,0)$. O processo 0 envia os valores v_1, v_2, v_3 para o processo 1. Então, o processo 1 recebe estes valores e os aloca em w_0, w_1, w_2. Desta forma, a saída impressa no terminal é

$$w = (1, 2, 3, 1, 0). \tag{3.1}$$

Verifique!

Código: sendRecvArray.cc

```
1  #include <stdio.h>
2
3  // API MPI
4  #include <mpi.h>
5
6  int main (int argc, char** argv) {
7    // Inicializa o MPI
8    MPI_Init(NULL, NULL);
9
10   // número total de processos
11   int world_size;
12   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
13
14   // ID (rank) do processo
```

```
15    int world_rank;
16    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
17
18    if (world_rank == 0) {
19
20      int v[5];
21      for (int i=0; i<5; i++)
22        v[i] = i;
23
24      MPI_Send (&v[1], 3, MPI_INT, 1,
25                0, MPI_COMM_WORLD);
26    } else {
27      int w[5];
28      int i=0;
29      for (int j=5; j --> 0; i++)
30        w[j] = i;
31
32      MPI_Recv (&w[0], 3, MPI_INT, 0,
33                0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
34      printf ("Processo 1: w=\n");
35      for (int i=0; i<5; i++)
36        printf ("%d ", w[i]);
37      printf("\n");
38    }
39
40    // Finaliza o MPI
41    MPI_Finalize ();
42
43    return 0;
44  }
```

3.2.2 Envio e recebimento assíncrono

O `MPI` também suporta rotinas MPI_Isend de envio e MPI_Irecv de recebimento assíncronos. Neste caso, o processo emissor envia o dado para outro processo e segue imediatamente a computação. O processo receptor deve conter uma rotina MPI_Irecv, mas também não aguarda sua conclusão

para seguir a computação.

As sintaxes destas rotinas são semelhantes as das rotinas `MPI_Send` e `MPI_Recv`.

```
int MPI_Isend(
  const void *buf,
  int count,
  MPI_Datatype datatype,
  int dest,
  int tag, MPI_Comm comm,
  MPI_Request *request)
```

```
int MPI_Irecv(
  void *buf,
  int count,
  MPI_Datatype datatype,
  int source,
  int tag,
  MPI_Comm comm,
  MPI_Request *request)
```

O último argumento permite verificar os envios e recebimentos.

Vamos estudar o seguinte código.

Código: isendRecv.cc

```
1  #include <stdio.h>
2
3  // API MPI
4  #include <mpi.h>
5
6  int main (int argc, char** argv) {
7    // Inicializa o MPI
8    MPI_Init(NULL, NULL);
9
10   // número total de processos
11   int world_size;
12   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
13
```

```
14    if (world_size < 2) {
15      printf ("Num. de processos deve"\
16              "maior que 2.\n");
17      int errorcode = -1;
18      MPI_Abort (MPI_COMM_WORLD, errorcode);
19    }
20
21    // ID (rank) do processo
22    int world_rank;
23    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
24
25    // MPI_Status & MPI_Request
26    MPI_Status status;
27    MPI_Request request;
28
29    if (world_rank == 0) {
30      double x = 3.1416;
31      MPI_Isend (&x, 1, MPI_DOUBLE, 1,
32                 0, MPI_COMM_WORLD, &request);
33    } else {
34      double y = 0.0;
35      MPI_Irecv (&y, 1, MPI_DOUBLE, 0,
36                 0, MPI_COMM_WORLD, &request);
37      double x = y + 1.0;
38      printf ("x = %f\n", x);
39      int recvd = 0;
40      while (!recvd)
41        MPI_Test (&request, &recvd, &status);
42      x = y + 1;
43      printf ("x = %f\n", x);
44    }
45
46    // Finaliza o MPI
47    MPI_Finalize ();
48
49    return 0;
50  }
```

Neste código, `MPI_Status` e `MPI_Request` são alocados nas linhas 26 e 27, respectivamente. O Processo 0 faz uma requisição de envio do número 3.1416 para o processo 1, não aguarda o recebimento e segue adiante. O processo 1 tem uma rotina de requisição de recebimento não assíncrona na linha 35. Neste momento, ele não necessariamente recebe o dado enviado pelo processador (isto pode ocorrer a qualquer momento mais adiante). Na linha 37, o valor de `y` deve ainda ser 0.0, veja a saída do código.

```
$ mpic++ isendRecv.cc
$ $ mpirun -np 2 ./a.out
x = 1.000000
x = 4.141600
```

Pode-se verificar se uma requisição de envio (ou recebimento) foi completata usando-se a rotina MPI_Test. A sua sintaxe é

```
int MPI_Test(
  MPI_Request *request,
  int *flag,
  MPI_Status *status)
```

O `flag == 0` caso a requisição ainda não foi completada e `flag == 1` caso a requisição foi executada.

No Código `isendRecv.cc` acima, as linhas de código 39-41 são utilizadas para fazê-lo aguardar até que a requisição de recebimento seja completada. Desta forma, na linha 42 o valor de `y` é 3.1416 (o valor enviado pelo processo 0. Verifique!

Observação 3.2.1. No Código `isendRecv.cc` acima, as linhas 39-41 podem ser substituídas pela rotina MPI_Wait, a qual tem sintaxe

```
int MPI_Wait(
  MPI_Request *request,
  MPI_Status *status)
```

Verifique!

Exercícios

E 3.2.1. Faça um código MPI para ser executado com 2 processadores.

Um processo aloca $x = 0$ e o outro processo aloca $y = 1$. Logo, os processos trocam os valores, de forma que ao final o processo zero tem $x = 1$ e o processo 1 tem $y = 0$.

E 3.2.2. Faça um código MPI para ser executado com 2 processadores. O processo 0 aloca um vetor de $n \geq 1$ elementos randômicos em ponto flutuante, envia o vetor para o processo 1. O processo 0, imprime no terminal a soma dos termos do vetor e o processo 1 imprime o produto dos termos do vetor.

E 3.2.3. Faça um código MPI para computar a média

$$\frac{1}{n}\sum_{i=0}^{n-1} x_i \tag{3.2}$$

onde x_i é um número em ponto flutuante e $n \geq 1$. Para a comunicação entre os processos, utilize apenas as rotinas `MPI_Send` e `MPI_Recv`.

E 3.2.4. Faça um código MPI para computação do produto interno entre dois vetores

$$x = (x_0, x_1, \ldots, x_n), \tag{3.3}$$
$$y = (y_0, y_1, \ldots, y_n). \tag{3.4}$$

Para a comunicação entre os processos, utilize apenas as rotinas `MPI_Send` e `MPI_Recv`. O processo 0 deve receber os resultados parciais dos demais processos e escrever na tela o valor computado do produto interno.

E 3.2.5. Modifique o código do exercício anterior (Exercício 3.2.4) de forma a fazer a comunicação entre os processos com as rotinas `MPI_Isend` e `MPI_Irecv`. Há vantagem em utilizar estas rotinas? Se sim, quais?

3.3 Comunicações coletivas

Nesta seção, vamos discutir sobre rotinas de comunicações MPI coletivas. Basicamente, rotinas de sincronização, envio e recebimento de dados envolvendo múltiplas instâncias de processamento ao mesmo tempo.

3.3.1 Barreira de sincronização

Podemos forçar a sincronização de todos os processos em um determinado ponto do código utilizando a rotina de sincronização MPI_Barrier

```
int MPI_Barrier (MPI_Comm comm)
```

Quando um processo encontra esta rotina ele aguarda todos os demais processos. No momento em que todos os processo tiverem alcançados esta rotina, todos são liberados para seguirem com suas computações.

Exemplo 3.3.1. No Código `barrier.cc`, abaixo, cada instância de processamento aguarda randomicamente até 3 segundos para alcançar a rotina de sincronização MPI_Barrier na linha 36. Em seguida, elas são liberadas juntas. Estude o código.

Código: barrier.cc

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  // API MPI
6  #include <mpi.h>
7
8  int main(int argc, char** argv) {
9
10   // Inicializa o MPI
11   MPI_Init(NULL, NULL);
12
13   // número total de processos
14   int world_size;
15   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
16
17   // ID (rank) do processo
18   int world_rank;
19   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
20
21   // cronometro
22   time_t init = time (NULL);
23
```

```
24    // semente do gerador randômico
25    srand (init + world_rank);
26
27    // max. of 3 segundos
28    size_t espera = rand() % 3000000;
29
30    usleep (espera);
31
32    printf ("%d chegou na barreira: %ld s.\n",
33            world_rank, (time (NULL) - init));
34
35    MPI_Barrier (MPI_COMM_WORLD);
36
37    printf ("%d saiu da  barreira: %ld s.\n",
38            world_rank, (time (NULL) - init));
39
40
41    // Finaliza o MPI
42    MPI_Finalize();
43
44    return 0;
45  }
```

Vamos observar o seguinte teste de rodagem

```
$ mpic++ barrier.cc
$ mpirun -np 2 a.out
1 chegou na barreira: 1 s.
0 chegou na barreira: 3 s.
0 saiu da  barreira: 3 s.
1 saiu da  barreira: 3 s.
```

Neste caso, o processo 1 foi o primeiro a alcançar a barreira de sincronização e permaneceu esperando aproximadamente 2 segundos até que o processo 0 alcançasse a barreira. Imediatamente após o processo 1 chegar a barreira, ambos seguiram suas computações. Rode várias vezes este código e analise as saídas!

Observação 3.3.1. No Código `barrier.cc` acima, o gerador de números

randômicos é inicializado com a semente

```
srand (init + world_rank);
```

onde, `init` é o tempo colhido pela rotina `time` no início do processamento (veja as linhas 22-25). Observamos que somar o identificado `rank` garante que cada processo inicie o gerador randômico com uma semente diferente.

3.3.2 Transmissão coletiva

A rotina de transmissão de dados MPI_Bcast permite o envio de dados de um processo para todos os demais. Sua sintaxe é a seguinte

```
int MPI_Bcast(
  void *buffer,
  int count,
  MPI_Datatype datatype,
  int root,
  MPI_Comm comm)
```

O primeiro argumento `buffer` aponta para o endereço da memória do dado a ser transmitido. O argumento `count` é a quantidade de dados sucessivos que serão transmitidos (tamanho do `buffer`). O tipo de dado é informado no argumento `datatype`. Por fim, `root` é o identificador `rank` do processo que está transmitindo e `comm` é o comunicador.

Exemplo 3.3.2. No seguinte Código `bcast.cc`, o processo 0 inicializa a variável de ponto flutuante $x = \pi$ (linhas 22-23) e, então, transmite ela para todos os demais processos (linhas 25-26). Por fim, cada processo imprime no terminal o valor alocado na sua variável x (linhas 28-29).

Código: bcast.cc

```
1 #include <stdio.h>
2 #include <math.h>
3
4 // API MPI
5 #include <mpi.h>
6
7 int main(int argc, char** argv) {
8
```

```
 9   // Inicializa o MPI
10   MPI_Init(NULL, NULL);
11
12   // número total de processos
13   int world_size;
14   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
15
16   // ID (rank) do processo
17   int world_rank;
18   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
19
20   double x;
21
22   if (world_rank == 0)
23     x = M_PI;
24
25   MPI_Bcast (&x, 1, MPI_DOUBLE,
26              0, MPI_COMM_WORLD);
27
28   printf ("Processo %d x = %f\n",
29           world_rank, x);
30
31   // Finaliza o MPI
32   MPI_Finalize();
33
34   return 0;
35 }
```

Vejamos o seguinte teste de rodagem

```
$ mpic++ bcast.cc
$ mpirun -np 3 ./a.out
Processo 0 x = 3.141593
Processo 1 x = 3.141593
Processo 2 x = 3.141593
```

3.3.3 Distribuição coletiva de dados

A rotina MPI_Scatter permite que um processo faça a distribuição uniforme de pedaços sequenciais de um *array* de dados para todos os demais processos. Sua sintaxe é a seguinte

```
int MPI_Scatter(
  const void *sendbuf,
  int sendcount,
  MPI_Datatype sendtype,
  void *recvbuf,
  int recvcount,
  MPI_Datatype recvtype,
  int root,
  MPI_Comm comm)
```

O primeiro argumento `sendbuf` aponta para o endereço de memória do *array* de dados a ser distribuído. O argumento `sendcount` é o tamanho do pedaço e `sendtype` é o tipo de dado a ser transmitido. Os argumentos `recvbuf`, `recvcount` e `recvtype` se referem ao ponteiro para o local de memória onde o dado recebido será alocado, o tamanho do pedaço a ser recebido e o tipo de dado, respectivamente. Por fim, o argumento `root` identifica o processo de origem da distribuição dos dados e `comm` é o comunicador.

Exemplo 3.3.3. No Código `scatter.cc` abaixo, o processo 0 aloca o vetor

$$v = (1, 2, \ldots, 10), \tag{3.5}$$

distribui pedaços sequenciais do vetor para cada processo no comunicador `MPI_COMM_WORLD` e, então, cada processo computa a soma dos elementos recebidos.

Código: scatter.cc

```
1 #include <stdio.h>
2 #include <math.h>
3
4 // API MPI
5 #include <mpi.h>
6
7 // gsl
8 #include <gsl/gsl_vector.h>
```

```
9
10 int main(int argc, char** argv) {
11
12   // Inicializa o MPI
13   MPI_Init(NULL, NULL);
14
15   // número total de processos
16   int world_size;
17   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
18
19   // ID (rank) do processo
20   int world_rank;
21   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
22
23   const size_t n = 10;
24   gsl_vector *v = gsl_vector_alloc (0);
25
26   if (world_rank == 0) {
27     v = gsl_vector_alloc (n);
28
29     for (size_t i=0; i<n; i++)
30       gsl_vector_set (v, i, i+1);
31   }
32
33   size_t my_n = n/world_size;
34   gsl_vector *my_v = gsl_vector_alloc (my_n);
35
36   MPI_Scatter (v->data, my_n, MPI_DOUBLE,
37                my_v->data, my_n, MPI_DOUBLE,
38                0, MPI_COMM_WORLD);
39
40   double soma = 0.0;
41   for (size_t i=0; i<my_n; i++) {
42     soma += gsl_vector_get (my_v, i);
43   }
44
45   printf ("Processo %d soma = %f\n",
46           world_rank, soma);
```

```
47
48    // Finaliza o MPI
49    MPI_Finalize();
50
51    return 0;
52  }
```

Vejamos o seguinte teste de rodagem

```
$ mpic++ scatter.cc -lgsl -lgslcblas
$ mpirun -np 2 ./a.out
Processo 0 soma = 15.000000
Processo 1 soma = 40.000000
```

Neste caso, o processo 0 recebe

$$\texttt{my\_v} = (1, 2, 3, 4, 5), \tag{3.6}$$

enquanto o processo 1 recebe

$$\texttt{my\_v} = (6, 7, 8, 9, 10). \tag{3.7}$$

Observação 3.3.2. Note que o MPI_Scatter distribuí apenas pedaços de *arrays* de mesmo tamanho. Para a distribuição de pedaços de tamanhos diferentes entre os processos, pode-se usar a rotina MPI_Scatterv. Veja os exercícios resolvidos abaixo.

3.3.4 Recebimento coletivo de dados distribuídos

A rotina MPI_Gather, permite que um processo receba simultaneamente dados que estão distribuídos entre os demais processos. Sua sintaxe é a seguinte

```
int MPI_Gather(
  const void *sendbuf,
  int sendcount,
  MPI_Datatype sendtype,
  void *recvbuf,
  int recvcount,
  MPI_Datatype recvtype,
```

```
  int root,
  MPI_Comm comm)
```

Sua sintaxe é parecida com a da rotina MPI_Scatter. Veja lá! Aqui, `root` é o identificador *rank* do processo receptor.

Exemplo 3.3.4. No Código `gather.cc`, cada processo i aloca um vetor

$$\texttt{my\_v} = (5i + 1, 5i + 2, \ldots, 5i + 5), \tag{3.8}$$

então, o processo 0 recebe estes vetores alocando-os em um único vetor

$$\texttt{v} = (1, 2, \ldots, 5n_p), \tag{3.9}$$

onde n_p é o número de processos inicializados.

Código: gather.cc

```
1  #include <stdio.h>
2  #include <math.h>
3
4  // API MPI
5  #include <mpi.h>
6
7  // gsl
8  #include <gsl/gsl_vector.h>
9
10 int main(int argc, char** argv) {
11
12   // Inicializa o MPI
13   MPI_Init(NULL, NULL);
14
15   // número total de processos
16   int world_size;
17   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
18
19   // ID (rank) do processo
20   int world_rank;
21   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
22
23   const size_t my_n = 5;
```

```
24    gsl_vector *my_v = gsl_vector_alloc (my_n);
25    for (size_t i=0; i<my_n; i++)
26        gsl_vector_set (my_v, i, 5*world_rank+i+1);
27
28    const size_t n = world_size*my_n;
29    gsl_vector *v = gsl_vector_alloc (0);
30    if (world_rank == 0) {
31      v = gsl_vector_alloc (n);
32    }
33
34    MPI_Gather (my_v->data, my_n, MPI_DOUBLE,
35                v->data, my_n, MPI_DOUBLE,
36                0, MPI_COMM_WORLD);
37
38    if (world_rank == 0) {
39      printf ("v = ");
40      for (size_t i=0; i<n; i++)
41        printf ("%f ", gsl_vector_get (v, i));
42      printf("\n");
43    }
44
45    // Finaliza o MPI
46    MPI_Finalize();
47
48    return 0;
49  }
```

Vejamos o seguinte teste de rodagem

```
$ mpic++ gather.cc -lgsl -lgslcblas
$ mpirun -np 2 ./a.out
v = 1.000000 2.000000 3.000000 4.000000 5.000000
6.000000 7.000000 8.000000 9.000000 10.000000
```

Neste caso, o processo 0 aloca

$$\texttt{my\_v} = (1, 2, 3, 4, 5) \tag{3.10}$$

e o processo 1 aloca

$$\texttt{my\_v} = (6, 7, 8, 9, 10). \tag{3.11}$$

Então, o processo 0, recebe os dois pedaços de cada um, formando o vetor

$$\mathtt{v} = (1, 2, \ldots, 10). \tag{3.12}$$

Verifique!

Observação 3.3.3. Para recebimento de pedaços distribuídos e de tamanhos diferentes, pode-se usar a rotina MPI_Gatherv. Veja os exercícios resolvidos abaixo.

Observação 3.3.4. Observamos que com a rotina MPI_Gather podemos juntar pedaços de dados distribuídos em um único processo. Analogamente, a rotina MPI_Allgather nos permite juntar os pedaços de dados distribuídos e ter uma cópia do todo em cada um dos processos. Sua sintaxe é a seguinte

```
int MPI_Allgather(
  const void *sendbuf,
  int  sendcount,
  MPI_Datatype sendtype,
  void *recvbuf,
  int recvcount,
  MPI_Datatype recvtype,
  MPI_Comm comm)
```

Note que esta rotina não contém o argumento `root`, pois neste caso todos os processos receberam os dados juntados na variável `recvbuf`!

Exercícios

E 3.3.1. Faça um código MPI para computar a média aritmética simples de n números randômicos em ponto flutuante.

E 3.3.2. Faça um código MPI para computar o produto interno de dois vetores de n elementos randômicos em ponto flutuante.

E 3.3.3. Faça um código MPI para computar a norma L^2 de um vetor de n elementos randômicos em ponto flutuante.

E 3.3.4. Faça um código MPI para computar

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt \tag{3.13}$$

usando a regra composta do ponto médio.

E 3.3.5. Faça uma implementação MPI do método de Jacobi para computar a solução de um sistema $Ax = b$ $n \times n$. Inicialize A e b com números randômicos em ponto flutuante.

3.4 Reduções

Reduções são rotinas que reduzem um conjunto de dados em um conjunto menor de dados. Um exemplo de redução é a rotina de calcular o traço de uma matriz quadrada. Aqui, vamos apresentar algumas soluções MPI de rotinas de redução.

A rotina MPI_Reduce, permite a redução de dados distribuídos em um processo. Sua sintaxe é a seguinte

```
int MPI_Reduce(
  const void *sendbuf,
  void *recvbuf,
  int count,
  MPI_Datatype datatype,
  MPI_Op op,
  int root,
  MPI_Comm comm)
```

O argumento `sendbuf` aponta para o endereço de memória do dado a ser enviado por cada processo, enquando que o argumento `recvbuf` aponta para o endereço de memória onde o resultado da redução será alocada no processo `root`. O argumento `count` é o número de dados enviados por cada processo e `datatype` é o tipo de dado a ser enviado (o qual deve ser igual ao tipo do resultado da redução). O argumento `comm` é o comunicador entre os processos envolvidos na redução. A operação de redução é definida pelo argumento `op` e pode ser um dos seguintes

`MPI_MAX`	`maximum`

```
MPI_MIN             minimum
MPI_SUM             sum
MPI_PROD            product
MPI_LAND            logical and
MPI_BAND            bit-wise and
MPI_LOR             logical or
MPI_BOR             bit-wise or
MPI_LXOR            logical xor
MPI_BXOR            bit-wise xor
MPI_MAXLOC          max value and location
MPI_MINLOC          min value and location
```

Exemplo 3.4.1. No seguinte código `reduce.cc`, cada processo aloca um número randômico na variável `x`. Em seguida, o processo 0 recebe o máximo entre os números alocados em todos os processos (inclusive no processo 0).

Código: reduce.cc

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // API MPI
5  #include <mpi.h>
6
7  int main(int argc, char** argv) {
8
9    // Inicializa o MPI
10   MPI_Init(NULL, NULL);
11
12   // número total de processos
13   int world_size;
14   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
15
16   // ID (rank) do processo
17   int world_rank;
18   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
19
20   // cronometro
21   time_t init = time (NULL);
```

```
22
23    // semente do gerador randômico
24    srand (init + world_rank);
25
26    double x = double (rand ()) / RAND_MAX;
27
28    printf ("%d: %f\n",
29            world_rank, x);
30
31    double y;
32    MPI_Reduce (&x, &y, 1, MPI_DOUBLE,
33                MPI_MAX, 0, MPI_COMM_WORLD);
34
35    if (world_rank == 0)
36      printf ("Máx. entre os números = %f\n",
37              y);
38    // Finaliza o MPI
39    MPI_Finalize();
40
41    return 0;
42  }
```

Segue um teste de rodagem:

```
$ mpic++ reduce.cc
$ mpirun -np 3 a.out
0: 0.417425
1: 0.776647
2: 0.633021
Máx. entre os números = 0.776647
```

Verifique!

No caso de uma *array* de dados, a operação de redução será feita para cada componente. Veja o próximo exemplo.

Exemplo 3.4.2. No `reduce2.cc`, cada processo aloca um vetor com dois números randômicos. Em seguida, o processo 0 recebe o vetor resultante da soma dos vetores alocados em cada processo (inclusive no processo 0).

Código: reduce2.cc

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // API MPI
5  #include <mpi.h>
6
7  int main(int argc, char** argv) {
8
9    // Inicializa o MPI
10   MPI_Init(NULL, NULL);
11
12   // número total de processos
13   int world_size;
14   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
15
16   // ID (rank) do processo
17   int world_rank;
18   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
19
20   // cronometro
21   time_t init = time (NULL);
22
23   // semente do gerador randômico
24   srand (init + world_rank);
25
26   double x[2] = {double (rand ()) / RAND_MAX,
27                  double (rand ()) / RAND_MAX};
28
29   printf ("%d: %f %f\n",
30           world_rank, x[0], x[1]);
31
32   double y[2];
33   MPI_Reduce (&x, &y, 2, MPI_DOUBLE,
34               MPI_SUM, 0, MPI_COMM_WORLD);
35
36   if (world_rank == 0)
```

```
37      printf ("Vetor soma = %f %f\n",
38               y[0], y[1]);
39    // Finaliza o MPI
40    MPI_Finalize();
41
42    return 0;
43  }
```

Segue um teste de rodagem:

```
$ mpic++ reduce2.cc
$ mpirun -np 3 a.out
0: 0.193702 0.035334
Vetor soma = 0.656458 0.843728
1: 0.051281 0.447279
2: 0.411475 0.361114
```

Verifique!

Observação 3.4.1. A rotina MPI_Allreduce executa uma redução de dados e o resultado é alocada em todos os processos. Sua sintaxe é similar a rotina MPI_Reduce, com exceção do argumento `root`, o qual não é necessário nessa rotina. Verifique!

Observação 3.4.2. As rotinas MPI_Ireduce e MPI_Iallreduce são versões assíncronas das rotinas MPI_Reduce e MPI_Allreduce, respectivamente.

Exercícios

E 3.4.1. Faça um código MPI em cada processo aloca um número randômico na variável `x`. Então, o processo 0 recebe o mínimo entre os números alocados em cada processo, inclusive nele mesmo.

E 3.4.2. Faça um código MPI em cada processo aloca um número randômico na variável `x`. Então, o processo 0 recebe o produtório entre os números alocados em cada processo, inclusive nele mesmo.

E 3.4.3. Faça um código MPI para computar a soma dos termos de um vetor de n números randômicos, com $n \gg n_p$, sendo n_p o número de proces-

sos. Use a rotina MPI_Reduce e certifique-se de que cada processo aloque somente os dados necessários, otimizando o uso de memória computacional.

E 3.4.4. Faça um código MPI para computar a norma do máximo de um vetor de n números randômicos, com $n \gg n_p$, sendo n_p o número de processos. Use a rotina MPI_Reduce e certifique-se de que cada processo aloque somente os dados necessários, otimizando o uso de memória computacional.

E 3.4.5. Faça um código MPI para computar a norma L^2 de um vetor de n números randômicos, com $n \gg n_p$, sendo n_p o número de processos. Use a rotina MPI_Allreduce de forma que cada processo contenha a norma computada ao final do código.

E 3.4.6. Faça um código MPI para computar

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt \tag{3.14}$$

usando a regra composta do ponto médio. Use a rotina MPI_Reduce.

3.5 Aplicação: Equação do calor

Nesta seção, vamos construir um código MPI para a resolução da equação do calor pelo método das diferenças finitas. Como um caso teste, vamos considerar a equação do calor

$$u_t - u_{xx} = 0, \quad 0 < x < 1, t > 0, \tag{3.15}$$

com condições de contorno

$$u(0,t) = u(1,t) = 0, t > 0, \tag{3.16}$$

e condições iniciais

$$u(x,0) = \text{sen}(\pi x), 0 \leq x \leq 1, \tag{3.17}$$

onde, $u = u(t,x)$.

Seguindo o método de Rothe[2][3], vamos começar pela discretização no tempo. Para tanto, vamos usar a notação $t_m = m \cdot h_t$, $m = 0,1,2,\ldots,M$, onde h_t

[2]Erich Hans Rothe, 1895 - 1988, matemático alemão. Fonte: Wikipédia.
[3]Também chamado de método das linhas.

é o passo no tempo e M o número total de iterações no tempo. Usando o método de Euler[4], a equação (3.15) é aproximada por

$$u(t_{m+1}, x) \approx u(t_m, x) + h_t u_{xx}(t_m, x), \quad 0 < x < 1, \tag{3.18}$$

onde $u(t_0, x) = u(0, x)$, dada pela condição inicial (3.17).

Agora, vamos denotar $x_i = ih_x$, $i = 0,1,2,\ldots,I$, onde $h_x = 1/I$ é o tamanho da malha (passo espacial). Então, usando a aproximação por diferenças finitas centrais de ordem 2, obtemos

$$u(t_{m+1}, x_i) \approx u(t_m, x_i) + h_t \left[\frac{u(t_m,x_{i-1}) - 2u(t_m, x_i) + u(t_m, x_{i+1})}{h_x^2} \right], \tag{3.19}$$

para $i = 1,2,\ldots,I-1$, observando que, das condições de contorno (3.16), temos

$$u(t_m, x_0) = u(t_m, x_I) = 0. \tag{3.20}$$

Em síntese, denotando $u_i^m \approx u(t_m, x_i)$, temos que uma aproximação da solução do problema de calor acima pode ser calculada pela seguinte iteração

$$u_i^{m+1} = u_i^m + \frac{h_t}{h_x^2} u_{i-1}^m - 2\frac{h_t}{h_x^2} u_i^m + \frac{h_t}{h_x^2} u_{i+1}^m, \tag{3.21}$$

com $u_i^0 = \text{sen}(\pi x_i)$ e $u_0^m = u_I^m = 0$.

Para construirmos um código MPI, vamos fazer a distribuição do processamento pela malha espacial entre os n_p processos disponíveis. Mais precisamente, cada processo $p = 0, 1, 2, \ldots, n_p - 1$, computará a solução u_i^m nos ponto de malha $i = i_p, i_p + 1, \ldots, f_p$, onde

$$i_p = p \left\lfloor \frac{I}{n_p} \right\rfloor, \quad p = 0,1,\ldots,n_p - 1, \tag{3.22}$$

$$f_p = (p+1) \left\lfloor \frac{I}{n_p} \right\rfloor, \quad p = 0,1,\ldots,n_p - 2, \tag{3.23}$$

com $f_{n_p-1} = I$. Ainda, por simplicidade e economia de memória computacional, vamos remover o superíndice, denotando por u a solução na iteração corrente e por u^0 a solução na iteração anterior.

[4]Leonhard Paul Euler, 1707 - 1783, matemático suíço. Fonte: Wikipédia.

Neste caso, a cada iteração $m = 0,1,2,\ldots,M$, o processo $p = 0$, irá computar

$$u_j = u_j^0 + \frac{h_t}{h_x^2}u_{j-1}^0 - 2\frac{h_t}{h_x^2}u_j^0 + \frac{h_t}{h_x^2}u_{j+1}^0, \tag{3.24}$$

para $j = 1,\ldots,f_0$, com as condições de contorno

$$u_0^0 = 0 \tag{3.25}$$
$$u_{f_0+1}^0 = u_{i_1+1}^0. \tag{3.26}$$

Ou seja, este passo requer a comunicação entre o processo 0 e o processo 1.

Os processos $p = 1,2,\ldots,n_p - 2$ irão computar

$$u_j = u_j^0 + \frac{h_t}{h_x^2}u_{j-1}^0 - 2\frac{h_t}{h_x^2}u_j^0 + \frac{h_t}{h_x^2}u_{j+1}^0, \tag{3.27}$$

para $j = i_p + 1,\ldots,f_p$, com as condições de contorno

$$u_{i_p}^0 = u_{f_{p-1}}^0 \tag{3.28}$$
$$u_{f_p+1}^0 = u_{i_{p+1}+1}^0. \tag{3.29}$$

Logo, este passo requer a comunicação entre os processos $p-1$, p e $p+1$.

Ainda, o processo $p = n_p - 1$ deve computar

$$u_j = u_j^0 + \frac{h_t}{h_x^2}u_{j-1}^0 - 2\frac{h_t}{h_x^2}u_j^0 + \frac{h_t}{h_x^2}u_{j+1}^0, \tag{3.30}$$

para $j = i_{n_p-1} + 1,\ldots,f_{n_p-1} - 1$, com as condições de contorno

$$u_{i_{n_p-1}}^0 = u_{f_{n_p-2}}^0 \tag{3.31}$$
$$u_{f_{n_p-1}}^0 = 0. \tag{3.32}$$

O que requer a comunicação entre os processos $n_p - 2$ e o $n_p - 1$.

O código abaixo `calor.cc` é uma implementação MPI deste algoritmo. Cada processo aloca e computa a solução em apenas um pedaço da malha, conforme descrito acima. As comunicações entre os processos ocorrem apenas nas linhas 60-80. Verifique!

Código: calor.cc

```
1  #include <stdio.h>
2  #include <math.h>
3
4  // API MPI
5  #include <mpi.h>
6
7  int main (int argc, char** argv) {
8    // Inicializa o MPI
9    MPI_Init(NULL, NULL);
10
11   // número total de processos
12   int world_size;
13   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
14
15   // ID (rank) do processo
16   int world_rank;
17   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
18
19   // parâmetros
20   size_t M = 1000;
21   size_t I = 10;
22
23   // tamanho dos passos discretos
24   double ht = 1e-3;
25   double hx = 1.0/I;
26   double cfl = ht/(hx*hx);
27
28   // malha espacial local
29   size_t ip = world_rank * int (I/world_size);
30   size_t fp = (world_rank+1) * int (I/world_size);
31   if (world_rank == world_size-1)
32     fp = I;
33   size_t my_I = fp - ip;
34
35   double x[my_I+1];
36   for (size_t j=0; j<=my_I; j++)
```

```
37      x[j] = (ip+j) * hx;
38
39    // solução local
40    double u0[my_I+1], u[my_I+1];
41
42    // condição inicial
43    for (size_t j=0; j<=my_I; j++) {
44      u0[j] = sin (M_PI * x[j]);
45    }
46
47    // condições de contorno de Dirichlet
48    if (world_rank == 0)
49      u[0] = 0.0;
50    if (world_rank == world_size-1)
51      u[my_I] = 0.0;
52
53    // auxiliares
54    double u00, u0I;
55
56
57    // iterações no tempo
58    for (size_t m=0; m<M; m++) {
59
60      if (world_rank == 0) {
61        MPI_Send (&u0[my_I], 1, MPI_DOUBLE,
62                  1, 0, MPI_COMM_WORLD);
63        MPI_Recv (&u0I, 1, MPI_DOUBLE,
64                  1, 0, MPI_COMM_WORLD,
65                  MPI_STATUS_IGNORE);
66      } else if (world_rank < world_size-1) {
67        MPI_Recv (&u00, 1, MPI_DOUBLE,
68                  world_rank-1, 0, MPI_COMM_WORLD,
69                  MPI_STATUS_IGNORE);
70        MPI_Send (&u0[my_I], 1, MPI_DOUBLE,
71                  world_rank+1, 0, MPI_COMM_WORLD);
72
73        MPI_Recv (&u0I, 1, MPI_DOUBLE,
74                  world_rank+1, 0, MPI_COMM_WORLD,
```

```
75                  MPI_STATUS_IGNORE);
76        MPI_Send (&u0[1], 1, MPI_DOUBLE,
77                  world_rank-1, 0, MPI_COMM_WORLD);
78      } else {
79        MPI_Recv (&u00, 1, MPI_DOUBLE,
80                  world_size-2, 0, MPI_COMM_WORLD,
81                  MPI_STATUS_IGNORE);
82        MPI_Send (&u0[1], 1, MPI_DOUBLE,
83                  world_size-2, 0, MPI_COMM_WORLD);
84      }
85
86      // atualização
87      u[1] = u0[1]
88        + cfl * u00
89        - 2*cfl * u0[1]
90        + cfl * u0[2];
91      for (size_t j=2; j<my_I; j++)
92        u[j] = u0[j]
93          + cfl * u0[j-1]
94          - 2*cfl * u0[j]
95          + cfl * u0[j+1];
96      if (world_rank < world_size-1)
97        u[my_I] = u0[my_I]
98          + cfl * u0[my_I-1]
99          - 2*cfl * u0[my_I]
100         + cfl * u0I;
101
102     // prepara nova iteração
103     for (size_t j=0; j<=my_I; j++)
104       u0[j] = u[j];
105
106   }
107
108
109   // Finaliza o MPI
110   MPI_Finalize ();
111
112   return 0;
```

```
113 }
```

No código acima, as comunicações entre os processos foram implementadas de forma cíclica. A primeira a ocorrer é o envio de $u^0_{f_p}$ do processo zero (linhas 61-62) e o recebimento de $u^0_{i_p}$ pelo processo 1 (linhas 67-69). Enquanto essa comunicação não for completada, todos os processos estarão aguardando, sincronizados nas linhas 67-69 e 79-81. Ocorrida a comunicação entre os processos 0 e 1, ocorrerá a comunicação entre o 1 e o 2, entre o 2 e o 3 e assim por diante, de forma cíclica (linhas 67-71 e 79-81).

As últimas comunicações também ocorrem de forma cíclica, começando pelo envio de $u^0_{i_p+1}$ pelo processo n_p-1 (linhas 81-83) e o recebimento de $u^0_{f_p+1}$ pelo processo n_p-2 (linhas 73-75). Em sequência, ocorrem as comunicações entre o processo n_p-2 e o processo n_p-3, entre o n_p-3 e o n_p-4, até o término das comunicações entre o processo 1 (linhas 76-77) e o processo 0 (linhas 63-65).

Exercícios

E 3.5.1. O problema (3.15)-(3.17) tem solução analítica

$$u(t,x) = e^{-\pi^2 t}\,\mathrm{sen}(\pi x). \tag{3.33}$$

Modifique o código `calor.cc` acima, de forma que a acada iteração m no tempo, seja impresso na tela o valor de $u^m(0.5)$ computado e o valor da solução analítica $u(t_m, 0.5)$. Faça refinamentos nas malhas temporal e espacial, até conseguir uma aproximação com três dígitos significativos corretos.

E 3.5.2. Modifique o código `calor.cc` acima, de forma que a cada iteração m no tempo, seja impresso na tela o valor do funcional

$$q(t) = \int_0^1 u(t,x)\,dx. \tag{3.34}$$

Valide os resultados por comparação com a solução analítica (veja o Exercício (3.5.1)).

E 3.5.3. Modifique o código `calor.cc` acima, de forma que as comunicações iniciem-se entre os processos n_p-1 e n_p-2. Ou seja, que a primeira

comunicação seja o envio de $u^0_{i_p+1}$ pelo processo $n_p - 1$ e o recebimento de $u^0_{f_p+1}$ pelo processo $n_p - 2$. E que, de forma cíclica, as demais comunicações ocorram. Poderia haver alguma vantagem em fazer esta modificação?

E 3.5.4. Modifique o código `calor.cc` acima, de forma que as comunicações sejam feitas de forma assíncrona, i.e. usando as rotinas MPI_Isend e MPI_Irecv. Há vantagens em se fazer esta modificação?

E 3.5.5. Faça um código MPI para computar uma aproximação da solução de

$$u_t - u_{xx} = 0, \quad 0 < x < 1, t > 0, \tag{3.35}$$

com condições de contorno periódicas

$$u(0,t) = u(1,t), t > 0, \tag{3.36}$$

e condições iniciais

$$u(x,0) = \text{sen}(\pi x), 0 \leq x \leq 1, \tag{3.37}$$

onde, $u = u(t, x)$.

Resposta dos Exercícios

Referências Bibliográficas

[1] D.P. Dimitri and J.N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 2015.

[2] A. Grama, A. Grupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison Wesley, 2. edition, 2003.