

Matemática Numérica Paralela

Pedro H A Konzen

10 de fevereiro de 2021

Licença

Este trabalho está licenciado sob a Licença Atribuição-CompartilhaIgual 4.0 Internacional Creative Commons. Para visualizar uma cópia desta licença, visite http://creativecommons.org/licenses/by-sa/4.0/deed.pt_BR ou mande uma carta para Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

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	iv
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	23
Respostas dos Exercícios	30
Referências Bibliográficas	31

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](#)

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

$$x(0) = x_0, \quad (1.2)$$

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 levanta 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 consegue 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 instâncias, 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.

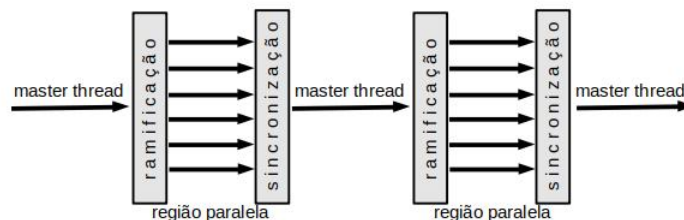


Figura 2.1: Fluxograma de um processo MP.

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 com¹

```
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
```

◇

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

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, 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 | }

◇

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

$$s = \sum_{i=0}^{99999999} 1 \quad (2.1)$$

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 distribuído 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 cláusula `nowait` também pode ser usado 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 científica [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

$$I = \int_{-1}^1 e^{-x^2} dx \quad (2.2)$$

usando a [regra composta do trapézio](#) com n subintervalos uniformes.

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

$$I = \int_{-1}^1 e^{-x^2} dx \quad (2.3)$$

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. Primeira-

mente, 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

$$a_{1,1}x_1 = b_1 \quad (2.4)$$

$$\vdots \quad (2.5)$$

$$a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,i-1}x_{i-1} + a_{i,i}x_i = b_i \quad (2.6)$$

$$\vdots \quad (2.7)$$

$$a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,i}x_i + \cdots + a_{n,n}x_n = b_n \quad (2.8)$$

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 $i = 0, \dots, n - 1$:

(a) Para $j = 0, \dots, i - 1$:

i. $b_i = b_i - A_{i,j}x_j$

(b) $x_i = \frac{b_i}{A_{i,i}}$

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

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

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

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

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

$$t_{fim} = n - 1 \quad (2.12)$$

Segue o pseudocódigo em paralelo.

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

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 $i = 0, \dots, n - 1$.

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("Iniciando ... \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 $j = 0, \dots, n-1$
 - i. Se $j \in \{t_{ini}, \dots, t_{fim}\}$
 - A. $x_j = \frac{b_j}{a_{j,j}}$
 - ii. Para $i \in \{t_{ini}, \dots, t_{fim}\} \wedge \{j+1, \dots, n-1\}$
 - 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!

◇

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

$$a_{1,1}x_{1,k} = \delta_{1,k} \quad (2.13)$$

$$\dots \quad (2.14)$$

$$a_{i,1}x_{1,k} + \dots + a_{i,i-1}x_{i-1,k} + a_{i,i}x_{i,k} = \delta_{i,k} \quad (2.15)$$

$$\dots \quad (2.16)$$

$$a_{n-1,1}x_{1,k} + \dots + a_{n-1,n-1}x_{n-1,k} = \delta_{n-1,k} \quad (2.17)$$

onde, $k = 0, \dots, n - 1$ 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 $x_k = (x_{1,k}, \dots, x_{n-1,k})$ e $b_k = (\delta_{1,k}, \dots, \delta_{n-1,k})$.

1. Região paralela

- (a) Para $k \in \{t_{ini}, \dots, t_{fim}\}$
 - i. resolve $Ax_k = b_k$

◇

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 $n \times n$.

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. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison Wesley, 2. edition, 2003.