Matemática Numérica Paralela

Pedro H A Konzen

25 de janeiro 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++.

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

Pedro H A Konzen

Sumário

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.

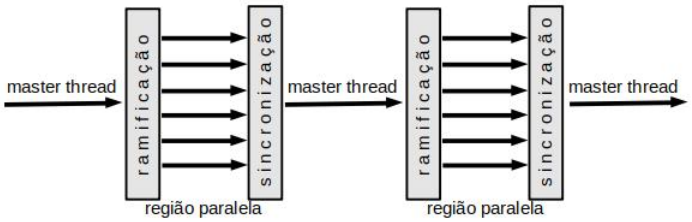

Figura 2.1: Fluxograma de um processo MP.

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 <iostream>
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
```

◊

[1] O 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 $(a - b) + (a/b)$.

Em construção ...

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.