Processamento Paralelo com OpenMP - parte 3

Paralelismo do laço while e o problema da condição de corrida (race condition)

Vimos que o laço *for* tem uma diretiva específica na OpenMP quando se trata de distribuir os valores entre as threads de modo não repetido, mas o laço *while* não tem.

Quando os valores, ou parâmetros, a serem processados seguem uma lei de formação simples (i++, i=2*i, $i=i^2$, etc.), o laço *for* oferece uma implementação mais simples e menos sujeita a *bugs* no mecanismo que manipula a lógica de controle por eventuais esquecimentos do programador. Mas no caso de algoritmos de controle da lógica mais complexos, muitas vezes fica difícil ou mesmo impossível implementar no laço for.

```
1  #include <iostream>
2  #include <stdio.h>
3  #include <math.h>
4  #include <omp.h>
5
6  using namespace std;
7
8  void calculo();
9
10 double y = 456;
11 double x = 10;
12
13 double soma = 0;
14 int cont = 1;
15 bool fim = false;
16 double xObtido;
17
18 int main() {
19 #pragma omp parallel private (fim)
20 {
21     while (true) {
22
23     #pragma omp critical
24     {
25       cont++;
26       if (cont <= 100 && x <= 10000) {
27         calculo();
28         soma += x;
29         xObtido = x;
30       }
31       else {
32         if (cont > 100) cont--;
33         fim = true;
34       }
35     }
```

```
36
37      //=== OUTRAS COISAS PARA FAZER COM O X OBTIDO
38      int tid = omp_get_thread_num();
39      double prov = xObtido;
40      for (int i=1; i<10000; i++) {
41        prov = xObtido + prov / i;
42      }
43
44      #pragma omp critical
45      {
46      printf("\nThread %d", tid);
47      printf(" x modificado = %.15f", prov);
48      }
49
50
51          if (fim) break;
52      }
53  }
54
55      printf("\nSoma %.15f", soma);
56      cout << "\nIteracoes = " << cont;
57
58      return 0;
59  }
60
61  void calculo() {
62      x = x + y / x;
63  }
64
```

Neste exemplo, um laço while (linha 21) executa um cálculo (supostamente complexo) com o valor de uma variável x sendo obtido um novo valor de x, sendo este valor um dos fatores para interrupção do laço além da quantidade de iterações (linha 26).

Os novos valores de x são somados cumulativamente a uma variável *soma* (linha 28).

O laço foi paralelizado (linha 19) e definido um trecho crítico (linha 23-35) que deve ser executado na ordem e por uma thread de cada vez, pois altera um conjunto de variáveis compartilhadas.

Uma coisa que não ressaltamos nas aulas anteriores, mas o faremos agora, é que na programação convencional a sequência das instruções é fundamental para o correto funcionamento do algoritmo, mas na programação paralela poderíamos dizer que é das questões mais críticas a serem analisadas.

O que ocorre é, que pelo fato das threads serem assíncronas entre si, um determinado instante uma thread A pode estar uma instrução à frente da thread B, mas logo a seguir (por razões que não iremos explorar aqui) a thread B passar à frente da A. Quando essas threads pretendem alterar um mesmo recurso, diz-se que estão em "condição de corrida" ou *race condition*.

Por exemplo, na linha 25 em que há o incremento do contador de iterações. Se este incremento estiver fora (antes) da diretiva da linha 23, o que poderia ocorrer?

Exemplo: duas threads, ThA e ThB.

1. ThA incrementa *cont*. Suponhamos que tenha atingido o valor 100.
2. ThB incrementa *cont* antes de ThA testar se *cont* passou de 100 (linha 26).
3. ThA que estaria com o valor de cont==100 e apta a executar o último cálculo, não executa devido à ThB ter incrementado logo em seguida, em que o valor de cont passou a ser 101. Logo, perdeu-se um cálculo.

Experimente fazer este teste. Passe o incremento de *cont* (linha 25) para fora da cláusula *critical*, por exemplo para a linha 22. Execute diversas vezes e observe o resultado final da variável *soma*.

Com relação, ainda, a este contador de iterações (cont), o teste de atingimento de quantidade é feito depois de que o contador é incrementado, portanto há a necessidade de excluir o valor incrementado e não utilizado (linha 32).

Bem, continuando, nosso algoritmo tem outras coisas a fazer com o novo x obtido. Observe que se houver uma grande extensão do algoritmo que não possa ser paralelizada, talvez seja mais vantagem deixá-lo serial. Analisamos isso com o fator *speedup* que veremos mais a frente.

A cláusula atomic

Atomic é semelhante à critical, porém aplicada a uma variável apenas. Por exemplo, para a variável compartilhada qualquer, por exemplo *var1*, ao invés de escrevermos:

```
var1 = x + y;
```

Escreveremos:

```
#pragma omp atomic
      var1 = x + y;
```

Nos sistemas operacionais atuais (e isso já não é de agora), os programas não tem acesso direto à memória. Há diversas construções de software/hardware para este controle, mas, em linhas gerais, o programa precisa solicitar o acesso à determinada posição de memória.

No caso de processos[1] concorrentes, se um processo obteve acesso à variável, o próximo que chegar no, digamos, nanosegundo seguinte, vai encontrar um bloqueio. Se o ambiente tiver um mecanismo de espera, este segundo processo irá aguardar em uma fila, caso contrário podem

[1] Estou generalizando a palavra "processo". Pode ser uma thread.

ocorrer erros nos resultados dos cálculos e até travamento do programa.

Este gerenciamento básico de conflito pode não ter uma otimização ideal, pois, em princípio, é uma situação não esperada. A cláusula *atomic* alerta o compilador de que a variável tem acesso concorrente e, portanto, é criada uma condição de controle mais eficiente[2].

Locks

Análogo ao mutex da PThread, o lock é uma posição de memória que serve de chave para se poder fazer algo ou não poder fazer. É um mecanismo de bloqueio, e a thread que tomar posse de um bloqueio sujeitará as demais a esperar pelo desbloqueio para que possam prosseguir no trecho bloqueado.

```
1   #include <stdio.h>
2   #include <omp.h>
3   #include <math.h>
4
5   int main(){
6     omp_lock_t bloq;
7     omp_init_lock(&bloq);
8
9     double x = 0;
10    double soma = 0;
11
12    #pragma omp parallel for private(x) shared(soma)
13    for (int i = 0; i < 1000; i++ )
14    {
15      double x = 0;
16      for (int j=0; j<10000; j++){
17        x += pow(j,0.5) * sin(j) * cos(j);
18      }
19      omp_set_lock(&bloq);
20      soma += x;
21      omp_unset_lock(&bloq);
22    }
23    omp_destroy_lock(&bloq);
24    printf("\n\nSoma final = %.15f",soma);
25  }
26
27
```

- A linha 6 declara um bloqueio que recebeu o nome de *bloq*.
- A linha 7 inicializa o bloqueio (o deixa pronto para uso).
- Na linha 19 a thread tenta tomar o bloqueio.
- Na linha 21 a thread libera o bloqueio.
- Na linha 23 o bloqueio é destruído.

O objetivo deste bloqueio é prevenir conflitos na alteração da variável *soma*. Experimente remover este bloqueio e execute várias vezes o programa para verificar se ocorre alguma diferença no resultado final.

[2] Evidentemente que não há necessidade de chamar a cláusula *atomic* se a variável estiver em um bloco *critical*.

A cláusula ordered

As threads são executadas de forma assíncrona, de forma que os seus resultados são obtidos à medida em que são concluídas. Como a primeira pode acabar por último, os resultados são publicados nessa ordem.

Em alguns casos pode ser interessante ordenar os resultados. Essa ordenação tem que ter, obviamente, um parâmetro de referência. O amplamente utilizado é o laço *for*. Caso se deseje que uma saída impressa se dê na ordem da geração dos valores do contador, temos que utilizar esta cláusula.

```
1  #include <stdio.h>
2  #include <omp.h>
3  #include <math.h>
4  #include <time.h>
5
6  int main() {
7
8      clock_t inicio = clock();
9      #pragma omp parallel for ordered schedule(dynamic)
10     for (int i=1; i<=100; i++){
11       double calculo = 10;
12       double calculoTemp;
13       for (int j=i; j<10000; j++){
14         calculo += (double) calculo/pow(i, 0.9);
15       }
16       #pragma omp ordered
17       printf("\nThread %d i=%d",  omp_get_thread_num(), i);
18       calculoTemp = calculo;
19     }
20     clock_t fim = clock();
21     double tempo = (fim - inicio) * 1000 / CLOCKS_PER_SEC;
22     printf("\nTempo de execucao = %.2f ms\n\n", tempo);
23     return 0;
24 }
25
```

```
> g++ -o exec main.cpp -fopenmp
> ./exec

Thread 3 i=1
Thread 2 i=2
Thread 1 i=3
Thread 0 i=4
Thread 3 i=5
Thread 2 i=6
Thread 1 i=7
Thread 0 i=8
Thread 3 i=9
Thread 2 i=10
Thread 1 i=11
Thread 0 i=12
Thread 3 i=13
Thread 2 i=14
Thread 1 i=15
Thread 0 i=16
Thread 3 i=17
Thread 2 i=18
Thread 1 i=19
Thread 0 i=20
Thread 3 i=21
Thread 2 i=22
Thread 1 i=23
```

Este código mostra um laço for paralelizado com a cláusula ordered (linha 9) e o trecho que deve ser ordenado (linha 16). À direita tem-se a saída resultante.

Caso se anule a diretiva da linha 16:

```c
#include <stdio.h>
#include <omp.h>
#include <math.h>
#include <time.h>

int main() {

    clock_t inicio = clock();
    #pragma omp parallel for ordered schedule(dynamic)
    for (int i=1; i<=100; i++){
      double calculo = 10;
      double calculoTemp;
      for (int j=i; j<10000; j++){
        calculo += (double) calculo/pow(i, 0.9);
      }
      //#pragma omp ordered
      printf("\nThread %d i=%d",  omp_get_thread_num(), i);
      calculoTemp = calculo;
    }
    clock_t fim = clock();
    double tempo = (fim - inicio) * 1000 / CLOCKS_PER_SEC;
    printf("\nTempo de execucao = %.2f ms\n\n", tempo);
    return 0;
}

```

```
> g++ -o exec main.cpp -fopenmp
> ./exec

Thread 2 i=2
Thread 3 i=1
Thread 1 i=3
Thread 2 i=6
Thread 3 i=5
Thread 0 i=4
Thread 1 i=7
Thread 2 i=10
Thread 0 i=8
Thread 3 i=9
Thread 1 i=11
Thread 2 i=14
Thread 3 i=13
Thread 0 i=12
Thread 1 i=15
Thread 0 i=16
Thread 2 i=18
Thread 3 i=17
Thread 1 i=19
Thread 0 i=20
Thread 0 i=24
Thread 3 i=21
```

A ordem de geração "i" não é levada em conta.

Evidentemente que há certo impacto no desempenho do algoritmo, pois cada thread pode executar por conta própria toda a parte que não tenha sido definida como ordered, mas, no trecho ordered, deverá aguardar chegar sua vez. Então, caso todas as threads estejam tendo suas execuções de "vento em pôpa", as esperas das threads do grupo pela sua vez não devem ser tão significativas, mas, se alguma thread empacar por algum motivo, todo o restante do grupo ficará esperando.

As cláusulas barrier e nowait

Barrier (barreira, em inglês) é um ponto em que as threads do grupo têm que aguardar a chegada de todas as demais para poderem continuar. Algumas diretivas têm uma barreira implícita ao final do seu bloco, outras não.

A diretiva genérica *#pragma omp parallel*, que inicializa as threads, possui uma barreia implícita ao seu final, que, neste caso específico, causa uma espera na thread 0 pela conclusão das demais para que o algoritmo possa prosseguir.

```
13  double salaries1 = 0;
14  double salaries2 = 0;
15
16  #pragma omp parallel shared(salaries1, salaries2)
17  {
18      #pragma omp for
19      for (int employee = 0; employee < 25000; employee++)
20      {
21          salaries1 += fetchTheSalary(employee, Co::Company1);
22      }
23
24      #pragma omp single
25      {
26          std::cout << "Salaries1: " << salaries1 << std::endl;
27      }
28
29      #pragma omp for
30      for (int employee = 0; employee < 25000; employee++)
31      {
32          salaries2 += fetchTheSalary(employee, Co::Company2);
33      }
34  }
35
36  std::cout << "Salaries2: " << salaries2 << std::endl;
37
```

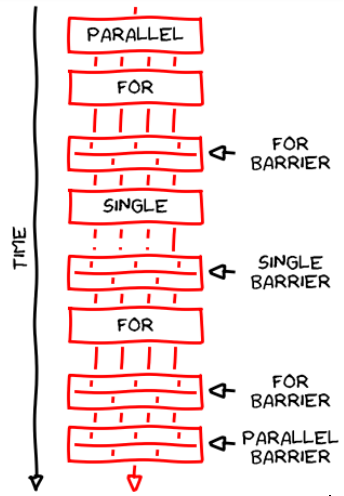

Figura 1 - Exemplo extraído de http://jakascorner.com/blog/2016/07/omp-barrier.html

Neste interessante exemplo, temos diversas diretivas *pragma* que possuem barreiras implícitas. À esquerda, o código e à direita, um esquema do tipo "linha do tempo" mostrando os blocos e respectivas barreiras.

Na linha 16 são inicializadas as threads e na linha 34 essas threads se encerram (continua apenas a thread 0, a master). A linha 34 possui a barreira implícita que obriga a thread 0 a aguardar pelo término das demais do grupo.

Na linha 18 foi aberto um bloco *for* compartilhado para obter a somatória dos salários dos empregados da Empresa 1 (Company1) utilizando o método *fetchTheSalary* que não é apresentado, e, no final deste bloco (linha 22) existe uma barreira implícita, ou seja, todas as threads têm que aguardar até que a última conclua a sua execução deste bloco.

A seguir, na linha 24, vem um cláusula *single* que também impõe uma barreira. Embora apenas uma thread irá executar o bloco single, as demais deverão aguardar sua conclusão.

A linha 29 tem um segundo bloco *for* compartilhado para obter a somatória dos salários dos empregados da Empresa 2 (Company2) que também impõem uma barreira ao final.

Ou seja, tem-se um algoritmo paralelizado, mas com diversas dependências que ocasionam esperas e, consequentemente, reduzem a sua eficiência.

Analisando o programa, podemos observar que o bloco da linha 24 executa uma mera impressão de um resultado que não é condição necessária para início do bloco seguinte. Logo, não é necessário que as outras threads aguardem por isso. Para que a barreira implícita seja eliminada, utiliza-se a cláusula *nowait* (não espere).

#pragma omp single **nowait**

Uma determinada thread, ao atingir este bloco, caso outra já o esteja executando[3], passa direto para a próxima instrução.

Continuando nossa análise, podemos observar que esta impressão não precisa ser executada no meio do processo de contabilização. Pode ficar para o final, junto com a impressão de Salaries2. Com isso pode-se também eliminar a barreira do primeiro bloco for.

O programa otimizado pode ficar assim:

```
13  double salaries1 = 0;
14  double salaries2 = 0;
15
16  #pragma omp parallel shared(salaries1, salaries2)
17  {
18      #pragma omp for nowait
19      for (int employee = 0; employee < 25000; employee++)
20      {
21          salaries1 += fetchTheSalary(employee, Co::Company1);
22      }
23
24      #pragma omp for nowait
25      for (int employee = 0; employee < 25000; employee++)
26      {
27          salaries2 += fetchTheSalary(employee, Co::Company2);
28      }
29  }
30
31  std::cout << "Salaries1: " << salaries1 << std::endl;
32  std::cout << "Salaries2: " << salaries2 << std::endl;
33
34
```

Para finalizar o programa, garantindo que não vá ocorrer nenhum problema com conflitos na atualização das variáveis salaries1 e salaries2, pode-se incluir a cláusula atomic.

```
#pragma omp atomic
      salaries1 += fetchTheSalary(employee, Co::Company1);
```

As cláusulas *critical* e *atomic* não possuem barreiras implícitas.

E, para finalizar este tópico, podemos ter algum trecho do bloco thread antes da barreira implícita final que, por alguma razão na lógica do programa, requeira que todas as threads aguardem para continuar em sincronia. Para isto existe a cláusula *barrier* que levanta uma barreira explícita.

#pragma omp **barrier**

Quando uma thread chega neste ponto, deverá aguardar até que todas cheguem. Só então todas do grupo poderão continuar.

[3] Se nenhuma thread ainda atingiu o bloco single, a primeira a chegar que assume a tarefa.

PROCESSAMENTO PARALELO

prof. Marcio Feitosa

A cláusula task

A *task* foi criada para possibilitar o paralelismo dentro de uma diretiva *single*. Pode parecer estranho, uma vez que *single* prevê que o bloco seja executado por apenas uma thread, mas o bloco *single* pode se compor de partes independentes e, por esta razão, nada impede que outra thead colabore em alguma parte, sempre mantendo, evidentemente, o critério de apenas uma estar executando aquela parte específica. Por exemplo, tem-se um bloco que, pelas conclusões do projetista do software, deverá ser *single* e que se compõe das seguintes tarefas (em macroinstruções):

1. contabilizar faturamento do ano (metodo1())
2. listar produtos mais vendidos (metodo2())
3. listar maiores clientes (metodo3())
4. listar maiores fornecedores (metodo4())

Caso essas quatro tarefas estejam dentro de um bloco thread, deverá ser declarado como single.

```
#pragma omp parallel
{
....
....
#pragma omp single
{
metodo1();
metodo2();
metodo3();
metodo4();
}
...
...
}
```

Se poderia fazer:

```
#pragma omp parallel
{
....
....
#pragma omp single
{
#pragma omp task
{metodo1();}
```

```
#pragma omp task
{metodo2();}
#pragma omp task
{metodo3();}
#pragma omp task
{metodo4();}
}
...
...
}
```

Uma alternativa a esta estrutura seria a divisão em blocos *single* com *nowait*:

```
#pragma omp parallel
{
....
....
#pragma omp single nowait
{metodo1();}
#pragma omp single nowait
{metodo2();}
#pragma omp single nowait
{metodo3();}
#pragma omp single nowait
{metodo4();}
...
...
}
```

Essa alternativa depende, é claro, da estrutura/complexidade do bloco a ser paralelizado.

Uma *task* pode ter sub-tasks, ou tasks filhas.

```
#pragma omp task // task mãe
{
      metodo1();
      #pragma omp task // task filha 1
      {...}
      #pragma omp task // task filha 2
      {...}
      // outras instruções abaixo
      ...
      ...
}
```

Pode ser necessário sincronizar as tasks, de forma que o programa só prossiga após a conclusão de todas declaradas anteriormente. A cláusula taskwait estabelece uma barreira para a conclusão de todas as tasks do mesmo nível. No código anterior, caso as instruções abaixo da *task filha 2* dependam do encerramento da 1 e também do encerramento da 2, utiliza-se a cláusula *taskwait* para aguardar o encerramento de todas[4].

```
#pragma omp task // task mãe
{
      metodo1();
      #pragma omp task // task filha 1
      {...}
      #pragma omp task // task filha 2
      {...}
      #pragma omp taskwait
      // outras instruções abaixo
      ...
      ...
}
```

A *taskwait* é geralmente aplicada em blocos envolvendo tasks filhas, uma vez que uma sequência de tasks principais podem ser sincronizadas pela cláusula *barrier*.

Há ainda a cláusula taskgroup que define um trecho (início-fim) em que é necessário um sincronismo. A cláusula taskwait vai aguardar todas as tasks definidas anteriormente, mas pode-se criar um grupo onde a necessidade de sincronismo começa em um ponto que pode estar após a declaração de outras tasks que não teriam a necessidade de se sincronizarem com essas que vem em seguida.

```
#pragma omp task // 1
{...}
#pragma omp taskgroup
{
      #pragma omp task // 2
      {...}
      #pragma omp task // 3
      {...}
      #pragma omp task // 4
      {...}
}
//-- INSTRUÇÃO DEPENDENTE DO RESULTADO GLOBAL DO TASKGROUP
```

[4] Todas as sub-tasks anteriores à taskwait.

Neste código as tasks 2, 3 e 4, que estão no taskgroup, devem ser concluídas para que as instruções seguintes possam ser executadas.

Para finalizar

Muito bem, existem outras diretivas e variações das já apresentadas, mas o que vimos até o momento já dá boas condições para se trabalhar um projeto de código paralelo.

É importante ter em mente que:

- Nem todo algoritmo paralelizável apresenta bons resultados. Em alguns casos chega a apresentar resultados piores do que na codificação serial.
- Um algoritmo paralelo que apresenta um excelente resultado em determinada plataforma (hardware + sistema operacional) pode não apresentar em outra.

Existem diversas métricas para se verificar qual o ganho que se tem com um programa paralelo em relação à sua versão serial. A medida mais utilizada é o *speedup*, que representa a relação entre o tempo de execução da versão serial e da versão paralela.

Exemplo:

- Execução serial: 77 segundos.
- Execução paralela: 31 segundos.
- $Speedup = \frac{77}{31} = 2{,}48$

Em termos práticos, o *speedup* indica o quanto o programa paralelo é mais rápido que a sua versão serial. No caso do exemplo, a versão paralela tem a execução 2,48 vezes mais rápida.

-0-0-0-0-0-0-0-0-

Exercícios:

1)

Você deverá criar uma matriz quadrada de números inteiros com capacidade suficiente para que se consiga medir os tempos de execução. A matriz deverá ser preenchida com números aleatórios. Para que o algoritmo tenha um trabalho mais significativo ele deverá:

- Obter a somatória de todas as células da matriz (a variável soma é bom que seja do tipo long).
- Obter a quantidade de elementos maiores que determinado valor.

- Obter a quantidade de elementos menores que determinado valor

Crie 3 versões deste algoritmo:

1. Sequencial (sem paralelização nenhuma).
2. Uma versão paralela com laço for.
3. Uma versão paralela com laço while.

Para que seja possível se apurar o speedup da versão 2 e da 3 em relação à versão 1, essas 3 versões precisam estar dentro do mesmo algoritmo, de forma que a execução de uma ou de outra se dê por opção do usuário (tem que ser um laço para que a execução possa ser feita várias vezes sem que a matriz seja recarregada). Ou então são executadas as 3 versões uma seguida da outra, mas é necessário que pergunte se deseja outra execução, caso contrário (reiniciando o programa) a matriz é recarregada com outros valores. Este último modelo pode ser melhor no sentido de que as execuções estão sendo realizadas em um intervalo de tempo em que a carga do servidor (repl.it) se mantém próxima entre as 3 execuções.

Para que o tempo de execução seja apurado, é necessário que várias execuções sejam realizadas para se poder tirar um valor médio.

Nas versões paralelas, utilize primeiramente *atomic* depois *lock* para verificar se ocorrem diferenças nos tempos[5].

Entregue o código e um relatório do seu experimento (Word, Notepad, Open Office).

2)

Para a mesma matriz do exercício 1, divida-a em 4 quadrantes iguais. Implemente o mesmo algoritmo sequencial e a seguir um algoritmo paralelo com uma seção single e, dentro dessa single, quatro tasks, cada uma especializada em um setor.

Os dados a serem obtidos são os mesmos do exercício 1.

Verifique o speedup.

Entregue o código e um relatório do seu experimento (Word, Notepad, Open Office).

Boa sorte.

[5] Não consegui apurar corretamente os tempos no repl.it. De qualquer forma, as proporções para efeito do cálculo do *speedup* devem ser as mesmas.