Programação de Arquiteturas com Memória Compartilhada Utilizando OpenMP - Parte 3

MCZA020-13 - Programação Paralela

Emilio Francesquini
e.francesquini@ufabc.edu.br

2020.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de Programação Paralela na UFABC.
- Estes slides são baseados naqueles produzidos por Peter Pacheco como parte do livro *An Introduction to Parallel Programming* disponíveis em: https://www.cs.usfca.edu/~peter/ipp/
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Algumas figuras foram obtidas em: http://pngimg.com

Produtores e consumidores

Filas

- Podem ser enxergadas como uma abstração de uma fila de consumidores aguardando para pagar pelas suas compras em um mercado
- Uma estrutura natural e que aparece recorrentemente em aplicações multithreaded.
- Por exemplo, suponha que haja diversos threads *produtores* e diversos *threads* consumidores.
  - Produtores produzem, por exemplo, requisições por dados
  - Consumidores podem consumir essas requisições produzindo ou encontrado os dados requisitados.

Passagem de mensagem

UFABC

- Cada thread pode, por exemplo, ter uma fila de mensagens compartilhada e, quando um thread desejar enviar uma mensagem para ou outro, ele pode enfileirar a mensagem na fila de mensagens do thread de destino.
- Um thread pode receber uma mensagem desenfileirando a mensagem que estiver no início da sua fila.

Rascunho da solução

UFABC

```
1  for (sent_msgs = 0; sent_msg < send_max; sent_msgs++) {
2      Send_msg();
3      Try_receive();
4  }
5
6  while (!Done())
7      Try_receive();
```

Enviando mensagens

UFABC

```
1       msg = random();
2       dest = random % thread_count;
3   #   pragma omp critical
4       Enqueue(queue, dest, my_rank, msg);
```

Recebendo mensagens

UFABC

```
1       if (queue_size == 0) return;
2       else if (queue_size == 1)
3   #       pragma omp critical
4           Dequeue(queue, &src, &msg);
5       else
6           Dequeue(queue, &src, &msg);
7       Print_message(src, msg);
```

- Como manter `queue_size` protegido contra condições de corrida?

Detectando a finalização da execução

UFABC

```
1  queue_size = enqueued - dequeued;
2  if (queue_size == 0 && done_sending == thread_count)
3      return TRUE;
4  else
5      return FALSE;
```

Cada thread incrementa `done_sending` quando acabar o seu laço.

Inicialização (1)

UFABC

- Quando o programa começar a executar, um único thread, o thread principal, lê os argumentos da linha de comando e aloca uma array de filas de mensagens com um elemento para cada thread.
- Essa array precisa ser compartilhada entre todos os threads já que cada thread pode enviar mensagens para qualquer outro thread e, portanto, pode enfileirar mensagens em qualquer uma das filas.

Inicialização (2)

UFABC

- Um ou mais threads podem terminar de alocar as suas filas antes dos demais.
- Precisamos de uma barreira explícita para que quando um thread encontrar a barreira ele fique esperando até que os demais threads do time atinjam a barreira também.
- Depois que todos os threads tiverem alcançado a barreira, então todos os membros do time podem prosseguir.

```
1 #pragma omp barrier
```

Mas pera aí!

UFABC

```
1  queue_size = enqueued - dequeued;
2  if (queue_size == 0 && done_sending == thread_count)
3      return TRUE;
4  else
5      return FALSE;
```

- E como incrementar `done_sending`?

A diretiva `atomic`

UFABC

```
1  #pragma omp atomic
```

- Ao contrário da diretiva **`critical`**, a diretiva **`atomic`** pode apenas proteger seções críticas que consistem de uma simples atribuição em C.
- Além disto, os formatos aceitos pela diretiva **`atomic`** são os seguintes:

```
1  x <op>= <expressão>; //Não pode fazer referência a x
2  x++;
3  x--;
4  ++x;
5  --x;
6
7  #pragma omp atomic
8  x += y++; //Protege escrita a x mas não a y
```

A diretiva `atomic`

UFABC

- `<op>` pode ser um dos seguintes operadores binários
  - `+ - * / & ^ | << >>`
- Muitos processadores oferecem instruções especiais e atômicas que fazem *load-modify-store* atomicamente
- Uma seção crítica com esse formato pode ser executada de maneira muito mais eficiente do que através de construções mais elaboradas e gerais para isolamento de sessões críticas.

Seções críticas em OpenMP

UFABC

Temos, neste momento, 3 seções críticas em nosso programa

- `done_sending++;`
- `Enqueue(q_p, my rank, mesg);`
- `Dequeue(q_p, &src, &mesg);`

Mas nem todas elas precisam ser isoladas:

- `done_sending++;` é independente das demais
- Entre o segundo e o terceiro não é preciso que sejam totalmente isolados
  - O Thread 0 pode estar enfileirando na caixa de mensagens do Thread 2
  - Enquanto isto o Thread 5 poderia estar enfileirando na caixa de mensagens do Thread 0

Seções críticas em OpenMP

UFABC

- Para o OpenMP temos neste momento 2 seções críticas
  - `done_sending++;` que é protegido pelo atomic
  - `Enqueue(q_p, my_rank, mesg);` e `Dequeue(q_p, &src, &mesg);` protegidos pelo `critical`
- Como todos os blocos que estão dentro do `critical` serão serializados, isso pode ser muito ruim para o desempenho.

Seções críticas em OpenMP

UFABC

- OpenMP tem suporte ao uso de nomes para seções críticas

```
1 #pragma omp critical(name)
```

- Quando fazemos isto, blocos protegidos por diretivas `critical` com nomes diferentes podem ser executadas simultaneamente.
- Contudo, nomes e regiões são definidos em tempo de compilação. Não é possível criar regiões críticas (tal qual fizemos com Pthread mutexes) em tempo de execução.
- Logo não resolve o nosso problema para as filas (pois o número de filas é definido pelo usuário como um parâmetro).

Travas (*locks*)

UFABC

- Uma trava (*lock*) é uma estrutura de dados e funções associadas que permitem o programador garantir exclusão mútua em uma seção crítica.
- Rascunho do modo de uso:

```
// Executado por uma thread
Inicializa a estrutura de lock;
...
// Executado por múltiplas threads
Tenta obter a trava;
Seçao crítica;
Destrava;
...
// Executado por uma thread
Destrói a estrutuda de lock;
```

Travas (*locks*)

UFABC

```
1  void omp_init_lock(omp_lock_t* lock_p     /* out */);
2  void omp_set_lock( omp_lock_t* lock_p     /* in/out */);
3  void omp_unset_lock(omp_lock_t* lock_p    /* in/out */);
4  void omp_destroy_lock(omp_lock_t* lock_p /* in/out */);
```

Solução usando locks

UFABC

Saímos disto:

```
1  #  pragma omp critical
2     Enqueue(queue, dest, my_rank, msg);
```

Para isto:

```
1  omp_set_lock(&q_p->lock);
2  Enqueue(queue, dest, my_rank, msg);
3  omp_unset_lock(&q_p->lock);
```

Solução usando locks

UFABC

Saímos disto:

```
1  #  pragma omp critical
2     Dequeue(q_p, &src, &msg);
```

Para isto:

```
1  omp_set_lock(&q_p->lock);
2  Dequeue(q_p, &src, &msg);
3  omp_unset_lock(&q_p->lock);
```

Algumas ressalvas...

UFABC

1. Você não deveria misturar diferentes tipos de exclusão mútua para uma única seção crítica.
2. O OpenMP não garante que as travas sejam justas.
3. Pode ser perigoso aninhar estruturas de exclusão mútua diferentes.

```
1  #pragma omp critical
2  y = f (x);
3  // E em outra thread
4  #pragma omp atomic
5  y++;
6  ...
7  double f(double x) { //chamado na linha 2
8  #    pragma omp critical
9       z = g(x); /* z é compartilhado */
10      ...
11 }
```

Exemplo: Multiplicação Matriz-Vetor

UFABC

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{bmatrix}$$

```
1  for (i = 0; i < m ; i++) {
2      y [i] = 0.0;
3      for (j = 0; j < n; j++)
4          y[i] += A[i][j]*x[j];
5  }
```

Exemplo: Multiplicação Matriz-Vetor

UFABC

```
1  #    pragma omp parallel for num_threads(thread_count) \
2           default(none) private(i,j) shared(A, x, y, m, n)
3       for (i = 0; i < m ; i++) {
4           y [i] = 0.0;
5           for (j = 0; j < n; j++)
6               y[i] += A[i][j]*x[j];
7       }
```

Table 5.4 Run-Times and Efficiencies of Matrix-Vector Multiplication (times in seconds)

	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Thread Safety

UFABC

```
1  void Tokenize(
2    char* lines[] /* in/out */,
3    int line count /* in */,
4    int thread count /* in */) {
5    int my_rank, i, j;
6    char *my_token;
7
8    ...
```

Thread Safety

UFABC

```
1   # pragma omp parallel num threads(thread_count) \
2       default(none) private(my_rank, i, j, my_token) \
3       shared(lines, line_count)
4     { my_rank = omp_get_thread_num();
5   #     pragma omp for schedule(static, 1)
6       for (i = 0; i < line_count; i++) {
7         printf("Thread %d > line %d = %s", my_rank, i,
8                lines[i]);
9         j = 0;
10        my_token = strtok(lines[i], " \t\n");
11        while (my_token != NULL) {
12          printf("Thread %d > token %d = %s\n", my_rank,
13                 j, my_token);
14          my_token = strtok(NULL, " \t\n");
15          j++;
16        }
17      } /* for i */
18    } /* omp parallel */
19  } /* Tokenize */
```

Conclusões

Conclusões

UFABC

- OpenMP é uma padrão para a programação de sistemas com memória compartilhada
- OpenMP utiliza tanto funções especiais quanto diretivas do preprocessador chamadas de pragmas
- Programas OpenMP iniciam diversos threads e não diversos processos
- Diversas diretivas do OpenMP pode ser modificadas por cláusulas

Conclusões

- Um dos maiores problemas para o desenvolvimento de programas em arquiteturas com memória compartilhada é a ocorrência de condições de corrida
- OpenMP provê diversos mecanismos para garantir exclusão mútua em seções críticas
  - Diretivas `critical`
  - Diretivas `critical` nomeadas
  - Diretivas `atomic`
  - Travas (*locks*) simples

Conclusões

UFABC

- Por padrão, a maior parte das implementações utiliza um particionamento por bloco das iterações do laço
- OpenMP oferece uma variedade de opções de escalonamento
- Em OpenMP o escopo da variável é a coleção de threads a partir dos quais a variável é acessível
- Uma redução é uma computação que repetidamente aplica a mesma operação de redução a uma sequência de operandos para obter um único resultado.