Arquiteturas Paralelas e OpenMP

Natanael Ramos
Rodolfo Labiapari Mansur Guimarães

14 de dezembro de 2015

Sumário

Recapitulando

Recapitulando sobre OpenMP

- Inserção do cabeçalho `omp.h`
- Forma de uso `# pragma omp <diretivas>`

Principais Diretivas	**Diretivas de Partilha de Dados**
`parallel`	`default(shared | none)`
`num_threads(<inteiro>)`	`shared(<lista_de_variaveis>)`
`parallel for`	`private(<lista_de_variaveis>)`
`for`	`reduction(<operadores>: <lista_de_variaveis>)`
Diretivas de Sincronização	**Funções da Biblioteca** <omp.h>
`critical`	`omp_get_thread_num()`
	`omp_get_num_threads()`

Ordenação

Bubble Sort - Loop-Carried Dependence

```
1 for (list_length = n; list_length >= 2; list_length)
2   for (i = 0; i < list_length - 1; i++)
3     if (a[i] > a[i+1]) {
4       tmp = a[i];
5       a[i] = a[i+1];
6       a[i+1] = tmp;
7     }
```

Bubble Sort - Loop-Carried Dependence

```
1 for (list_length = n; list_length >= 2; list_length)
2   for (i = 0; i < list_length - 1; i++)
3     if (a[i] > a[i+1]) {
4       tmp = a[i];
5       a[i] = a[i+1];
6       a[i+1] = tmp;
7     }
```

- Existe uma dependência no loop externo
  - Em qualquer iteração, o loop contém listas de dependência com iterações passadas.
- Exemplo
  - Se na primeira execução temos `a = {3, 4, 1, 2}`, e na segunda teremos `a = {3, 1, 2, 4}`.

Bubble Sort - Loop-Carried Dependence

```
1 for (list_length = n; list_length >= 2; list_length)
2   for (i = 0; i < list_length - 1; i++)
3     if (a[i] > a[i+1]) {
4       tmp = a[i];
5       a[i] = a[i+1];
6       a[i+1] = tmp;
7     }
```

- Existe uma dependência no loop externo
  - Em qualquer iteração, o loop contém listas de dependência com iterações passadas.
- Exemplo
  - Se na primeira execução temos `a = {3, 4, 1, 2}`, e na segunda teremos `a = {3, 1, 2, 4}`.
  - O `4` **é removido das próximas iterações**.

Bubble Sort - Loop-Carried Dependence

```
1 for (list_length = n; list_length >= 2; list_length)
2   for (i = 0; i < list_length - 1; i++)
3     if (a[i] > a[i+1]) {
4       tmp = a[i];
5       a[i] = a[i+1];
6       a[i+1] = tmp;
7     }
```

- Existe uma dependência no loop externo
  - Em qualquer iteração, o loop contém listas de dependência com iterações passadas.
- Exemplo
  - Se na primeira execução temos `a = {3, 4, 1, 2}`, e na segunda teremos `a = {3, 1, 2, 4}`.
  - O `4` **é removido das próximas iterações**.
  - Se a segunda iteração executar antes, na próxima iteração o `4` **estará disponível para permutação** sendo um erro de execução.

Bubble Sort - Loop-Carried Dependence

```
1 for (list_length = n; list_length >= 2; list_length)
2    for (i = 0; i < list_length 1; i++)
3       if (a[i] > a[i+1]) {
4          tmp = a[i];
5          a[i] = a[i+1];
6          a[i+1] = tmp;
7       }
```

- Não só no externo, mas existe outra dependência no loop interno
  - Elementos da iteração i também são utilizados na iteração $i - 1$;
- Exemplo de solução ao problema
  - Se na iteração $i - 1$ **não houver** troca entre `a[i-1]` e `a[i]`, então `a[i]` e `a[i+1]` podem ser comparados.
  - Se na iteração $i - 1$ **houver** troca entre `a[i-1]` e `a[i]`, então `a[i-1]` e `a[i+1]` podem ser comparados.

Bubble Sort

- Entretanto, reescrever o algoritmo não é algo fácil.
- **Encontrar** dependências é relativamente fácil.
- O difícil (ou impossível) é tratá-las.

- A diretiva `parallel for` não é solução pra tudo.

- O Algoritmo Bubble Sort **não pode ser paralelizável** utilizando esta diretiva.

Odd-Even Transposition Sort

```
1 for (phase = 0; phase < n; phase++)
2    if (phase % 2 == 0)
3       for (i = 1; i < n; i += 2)
4          if (a[i-1] > a[i]) Swap(&a[i-1],&a[i]);
5    else
6       for (i = 1; i < n 1; i += 2)
7          if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

Odd-Even Transposition Sort

```
1 for (phase = 0; phase < n; phase++)
2    if (phase % 2 == 0)
3       for (i = 1; i < n; i += 2)
4          if (a[i-1] > a[i]) Swap(&a[i-1],&a[i]);
5    else
6       for (i = 1; i < n 1; i += 2)
7          if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

- Similar ao Bubble, mas com **mais oportunidades de paralelismo**.
  - **Par:**
    - Cada `a[i]` ímpar é comparado com `a[i-1]`.
    - Se for menor, são permutados.
  - **Ímpar:**
    - Cada `a[i]` ímpar é comparado com `a[i+1]`.
    - Se for maior, são permutados.
- Depois de *n* fases, a lista estará ordenada.

Odd-Even Transposition Sort

```
1 for (phase = 0; phase < n; phase++)
2    if (phase % 2 == 0)
3       for (i = 1; i < n; i += 2)
4          if (a[i-1] > a[i])
5             Swap(&a[i-1],&a[i]);
6    else
7       for (i = 1; i < n 1; i += 2)
8          if (a[i] > a[i+1])
9             Swap(&a[i], &a[i+1]);
```

Table 5.1 Serial Odd-Even Transposition Sort

	Subscript in Array						
Phase	**0**		**1**		**2**		**3**
0	9	↔	7		8	↔	6
	7		9		6		8
1	7		9	↔	6		8
	7		6		9		8
2	7	↔	6		9	↔	8
	6		7		8		9
3	6		7	↔	8		9
	6		7		8		9

Figura 1: Algoritmo de Ordenação Serial de Transposição Ímpar-Par.

Odd-Even Transposition Sort - Loop-Carried Dependence

```
1 for (phase = 0; phase < n; phase++)
2    if (phase % 2 == 0)
3       for (i = 1; i < n; i += 2)
4          if (a[i 1] > a[i]) Swap(&a[i 1],&a[i]);
5    else
6       for (i = 1; i < n 1; i += 2)
7          if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

- Já o loop interno não aparenta ter dependências.
- Já no externo...

Odd-Even Transposition Sort - Loop-Carried Dependence

```
1 for (phase = 0; phase < n; phase++)
2    if (phase % 2 == 0)
3       for (i = 1; i < n; i += 2)
4          if (a[i 1] > a[i]) Swap(&a[i 1],&a[i]);
5    else
6       for (i = 1; i < n 1; i += 2)
7          if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

- Já o loop interno não aparenta ter dependências.
- Já no externo... Supondo o vetor `a = {9, 7, 8, 6}`.
- Sequencial
  - ***Fase 0*:** (9,7) e (8,6) resultando no vetor `a = {7, 9, 6, 8}`.
  - ***Fase 1*:** (6, 9) são permutados.

Odd-Even Transposition Sort - Loop-Carried Dependence

```
1 for (phase = 0; phase < n; phase++)
2    if (phase % 2 == 0)
3       for (i = 1; i < n; i += 2)
4          if (a[i 1] > a[i]) Swap(&a[i 1],&a[i]);
5    else
6       for (i = 1; i < n 1; i += 2)
7          if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

- Já o loop interno não aparenta ter dependências.
- Já no externo... Supondo o vetor `a = {9, 7, 8, 6}`.
- Sequencial
  - ***Fase 0*:** (9,7) e (8,6) resultando no vetor `a = {7, 9, 6, 8}`.
  - ***Fase 1*:** (6, 9) são permutados.
- Paralelo executará todas as fases concorrentemente.
  - Assim, na *fase 1*, será comparado o par (7,8) e não o (6, 9).

Odd-Even Transposition Sort - Impasses

- Por mais que os loops internos não tenham nenhum tipo de dependência, deve-se atentar ao loop externo.
- **Potencial 1:**
  - Para que a diretiva `parallel for` funcione, deve-se ter certeza que todas as threads da *fase p* terminaram antes do início de *fase p+1*.
- **Potencial 2:**
  - *Overhead* de *forking* e *joining* de threads.
- Tem-se duas implementações paralelas sobre este algoritmo:

Odd-Even Transposition Sort - Implementação 1

```
1  for (phase = 0; phase < n; phase++) {
2     if (phase % 2 == 0)
3  #     pragma omp parallel for num_threads(thread_count) \
4           default(none) shared(a, n) private(i, tmp)
5        for(i=1;i<n;i+=2) {
6           if (a[i-1] > a[i]) {
7              tmp = a[i-1];
8              a[i-1] = a[i];
9              a[i] = tmp;
10          }
11       } else
12 #     pragma omp parallel for num_threads(thread_count) \
13          default(none) shared(a, n) private(i, tmp)
14       for(i=1;i<n 1;i+=2) {
15          if (a[i] > a[i+1]) {
16             tmp = a[i+1];
17             a[i+1] = a[i];
18             a[i] = tmp;
19          }
20       }
21 }
```

- O `for` externo não foi paralelizado.
- Cada `for` interno foi paralelizado separadamente.

Odd-Even Transposition Sort - Implementação 2

```
#    pragma omp parallel num_threads(thread_count) \
      default(none) shared(a, n) private(i, tmp, phase)
for (phase = 0; phase < n; phase++) {
   if (phase % 2 == 0)
#      pragma omp for
      for(i=1;i<n;i+=2) {
         if (a[i-1] > a[i]) {
            tmp = a[i-1];
            a[i-1] = a[i];
            a[i] = tmp;
         }
      } else
#      pragma omp for
      for(i=1;i<n 1;i+=2) {
         if (a[i] > a[i+1]) {
            tmp = a[i+1];
            a[i+1] = a[i];
            a[i] = tmp;
         }
      }
}
```

- O algoritmo inteiro foi paralelizado.
- Existe uma nova diretiva em cada `for` interno.

Odd-Even Transposition Sort - Comparações

Table 5.2 Odd-Even Sort with Two `parallel for` Directives and Two `for` Directives (times are in seconds)

`thread_count`	1	2	3	4
Two `parallel for` directives	0.770	0.453	0.358	0.305
Two `for` directives	0.732	0.376	0.294	0.239

Figura 2: Comparação em segundos entre o mesmo algoritmo sobre diretivas diferentes.

- A diferença é pelo motivo
  - O primeiro algoritmo realiza o *forking* e *joining* **para cada iteração**.
  - Já o segundo **reutiliza as mesmas threads** já que quantidade é sempre a mesma.
- Assim, é possível criar um time de threads **antes de loops** com a diretiva `parallel`.
- A diretiva `for`, diferentemente de `parallel for`, **não *fork* nenhuma threads**. Ela utiliza threads já prontas para uso.
- A segunda implementação obteve uma melhoria de 17%.

Escalonamento de Loops

Escalonamento de Loops - Introdução

- A atribuição das iterações para as threads é dependente do sistema.
- OpenMP possui uma implementação rude de particionamento de bloco:
  - n iterações. Então $n/\texttt{thread\_count}$ iterações serão atribuído a thread 0, ou seja n/t.
  - Os próximos $n/\texttt{thread\_count}$ serão atribuído a thread 1, ou seja $n/2t$.
  - Os próximos $n/\texttt{thread\_count}$ serão atribuído a thread 2, ou seja $n/4t$.
- Algumas atribuições de iteração de threads serão menos otimizadas.
- Existem **inúmeras situações** onde isso **não seria ideal** tal como o exemplo abaixo.

```
1 sum = 0.0;
2 for (i = 0; i <= n; i++)
3    sum += f(i);
```

- Supondo que o 'esforço' da função `f` **depende da grandeza de** $|i|$.
- A divisão de blocos dará muito mais tarefas para a thread $\texttt{thread\_count} - 1$ do que a thread 0.

Escalonamento de Loops - Round Robin

- Uma melhor atribuição é utilizando **particionamento cíclico**.
  - **A iteração é atribuída, uma em cada vez**, no escalonamento “round-robin”.

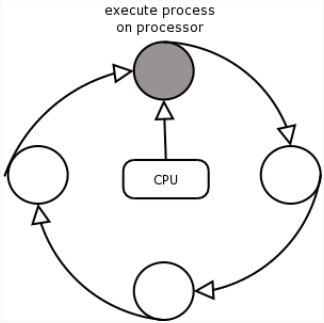

Figura 3: Algoritmo de Escalonamento de Processo Round-Robin.

Escalonamento de Loops - Round Robin

- Uma melhor atribuição é utilizando **particionamento cíclico**.
  - **A iteração é atribuída, uma em cada vez**, no escalonamento “round-robin”.

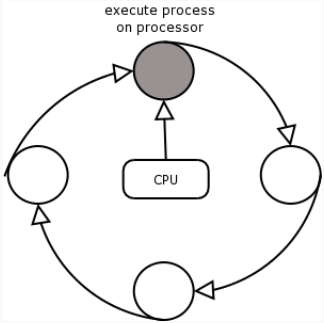

Figura 3: Algoritmo de Escalonamento de Processo Round-Robin.

Thread	Iterations
0	0, n/t, 2n/t, …
1	1, n/t + 1, 2n/t + 1, …
⋮	⋮
t − 1	t − 1, n/t + t − 1, 2n/t + t − 1, …

Figura 4: Divisão das iterações com o algoritmo Round-Robin no exemplo acima.

Escalonamento de Loops - Comparação

```
1 sum = 0.0;
2 for (i = 0; i <= n; i++)
3    sum += f(i);
```

Escalonamento de Loops - Comparação

```
1 sum = 0.0;
2 for (i = 0; i <= n; i++)
3    sum += f(i);
```

```
1 double f(int i) {
2    int j, start = i*(i+1)/2, finish = start + i;
3    double return_val = 0.0;
4    for (j = start; j <= finish; j++) {
5       return val += sin(j);
6    }
7    return return_val;
8 } /*f*/
```

- **Definição:** Se `f(i)` leva tempo *i*, `f(2i)` levará o dobro.

Escalonamento de Loops - Comparação

```
1 sum = 0.0;
2 for (i = 0; i <= n; i++)
3    sum += f(i);
```

```
1 double f(int i) {
2    int j, start = i*(i+1)/2, finish = start + i;
3    double return_val = 0.0;
4    for (j = start; j <= finish; j++) {
5       return val += sin(j);
6    }
7    return return_val;
8 } /*f*/
```

- **Definição:** Se `f(i)` leva tempo i, `f(2i)` levará o dobro.
- **Sem escalonamento cíclico** com $n = 10.000$:
  - Uma única thread, leva-se $3,57s$.
  - Duas threads, os blocos são divididos e leva-se $2,76s$. *Speedup* de 1,33.

Escalonamento de Loops - Comparação

```
1 sum = 0.0;
2 for (i = 0; i <= n; i++)
3    sum += f(i);
```

```
1 double f(int i) {
2    int j, start = i*(i+1)/2, finish = start + i;
3    double return_val = 0.0;
4    for (j = start; j <= finish; j++) {
5       return val += sin(j);
6    }
7    return return_val;
8 } /*f*/
```

- **Definição:** Se `f(i)` leva tempo i, `f(2i)` levará o dobro.
- **Sem escalonamento cíclico** com $n = 10.000$:
  - Uma única thread, leva-se $3,57s$.
  - Duas threads, os blocos são divididos e leva-se $2,76s$. *Speedup* de 1,33.
- **Com escalonamento cíclico** com $n = 10.000$
  - Com 2 threads vai para $1,84s$. *Speedup* de 1,99 e 1,5 respectivamente.
- Assim, existe a cláusula de escalonamento `schedule` para diretivas `for`.

Cláusula `schedule`

- Escalonamento padrão usa-se somente a cláusula `reduction` com *algum* `for`

```
1 sum = 0.0;
2 # pragma omp parallel for num_threads(thread_count) \
3    reduction(+:sum)
4 for (i = 0; i <= n; i++)
5    sum += f(i);
```

- Para um escalonamento cíclico utiliza-se também a cláusula `schedule`

```
1 sum = 0.0;
2 # pragma omp parallel for num_threads(thread_count) \
3    reduction(+:sum) schedule(static,1)
4 for (i = 0; i <= n; i++)
5    sum += f(i);
```

Cláusula `schedule`

- Possui a forma `schedule(<type> [, <chunksize>])` sendo
- **O parâmetro** `type`
  - `static`: Iterações podem ser atribuídas **antes da sua execução**;
  - `dynamic` ou `guided`: Iterações podem ser atribuídas **enquanto o loop é executado**
    - Se depois que uma thread é completada na iteração atual, pode requisitar mais em tempo de execução.
  - `auto`: O **compilador e sistema determinará** o escalonamento.
  - `runtime`: Será determinado **em tempo de execução**.
- **O parâmetro** `chunksize`
  - **Definição:** *chunk* de um iteração é **um bloco de iterações que pode ser executado num loop** serial.
  - **Definição:** `chunksize` é o número de iterações no bloco.
  - Somente `static`, `dynamic` e `guided` possuem essa opção.

Cláusula `schedule` - `static`

- O sistema atribui iterações *chunk* de `chunksize` para cada thread por meio do escalonamento cíclico.
- Exemplo com 12 iterações (`a = {0, 1, ... , 11}`)

Cláusula `schedule` - `static`

- O sistema atribui iterações *chunk* de `chunksize` para cada thread por meio do escalonamento cíclico.
- Exemplo com 12 iterações (`a = {0, 1, ... , 11}`)
  - **Se chunksize for 1** (`schedule(static, 1)`)
    - **Thread 0:** 0, 3, 6, 9
    - **Thread 1:** 1, 4, 7, 10
    - **Thread 2:** 2, 5, 8, 11

Cláusula `schedule` - `static`

- O sistema atribui iterações *chunk* de `chunksize` para cada thread por meio do escalonamento cíclico.
- Exemplo com 12 iterações (`a = {0, 1, ... , 11}`)
  - **Se chunksize for 1** (`schedule(static, 1)`)
    - **Thread 0:** 0, 3, 6, 9
    - **Thread 1:** 1, 4, 7, 10
    - **Thread 2:** 2, 5, 8, 11
  - **Se chunksize for 2** (`schedule(static, 2)`)
    - **Thread 0:** 0, 1, 6, 7
    - **Thread 1:** 2, 3, 8, 9
    - **Thread 2:** 4, 5, 10, 11

Cláusula `schedule` - `static`

- O sistema atribui iterações *chunk* de `chunksize` para cada thread por meio do escalonamento cíclico.
- Exemplo com 12 iterações (`a = {0, 1, ... , 11}`)
  - **Se chunksize for 1** (`schedule(static, 1)`)
    - **Thread 0:** 0, 3, 6, 9
    - **Thread 1:** 1, 4, 7, 10
    - **Thread 2:** 2, 5, 8, 11
  - **Se chunksize for 2** (`schedule(static, 2)`)
    - **Thread 0:** 0, 1, 6, 7
    - **Thread 1:** 2, 3, 8, 9
    - **Thread 2:** 4, 5, 10, 11
  - **Se chunksize for 4** (`schedule(static, 4)`)
    - **Thread 0:** 0, 1, 2, 3
    - **Thread 1:** 4, 5, 6, 7
    - **Thread 2:** 8, 9, 10, 11
- Se o `chucksize` for omitido, será usado uma aproximação de $i/$`thread_count`

Cláusula `schedule` - `dynamic` e `guided`

- **Dynamic**
  - Iterações são quebradas em chunks de `chunksize` de iterações consecutivas.
  - Cada thread executa um chunk e quando ela termina, ela requisita outro em tempo de execução.
  - Isso é executado até que as iterações sejam completadas.
- **Guided**
  - Tal como o `dynamic`.
  - Quando um chunk é completado, o tamanho do novo chunk é decrementado.
  - Exemplo, de $n = 10.000$ e 2 threads:
    - A primeira thread ficará com $9999/2 \approx 5000$;
    - A segunda ficará com $4999/2 \approx 2500$ e assim segue.

Cláusula `schedule` - `dynamic` e `guided`

Table 5.3 Assignment of Trapezoidal Rule Iterations 1–9999 using a `guided` Schedule with Two Threads

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1–5000	5000	4999
1	5001–7500	2500	2499
1	7501–8750	1250	1249
1	8751–9375	625	624
0	9376–9687	312	312
1	9688–9843	156	156
0	9844–9921	78	78
1	9922–9960	39	39
1	9961–9980	20	19
1	9981–9990	10	9
1	9991–9995	5	4
0	9996–9997	2	2
1	9998–9998	1	1
0	9999–9999	1	0

Figura 5: Tabela de da quantidade de chunks.

Cláusula `schedule` - `runtime`

- Antes, deve-se definir variável de ambiente
  - Valores nomeados que podem ser acessados em tempo de execução.
  - Disponível no ambiente do programa.
- Quando `schedule(runtime)` é especificado, o sistema utiliza a variável de ambiente `OMP_SCHEDULE`.
  - Ela pode ter qualquer valor que poderia ser usado em `static, dynamic`, ou `guided`.
  - `$ export OMP_SCHEDULE=‘‘static,1’’`

Cláusula `schedule` - Como decidir?

- Que tipo de escalonamento e chunksize deveríamos usar?
- Como é possível perceber, existe alguns tipos de overheads em alguns tipos de escalonamento
  1. A sobrecarga é maior para dynamic do que static.
  2. Há sobrecarga no guided em árvores.
  3. Caso contrário, se achar que o escalonamento pode ser melhorado, deverá tentar abordagens diferentes.
  4. Se deseja-se um desempenho satisfatório sem o uso de `schedule`, deve-se ir além do uso de escalonamentos.
- Como no primeiro exemplo, se a melhoria de 1,99 de *speedup* é satisfatório, deve-se parar.
- Entretanto, se for variar o número de threads e o número de iterações, deve ser experimentado outras possibilidades.
- Achar o 'ótimo escalonável' está diretamente relacionado ao número de iterações e threads.

Cláusula `schedule` - Explorar Escalonadores

- Se a performance do escalonador padrão não tiver aceitável, pode-se trocar o tipo de escalonador
  - Existe uma lista ampla de escalonadores.

- Algumas ideias para explorar alguns escalonadores antes de outros
  1. Se cada iteração exige a **mesma quantidade de computação**, então o escalonador padrão dará melhor performance.
  2. Se o custo das iterações cresce ou decresce de forma linear, então o `static` com chunk-size pequeno dará grande performance.
  3. Se o custo não pode ser determinado
     - Pode-se fazer variações de escalonamento.
     - O `schedule(runtime)` também pode ser utilizado também.

Cláusula `schedule` - Explorar Escalonadores

- Se a performance do escalonador padrão não tiver aceitável, pode-se trocar o tipo de escalonador
  - Existe uma lista ampla de escalonadores.
- Algumas ideias para explorar alguns escalonadores antes de outros
  1. Se cada iteração exige a **mesma quantidade de computação**, então o escalonador padrão dará melhor performance.
  2. Se o custo das iterações cresce ou decresce de forma linear, então o `static` com chunk-size pequeno dará grande performance.
  3. Se o custo não pode ser determinado
     - Pode-se fazer variações de escalonamento.
     - O `schedule(runtime)` também pode ser utilizado também.
- Mesmo assim, pode ter a possibilidade do loop 'não ser muito bem paralelizável' ou nenhum escalonador dará a performance requerida.

Produtor e Consumidor

Introdução

- Um olhar sobre paralelismo em atividades que não são fáceis de lidar usando a diretiva `parallel for` ou `for`.

- Fila é uma estrutura de dados natural usado em várias aplicações multithreads.
- Em aplicações de troca de mensagem por memória compartilhada
  - Cada thread deve ter uma fila de memória compartilhada.
  - Quando uma thread envia uma mensagem, 'enfilera' a mensagem na fila do destinatário.

Troca de Mensagem - Pseudocódigo

- Supondo um implementação simples de troca de mensagem no qual
  - Cada thread irá gerar um número inteiro randômico que serão as mensagens;
  - E um número randômico de destinatários.
- Passos:
  1. Depois de criado as mensagens, cada thread checará sua própria lista de mensagens;
  2. Se tiver mensagem, retira da fila e imprime na na tela;
  3. Cada thread alternará entre tentar enviar e receber mensagens;
  4. O usuário definirá o número de mensagens que cada thread deve enviar;
  5. Quando uma thread enviar todas as mensagens, ela ficará recebendo mensagens até que todas acabarem também e serão fechadas.

Troca de Mensagem - Pseudocódigo

- Supondo um implementação simples de troca de mensagem no qual
  - Cada thread irá gerar um número inteiro randômico que serão as mensagens;
  - E um número randômico de destinatários.
- Passos:
  1. Depois de criado as mensagens, cada thread checará sua própria lista de mensagens;
  2. Se tiver mensagem, retira da fila e imprime na na tela;
  3. Cada thread alternará entre tentar enviar e receber mensagens;
  4. O usuário definirá o número de mensagens que cada thread deve enviar;
  5. Quando uma thread enviar todas as mensagens, ela ficará recebendo mensagens até que todas acabarem também e serão fechadas.

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

Troca de Mensagem - Envio de mensagens

- Acessar uma fila de mensagem provavelmente é uma seção crítica.
- Por mais que não olhemos os detalhes da implementação da fila, deve-se tomar medidas de prevenção usando seção crítica.
- Pseudocódigo do `Send_msg()`

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

- É permitido que a thread envie mensagem pra si mesmo.

Troca de Mensagem - Recebimento de mensagens

- A questão de sincronização para recebimento de mensagens é diferente do envio.
- Dono de sua própria fila, irá retirar uma a uma.
- Se existe pelo menos duas mensagens na fila, a `Dequeue` não pode conflitar com `Enqueue`.
  - Mantendo o controle da fila, mantém-se a sincronização.
- Mas, e a variável que armazena o tamanho da lista?
- Tem-se 2 variáveis com o número de mensagens

```
1 queue_size = enqueued - dequeued
```

- `dequeued` é atualizado por seu próprio dono.
- Entrentanto, uma thread pode estar atualizando `enqueued` enquando outra usa o valor de `queue_size`

Troca de Mensagem - Recebimento de mensagens

- Assim, o `Try_receive` será implementado

```
1  queue_size = enqueued - dequeued;
2
3  if (queue_size == 0)
4     return;
5
6  else if (queue_size == 1)
7     #   pragma omp critical
8     Dequeue(queue, &src, &mesg);
9
10 else
11    Dequeue(queue, &src, &mesg);
12
13 Print_message(src, mesg);
```

Troca de Mensagem - Detecção de Término

- Deve-se verificar o fim da execução do processo.
- É fácil percebe que esta implementação possui problemas.
  - Quando uma thread *u* estiver computado que `queue_size == 0`, outra thread *v* provavelmente enviará uma mensagem.
  - Se isso acontecer, a mensagem enviada nunca será lida.
- Para isso, utiliza um `done_sending` que será incrementado a cada finaliação de envio das threads. Permitindo a implementação

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

Troca de Mensagem - Detecção de Término

- Deve-se verificar o fim da execução do processo.
- É fácil percebe que esta implementação possui problemas.
  - Quando uma thread *u* estiver computado que `queue_size == 0`, outra thread *v* provavelmente enviará uma mensagem.
  - Se isso acontecer, a mensagem enviada nunca será lida.
- Para isso, utiliza um `done_sending` que será incrementado a cada finaliação de envio das threads. Permitindo a implementação

```
queue_size = enqueued - dequeued;

if (queue_size == 0)
   return TRUE;
else return FALSE;
```

```
queue_size = enqueued - dequeued;

if (queue_size == 0 &&
            done_sending == thread_count)
   return TRUE;
else return FALSE;
```

Troca de Mensagem - Iniciando

- Quando o programa inicia, é recebido por parâmetro a quantidade de threads.
- Também é alocado a fila de mensagens, uma para cada thread.
  - Este arranjo precisa ser compartilhado entre as threads.
- Existirá os seguintes armazenamentos;
  - Lista de mensagem;
  - Um ponteiro para o índice da cauda da fila;
  - Um ponteiro para o índice da cabeça da fila;
  - Um contaor de mensagens enfileiradas; e
  - Um contador de mensagens desenfileiradas.
- Após a alocação, pode-se iniciar as threadas usando a diretiva `parallel` e cada uma alocando sua fila individual.

Troca de Mensagem - Iniciando

- É provável que aconteça que algumas threads acabará de alocar antes das outras e começará a enviar as mensagens mesmo que outras threads não tenham terminado a alocação.
  - *Segmentation Fault*.
- Deve-se fazer com que nenhuma thread começe antes que todas tenham alocado. E com isso utiliza-se barreiras.
  - Enquanto todas as threads do time não completarem a ação, as outras não podem continuar.
  - Esta barreira é explícita sendo sua diretiva

```
# pragma omp barrier
```

Troca de Mensagem - Diretiva atomic

- Após completado as diretivas de envio, cada thread incrementa `done_sending` no final do loop.
- `done_sending` é uma seção crítica e poderíamos protegê-la com a diretiva `critical`.
- Mas OpenMP provê uma diretiva potencialmente de alta performance. A diretiva atômica.

```
# pragma omp atomic
```

Troca de Mensagem - Diretiva atomic

- Ao contrário da `critical`, ela só protege seções críticas que consistem numa única atribuição. Tal como:

  `x <op>= <expression>;`

  `x++;`

  `++x;`

  `x--;`

  `--x;`
- Sendo `<op>` um operador binário: `+`, `*`, `-`, `/`, `&`, `^`, `|`, `<<`, ou `>>` e `<expression>` não pode referenciar `x`.
- A expressão não está inclusa na seção crítica. Somente a atribuição do resultado dela à variável que está.

```
1 #  pragma omp atomic
2   x += y++;
```

Troca de Mensagem - Seção Crítica e locks

- Algumas precauções sobre a diretiva `critical`.
- Em todo os exemplos, existe mais de uma seção crítica.
- Entretanto, o uso de seção crítica na passagem de mensagens é mais complexa.
- Se examinarmos existe 3 blocos que utilizam diretivas `critical` e `atomic`
  - `done_sending++;`, `Enqueue(q_p, my_rank, mesg);` e `Dequeue(q_p, &src, &mesg);`.
- Entretanto, não é necessário que as três tenha completo acesso exclusivo. Inclusive as duas últimas.
  - Seria viável a threado 0 enfileirar uma mensagem na thread 1 enquanto esta enfilera uma mensagem em thread 2.
- OpenMP provês duas seções críticas distintas. O `atomic` e o 'composto' de seções críticas que enfileiramos e retiramos mensagens.
- Desde que aplique exclusão mútua entre as threads de execução seriável, esse é o comportamento principal do OpenMP.
  - Todas os blocos de seções críticas são parte de um composto de seção crítica.

Troca de Mensagem - Seção Crítica e locks

- É possível dar um nome a diretiva crítica: `# pragma omp critical(name)`.
- Fazendo isso, dois blocos protegidos com `critical` com diferentes nomes podem executar simultaneamente.
  - Os nomes são definitos em tempo de compilação;
- Entretanto, perecisamos definir algo que seja em tempo de execução. E por esse motivo, nomear `critical` não é suficente.
- Uma alternativa é utilizar locks.

Troca de Mensagem - Lock

- Lock consistem de estruturas de dados e funções que permitem o programador explicitar a exclusão mútua numa seção crítica.
- O uso do lock pode ser descrito pelo seguinte pseudocódigo:

```
1 /* Executed by one thread */
2 Initialize the lock data structure;
3 ...
4 /* Executed by multiple threads */
5 Attempt to lock or set the lock data structure; Critical section;
6 Unlock or unset the lock data structure;
7 ...
8 /* Executed by one thread */
9 Destroy the lock data structure;
```

- Inicialização
  - A estrutura de dados é compartilhada com as threads que irão executar a seção crítica.
  - A thread master irá inicializar o lock. Quanto todas estiver utilizando, uma deverá qubrá-lo.
- Execução
  - Depois que uma entrar na secão crítica, comeca a tentativa de *definir* a estrutura de dados

Cláusula `schedule` - Lock

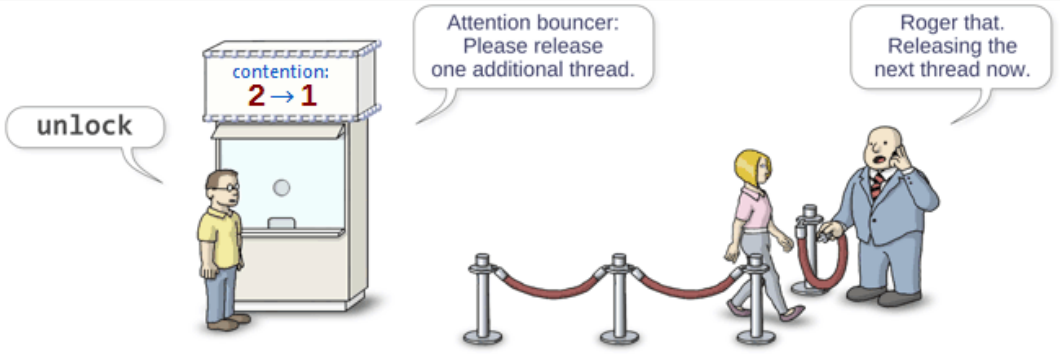

Figura 6: Exemplo de lock.

Cláusula `schedule` - Lock

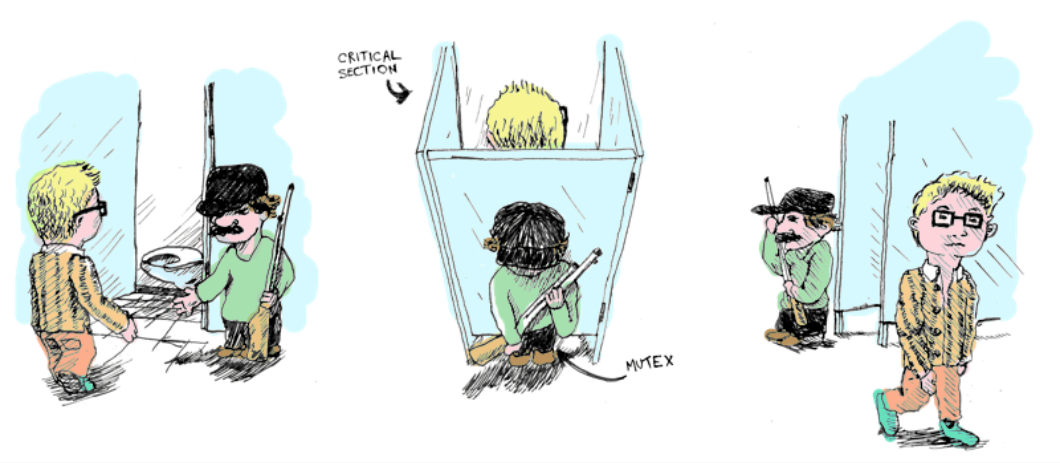

Figura 7: Outro exemplo de lock.

Troca de Mensagem - Lock

- OpenMP possui dois tipos de lock
  - Simple: Pode ser ‘setado’ somente depois que for ‘des-setado’.
  - Nested: Pode ser definido múltilas vezes pela mesma thread antes do ‘des-set’.
- Um tipo de OpenMP simple lock é o `omp_lock_t`, e e possui as funções de definição:

```
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 */);
```

- Tudo isso é especificado em `omp.h`.

Troca de Mensagem - Lock

- Queremos assegurar a exclusão mútua em cada fila de mensagem e não em um bloco de código particular.
- Pode-se trocar os códigos

```
1 #   pragma omp critical
2   /*q_p = msg_queues[dest]*/
3   Enqueue(q_p, my_rank, mesg);
```

```
1 #   pragma omp critical
2   /*q_p = msg_queues[my_rank]*/
3   Dequeue(q_p, &src, &mesg);
```

- Por

```
1   /*q_p = msg_queues[dest]*/
2   omp_set_lock(&q_p->lock);
3   Enqueue(q_p, my_rank, mesg);
4   omp_unset_lock(&q_p->lock);
```

- Por

```
1   /*q_p = msg_queues[my_rank]*/
2   omp_set_lock(&q_p->lock);
3   Dequeue(q_p, &src, &mesg);
4   omp_unset_lock(&q_p->lock);
```

Diretiva `critical`, Diretiva `atomic` ou lock

- A diretiva `atomic` é potencialmente mais rápida para obter exclusão mútua.
- Se a seção crítica consiste numa atribuição que querer uma forma, então utiliza-se tão bem o atomic quanto as outras.
- Impões exclusão mútua a todas as diretivas `atomic`. Essa é a forma que a `critical` sem nomeação comporta.
- Se isso pode ser um problema, por exemplo, ter várias seções críticas diferentes, então deve-se utilizar `critical` nomeado ou locks.

Diretiva `critical`, Diretiva `atomic` ou lock

- Supondo que cada um dos códigos abaixo é executado em duas threads diferenes

```
1 #   pragma omp critical
2     x++;
```

```
1 #   pragma omp critical
2     y++;
```

- Mesmo se tiverem em locais de memória diferente, se uma thread executa o código a esquerda então nenhum outro executa a direita.
- Entretanto, ambas as `critical` (nomeadas e não nomeadas) são fácels de trabalhar.
- Não existe uma diferença significativa entre a `critical` e locks.
  - Se não pode utilizar uma diretiva atômica, qualquer uma das duas satisfaz.
- Locks deve ser usado quando é necessária a exclusão mútua para uma **estrutura de dados** em vez de um **bloco de código**.

Ressalvas

- Não deve juntar diferentes tipos de de exlusão para uma única seção.

```
1 #   pragma omp atomic
2     x += f(y);
```

```
1 #   pragma omp critical
2     x = g(x);
```

  - A atualização de `x` à direita não tem uma forma definida pela diretiva `atomic` e por isso utilizou o `critical`.
  - O `critical` não excluirá a ação do `atomic` gerando possível resultados incorretos.
  - Deve-se reescrever a função `g()` pra que seja da forma da diretiva `atomic`, ou proteger ambos os blocos com `critical`.

Ressalvas

- Não existe garantia de justiça na construção de exclusão mútua.

```
1    while (1) {
2        ...
3  #      pragma omp critical
4        x = g(my_rank);
5        ...
6    }
```

  - Uma thread pode ser bloqueada pra sempre esperando a seção crítica.

Ressalvas

- Pode ser perigoso aninhar exclusões mútuas
  - Isso é uma garantia para **deadlock**.
  - Quando uma thread tenta entrar na segunda seção crítica, ela fica bloqueada pra sempre.
  - Se uma thread *u* está executando dentro da primeira seção crítica, nenhuma thread pode estar dentro da segunda.
  - A thread *u* espera a segunda seção crítica e a thread *v*, dentro da segunda, espera a primeira.

```
1 #   pragma omp critical
2     y = f(x);
3     ...
4     double f(double x) {
5 #       pragma omp critical
6         z = g(x); // z is shared
7     }
```

Ressalvas

- Pode ser perigoso aninhar exclusões mútuas
  - Isso é uma garantia para **deadlock**.
  - Quando uma thread tenta entrar na segunda seção crítica, ela fica bloqueada pra sempre.
  - Se uma thread *u* está executando dentro da primeira seção crítica, nenhuma thread pode estar dentro da segunda.
  - A thread *u* espera a segunda seção crítica e a thread *v*, dentro da segunda, espera a primeira.

```
1 #   pragma omp critical
2     y = f(x);
3     ...
4     double f(double x) {
5 #       pragma omp critical
6         z = g(x); // z is shared
7     }
```

```
1 #   pragma omp critical(one)
2     y = f(x);
3     ...
4     double f(double x) {
5 #       pragma omp critical(two)
6         z = g(x); // z é global
7     }
```

Ressalvas

```
1 #   pragma omp critical
2     y = f(x);
3     ...
4     double f(double x) {
5 #       pragma omp critical
6         z = g(x); // z is shared
7     }
```

```
1 #   pragma omp critical(one)
2     y = f(x);
3     ...
4     double f(double x) {
5 #       pragma omp critical(two)
6         z = g(x); // z é global
7     }
```

Time	Thread u	Thread v
0	Enter crit. sect. one	Enter crit. sect. two
1	Attempt to enter two	Attempt to enter one
2	Block	Block

Figura 8: Deadlock.

Thread-Safefy

Introdução

- Outro problema que ocorre com o compartilhamento de memória é o thead-safety.
- Um bloco de código é thread-safety se pode ser executado simultaneamente por várias threads sem causar problemas.
- Um exemplo é múltiplas threads “tokenizear” um arquivo.
- Uma abordagem seria dividir o arquivo em linhas de texto e colocar cada thread para executar com escalonamento round-robin.
- Existindo n linhas
  - Thread 0 fica com n;
  - Thread 1 fica com $n + 1$;
  - Thread t fica com $n + t$;
  - Thread 0 fica com $n + t + 1$ e segue;

Introdução

- Existe a função `strtok` no `string.h`.
- Ela retorna tokens de acordo com separador definido. Segue o seguinte protótipo

```
1 char * strtok(char* string /* in/out */,
2           const char * separators /* in */);
```

- Seu uso é um pouco incomum
  - Devemos passar a cadeia `\t\n` como o argumento separadores dos tokens.
  - A primeira vez que ele é chamado, o argumento `string` deve ser o texto a ser indexado.
  - Para chamadas subsequêntes, `string` deverá ser NULL.
  - Assim, na primeira chamada, `strtok()` armazena o ponteiro e para chamadas seguintes ele retorna tokens de sucessivas tomadas a partir da cópia em cache.

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

Introdução

- Executando a primeira vez o código com duas threads e com 4 entradas: “Pease porridge hot.” “Pease porridge cold.” “Pease porridge in the pot” “Nine days old.”.
  - Obtêm o resultado corretamente.
- Ao executar pela segunda vez obtêm o seguinte resultado

```
Thread 0 > line 0 = Pease porridge hot.
Thread 1 > line 1 = Pease porridge cold.
Thread 0 > token 0 = Pease
Thread 1 > token 0 = Pease
Thread 0 > token 1 = porridge
Thread 1 > token 1 = cold.
Thread 0 > line 2 = Pease porridge in the pot
Thread 1 > line 3 = Nine days old.
Thread 0 > token 0 = Pease
Thread 1 > token 0 = Nine
Thread 0 > token 1 = days
Thread 1 > token 1 = old.
```

- O que aconteceu?
  - Apesar `lines` ser argumento só de entrada, `strtok` modifica o vetor.
  - Quando `Tokenize` retornar o vetor `lines` estará modificado pois `strtok` armazena em cache a linha de entrada.
  - `strtok` armazena os valores numa variável estática persistindo-a.
  - Essa persistência é 'cache compartilhado'.
  - A thraead 0 encontrou um token (`dias`) que deveria estar na saída da thraead 1.
- Portanto, `strtok` não é thread-safety.
- Isso não é incomum nas funções das bibliotecas em C.
  - Gerador de número randômico do `stdlib.c` e a função de conversão de tempo local time.c também falham.

- Em alguns casos o padrão C especifica suplentes.

```
1 char * strtok_r(char* string /* in/out */,
2           const char* separators /* in */,
3           char** saveptr_p /* in/out */);
```

- `_r` significa reentrante. Também usado como sinônimo de thread-safe.
- Mantém o controle de onde a função resolvendo o problema do cache.
- trocar as chamadas de funções resolve o problema da nossa função.
  - Devemos passar a cadeia `\t\n` como o argumento separadores dos tokens.
  - A primeira vez que ele é chamado, o argumento `string` deve ser o texto a ser indexado.
  - Para chamadas subsequêntes, `string` deverá ser NULL.
  - Assim, na primeira chamada, `strtok()` armazena o ponteiro e para chamadas seguintes ele retorna tokens de sucessivas tomadas a partir da cópia em cache.

Programas incorretos podem produzir saídas corretas

- Nosso primeiro código do tokenizer exibe armadilhas
  - A primeira execução é correta.
  - Teve que executar novamente pra perceber o erro.
- Isso não é raro em programas paralelos. Principalmente programas com memória compartilhada.

Arquiteturas Paralelas e OpenMP

Natanael Ramos
Rodolfo Labiapari Mansur Guimarães

14 de dezembro de 2015