Notas dos slides

APRESENTAÇÃO

O presente conjunto de slides pertence à coleção produzida para a disciplina *Introdução ao Processamento Paralelo e Distribuído* ofertada aos cursos de bacharelado em Ciência da Computação e em Engenharia da Computação pelo Centro de Desenvolvimento Tecnológico da Universidade Federal de Pelotas.

Os slides disponibilizados complementam as videoaulas produzidas e tratam de pontos específicos da disciplina. Embora tenham sido produzidos para ser assistidos de forma independente, a sequência informada reflete o encadeamento dos assuntos no desenvolvimento do conteúdo programático previsto para a disciplina.

computação
Universidade Federal de Pelotas

universidade federal de pelotas
CDTec
centro de desenvolvimento tecnológico

Programação Multithread

Programação com OpenMP

Notas da videoaula

DESCRIÇÃO

Esta videoaula sumariza os elementos principais de programação com Pthreads.

OBJETIVOS

Apresentar os conceitos fundamentais de programação com OpenMP. Capacitar o aluno a desenvolver programas simples e lhe dar autonomia para identificar e explorar recursos de programação para desenvolvimento de programas mais complexos.

All together now.
All together now.
All together now.
All together now.
All together now.
All together now.
Bom bom bom bompa bom.

The Beatles

OpenMP

Definido a partir de um esforço colaborativo entre diversas empresas de TI (como Digital, IBM, Intel e SGI).

Realidade: projetos de arquiteturas multiprocessadas seriam commodities em pouco tempo e não existia uma ferramenta portável para explorar este tipo de hardware.

Meta: Capacitar programadores à explorá-las de forma eficiente.

Architecture Review Board (ARB)

1997

The Persistence of Memory
Salvador Dalí

OpenMP

API definida para ser incorporada a linguagem Fortran e C. Assim, toda semântica operacional da linguagem nativa é aceita, facilitando a manipulação de dados no programa.

Realidade da época:

- Grande demanda computacional em aplicações tipicamente desenvolvidas em Fortran e C;
- O foco do consumo estava em aplicações científicas (sistemas meteorológicos, modelos de simulação, ...);
- Embora mais populares, arquiteturas multiprocessadas ainda eram restritas a nichos.

OpenMP: Evolução

Da sua primeira especificação, a OpenMP evoluiu, buscando melhor atender a demanda por programação multithread em ambientes com memória compartilhada:

- Até versão 2.5: Ajustes na especificação da API; Suporte a Fortran e C/C++; Paralelismo de dados e laços paralelos;
- 3.0 (2008): Paralelismo aninhado (tarefas explícitas);
- 4.0 (2013): Suporte a aceleradores; Suporte às instruções atômicas e SIMD; grupo de tarefas;
- 5.0 (2018): Melhorias nas operações de grupos de tarefas; tratamentos com laços controlados por iteradores (C++).

OpenMP vs. Pthreads

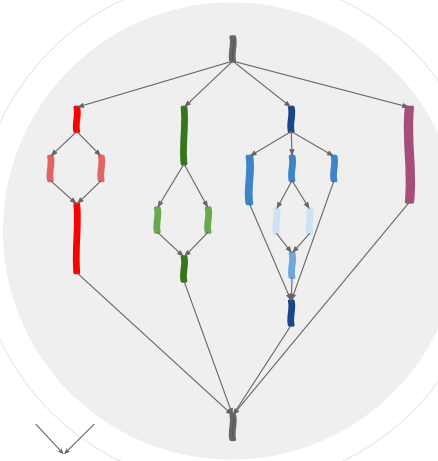
OpenMP foi concebido para explorar de forma eficiente hardware paralelo. Para tanto, impõe (restringe) um modelo de programação para descrição da concorrência na aplicação em prol de uma heurística de escalonamento eficiente. O programador, não tem, em consequência, toda a liberdade para descrever seu algoritmo concorrente, por outro lado, ao adaptá-lo à estratégia de escalonamento, não necessita controlar a distribuição da carga de processamento gerada na arquitetura disponível.

OpenMP

- Implementa modelo de threads N *x* M;
  - A unidade de trabalho: **tarefa**;
  - A unidade de escalonamento: **thread**;
    - O conjunto de threads é denominado pool de execução;
- Impõe o modelo de execução fork/join aninhado;
- As tarefas são anônimas;
- Conceito de Região Paralela e diferentes semânticas de criação de tarefas;
- Dois mecanismos básicos de escalonamento:
  - Worksharing
  - Recursivo

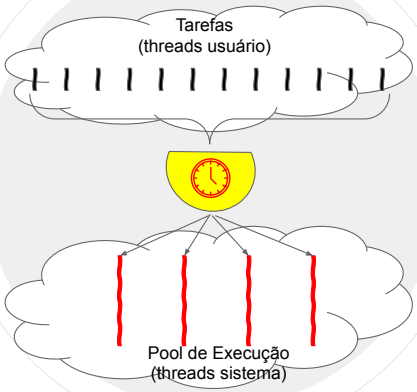
Modelo Básico

- A execução do programa segue o modelo fork/join aninhado:
  - Tarefas podem criar novas tarefas em uma operação síncrona (**fork**)
  - A tarefa criadora somente está habilitada a continuar sua execução quando todas as tarefas criadas terminarem (**join**)
- Conceito: **Região Paralela**
  - Região produtora de tarefas

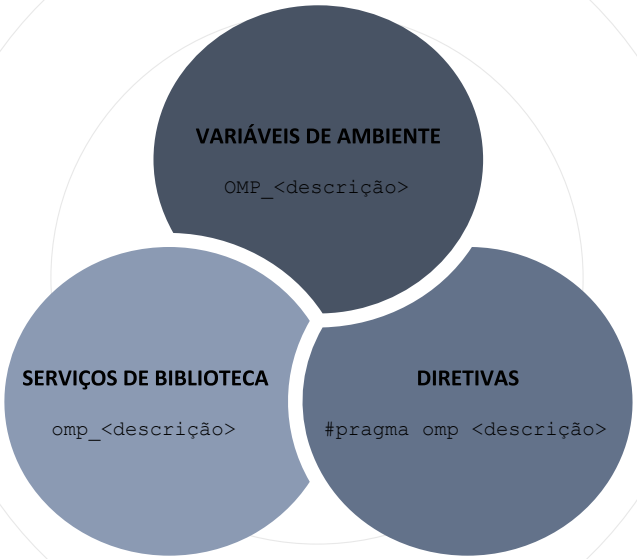


Modelo Básico

- O programador descreve a concorrência de seu programa em termos de **tarefas**
  - Threads usuário
- O runtime fornece um mecanismo de **escalonamento** e um **pool de execução**
  - O pool de execução é composto de um conjunto de **threads** sistema
  - O escalonador promove a alocação das tarefas sobre os threads
- O Sistema Operacional é responsável pelo escalonamento dos threads do pool de execução.



API



API: Variáveis de Ambiente

Existem 4 variáveis de ambientes definidas no padrão, que definem alguns comportamentos default para todas aplicações OpenMP:

- `OMP SCHEDULE`: Identifica o mecanismo de escalonamento default para o worksharing (static, dynamic, guided);
- `OMP NUM THREADS`: Define o número de unidades de escalonamento em nível de sistema operacional;
- `OMP DYNAMIC`: Habilita, ou não, o ajuste em tempo de execução do número de threads (TRUE,FALSE);
- `OMP NESTED`: Habilita, ou não, a criação aninha de pools de execução (TRUE,FALSE).

Outras variáveis de ambiente podem ser incluídas pelos fornecedores.

API: Serviços de Biblioteca

Promovem a interação entre o programa aplicativo e o *runtime*:

- `int omp get_num_threads()`: retorna o número de threads no pool de execução;
- `void omp set num_threads(int n)`: define o número de threads no pool de execução;
- `int omp get thread num()`: retorna o número do threads corrente;
- `int omp get num_procs()`: retorna o número de *cores* disponíveis para execução;
- `int omp in parallel()`: retorna o número de *cores* disponíveis para execução.não zero caso esteja em uma região paralela;
- Primitivas para manipulação de mutex.

Este são apenas exemplo.

API: Diretivas

As diretivas permitem, efetivamente, realizar a programação em OpenMP, ou seja, descrever a concorrência e realizar sincronizações. O formato de uma diretiva, em C, é:

```
#pragma omp <diretiva> [<Cláusulas>]
```

- **Lista de Cláusulas**: Parâmetros às diretivas.
- **Diretiva OpenMP**: Especifica o comportamento desejado na manipulação da concorrência do programa.
- **Sentinela OpenMP**: informa ao pré-processador a existência de uma diretiva OpenMP na linha

DIRETIVAS

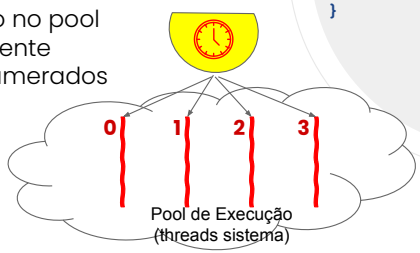
```
#pragma omp <descrição>
```


Região Paralela

Define uma zona produtora de tarefas. Instância um pool de execução responsável pelo escalonamento e execução das tarefas criadas.

O número de threads a ser criado no pool de execução depende é previamente configurado, sendo os threads numerados a partir de 0 (zero).

O thread 0 é chamado **master**.




```
#pragma omp parallel
{
 ...
}
```


Região Paralela

Existem diversos mecanismos para criação de tarefas. O modelo básico é o **fork**/**join** síncrono: uma tarefa cria outras tarefas, suspendendo sua execução, estando desbloqueada quando todos as tarefas criadas tiverem concluído. O thread sobre o qual a tarefa instanciou as novas tarefas é o thread master para o pool de execução e, estando suspensa a execução da tarefa corrente, este thread master participa da computação das tarefas criadas.

Existem diferentes semânticas de criação de tarefas, determinadas por diferentes recursos oferecidos pela API.

```
#pragma omp parallel
{
 ...
}
```


Definição do Corpo de Tarefas

Corpo de Tarefas: 1

Uma sequência de instruções da linguagem base entre duas diretivas OpenMP representa o corpo de uma tarefa. Esta tarefa é instanciada tantas vezes quanto forem o número de threads no pool de execução.

No exemplo, supondo a existência de quatro threads no pool de execução, serão criadas quatro tarefas idênticas, cada tarefa executará sobre um dos threads e imprimirá o número do thread correspondente (0, 1, 2 e 3, no caso).

```
#pragma omp parallel
{
 printf("TID = %d\n",
        omp_thread_num());
}
```


Corpo de Tarefas: 1

Neste outro exemplo, ainda supondo a existência de quatro threads no pool de execução, também serão criadas quatro tarefas idênticas, cada uma executando sobre um dos threads do pool de execução. No entanto, cada tarefa imprimirá um **Oi!** E executará um loop de 10 iterações imprimindo o número do thread correspondente (Oi! 0 0 0 0..., Oi! 1 1 1 1..., Oi! 2 2 2 2... e Oi! 3 3 3 3...).

Atenção: as impressões, na tela, irão aparecer entrelaçadas, pois as tarefas executam concorrentemente entre si.

```
#pragma omp parallel private(i)
{
 printf("Oi!\n");
 for(i=0;i<10;++i)
  printf("TID = %d\n",
         omp_thread_num());
}
```


Corpo de Tarefas: 2

As diretivas `master` e `single` permitem a criação de uma única tarefa responsável pela execução do código indicado. Este código pode ser tanto um comando composto (conjunto de instruções delimitados por `{` e `}`, em C) ou uma instrução única.

```
#pragma omp parallel
{
 #pragma omp master
 {
   ...
 }
 #pragma omp single
 {
   ...
 }
}
```


Corpo de Tarefas: 3

A diretiva `sections` encapsula descritores de tarefas identificados pela diretiva `section`. Cada `section` define o corpo para uma tarefa, instanciada uma única vez e executada por um dos threads do time. Não há, portanto, relação direta entre o número de tarefas criadas e o número de tarefas no pool de execução.

```
#pragma omp parallel
{
 #pragma omp sections
 {
  #pragma omp section
   { ... }
  #pragma omp section
   { ... }
 }
}
```

Corpo de Tarefas: 3

Neste exemplo, serão criadas três tarefas, independente do número de threads no pool de execução.

```
#pragma omp parallel
{
 #pragma omp sections
 {
  #pragma omp section
   printf("A");
  #pragma omp section
   printf("B");
  #pragma omp section
   printf("C");
 }
}
```


Corpo de Tarefas: 4

É possível instanciar um laço paralelo. Neste caso, o conjunto de iterações é particionado em tarefas, cada tarefa executando um subconjunto do total de iterações. Importante notar que o conjunto de iterações será executado uma única vez.

```
#pragma omp parallel
{
 #pragma omp for
  for(i=BEGIN;i<END;++i) {
   ...
  }
}
```


Corpo de Tarefas: 4

Neste exemplo, a saída do programa será a impressão dos números entre 0 e 9, uma única vez, independente do número de threads no pool de execução.

```
#pragma omp parallel
{
 #pragma omp for
  for(i=0;i<10;++i) {
   printf("%d\n",i);
  }
}
```


Corpo de Tarefas: 4

Neste outro exemplo é comparado um laço não paralelo e um laço paralelo.

Supondo que existam quatro threads no pool de execução, serão criadas quatro tarefas para executar o conjunto de instruções entre duas diretivas OpenMP (cada uma destas tarefas imprimirá dez vezes o número do thread sobre o qual está executando. As tarefas criadas no laço paralelo serão distribuídas entre os threads do pool de execução e uma única sequência de números entre 0 e 9 será impressa.

```
#pragma omp parallel private(i)
{
 for(i=0;i<10;++i)
  printf("%d\n",
         omp_thread_num());
 #pragma omp for
  for(i=0;i<10;++i) {
   printf("%d\n",i);
  }
}
```

Corpo de Tarefas: 4

A variável de iteração, o valor inicial e o valor final precisam ser do tipo inteiro com sinal. A variável de iteração é, por default, **`private`**.

A variável de iteração pode ser, apenas, incrementada ou decrementada (pré ou pós fixado). Não é permitida nenhuma outra operação.

Chunk número de iterações atribuído a cada tarefa. O comprimento do *chunk* e a estratégia de distribuição das tarefas sobre os threads do pool de execução podem ser definida pela cláusulas **`schedule`** no laço paralelo.

```
#pragma omp for schedule(...)
 for(i=BEGIN;i<END;++i){
   ...
 }
```


Corpo de Tarefas: 4

Por default, quer dizer, se a cláusula **`schedule(...)`** não for usada, **chunk=1** e as tarefas são alocadas de forma **estática** aos threads. Supondo o laço do exemplo ao lado, a distribuição das tarefas seria a seguinte:

Threads							
0		1		2		3	
Tarefa	Iterações	Tarefa	Iterações	Tarefa	Iterações	Tarefa	Iterações
0	0	1	1	2	2	3	3
4	4	5	5	6	6	7	7
8	8	9	9	10	10	11	11
96	96	97	97	98	98	99	99

```
omp_set_num_threads(4);
#pragma omp parallel
 #pragma omp for
  for(i=0;i<100;++i) {
    ...
  }
```

100 tarefas criadas.

Corpo de Tarefas: 4

Neste segundo exemplo, **chunk=4**, com alocação ainda **estática**. A distribuição das tarefas agora é:

Threads							
0		1		2		3	
Tarefa	Iterações	Tarefa	Iterações	Tarefa	Iterações	Tarefa	Iterações
0	0-3	1	4-7	2	8-11	3	11-12
4	16-19	5	20-24	6	24-27	7	28-31
8	32-35	9	36-39	10	40-42	11	33-47
21	84-87	22	88-91	23	92-95	24	96-99

25 tarefas criadas.

```
omp_set_num_threads(4);
#pragma omp parallel
 #pragma omp for schedule(static,4)
  for(i=0;i<100;++i) {
    ...
  }
```


Algoritmos Recursivos

`task` / `taskwait`

Corpo de Tarefas: 5

Preâmbulo: até o momento, a criação de tarefas é realizada de forma implícita, associada a semântica das diretivas OpenMP. Eventualmente é o próprio runtime quem faz a decomposição das tarefas. Também de forma implícita ocorre a sincronização ao término da execução das tarefas. A diretiva `task`, por outro lado, é um recurso de programação que explicita a criação de tarefas, necessitando que seja realizada também de forma explícita, a sincronização entre elas por meio da diretiva `taskwait`.

O uso de `task`/`taskwait` permite implementar algoritmos recursivos.

```
#pragma omp task
{
 ...
}
#pragma omp taskwait
```


Corpo de Tarefas: 5

No exemplo, uma tarefa em execução cria uma nova tarefa para executar a função `foo`. A nova tarefa é criada e a tarefa original segue sua execução, no caso, executando a função `bar`. A primitiva é invocada quando a tarefa original necessitar que todas as tarefas que ela própria tenha criado tenham terminado.

```
{
 ...
 #pragma omp task
  foo();
 bar();
 #pragma omp taskwait
 ...
}
```


Corpo de Tarefas: 5

Atenção: observe o código e assumam que existem 4 threads no pool de execução. Neste código, a exemplo do apresentado em **Corpo de Tarefa: 1**, serão criadas 4 instâncias de tarefas para executar `printf("A")` e outras 4 para `printf("C")`. Da mesma forma, serão instanciadas tantas tarefas com a diretiva `task` quanto forem o número de threads do pool de execução.

Neste exemplo são impressos 4 sequências A B C. Uma sequência por thread no pool de execução.

```
#pragma omp parallel
{
 printf("A");
 #pragma omp task
  printf("B");
 #pragma omp taskwait
  printf("C");
}
```


Corpo de Tarefas: 5

Caso seja necessário especificar que a diretiva `task` deve criar uma única instância da tarefa, então ela deve ser incluída em um bloco `single` (ou `master`).

Neste novo exemplo, serão impressos 4 As e Cs, mas apenas um B.

```
#pragma omp parallel
{
 printf("A");
 #pragma omp single
  #pragma omp task
   printf("B");
 #pragma omp taskwait
  printf("C");
}
```

Corpo de Tarefas: 5

Exemplo prático: Cálculo da *n*-ésima posição de Fatorial, de forma recursiva.

A função `fibo`, cria, recursivamente, tarefas para cálculo das dependência do cálculo.

```
int fibo( int n ) {
 int r1, r2;
 if( n < 2 ) return n;
 #pragma omp task
  r1 = fibo(n-1);
 #pragma omp task
  r2 = fibo(n-2);
 #pragma omp taskwait
 return r1+r2;
}
```


Corpo de Tarefas: 5

Exemplo prático: Cálculo da *n*-ésima posição de Fatorial, de forma recursiva.

A função `main`, instancia um pool de execução e, sobre ele, instancia uma única tarefa para ser a raíz para o cálculo recursivo da *n*-ésima posição de Fibonacci.

Seria um erro retirar o `single`, pois seriam instanciadas tantas raízes para o cálculo de Fibonacci quanto fossem o número de threads no pool de execução.

```
int main() {
 int r;
 #pragma omp parallel
 {
  #pragma omp single
    r  = fibo(10);
 }
 printf("Res = %d\n",r);
}
```


Compartilhamento de Dados

Modelo de Memória

Sendo implementada como uma ferramenta de programação multithread, todo o espaço de endereçamento do processo é virtualmente acessível a todas tarefas. O não existe nenhuma sincronização implícita aos dados.

Estratégia: Delimitação do escopo de visibilidade.

Mecanismo: Parametrização das tarefas na criação.

Parametros às tarefas

O escopo que envolve uma tarefa é recebido, implicitamente, na sua criação. Assim, todos identificadores do escopo envolvente são reconhecidos. Parametrizar a criação de tarefas permite restringir os modos de acesso à memória a qual cada um destes identificadores se refere.

Cláusulas: identificam os modos de acesso permitidos aos dados.

Parametros às tarefas

O escopo que envolve uma tarefa é recebido, implicitamente, na sua criação. Assim, todos identificadores do escopo envolvente são reconhecidos. Parametrizar a criação de tarefas permite restringir os modos de acesso à memória a qual cada um destes identificadores se refere.

Cláusulas: identificam os modos de acesso permitidos aos dados.

```
#pragma omp <diretiva> <Cláusulas>
{
 ...
}
```


Parâmetros às tarefas

Cláusulas: são parâmetros das diretiva, recebendo uma lista de identificadores acessíveis pelas tarefas durante a execução e aplicando a eles a semântica apropriada.

- `shared`
- `private`
- `firstprivate`
- `lastprivate`
- `Reduction`

Esta lista não é exaustiva, mas completa o suficiente.

A cláusulas determinam a semântica de acesso aos dados compartilhados. Em tempo de compilação, a referência dos identificadores pode ser alterada em função da semântica adotada.

Parâmetros às tarefas

Cláusulas:

- `shared`

  Semântica CRCW, sem garantia de coerência no acesso.

Modo de acesso default na maior parte dos casos. Localmente a cada tarefa, o identificador shared referencia ao dado compartilhado no escopo envolvente. Nenhum mecanismo de controle de acesso implícito existe.

```
int cont = 10;
#pragma omp parallel shared(cont)
{
 ++cont;
}
```

Condição de Corrida detectada!
Não é possível determinar o valor final de `cont`.

Parâmetros às tarefas

Cláusulas:

- `shared`

  Nos leva a introduzir uma nova diretiva: **`critical`**

  Esta diretiva opera de forma similar a um mutex, permitindo a execução de uma instrução (ou de um comando composto) em regime de exclusão mútua.

```
int cont = 10;
#pragma omp parallel shared(cont)
{
 #pragma omp critical
 ++cont;
}
```

Caso o número de threads seja 4, o resultado final armazenado em `cont` é 14!

Parâmetros às tarefas

Cláusulas:

- `shared`

É possível, ainda, associar um rótulo à diretiva **`critical`**, fazendo com que o regime de exclusão mútua seja garantido apenas entre aqueles que utilizarem o mesmo rótulo.

```
int cont = 10;
#pragma omp parallel shared(cont)
{
 #pragma omp critical contador
 ++cont;
}
```


Parâmetros às tarefas

Cláusulas:

- `private`

  Semântica CRCW, com garantia de coerência no acesso, preservando o valor original do dado no escopo envolvente.

Uma nova instância do dado é alocada localmente a cada tarefa, passando o identificador a referenciá-lo. O valor inicial da variável local é indeterminado e o dado original no escopo envolvente permanece intocado durante a execução das tarefas.

```
int x=1;
#pragma omp parallel private(x)
{
 x = foo();
}
```


Parâmetros às tarefas

Cláusulas:

- `firstprivate`

  Semântica CRCW, com garantia de coerência no acesso, preservando o valor original do dado no escopo envolvente.

Uma nova instância do dado é alocada localmente a cada tarefa, passando o identificador a referenciá-lo. O valor inicial da variável local é uma cópia do dado original no escopo envolvente. Durante a execução das tarefas, o dado no escopo envolvente permanece intocado.

```
int x=1;
#pragma omp parallel firstprivate(x)
{
 x = foo(x);
}
```

Parâmetros às tarefas

Cláusulas:

- `lastprivate`

  Semântica CRCW, com garantia de coerência no acesso, alterando o valor original do dado no escopo envolvente ao final da execução de todo conjunto de tarefas.

Uma nova instância do dado é alocada localmente a cada tarefa, passando o identificador a referenciá-lo. O valor inicial da variável local é indeterminado. Ao final da execução de todas as tarefas, o dado armazenado na última tarefa a terminar é copiado para a variável no escopo envolvente.

```
int x;
#pragma omp parallel lastprivate(x)
{
 x = foo(x);
}
```


Parâmetros às tarefas

Cláusulas:

- `firstprivate e lastprivate`

Nada impede de um identificador ser declarado como first e lastprivate. Neste caso, todas as tarefas terão uma instância local do identificador inicializada com o valor anotado no escopo envolvente e, ao final da execução de todas as tarefas, o valor anotado no escopo da última tarefa a executar é copiado para o dado original.

```
{
 int x = 1;
 #pragma omp parallel lastprivate(x)\
                      firstprivate(x)
 {
  x = foo(x);
 }
```


Parâmetros às tarefas

Cláusulas:

- `reduction`

  Semântica CRCW, com garantia de coerência no acesso, alterando o valor original do dado no escopo envolvente ao final da execução de todo conjunto de tarefas.

A cláusula requer, além do identificador, a **operação de redução** a ser realizada. Todos as tarefas possuem uma instância local do dado e operam sobre ela. Ao final, o dado original é recebe a aplicação da operação indicada sobre todos os resultados parciais computados individualmente pelas tarefas.

```
{
int cont = 10;
#pragma omp parallel reduction(+:cont
{
 ++cont;
}
```

Caso o número de threads seja 4, o resultado final armazenado em `cont` é 14!

Parâmetros às tarefas

Cláusulas:

- `reduction`

A instância local da variável de redução é inicializada com o valor identidade da operação selecionada. O conjunto de operações válidas é limitado.

Operação	Operador	Identidade
Adição	+	0
Subtração	-	0
Multiplicação	*	1
E Aritmético	&	~0
OU Aritmético	|	0
XOR Aritmético	^	0
E Lógico	&&	1
OU Lógico	||	0

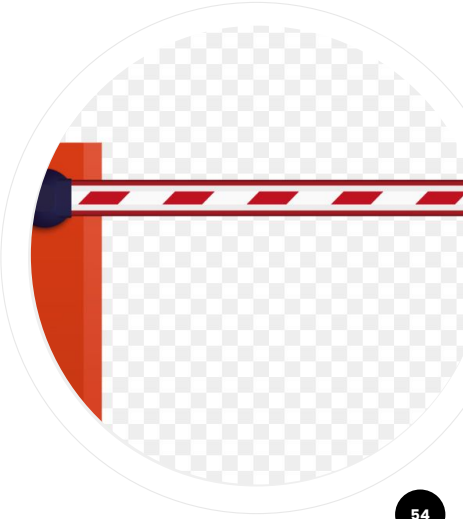
Escalonamento

Worksharing

De modo geral, entende-se que:

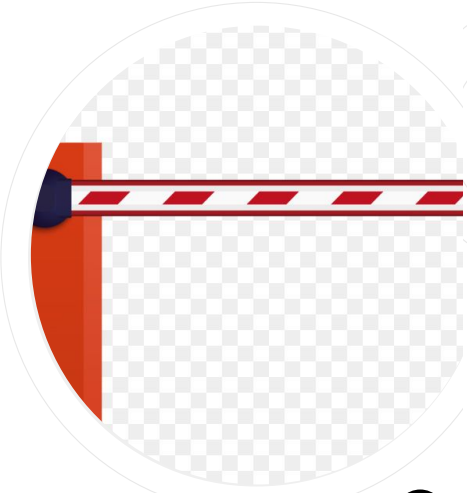
- Todas as tarefas são criadas ao mesmo tempo na operação *fork*
- Uma tarefa ao é suspensa (bloqueada) ao realizar um fork
- Uma tarefa suspensa é retomada quando ocorrer o *join*, ou seja, quando todas as tarefas criadas tenham terminado

Condição de barreira



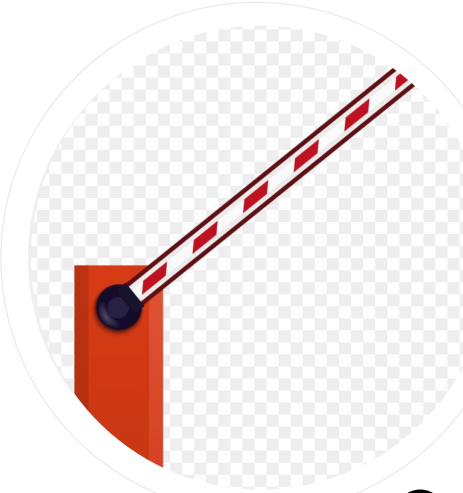
Worksharing

Todas as tarefas são armazenadas em uma lista de trabalho compartilhada, prontas para serem consumidas (*executadas*) pelos threads do pool de execução. Não há distinção entre as tarefas, elas possuem, sob a ótica do escalonamento, as mesmas propriedades.



Worksharing

A cláusula `nowait`, adicionada a uma diretiva de paralelização, flexibiliza a execução a condição de barreira, permitindo que os threads, ao terminarem seu quinhão de tarefas, possam avançar na execução. Assim, uma nova diretiva de paralelização pode ser executada e as novas tarefas participarão do mesmo escalonamento.



Worksharing: Laço paralelo

Em um laço paralelo, o espaço de iteração é fracionado em *chunks* e estes chunks são alocados aos threads. Além de ser possível indicar o tamanho do *chunk*, é possível selecionar uma entre três estratégias de escalonamento:

- `static`
- `dynamic`
- `guided`

Existe também default e auto, mas eles aplicam uma das três estratégias indicadas

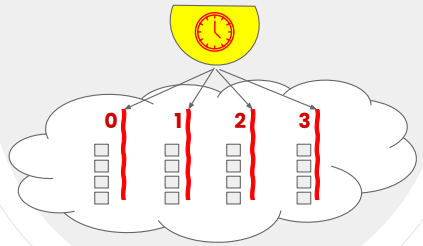
Worksharing: Laço paralelo

- `static`

As tarefas criadas são alocadas diretamente aos threads aos quais irão executar.

```
#pragma omp for schedule(static,X)
```

Ideal quando o custo computacional não varia conforme a iteração sendo executada. Todos os threads executarão a mesma carga de trabalho.



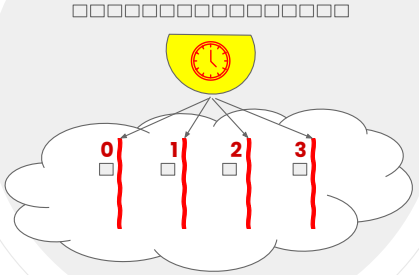
Worksharing: Laço paralelo

- `dynamic`

As tarefas criadas são alocadas em uma lista de trabalho compartilhada. O ideal é que o *chunk* selecionado seja suficiente para criar tarefas de forma que haja uma reserva nesta lista.

```
#pragma omp for schedule(dynamic,X)
```

Ideal quando o custo computacional depende da iteração sendo executada. Se isso ocorre, os custos computacionais das tarefas muda conforme o chunk executado. Esta estratégia busca balancear o custo computacional. Aspecto a ser considerado: há overhead na sincronização à lista de trabalho compartilhada.



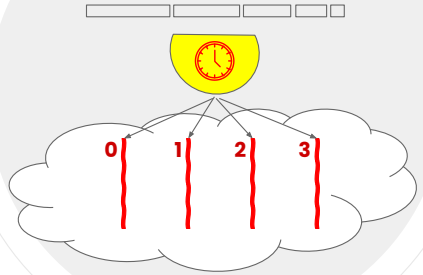
Worksharing: Laço paralelo

- `guided`

Semelhante ao dynamic, mantém uma lista de trabalho compartilhada. No entanto, o tamanho do chunk é decrescente em função da ordem de criação da tarefa.

```
#pragma omp for schedule(guided,X)
```

A ser utilizado em conjunto com uma cláusula nowait que o preceda, contemplando com trabalho os threads a medida em que eles encontram um novo laço paralelo.



A ideia é: quem chegou primeiro, adianta o trabalho!

Escalonando algoritmos recursivos

- `task`
- `taskwait`

As diretivas **`task`**/**`taskwait`** permitem a construção de algoritmos recursivos. Neste caso, as tarefas possuem características distintas:

- mais alto na recursão uma tarefa está, mais trabalho acumulado ela possui
- terminar uma tarefa folha permite recuar no ramo da recursão e liberar memória

Selecionar a próxima tarefa a ser executada afeta o desempenho

Escalonando algoritmos recursivos

- `task`
- `taskwait`

Selecionar a próxima tarefa a ser executada afeta o desempenho

- Ao selecionar em profundidade, busca-se concluir um ramo, explorando a localidade de referência das tarefas
- Ao selecionar em largura, aumenta-se o grau de paralelismo a ser explorado

Escalonando algoritmos recursivos

- `task`
- `taskwait`

Estratégia:

As tarefas, ao serem criadas, são, por default **`tied`**. Isso é, possuem uma propriedade que indica uma relação de dependência forte com a tarefa que a criou. Alternativamente, uma tarefa pode ser criada como **`untie`**d, sendo esta propriedade explicitada como uma cláusula à diretiva **`task`**. Em um determinado thread, a prioridade será dada para executar as tarefas em profundidade, explorando a localidade de referência e buscando resolver as dependências.

Escalonando algoritmos recursivos

- `task`
- `taskwait`

Execução:

Existem duas listas: uma de tarefas prontas para serem executadas (recém criadas ou com requisitos de sincronização satisfeitos) e outra de tarefas bloqueadas em uma sincronização.

A decisão de escalonamento ocorre quando uma tarefa é criada ou termina ou ainda quando uma operação de sincronização (**`taskwait`**, **`critical`**,...) é executada.

Escalonando algoritmos recursivos

- `task`
- `taskwait`

Execução:

Ao entrar em operação o escalonador:

- inicia a execução de uma tarefa que esteja pronta, não importando que seja `tied` ou `untied` às tarefas em execução

  OU

- retoma a execução de uma tarefa que estava bloqueada e que esteja pronta novamente

A decisão depende do estado da lista de tarefas bloqueadas em um determinado thread.

Escalonando algoritmos recursivos

- `task`
- `taskwait`

Execução:

A decisão depende do estado da lista de tarefas bloqueadas em um determinado thread.

- se esta lista está vazia, uma nova tarefa `tied` poderá ser lançada
- caso contrário, uma nova tarefa `tied` somente poderá ser lançada caso ela seja descendente de todas as tarefas que estejam na lista de bloqueadas

Programando Concorrente

1. Crie um programa que receba como entrada o número de threads a ser criado no pool de execução e o tamanho de um vetor de inteiros. Inicialmente o vetor deve ser criado e, então, inicializado com valores quaisquer. Este programa deve imprimir, como saída, o somatório de todos os valores do vetor.
2. Avalie o desempenho do programa desenvolvido variando o número de threads no pool de execução e o tamanho do vetor.

Atividade de Acompanhamento
https://forms.gle/hH7ty1KbR1SmwPQi8

Programando Concorrente

3. Implemente duas versões de um algoritmo recursivo para cálculo de Fibonacci:
   a. Utilizando a criação de tarefas recursivas utilizando task/taskwait
   b. Utilizando sections/section.
4. Compare o desempenho das implementações realizadas, variando a posição n calculado na série e o número de threads no pool de execução.


Obrigado!