OpenMP

João Marcelo Uchôa de Alencar
joao.marcelo@ufc.br
UFC-Quixadá

[1] https://computing.llnl.gov/tutorials/openMP/

Introdução

- OpenMP é uma *Application Program Interface* - API;
- assim como o MPI, é definida por um consórcio da indústria e academia;
- um modelo portável e escalável para desenvolvedores de aplicações de memória compartilhada;
- suportado oficialmente por C/C++ e Fortran.

Introdução

O OpenMP é:

- Uma API que pode ser usada para controlar de maneira explícita paralelismo *multithread* de **memória compartilhada**;
- são três suas espécies de componentes principais:
  - Diretivas de Compilação;
  - rotinas de biblioteca em tempo de execução;
  - variáveis de ambiente.

Introdução

O OpenMP **não** é:

- Feito para memória distribuída (quando usado sozinho);
- implementado de forma idêntica por todos os compiladores;
- uma garantia de que os acessos à memória compartilhada são feitos da melhor forma possível;
- imune a problemas de dependência de dados e concorrência;
- projetado para E/S paralela.

Objetivos do OpenMP

- Padronização:
  - Fornece um padrão consistente entre uma variedade de arquiteturas de memória compartilhada;
  - definido e apoiado em conjunto por diversos fabricantes e desenvolvedores.
- simplicidade:
  - Estabelece um conjunto limitado e bem definido de diretivas;
  - um alto nível de paralelismo pode ser atingido com poucas linhas de código;
  - entretanto, com o tempo novas funcionalidades tem colocado em risco essa característica.

Objetivos do OpenMP

- Facilidade de uso:
  - Favorece uma metodologia de paralelismo incremental, o desenvolvedor não precisa considerar chamadas de comunicação desde o início do desenvolvimento;
  - diferentes níveis de granularidade.
- portabilidade:
  - API suportada por C/C++ e Fortran;
  - um fórum público para discussão de novas funcionalidades;
  - a maioria das grandes plataformas têm suporte.

História

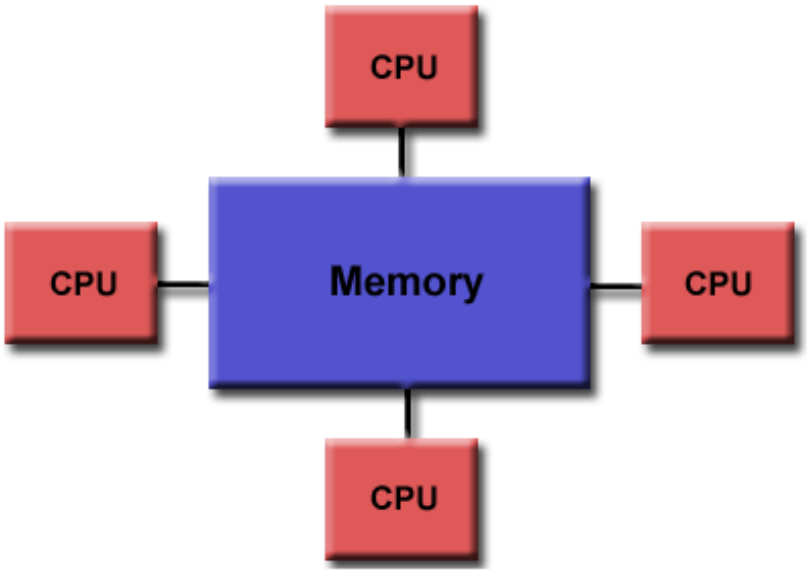
- Na década de 90, vários fabricantes forneciam diretivas similares, mas incompatíveis, para Fortran:
  - A partir de um laço em Fortran, o desenvolvedor adicionava uma diretiva que distribuia as iterações em *threads*;
  - o compilador era responsável pela criação e gerência dos *threads*;
- existiam muitas similaridades entre as implementações, mas eram incompatíveis;
- houve uma tentativa de padronização em 1994 (ANSI X3H5), mas perdeu tração com o advento dos *clusters*;
- entretanto, com o tempo, as máquinas SMP/SMT começaram a ser lançadas;
- em 1997, a padronização ganhou novo fôlego;
- *OpenMP Architecture Review Board (ARB)*.

Histórico de Versões

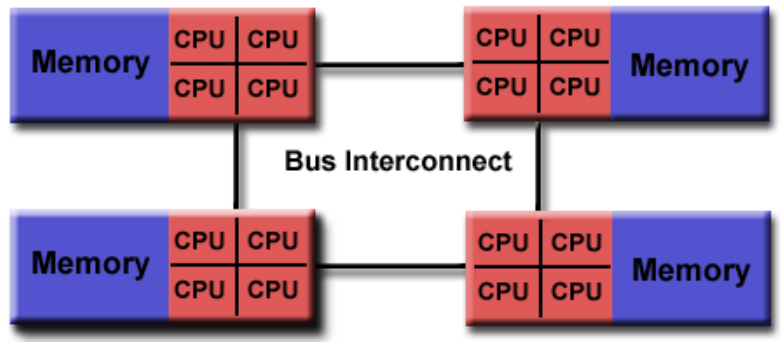
Data	Versão
1997	Fortran 1.0
1998	C/C++ 1.0
1999	Fortran 1.1
2000	Fortran 2.0
2002	C/C++ 2.0
2005	OpenMP 2.5
2008	OpenMP 3.0
2011	OpenMP 3.1
2013	OpenMP 4.0
2015	OpenMP 4.5

Vamos basear nossos estudos na versão 3.0/3.1. Mais informações em `http://openmp.org/`.

Modelo de Memória Compartilhada



Acesso Uniforme a Memória.



Acesso não Uniforme a Memória.

OpenMP é projetado para máquinas com vários núcleos acessando uma memória compartilhada.

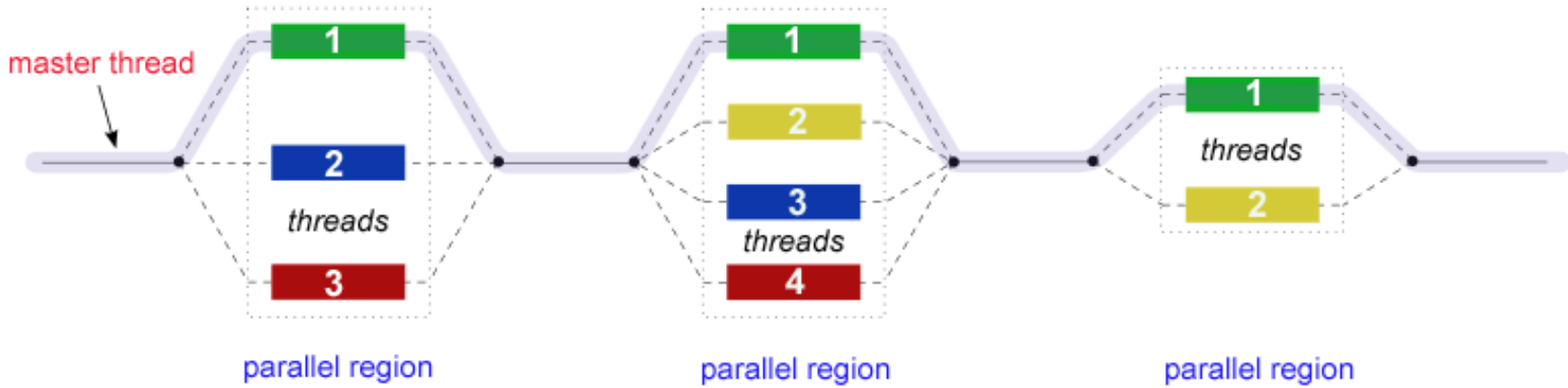
Paralelismo Baseado em *Threads*

- OpenMP atinge o paralelismo através exclusivamente de *threads*;
- considere *thread* como a menor unidade de processamento que pode ser escalonada pelo sistema operacional;
- a implementação interna de *threads* dentro do *kernel* varia entre sistemas e foge do escopo da especificação do OpenMP;
- *threads* estão associadas a um processo;
- em geral, tenta-se manter o número de *threads* igual ao número de núcleos/processadores disponíveis, mas cabe ao desenvolvedor escolher a cardinalidade dentro de uma aplicação.

Paralelismo Explícito

- OpenMP **não** é paralelismo automático. Facilita a descrição, mas ainda cabe ao desenvolvedor descrever o paralelismo de forma explícita;
- essa descrição pode ser simples: a partir de um programa serial correto, gradativamente adicionar diretivas até atingir o nível de paralelismo desejado;
- mas há possibilidade para maior elaboração usando subrotinas e paralelismo aninhado (*threads* criando *threads*).

Modelo *Fork-Join*



Modelo *Fork-Join*

- O OpenMP adota o modelo *fork-join* para execução paralela;
- todos os programas OpenMP começam como um único processo, ou *master thread*. Ele executa de forma serial até encontrar a primeira região paralela;
- **FORK**: o *master thread* cria então um time de *threads* paralelos;
- as sentenças do programa definidas pelos construtores da região paralela são executadas em paralelo em todos os *threads*;
- **JOIN**: quando os *threads* completam as instruções da região paralela, eles sincronizam e finalizam. Resta apenas o *master thread*;
- o número de regiões paralelas e quantidade de *threads* em cada uma é arbitrário.

Escopo de Dados

- Já que se trata de uma solução para memória compartilhada, por *default*, os dados na região paralela são compartilhados;
- todos os *threads* da região paralela podem acessar os dados simultaneamente;
- OpenMP fornece meios de controlar esse acesso para garantir integridade dos cálculos;
- mais adiante veremos como o acesso a uma região crítica pode ser feito.

Paralelismo Aninhado

- Paralelismo aninhado:
  - Podemos criar regiões paralelas dentro de regiões paralelas;
  - a questão é que o suporte varia de acordo com a implementação do compilador.
- *threads* dinâmicos:
  - Caso seja necessário, o desenvolvedor pode alterar o número de *threads* em uma região paralela durante a execução;
  - novamente, o compilador precisa suportar essa *feature*.

Entrada e Saída

- E/S:
  - OpenMP não trata de E/S paralela, se vários *threads* acessam um mesmo arquivo, o desenvolvedor tem que tratar o acesso;
  - agora se a E/S ocorre em arquivos diferentes, não há problema;
- modelo de memória:
  - OpenMP fornece uma *consistência relaxada*: *threads* podem manter cópias locais e não precisam manter consistência com as cópias globais em tempo real;
  - o desenvolvedor pode executar uma operação *flush* e forçar a atualização.

Visão Geral da API OpenMP

- A API do OpenMP versão 3.1 tem os seguintes grupos de componentes:
  - 19 diretivas de compilação;
  - 32 rotinas de ambiente de execução;
  - 9 variáveis de ambiente.
- o desenvolvedor decide como usar esses componentes, mas em geral apenas alguns são mais utilizados;
- os compiladores divergem no total de componentes suportados;
- quanto mais antiga a especificação, maior chance de todos seus componentes estarem suportados pelos compiladores atuais.

Diretivas de Compilação

- Diretivas inicialmente aparentam ser simples **comentários** no código, sem efeito aparente;
- mas informando um parâmetro no momento da compilação, o desenvolvedor pode levar o compilador a gerar paralelismo;
- propósitos das diretivas no OpenMP:
  - Iniciar uma região paralela;
  - dividir blocos de código entre *threads*;
  - distribuir iterações de um laço entre *threads*;
  - serializar seções do código;
  - sincronização.

Diretivas de Compilação

Formato geral:

```
sentinela      nome-da-diretiva [clausulas,...]
#pragma omp    parallel         default(shared) private(i)
```

Rotinas de Ambiente de Execução

- Configurar e recuperar o número de *threads*;
- recuperar o identificador do *thread*;
- configurar o comportamento dinâmico do *thread*;
- determinar o nível da região paralela que o *thread* se encontra;
- configurar o paralelismo aninhado;
- configurar travas para controle de regiões críticas;
- medir o tempo de execução.

Rotinas de Ambiente de Execução

Formato geral:

```
#include <omp.h>
int omp_get_num_threads(void)
```

É preciso incluir o cabeçalho *omp.h*.

Variáveis de Ambiente

- Configuração o número *default* de *threads*;
- definir como as iterações são divididas;
- ligar um *thread* a um processador específico;
- configurar o paralelismo aninhado;
- configurar o tamanho da pilha;
- configurar o escalonamento de *threads*.

Algumas ações podem ser feitas tanto por rotinas quanto por variáveis de ambiente. A diferença é que a variável de ambiente não envolve alterações no código.

Variáveis de Ambiente

Formato geral:

```
$ export OMP_NUM_THREADS=8
```

É preciso definí-las para cada sessão do *Shell* ou configurá-las para serem automaticamente carregadas.

Estrutura Geral de um Programa OpenMP

```
#include <omp.h>

main ()  {
   int var1, var2, var3;
   ...
   // Código Serial
   ...
   // Início de Região paralela
   #pragma omp parallel private(var1, var2) shared(var3)
   {
      ...
      // Região paralela Executada pelos threads
      ...
   }
   // Código Serial
   ...
}
```

Compilando um Programa OpenMP

```
$ gcc -fopenmp olamundo.c -o olamundo
```

Diretivas OpenMP

# pragma omp	nome da diretiva	[cláusula, ...]	nova linha
obrigatório	posição fixa	opcional	obrigatória

```
#pragma omp parallel default(shared) private(beta,pi)
```

Diretivas OpenMP

- *Case sensitive*;
- estão de acordo com as regras gerais de diretivas para C/C++;
- só pode ser usado um **nome de diretiva** por declaração de diretiva;
- cada diretiva se aplica a próxima declaração, ou a um bloco seguinte;
- se a diretiva começar a ficar muito longa, pode ocupar várias linhas separadas por \;

Escopo de Diretivas

- Estático:
  - Código encapsulado entre o começo e o fim de um região paralela;
  - o efeito da diretiva não abrange várias rotinas ou outros arquivos de código.
- diretiva órfã:
  - É uma diretiva que aparece independente e qualquer outra diretiva externa;
  - tem seu efeito em várias rotinas e arquivos de código.
- dinâmico:
  - combina os dois escopos acima para definir o escopo dinâmico de um trecho de código.

Diretiva PARALLEL

Uma região paralela é um bloco de código que será executado por múltiplos *threads*.

```
#pragma omp parallel [clausula ...]  novalinha
                     if (expressao escalar)
                     private (lista)
                     shared (lista)
                     default (shared | none)
                     firstprivate (lista)
                     reduction (operator: lista)
                     copyin (lista)
                     num_threads (expressao inteira)
// bloco estruturado
```

Diretiva PARALLEL

- Quando um *thread* atinge uma diretiva PARALLEL, ele cria um time de *threads* e se torna o mestre deles. O mestre é membro do time e dentro da região paralela tem identificador com valor 0;
- a partir do início da região paralela, o código é duplicado para todos os *threads*;
- há uma barreira implícita no fim da região paralela. Apenas o *thread* mestre continua após essa barreira;
- se algum dos *threads* finalizar enquanto na região paralela, todos os outros também finalizam. O trabalho feito fica indefinido.

Quantas *Threads*?

- Avaliação da cláusula *IF*;
- configuração da cláusula *NUM_THREADS*;
- uso da função *omp_set_num_threads*();
- configuração da variável de ambiente *OMP_NUM_THREADS*;
- padrão da implementação, em geral, a quantidade de núcleos ou processadores presentes.

Os *threads* são numerados de 0 a $N - 1$.

Threads Dinâmicos

Nem sempre estão habilitados no compilador.

- Use a função *omp_get_dynamic*() para determinar se podem ser habilitados;
- se a função retornar verdadeiro, existem duas maneiras para ativar *threads* dinâmicos:
  - *omp_set_dynamic*();
  - a variável de ambiente *OMP_DYNAMIC*.

Regiões Paralelas Aninhadas

Outra funcionalidade que nem sempre é suportada.

- Use a função *omp_get_nested*() para determinar se são suportadas;
- novamente, duas maneiras para ativá-las:
  - *open_set_nested*();
  - a variável de ambiente *OMP_NESTED*.
- se não tiver suporte, uma região paralela dentro de outra região paralela irá resultar na criação de um time de *threads* com um único *thread*.

Cláusulas

- As cláusulas permite configuração paralela;
- vamos ver com detalhes cada uma delas;
- mas por enquanto, por exemplo, a cláusula *IF*:
  - recebe uma expressão que pode ser avaliada para verdadeiro ou falso;
  - se a avaliação retornar verdadeiro, a região paralela é executada por um grupo de *threads* paralelos;
  - se retornar falso, a região paralela é executada de forma serial por um único *thread*.

Restrições

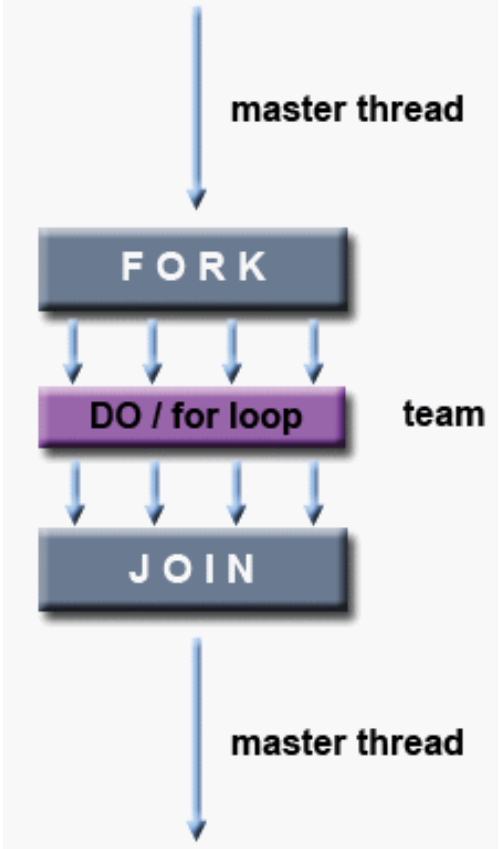
- Uma região paralela deve ser um bloco estruturado;
- é ilegal saltar para fora de uma região paralela (*goto*);
- apenas uma cláusula *IF* é permitida;
- apenas uma cláusula *NUM_THREADS* é permitida;
- alterar a ordem das cláusulas não irá alterar a criação da região.

Compartilhamento de Trabalho

- As diretivas dessa espécie são responsáveis em dividir o trabalho entre as *threads* de uma região paralela criada por um diretiva PARALLEL;
- elas devem estar associadas a uma região paralela, não criam novos *threads* por si só;
- não há uma barreira na entrada de uma diretiva de compartilhamento de trabalho, mas há uma na saída.

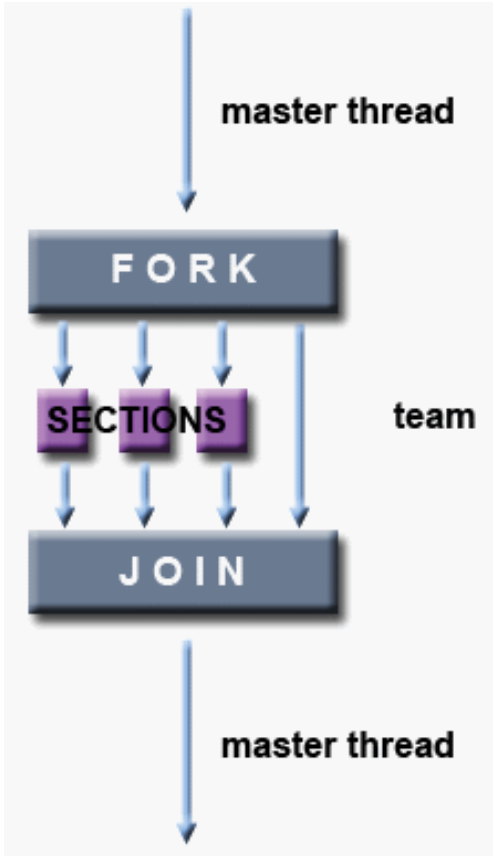
For Paralelo

Paralelismo de Dados: distribui as iterações de um laço entre os *threads* de uma região paralela.



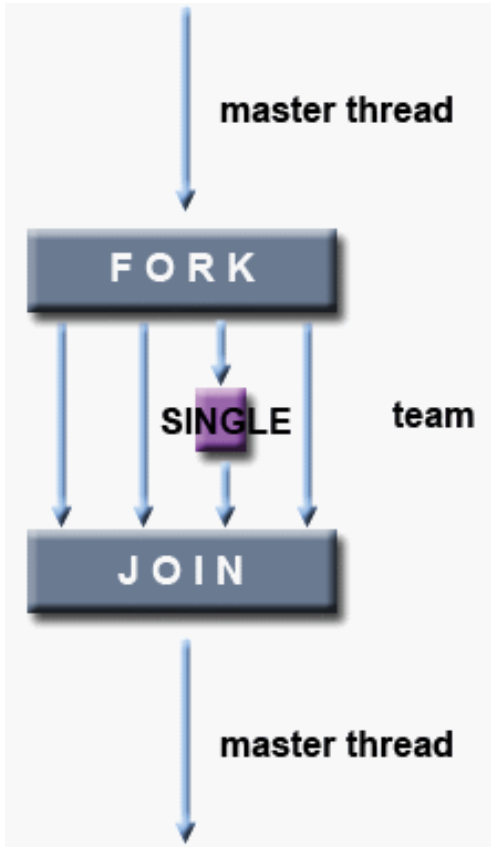
Sections

Paralelismo Funcional: define seções diferentes e discretas de código, cada uma executada por um *thread*.



Single

Força uma parte de uma região paralela ser executada por apenas um *thread*.



Restrições

- As cláusulas de distribuição de compartilhamento de trabalho deve estar associadas a uma região paralela;
- todos os *threads* precisam avaliar essas cláusulas;
- se mais de uma cláusula for usada, elas devem aparecer na mesma ordem para todos os *threads*.

For Paralelo

```
#pragma omp for [cláusula ...]  novalinha
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait

// laço for
```

Cláusula *SCHEDULE*

Determina como iterações do laço são distribuídas entre o *thread*.

- STATIC: o total de iterações é dividido pela quantidade de *threads*. O resultado, chamado *chunk*, determina quantas iterações cada *thread* executará. Os *threads* recebem as iterações pela ordem dos identificadores;
- DYNAMIC: as iterações são divididas em um tamanho fixo também chamado *chunk*. Cada *thread* recebe uma *chunk* e a medida que termina a execução, faz requisição de uma nova *chunk*;
- GUIDED: similar a DYNAMIC, mas o *chunk* diminui cada vez que um *thread* faz uma nova requisição;
- RUNTIME: força a distribuição a ocorrer apenas durante a execução (ela pode ser pré-definida na compilação);
- AUTO: deixa a cargo do sistema operacional e compilador.

Outras Cláusulas do *For*

- NO WAIT: desativa a barreira ao final do laço paralelo;
- ORDERED: força e execução as iterações na ordem do programa paralelo;
- COLLAPSE: no caso de um laço dentro um laço, pode transformar todas as iterações em um grande laço que é paralelizado.
- veremos outras cláusulas de escopo de dados mais adiante.

Restrições

- O laço a ser paralelizado não pode ser um *while*;
- a corretude do programa não pode depender da ordem de execução das iterações do laço;
- o tamanho do *chunk* deve ser definido por uma expressão inteira;
- ORDERED, COLLAPSE e SCHEDULE só podem aparecer uma por vez.

Sections

- Ao contrário da *for*, essa diretiva define trechos de códigos diferentes para execução entre os *threads*;
- temos uma diretiva externa, *sections*, que informa que em seguida estão diretivas de código para os *threads*, chamadas *section*;
- cada *section* é executada por *thread*;
- *sections* diferentes podem ser executadas por *threads* diferentes;
- se um *thread* for muito rápido, pode executar mais de uma *section*.

Sections

```
#pragma omp sections [cláusula ...]  novalinha
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
  {

  #pragma omp section   novalinha

     bloco estruturado

  #pragma omp section   novalinha

     bloco estruturado

  }
```

Sections

- Existe uma barreira implícita ao fim de uma diretiva *sections*;
- a barreira pode ser desativada pelo uso da cláusula *nowait*;
- veremos mais adiante as cláusulas de compartilhamento de variáveis;
- o que acontece se o número de *sections* é diferente do número de *threads*?
  - Mais *threads* do que *sections*, alguns *threads* ficarão ociosos;
  - mais *sections* do que *threads*, o sistema operacional irá escalonar as *sections* entre as *threads*.
- não há como saber qual *thread* executa qual *section*.

Single

- A diretiva *single* especifica que o código seguinte é para ser executado por apenas um *thread*;
- é útil para lidar com entrada e saída;
- os outros *threads* que não foram escolhidos para executar o código *single* esperam ao final do bloco, em uma barreira;
- essa barreira pode ser desabilitada pela cláusula *nowait*.

Single

```
#pragma omp single [cláusula ...]  novalinha
                   private (list)
                   firstprivate (list)
                   nowait

    bloco estruturado
```

Combinação de Diretivas de Compartilhamento de Trabalho

- Podemos combinar as diretivas para facilitar a criação de regiões paralelas:
  - *parallel for*;
  - *parallel sections*.
- na maioria dos casos, usar a combinação tem o mesmo efeito de declaração uma região com *parallel* e logo em seguida aplicar uma cláusula *for* ou *sections*;

Task

- Cada *section* está agrupada na seção maior *sections*, sendo os *threads* só deixam esse bloco estruturado quando todas as *sections* forem executadas;
- a diretiva *task* permite maior flexibilidade;
- *tasks* são enfileiradas e executadas sempre que possível nos *task scheduling points*;
- uma *task* pode ser movida de um *thread* para outro;

Task versus Sections

```
#pragma omp sections
{
   #pragma omp section
   foo();
   #pragma omp section
   bar();
}
#pragma omp single nowait
{
   #pragma omp task
   foo();
   #pragma omp task
   bar();
}
#pragma omp taskwait
```

Task

- *taskwait* é uma barreira: a execução é pausada até todas as *tasks* executem;
- usamos *single* para que um único *thread* crie duas *tasks*;
- a cláusula *nowait* faz com que os outros *threads* que não são escolhidos para executar o bloco *single* saltem até a barreira *taskwait*;
- quando todos chegam a *taskwait*, todas as tarefas criadas são escalonadas entre os *threads*.

As *sections* são executadas assim que encontradas. As *tasks* são criadas, não executam de imediato, apenas em uma barreira (implícita ou explícita) é que são distribuídas entre os *threads*.

Sincronização

Considere um exemplo no qual dois *threads* buscam atualizar uma mesma variável.

```
// Thread 01
update(x)
{
    x = x + 1
}
x = 0
update(x)
print(x)
```

```
// Thread 02
update(x)
{
    x = x + 1
}
x = 0
update(x)
print(x)
```

Sincronização

- Uma possível sequência de execução:
  1. *thread* 1 inicializa x para 0 e invoca *update*(x);
  2. *thread* 1 adiciona 1 a x. x agora é igual a 1;
  3. *thread* 2 inicializa x para 0 e invoca *update*(x). x agora é igual a 0;
  4. *thread* 1 imprime x, que é igual a 0 no lugar de 1;
  5. *thread* 2 adiciona 1 a x. x agora é igual a 1;
  6. *thread* 2 imprime x como 1.
- é preciso sincronizar o acesso a x.

Diretiva MASTER

- Determina que um *thread* específico, o mestre, execute a seção de código;
- não há barreira, os outros *threads* podem continuar.

```
#pragma omp master  novalinha

   bloco estruturado
```

Diretiva CRITICAL

- Determina que um trecho de código seja executado apenas por um *thread* por vez;
- se mais de um *thread* tentar executar um trecho de código de uma diretiva CRITICAL, haverá bloqueio;
- é possível nomear as regiões CRITICAL:
  - regiões CRITICAL diferentes com o mesmo nome são tratadas como uma única região;
  - todas as regiões em nome são tratadas como uma única região.

```
#pragma omp critical ( nome )  novalinha

  bloco estruturado
```

Diretiva BARRIER

- Sincroniza todos os *threads* do time;
- quando uma barreira é atingida, o *thread* espera até todos os outros também atinjam;
- todos os *threads* ou nenhum transpõem a barreira;
- não é possível usar condições para apenas um subgrupo dos *threads* encontrarem a barreira.

```
#pragma omp barrier novalinha
```

Diretiva TASKWAIT

- Força que os *threads* aguardem a finalização de qualquer *task* criada antes da diretiva.

```
#pragma omp taskwait novalinha
```

Diretiva ATOMIC

- Especifica uma região da memória que deve ser atualizada de forma atômica;
- é uma maneira de criar uma região CRITICAL composta de apenas uma expressão.

```
#pragma omp atomic novalinha

   expressao
```

Diretiva FLUSH

- Força que todos os *threads* atualizem a memória global do processo com os valores das suas cópias;
- em um sistema que use um protocolo de coerência de *cache*, a FLUSH ainda é necessária pois os *threads* podem ter cópias em registradores;
- existe um FLUSH implícito na entrada ou saída da maioria das diretivas que vimos até agora.

```
#pragma omp flush (variaveis)  novalinha
```

Diretiva ORDERED

- Especifica que as iterações de um laço devem ser executadas na mesma ordem que seriam executadas em serial;
- *threads* terão que esperar para executar seu *chunk* se as iterações anteriores não tiverem concluído;
- faz sentido no uso da diretiva FOR;
- permite que uma seção do laço seja executada em paralelo, mas que outra parte exija a ordem.

Diretiva ORDERED

```
#pragma omp for ordered [clausulas...]
      (inicio do laco)

#pragma omp ordered  novalinha
      bloco estruturado

      (fim do laco)
```

Diretiva THREADPRIVATE

- Serve para tornar persistente na memória particular de um *thread* variáveis globais entre diversas regiões paralelas;
- cria uma cópia local para cada *thread*;
- otimiza o uso da *cache*;
- a diretiva COPYIN permite transferir o valor anterior para as cópias locais;
- só funciona se o número de *threads* entre as regiões paralelas se mantiver o mesmo.

```
#pragma omp threadprivate (variaveis)
```

Cláusulas de Escopo de Dados

- A maioria das variáveis são compartilhadas por padrão;
- em C/C++, isso inclui as variáveis globais do arquivo e as estáticas;
- as variáveis privadas incluem:
  - Índices de laços;
  - variáveis na pilha de funções chamadas dentro de uma região paralela;
- As cláusulas de escopo de dados devem ser usadas em conjunto com diretivas PARALLEL, FOR, SECTIONS, etc.

Cláusulas de Escopo de Dados

- As cláusulas de escopo de dados são:
  - PRIVATE;
  - FIRSTPRIVATE;
  - LASTPRIVATE;
  - SHARED;
  - DEFAULT;
  - COPYIN.
  - REDUCTION;
- permite controlar o ambiente de dados durante a execução de regiões paralelas;
  - Definem como e quais variáveis da região serial são transferidas para a região paralela, e como são trazidas de volta;
  - definem quais variáveis são visíveis a todos os *threads* em uma região paralela, e quais são privadas.

Cláusula PRIVATE

- Declara as variáveis na lista privadas de cada *thread*;
- um novo objeto do mesmo tipo da variável global é criado em cada *thread*;
- todas as referências para o objeto original são direcionadas para o novo objeto;
- as variáveis marcadas com PRIVATE devem ser consideradas não inicializadas.

```
private (lista)
```

PRIVATE versus THREADPRIVATE

	PRIVATE	THREADPRIVATE
Item de Dados	variáveis	variáveis
Declaração	início da região	globais
Persistência	Não	Sim
Escopo	Léxico	Dinâmico
Inicialização	FIRSTPRIVATE	COPYIN

Cláusula SHARED

- Indica que as variáveis na lista pertencem a todas os *threads*;
- uma variável SHARED existe apenas uma posição na memória principal, o mesmo endereço é acessado por todas os *threads*;
- é responsabilidade do desenvolvedor sincronizar o acesso;
- mesmo sendo o padrão é muitos casos, é interessante mesmo assim marcar a variável como SHARED por questões de legibilidade.

```
shared (lista)
```

Cláusula DEFAULT

- Permite definir o escopo de qualquer variável não mencionada de forma explícita;
- precisa ser usada em conjunto com um das outras cláusulas de escopo;
- o valor NONE exige que o desenvolvedor defina o escopo de todas as outras variáveis;
- só pode existir uma versão de DEFAULT.

```
default (shared | none | private | ...)
```

Cláusula FIRSTPRIVATE

- Combina o comportamento de PRIVATE com inicialização das variáveis;
- as variáveis são inicializadas com os valores que possuíam imediatamente antes da região paralela.

```
firstprivate (lista)
```

Cláusula LASTPRIVATE

- Sinaliza a cópia do valor da variável na última iteração de um laço ou seção de volta para a variável original;
- é um comportamento um pouco imprevisível, visto que não temos como saber exatamente qual será o último *thread* a executar a última iteração ou seção;
- mas é útil para depuração e controle da execução.

```
lastprivate(lista)
```

Cláusula COPYIN

- Permite fornecer o mesmo valor para variáveis já marcadas como THREADPRIVATE para todos os *threads* num time;
- a lista contém as variáveis para copiar;
- o *thread* mestre é usado como origem da cópia.

```
copyin (lista)
```

Cláusula COPYPRIVATE

- Permite que um *thread* copie o valor de uma variável privada em todas as outras versões privadas da variável em outros *threads*;
- pense como um *broadcast*;
- deve ser usada em associação a uma diretiva SINGLE.

```
copyprivate (lista)
```

Cláusula REDUCTION

- Realiza uma redução nas variáveis descritas na lista;
- uma cópia privada de cada variável é criada em cada *thread*;
- ao final da redução, todas as cópias são usadas como operandos e o valor final é armazenado na variável global.

```
reduction (operator: lista...)
```

Cláusula REDUCTION

Operação	Operador	Inicialização
Adição	+	0
Multiplicação	*	1
Subtração	-	0
AND Lógico	&&	0
OR Lógico	||	0
AND *bits*	&	1
OR *bits*	|	0
OR Exclusivo *bits*	^	0
Máximo	max	
Mínimo	min	

Cláusula REDUCTION

- O tipo da lista deve ser compatível com a operação;
- as variáveis não podem ser declaradas SHARED ou PRIVATE;
- é preciso ter cuidado com números de ponto flutuante.

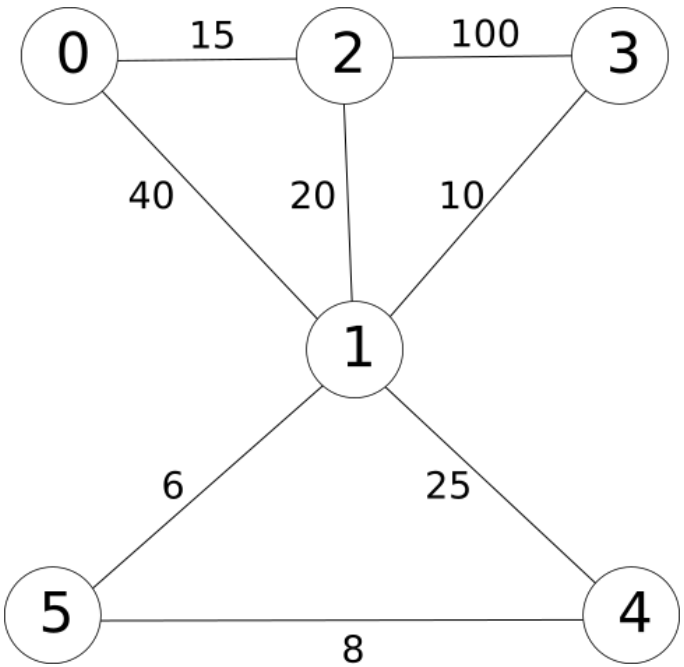
Estudo de Caso de Paralelização com OpenMP

- Dado um grafo $G = (V, E)$, encontrar a distância mínima de um vértice a todos os outros;
- algoritmo de Dijkstra;
- vamos apresentar o algoritmo em pseudocódigo;
- considerar a implementação serial;
- discutir como usar as diretivas OpenMP para atingir paralelismo.

Algoritmo de Dijkstra

```
procedimento Dijkstra(grafo, origem):
    criar um conjunto de vertices Q
    para cada vertice v no grafo:
        dist[v] = INFINITO
        prev[v] = INDEFINIDO
        adicione v a Q
    dist[origem] = 0
    enquanto Q nao vazio:
        u = vertice em Q com menor valor de dist[u]
        remove u de Q
        para cada vizinho v de u, sendo v ainda em Q:
            alt = dist[u] + distancia(u, v)
            se alt < dist[v]:
                dist[v] = alt
                prev[v] = u
    retorn dist, prev
```

Algoritmo de Dijkstra - Exemplo



Origem Destino	0	1	2	3	4	5
0	0	40	15	-	-	-
1	40	0	20	10	25	6
2	15	20	0	100	-	-
3	-	10	100	0	-	-
4	-	25	-	-	0	8
5	-	6	-	-	8	0