Capítulo 2

Programação Paralela com OpenMP: Modelo de Tarefas

Calebe P. Bianchini, Gabriel P. Silva

Abstract

This mini-course aims to present parallel programming techniques using the OpenMP task parallelization model. The various directives and clauses available to implement this model will be covered, as well as the details that must be observed to ensure the correctness and best performance of the final code.

Resumo

Este minicurso tem como objetivo apresentar técnicas de programação paralela utilizando o modelo de paralelização de tarefas do OpenMP. Serão abordadas as diversas diretivas e cláusulas disponíveis para implementar esse modelo, além dos detalhes que devem ser observados para garantir a corretude e melhor desempenho do código final.

2.1. Introdução

O OpenMP [OpenMP ARB, 2015] é um padrão que define uma API (Interface de Programação de Aplicações) para a programação de múltiplas *threads* em computação paralela. Ele é usado principalmente em sistemas com múltiplos processadores ou núcleos de processamento com memória compartilhada para acelerar a execução de programas.

Os princípios fundamentais do OpenMP incluem: facilidade de uso, com emprego de diretivas de compilador; portabilidade, podendo ser compilado e executado em diferentes sistemas sem modificações significativas; uso de um modelo de programação baseado em *threads*; utilização de *pragmas* (linhas iniciadas por **#pragma**) inseridas no código-fonte como diretivas para informar ao compilador como o paralelismo deve ser aplicado; a criação, nas regiões paralelas, de múltiplas *threads* para executar o código paralelamente; a criação, normalmente, de uma equipe de *threads* trabalhadoras logo na primeira região paralela encontrada, como forma de otimizar o desempenho; uso de mecanismos para

controlar e sincronizar as *threads*, como diretivas para especificar regiões críticas (que devem ser executadas por apenas uma *thread* por vez) e para lidar com a exclusão mútua.

O paralelismo de laços [Silva et al., 2022] foi amplamente explorado nas primeiras versões do padrão OpenMP, sendo utilizado para acelerar significativamente a execução de laços que são computacionalmente intensivos. Essa técnica consiste em dividir as iterações de um laço em blocos de iterações com tamanho igual, onde cada bloco é executado por uma *thread* diferente. Para que o paralelismo de laços funcione corretamente, as iterações do laço não podem ter dependências entre si, ou seja, o resultado de uma iteração não pode depender do resultado de outra iteração.

Entretanto, ao lidar com problemas em que o fluxo de execução não é regular, a exploração do paralelismo torna-se mais complexa, mesmo quando há várias partes que podem ser executadas de forma independente. Nessas situações, é viável empregar o paralelismo de tarefas (*tasks*), que consiste na execução simultânea de diferentes tarefas independentes dentro do programa [Bianchini et al., 2019]. Essa abordagem permite uma melhor exploração do paralelismo e, consequentemente, melhora no desempenho [van der Pas et al., 2017].

O objetivo deste minicurso é apresentar o modelo de paralelismo de tarefas por meio de uma fundamentação teórica, mas também mostrando o seu impacto em exemplos práticos com o uso do OpenMP 4.5.

2.2. Tarefas

Este seção apresenta os principais conceitos relacionado ao modelo de programação de tarefas do OpenMP [OpenMP ARB, 2015, van der Pas et al., 2017].

2.2.1. Terminologia

As tarefas são unidades de trabalho cuja execução pode ser adiada ou realizada imediatamente. Elas são compostas pelo código a ser executado, um ambiente de dados (que é iniciado na sua criação) e variáveis de controle internas (IVCs em inglês – veja a Seção 2.12). O programador deve garantir que tarefas diferentes possam ser executadas simultaneamente, sem conflito no acesso aos dados compartilhados.

As tarefas são geradas quando uma *thread* encontra uma construção **task, taskloop, paralell, target** ou **teams** (ou qualquer construção combinada que especifique alguma dessas construções). Uma **região de tarefa** é uma região composta por todo o código encontrado durante a execução de uma tarefa, sendo que uma região paralela é composta por uma ou mais regiões de tarefas implícitas.

As tarefas podem receber diversas classificações, que serão utilizadas ao longo deste texto, tais como: **tarefa explícita**, que é gerada quando uma construção **task** é encontrada; uma **tarefa implícita** é gerada por uma região paralela implícita ou quando uma construção **parallel** é encontrada; para uma determinada *thread*, a **tarefa atual** correspondente à região de tarefa que ela está executando; uma tarefa é uma **tarefa filha** da sua região de tarefa geradora, sendo que uma região de tarefa filha não faz parte da região de tarefa geradora; as **tarefas irmãs** são tarefas filhas de uma mesma região de tarefa; uma **tarefa descendente** é a tarefa filha de uma região de tarefa ou de uma de suas

regiões de tarefas descendentes.

Uma **tarefa não vinculada** é uma tarefa que, quando sua região de tarefa é suspensa, pode ser retomada por qualquer *thread* na equipe. Ou seja, a tarefa não está vinculada a nenhuma *thread*. Uma **tarefa não postergada** é uma tarefa para a qual a execução não é adiada em relação à sua região de tarefa geradora. Ou seja, sua região de tarefa geradora é suspensa até que a execução da *tarefa não postergada* seja concluída. Uma **tarefa incluída** é uma tarefa para a qual a execução é sequencialmente incluída na região de tarefa geradora. Ou seja, uma tarefa incluída não é adiada e é executada imediatamente pela *thread* que a encontra. Uma **tarefa mesclada** é aquela na qual o ambiente de dados, incluindo as ICVs, é o mesmo que o de sua região de tarefa geradora. Tanto uma *tarefa não postergada* quanto uma *tarefa incluída* podem se tornar uma **tarefa mesclada**. Uma **tarefa final** força todas as suas tarefas filhas a se tornarem *tarefas finais e incluídas*.

As tarefas também podem ter uma relação de ordenação entre duas tarefas irmãs: a tarefa dependente e uma tarefa predecessora previamente gerada. A **dependência de tarefa** é cumprida quando a tarefa predecessora foi concluída. Em outras palavras, a **tarefa dependente** não pode ser executada até que suas **tarefas predecessoras** tenham sido concluídas.

Finalmente, devemos notar que no OpenMP, embora os termos *construct* e *directive* sejam conceitos relacionados e utilizados indistintamente ao longo deste texto, eles têm significados ligeiramente diferentes:

- **Directive (Diretiva)**: é uma instrução específica no código-fonte que é reconhecida pelo compilador OpenMP e que instrui as *threads* sobre como proceder em uma região paralela. As diretivas iniciam por **#pragma omp** no código fonte C/C++, seguidas por uma palavra-chave que define a ação a ser tomada (por exemplo, ‘parallel‘, ‘for‘, ‘sections‘, ‘task‘, entre outras). As diretivas podem ser seguidas por um bloco de código delimitado por chaves ‘{}‘ ou um comando específico.
- **Construct (Construção)**: Refere-se à combinação de uma diretiva OpenMP específica e o código delimitado por essa diretiva. Por exemplo, um **#pragma omp parallel** em C/C++ é uma diretiva que cria uma região paralela, e todo o bloco de código dentro desse ‘parallel‘ é chamado de ”construct” porque é a combinação da diretiva ‘parallel‘ com o código que está sendo executado em paralelo.

2.2.2. Modelo de Tarefas

Apesar de ser um conceito simples em sua essência, o uso das tarefas apresenta diversas nuances e particularidades que procuraremos mostrar a seguir.

No OpenMP, as construções como **sections**, **for** e **single** são utilizadas para o paralelismo de dados, enquanto a construção **task** é projetada para o paralelismo de tarefas, frequentemente lidando com padrões irregulares no acesso à memória. Para um melhor entendimento dessas construções utilizadas para paralelismo de dados, veja a referência [Silva et al., 2022].

A diretiva **task** cria tarefas independentes que podem ser executadas por qualquer *thread* disponível. Elas podem ser atribuídas dinamicamente às *threads,* permitindo uma

distribuição mais flexível do trabalho. Cada *thread* pode executar uma ou mais tarefas alocadas no *pool* de tarefas. O seu uso é adequado para dividir o trabalho em tarefas menores e independentes, sendo escalável para um grande número de tarefas, sem a necessidade de determinar antecipadamente a estrutura de trabalho.

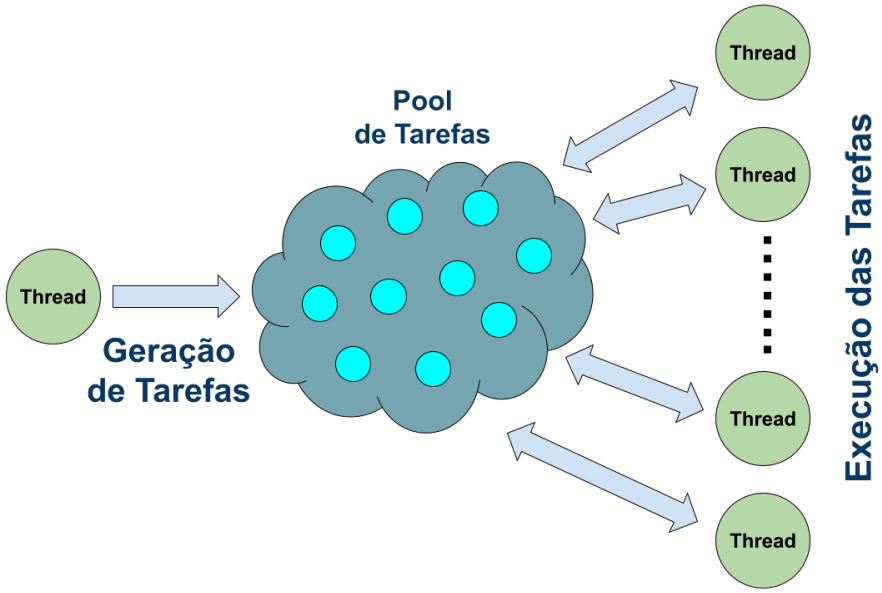

Figura 2.1. Modelo de Tarefas do OpenMP

O modelo de tarefas do OpenMP é representado na Figura 2.1, onde uma *thread* em execução gera as tarefas e as registra em um *pool* de tarefas. Estas, por sua vez, serão retiradas e executadas por uma equipe de *threads* até que não haja mais tarefas no *pool* de tarefas. O ambiente de execução OpenMP é quem decide sobre o escalonamento das tarefas, ou seja, sempre que uma *thread* alcança um ponto de escalonamento de tarefas (veja a Seção 2.2.3), o ambiente de execução pode realizar uma troca de tarefas, iniciando ou retomando a execução de uma tarefa diferente que esteja ligada à equipe de *threads* atual.

Uma vez que uma *thread* começa a executar uma tarefa, ela é designada para executar a tarefa até a sua conclusão, mesmo que possa suspender sua execução em um ponto de escalonamento para retomá-la mais tarde. Ou seja, a menos que uma cláusula **untied** (veja a Seção 2.3) seja utilizada na declaração da tarefa, a *thread* sempre estará vinculada à tarefa que iniciou.

As tarefas podem ser usadas para executar partes do trabalho em laços do tipo **while**; percorrer nós em uma lista encadeada ou em uma árvore de grafo; ou ainda em um laço normal (com uma construção **taskloop** – veja a Seção 2.6). Ao contrário do compartilhamento de trabalho, com escalonamento estático, das iterações de um laço (como no caso do **parallel for**), uma tarefa frequentemente é enfileirada e depois desenfileirada para execução por qualquer uma das *threads* de uma equipe dentro de uma região paralela. A geração de tarefas pode ser feita por um única *thread* geradora (criando tarefas irmãs) ou

por várias *threads* geradoras como no percurso recursivo de uma árvore de grafo.

2.2.3. Pontos de Escalonamento

Um ponto de escalonamento de tarefa é um ponto durante a execução da região de tarefa atual no qual ela pode ser suspensa para ser retomada mais tarde; ou o ponto de conclusão da tarefa, após o qual a *thread* em execução pode mudar para uma região de tarefa diferente. Quando uma *thread* encontra um ponto de escalonamento de tarefa, ela pode tomar uma das seguintes ações:

- iniciar a execução de uma *tarefa vinculada* ligada à equipe de *threads* atual;
- retomar qualquer região de tarefa suspensa, ligada à equipe atual e à qual a *thread* está vinculada;
- iniciar a execução de uma *tarefa não vinculada* ligada à equipe atual;
- retomar qualquer *região de tarefa não vinculada* suspensa, ligada à equipe atual.

Se mais de uma das opções acima estiverem disponíveis, o resultado é indefinido. Os pontos de escalonamento de tarefas dividem dinamicamente as regiões de tarefas em partes, onde cada parte é executada sem interrupções do início ao fim. As diferentes partes da mesma região de tarefas são executadas na ordem em que são encontradas. Na ausência de construções de sincronização de tarefas, a ordem na qual as *threads* executam as partes de diferentes tarefas escalonáveis é indeterminada. As tarefas são criadas explicitamente nos seguintes pontos de escalonamento:

i- Quando se encontra uma construção **task** → uma tarefa explícita é criada. Veja mais detalhes na Seção 2.3;

ii- Quando se encontra uma construção **taskloop** → tarefas explícitas para partes das iterações do laço são criadas. Essa diretiva oferece controles semelhantes aos encontrados na construção **task**. Veja mais detalhes na Seção 2.6;

iii- Quando se encontra uma construção **target** → uma tarefa *target*, para ser executada em outro dispositivo, é criada. Veja mais detalhes na Seção 2.11.

Além disso, os pontos de escalonamento podem ocorrer implicitamente nos seguintes locais:

- no ponto imediatamente seguinte à geração de uma tarefa explícita com a diretiva **task**;
- após o ponto de conclusão de uma região de tarefa, ao final da diretiva **task**;
- em uma região **taskyield**;
- em uma região **taskwait**;
- no final de uma região **taskgroup**;

- em uma região de barreira implícita ou explícita;
- o ponto imediatamente após a geração de uma região **target**;

2.2.4. Pontos de Sincronização

Em certos pontos do código, que são chamados de pontos de sincronização de tarefas, é garantido que todas as tarefas anteriores já terminaram. Os pontos de sincronização explícitos de tarefas, com uso de diretivas, podem ser:

1. **#pragma omp taskwait**: Esta diretiva é usada para aguardar explicitamente a conclusão de todas as tarefas associadas antes de continuar a execução do código. É uma forma de sincronização explícita para tarefas. Veja mais detalhes na Seção 2.8.
2. **#pragma omp barrier**: Embora seja mais comumente usado para sincronizar *threads* em uma região paralela, pode também ser utilizado para sincronizar tarefas. Este comando espera que todas as *threads* ou tarefas ativas naquele ponto terminem antes de prosseguir. Veja mais detalhes na Seção 2.7.
3. **#pragma omp taskgroup**: É usado para definir um grupo de tarefas, permitindo aguardar a conclusão de todas as tarefas em um determinado grupo. Todas as tarefas dentro do grupo devem ser concluídas antes que a execução do código prossiga após a diretiva **taskgroup**. Veja mais detalhes na Seção 2.9.

No OpenMP, embora haja pontos de sincronização explícitos para garantir a execução das tarefas, há também pontos implícitos nos quais é garantido que as tarefas anteriores já tenham sido executadas. Estes pontos de sincronização implícitos ocorrem, por exemplo, em todas as barreiras implícitas do OpenMP, como as que ocorrem ao final de uma região paralela ou de uma diretiva **single**.

2.3. Diretiva task

2.3.1. Sintaxe

A diretiva **task** é utilizada na criação de tarefas e possui diversas cláusulas que modificam o seu comportamento. A sua sintaxe é a seguinte:

#pragma omp task [cláusula[[,] cláusula] ...]
{bloco-estruturado}

Quando uma *thread* encontra uma construção **task**, uma tarefa é gerada a partir do código associado ao bloco estruturado correspondente. Por bloco estruturado entenda-se um trecho de código delimitado por chaves { } que define uma unidade lógica e coesa de operações, com uma única entrada no topo e uma única saída na parte inferior, sendo que o acesso ao bloco estruturado não pode ser resultado de um desvio e o ponto de saída não pode ser um desvio para fora do bloco estruturado.

A *thread* que encontra a diretiva **task** pode executar imediatamente a tarefa ou adiar sua execução. Neste último caso, qualquer *thread* na equipe pode executar a tarefa. A conclusão da tarefa pode ser garantida com uso de construções de sincronização de tarefas descritas anteriormente na Seção 2.2.4.

2.3.2. Exemplos

No Exemplo 2.1 observamos que a tarefa deve ser criada dentro de uma região paralela, mas com uso da diretiva **single**. Ou seja, apenas uma *thread* deve encontrar a diretiva **task**, caso contrário, múltiplas tarefas seriam criadas, em número igual ao total de *threads* da região paralela. Neste exemplo não há uma ordem pré determinada para a impressão das mensagens.

```
1  int main() {
2      #pragma omp parallel
3      {
4          #pragma omp single
5          {
6              #pragma omp task
7              {
8              printf("Esta é uma tarefa executada por uma thread
   ↪ paralela.\n");
9              }
10             printf("Esta é a execução fora da tarefa.\n");
11         }
12     }
13     return 0;
14 }
```

Exemplo 2.1. Forma de Utilização

```
1  typedef struct node node;
2  struct node {
3      int data;
4      node * next;
5  };
6  void process(node * p) {
7       /* o trabalho é feito aqui */
8  }
9  void increment_list_items(node * head) {
10      #pragma omp parallel
11      {
12          #pragma omp single
13          {
14              node * p = head;
15              while (p) {
16                   #pragma omp task
17                   // p é firstprivate por padrão
18                   process(p);
19                   p = p->next;
20              }
21          }
22      }
23 }
```

Exemplo 2.2. Lista Encadeada

O Exemplo 2.2 [OpenMP ARB, 2016] demonstra como usar a diretiva **task** para processar elementos de uma lista encadeada em paralelo. A *thread* que executa a região única (‘single‘) gera todas as tarefas explícitas, que são então executadas pelas *threads* na equipe atual. O ponteiro ’p’ é **firstprivate** por padrão na diretiva **task**, portanto não é necessário especificá-lo em uma cláusula **firstprivate**.

```
1  int fib(int n) {
2  int i, j;
3      if (n<2)
4          return n;
5      else {
6          #pragma omp task shared(i)
7          i=fib(n-1);
8          #pragma omp task shared(j)
9          j=fib(n-2);
10         #pragma omp taskwait
11         return i+j;
12 }
13 int main() {
14 int n = 20;
15     #pragma omp parallel
16     {
17         #pragma omp single
18         fib(n);
19     }
20 }
```

Exemplo 2.3. Fibonacci

No Exemplo 2.3 a função *fib()* deve ser chamada de dentro de uma região paralela para que as diferentes tarefas especificadas sejam executadas em paralelo. Contudo, apenas uma *thread* da região paralela deve chamar *fib()* a menos que se deseje múltiplos cálculos de Fibonacci concorrentes. A diretiva **taskwait** é obrigatória para a correta execução da função, assim como a declaração das variáveis ’i’ e ’j’ como compartilhadas. Esse código apresenta uma certa ineficiência por gerar tarefas para valores muito pequenos de ’n’. Uma possível solução é o uso das cláusulas **if** ou **final**, que são apresentadas na seção a seguir.

2.3.3. Cláusulas

Tabela 2.1. Cláusulas da diretiva task

if ([task :] expressao-escalar)	**private** (lista)
final (expressao-escalar)	**firstprivate** (lista)
untied	**shared** (lista)
default (shared | none)	**depend** (tipo-de-dependencia : lista)
mergeable	**priority** (valor-da-prioridade)

Várias cláusulas podem ser usadas para gerenciar e otimizar a geração de tarefas, assim como reduzir a sobrecarga de execução e realizar o balanceamento de carga. A Ta-

bela 2.1 apresenta as cláusulas que podem ser utilizadas com a diretiva **task**. As cláusulas **if** e **final** são usadas para controlar a geração e a execução das tarefas.

A cláusula **if** recebe uma expressão booleana que determina se a tarefa deve ser gerada ou não. Se a expressão for **verdadeira**, a tarefa é gerada e colocada no *pool* de tarefas para ser executada por uma das *threads* da equipe. Se a expressão for **falsa**, uma *tarefa não postergável* é gerada e a *thread* que a encontra deve executá-la sequencialmente e apenas ao final retomar a execução da região de tarefa atual.

Se a expressão da cláusula **final** é avaliada como **verdadeira**, a tarefa gerada será uma *tarefa final e incluída*. Neste caso, todas as tarefas que ela gerar serão executadas sequencialmente pela *thread* que a encontrou, sem serem colocadas no *pool*. Todas as construções **task** encontradas durante a execução de uma *tarefa final* gerarão *tarefas finais e incluídas*. Se a tarefa não for final, as tarefas que ela gerar serão tratadas normalmente.

No máximo uma cláusula **if** ou **final** pode aparecer na diretiva **task**. A forma de uso das cláusulas **if** e **final** pode ser vista no Exemplo 2.4. Nesse trecho de código, a função *foo (n)* é criada como uma tarefa se 'n' for maior que 4. Se 'n' for menor que 2, a tarefa é final e todas as tarefas geradas serão executadas sequencialmente. Caso contrário, as tarefas geradas serão colocadas no *pool* para serem executadas por outras *threads*.

```
1 #pragma omp parallel
2 {
3   #pragma omp single
4   {
5     #pragma omp task if (n > 4) final (n < 2)
6     foo (n);
7   }
8 }
```

Exemplo 2.4. Cláusulas if e final

O Exemplo 2.5 ilustra a diferença entre as cláusulas **if** e **final**. A cláusula **if** tem um efeito local. No primeiro conjunto de tarefas, aquela que possui a cláusula **if** será não postergável, mas a tarefa aninhada dentro dessa tarefa não será afetada pela cláusula **if** e será criada normalmente. Por outro lado, a cláusula **final** afeta todas as construções **task** na *região de tarefa final* mas não a própria *tarefa final*. No segundo conjunto de tarefas, as tarefas aninhadas serão criadas como *tarefas incluídas*. Note também que as condições para as cláusulas **if** e **final** são normalmente opostas.

A cláusula **final** pode ser utilizada para otimizar a execução do exemplo do cálculo da série de Fibonacci, evitando a geração de tarefas, quando o valor de 'n' for muito pequeno, como mostrado no Exemplo 2.6. Meça o tempo de execução das duas versões e verifique a diferença.

A expressão da cláusula **if** e a expressão da cláusula **final** são avaliadas no contexto fora da construção **task**, e não é especificada nenhuma ordem para essas avaliações.

```
1  void bar(void);
2
3  void foo () {
4  int i;
5      #pragma omp task if(0) // Esta tarefa é não postergável
6      {
7          #pragma omp task // Esta tarefa é uma tarefa regular
8          for (i = 0; i < 3; i++) {
9               #pragma omp task // Esta tarefa é uma tarefa regular
10              bar();
11         }
12     }
13     #pragma omp task final(1) // Esta tarefa é uma tarefa regular
14     {
15         #pragma omp task // Esta tarefa é incluída
16         for (i = 0; i < 3; i++) {
17              #pragma omp task // Esta tarefa também é incluída
18              bar();
19         }
20     }
21 }
```

Exemplo 2.5. Cláusulas if e final

```
1  int fib(int n) {
2  int i, j;
3      if (n<2)
4          return n;
5      else {
6          #pragma omp task shared(i) final (n<10)
7          i=fib(n-1);
8          #pragma omp task shared(j) final (n<10)
9          j=fib(n-2);
10         #pragma omp taskwait
11         return i+j;
12 }
```

Exemplo 2.6. Fibonacci otimizado

Uma *thread* que encontra um ponto de escalonamento de tarefas dentro da região de tarefa pode suspender temporariamente a execução dessa região de tarefa. Por padrão, uma tarefa é vinculada e sua região de tarefa suspensa só pode ser retomada pela *thread* que iniciou sua execução. Contudo, se a cláusula **untied** estiver presente em uma construção **task**, qualquer *thread* na equipe pode retomar a execução da região de tarefa após uma suspensão. A cláusula **untied** é ignorada se uma cláusula **final** estiver presente na mesma construção **task** e a expressão da cláusula **final** for avaliada como verdadeira, ou se a tarefa for uma *tarefa incluída*.

Reforçando o que já dissemos, a construção **task** inclui um ponto de escalonamento de tarefas imediatamente após a geração da tarefa explícita e outro também no seu ponto de conclusão.

```
1  #define LARGE_NUMBER 10000000
2  double item[LARGE_NUMBER];
3  extern void process(double);
4
5  int main() {
6      #pragma omp parallel
7      {
8          #pragma omp single
9          {
10             int i;
11             #pragma omp task untied
12             // i é firstprivate, item é shared
13             {
14                  for (i=0; i<LARGE_NUMBER; i++)
15                       #pragma omp task
16                       process(item[i]);
17                  }
18             }
19     }
20     return 0;
21 }
```

Exemplo 2.7. Cláusula untied

No Exemplo 2.7, as tarefas são geradas em um laço executado por uma *tarefa não vinculada* ("untied task"). Durante essa geração, o limite de tarefas não atribuídas do ambiente de execução pode ser atingido. Nesse caso, a *thread* responsável pelo laço de geração de tarefas, ao alcançar o ponto de escalonamento de tarefas na diretiva **task**, pode suspender a tarefa atual e iniciar a execução de uma das tarefas ainda não atribuídas.

Se as outras *threads* finalizarem as demais tarefas antes que a *thread* responsável pelo laço termine a execução da sua tarefa, qualquer outra *thread* está qualificada para retomar a execução do laço de geração de tarefas, já ele está sendo executado por uma *tarefa não vinculada*.

Sem o uso desta cláusula, as outras *threads* seriam forçadas a ficar inativas até que a *thread* geradora concluísse sua tarefa mais demorada, já que o laço de geração de tarefas estaria sendo executado por uma *tarefa vinculada*.

Quando a cláusula **mergeable** está presente em uma construção **task**, a tarefa gerada é uma *tarefa mesclável*, ou seja, uma tarefa para a qual o ambiente de dados, incluindo as variáveis de controle internas (veja a Seção 2.12), é o mesmo que o de sua região de tarefa geradora. Isso significa que o compilador pode decidir unir duas ou mais tarefas em uma única tarefa se elas forem executadas na mesma *thread*, com o objetivo de melhorar a eficiência e reduzir a sobrecarga de execução de tarefas pequenas.

Ao adicionar a cláusula **mergeable** a uma diretiva **task**, você permite que o ambiente de execução faça essa otimização, caso julgue apropriado. Isso pode economizar os custos associados à criação, escalonamento e execução de tarefas individuais separadas, como em situações onde você tem muitas tarefas pequenas ou quando a fusão dessas tarefas pode resultar em uma redução da sobrecarga de gerenciamento de tarefas.

```
1  #include <stdio.h>
2  void foo ( ) {
3      int x = 2;
4      #pragma omp task shared(x) mergeable
5      {
6          x++;
7      }
8      #pragma omp taskwait
9      printf("%d\n",x); // imprime 3
10 }
```

Exemplo 2.8. Cláusula mergeable

No Exemplo 2.8 o uso da cláusula **mergeable** é seguro. Como 'x' é uma variável compartilhada, o resultado não depende se a tarefa é mesclada ou não (ou seja, a tarefa sempre incrementará a mesma variável e sempre calculará o mesmo valor para 'x').

```
1  #include <stdio.h>
2  void foo () {
3  int x = 2;
4      #pragma omp task mergeable
5      {
6      x++;
7      }
8      #pragma omp taskwait
9      printf("%d\n",x); // imprime 2 ou 3
10 }
```

Exemplo 2.9. Cláusula mergeable

O Exemplo 2.9 [OpenMP ARB, 2016] demonstra um uso incorreto da cláusula **mergeable**. Neste exemplo, a tarefa criada acessará diferentes instâncias da variável 'x' se a tarefa não for mesclada, já que 'x' é **firstprivate**. Porém, acessará a mesma variável 'x' se a tarefa for mesclada. Como resultado, o comportamento do programa é indefinido e ele pode imprimir dois valores diferentes para 'x' dependendo das decisões tomadas pelo ambiente de execução.

A cláusula **priority** é uma sugestão para o escalonador de prioridade da tarefa gerada, através de um valor numérico não negativo. As tarefas com valor numérico mais alto tem recomendação para serem executadas antes das tarefas de prioridade mais baixa. A prioridade padrão, quando nenhuma cláusula **priority** é especificada, é zero (a menor prioridade).

Se um valor for especificado na cláusula **priority** for maior do que o valor máximo definido (ICV *max-task-priority-var*), então o ambiente de execução usará o valor dessa ICV. Um programa que dependa da ordem de execução das tarefas determinada por esse valor de prioridade pode ter um comportamento indeterminado, já que a prioridade é apenas uma sugestão e não é obrigatória.

```
1  void compute_array (float *node, int M);
2  void compute_matrix (float *array, int N, int M) {
3      int i;
4      #pragma omp parallel private(i)
5           #pragma omp single
6           {
7                for (i=0;i<N; i++) {
8                     #pragma omp task priority(i)
9                     compute_array(&array[i*M], M);
10               }
11          }
12 }
```

Exemplo 2.10. Cláusula priority

No Exemplo 2.10 calculamos *arrays* em uma matriz por meio de uma rotina 'compute_array'. Cada tarefa tem um valor de prioridade igual ao valor da variável de laço 'i' no momento de sua criação. Uma prioridade maior em uma tarefa significa que ela é candidata a ser executada mais cedo. A criação de tarefas ocorre em ordem crescente (de acordo com o espaço de iteração do laço), mas uma sugestão, por meio da cláusula **priority**, é fornecida para reverter a ordem de execução.

No máximo uma cláusula **priority** pode aparecer na diretiva **task**. Um programa que se desvia para dentro ou para fora de uma região de tarefa viola as especificações. Um programa não deve depender de qualquer ordenação das avaliações das cláusulas da diretiva **task**, nem de quaisquer efeitos colaterais das avaliações das cláusulas.

As demais cláusulas **default**, **private**, **firstprivate** e **shared**, são utilizadas para determinar o escopo das variáveis e têm o seu comportamento idêntico ao utilizado nas demais diretivas do OpenMP (veja mais detalhes em [Silva et al., 2022]).

A execução ordenada das tarefas pode ser realizada especificando as dependências com uma cláusula **depend**. Vamos estudar a cláusula **depend** com mais detalhes na Seção 2.5.

2.4. Escopo das variáveis

No OpenMP, a diretiva **task** é usada para criar tarefas independentes que podem ser aninhadas, criando uma hierarquia de tarefas. Mas uma complicação adicional que surge é determinar o escopo das variáveis utilizadas nesse conjunto de tarefas, que podem ter o seguinte comportamento:

1. Variáveis privadas: Normalmente, cada tarefa possui cópias privadas de suas variáveis locais. Ou seja, as variáveis declaradas dentro de uma tarefa não são acessíveis por outras tarefas.

2. Variáveis compartilhadas: Variáveis definidas fora do escopo das tarefas, como variáveis globais ou variáveis declaradas antes da região de tarefa, são compartilhadas por todas as tarefas criadas. Isso significa que todas as tarefas podem acessar e modificar essas variáveis.

3. Cláusulas de compartilhamento de dados: O OpenMP oferece cláusulas como **shared**, **private**, **firstprivate**, **lastprivate**, entre outras, que permitem controlar o escopo e o compartilhamento de variáveis entre as tarefas. Por exemplo, a cláusula **private** pode ser usada para garantir que cada tarefa tenha sua própria cópia privada de uma variável, enquanto a cláusula **shared** permite que todas as tarefas acessem e compartilhem uma variável comum.

4. Escopo das variáveis dentro de tarefas aninhadas: Se uma tarefa cria outras tarefas (tarefas aninhadas), as variáveis do escopo da tarefa pai são compartilhadas com as tarefas filhas, a menos que sejam explicitamente definidas como privadas.

Mas não é só isso. Os atributos de compartilhamento de dados de variáveis referenciadas em uma construção OpenMP podem ser *predeterminados*, *explicitamente determinados* ou *implicitamente determinados*, de acordo com as regras descritas a seguir.

Certas variáveis e objetos, relativos às tarefas, possuem atributos de compartilhamento de dados *predeterminados* da seguinte forma:

- Variáveis presentes em diretivas **threadprivate** são *threadprivate*.

```
1          int A[SIZE];
2          #pragma omp threadprivate(A)
3          // ...
4           #pragma omp task
5           {
6                 // A: threadprivate
7           }
8
```

- Variáveis com duração de armazenamento automático declaradas em um escopo dentro da construção são privadas.

```
1         #pragma omp task
2         {
3             int x = MN;
4             // Escopo de x: privado
5          }
6
```

- Objetos com duração de armazenamento dinâmico (malloc, new, etc.) são compartilhados.

```
1         int *p;
2         p = malloc(sizeof(float)*SIZE);
3         #pragma omp task
4         {
5              // *p: compartilhado
6         }
7
```

- Membros de dados estáticos são compartilhados.

```
1         class foo {
2               static int s = MN;
3           };
4         #pragma omp task
5         {
6               foo_A.s; // s@foo: compartilhado
7         }
8
```

- Variáveis com duração de armazenamento estático declaradas em um escopo dentro da construção são compartilhadas.

```
1         #pragma omp task
2         {
3               static int y;
4               // Escopo de y: compartilhado
5         }
6
```

- A(s) variável(eis) de iteração do laço nos laços for associados a uma construção **for**, **parallel for**, **taskloop** ou **distribute** é(são) privada(s).

Note que as variáveis com atributos de compartilhamento de dados *predeterminados* não podem ser listadas em cláusulas de atributos de compartilhamento de dados, exceto nos casos listados abaixo, o que substitui os atributos de compartilhamento de dados predeterminados da variável. Listamos aqui apenas o caso que se aplica às tarefas.

- A(s) variável(eis) de iteração do laço nos laços **for** associados a uma construção **for**, **parallel for**, **taskloop** ou **distribute** podem ser listadas em uma cláusula **private** ou **lastprivate**.

As variáveis com atributos de compartilhamento de dados *explicitamente determinados* são aquelas que são referenciadas em uma determinada construção e que estão listadas em uma cláusula de atributo de compartilhamento de dados (**private**, **shared**, **firstprivate**, etc.) naquela construção.

As variáveis com atributos de compartilhamento de dados *implicitamente determinados* são aquelas referenciadas dentro de uma construção específica, mas que não possuem atributos de compartilhamento de dados *predefinidos* e não estão *explicitamente* listadas em uma cláusula que defina tais atributos naquela construção. As suas regras são as seguintes:

- Em uma construção **parallel**, **teams** ou de geração de tarefas, os atributos de compartilhamento de dados dessas variáveis são determinados pela cláusula **default**. Se nenhuma cláusula **default** estiver presente, essas variáveis são compartilhadas.

- Para construções que não sejam construções de geração de tarefas ou construções **target**, se nenhuma cláusula **default** estiver presente, essas variáveis referenciam as variáveis com os mesmos nomes que existem no contexto envolvente.
- Em uma construção **target**, variáveis que não são mapeadas após a aplicação das regras de atributo de mapeamento de dados são **firstprivate**.
- Em uma construção de geração de tarefas órfãs, se nenhuma cláusula **default** estiver presente, os argumentos formais passados por referência são tratados como **firstprivate**.
- Em uma construção de geração de tarefas, se nenhuma cláusula **default** estiver presente, uma variável para a qual o atributo de compartilhamento de dados não é determinado pelas regras mencionadas anteriormente e que, no contexto envolvente, é determinada como compartilhada por todas as tarefas implícitas vinculadas à equipe atual, é compartilhada.
- Em uma construção de geração de tarefas, se nenhuma cláusula **default** estiver presente, uma variável para a qual o atributo de compartilhamento de dados não é determinado pelas regras mencionadas anteriormente é tratada como **firstprivate**.
- Um item da lista que aparece em uma cláusula **reduction** de uma construção de compartilhamento de trabalho não deve aparecer em uma cláusula **firstprivate** em uma construção de tarefa que seja encontrada durante a execução de qualquer uma das regiões de compartilhamento de trabalho resultantes.

2.5. Cláusula depend

A cláusula **depend** impõe restrições adicionais no escalonamento de tarefas ou iterações de laços. Essas restrições estabelecem dependências apenas entre tarefas irmãs ou entre as iterações de um laço. A sua sintaxe é a seguinte:

depend(tipo-de-dependencia : lista)

onde *tipo-de-dependencia* é um dos seguintes[1]: *in*, *out* ou *inout*.

Para o tipo de dependência *in*, se o local de armazenamento de pelo menos um dos itens da lista for o mesmo que o local de armazenamento de um item da lista de dependência *out* ou *inout* de uma construção de tarefa a partir da qual uma tarefa irmã foi gerada anteriormente, então a tarefa gerada será uma tarefa dependente dessa tarefa irmã.

Para os tipos de dependência *out* e *inout*, se o local de armazenamento de pelo menos um dos itens da lista for o mesmo que o local de armazenamento de um item da lista de dependência *in*, *out* ou *inout* de uma construção de tarefa a partir da qual uma tarefa irmã foi gerada anteriormente, então a tarefa gerada será uma tarefa dependente dessa tarefa irmã. Algumas das restrições da cláusula **depend** são:

- Itens da lista usados em cláusulas **depend** da mesma tarefa ou tarefas irmãs devem indicar locais de armazenamento idênticos ou locais de armazenamento disjuntos.

[1]Nota: Por simplificação não abordamos neste texto as dependências do tipo *source* ou *sink*.

- Itens da lista usados em cláusulas **depend** não podem ser seções de array de comprimento zero.
- Uma variável que faz parte de outra variável (como um elemento de uma estrutura), mas que não é um elemento de *array* ou uma seção de *array*, não pode aparecer em uma cláusula **depend**.

```
1  #include <stdio.h>
2  int main()  {
3      int x = 1;
4      #pragma omp parallel
5          #pragma omp single
6          {
7              #pragma omp task shared(x) depend(out: x)
8              x = 2;
9              #pragma omp task shared(x) depend(in: x)
10             printf("x = %d\n", x);
11         }
12     return 0;
13 }
```

Exemplo 2.11. Dependencia entre tarefas

2.6. Diretiva taskloop

A diretiva **taskloop** especifica que as iterações de um ou mais laços associados serão executadas em paralelo usando tarefas do OpenMP. As iterações são distribuídas entre as tarefas criadas pela construção e escalonadas para execução.

#pragma omp taskloop [cláusula[[,] cláusula] ...] new-line
for-loops

A diretiva **taskloop** impõe restrições na estrutura de todos os *for-loops* associados, que devem ter uma forma de laço canônica. A ordem de criação das tarefas do laço é indefinida, logo os programas que dependam de qualquer ordem de execução das iterações lógicas do laço violam as especificações.

Tabela 2.2. Cláusulas da diretiva taskloop

if ([taskloop :] expressao-escalar)	**private** (lista)
final (expressao-escalar)	**firstprivate** (lista)
untied	**shared** (lista)
default (shared | none)	**depend** (tipo-de-dependencia : lista)
mergeable	**priority** (valor-da-prioridade)
lastprivate(lista)	**collapse**(n)
grainsize(grain-size)	**num_tasks**(num-tasks)
nogroup	

O parâmetro da cláusula **grainsize** deve ser uma expressão de número inteiro positivo, sendo que o número de iterações lógicas do laço atribuídas a cada tarefa criada é

maior ou igual ao mínimo entre o valor da expressão de *grain-size* e o número de iterações lógicas do laço, porém menor que **duas vezes** o valor da expressão de *grain-size*.

Se a cláusula **num_tasks** for especificada, a construção **taskloop** criará tantas tarefas quanto o mínimo entre a expressão *num-tasks* e o número de iterações lógicas do laço. Cada tarefa deve ter pelo menos uma iteração lógica do laço. O parâmetro da cláusula **num_tasks** deve ter como resultado um número inteiro positivo.

Se nenhuma cláusula **grainsize** ou **num_tasks** estiver presente, o número de tarefas de laço criadas e o número de iterações lógicas do laço atribuídas a essas tarefas são definidos pela implementação.

A cláusula **collapse** pode ser usada para especificar quantos laços estão associados à construção **taskloop**. O parâmetro da cláusula **collapse** deve ser uma expressão que resulte em um número inteiro positivo constante. Se nenhuma cláusula **collapse** estiver presente, o único laço associado à construção **taskloop** é aquele que imediatamente segue a diretiva **taskloop**. Se mais de um laço estiver associado à construção **taskloop**, então as iterações de todos os laços associados são colapsadas em um espaço de iteração maior que é então dividido de acordo com as cláusulas **grainsize** e **num_tasks**. A execução sequencial das iterações em todos os laços associados determina a ordem das iterações no espaço de iteração colapsado.

O contador de iteração para cada laço associado é calculado antes da entrada para o laço mais externo. Se a execução de qualquer laço associado alterar quaisquer dos valores usados para calcular qualquer um dos contadores de iteração, então o comportamento é indefinido.

As cláusulas **if**, **final**, **priority**, **untied**, **depend**, **mergeable** quando presentes em uma construção **taskloop** tem comportamento similar ao descrito para a construção **task** (veja a Seção 2.3).

As demais cláusulas **default**, **private**, **firstprivate**, **lastprivate**, **shared**, são utilizadas para determinar o escopo das variáveis e têm o seu comportamento idêntico ao utilizado nas demais diretivas do OpenMP (veja mais detalhes em [Silva et al., 2022]).

Por padrão, a construção **taskloop** executa como se estivesse contida em uma construção **taskgroup** sem nenhuma declaração ou diretiva fora da construção **taskloop**. Assim, a construção **taskloop** cria uma região **taskgroup** implícita. Se a cláusula **nogroup** estiver presente, nenhuma região de **taskgroup** implícita será criada. As restrições da construção **taskloop** são as seguintes:

- Um programa que entra ou sai de uma região **taskloop** está em desacordo com as especificações.
- Todos os laços associados à construção **taskloop** devem ser perfeitamente aninhados; ou seja, não deve haver código intermediário nem nenhuma diretiva OpenMP entre quaisquer dois laços.
- No máximo uma cláusula **grainsize** ou uma cláusula **num_tasks** pode aparecer em uma diretiva **taskloop**, que são mutuamente exclusivas e não podem aparecer na mesma diretiva **taskloop**.

```
1  void long_running_task(void);
2  void loop_body(int i, int j);
3  void parallel_work(void) {
4  int i, j;
5    #pragma omp taskgroup
6    {
7      #pragma omp task
8      long_running_task(); // pode executar em paralelo
9      #pragma omp taskloop private(j) grainsize(500) nogroup
10     for (i = 0; i < 10000; i++) { // pode executar em paralelo
11         for (j = 0; j < i; j++) {
12                 loop_body(i, j);
13         }
14     }
15   }
16 }
```

Exemplo 2.12. Diretiva taskloop

O Exemplo 2.12 [OpenMP ARB, 2016] mostra como executar uma tarefa de longa duração simultaneamente com tarefas criadas com uma diretiva **taskloop** para um laço com quantidades desequilibradas de trabalho para suas iterações. A cláusula **grainsize** especifica que cada tarefa deve executar pelo menos 500 iterações do laço. A cláusula **nogroup** remove o grupo de tarefas implícito da construção **taskloop**; a construção explícita **taskgroup** no exemplo garante que a função não seja encerrada antes que a tarefa de longa duração e os laços tenham concluído a sua execução.

2.7. Diretiva barrier

A construção **barrier** é uma diretiva independente que especifica uma barreira explícita no ponto em que a construção aparece. A sua sintaxe é a seguinte:

#pragma omp barrier new-line

Todas as *threads* da equipe que executam a região paralela vinculada devem executar a região de barreira e completar a execução de todas as tarefas explícitas vinculadas a esta região paralela antes que qualquer uma delas continue a sua execução além da barreira. A região de barreira inclui um ponto implícito de escalonamento de tarefa na região de tarefa atual. As seguintes restrições se aplicam à construção **barrier**:

- Cada região de barreira deve ser encontrada por todas as *threads* em uma equipe ou por nenhuma delas, a menos que o cancelamento tenha sido solicitado para a região paralela mais interna que a envolve.
- A sequência de regiões de compartilhamento de trabalho e regiões de barreira encontradas deve ser a mesma para todas as *threads* em uma equipe.
- As diretivas **flush**, **barrier**, **taskwait** e **taskyield** são diretivas independentes e não podem ser a subdeclaração imediata de uma instrução **if**.

2.8. Diretiva taskwait

A construção **taskwait** é uma diretiva independente que especifica uma espera pela conclusão das tarefas filhas (mas que não inclui suas descendentes) da tarefa atual, cuja sintaxe da construção é a seguinte:

#pragma omp taskwait

A região **taskwait** inclui um ponto implícito de escalonamento de tarefas na região da tarefa atual, que é suspensa até que todas as tarefas filhas que foram geradas antes da região **taskwait** completem sua execução. Embora pareça simples, alguns detalhes importantes devem ser observados quando do uso de tarefas, que o uso da diretiva **taskwait** pode ajudar.

```
1  struct node {
2  struct node *left;
3  struct node *right;
4  };
5  extern void process(struct node *);
6  void postorder_traverse( struct node *p ) {
7      if (p->left)
8          #pragma omp task // p é firstprivate por padrão
9          postorder_traverse(p->left);
10     if (p->right)
11         #pragma omp task // p é firstprivate por padrão
12         postorder_traverse(p->right);
13     #pragma omp taskwait
14     process(p);
15 }
```

Exemplo 2.13. Diretiva taskwait

O Exemplo 2.13 [OpenMP ARB, 2016] mostra como percorrer uma estrutura semelhante a uma árvore usando tarefas explícitas. Observe que a função **postorder_traverse** deve ser chamada de dentro de uma região paralela para que as tarefas especificadas sejam executadas em paralelo.

Espera-se que a travessia seja feita em pós-ordem, como no código sequencial. Entenda-se por pós-ordem como um tipo de travessia ou ordenação dos elementos em que os nós filhos ou subordinados são visitados (processados) antes de se visitar o próprio nó raiz. É uma abordagem para percorrer a estrutura de forma recursiva em que os nós inferiores são visitados sempre antes dos nós superiores.

Também observe que, sem uso da diretiva **taskwait**, as tarefas serão executadas sem uma ordem especificada. Neste exemplo, forçamos uma travessia em pós-ordem da árvore adicionando uma diretiva **taskwait**, que garante que os filhos esquerdo e direito foram executados antes de processarmos o nó atual.

2.9. Diretiva taskgroup

Um **conjunto taskgroup** é um conjunto de tarefas que são logicamente agrupadas por uma região **taskgroup**. A construção **taskgroup** especifica uma espera pela conclusão

das tarefas filhas da tarefa atual e *todas* as suas tarefas descendentes. A sua sintaxe é:

#pragma omp taskgroup new-line
bloco-estruturado

Quando uma *thread* encontra uma construção **taskgroup**, ela começa a executar a região. Todas as tarefas filhas geradas na região **taskgroup** e todas as suas descendentes que se vinculam à mesma região paralela que a região **taskgroup** fazem parte do conjunto de tarefas associado à região **taskgroup**.

Há um ponto implícito de escalonamento de tarefa no final da região **taskgroup**. A tarefa atual é suspensa no ponto de escalonamento de tarefa até que todas as tarefas no **conjunto taskgroup** concluam a sua execução.

```
1  int main() {
2     int i;
3     tree_type tree;
4     init_tree(tree);
5     #pragma omp parallel
6     #pragma omp single
7     {
8         #pragma omp task
9         start_background_work();
10        for (i = 0; i < max_steps; i++) {
11             #pragma omp taskgroup
12             {
13                 #pragma omp task
14                 compute_tree(tree);
15              } // espera a travessia da árvore neste passo
16        check_step();
17        }
18   } // apenas aqui espera-se que a tarefa em
19     // background esteja completada
20  print_results();
21  return 0;
22  }
```

Exemplo 2.14. Diretiva taskgroup

No Exemplo 2.14 [OpenMP ARB, 2016], as tarefas são agrupadas e sincronizadas usando a construção **taskgroup**. Inicialmente, uma tarefa (a tarefa que executa a chamada *start_background_work()* é criada na região paralela e, mais tarde, uma travessia de árvore paralela é iniciada (a tarefa que executa a raiz das chamadas recursivas *compute_tree()*). Ao sincronizar as tarefas no final de cada travessia da árvore, o uso da construção **taskgroup** garante que a tarefa em segundo plano anteriormente iniciada não participe da sincronização e permaneça livre para ser executada em paralelo. Isso é oposto ao comportamento da construção **taskwait**, que incluiria as tarefas em segundo plano na sincronização.

2.10. Diretiva taskyield

A diretiva **taskyield** especifica que a tarefa atual pode ser suspensa em favor da execução de alguma outra tarefa. A sua sintaxe é a seguinte:

#pragma omp taskyield new-line

A região **taskyield** inclui um ponto explícito de escalonamento de tarefa na região de tarefa atual.

O Exemplo 2.15 [OpenMP ARB, 2016] ilustra o uso da diretiva **taskyield**. As tarefas no exemplo calculam algo útil e, em seguida, realizam algumas computações que precisam ser feitas em uma região crítica. Ao utilizar **taskyield** quando uma tarefa não consegue obter acesso à região crítica, a implementação pode suspender a tarefa atual e escalonar alguma outra tarefa que possa realizar alguma operação útil.

```
1  void algumacoisa_util (void);
2  void algumacoisa_critica (void);
3  void foo (omp_lock_t * lock, int n)
4  {
5      int i;
6      for ( i = 0; i < n; i++ )
7      #pragma omp task
8      {
9          algumacoisa_util();
10         while ( !omp_test_lock(lock) ) {
11         #pragma omp taskyield
12         }
13     algumacoisa_critica();
14     omp_unset_lock(lock);
15     }
16 }
```

Exemplo 2.15. Diretiva taskyield

2.11. Diretiva target

O OpenMP pode direcionar a execução de certos trechos de código para aceleradores (como GPUs) ou outros dispositivos que estejam conectados ao computador hospedeiro. Isso pode ser realizado com o uso da diretiva **target**.

Uma tarefa **target** é uma *tarefa mesclável* que é gerada por uma construção **target**, **target enter data**, **target exit data** ou **target update**. A construção **target** consiste em uma diretiva de **target** e uma região de execução. A região *target* é executada no dispositivo padrão ou no dispositivo especificado na cláusula **device**, normalmente um acelerador.

Infelizmente não foi possível apresentar maiores detalhes dessas diretivas neste minicurso e apresentamos apenas o Exemplo 2.16 onde a diretiva **target** cria um ambiente de execução para a execução paralela do laço em um dispositivo de destino (como uma GPU), distribuindo as iterações do laço entre as *threads* da equipe.

```
1  int main() {
2      const int array_size = 100;
3      int array[array_size];
4      #pragma omp parallel for
5      for (int i = 0; i < array_size; ++i) {
6          array[i] = i;
7      }
8      #pragma omp target
9      #pragma omp teams distribute parallel for
10     for (int i = 0; i < array_size; ++i) {
11         printf("Thread %d: Valor %d do array.\n",\
12         omp_get_thread_num(), array[i]);
13     }
14     return 0;
15 }
```

Exemplo 2.16. Diretiva target

2.12. Variáveis Internas de Controle

As ICVs armazenam informações como o número de *threads* a serem utilizadas para futuras regiões paralelas, o escalonamento a ser utilizado para laços de compartilhamento de trabalho e se o paralelismo aninhado está habilitado ou não.

Elas recebem os valores iniciais do próprio ambiente de execução, seja a partir de variáveis de ambiente do OpenMP ou através de chamadas para rotinas da API do OpenMP. O programa só pode recuperar os valores dessas ICVs através das rotinas da API do OpenMP. Existe uma correspondência entre as ICVs e as variáveis de ambiente que pode ser vista na Tabela 2.3.

Tabela 2.3. ICV e Variáveis de Ambiente

ICV	Variável de Ambiente
dyn-var	OMP_DYNAMIC
nest-var	OMP_NESTED
nthreads-var	OMP_NUM_THREADS
run-sched-var	OMP_SCHEDULE
bind-var	OMP_PROC_BIND
stacksize-var	OMP_STACKSIZE
wait-policy-var	OMP_WAIT_POLICY
thread-limit-var	OMP_THREAD_LIMIT
max-active-levels-var	OMP_MAX_ACTIVE_LEVELS
place-partition-var	OMP_PLACES
cancel-var	OMP_CANCELLATION
default-device-var	OMP_DEFAULT_DEVICE
max-task-priority-var	OMP_MAX_TASK_PRIORITY

2.13. Estudos de caso - Quicksort

Uma exemplo prático da aplicação das técnicas de paralelismo de tarefas é no algoritmo de ordenação Quicksort. Ele pode ser descrito facilmente como uma operação recursiva da seguinte maneira [Kumar, 2002][van der Pas et al., 2017]:

1. Escolha um elemento do arranjo de dados, chamando-o de *pivô*.
2. Coloque o *pivô* em sua posição final e ordenado em relação aos demais elemento do arranjo, mantendo os menores elementos à esquerda dele, bem como os maiores elementos à direita. Essa fase é chamada de *particionamento*.
3. *Recursivamente*, faça os passos 1 e 2 para as partes esquerda e direita do arranjo.
4. A *recursão* termina quando não houver nenhum outro elemento a ser ordenado.

Uma forma simplificada de visualizar a recursão existente na execução do algoritmo Quicksort é por meio de uma árvore. Uma forma de visualizar esta árvore é apresentada na Figura 2.2: cada nível da árvore representa uma divisão do arranjo em uma chamada recursiva; em cada nível da árvore estão circulados os *pivôs* em cor vermelha; a altura da árvore está anotado com a letra h, sendo o *nó raiz* indicado por $h = 0$.

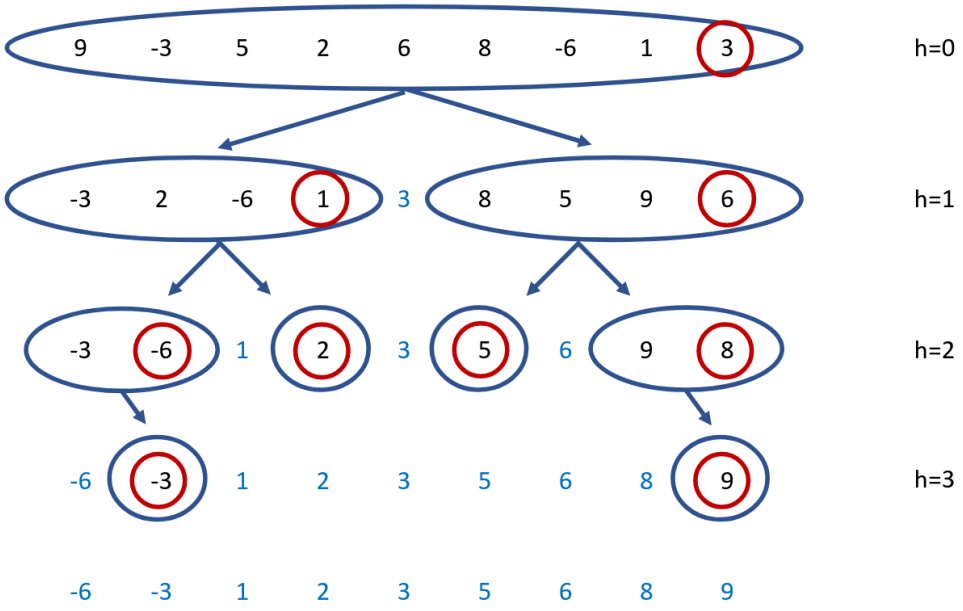

Figura 2.2. Árvore de recursão do Quicksort.

Sabe-se que a escolha do *pivô* é fundamental para o comportamento estável do Quicksort. Considerando uma distribuição ideal do arranjo, pode-se dizer que em cada chamada recursiva, o arranjo é dividido ao meio. Uma solução clássica do algoritmo recursivo é apresentada no Exemplo 2.17, onde cada divisão do arranjo é dividido em aproximadamente $(baixo + alto)/2$ elementos [van der Pas et al., 2017][Beukman, 2021].

```
1  // Função Quicksort simples
2  void quicksort(int arr[], int baixo, int alto) {
3    if (baixo < alto) {
4      // Particionar o array e obter o índice do pivô
5      int pi = particionar(arr, baixo, alto);
6
7      // Executar uma chamada recursiva para cada metade do array
8      quicksort(arr, baixo, pi - 1);
9      quicksort(arr, pi + 1, alto);
10   }
11 }
```

Exemplo 2.17. Código exemplo do algoritmo de ordenação Quicksort

Uma solução paralela trivial desse algoritmo seria anotar cada chamada recursiva do algoritmo com uma diretiva **omp task**. Assim, cada ramo da árvore de recursão seria executada por uma tarefa diferente, permitindo uma independência entre cada uma delas. O Exemplo 2.18 apresenta a estrutura dessa solução.

Veja que na função *main()* foi adicionada a região paralela (**omp parallel**) e, dentro dela, uma diretiva **single**. Ela garante que somente uma thread execute o bloc se código seguinte que, neste caso, é a primeira chamada para a função *quicksort_paralelo()*. Também foi usada uma cláusula **nowait** para que não haja barreira implícita nesta cláusula (veja mais detalhes sobre essa construção em [Silva et al., 2022].

A função *quicksort_paralelo()* praticamente mantém sua estrutura original - neste caso, optou-se por trocar o nome da função apenas para identificá-la mais facilmente. nas chamadas recursivas foi adicionada a diretiva **omp task**. É importante lembrar sobre o controle das variáveis utilizadas. Uma boa prática é alterar o comportamento padrão de **shared** para **none** e identificar as variáveis utilizadas individualmente. Neste exemplo, foram utilizadas as cláusulas **firstprivate** para as variáveis de controle de índice e **shared** para o compartilhamento do ponteiro do arranjo de dados.

Esta solução funciona bem, pois existe a criação das tarefas e suas execuções são independentes, e que cada uma delas ordena sua parte do arranjo. Porém, para um arranjo suficientemente grande, ela pode apresentar uma sobrecarga na criação de tarefas.

Observe, por exemplo, o nível $h = 2$ de altura da árvore na Figura 2.2. Nela, são criadas mais quatro tarefas devido as chamadas recursivas existente no algoritmo, além das tarefas que já foram criadas até o nível anterior (com $h = 1$ foram criadas duas tarefas, e com $h = 0$, foi criada uma tarefa). Podemos perceber que a quantidade de tarefas criadas e enfileiradas poderá ser suficientemente grande devido ao resultado de todas as chamadas recursivas, para todos os níveis da árvore gerada, derivada do número de elementos no arranjo. Por isso, esta solução pode sofrer uma degradação de desempenho pela sobrecarga de tarefas criadas e o seu gerenciamento.

A solução adotada para resolver o problema de sobrecarga é apresentado no Exemplo 2.19. Nele, é introduzido um valor denominado *CORTE*, que é resultado de uma análise experimental sobre o tamanho mínimo que um arranjo deve ter para que, a partir desse ponto, seja optado pela chamada recursiva da função sem a criação de novas tarefas.

```
1  // Função Quicksort paralela usando OpenMP
2  void quicksort_paralelo(int arr[], int baixo, int alto) {
3    if (baixo < alto) {
4      // Particionar o array e obter o índice do pivô
5      int pi = particionar(arr, baixo, alto);
6
7      // Executar a chamada recursiva para cada metade do array em
       ↪ paralelo
8      #pragma omp task default(none) shared (arr) firstprivate(baixo
       ↪ , pi)
9      quicksort_paralelo(arr, baixo, pi - 1);
10     #pragma omp task default(none) shared (arr) firstprivate(alto,
       ↪  pi)
11     quicksort_paralelo(arr, pi + 1, alto);
12   }
13 }
14
15 // Função principal
16 int main()
17 {
18   // Seed para a função rand
19   srand(time(NULL));
20
21   int tamanho = TAMANHO; // Tamanho do vetor
22
23   // Alocar memória dinamicamente para o vetor
24   int *arr = (int *)malloc(tamanho * sizeof(int));
25
26   // Preencher o vetor com valores aleatórios
27   for (int i = 0; i < tamanho; i++)   {
28     arr[i] = rand() % tamanho; // Gera valores aleatórios entre 0
       ↪ e tamanho
29   }
30
31   printf("Array original (%d): \n", tamanho);
32   imprimir_array(arr);
33
34   // Iniciar região paralela
35   #pragma omp parallel default(none) shared(arr, tamanho)
36   {
37     // Executar Quicksort paralelo
38     #pragma omp single nowait
39     quicksort_paralelo(arr, 0, tamanho - 1);
40   }
41
42   printf("\nArray ordenado (%d): \n", tamanho);
43   imprimir_array(arr);
44   if(validar_array_ordenado(arr, tamanho)) {
45     fprintf(stderr, "ERRO ao ordenar\n");
46   }
47   return 0;
48 }
```

Exemplo 2.18. Código exemplo trivial do algoritmo de ordenação Quicksort com *tasks*

```
1  // Função Quicksort paralela usando OpenMP
2  void quicksort_paralelo(int arr[], int baixo, int alto) {
3    if (baixo < alto) {
4      // Particionar o array e obter o índice do pivô
5      int pi = particionar(arr, baixo, alto);
6
7      // Executar a chamada recursiva para cada metade do array em
       ↪ paralelo
8      #pragma omp task final((pi - baixo + 1) < CORTE) mergeable
       ↪ default(none) shared(arr) firstprivate(baixo, pi)
9      quicksort_paralelo(arr, baixo, pi - 1);
10     #pragma omp task final((alto - pi + 1) < CORTE) mergeable
       ↪ default(none) shared(arr) firstprivate(alto, pi)
11     quicksort_paralelo(arr, pi + 1, alto);
12   }
13 }
```

Exemplo 2.19. Código exemplo com CORTE do algoritmo Quicksort com *tasks* paralelas

Nesta nova versão, ajustada para resolver o problema da sobrecarga, usamos a cláusula **final** nas criação das tarefas (revise o assunto na Seção 2.3.3) que avalia o tamanho do arranjo nas chamadas recursivas. Se esse tamanho for pequeno o suficiente ($< CORTE$), deseja-se não criar uma nova tarefas, mas forçar a *thread* vigente executar sequencialmente a chamada recursiva até finalizar toda a árvore de recursão desse ramo. Essa estratégia é utilizada nos dois ramos da recursão do algoritmo do Quicksort.

Por outro lado, se o tamanho do arranjo ainda for grande o suficiente para a criação de uma nova tarefa, permitimos que o OpenMP crie a tarefa e coloque-a no *pool* de execuções.

Por fim, além das considerações tradicionais sobre o algoritmo de Quicksort, como a escolha e localização do pivô, é importante analisar os fatores do paralelismo usando tarefas. Vale ressaltar, novamente, que um desses fatores é derivado da técnica de divisão-e-conquista aplicado no algoritmo clássico. Conforme a divisão acontece, novas tarefas são cridas e, por isso, pode acontecer uma sobrecarga no ambiente de execução.

Outro fator importante seria entender o comportamento de cada solução proposta em uma arquitetura, não só para compreender os fatores de sobrecarga, mas também analisar a escalabilidade da solução. Essa análise envolveria o aumento da quantidade de processadores (ou *cores*, como são chamados atualmente), mantendo o mesmo tamanho da entrada, como previsto na escalabilidade forte de Amdahl. Uma outra análise consideraria um aumento proporcional do tamanho da entrada comparado à quantidade de processadores utilizados, como discutido na escalabilidade fraca de Gustafson – veja mais detalhes em [Silva et al., 2022].

Um último fator importante é a busca por um valor limite de *CORTE*. Sendo ele descoberto de forma experimental, seria necessário realizar diversos experimentos em diferentes tipos de arquiteturas para encontrar um melhor valor.

Referências

[Beukman, 2021] Beukman, M. (2021). Parallel quicksort using openmp. Medium article at https://mcbeukman.medium.com/parallel-quicksort-using-openmp-9d18d7468cac.

[Bianchini et al., 2019] Bianchini, C. P., Vilabôas, F. G., and Castro, L. N. (2019). Paralelismo de tarefas utilizando openmp 4.5. In *Minicurso da ERAD/RS 2019*. https://setrem.com.br/erad2019/data/pdf/minicursos/mc06.pdf.

[Kumar, 2002] Kumar, V. (2002). *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

[OpenMP ARB, 2015] OpenMP ARB (2015). *OpenMP Application Programming Interface*. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[OpenMP ARB, 2016] OpenMP ARB (2016). *OpenMP Application Programming Interface Examples*. https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf.

[Silva et al., 2022] Silva, G., Bianchini, C., and Costa, E. B. (2022). *Programação Paralela e Distribuída*. Ed. Casa do Código. https://www.casadocodigo.com.br/pages/sumario-programacao-paralela.

[van der Pas et al., 2017] van der Pas, R., Stotzer, E., and Terboven, C. (2017). *Using OpenMP—The Next Step: Affinity, Accelerators, Tasking, and SIMD*. Scientific and Engineering Computation. MIT Press.