Capítulo 1

Diretivas Paralelas de OpenMP: Um Estudo de Caso

Claudio Schepke, Natiele Lucca
Universidade Federal do Pampa

João Vicente Ferreira Lima
Universidade Federal de Santa Maria

Resumo

OpenMP é uma interface de programação paralela usada para a geração implícita de threads em CPUs e GPUs, o que possibilita uma computação heterogênea. O padrão é baseado em diretivas de compilação, isto é, linhas de comando são inseridas em um código-fonte e tratadas em tempo de pré-compilação. O paralelismo é gerado pelas diretivas, as quais podem delimitar a região concorrente de código. Apesar da simplicidade de injeção de paralelismo em um determinado programa, a grande maioria dos usuários da interface limita-se a utilizar recursos elementares, como o paralelismo de laços em ambientes multiprocessados em detrimento de outras formas de invocação de operações concorrentes não tão difundidas. Este capítulo tem como objetivo apresentar como diferentes tipos de diretivas de OpenMP podem ser chamadas e seus impactos no desempenho paralelo em uma aplicação. Este estudo de caso busca mensurar especialmente a performance dos recursos disponíveis pelas versões mais recentes do padrão OpenMP.

1.1. Introdução

A interface de programação paralela OpenMP provê um conjunto de diretivas de compilador, bibliotecas de rotinas/funções, variáveis de ambiente e suporte a ferramentas para o gerenciamento, depuração e análise de programas paralelos em memória compartilhada e em aceleradores (OpenMP 2023). As diretivas multiplataforma estendem as linguagens de programação Fortran, C e C++ com construções do tipo: SPMD (*Single Program Multiple Data*), tarefas (*tasks*), dispositivos (*device*), distribuição de trabalho e sincronização. Além disso, as diretivas fornecem suporte para compartilhamento, mapeamento e privatização de dados. O controle do ambiente em tempo de execução é suportado por rotinas/funções da biblioteca e variáveis de ambiente. Os compiladores que suportam a

interface OpenMP geralmente incluem opções para habilitar ou desabilitar a interpretação de algumas ou todas as diretivas OpenMP por linha de comando.

A primeira especificação de OpenMP surgiu, para a escrita de códigos em Fortran, em outubro de 1997, e, para C/C++, em outubro de 1998, em uma tentativa das principais fabricantes de computadores da época de unificar o padrão de diretivas de compartilhamento de memória (Dagum and Menon 1998). Com o passar do tempo, outros fabricantes também passaram a colaborar com a especificação através de um conselho de revisão de arquitetura. Pelo fato da especificação ser um padrão dominante para um ambiente de programação paralela em memória compartilhada, diferentes fornecedores de compiladores suportam implementações da interface. Com isso, a API fornece um modelo de programação paralelo que é portável entre diferentes arquiteturas de computadores.

Em novembro de 2022 foi anunciada uma primeira descrição da versão 6 da especificação da interface OpenMP. Nela foram adicionadas novas funcionalidades e removidos recursos já considerados obsoletos em versões de especificações mais antigas da API. Embora uma apresentação da versão 6 já tenha sido feita, continuam ainda atuais e válidas as versões 5.2 (novembro de 2021) e 5.1 (novembro de 2020) da especificação. Vale lembrar que nem todos os compiladores implementam as modificações que ocorrem em novas versões das especificações imediatamente. Outros compiladores implementam apenas alguns dos recursos de uma determinada versão avançada da especificação.

Além do atraso natural existente entre o lançamento de uma versão da especificação e a implementação dela em software para uma nova versão de um compilador, há também uma aparente demora na utilização dos recursos por parte dos programadores. Por exemplo, a versão 3 introduziu a noção de tarefa (*tasks*) em maio de 2008. A versão 4 incorporou a construção de diretivas para uso de aceleradores (GPUs, coprocessadores, FPGAs, Cell BE, ...) e operações do tipo SIMD (*Single Instruction Multiple Data*) em julho de 2013. Já a versão 5, desde novembro de 2018, provê melhorias como: suporte completo para aceleradores, depuração aprimorada e análise de desempenho, suporte para uma construção de laços que permite que o compilador escolha uma boa implementação para um dispositivo específico e sistemas de memória multinível. Embora as versões mais recentes incorporam um poder de abstração significativamente alto e permitem instanciar rotinas/funções em multi-core, GPUs e outros tipos de hardware, a popularização destas diretivas tem sido lenta.

Diante do que foi previamente exposto, este capítulo apresenta algumas diretivas paralelas fornecidas por OpenMP. Na sequência, como estudo de caso, a paralelização de uma aplicação de dinâmica dos fluidos ((da Silva et al. 2022b)) é acelerada através da adoção de diferentes diretivas que possibilitam executar o código, tanto em arquitetura multi-core como em GPU. Cada uma das versões de código com diretiva paralela distinta tem o seu tempo de execução mensurado para fins de comparação.

As próximas seções apresentam aspectos iniciais que precisam ser observados na paralelização de aplicações (Seção 1.2), uma discriminação das diretivas paralelas de OpenMP que podem ser usadas (Seção 1.3 e uma descrição da aplicação que servirá como estudo de caso (Seção 1.4). Na sequência, são mostrados exemplos de trechos de código implementados com as diretivas paralelas de OpenMP (Seção 1.5), seguido dos resultados experimentais obtidos para a avaliação da performance para cada tipo de diretiva adotada

(Seção 1.6). O capítulo encerra com a conclusão (Seção 1.7).

1.2. Paralelização de Aplicações

Antes de iniciar propriamente a paralelização de qualquer aplicação é necessário um estudo prévio, a fim de identificar os trechos de código que podem ser paralelizados. Um código possui trechos que não são paralelizáveis. Este é o caso de etapas de pré e pós processamento, que incluem a leitura ou escrita de arquivos, a alocação de memória e a inicialização de variáveis. Em outras situações, há trechos de códigos em que o custo de instanciação da execução paralela não compensa o pouco tempo de execução, devido a natureza das operações ou a baixa carga de computação.

Muitos algoritmos têm uma característica iterativa, onde uma etapa depende da computação da etapa anterior e internamente a cada iteração há computações que podem ocorrer concorrentemente. A dependência entre os dados é um fator importante na paralelização de aplicações uma vez que, se não tratado adequadamente, pode gerar resultados incorretos devido ao acesso de posições de memória com valores ainda não atualizados.

Uma característica que deve ser evitada na programação paralela são sincronizações sucessivas em blocos paralelos. Um bloco paralelo realiza o lançamento de *n threads*. Entretanto sucessivas pausas para sincronizar os resultados parciais reduzem significativamente a eficiência gerada pelo paralelismo, uma vez que cada *thread* pode ter um tempo de computação distinto, o que demanda de esperas para sair da barreira (sincronização).

O código que faz uso da GPU deve obter ganho de desempenho superior ao da execução do bloco em CPU. A GPU possibilita a execução de blocos com grande volume de dados em um tempo significativamente inferior ao da CPU. Mas, todos os dados manipulados no bloco paralelo devem ser copiados para a memória da GPU e os dados retornados devem ser copiados para a memória da CPU. Essas cópias podem inviabilizar algumas paralelizações, pois a análise não deve considerar apenas a execução das instruções, mas também a sincronização das memórias.

Uma abordagem objetiva para identificar trechos de código paralelos é fazer uso de ferramentas de perfilamento de aplicações. A ferramenta gprof (Graham et al. 1982), por exemplo, faz coletas estatísticas do tempo de execução demandado por cada rotina que compõe o código e tem sido usado por muitos programadores para identificar inicialmente as funções mais custosas do código.

A performance de um código OpenMP pode ser avaliada pelo *speedup* (S). O *speedup* é definido como a razão entre o tempo de computação do algoritmo serial (T_{serial}) e o tempo de computação do algoritmo paralelo ($T_{paralelo}$), dado pela Equação 1. O *speedup* mostra o ganho efetivo do tempo de processamento do algoritmo paralelo sobre o algoritmo serial.

$$S = \frac{T_{serial}}{T_{paralelo}} \tag{1}$$

Quando o tempo paralelo é exatamente igual ao tempo sequencial, o *speedup* é igual a 1. Neste caso não há ganho de desempenho. Uma outra forma de mensurar o quanto uma versão paralela é melhor que a versão sequencial é considerar o percentual de

```
1 int fibo( int n ){
2  if( n < 2 ) return n;
3  int x = fibo(n-1);
4  int y = fibo(n-2);
5  return x + y;
6 }
```

```
1 for( i = 0; i < n; i++ )
2  compute_job(i);
```

ganho de desempenho apresentado na Equação 2.

$$S = \frac{T_{serial} - T_{paralelo}}{T_{paralelo}} * 100 \tag{2}$$

Algoritmos paralelos dependem da divisão da computação entre as unidades de processamento disponíveis. Desta forma, processos, threads ou fluxos concorrentes em GPU recebem preferencialmente uma quantidade equilibrada de carga de trabalho. Idealmente, também precisa-se coordenar as unidades de computação para sincronização e comunicação. Foster (Foster 1995) descreve um roteiro de quatro passos para desenvolver um programa paralelo: particionamento, comunicação, aglomeração e mapeamento.

Em relação à fase de particionamento, que indica onde deve haver paralelismo, as estratégias possíveis podem ser baseadas na concorrência sobre os dados ou na concorrência entre trechos de código independentes (Lima et al. 2021):

- O *Paralelismo de dados*, também conhecido como *decomposição de dados* ou *decomposição de domínio*, é um método recorrente para expressar concorrência em algoritmos. Nesse modelo de paralelismo, os dados associados ao domínio são particionados e então mapeados para tarefas concorrentes. Os dados dessa decomposição podem ser dados de entrada, saída, ou intermediários, ou seguem *Owner-Computes Rule* (OCR).

- O *Paralelismo de tarefas*, também chamado de *paralelismo funcional* ou *paralelismo de controle*, decompõe a computação ao invés dos dados manipulados. Esse modelo de paralelismo pode ser utilizado em tarefas que realizam computações distintas e independentes. Todavia, o paralelismo de tarefas é utilizado em algoritmos onde as tarefas podem ter dependências que resultam em um Grafo Acíclico Direcionado (*Directed Acyclic Graph - (DAG)* (Gautier et al. 2007). Algoritmos recursivos são um exemplo direto de paralelismo de tarefas em que a chamada recursiva é substituída por uma tarefa e por uma sincronização para esperar os resultados se necessário. Outro exemplo pode ser descrito por laços paralelos em que cada iteração é mapeada para uma tarefa sem dependências.

O exemplo a seguir ilustra os dois tipos de paralelismo, em que se pode substituir cada chamada de função pela criação de uma tarefa concorrente.

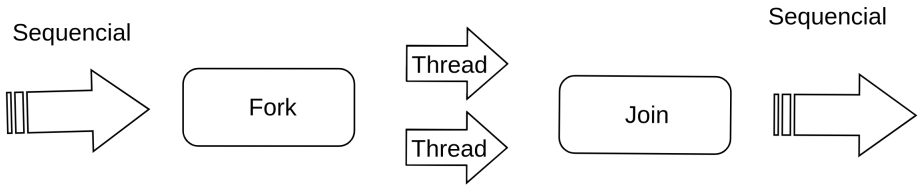

Figura 1.1. Instanciação de novas *threads* (*fork*) e término da execução (*join*).

1.3. Diretivas Paralelas de OpenMP

A API de OpenMP é fundamentada no modelo de execução *fork-join*. Esse modelo possui uma *thread* mestre que inicia a execução e gera *threads* de trabalho para executar as tarefas em paralelo (Chapman et al. 1999). OpenMP aplica o modelo em segmentos do código que são informados pelo programador. Dessa forma, um código sequencial é executado pela *thread* mestre até um bloco ou área de execução paralela, conforme apresentado na Figura 1.1.

O início da área paralela é demarcado por uma diretiva OpenMP que é responsável por sinalizar que as *threads* de trabalho devem ser lançadas (*fork*). Uma diretiva é precedida obrigatoriamente por `#pragma omp` (em C/C++) ou `!$omp` (em FORTRAN) e seguida por atributos opcionais. Todo o código seguinte é executado em paralelo pelas *threads* até o fim da área paralela que pode ser demarcado explicitamente como o símbolo de `}` (em C/C++) ou `!%omp end` (em FORTRAN), ou implícito, como por exemplo, em um laço de repetição `do/for`, onde o fim do laço também é o fim da área paralela. O fim da área paralela implica no encerramento das *threads* de trabalho, sincronização (*fork*) e retorno da *thread* mestre para a execução (*join*).

Seguem algumas diretivas que compõem a API OpenMP e que tradicionalmente são usadas pelos desenvolvedores (OpenMP 2023):

- `parallel`: Essa diretiva especifica que uma área do código será executada por *n threads*.

C / C++

```
1 #pragma omp parallel
2 {
3    // Bloco Paralelo
4 }
```

Fortran

```
1 !$omp parallel
2   ! Bloco Paralelo
3 !$omp end parallel
```

- `for`: Essa diretiva especifica como o trabalho será dividido entre as *threads*, mas não instância nenhuma *thread*. Ou seja, para que o paralelismo de laços ocorra, é necessário que esta chamada esteja precedida por uma chamada `parallel`, como no item anterior.

C / C++

```
1 #pragma omp for
2 for(i=1; i<imax; i++){
3     Bloco
4 }
```

Fortran

```
1 !$omp do
2   do i, imax
3     Bloco
4   enddo
5 !$omp end do
```

- `parallel for`: Especifica a construção de um laço paralelo, sendo que as iterações do laço de repetição serão distribuídas e executadas por *n threads*. Ou em outras palavras, ela é uma combinação dos dois itens anteriores, produzindo o mesmo resultado. A grande maioria dos programadores adota este estilo de chamada, sendo que o paralelismo de laços ainda é um dos modelos mais utilizados em OpenMP.

C / C++

```
1 #pragma omp parallel for
2 for(i=1; i<imax; i++){
3     Bloco Paralelo
4 }
```

Fortran

```
1 !$omp parallel do
2   do i, imax
3     Bloco Paralelo
4   enddo
5 !$omp end parallel do
```

- `simd`: Essa diretiva descreve que algumas iterações de um laço de repetição podem ser executadas simultaneamente por unidades vetoriais.

C / C++

```
1 #pragma omp simd
2 for(i=1; i<imax; i++){
3     Bloco
4 }
```

Fortran

```
1 !$omp simd
2   do i, imax
3     Bloco
4   enddo
5 !$omp end simd
```

- `for simd`: Essa diretiva especifica que um laço pode ser dividido em *n threads* que executam algumas iterações simultaneamente por unidades vetoriais.

C / C++

```
1 #pragma omp do simd
2 for(i=1; i<imax; i++){
3     Bloco
4 }
```

Fortran

```
1 !$omp do simd
2   do i, imax
3     Bloco
4   enddo
5 !$omp end do simd
```

- `target`: Mapeia variáveis para um ambiente de dados de um dispositivo e executa a construção neste dispositivo.

C / C++

```
1 #pragma omp target
2 for(i=1; i<imax; i++){
3     Bloco
4 }
```

Fortran

```
1 !$omp target
2   do i, imax
3     Bloco
4   enddo
5 !$omp end target
```

- `task`: Essa diretiva cria unidades de trabalho independentes. O uso da diretiva pode definir explicitamente uma tarefa. A diretiva identifica um bloco de código a ser executado em paralelo com o código fora da região de tarefa. As diretivas de tarefa podem ser para paralelizar algoritmos irregulares, onde as unidades de trabalho são geradas dinamicamente, como em estruturas recursivas ou laços do tipo *enquanto* (`while`). A programação com tarefas facilita a paralelização das aplicações.

C / C++

```
1 #pragma omp task
2   function(a)
3 }
4 #pragma omp task
5   function(b)
6 }
```

Fortran

```
1 !$omp task
2   function(a)
3 !$omp end task
4 !$omp task
5   function(b)
6 !$omp end task
```

Em OpenMP também há construções de compartilhamento de trabalho, como:

- `section`: A construção de seções é uma construção de compartilhamento de trabalho não iterativo que contém um conjunto de blocos estruturados que devem ser distribuídos e executados pelas *threads* de um grupo. Cada bloco estruturado é executado uma vez por uma das *threads* do grupo, no contexto de sua tarefa implícita.

- `single`: a construção especifica que o bloco estruturado associado é executado por apenas uma das *threads* do grupo (não necessariamente a *thread* principal), no contexto de sua tarefa implícita. As outras *threads* do grupo, que não executam o bloco, esperam em uma barreira implícita no final de uma única região, a menos que uma cláusula `nowait` seja especificada.

- `workshare`: A construção de compartilhamento de trabalho divide a execução do bloco estruturado fechado em unidades de trabalho separadas e faz com que as *threads* do grupo compartilhem o trabalho de modo que cada unidade seja executada apenas uma vez por uma *thread*, no contexto de sua tarefa implícita.

Na sequência são apresentados alguns atributos da API OpenMP, os quais podem ser associados às diretivas previamente apresentadas (OpenMP 2023). Para todos os casos, `lista` representa uma ou mais variáveis.

- `private (lista)`: Esse atributo informa que o bloco paralelo possui variáveis privadas para cada uma das *n threads*. As variáveis do bloco que não são informadas na lista são públicas.

- `shared (lista)`: O atributo especifica que as variáveis são públicas e compartilhadas entre as *n threads*.

- `num_threads(int n)`: Este atributo determina o número *n* de *threads* utilizadas no bloco paralelo. O valor de *n* é válido apenas para o bloco em que foi definido.

- `reduction (operador: lista)`: A redução é utilizada para executar cálculos em paralelos. Cada *thread* tem seu valor parcial. Ao final da região paralela o valor final da variável é atualizado com os cálculos parciais. O operador pode ser, por exemplo $+$, $-$, $*$, *max* e *min*.

- `nowait`: Uma diretiva OpenMP possui uma barreira implícita ao seu fim, com o objetivo de garantir a sincronização. Entretanto a diretiva `nowait` omite a existência dessa barreira. Dessa forma, as *threads* não ficam em espera até que as demais também terminem o trabalho.

- `collapse`: a cláusula associa um ou mais laços à diretiva, com o objetivo de identificar a profundidade *n* dos laços aninados, que podem aplicar a semântica da diretiva.

As variáveis de ambiente do OpenMP especificam características que afetam a execução dos programas. Seguem algumas variáveis (OpenMP 2023):

```
1  #pragma omp parallel
2  {
3  #pragma omp single
4   {
5    node* p = head;
6    while(p) {
7  #pragma omp task firstprivate(p)
8     process(p);
9     p = p->next;
10   }
11 #pragma omp taskwait
12  }
13 }
```

Figura 1.2. Exemplo de tarefas OpenMP para visitar elementos de uma lista encadeada.

- `OMP_NUM_THREADS`: Especifica o número *n* de *threads* utilizados nos blocos paralelos do algoritmo.
- `OMP_SCHEDULE`: A variável de ambiente controla o tipo de programação e o tamanho do bloco de todas as diretivas de *loop* do tipo `runtime` com as opções `static`, `dynamic`, `guided` ou `auto`.
- `OMP_THREAD_LIMIT`: Descreve o número máximo de *threads*.
- `OMP_NESTED`: Permite ativar ou desativar o paralelismo aninhado.
- `OMP_STACKSIZE`: Especifica o tamanho da pilha para as *threads*.

1.3.1. Paralelismo de Tarefas

A partir de sua versão 3.0, o OpenMP suporta o paralelismo de tarefas através da construção `task` para tarefas explícitas e `taskwait` para sincronização. A Figura 1.2 ilustra uma função para percorrer listas com criação de tarefas OpenMP. Note que em relação aos outros programas OpenMP, as tarefas são criadas dentro de uma construção `single` na linha 3. Isso se deve ao fato da região paralela executar o mesmo código em todas as threads, o que não é desejado nesse caso. Aqui a execução inicia com uma única thread apenas para que novas tarefas sejam criadas em seguida. Na linha 7 uma nova tarefa é criada para a função `process`. Note que a cláusula `firstprivate` é usada pois a variável `p` é modificada na próxima linha. Caso contrário, haveria uma condição de corrida entre a thread que cria tarefas e a nova tarefa.

O paralelismo de tarefas desenrola sua execução em um DAG onde as dependências são descritas pela estrutura recursiva do programa. Esse modo de execução, denominado *fully strict mode*, define que as relações de dependências ocorrem somente entre nós raízes e folhas com ligação direta.

```
1 #pragma omp taskgroup
2 {
3 #pragma omp task depend(in:data) depend(out:result)
4  foo(data, result);
5 }
```

Figura 1.3. Exemplo simples de tarefas OpenMP com dependências de dados.

Por outro lado, o paralelismo com dependências de dados controla a execução por meio de um grafo de fluxo de dados ou *Data Flow Graph* (DFG) (Gautier et al. 2007). O controle de execução é feito exclusivamente pelo fluxo de dados da aplicação e depende do modo de acesso descrito pela tarefa.

Os modos de acesso que podem ser listados, de uma forma genérica, são:

- **Read only (RO ou R)** - somente leitura, sem permissão para modificar.
- **Write only (WO ou W)** - somente escrita, sem leitura de dados de entrada.
- **Read and write (RW)** - ou modo exclusivo, com leitura e escrita.

O OpenMP versão 4.0 incluiu o uso de diretivas para expressar dependências de dados em tarefas. A diretiva `depend` de uma construção `task` lista as dependências de dados que podem ser:

- **in** – somente leitura.
- **out** – somente escrita.
- **inout** – leitura e escrita.

Além disso, a API inclui a construção de sincronização *taskgroup* que permite a sincronização implícita ao final do bloco de código a fim de esperar por todas as tarefas criadas recursivamente, o que não era possível com a diretiva **taskwait**. A Figura 1.3 demonstra um exemplo simples da criação de tarefas OpenMP com dependências de dados de entrada (`in`) e saída (`out`), além da sincronização recursiva para este bloco de código (`taskgroup`).

1.3.2. Aceleradores

O padrão OpenMP 4 ou superior inclui diretivas de execução de trechos de código em aceleradores por meio do modelo de execução *host-centric* onde a CPU principal, ou *host*, é o lugar onde a execução do programa inicia e o *device* é o acelerador para execução de trechos de código. O acelerador pode executar iterações de laços paralelos por meio de grupos de threads, chamados *teams*, que cooperam a fim de executar o trabalho.

```
1 int n = 1024;
2 float a = 32.0f, b = 17.0f;
3 float x[1024], y[1024];
4
5 #pragma omp target teams map (to:x[0:n]) map(tofrom:y[0:n])
6 #pragma omp distribute parallel for
7 for(int i= 0; i < n; i++) {
8   y[i] = a*x[i] + b*y[i];
9 }
```

Figura 1.4. Exemplo simples de uso de OpenMP para aceleradores.

A construção `target` muda o controle de execução do *host* para o acelerador e a construção `teams` cria um grupo de threads semelhante à construção `parallel`. Apenas algumas construções podem estar aninhadas a um `teams` como `distribute` e `parallel`. A construção `distribute` distribui as iterações de um laço entre as threads do grupo no acelerador. Outros atributos de `distribute` podem determinar o escalonamento e o grão de trabalho a cada thread (`dist_schedule`).

A diretiva `map` descreve o mapeamento explícito de variáveis ao ambiente de dados do acelerador. O tipo de mapeamento de dados com a diretiva `map` pode ser:

- `alloc` - aloca memória para a variável correspondente;
- `to` - aloca memória e copia o valor original para esta variável na entrada;
- `from` - aloca memória e copia o valor dela para a variável original na saída;
- `tofrom` - é o padrão, onde copia o valor na entrada e saída da região.

A construção `target` pode ser acompanhada da diretiva `nowait`, indicando que a CPU não espera o término do código na região `target`. A diretiva `depend` também pode ser utilizada a fim de sincronizar trechos de código assíncronos com `nowait`.

A Figura 1.4 demonstra um exemplo simples de programa SAXY de um laço executado em um acelerador com a construção `target`. Primeiramente a construção `target` (linha 5), seguida da construção `teams`, define a região a ser acelerada com um grupo de threads juntamente com o mapeamento dos vetores `x` e `y`. O vetor `x` é um dado de entrada e o vetor `y` é um dado de entrada e saída. Cada vetor tem o atributo `[0:n]` que define o tamanho do dado mapeado, sendo o vetor inteiro, em nosso exemplo. Em seguida, a construção `distribute parallel for` permite que o compilador execute um laço paralelo dentro da região acelerada no grupo de threads definido anteriormente.

1.4. Estudo de caso: Simulação de Secagem de Grãos

Uma aplicação de simulação de secagem de grãos foi escolhida como estudo de caso (de Oliveira 2020). Tal aplicação representa os grãos e o espaço entre os grãos como um sistema de escoa-

Algoritmo 1: Etapa iterativa do algoritmo

```
max_iterations ← 20.000 // número máximo de iterações
t0 ← 0 // tempo inicial
t ← 0.04 // tempo final
dt ← 0.01
while t0 < t do
  t0 ← to + dt
  i ← 1
  // calculando a convergência
  while i ≠ max_iterations do
    solve_U()
    solve_V()
    solve_P()
    solve_Z()
    convergence()
  end while
end while
```

mento em meios porosos, um típico problema da área de Dinâmica dos Fluidos (Lucca 2022). Assim, tem-se uma aplicação científica à disposição para a inserção de diferentes diretivas paralelas de OpenMP[1].

1.4.1. Algoritmo da Aplicação

A aplicação simula a secagem dos grãos através da passagem de ar quente, o que faz com que a transferência de temperatura para os grãos tenham como efeito a remoção da umidade (da Silva et al. 2022a). A simulação é baseada na discretização bidimensional usando volumes finitos para o problema, o qual é descrito matematicamente através das equações de Navier-Stokes. A transição do tempo também ocorre de forma discreta.

A aplicação modela as etapas de leitura dos valores de entrada, alocação e inicialização de dados, iteração do passo de tempo discreto, escrita dos resultados de saída em arquivos e desalocação de memória. A etapa que mais demanda de tempo de processamento é a iteração do passo de tempo discreto (da Silva et al. 2022c).

O Algoritmo 1 apresenta as rotinas invocadas na etapa iterativa da simulação. O laço principal itera o passo de tempo discreto. Em cada passo, são computadas rotinas responsáveis pela interação das propriedades físicas calculadas. Isto é feito iterativamente enquanto um critério de parada (`convergence()`) previamente definido ou um limite máximo de iterações não for atingido. Para cada tempo discreto há um número máximo de iterações até que se chegue a convergência dos valores de resíduo das propriedades de continuidade e momentos (U, V e P) do código. Nesta etapa iterativa são calculados as equações de Momento advindas de uma técnica conhecida como *Quick Scheme* (`solve_U` e `solve_V`), além de uma equação de Continuidade (`solve_P`) e de uma equação de Energia (`solve_Z`).

[1] https://github.com/GabrielDT02/porous-media-application

1.4.2. Rotinas e Subrotinas da Etapa Iterativa

As rotinas `solve_U`, `solve_V`, `solve_P` e `solve_Z` invocam sub-rotinas que iteram sobre todo o domínio bidimensional, para cada uma das propriedades físicas computadas. As sub-rotinas também realizam operações distintas para os elementos da borda do domínio. Estes exigem interações distintas entre os pontos. Também é necessário a invocação de sub-rotinas específicas para a aplicação das condições de contorno do problema.

Todas as rotinas e subrotinas responsáveis pela computação das equações de quantidade de movimento na dimensão horizontal e vertical, de continuidade e de energia estão implementadas no arquivo `equations.f90`. As rotinas `solve_U` e `solve_V` representam o tempo de, respectivamente, 43% e 41% do tempo total de execução do código sequencial, restando 7% para `solve_Z` e 1% para `solve_P` (Lucca et al. 2023).

1.5. Paralelização da Aplicação

As próximas subseções apresentam as formas como os trechos de código foram paralelizados.

1.5.1. `parallel do`

O laço externo da aplicação possui dependência temporal entre uma iteração e outra e, portanto, cada passo de tempo discreto não pode ser executado concorrentemente. Ou seja, uma etapa depende da etapa anterior, o que é uma característica de aplicações semelhantes. Situação semelhante acontece com o laço em que as operações continuam se repetindo enquanto a convergência não é atingida. Neste caso, é preciso calcular a convergência primeiro para saber se uma nova etapa iterativa precisa ser computada.

Já as quatro rotinas (`solve_U`, `solve_V`, `solve_P` e `solve_Z`) chamadas na etapa iterativa possuem trechos de código que podem ser executados concorrentemente. Como cada elemento a ser computado é o mesmo, o paralelismo de laços tende a ser uma abordagem eficiente para garantir um bom desempenho paralelo. Essencialmente há um padrão em cada rotina: há 4 laços aninhados que percorrem a representação bidimensional do domínio do problema. Assim, cada chamada (`do`) pode ser paralelizada com `!$omp parallel do`, com as devidas indicações de variáveis privadas especificadas com o atributo `private`. Na implementação do `!$omp parallel do`, foi necessário apenas identificar os laços internos da aplicação e incluir a diretiva apropriada. Foram encontrados 20 laços (ou laços aninhados) de potencial para a aplicação de `parallel do`. Alguns desses laços são chamados mais vezes (sub-rotinas) no mesmo passo discreto. Portanto, o número total de laços paralelos invocados é de 36 para cada etapa iterativa do laço de convergência, sendo 12 para `solve_U`, 12 para `solve_V`, 6 para `solve_P` e 6 para `solve_Z`. O algoritmo 2 mostra um exemplo de paralelismo de laço aplicado a um trecho de código da rotina `solve_U`. Aplicações semelhantes são feitas para os demais laços identificados.

1.5.2. `parallel task`

Uma outra abordagem de paralelização possível é fazer uso da diretiva `!$omp task`, uma vez que existem trechos de computação que podem ser executados concorrentemente. Isto é, existem rotinas que a cada iteração do tempo discreto do código são executados

Algoritmo 2: Implementação paralela usando a diretiva `parallel do` em um laço da função `solve_U`

```
1
1 !$omp parallel do private(i,j)
2 DO j=2,jmax-1
3   DO i=3,imax-1
4     res_u(i,j) =((um(i,j)-um_tau(i,j)) +RU(i,j)*dt)*dtau
5     ui(i,j) = ( um_tau(i,j) + res_u(i,j))
6   ENDDO
7 ENDDO
8 !$omp end parallel do
```

sequencialmente, mas que poderiam ser executados concorrentemente, pois operam sobre conjuntos de dados distintos. Como exemplo tem-se as operações que são feitas em `solve_U` e `solve_V`, pois são de dimensões distintas (em x e em y).

A Figura 1.5 apresenta algumas modificações necessárias para que os trechos de código de ambas as rotinas possam ser executados concorrentemente. Isso acontece porque as condições de contorno operam sobre as estruturas de dados utilizadas tanto em `solve_U`, como em `solve_V`, ou seja, tal operação precisa ser feita por um único fluxo de execução.

O Algoritmo 3 apresenta a invocação de tarefas concorrentes para executar as rotinas `solve_U` e `solve_V`.

1.5.3. `parallel sections`

A diretiva de seções paralelas do OpenMP identifica seções de código para dividir entre todas as threads. De acordo com a especificação do padrão OpenMP, a construção de uma seção é uma construção de compartilhamento de trabalho não iterativa. Ele contém um conjunto de blocos estruturados a serem distribuídos e executados pelas threads em um grupo. A execução de um bloco estruturado é feita uma vez por uma das threads do grupo, considerando o contexto de sua tarefa implícita (OpenMP 2023). Isso permite a definição de concorrência em alto nível.

Na aplicação, pode-se executar simultaneamente `solve_U` e `solve_V`, duas das principais rotinas chamadas em um laço iterativo. Para utilizar de forma eficiente as seções paralelas, foi necessário reestruturar o código para manter a sequência de execução entre os métodos de resolução para execução paralela em eixos diferentes (x e y) da diretiva `teams`. Uma região definida permite que o código seja específico em paralelo. Cada seção cria uma thread que executa uma tarefa independente.

Nesta nova estrutura, foi possível inserir a diretiva `teams` para executar o *solve_U*$_i$ e `solve_V`$_i$ em paralelo, para i=1, 2 e 3, como apresentado na Figura 1.5. As diretivas `task` e *section* são de muitas maneiras semelhantes. As seções incluídas estão dentro da construção `sections` e as threads não sairão dela até a execução de todas as *sections*.

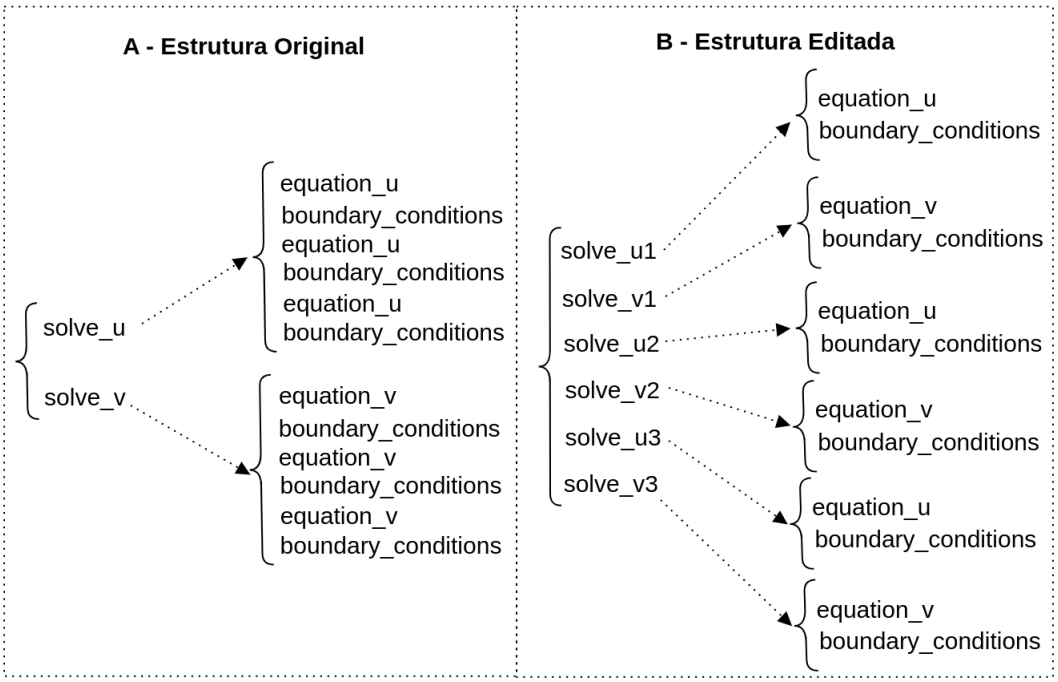

Figura 1.5. Modificações das funções `solve_U` e `solve_V` para a obtenção do paralelismo de tarefas

Algoritmo 3: Uma região de código usando a diretiva `!$omp task`

```
1
1  DO WHILE (time .LT. final_time)
2    time = time + dt
3    DO WHILE(itc.LT.itc_max)
4      !$omp parallel
5
6      !$omp task
7      CALL solve_U1()
8      !$omp end task
9
10     !$omp task
11     CALL solve_V1()
12     !$omp end task
13
14     !$omp taskwait
15
16     !$omp end parallel
17     ...
18     CALL solve_P()
19     CALL solve_Z()
20     CALL convergence()
21   ENDDO
22 ENDDO
```

Algoritmo 4: Trecho de código usando a diretiva `!$omp sections`

```
1
1  DO WHILE (time .LT. final_time)
2   time = time + dt
3   DO WHILE(itc.LT.itc_max)
4    !$omp parallel sections
5    ...
6    !$omp section
7    CALL solve_U1()
8    !$omp section
9    CALL solve_V1()
10   ...
11   !$omp end parallel sections
12   ...
13   CALL solve_P()
14   CALL solve_Z()
15   CALL convergence()
16  ENDDO
17 ENDDO
```

1.5.4. `target teams distribute parallel do`

A diretiva `target` permite a seleção de um tipo de arquitetura (Multi-core ou GPU), onde a execução do segmento de código será sequencial. A combinação com outras diretivas, como a diretiva `teams` apresentada anteriormente, gera paralelismo. A diretiva nos permite explorar o recurso *Streaming Multiprocessor* disponível nas GPUs ou nos núcleos do Multi-core. Também é possível usar a diretiva `distribute` para explorar o paralelismo. Esta diretiva distribui as iterações de um laço do tipo *do-loop* entre as threads mestres. Também é possível combinar com a diretiva *parallel do*.

O Algoritmo 5 mostra a definição de uma região do tipo `target`, onde usou-se a diretiva `teams`, e um trecho de código chama uma operação `distribute parallel do`.

1.6. Resultados Experimentais

Foi definido um caso de teste para uma malha de tamanho de 100×124 pontos para avaliar o desempenho das implementações paralelas. Foram realizadas 30 execuções para obter o tempo médio para cada um dos casos de teste. Consideramos um tempo de simulação de $0,04$. O tempo começa em $0,0$. O intervalo de tempo discreto é $0,01$ (Δt). Em todos os casos, o número máximo de iterações usadas para convergência em cada tempo discreto foi de 20.000.

O código foi compilado com o compilador pgf90 (nvfortran), usando o NVidia HPC_SDK 21.2 *toolkit*, com a adição das flags `-O3`, `-fopenmp` e `Minfo=all`. A composição do ambiente computacional utilizado neste trabalho para execução dos testes é de dois processadores Intel Xeon CPU E5-2650 octa-core e uma GPU Nvidia Quadro

Algoritmo 5: Trecho de código usando a diretiva `!$omp target`

```
1
1  !$OMP TARGET
2  ...
3  !$OMP TEAMS
4  ...
5  !$omp distribute parallel do collapse(2)
6  do i=2,imax-1
7   do j=2,jmax-1
8    pi(i,j) = p(i,j) + 8.d-2 * RP(i,j)*c2
9   enddo
10 enddo
11 !$omp distribute parallel do
12 do i=1,imax
13  pi(i,1 ) = pi(i,2)
14  pi(i,jmax) = pi(i,jmax-1) + 1.d0*(pi(i,jmax-1)-pi(i,jmax-2))
15 enddo
16 !$omp distribute parallel do
17 do j=1,jmax
18  pi(1,j) = pi(2,j)
19  pi(imax,j) = pi(imax-1,j)
20 enddo
21 ...
22 !$OMP END TEAMS
23 ...
24 !$OMP END TARGET
```

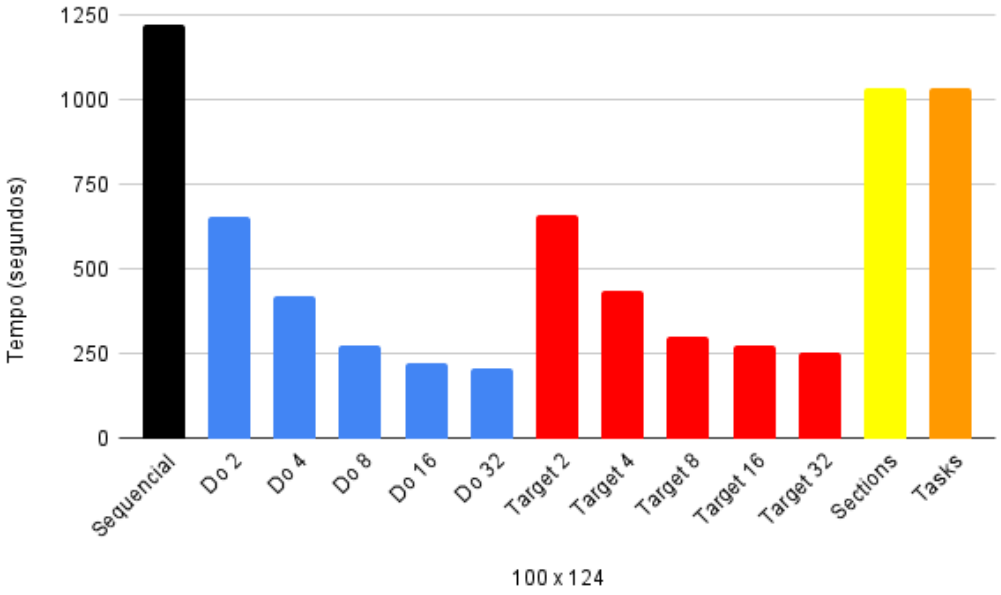

Figura 1.6. Tempo de execução para uma malha de tamanho 100×124**.**

M5000.

A Figura 1.6 apresenta o tempo médio de execução em segundos para uma malha de configuração de tamanho 100×124 pontos. A figura mostra as versões: sequencial, OpenMP `parallel do`, `sections`, `tasks` e `target`. O número de threads utilizado para avaliar as implementações de OpenMP foram 2, 4, 8, 16 e 32. Também foi calculado o desvio padrão para esses tempos médios. O desvio padrão permaneceu baixo e seu valor é inferior a 1% do tempo total de execução. Também foi avaliado e garantido a compatibilidade numérica das versões paralelas em relação à versão sequencial. Ou seja, todos os códigos precisam produzir valores idênticos para os resultados produzidos.

A implementação paralela de OpenMP fornece redução do tempo de execução. Em todos os experimentos, adicionar mais threads resulta em maior redução no tempo de execução. Uma exceção ocorre quando usamos mais threads do que núcleos físicos (teste com 32 threads). Neste caso, o ganho foi menor que no teste com 16 threads. O *speedup* com a diretiva `parallel do` foi de $5,53$, considerando o uso de 16 threads. A execução em cada etapa iterativa consiste em 36 loops paralelos. Só não foram paralelizados os laços das operações de condições de contorno devido à simplicidade dos cálculos, ou seja, apenas operações de atribuição. Esses resultados são compatíveis com os valores encontrados em outros trabalhos relacionados. Os resultados de *speedup* poderiam ser mais expressivos caso uma malha de tamanho maior fosse utilizada.

As diretivas `tasks` e `sections` fornecem resultados semelhantes, até mesmo pelo fato de realizarem chamadas semelhantes. No entanto, esses resultados são apenas um pouco melhores do que a implementação sequencial. A aceleração foi de $1,18$. A aplicação é executada simultaneamente apenas por solve_u e solve_v por um grupo de threads. Foram utilizadas 32 threads como padrão nos experimentos, mas de fato o número

de seções ou tarefas simultâneas é limitado às operações nas dimensões x e y.

Os resultados do OpenMP `target` apresentam resultados próximos ao `parallel do`. O *speedup* foi de $4,50$ usando 16 threads. Na implementação, a versão `target` chama `teams` e `distribute parallel do` para executar os laços simultaneamente.

1.7. Conclusão

Paralelizar uma aplicação possui desafios. Muitas vezes é necessário reescrever ou realizar adaptações no código sequencial, para que o mesmo possa ser executado concorrentemente, ou seja, sem dependência de dados ou de operações. Posteriormente, deve-se partir para a paralelização do código. Algumas técnicas de paralelismo podem ser escolhidas por serem mais adequadas para uma determinada classe de problemas ou devido às características que o domínio do problema possui. Todos os casos de teste apresentados obtiveram *speedup* positivo, ou seja, obtiveram ganho de desempenho. Além do ganho de desempenho também é preciso garantir a equivalência numérica dos resultados, ou seja, uma versão paralela não pode resultar em valores inconsistentes da solução do programa sequencial.

Neste capítulo foi descrita uma aplicação e apresentadas formas de paralelização que puderam ser aplicadas usando a interface de programação OpenMP. As especificações mais recentes de OpenMP permitem tanto a criação de tarefas paralelas quanto o uso de GPUs através de diretivas `target`. Desta forma, OpenMP aparece com uma alternativa para que que uma aplicação possa ser paralelizada tanto em um ambiente multi-core, quanto many-core, deixando de ser utilizado somente o tradicional paralelismo de laços através da combinação das diretivas `parallel` e `for`.

Referências

aaaa aaaa.

Chapman et al. 1999 Chapman, B., Mehrotra, P., and Zima, H. (1999). Enhancing OpenMP with features for locality control. In *Proceedings of Eighth ECMWF Workshop on the Use of Parallel Processors in Meteorology*, pages 301–13, Singapore. Towards Teracomputing. World Scientific Publishing.

da Silva et al. 2022a da Silva, H. U., Lucca, N., Schepke, C., de Oliveira, D. P., and da Cruz Cristaldo, C. F. (2022a). Parallel OpenMP and OpenACC Porous Media Simulation. *The Journal of Supercomputing*.

da Silva et al. 2022b da Silva, H. U., Schepke, C., da Cruz Cristaldo, C. F., de Oliveira, D. P., and Lucca, N. (2022b). An Efficient Parallel Model for Coupled Open-Porous Medium Problem Applied to Grain Drying Processing. In Gitler, I., Barrios Hernández, C. J., and Meneses, E., editors, *High Performance Computing*, pages 250–264, Cham. Springer International Publishing.

da Silva et al. 2022c da Silva, H. U., Schepke, C., Lucca, N., da Cruz Cristaldo, C. F., and de Oliveira, D. P. (2022c). Parallel OpenMP and OpenACC Mixing Layer

Simulation. In *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, volume 1, pages 181–188.

Dagum and Menon 1998 Dagum, L. and Menon, R. (1998). OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55.

de Oliveira 2020 de Oliveira, D. P. (2020). Fluid Flow Through Porous Media with the One Domain Approach: A Simple Model for Grains Drying. Dissertação de mestrado, Universidade Federal do Pampa, Alegrete.

Foster 1995 Foster, I. (1995). *Designing and Building Parallel Programs: Concepts and tools for Parallel Software Engineering*. Addison Wesley, Reading, MA.

Gautier et al. 2007 Gautier, T., Besseron, X., and Pigeon, L. (2007). KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *2007 international workshop on Parallel symbolic computation*, pages 15–23, Waterloo, Canada. ACM.

Graham et al. 1982 Graham, S. L., Kessler, P. B., and Mckusick, M. K. (1982). Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, 17(6):120–126.

Lima et al. 2021 Lima, J. V. F., Schepke, C., and Lucca, N. (2021). Além de Simplesmente: #pragma omp parallel for. In Charão, A. and da Silva Serpa, M., editors, *Minicursos da XXI Escola Regional de Alto Desempenho da Região Sul*, chapter 4, pages 86–103. Sociedade Brasileira de Computação, Porto Alegre/RS.

Lucca 2022 Lucca, N. (2022). *Avaliação de estratégias de paralelismo em simulação de meios porosos*. PhD thesis, Dissertação (Mestrado Profissional em Engenharia de Software) – Universidade Federal do Pampa, Campus Alegrete, Alegrete/RS.

Lucca et al. 2023 Lucca, N., Schepke, C., and Tremarin, G. D. (2023). Parallel Directives Evaluation in Porous Media Application: A Case Study. In *2023 31th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, volume 1.

OpenMP 2023 OpenMP (2023). The OpenMP API Specification for Parallel Programming.