Capítulo 4

Além de Simplesmente: #pragma omp parallel for

João Vicente Ferreira Lima - jvlima@inf.ufsm.br[1]
Claudio Schepke - claudioschepke@unipampa.edu.br[2]
Natiele Lucca - natielelucca@gmail.com[3]

Resumo

OpenMP tem sido o padrão de fato para a programação em memória compartilhada. No entanto, a maioria dos programadores explora apenas o paralelismo de laços, deixando de usar novos e outros recursos disponíveis nas versões mais recentes da especificação de OpenMP (3, 4 e 5). Com isso, outras abordagens paralelas não tem sido tão difundidas. Além disso, disparar tarefas em CPU e GPU usando uma única interface de programação é um grande atrativo para a paralelização de aplicações. Neste contexto, este capítulo tem como objetivo aprofundar a programação paralela em aplicações, com recursos considerados avançados de OpenMP, geralmente não adotados ou vistos nas disciplinas introdutórias de programação paralela. Para tanto, são apresentadas técnicas de exploração do paralelismo disponibilizado pelas diretivas de execução concorrente de OpenMP em diferentes trechos de código de duas aplicações científicas.

[1]João Lima possui graduação em Ciência da Computação pela Universidade Federal de Santa Maria (2006), mestrado em Computação pela Universidade Federal do Rio Grande do Sul (2009) e doutorado em Computação em co-tutela entre a Université de Grenoble e Universidade Federal do Rio Grande do Sul (2014). Atualmente é Professor Adjunto do Departamento de Linguagens e Sistemas de Computação da Universidade Federal de Santa Maria. Tem experiência na área de Ciência da Computação, com ênfase em Processamento Paralelo de Alto Desempenho, atuando principalmente nos seguintes temas: programação paralela e linguagens de programação.

[2]Claudio Schepke possui graduação em Ciência da Computação pela Universidade Federal de Santa Maria (2005) e mestrado (2007) e doutorado (2012) em Computação pela Universidade Federal do Rio Grande do Sul, sendo este feito na modalidade sanduíche na Technische Universität Berlin, Alemanha (2010-2011). É professor adjunto da Universidade Federal do Pampa (UNIPAMPA), campus Alegrete/RS desde 2012. Tem experiência na área de Ciência da Computação, com ênfase em Processamento Paralelo e Distribuído, atuando principalmente nos seguintes temas: processamento de alto desempenho, programação paralela, aplicações científicas e computação em nuvem.

[3]Natiele Lucca é graduada em Ciência da Computação pela Universidade Federal do Pampa (UNIPAMPA) (2020). Atualmente é mestranda do Programa de Pós-Graduação em Engenharia de Software (PPGES) da UNIPAMPA/Alegrete. Tem experiência em programação paralela e algoritmos bio-inspirados.

4.1. Introdução

Uma das motivações para o desenvolvimento de programas paralelos é acelerar aplicações científicas. Aplicações deste tipo geralmente demandam de um grande tempo de computação para uma versão com um único fluxo de execução de código, o que pode levar minutos, horas ou até mesmo dias, dependendo do tamanho do domínio ou resolução do problema adotado. Uma das maneiras de gerar paralelismo de maneira simples e eficiente a partir de um código-fonte é inserir diretivas (pragmas). Diretivas de pré-compilação possibilitam a geração de código específico e automatizado pela conversão das instruções. Desta forma, as instruções paralelas podem ser incluídas no código antes da compilação de fato do mesmo.

Diretivas paralelas geram fluxos concorrentes de código, que podem ser executados tanto em arquiteturas multi-core como many-core. Este capítulo aborda a prática de programação com diretivas paralelas e tem como objetivo apresentar técnicas de exploração de paralelismo em diferentes trechos de código para um conjunto de aplicações científicas usando a interface de programação OpenMP. Neste sentido, são demonstrados exemplos reais do impacto do uso de pragmas no desempenho de códigos, incluindo situações em que a granularidade impede que se obtenha a aceleração do programa.

4.2. Arquiteturas Paralelas

A execução de aplicações pode ser feita em diferentes tipos e níveis de paralelismo. Multi-core com unidades vetoriais expressivas, processadores vetoriais, coprocessadores e GPUs tem-se destacado na composição de computadores e combinados na formação de *clusters* de alta performance, conforme ilustrado na Figura 4.1. Nesta seção são discutidas as diferentes formas de paralelismo oferecidas pelas arquiteturas disponíveis atualmente.

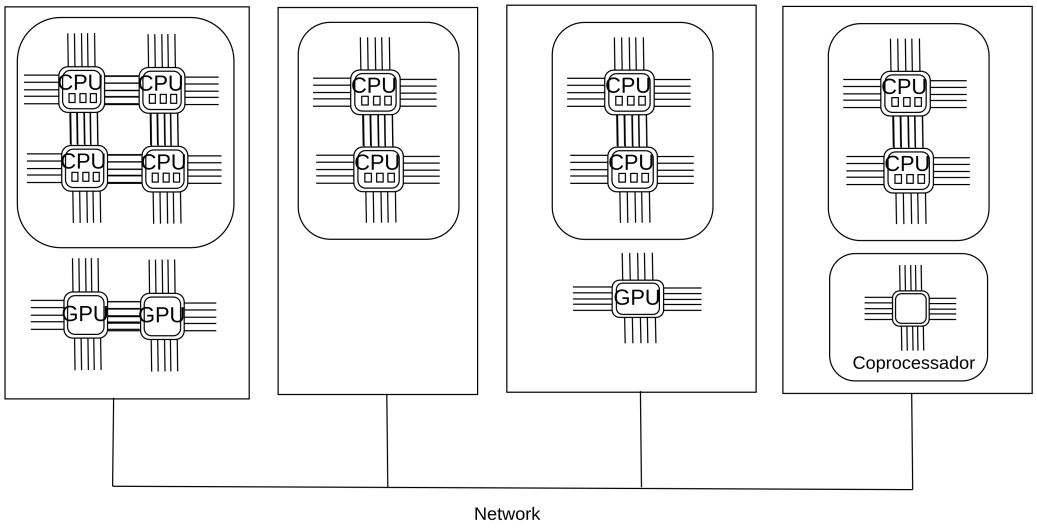

Figura 4.1. Exemplo de arquitetura de *cluster* com composição heterogênea de nós.

A concorrência das instruções, além da concepção clássica de processadores (pipeline e superescalar), visto tradicionalmente em disciplinas de arquitetura e organização

de computadores, pode ainda ser feito através de instruções vetoriais. Embora existam arquiteturas com processadores especificamente vetoriais (como é o caso dos processadores NEC SX-Aurora TSUBASA - Vector Engine (NEC Corporation, 2021)), qualquer core de um processador de propósito geral também possui uma unidade para a execução de instruções vetoriais. Tem-se visto inclusive, com o passar dos anos, o aumento do número de operações que podem ser realizadas simultaneamente. Por exemplo, em `AVX512` pode-se executar 32 operações de ponto flutuante de precisão dupla ou 64 operações de ponto flutuante de precisão simples por ciclo de clock ou oito inteiros de 64 bits ou dezesseis inteiros de 32 bits com até duas unidades de operação fundida multiplicação-adição - FMA (CORPORATION, 2021a). Neste sentido, basta ativar, no momento da compilação do código-fonte, *flags* que orientam o compilador a gerar instruções para essas unidades vetoriais (`AVX512, AVX2, AVX, SSE4_1, SSE4_2, SSE, MMX`). Uma dica de GCC é a opção `-O3` ou `-ftree-vectorize` mais `mavx`.

Especificamente em relação à arquitetura multicore, uma grande variedade de modelos e quantidades de core estão à disposição para a implementação da concorrência em nível de processo ou *thread*. Estes processadores possuem também diferentes quantidades de memória *cache* e frequência de *clock* e de acesso ao barramento. Não é tão comum em computadores pessoais, mas um computador pode também ser composto por mais de um processador multicore, aumentando ainda mais o número de unidades de computação presentes em uma única máquina (placa-mãe). Atualmente, como exemplos, há processadores com até 28 cores / 56 *threads* para processadores Intel (Xeon Platinum 8380H, 2.90 GHz / 4.30 GHz, 38.5 MB de Cache, TDP 250 W e 14 nm de litografia (CORPORATION, 2021b)) e 64 cores / 128 *threads* para processadores AMD (EPYC Embedded 7H12, 2.6GHz / 3.3GHz, 256 MB de Cache, TDP 280 W e 7/14 nm de litografia (Advanced Micro Devices, Inc, 2021)). Além dos processadores multi-core tradicionais, existem os conhecidos aceleradores de hardware, como cooprecessadores e GPUs.

A ideia do uso de coprocessadores ressurgiu nos meados de 2012 e aparentemente já foi descartada, embora a arquitetura ainda encontra-se presente em diversas máquinas em produção. Os coprocessadores Intel Xeon Phi foram criados para otimizar a relação performance por watt para aplicações com cargas de trabalho altamente paralelizáveis (Gonçalves; Girardi; Schepke, 2018). Foram também disponibilizados um conjunto de ferramentas de software para utilizar esta arquitetura *manycore* com suporte a instruções vetoriais (SIMD). Nos modelos *Knights Corner* há versões com até 61 cores físicos de 1.2GHz e com 4 *threads* de hardware por core, o que possibilita a execução simultânea em até 244 *threads* físicas. Os coprocessadores disponibilizam instruções SIMD com tamanho de 512 *bits*, com 32 registradores nativos, que suportam um desempenho de pico com precisão dupla de até 1 teraFLOPS/s Nos modelos *Knights Landing* há versões com até 72 cores, também com 4 *threads* de hardware por core.

Já a computação de propósito geral em Unidades de Processamento Gráfico (GPU), abordagem conhecida como GPGPU, superou as expectativas iniciais, e mantém-se como uma alternativa aos processadores convencionais, em parte devido à facilidade com que esses processadores podem explorar o paralelismo em grande escala. O desempenho de GPUs contemporâneas cresceu mais rapidamente do que o provido pelas arquiteturas multicores. Isso beneficiou aplicações bem estabelecidas ou clássicas da computação científica, como por exemplo as operações que envolvem álgebra linear, que demandam

historicamente de muito tempo de processamento para as simulações computacionais. Mais recentemente, com o ressurgimento da área de Inteligência Artificial com a noção de *Deep Learning*, GPUs mostraram-se uma ótima alternativa para acelerar o treinamento de algoritmos desenvolvidos nesta área, o que inclusive modificou os tipos e tamanho das operações suportadas por novas versões de GPUs. Um modelo de GPU como o A100 da NVIDIA, por exemplo, tem condições de processar até 9,7 TFlops de ponto flutuante de precisão dupla ou 9,5 TFlops usando *Tensor Cores*. Para este modelo há um total de 6.912 núcleos CUDA, 40 GB de memória e 1.6 TB de largura de banda (NVIDIA, 2021).

Diante de tudo isso, tem-se atualmente uma composição bastante heterogênea em termos de arquitetura de computadores disponíveis. Por outro lado, não há uma interface de padrão estabelecida que possibilite programar todos os modelos de arquitetura existentes. Excluindo-se a computação inter-computadores (*clusters*), que necessitam de uma biblioteca de troca de mensagens como *Message-Passing Interface* - MPI, OpenMP é uma das alternativas para a geração de código do tipo SIMD (instruções vetoriais), multicore e aceleradores de hardware.

4.3. A Interface de Programação OpenMP

OpenMP é uma API para programação paralela de memória compartilhada e multiplataforma disponível em C/C++ e Fortran (OPENMP, 2021). A API 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; MEHROTRA; ZIMA, 1998). 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 4.2.

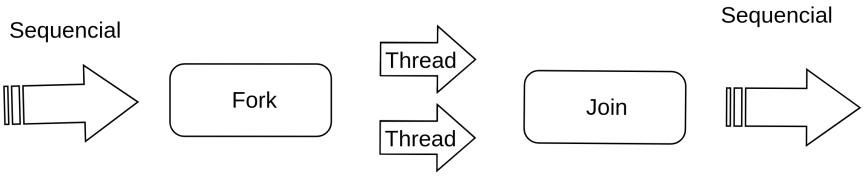

Figura 4.2. Instanciação de novas *threads* (*fork*) e término da execução (*join*).

O inicio da área paralela é demarcado por uma diretiva OpenMP que é responsável por sinalizar que as *threads* de trabalho devem ser lançadas (*fork*). 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 `}` ou `!%OMP END` ou implícito, como por exemplo, em um laço de repetição `for`, onde o fim do laço de repetição 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.

A API OpenMP possui um conjunto de diretivas de compilação, uma biblioteca de rotinas de tempo de execução e variáveis de ambiente para a programação paralela (TORELLI; BRUNO, 2004). Uma diretiva é precedida obrigatoriamente por `#pragma omp` (em C) ou `!$omp` (em FORTRAN) e seguida por `[atributos]`, sendo que os atributos são opcionais. Seguem algumas diretivas que compõem a API OpenMP e que

tradicionalmente são usadas pelos desenvolvedores (OPENMP, 2021):

- `parallel`: Essa diretiva descreve que a uma área do código será executada por *n threads*, sendo *n* o número de *threads* especificados por um atributo, chamada de função ou variável de ambiente.
- `for`: Essa diretiva especifica que as iterações do laço de repetição serão executadas em paralelo por *n threads*.
- `parallel for`: Especifica a construção de um laço paralelo, sendo que o laço será executado por *n threads*.
- `simd`: Essa diretiva descreve que algumas iterações de um laço de repetição podem ser executadas simultaneamente por unidades vetoriais.
- `for simd`: Essa diretiva especifica que um laço pode ser dividido em *n threads* que executam algumas iterações simultaneamente por unidades vetoriais.
- `target`: Mapeia variáveis para um ambiente de dados do dispositivo e executa a construção nesse dispositivo.
- `task`: Define uma tarefa explicitamente.

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 pelos *threads* em uma equipe. Cada bloco estruturado é executado uma vez por uma das *threads* da equipe no contexto de sua tarefa implícita.
- `single`: a construção especifica que o bloco estruturado associado é executado por apenas uma das *threads* na equipe (não necessariamente a *thread* principal), no contexto de sua tarefa implícita. As outras *threads* na equipe, 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* da equipe 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 (OPENMP, 2021). 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)`: Esse 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.

As variáveis de ambiente do OpenMP especificam características que afetam a execução dos programas. Seguem algumas variáveis (OPENMP, 2021):

- `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, or 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*.

4.3.1. Paralelismo de Tarefas

O desenvolvimento de algoritmos paralelos em geral necessita da divisão de trabalho entre as unidades de processamento (PU), processos ou threads, onde cada PU recebe aproximadamente a mesma quantia de trabalho. Idealmente, também precisa-se coordenar os PUs para sincronizar e comunicar entre si. Foster (FOSTER, 1995) descreve um roteiro de quatro passos para desenvolver um programa paralelo: particionamento, comunicação, aglomeração e mapeamento. *Paralelismo de dados* e *paralelismo de tarefas* estão diretamente relacionados com a fase de particionamento, onde expõe-se as oportunidades de concorrência. As duas estratégias dividem o problema em pedaços pequenos baseados nos dados ou na computação, respectivamente.

Paralelismo de dados, também *decomposição de dados* ou *decomposição de domínio*, é um método recorrente para expressar concorrência em algoritmos. Nesse modelo de programação, os dados associados com o problema são particionados e então mapeados para tarefas. Os dados dessa decomposição podem ser dados de entrada, saída, ou intermediários, ou seguem *owner-computes rule* (OCR).

Paralelismo de tarefas, também *paralelismo funcional* or *paralelismo de controle*, representa uma forma diferente e complementar de expressar paralelismo. Essa estratégia decompõe a computação ao invés dos dados manipulados. Esse modelo de programação pode ser utilizado em tarefas que realizam computações diferentes e são independentes. Todavia, o paralelismo de tarefas é utilizado em algoritmos onde as tarefas podem ter dependências que resultam em um grafo acíclico direcionado (DAG). Quando as dependências entre tarefas são associadas aos dados, o algoritmo gera um grafo de fluxo de dados ou *data flow graph* (DFG) (GAUTIER; BESSERON; PIGEON, 2007).

Por exemplo, 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. A Figura 4.3 ilustra ambos os exemplos em que pode-se substituir cada chamada de função pela criação de uma tarefa concorrente.

```
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);
```

Figura 4.3. Exemplos em que o paralelismo de tarefas pode ser aplicado para algoritmos recursivos (esquerda) e laços paralelos (direita).

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 4.4 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 quero que a execução inicie 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 usei a cláusula `firstprivate` 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.

4.3.2. Dependências de Dados

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. 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; BESSERON; PIGEON, 2007). O controle de execução é feito exclusivamente pelo fluxo de dados da aplicação e depende do modo de acesso descrito pela tarefa.

```
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 4.4. Exemplo de tarefas OpenMP para visitar elementos de uma lista encadeada.

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 4.5 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`).

4.3.3. Aceleradores

O padrão OpenMP 4.5 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

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

Figura 4.5. Exemplo simples de tarefas OpenMP com dependências de dados.

onde a execução do programa inicia e o *device* seria 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.

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 do `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 4.6 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 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.

4.4. Exemplos de Aplicações Científicas

Duas aplicações científicas foram consideradas para a inserção de diretivas paralelas. As próximas subseções descrevem as aplicações que e apresentam as formas como os trechos

```
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 4.6. Exemplo simples de uso de OpenMP para aceleradores.

de código foram paralelizados.

4.4.1. Aplicação de Meios Porosos

A alta eficiência do controle de secagem pode superar as perdas de grãos. Evitar a secagem excessiva ou insuficiente é o começo para evitar perdas por secagem. O gasto de muitos recursos para prever a temperatura ideal do grão, a taxa de secagem ao ar, a umidade do grão e o tempo de secagem são necessários para se obter uma melhor eficiência do processo, conforme ilustrado na Figura 4.7. Esses recursos, como experimentos, equipamentos e mão de obra, tornam o estudo experimental mais caro. Uma forma de reduzir esses investimentos é por meio de simulações numéricas. Soluções numéricas de equações da mecânica dos fluidos são utilizadas para representar, com alguns pressupostos de acordo com as condições iniciais e de contorno, fenômenos naturais e artificiais.

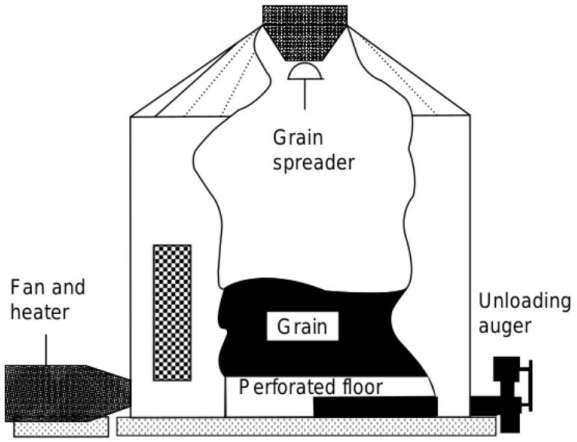

Figura 4.7. Processamento de grãos.

A secagem de grãos é um processo de transferência de calor e massa entre o grão e o ar. Há um movimento de energia do fluxo de ar quente através dos granéis, pelo processo de convecção, que se distribui rapidamente na massa do grão, vaporizando parte da água

do grão. Enquanto isso, a água dentro do grão é transferida pelo processo de difusão como um movimento fluido e como um processo de convecção na superfície úmida.

Considerando que a massa do grão é uma quantidade de espaços sólidos e vazios (orifícios) pelos quais um fluido pode passar, pode-se assumir a secagem do grão como um problema de meio aberto-poroso acoplado. A modelagem matemática e simulação computacional são amplamente utilizadas para descrever a convecção em um fluxo livre com um obstáculo poroso. A previsão da taxa de fluxo que passa através e ao redor de um meio poroso pode ser encontrada em muitos estudos na literatura. Ele usa a formulação da lei de Darcy e suas modificações atuais na parte porosa e a formulação de Navier-Stokes na parte aberta. No entanto, deve-se levar em consideração a mudança abrupta do fluxo livre e do meio poroso, criando uma zona de transição, conforme Figura 4.8.

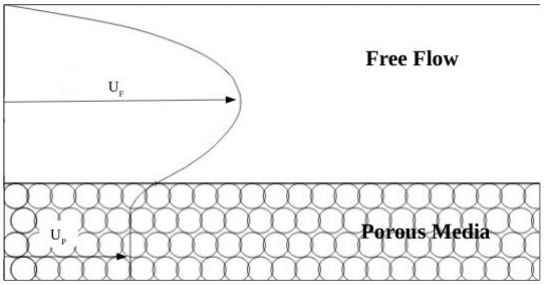

Figura 4.8. O fluxo em meio livre e em meio poroso

A aplicação aqui apresentada modela um problema de convecção através de um meio poroso cilíndrico, adotando uma abordagem de domínio único (OLIVEIRA, 2020). O problema clássico de um fluxo ao redor de um obstáculo de cilindro circular é amplamente estudado em engenharia, principalmente na forma de suprimir o derramamento de vórtices. Assim, um esquema de dinâmica dos fluidos computacional é implementado em FORTRAN usando Volume Finitos para simular e computar as soluções numéricas. O fluxograma do algoritmo é apresentado na Figura 4.9.

Atualmente, o trabalho em desenvolvimento, denominado projeto *Poros*, realiza esta resolução de forma sequencial, utilizando como base as equações de Navier-Stokes (CONSTANTIN; FOIAS, 1988). No entanto, o desempenho obtido fica aquém das necessidades de tempo e poder de processamento existentes, sendo desejada a otimização da execução na busca por ganhos de eficiência na resolução do problema. Dessa forma, este trabalho apresenta melhorias aplicadas ao código, com a utilização de paralelismo através da biblioteca OpenMP (CHANDRASEKARAN; JUCKELAND, 2017).

4.4.2. Método de Lattice-Boltzmann

O MLB é um método numérico iterativo discreto utilizado para a modelagem e simulação mesoscópica de fluxos de fluidos (SUCCI, 2001). A estrutura principal do algoritmo do MLB é constituido de um laço de repetição no qual são feitas operações de movimentação dos dados, simulando um fluxo de fluido. Tais operações consistem na propagação e relação das partículas, além de cálculos de valores macroscópicos e tratamento das condições

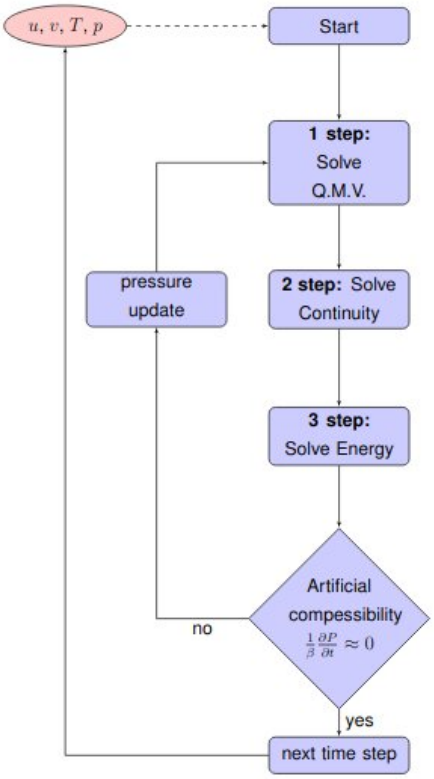

Figura 4.9. Fluxograma do algoritmo de simulação do meio poroso

de contorno (*Bounce Back*, conforme apresentado na Figura 4.10. O critério de parada do laço principal pode ser determinado pelo número de iterações ou por outro fator, como a estabilização do fluxo.

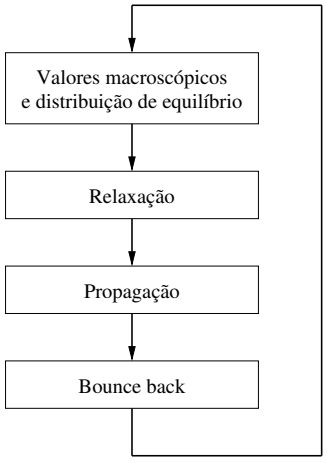

Figura 4.10. Algoritmo do MLB

Para a implementação foi adotado o modelo de reticulado mais utilizado do mé-

todo para o modo bidimensional, com 9 direções de propagação das partículas (CHEN; DOOLEN, 1998). Tal modelo é conhecido por D2Q9, sendo este ilustrado na Figura 4.11.

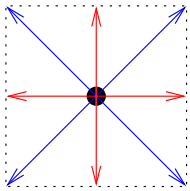

Figura 4.11. Modelos de reticulado bidimencional e tridimencional

A implementação feita consistiu em obter valores macroscópicos, tais como velocidade e pressão, para um fluxo de fluido através de um canal com obstáculos. Como parâmetro de entrada são repassadas algumas informações ao programa através de dois arquivos: um que contém informações genéricas tais como as propriedades macroscópicas e parâmetros de configuração, e o segundo que contém a estrutura de pontos do reticulado (limites e barreiras). Essas informações são armazenadas em duas estruturas de dados independentes.

O estudo de caso escolhido para avaliar a implementação paralela consiste em uma simulação de um fluxo de fluido cruzando canais com obstáculos. A distribuição dos obstáculos é composta de 5 barreiras dispostas ciclicamente ao longo do eixo x, conforme indicado na ilustração da esquerda da Figura 4.12. O tamanho de cada barreira é igual a metade do número de pontos que compõem os elementos da dimensão y. Já a distância entre cada uma das barreiras também é fixa, tendo-se utilizado para isso o valor de $1/5$ do tamanho total de pontos da dimensão x. Nos testes, variou-se o tamanho do reticulado, para avaliar a performance paralela.

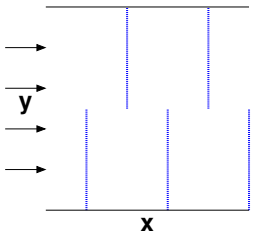

Figura 4.12. Disposição das barreiras no reticulado bidimensional

4.5. 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 tem 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.

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 paralelizáveis é fazer uso de ferramentas de perfilamento de aplicações. A ferramenta gprof (GRAHAM; KESSLER; MCKUSICK, 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 ganho de desempenho apresentado na Equação 2.

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

4.6. Paralelização de Poros

O algoritmo Poros é composto por um grande laço iterativo que percorre a transição de um tempo discretizado. 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 (P, U e V) do código. Nesta etapa iterativa são calculados as equações de Momento de *Quick Scheme* (`solve_U`, `solve_V`), equação de continuidade (`solve_P`)

e equação de Energia (`solve_Z`). Todas as rotinas dessas equações estão implementadas no arquivo `equations.f90`. As rotinas `solve_U` e `solve_V` representam o tempo respectivamente de 43% e 41% do tempo de execução total do código, restando 7% para `solve_Z` e 1% para `solve_P`. Existem ainda as funções `upwind_U` e `upwind_W` que demandam 2% do tempo de execução para cada uma.

As quatro funções executadas na etapa iterativa possuem trechos de código que podem ser paralelizados com OpenMP. Essencialmente há um padrão em cada função: há 4 laços aninhados que percorrem a representação bidimensional do domínio do problema. Assim, cada trecho (`do`) pode ser paralelizado com `!$omp parallel do`, com as devidas variáveis específicas indicadas com `private`. 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. A Figura 4.13 demonstra como o paralelismo de laços foi aplicado um trecho de código da rotina `solve_U`.

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

Figura 4.13. Implementação de laços paralelos usando a diretiva `parallel do` na função `solve_U`.

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 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).

4.7. Paralelização do Método de Lattice Boltzmann

A paralelização do Método de Lattice Boltzmann apresentada neste capítulo engloba essencialmente a utilização de diretivas do tipo `target` Cada uma das operações da etapa iterativa do método foi implementada como uma função específica. Desta forma, tem-se as funções `redistribute()`, `propagate()`, `bounceback()` e `relaxation()` que operam sobre os elementos do reticulado.

As quatro funções podem ser implementadas de maneira semelhante, fazendo uso da computação em GPU, através da diretiva `target`, variando apenas quais são as variáveis privadas em cada função. A Figura 4.15 apresenta a paralelização da função `bounceback()` usando a opção `target` de paralelismo. O opção `collapse(2)`

indica a junção dos 2 primeiros laços aninhados. Especificamente, esta função realiza ainda uma operação de redução.

```
1  #pragma omp target map(to: lx, ly, n, x, y)
2  #pragma omp teams distribute parallel for private(x, y) /
      collapse(2)
3  for (x = 1; x < lx - 1; x++) {
4   for(y = 1; y < ly - 1; y++) {
5    if (obst[x * ly + y] == true) {
6     int base = (x * ly + y) * n;
7     node[base + 1] = temp[base + 3];
8     node[base + 2] = temp[base + 4];
9     node[base + 3] = temp[base + 1];
10    node[base + 4] = temp[base + 2];
11    node[base + 5] = temp[base + 7];
12    node[base + 6] = temp[base + 8];
13    node[base + 7] = temp[base + 5];
14    node[base + 8] = temp[base + 6];
15   }
16  }
17 }
```

Figura 4.14. Implementação de tarefas usando a diretiva `target` para a função `bounceback()`.

Além das 4 funções executadas na etapa iterativa, uma função de verificação da solução numérica (`check_density()`) também pode ser invocada a qualquer momento da execução da etapa iterativa ou somente ao final da execução. A Figura 4.15 apresenta a paralelização da função `check_density()` usando a opção `teams` de paralelismo.

```
1  #pragma omp teams distribute parallel for private(x, y, _n)/
       reduction(+: n_sum) collapse(2)
2  for (x = 1; x < lx - 1; x++) {
3   for (y = 1; y < ly - 1; y++) {
4     int base = (x * ly + y) * n;
5    for (_n = 0; _n < n; _n++) {
6     n_sum = n_sum + node[base + _n];
7    }
8   }
9  }
```

Figura 4.15. Implementação de tarefas usando `teams` para a função `check_density()`.

Além da paralelização das chamadas, antes da etapa iterativa, é necessário co-

piar os dados para a GPU, conforme apresentado na Figura 4.16. Estas 3 estruturas de dados são manipuladas pelas funções chamadas a cada iteração e não é necessário fazer sincronização a cada iteração.

```
1 #pragma omp target data map(tofrom: temp[0:node_sz], node/
     [0:node_sz], obst[0:obst_sz])
2 {
3   for (time = 0; time < properties->t_max; time++) {
4     ...
5   }
6 }
```

Figura 4.16. Copia das 3 estruturas de dados utilizadas na etapa iterativa para a GPU.

4.8. 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 as características que o domínio do problema possui. 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 foram descritas duas aplicações 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 quando 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 multicore, quanto many-core, deixando de ser utilizado somente o tradicional paralelismo de laços através da combinação das diretivas `parallel` e `for`.

Referências

Advanced Micro Devices, Inc. *AMD EPYC 7H12*. 2021. páginas 3

CHANDRASEKARAN, S.; JUCKELAND, G. *OpenACC for Programmers: Concepts and Strategies*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2017. ISBN 0134694287. páginas 11

CHAPMAN, B.; MEHROTRA, P.; ZIMA, H. Enhancing openmp with features for locality control. In: CITESEER. *Proc. ECWMF Workshop" Towards Teracomputing-The Use of Parallel Processors in Meteorology*. Austrian: PSU, 1998. páginas 4

CHEN, S.; DOOLEN, G. D. Lattice Boltzmann Method for Fluid Flows. *Annual Review of Fluid Mechanics*, v. 30, p. 329–364, 1998. páginas 13

CONSTANTIN, P.; FOIAS, C. *Navier-Stokes Equations*. [S.l.]: University of Chicago Press, 1988. páginas 11

CORPORATION, I. *Intel Advanced Vector Extensions 512 (Intel AVX-512)*. 2021. páginas 3

CORPORATION, I. *Processador Intel Xeon Platinum 8380H*. 2021. páginas 3

FOSTER, I. *Designing and Building Parallel Programs: Concepts and tools for Parallel Software Engineering*. Reading, MA: Addison Wesley, 1995. páginas 6

GAUTIER, T.; BESSERON, X.; PIGEON, L. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: *2007 international workshop on Parallel symbolic computation*. Waterloo, Canada: ACM, 2007. p. 15–23. Disponível em: <https://hal.inria.fr/hal-00684843>. páginas 7

Gonçalves, R.; Girardi, A.; Schepke, C. Performance and energy consumption analysis of coprocessors using different programming models. In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. [S.l.: s.n.], 2018. p. 508–512. páginas 3

GRAHAM, S. L.; KESSLER, P. B.; MCKUSICK, M. K. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 17, n. 6, p. 120–126, jun. 1982. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/872726.806987>. páginas 14

NEC Corporation. *NEC SX-Aurora TSUBASA - Vector Engine*. 2021. páginas 3

NVIDIA. *GPU NVIDIA A100*. 2021. páginas 4

OLIVEIRA, D. P. de. *Fluid Flow Through Porous Media With The One Domain Approach: A Simple Model For Grains Drying*. 49 p. Dissertação (Mestrado) — Universidade Federal do Pampa, Alegrete, 2020. páginas 11

OPENMP. *The OpenMP API specification for parallel programming*. 2021. Disponível em: https://www.openmp.org. Disponível em: <https://www.openmp.org/>. páginas 4, 5, 6

SUCCI, S. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. New York, USA: Oxford University Press, 2001. ISBN 0-19-850398-9. páginas 11

TORELLI, J. C.; BRUNO, O. M. Programação paralela em smps com openmp e posix threads: um estudo comparativo. In: *Anais do IV Congresso Brasileiro de Computaçao (CBComp)*. São Carlos, SP: Instituto de Ciências Matemáticas e de Computação Universidade de São Paulo, 2004. v. 1, p. 486–491. páginas 4