 DOI: 10.5753/sbc.16630.8

Capítulo 2

Diretivas Paralelas de Programação[1]

Claudio Schepke - claudioschepke@unipampa.edu.br[2]
Vinícius Garcia Pinto - vinicius.pinto@furg.br[3]

Resumo

Este capítulo tem como objetivo abordar a paralelização de aplicações usando diretivas das interfaces de programação paralela OpenMP e OpenACC. O uso de diretivas tem sido a forma mais fácil de acelerar uma aplicação tanto em CPU como em GPU, sem a necessidade de conhecimentos profundos a respeito de programação paralela. Através de exemplos práticos de aplicações consideradas simples, mas não triviais como as vistas geralmente em cursos introdutórios, almeja-se apresentar os principais recursos para a execução concorrente de tarefas e de laços paralelos. As interfaces paralelas selecionadas também permitem o direcionamento da arquitetura-alvo, o que possibilita o uso de uma mesma interface tanto para CPU como para GPU.

[1] DOI: 10.5753/sbc.16630.8.2

[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). Realizou um pós-doutorado no Instituto Politécnico de Bragança, Portugal (2024-2025), trabalhando com programação paralela. É professor associado da Universidade Federal do Pampa (UNIPAMPA), campus Alegrete/RS desde 2012, ministrando disciplinas de arquitetura de computadores, sistemas operacionais, compiladores e processamento paralelo. Tem experiência na área de Ciência da Computação, com ênfase em Computação de Alto Desempenho, atuando principalmente nos seguintes temas: interfaces de programação paralela, aplicações científicas e computação em nuvem.

[3] Vinícius Garcia Pinto é doutor em Computação pelo Programa de Pós-Graduação em Computação da UFRGS em co-tutela com a Université Grenoble Alpes / França (2018); Mestre em Computação pelo Programa de Pós-Graduação em Computação da Universidade Federal do Rio Grande do Sul (2013). Bacharel em Ciência da Computação pela Universidade Federal de Santa Maria (2010). Foi professor da Faculdade São Francisco de Assis (2013-2014), da Universidade de Caxias do Sul (2018-2019) e professor substituto da Universidade Federal do Rio Grande do Sul (2019-2021) onde também realizou estágio de pós-doutorado. Desde 2021 é professor do Centro de Ciências Computacionais da Universidade Federal do Rio Grande, atuando em disciplinas de programação e sistemas operacionais.

2.1. Introdução

Programar paralelamente, isto é, gerar múltiplos fluxos de execução ou processos que executam colaborativamente, já foi considerado muito difícil. Para simplificar o desenvolvimento de aplicações de alto desempenho, pode-se fazer uso de diferentes interfaces de programação paralela, o que inclui bibliotecas de programação, linguagens de programação paralelas, extensões de linguagem, linguagens de domínio específico ou *frameworks* de desenvolvimento. No que tange à aceleração via execução concorrente de códigos sequenciais já existentes, o uso de diretivas paralelas de programação simplifica a etapa de escrita de código-fonte.

Diretivas são comandos ou palavras-chave utilizadas para a geração de código em tempo de pré-compilação. Isto é, antes que o código-fonte seja propriamente compilado, os termos inseridos nos códigos são substituídos e novas linhas de código são injetadas. No caso das diretivas (*pragmas*) paralelas, estas são convertidas para um código que gere, por exemplo, fluxos concorrentes de código (*threads*) que serão executados em arquiteturas multi-core ou GPU, por exemplo.

Neste contexto, este capítulo mostra como o desenvolvimento de programas pode beneficiar-se do uso de diretivas de programação paralelas. A Seção 2.2 irá apresentar as métricas de desempenho que vêm sendo utilizadas para avaliar os ganhos de desempenho no caso de execuções paralelas. Na sequência, são mostrados conceitos e exemplos de técnicas de programação usando OpenMP, na Seção 2.3, e OpenACC, na Seção 2.4. Por fim, na Seção 2.5, a conclusão discute as considerações finais sobre o assunto. Desta forma, a contribuição deste capítulo é oferecer uma introdução de forma intuitiva e fácil sobre como iniciar um programa paralelo, sem a necessidade de conceitos mais profundos de arquitetura de computadores e sistemas operacionais.

2.2. Métricas de desempenho

O uso de diretivas de compilação não necessariamente agrega alto desempenho a um código. Neste sentido, é importante fazer uso de métricas para avaliar o desempenho de programas, com o intuito de investigar o real ganho paralelo. Pela **Lei de Amdahl**, nem todo um programa pode ser paralelamente executado. Considerando o clássico conceito de Entrada, Processamento e Saída baseado no **Modelo de Von Neumann** de computação, operações de pré- e pós-processamento são normalmente seriais, pois envolvem o acesso ou escrita linear de informações, através da memória, de arquivos, por exemplo. Para alguns casos, esse armazenamento também poderia ser fragmentado e realizado concorrentemente, ainda que com limitações tecnológicas.

De todo modo, presume-se que a maior parte do tempo será demandada no processamento, muitas vezes composta de operações iterativas ou etapas independentes. No caso de **operações com laços**, se cada iteração depender linearmente da anterior, não será possível a quebra em paralelo das operações. Essa situação ocorre em dependências temporais. Todavia, mesmo em dependências temporais, internamente, em cada etapa iterativa, haverá a chance de que existam operações que poderão ser executadas concorrentemente sobre os dados. No caso de **etapas de processamento**, novamente podem haver dependências entre cada fase ou computações naturalmente sobre tarefas paralelas. Ainda que possam existir dependências entre as etapas, é possível dividir a computação

dos dados, gerando uma espécie de duto de processamento.

O **tempo de processamento** é talvez o primeiro fator a ser considerado para avaliar o desempenho das implementações. Em um mundo computacional multi-tarefa, vale o tempo real transcorrido e não apenas o tempo em que a CPU ficou alocada para um determinado processo. Ou seja, é preciso considerar formas de coletar o tempo de processamento. Gerenciadores de tarefas, monitores de processamento em GPU ou a execução de chamadas externas à execução de um programa são algumas das alternativas. No caso da execução do comando `time`, é possível obter os tempos de execução específicos da aplicação executada em modo usuário, modo sistema operacional e valores totais. Alguns programadores preferem utilizar a sua própria função baseada em chamadas de sistema Linux, como `gettimeofday`, `clock_gettime` ou `cpu_time` de bibliotecas de funções de relógio ou tempo disponíveis em linguagens como C e Fortran. Outros preferem fazer uso de funções ou rotinas encapsuladas nas próprias interfaces de programação paralela como `omp_get_wtime` de OpenMP ou a flag `pgi_acc_time` de OpenACC. É importante lembrar que as métricas podem ter uma determinada granularidade de coleta (microssegundos, por exemplo) ou cumulativamente retornar a soma dos tempos de execução de cada fluxo paralelo, o que pode não ser precisamente satisfatório para execuções curtas ou não ser o exato objetivo da medição.

Além do tempo, são considerados também como métricas o ganho de desempenho (*speedup*) e a eficiência computacional.

No primeiro caso, a ideia do **ganho de desempenho** é avaliar quantas vezes um programa paralelo é mais rápido em relação ao tempo de execução sequencial. Hipoteticamente, usando N recursos replicados, o ganho deveria ser de N, para o trecho em paralelo considerado. Mesmo considerando apenas os trechos que podem ser paralelizados, não é possível acelerar linearmente uma aplicação, especialmente quando um número maior de fluxos paralelos é utilizado. Isso decorre de custos inerentes com a instanciação de tarefas paralelas, da divisão irregular da carga de processamento e da própria natureza da computação das instruções, como níveis de cache, acesso concorrente à memória, maior número de operações de saltos de instruções. Este último, inclusive, pode prover um ganho levemente superlinear para execuções de entradas bastante específicas, considerando um pequeno número de fluxos de execução. Todavia, o mais natural é um comportamento de aceleração em escala logarítmica, tendendo a um limite e eventualmente caindo depois que a granularidade de cada tarefa paralela tornar-se muito pequena.

No segundo caso, a **eficiência** considera uma valoração percentual. Ou seja, a razão entre o tempo paralelo dividido pelo tempo sequencial e o número de recursos paralelos considerados resulta em um valor entre 0 e 1. Quanto mais próximo de 1, maior é a eficiência. Pelos mesmos motivos apresentados para o ganho paralelo, é impossível a eficiência permanecer constante à medida que o número de unidades de computação paralela aumenta. Assim, o comportamento da curva de eficiência tenderá a 0 para um número grande de fluxos de execução. Diante disso, o que se deseja em uma implementação paralela é adiar o máximo possível o declínio da eficiência à medida em que há mais processamento ocorrendo em paralelo.

Uma análise crítica que também precisa ser feita em relação à aceleração paralela é o tempo sequencial de referência a ser considerado. Alguns advogam medir apenas

a proporção entre os trechos que foram paralelizados. No entanto, isso é injusto, pois o que importa é reduzir o tempo total de processamento de uma aplicação. Por outro lado, o tempo de computação também não deveria ser comparado quando arquiteturas computacionais distintas são avaliadas.

Para solucionar esse problema, pode-se utilizar a noção de número de **operações de ponto flutuante por intervalo de tempo**. A escolha tradicional por operações com números reais deve-se à predominância delas na computação de propósito geral e, mais expressivamente, na computação científica. Através desta métrica de coleta, é possível comparar arquiteturas distintas como multicore e GPU.

Outra métrica muito em voga é a capacidade de processamento em relação ao **consumo energético**. A preocupação com a computação verde incorporou medições ou estimativas aproximadas do uso de energia em relação ao número de pontos flutuantes computados. Assim, a métrica de desempenho (flop) por Watt pode ser utilizada.

Outra forma de tratar a capacidade de desempenho de um programa é afixar o tempo de processamento de acordo com o valor sequencial e ajustar o **volume de processamento** para que o mesmo se adéque ao tempo sequencial. Por exemplo, se a complexidade computacional é quadrática, então é possível, em teoria, computar o dobro de dados tendo quatro vezes mais capacidade de processamento à disposição em relação ao volume de dados originais e tempo sequencial. Desta forma, o que se deseja é aumentar o tamanho do problema ou domínio a ser computado, de maneira a garantir a execução em um tempo aceitável.

Por fim, cabe citar as informações estatísticas realizadas por **contadores de hardware** como um meio mais fino de avaliar o desempenho de uma aplicação. Ou seja, é interessante coletar a quantidade de cada tipo de operação elementar a ser executada pelo hardware, de maneira a comparar as execuções sequenciais e paralelas. Em um cenário ideal, o número e o tipo de instruções mantêm-se bastante próximos dos valores da execução sequencial. Todavia, é natural um acréscimo no somatório do número total de instruções paralelas, uma vez que determinadas operações terão que ser computadas em cada fluxo paralelo distinto. Somado a isso, tem-se os níveis de cache e as replicações de alguns desses níveis nos diferentes cores. No caso de execuções paralelas, espera-se também distribuir a carga de computação da maneira mais **balanceada** possível, especialmente se o hardware for homogêneo.

Ainda em relação à estatística, é importante usar as melhores práticas recomendadas. Assim, aspectos como o **número mínimo de execuções** dos experimentos para garantir um percentual de **confiança** aceitável, **média**, **mediana** e **desvio padrão** são passos que precisam ser seguidos e cujos valores precisam ser coletados. Outros pontos que envolvem a coleta de dados nos experimentos têm a ver com a execução dos testes. Por exemplo, que abordagem deve ser seguida: executar várias vezes consecutivamente sem intervalos, excluindo da coleta a primeira execução que foi gerada "a frio" ou intercalar cada execução de maneira que cada teste seja feito "isoladamente", como de fato aconteceria na realidade? No primeiro caso, os valores estatísticos de controle estariam menos discrepantes. No segundo caso, os experimentos são mais condizentes com o que aconteceria em um cenário real. Uma maior discussão a respeito de boas práticas de experimentação em computação paralela pode ser vista em (PINTO; NESI; SCHNORR,

2020).

2.3. Diretivas de OpenMP

OpenMP (*Open Multi-Processing*) é uma API aberta focada na escrita de aplicações paralelas. As diferentes versões da API são padronizadas pelo *OpenMP Architecture Review Board* e são suportadas por diversos compiladores. Desde a versão 1.0 lançada ainda nos anos 1990, a API tem sido incrementada, adicionando suporte a novos recursos de *Sardware* como arquiteturas NUMA, vetorização e GPUs, bem como novos recursos de *Software*, como paralelismo de tarefas e mecanismos de sincronização. A versão mais recente é a 6.0 lançada em novembro de 2024 (OpenMP, 2024). Um aspecto importante a ser destacado é que as funcionalidades mais recentes adicionadas na especificação nem sempre são plenamente suportadas pelos compiladores mais populares como GCC e LLVM/Clang. No momento da escrita deste capítulo, o Clang[4] e o GCC[5] suportam plenamente a versão 4.5 do OpenMP e apenas parcialmente as versões mais recentes como 5.0, 5.1 e 6.0. Neste material, vamos dar maior ênfase naquelas funcionalidades da API que já estão consolidadas e com suporte integral nos mais diversos compiladores.

Podemos interagir com a API do OpenMP de três maneiras distintas:

- adicionando diretivas de compilador;
- fazendo chamadas à biblioteca de rotinas; e
- definindo variáveis de ambiente.

Do ponto de vista da pilha de Software, pode-se dizer que OpenMP fica localizado numa camada intermediária. No topo temos a aplicação do usuário final, enquanto na base temos o suporte do Sistema Operacional à funcionalidades como *threads* e primitivas de sincronização de memória. A execução de um programa OpenMP faz uso de um ambiente de execução conhecido como *OpenMP Runtime Library* que é responsável por questões como a criação e o gerenciamento das *threads*, sincronização e escalonamento. Cada compilador usualmente implementa o seu próprio ambiente de execução como o `libgomp` do GCC ou o `libomp` do LLVM.

OpenMP surgiu como uma ferramenta para paralelização incremental de códigos já existentes. Dessa forma, vamos começar o estudo das diretivas com o mais simples e primitivo dos programas: um "Olá Mundo", ilustrado no Código 2.1. Neste exemplo temos um programa em linguagem `C`. Cabe ressaltar que OpenMP também suporta as linguagens `C++` e `Fortran`. Alguém familiarizado com a programação em `C` notará rapidamente que a versão paralela com OpenMP é muito similar a um "Olá Mundo" tradicional (e sequencial). As diferenças se resumem às linhas de código 2 e 5. Ambas são diretivas de pré-processamento. A primeira é o já conhecido `#include` que serve para incluir um arquivo de cabeçalho. Já a segunda (`#pragma omp parallel`) é uma diretiva específica do OpenMP e é utilizada para delimitar uma região paralela. Isto é, o trecho de código associado a esta diretiva será executado em paralelo por múltiplas *threads*. Antes

[4]<https://clang.llvm.org/docs/OpenMPSupport.html#openmp-6-0-implementation-details>
[5]<https://gcc.gnu.org/projects/gomp/#implementation-status>

de executarmos tal código, precisamos compilá-lo adicionando a *flag* `fopenmp` à linha de compilação como, por exemplo: `gcc ola-mundo.c -fopenmp`.

```
1 #include<stdio.h>
2 #include<omp.h>
3 int main (){
4
5  #pragma omp parallel
6  printf("Ola Mundo!\n");
7
8 }
```

Código 2.1. Ola Mundo Paralelo Com OpenMP

Em seguida, podemos executar tal programa e observar sua possível saída:

```
Ola Mundo!
Ola Mundo!
Ola Mundo!
Ola Mundo!
```

Nosso programa exibiu quatro vezes a mensagem `"Ola Mundo!"`. Isto aconteceu pois o `printf` estava associado a uma região paralela. Logo, ele foi executado, em paralelo, por várias *threads*. No exemplo acima, podemos deduzir que o número de *threads* usado foi quatro.

O número de *threads* a ser utilizado na execução é um dos parâmetros chave a ser configurado em um programa paralelo. O número ideal usualmente depende de várias variáveis como a carga de trabalho, a estrutura do algoritmo utilizado e a interação com outras bibliotecas paralelas. Por padrão, na ausência de qualquer outra informação, OpenMP utiliza como número de *threads* o número de CPUs virtuais/lógicas disponíveis no sistema. Caso necessário, este valor pode ser alterado de pelo menos três maneiras. A primeira delas não requer alterar o código e pode ser feita para a execução do programa, ao setar a variável de ambiente `OMP_NUM_THREADS`. A segunda é fazendo uso da chamada de rotina `omp_set_num_threads()` diretamente no código do programa. Isto valerá para as regiões paralelas subsequentes. Já a terceira consiste em especificar o número de *threads* diretamente na diretiva `#omp parallel` adicionando a cláusula `num_threads`.

Um dos principais desafios ao se paralelizar um programa diz respeito ao gerenciamento do acesso (possivelmente concorrente) às variáveis. Em programas sequenciais mais simples costumamos levar em conta que as variáveis podem ser globais ao programa inteiro ou locais a uma determinada função ou região. Isto vai determinar os trechos de código onde a variável é acessível (escopo) e seu tempo de vida. Quando adicionamos paralelismo, também acrescentamos mais níveis de escopo e tempo de vida já que podemos ter variáveis que são totalmente privadas a cada *thread*, variáveis que são compartilhadas (i.e., globais) entre *threads* mas locais a uma determinada parte do programa, ou ainda

variáveis que em dado momento são privadas mas que ao final são reduzidas a um valor único compartilhado.

Vejamos o comportamento padrão do OpenMP para algumas situações. São compartilhadas, isto é, há uma única instância, as variáveis declaradas fora de uma região paralela, as variáveis estáticas e as variáveis alocadas dinamicamente (e.g., uso de `malloc`). São privadas, isto é, há uma instância para cada *thread*, as variáveis declaradas dentro de uma região paralela e as variáveis de funções chamadas dentro de uma região paralela.

Há situações, porém, em que o comportamento padrão é inadequado. Nestes casos, podemos alterar tal comportamento acrescentando informações a uma diretiva utilizando cláusulas. A cláusula `shared` define que a variável em questão será compartilhada entre todas as *threads* que executam a região paralela em questão. Já a cláusula `private` indica ao OpenMP que devem ser criadas novas instâncias da variável para que cada *thread* acesse apenas a sua cópia privada. Por padrão, estas novas instâncias não são inicializadas. Caso necessário, podemos inicializar as cópias com o valor original utilizando a variação `firstprivate`.

Até aqui, vimos que o código interno a uma região paralela é executado igualmente por todas as *threads*. Porém, há várias situações em que alguns trechos de código não necessitam ser executados por todas as *threads* (e.g., inicialização, leitura de entradas) ou que não podem ser executados mais de uma vez (e.g., envio de mensagens, alocação/-liberação de memória) ou ainda que podem executar mais de uma vez mas não simultaneamente. Nestes casos, podemos empregar as diretivas `single`, `master`, `masked` e `critical`. Nos dois primeiros casos, os comandos associados serão executados uma única vez porém há duas diferenças importantes. Na diretiva `single`, o comando associado poderá ser executado por qualquer uma das *threads*, i.e, a primeira a alcançar a diretiva, enquanto a diretiva `master` restringe a execução à *thread* cujo identificador é 0. A segunda diferença é que na diretiva `single` há uma barreira implícita, que obriga as demais *threads* a aguardarem até que a execução dos comandos associados seja concluída, o que não ocorre no caso da diretiva `master`. Importante ressaltar também que a diretiva `master` está marcada como obsoleta desde a versão 5.1. Sugere-se que seja substituída pela diretiva `masked` que tem a mesma semântica desde que seja usada sem uma cláusula `filter`. A vantagem é que, usando `masked filter(<id>)`, podemos restringir a execução a outras *threads* que não a *thread* de identificador 0. A expressão associada à cláusula `filter` aceita construções mais complexas desde que resultem em um inteiro. Por exemplo, a estrutura `masked filter(omp_get_thread_num()%3)` restringe a execução dos comandos às *threads* 0, 1 e 2. Por fim, vamos analisar o comportamento da diretiva `critical`. Esta diretiva restringe a execução dos comandos temporariamente a apenas uma *thread* de cada vez. Ou seja, permite definir os comandos associados como uma região crítica.

O Código 2.2 apresenta uma nova versão do programa "Olá Mundo" visto anteriormente. Este código exemplifica o uso de algumas das estruturas que discutimos. Na linha 8 vemos que o número de *threads* foi definido para três, utilizando a chamada de rotina `omp_set_num_threads`. Na linha 10 definimos uma região paralela que engloba o bloco de código que começa na linha 11 e termina na linha 20. Todo este trecho executará em paralelo nas três *threads*, exceto o comando da linha 17 que é precedido

pela diretiva `single`. Vamos agora analisar o escopo das três variáveis declaradas no código. Inicialmente, na linha 6, foram declaradas as variáveis `id` e `total`. A linha 6 está fora da região paralela, logo, por padrão, ambas variáveis seriam compartilhadas entre as *threads*. Porém, na linha 10, usamos as cláusulas `shared` e `private`. A cláusula `shared(total)` reforça que a variável `total` continuará sendo única e compartilhada entre todas as *threads* enquanto a cláusula `private(id)` altera o comportamento padrão para a variável `id`. Dessa forma, OpenMP criará uma nova variável `id` para cada *thread*, apesar de possuírem o mesmo nome. O fato é que dentro da região paralela teremos três novas variáveis, cada uma delas sendo privada a uma dada *thread*. Na linha 11, temos a declaração da variável `x`. Como tal declaração acontece já dentro da região paralela (linhas 11 a 20), cada *thread* declarará a sua variável `x` de uso privativo.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<omp.h>
4
5  int main (){
6   int id, total;
7
8   omp_set_num_threads(3);
9
10  #pragma omp parallel shared(total) private(id)
11  {
12   id = omp_get_thread_num();
13
14   int x = rand() % 100;
15
16   #pragma omp single
17   total = omp_get_num_threads();
18
19   printf("Ola Mundo! (thread %d de %d sorteou %d) \n", /
        id, total, x);
20  }
21  printf("Tchau!\n");
22 }
```

Código 2.2. Nova versão do Olá Mundo Paralelo Com OpenMP

Com isso, temos uma possível saída para o programa anterior:

```
Ola Mundo! (thread 2 de 3 sorteou 86)
Ola Mundo! (thread 0 de 3 sorteou 77)
Ola Mundo! (thread 1 de 3 sorteou 83)
Tchau!
```

Esta não é a única resposta possível uma vez que a maneira como o Sistema Operacional escalona as *threads* pode mudar de uma execução pra outra. Como o número de

threads foi configurado em três, teremos a mensagem "Ola Mundo" sendo impressa três vezes com valores distintos para `id` e `x`. Cada *thread* descobre seu identificador e sorteia um número inteiro. Como cada uma delas possui instâncias locais das variáveis `id` e `x`, não há conflitos de escrita concorrente. A variável `total` é compartilhada entre as três *threads*. Mas graças à diretiva `single`, seu valor sempre será consistente, uma vez que é escrito por apenas uma *thread* e porque existe uma barreira implícita obrigando as demais *threads* a não prosseguirem até a conclusão da execução do comando de escrita. Cabe ressaltar que, leituras concorrentes à variável `total`, como temos na linha `19`, não geram conflitos, pois não fazem alteração na mesma. Por fim, temos a impressão de uma única mensagem de "Tchau". Isto ocorre pois o comando da linha 21 está fora da região paralela. Por consequência, neste momento o código está sendo executado apenas pela *thread* principal do programa.

O paralelismo frequentemente é empregado em programas cuja execução é dominada pelo tempo necessário para efetuar operações custosas contidas em laços de repetição. Para tratar destes casos, OpenMP provê a diretiva `omp for`. Dentre as diretivas de divisão de trabalho de OpenMP, esta é a mais conhecida e amplamente utilizada (QUEVEDO et al., 2024). Ela consiste em dividir as iterações do laço entre as diversas *threads*. Com isso, várias iterações podem ser processadas em paralelo por diferentes *threads*. É importante ressaltar que cabe a quem programa analisar se não há dependências que impeçam a execução das iterações em paralelo. O Código 2.3 ilustra um exemplo desta diretiva. A diretiva `for` da linha 6 provoca o particionamento das iterações do laço das linhas 7 e 8, entre as quatro *threads* da região paralela. Por fins didáticos, na linha 8 fizemos cada *thread* escrever o respectivo `id` na posição do vetor em que esta trabalhou. Ao observar a saída do programa, notamos que as *threads* com `id` 0 e 1 receberam cada uma 8 iterações e, por consequência, escreveram em 8 posições do vetor. Já as *threads* com `id` 2 e 3 receberam 7 iterações cada.

```
1  #pragma omp parallel private(id) num_threads(4)
2  {
3   id = omp_get_thread_num();
4   printf("Thread %d executando!\n", id);
5
6   #pragma omp for
7   for(int i = 0; i < MAX; i++)
8    v[i] = id;
9  }
10 for(int i = 0; i < MAX; i++)
11  printf("%d ", v[i]);
```

Código 2.3. Diretiva de divisão de Trabalho `for`

```
Thread 2 executando!
Thread 1 executando!
Thread 3 executando!
Thread 0 executando!
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3 3 3
```

A diretiva `for` opcionalmente pode ser estendida com a cláusula `schedule`, que permite controlar se a distribuição das iterações entre as *threads* será estática (`static`) ou dinâmica (`dynamic`), bem como definir o tamanho (máximo) do subconjunto de iterações distribuído. Veja, no Código 2.4, um trecho que exemplifica o uso da diretiva `schedule` no particionamento das *threads* e o reflexo na saída do programa.

```
#pragma omp for schedule(static, 3)
for(int i = 0; i < MAX; i++)
 v[i] = id;
```

Código 2.4. Diretiva schedule

```
0 0 0 1 1 1 2 2 2 3 3 3 0 0 0 1 1 1 2 2 2 3 3 3 0 0 0 1 1 1
```

Outra situação recorrente em programas paralelos é aquela em que cada *thread* calcula localmente uma parte da solução e estas partes precisam ser combinadas para que se obtenha a solução integral. Tal situação é tratada com uma operação de redução, que no OpenMP é provida pela diretiva `reduction`. Uma operação de redução aplica um operador sobre uma variável. Os operadores mais frequentes são soma (+), multiplicação (/), máximo (`max`) e mínimo (`min`). O Código 2.5 a seguir ilustra o uso da diretiva `reduction` para computar um somatório de valores. Cada *thread* terá uma instância privada da variável `total`, que será inicializada com o elemento neutro da operação em questão, que neste caso é 0. Todas as atualizações parciais ocorrem na instância privada. Ao final, cada valor computado localmente é incorporado por OpenMP à variável original de maneira segura sem que seja necessária a inclusão explícita de uma diretiva de sincronização como `critical`.

```
#pragma omp for reduction(+ : total)
for(int i = 0; i < MAX; i++){
 v[i] = id;
 total += v[i];
}
```

Código 2.5. Exemplo de uma operação de redução

Além da paralelização de laços, OpenMP também suporta o paralelismo de tarefas com a diretiva `task`. Neste modelo não há um mapeamento exato e fixo do trabalho às *threads*. A ideia é desacoplar o algoritmo da plataforma, definindo tarefas, ou seja, trechos de código que podem executar em paralelo. Além do código, cada tarefa está associada a um ambiente de dados. Na ausência de pontos de sincronização ditos "globais", as tarefas podem ser executadas em qualquer ordem e por qualquer *thread*. Uma prática usual é criar uma grande quantidade de tarefas que serão posteriormente consumidas pelas *threads*.

Embora o suporte inicial à tarefas tenha sido adicionado em 2008 na versão 3.0 da especificação, seu uso ainda permanece menos frequente que as diretivas de laços paralelos. Algumas categorias de algoritmos como divisão e conquista, operações em matrizes por blocos e busca em grafos podem ser mais facilmente paralelizados utilizando

o paralelismo de tarefas. O trecho apresentado no Código 2.6 ilustra o funcionamento das diretivas `task` e `taskwait`. A diretiva `task` (linhas 1 e 6) é usada para definir duas novas tarefas. A primeira consiste do código entre as linhas 2 e 5 e a segunda do código entre as linhas 7 e 10. As duas novas tarefas, assim como o comando da linha 11, podem ser executados em qualquer ordem por qualquer *thread* (veja a seguir uma saída possível). O comando da linha 13 faz uso das variáveis `x` e `y`, que são escritas pelas tarefas criadas anteriormente. Dessa forma, antes de executá-lo precisamos garantir que tais tarefas já tenham terminado. Esta sincronização é obtida por meio da diretiva `taskwait`, da linha 12, que bloqueia a execução até que todas as tarefas criadas anteriormente no mesmo escopo já tenham terminado. Caso necessário, é possível estender a diretiva `task` com cláusulas como `if`, para limitar a criação de novas tarefas ao atendimento de alguma condição, e `priority` que permite dar ao ambiente de execução alguma sugestão de quais tarefas podem ter sua execução priorizada. A prioridade é dada por um número inteiro. Quanto maior este valor, mais prioritária é a tarefa sugerindo ao ambiente de execução que ela deve ser executada antes daquelas cujo valor de prioridade é menor.

```
1  #pragma omp task
2  {
3   printf("tarefa A!! \n");
4   x = 123;
5  }
6  #pragma omp task
7  {
8   printf("tarefa B!! \n");
9   y = -10;
10 }
11 printf("outra mensagem qualquer\n");
12 #pragma omp taskwait
13 printf("as duas tarefas acabaram e x*y=%d\n", x*y);
```

Código 2.6. Exemplo de uso da diretiva `task`

```
outra mensagem qualquer
tarefa B!!
tarefa A!!
as duas tarefas acabaram e x*y=-1230
```

Outra maneira de se impor alguma ordem na execução das tarefas é por meio da cláusula `depend`. Esta cláusula permite especificar para cada tarefa quais dados são lidos e quais dados são escritos. Com base nestas informações combinadas com a ordem de criação das tarefas, OpenMP pode inferir quais tarefas são independentes e, portanto, podem executar em paralelo e quais não podem. Veja no Código 2.7 uma versão alternativa que utiliza `depend` para descrever como os dados compartilhados são acessados por cada tarefa. As diretivas das linhas 1 e 6 agora especificam quais tarefas produzem os dados `x` e `y`. Já a diretiva da linha 12 especifica que a tarefa a seguir consome os dados `x` e `y`. Logo, tal tarefa não pode ser executada antes que tais dados já tenham sido produzidos. Além

dos parâmetros `in` e `out`, a cláusula `depend` também pode ser usada com o parâmetro `inout` para os casos em que um dado é tanto lido quanto escrito. Uma discussão mais detalhada sobre o paralelismo de tarefas em OpenMP pode ser encontrada em (NESI et al., 2021). Complementarmente ao paralelismo de tarefas, formato das diretivas em Fortran, bem como exemplos e avaliações comparativas das diretivas de OpenMP podem ser encontradas em (SCHEPKE; LUCCA; LIMA, 2023).

```
1  #pragma omp task depend(out:x)
2  {
3   printf("tarefa A!! \n");
4   x = 123;
5  }
6  #pragma omp task depend(out:y)
7  {
8   printf("tarefa B!! \n");
9   y = -10;
10 }
11 printf("outra mensagem qualquer\n");
12 #pragma omp task depend(in:x) depend(in:y)
13 printf("as duas tarefas acabaram e x*y=%d\n", x*y);
```

Código 2.7. Exemplo de uso da diretiva `depend`

2.4. Diretivas de OpenACC

Embora tanto OpenMP como OpenACC possam gerar código tanto para CPU como para GPU, tradicionalmente associa-se a segunda interface à programação paralela para GPUs. Quando se fala em programação para GPUs, logo tem-se a ideia de que é preciso fazer uso de CUDA. Todavia, esta interface torna muito explícitas as características do hardware e exige a compreensão de noções da arquitetura e do modelo de programação. Algo similar acontece com a interface OpenCL. Com isso, OpenACC torna-se uma boa alternativa de abstração das camadas inferiores, possibilitando ver apenas os aspectos de aplicação.

OpenACC é uma API desenvolvida em 2011 para programação em arquiteturas heterogêneas com aceleradores. O modelo de programação usa diretivas de alto nível, assim como o OpenMP, para expressar as áreas ou blocos paralelos (CHANDRASEKARAN; JUCKELAND, 2017). A versão mais recente da API do OpenACC é a 3.3, lançada em novembro de 2022.

A API está disponível para as linguagens C/C++ e FORTRAN (KIRK; WENMEI, 2016). Tal qual OpenMP, OpenACC possui diretivas padronizadas por `#pragma acc` para C/C++, seguida de `[atributos]` opcionais. Essas diretivas especificam *loops* e blocos de código que podem ser enviados da CPU para um acelerador. A seguir, são descritas algumas diretivas compatíveis com C/C++ e Fortran.

Antes de mais nada, é preciso lembrar que a programação em GPU envolve o uso de *Non-Uniform Memory Access* (NUMA). Isto é, há pelo menos duas memórias distintas: a que a CPU acessa e a que a GPU acessa, que inclusive podem ser fabricadas com

diferentes tecnologias. Por mais que haja abordagens de unificar este acesso à memória, de modo geral é preciso fazer a movimentação (via placa-mãe) entre a memória principal e a da GPU. Para tanto, pode-se utilizar `data copy`, como diretiva para a cópia de dados entre CPU e GPU, garantindo a sincronização dos dados, e `present`, como diretiva específica para o compilador "estar ciente" de que as variáveis estão presentes na memória da GPU e que não é necessário fazer cópia. As diretivas `copyin` e `copyout` são opções similares, mas cuja transferência de dados ocorre apenas para GPU e da GPU para a CPU. Também é possível usar a diretiva `create`, para apenas alocar memória na GPU, sem a cópia de dados. Por fim, pode-se utilizar `enter data` ou `exit data`, como uma diretiva que garanta a presença e término de existência dos dados, ou formalmente define o início e o fim de uma vida útil de dados não estruturados.

Em relação ao paralelismo, há algumas diretivas bastante úteis. A diretiva `kernels` permite que o compilador tome as decisões de paralelismo. A diretiva `parallel` impõe ao compilador que o trecho de código indicado na sequência deve ser paralelizado. Dependendo do uso, a diretiva pode produzir resultados equivocados. Já a diretiva `loop independent` especifica que o laço indicado pode ser paralelizado e ainda executado independentemente. Algumas dessas diretivas possuem outras opções parecidas e com funcionalidades bastante próximas (LUCCA, 2020).

Um exemplo de utilização de exploração do paralelismo de laços está apresentado no Código 2.8.

```
1 !$acc parallel loop collapse(2)
2   DO i=2,imax
3     DO j=2,jmax
4       a(i,j) = b(i,j) * c(i,j)
5     ENDDO
6   ENDDO
7 !$acc end parallel
```

Código 2.8. Exemplo do uso da diretiva `!$acc parallel loop`

Apesar das interfaces OpenMP, OpenACC e OpenMP Target ((LIMA; SCHEPKE; LUCCA, 2021)) utilizarem diretivas para especificar o paralelismo, cada interface tem suas características e diretivas, conforme apresentado em (LUCCA, 2020). A Tabela 2.1 apresenta de forma resumida uma comparação da equivalência entre a sintaxe do OpenMP clássico com diretivas que são executadas em CPU, do OpenACC e do OpenMP Target para execução de tarefas em GPU. As APIs diferem em sua abordagem para especificar a paralelização. Como todas as versões utilizam diretivas, a proposta é especificar as equivalências entre as APIs. Embora as diretivas sejam semelhantes em cada API, o paralelismo é especificado de uma forma diferente (SILVA et al., 2022).

2.5. Conclusão

Injetar diretivas é o meio mais simples de produzir operações paralelas, sem a necessidade de conhecer diversos aspectos específicos relacionados à arquitetura de computadores e ao gerenciamento de fluxos de execução. Isso não significa que magicamente o programa

Tabela 2.1. Relação entre as diretivas de OpenMP, OpenACC e OpenMP Target para GPU

OpenMP CPU	OpenACC	OpenMP Target (GPU)
omp parallel	acc parallel	omp target
omp parallel for	acc loop gang	omp teams distribute
-	acc loop vector	omp parallel for
-	acc data copyin(<var>)	omp data map(to:<var>)
-	acc data copyout(<var>)	omp data map(to:<var>)
-	acc data create(<var>)	omp data map(alloc:<var>)
omp parallel simd	-	omp parallel simd
omp task	-	omp task
-	routine	-
-	acc update	target update

resultante produzirá alto desempenho. Se assim o fosse, ambientes de auto-paralelização poderiam existir e serem adotados.

Assim, após conhecer como as diretivas funcionam e os resultados que elas produzem, é importante saber identificar em que trecho de código vale a pena aplicar a paralelização e qual abordagem produziria um efeito de ganho de desempenho mais expressivo. O assunto aqui apresentado está longe de ser esgotado, e possui lacunas, especialmente na questão de como identificar em códigos fonte, onde vale a pena buscar uma paralelização. Todavia, as ferramentas foram expostas, basta agora praticar.

Referências

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

KIRK, D. B.; WEN-MEI, W. H. *Programming massively parallel processors: a hands-on approach*. [S.l.]: Morgan Kaufmann, 2016.

LIMA, J. V. F.; SCHEPKE, C.; LUCCA, N. Além de Simplesmente: #pragma omp parallel for. In: CHARãO, A.; SERPA, M. da S. (Ed.). *Minicursos da XXI Escola Regional de Alto Desempenho da Região Sul*. Porto Alegre/RS: Sociedade Brasileira de Computação, 2021. chp. 4, p. 86–103.

LUCCA, C. S. N. Programando Aplicações com Diretivas Paralelas. In: BOIS, M. C. A. D. (Ed.). *Minicursos da XX Escola Regional de Alto Desempenho da Região Sul*. Porto Alegre: Sociedade Brasileira de Computação, 2020. p. 89–104. ISBN https://doi.org/10.5753/sbc.4400.9.5.

NESI, L. L. et al. Desenvolvimento de Aplicações Baseadas em Tarefas com OpenMP Tasks. In: CHARãO, A.; SERPA, M. (Ed.). *Minicursos da XXI Escola Regional de Alto Desempenho da Região Sul*. Sociedade Brasileira de Computação - SBC, 2021. p. 131–152. ISBN 9786587003504. Available from Internet: <http://dx.doi.org/10.5753/sbc.6150.4.6>.

OpenMP. *OpenMP Application Program Interface Version 6.0*. OpenMP Architecture Review Board, 2024. Available from Internet: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf>.

PINTO, V. G.; NESI, L. L.; SCHNORR, L. M. Boas Práticas para Experimentos Computacionais de Alto Desempenho. In: BOIS, A. D.; CASTRO, M. (Ed.). *Minicursos da XX Escola Regional de Alto Desempenho da Região Sul*. Sociedade Brasileira de Computação - SBC, 2020. p. 1–19. ISBN 9786587003504. Available from Internet: <https://doi.org/10.5753/sbc.4400.9.1>.

QUEVEDO, C. et al. An Empirical Study of OpenMP Directive Usage in Open-Source Projects on GitHub. In: *Anais do XXV Simpósio em Sistemas Computacionais de Alto Desempenho*. Porto Alegre, RS, Brasil: SBC, 2024. p. 144–155. ISSN 0000-0000. Available from Internet: <https://sol.sbc.org.br/index.php/sscad/article/view/30993>.

SCHEPKE, C.; LUCCA, N.; LIMA, J. V. F. Diretivas Paralelas de OpenMP: Um Estudo de Caso. In: PADOIN, E. L.; GALANTE, G.; RIGHI, R. (Ed.). *Minicursos da XXIII Escola Regional de Alto Desempenho da Região Sul*. Porto Alegre: Sociedade Brasileira de Computação, 2023. ISBN https://doi.org/10.5753/sbc.11938.7.1.

SILVA, H. U. da et al. Parallel OpenMP and OpenACC Porous Media Simulation. *The Journal of Supercomputing*, 2022. Available from Internet: <https://doi.org/10.1007/s11227-022-05004-2>.