Capítulo 2

Como programar aplicações de alto desempenho com produtividade

Álvaro Luiz Fazenda e Denise Stringhini

Abstract

The development of scientific applications that use high performance computing resources historically is considered complex and specialized. At first, the programmer must know the characteristics of the problem in order to perform its computational modeling. Considering the use of parallel hardware, it must still know its architectural characteristics, as well as paradigms, parallel languages and possible frameworks to be used. This chapter presents the basic and intermediate principles of parallel directive-based programming, applicable to problems commonly encountered in scientific applications, such as numerical modeling and computational intelligence, for example. Directive-based programming techniques are presented using OpenMP for multicore computers and OpenACC for machines equipped with GPU-type accelerators.

Resumo

O desenvolvimento de aplicações científicas que se utilizam de recursos de computação de alto desempenho historicamente é considerado complexo e especializado. A princípio, o programador deve conhecer as características do problema de forma a realizar sua modelagem computacional. Considerando o uso de um hardware paralelo, ele ainda deve conhecer suas características arquiteturais, assim como paradigmas, linguagens paralelas e possíveis frameworks a serem empregados. O presente capítulo apresenta os princípios básicos e intermediários da programação paralela baseada em diretivas, aplicáveis a problemas comumente encontrados em aplicações científicas, tais como modelagem numérica e inteligência computacional, por exemplo. Serão apresentadas técnicas de programação por diretivas usando OpenMP para computadores *multicore* e OpenACC para máquinas equipadas com aceleradores do tipo GPU.

2.1. Introdução

Nas últimas décadas tem-se observado uma crescente necessidade no uso plataformas de hardware e software capazes de processar grandes volumes de dados e executar cálculos complexos. O cenário de HPC (*High Performance Computing* ou Processamento de Alto Desempenho - PAD) é composto de uma grande variedade de tipos e de plataformas disponíveis, portanto uma compreensão das características básicas destes sistemas se faz necessária para a tomada de decisão em qualquer tipo de investimento na área.

Sistemas de alto desempenho são capazes de processar grandes volumes de dados e executar cálculos complexos. Por exemplo, pode-se pensar em uma aplicação para previsão de tempo. Este tipo de aplicação, normalmente, divide a atmosfera em uma grande quantidade de volumes tridimensionais e aplica a cada volume uma série de cálculos físicos, baseados em modelagem matemática, utilizando-se de determinados dados de entrada e, repetindo os cálculos de forma iterativa, conseguem determinar o estado de várias propriedades físicas, permitindo assim prever o estado futuro da atmosfera. Quanto menor o volume tridimensional utilizado, mais precisa a previsão tenderá a ser, porém a quantidade de volumes será maior para uma mesma área de interesse e, portanto, mais demanda computacional que implica em mais tempo de processamento, considerando um mesmo recurso computacional. Dessa forma, pode-se imaginar um caso onde se pode obter alta precisão nas informações geradas, porém, finalizada em tempo proibitivo para utilidade prática. Este tipo de aplicação, assim como qualquer outra que necessite de respostas mais rápidas para problemas complexos, exige que sistemas computacionais de alto desempenho sejam empregados.

Sistemas de PAD possuem características bem específicas de hardware e software que os classificam como sistemas que, para um determinado instante no tempo, apresentam desempenho superior ao comparado a computadores usuais, de propósito geral. O objetivo primordial destes tipos de sistema é a obtenção de desempenho, portanto não devem ser confundidos com sistemas puramente distribuídos. Sistemas distribuídos têm características mais abrangentes, não tendo necessariamente um compromisso com a obtenção de desempenho computacional. Possuem um carácter mais funcional voltado ao compartilhamento de recursos.

Para se ter uma ideia do tamanho e capacidade dos maiores sistemas de PAD atualmente, vale a pena consultar a lista Top 500 (TOP500, 2018) que enumera bi-anualmente as 500 máquinas mais rápidas do mundo para uma determinada aplicação padrão. No momento da escrita deste texto (abril de 2019), a máquina mais rápida do mundo é a *Summit*, localizada no *Oak Ridge National Laboratory* nos Estados Unidos. *Summit* é uma máquina fabricada pela IBM que possui 2.397.824 *cores* (núcleos de computação ou processadores) e atingiu 143,5 petaflops ao executar o *benchmark* de referência utilizado pelo Top 500. Além do processador *multicore* da família *Power* da IBM, a Summit é equipada com placas do tipo GPU (*Graphics Processing Unit*) da NVidia, que operam como co-processadores aritméticos permitindo grande aceleração nos cálculos científicos, além de contar com uma rede de interconexão de baixa latência do tipo Infiniband da Mellanox. Embora estas máquinas estejam fora do alcance financeiro de grande parte das empresas e institutos de pesquisa e desenvolvimento,

sobretudo no Brasil, os fabricantes oferecem versões menores com o objetivo de atender a diferentes demandas de mercado.

Considerando-se a complexidade do hardware com seus vários níveis de paralelismo, não é difícil imaginar que a programação destas máquinas exige um bom nível de especialização para que seja possível a obtenção de desempenho. A cada nível de paralelismo (por exemplo: nível de *threads* X nível de processos) diferentes paradigmas de programação paralela podem ser empregados. Mesmo com a utilização de *frameworks* e bibliotecas para desenvolvimento de aplicações paralelas, os softwares de PAD ainda requerem algum nível de especialização e de compreensão do hardware utilizado para que se obtenha um desempenho satisfatório.

O presente capítulo tem o intuito de abordar os paradigmas básicos de programação necessários a utilização de equipamentos mais simples, porém derivados dos atuais supercomputadores. O texto aborda a programação para equipamentos de alto desempenho de uma maneira introdutória, descrevendo paradigmas e técnicas de programação que se aplicam a uma arquitetura paralela de memória compartilhada, bem como a utilização de aceleradores de desempenho do tipo GPU. Neste sentido, são abordados conjuntos de diretivas de programação que podem ser utilizados em conjunto com as linguagens C/C++ para a programação de *multicores* e de GPUs. Para programação de *multicores* é utilizada a programação com diretivas aderentes ao padrão OpenMP (Chapman, 2008), e para programação de equipamentos dotados de GPUs como co-processadores numéricos, será apresentada programação com diretivas do padrão OpenACC (OpenACC, 2019).

Os pré-requisitos esperados para os leitores deste documento é que possuam conhecimentos sobre linguagem de programação C e estruturas de dados. Conhecimentos básicos sobre sistemas operacionais e arquitetura e organização de computadores, bem como arquiteturas do tipo GPU são desejáveis, porém não obrigatórios.

Este documento está dividido da seguinte forma: o segundo subcapítulo aborda a arquitetura de sistemas computacionais de Alto Desempenho de forma a contextualizar os modelos de programação abordados num momento posterior. O terceiro subcapítulo aborda algumas técnicas básicas de projeto e extração de paralelismo considerando o paradigma de memória compartilhada, essencial para que se possa programar através das diretivas paralelas a serem apresentadas posteriormente. O subcapítulo quatro detalha a programação por diretivas OpenMP aplicado a sistemas de memória compartilhada, detalhando um exemplo de aplicação prática. O quinto subcapítulo discorre sobre a programação para aceleradores do tipo GPU, com especial ênfase para a programação com diretivas do padrão OpenACC, mostrando exemplos simples de aplicação, bem como a aceleração do mesmo estudo de caso utilizado no subcapítulo anterior, comparando o desempenho com o programa em OpenMP. O sexto e último subcapítulo apenas ressalta as conclusões.

2.2. Modelo hierárquico de sistemas computacionais de alto desempenho

A arquitetura de sistemas computacionais de alto desempenho compreende diferentes tipos de paralelismo que podem ser vistos a partir de um modelo hierárquico de composição, partindo de múltiplos *cores* até o sistema completo com uma série de *racks*, incluindo componentes adicionais conhecidos atualmente por aceleradores. Este subcapítulo aborda o tema de forma simplificada, enfatizando a arquitetura de um único nó computacional de alto desempenho comum nas atuais arquiteturas de supercomputadores. Vale salientar, no entanto, que o sistema completo de um supercomputador conta com outras possibilidades de componentes não abordados aqui, tais como nós de acesso (*login*) e sistema de arquivos. Um bom exemplo deste tipo de arquitetura hierárquica é a arquitetura Blue Gene/Q da IBM (Gilge, 2014). A seguir, são apresentadas as características básicas de um módulo básico para processador e memória compartilhada.

2.2.1. Módulo básico para processador e memória compartilhada

Em termos de hardware, o módulo básico de um supercomputador basicamente é composto por um ou mais processadores do tipo *multicore* (mais comumente da conhecida família Intel x86) que possui normalmente uma dezena de *cores* (ou núcleos) efetivos e uma memória principal da ordem de centenas de Gbytes. Considerando-se arquiteturas paralelas, este modelo se encaixa entre as chamadas arquiteturas com **memória compartilhada** ou **multiprocessador**.

Nos multiprocessadores, todos os núcleos acessam uma memória compartilhada através de uma rede de interconexão. Nas máquinas mais comuns, a rede de interconexão comumente empregada é o barramento que liga os núcleos à memória. Entretanto, outras interconexões mais sofisticadas, podem ser usadas em máquinas com uma quantidade maior de processadores e memória.

Os diversos níveis da hierarquia de memória (registradores, memórias *cache*, memória principal) afetam diretamente o desempenho das aplicações. Porém, neste texto introdutório as questões relacionadas à hierarquia (Hennessy e Patterson, 2007) são apenas brevemente tratadas. No contexto de arquiteturas paralelas, a memória é comumente abordada a partir de uma classificação básica utilizada para diferenciar as máquinas paralelas: memória compartilhada e memória distribuída (não compartilhada).

A memória compartilhada significa que o espaço de endereçamento é único. Assim, pode-se imaginar que a memória principal, mesmo sendo composta fisicamente por vários bancos, é endereçada pela mesma sequência de endereços. Assim, diferentes processadores ou *cores* podem acessar um mesmo endereço físico de memória, o que permite que compartilhem dados. Na prática, utiliza-se comumente a programação através de *threads* com variáveis compartilhadas.

Antes de continuar, entretanto, vale ressaltar as diferenças básicas entre processos e *threads*. Um processo é uma unidade de programa em execução que inclui a posse de um espaço de memória para guardar a imagem do processo. A imagem do processo inclui o código do programa, os dados globais, os dados locais (pilha), a área

de memória dinâmica (*heap*), além do contador de programa (PC – *program counter*) e demais atributos que definem o processo. Já as *threads* são normalmente unidades de execução dentro do escopo de um processo. Assim, elas compartilham a maior parte do espaço de endereço (memória) de um processo. A sua área própria terá apenas o PC e a pilha (dados locais). As variáveis globais e a área de memória livre são automaticamente compartilhadas entre as *threads* de um mesmo processo, o que possibilita o compartilhamento de variáveis, muito útil na programação paralela deste tipo de arquitetura. Além disso, o fato de possuírem uma quantidade menor de recursos a serem gerenciados, torna as *threads* unidades de processamento concorrente de menor custo de criação e encerramento comparativamente aos processos. Por esta razão, são empregadas no paralelismo de aplicações em arquiteturas do tipo *multicore*.

Este texto trata de modelos de de programação com memória compartilhada, mas vale a pena diferenciá-lo do modelo de memória distribuída. A definição clássica para arquitetura de memória distribuída envolve os itens básicos que todo computador deve possuir: processador e memória. Cada processador possui sua própria memória local, com endereçamento de memória particular e a comunicação acontece por algum meio físico de comunicação (rede de interconexão). Essa representação também é chamada de arquitetura de **multicomputadores**, normalmente conhecidos como *clusters* ou aglomerados de computadores. Ao contrário das arquiteturas com memória compartilhada, as de memória distribuída são altamente escaláveis, permitindo centenas de milhares de *cores* devido principalmente ao desacoplamento de memória. É comum, encontrar-se máquinas híbridas, contendo ambos os paradigmas de memória.

2.2.1.1. Arquitetura *multicore*

Um *multicore* combina dois ou mais núcleos (*cores)*, em uma única peça de silício (ou pastilha – *die*). Normalmente um núcleo consiste de todos os componentes de um processador independente, tais como registradores, ALU (*Arithmetic and Logic Unit* - unidade aritmética e lógica), *pipelines*, unidades de controle, *caches* de dados (de L1 até L3, em alguns casos) e de instruções.

Num nível mais alto de descrição, as principais variáveis em uma organização *multicore* são as seguintes (Stallings, 2009):

- Número de núcleos de processamento no chip.
- Número de níveis de memória *cache*.
- Quantidade de memória *cache* que é compartilhada ou dedicada em cada núcleo.

Para as *caches* não compartilhadas, o hardware inclui a implementação de um protocolo que garante a coerência das *caches* em caso de alteração de seu conteúdo pelo respectivo núcleo. Maiores detalhes sobre protocolos de coerência de *cache* podem ser encontrados em Stallings (2009) e Hennessy e Patterson (2007).

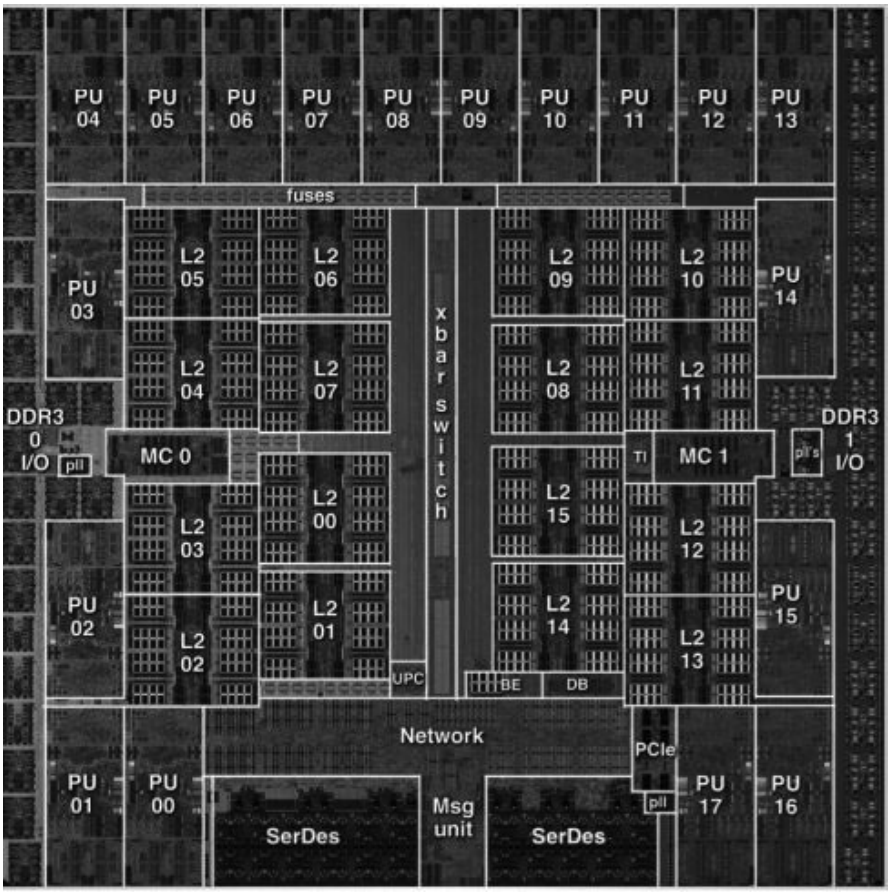

Figura 2.1 - Arquitetura multicore de uma CPU do sistema IBM Blue Gene/Q
Fonte: Stephan, 2015 (*licença CC 4.0*)

A título de exemplo, na figura 2.1 é possível visualizar a arquitetura interna de uma CPU utilizada no sistema IBM Blue Gene/Q. Nesta imagem pode-se identificar os 16 núcleos paralelos por *PUs* (*Processing Units*) e vários bancos de memória *cache* L2, que são compartilhados por todos os núcleos presentes. Além disso é possível verificar a existência de dois módulos de controle de memória (*Memory Controllers - MC 0* e *MC 1*), que permitem a movimentação de dados entre a memória principal externa, a memória *cache* interna e os registradores.

2.2.1.2. Computação heterogênea com GPU como co-processadores

O termo **computação heterogênea** tem sido empregado para designar o uso de diferentes arquiteturas de processamento em um mesmo sistema computacional, utilizados conjuntamente, a fim de se obter melhor desempenho das aplicações. Neste tipo de sistema é comum encontrar-se dispositivos conhecidos por **aceleradores**, os quais, na maioria das vezes, atuam como um co-processadores, ou seja, unidades extra de processamento de código dependente de um processador tradicional. Entre as alternativas existentes destacam-se arquiteturas como as GPUs (*Graphics Processing*

Units). As unidades aceleradoras podem estar fisicamente conectadas a um nó de processamento que conta com um processador tradicional, ou representarem um nó computacional por completo.

As GPUs são compostas de centenas ou até milhares de núcleos (*cores*) simples que executam o mesmo código através de centenas a milhares de *threads* concorrentes. Esta abordagem se opõe ao modelo tradicional de processadores *multicore*, onde algumas unidades de núcleos completos e independentes são capazes de executar *threads* ou processos independentes. As GPUs contam com unidades de controle e de execução mais simples, onde uma unidade de despacho envia apenas uma instrução para um conjunto de núcleos que a executarão em ordem. Este modelo de execução onde várias *threads* são executadas simultaneamente em vários *cores* das GPUs é conhecido como SIMT (*Single Instruction Multiple Threads*), derivado do clássico termo SIMD (*Single Instruction Multiple Data*) (Flynn, 1972). O modelo SIMT impõe um sincronismo na execução das *threads* que executam em grupo nas unidades de multiprocessadas das GPUs. O modelo de *threads* das GPUs será abordado na seção que trata deste modelo de programação. No entanto, a arquitetura das GPUs não será detalhada neste texto por ser altamente especializada, maiores detalhes podem ser encontrados em Kirk e Hwu (2010).

Entre os demais modelos de aceleradores existentes cita-se o uso de FPGAs (Escobar et al. 2016), recentemente impulsionado pela compra da Altera (fabricante de FPGAs) pela Intel. Entretanto, o uso de FPGAs como aceleradores para alto desempenho ainda necessita de modelos de programação com um nível de abstração aceitável pelos desenvolvedores. Neste contexto, a biblioteca OpenCL tem sido utilizada com sucesso para o desenvolvimento de aplicações em FPGA, como por exemplo em Hassan et al. 2018. Este tipo de acelerador, no entanto, está fora do escopo deste texto.

2.3. Paralelização de aplicações

Se a aplicação já existe (código legado), ela poderá ser paralelizada, o que muitas vezes exige uma refatoração para que usufrua do processamento *multicore* ou GPU. Breshears, 2009 sugere quatro passos na paralelização de aplicações: análise, projeto e implementação, testes de correção e análise de desempenho. Os tópicos a seguir abordam estes quatro passos.

2.3.1 Análise

Neste primeiro passo realiza-se a identificação dos possíveis pontos onde se possa implementar concorrência. Esta só poderá ser implementada em pontos de código onde não haja dependência de dados. Além disso, sugere-se escolher os trechos de paralelização a partir da medição de tempo em trechos do código onde haja muito processamento (*hot spots*). Normalmente estes trechos podem ser identificados por laços.

2.3.1.1 Análise de dependências

A implementação de paralelismo implica na execução concorrente (simultânea) de trechos de código. Considerando-se arquiteturas com memória compartilhada é necessário evitar o acesso concorrente a posições de memória (variáveis) que estão sofrendo operações de leitura e escrita de forma intercalada. Este tipo de atualização intercalada de variáveis compartilhadas pode gerar inconsistências na execução do código e são conhecidas como condições de corrida (*race conditions*).

A análise de dependências é o mecanismo formal que permite identificar limitantes à aplicação de concorrência para paralelização em um dado algoritmo originalmente sequencial. A técnica é especialmente útil quando aplicada em um programa que sofrerá uma decomposição de domínio, ou seja, concorrência através da divisão de dados a serem processados por diferentes processadores, como acontece frequentemente em laços que computam sobre *arrays*. A análise de dependências é também muito utilizada por compiladores para fazer paralelização ou vetorização automáticas, além de permitir execução especulativa e fora de ordem de instruções. Entretanto, a solução de um possível problema de dependência pode indicar transformações no código (refatoração) para permitir concorrência.

No algoritmo 2.1, por exemplo, pode-se ver dois laços implementados em linguagem C/C++ onde considera-se as variáveis internas passíveis de compartilhamento. No primeiro (à esquerda), é possível observar que cada iteração pode ser independente uma da outra, haja vista que não há acesso de leitura e escrita intercalada entre as variáveis internas. Deveria-se apenas cuidar para que o índice do laço não fosse compartilhado caso os trechos sejam concorrentes.

No segundo trecho (à direita), existe uma variável interna que pode sofrer operações de leitura e escrita de forma intercalada, pois trata-se de um acumulador (variável `SUM`). Em caso de implementar-se concorrência no código da direita, haveria a necessidade de refatoração para permitir o acesso concorrente ao acumulador sem o perigo de atualizações incorretas. Além disso, a posição dada pelo índice "*i*" do vetor `X` está também sofrendo operações de leitura e escrita durante as iterações do laço, ocasionando dependência de dados entre as duas primeiras linhas. Note-se que a primeira deve ser avaliada antes da segunda para que a posição corrente do *array* X esteja atualizada antes de ser utilizada na linha seguinte. Caso seja os comandos sejam executados de forma concorrente, não será possível garantir esta ordem.

```
for(i=0; i<N; i++) {
    X[i] = Y[i] + Z[i];
    A[i] = Y[i] + delta;

}
```

```
SUM = 0;
for(i=0; i<N; i++) {
    X[i] = Y[i] + Z[i];
    A[i] = X[i] + delta;
    SUM += A[i];
}
```

Algoritmo 2.1 - Exemplo de laços sem e com dependência de dados

Existem vários tipos diferentes de dependências entre instruções de um algoritmo. O fato de haver dependência não significa necessariamente que o fragmento do código não pode ser transformado em um código com concorrência, para tanto deve-se verificar que tipo de dependência se trata. Algumas formas permitem a técnica de vetorização (não abordada neste texto) mas não a paralelização do código, outras o oposto.

De acordo com Dowd e Severance (1998), as dependências podem ser classificadas em três tipos:

a) Dependência de Fluxo - **RAW** (*Read After Write*): a leitura depois da escrita pode ocorrer quando uma variável sofre operação de escrita em uma primeira instrução (atribuição) e subsequentemente é utilizada na próxima instrução (operação de leitura). O problema ocorre caso o inverso aconteça por conta da concorrência.
b) Anti-dependência - **WAR** (*Write After Read*): a escrita depois da leitura pode ocorrer quando uma variável sofre operação de leitura em uma primeira instrução e subsequentemente é atribuída na próxima instrução, o que significa sofrer operação de escrita. Novamente, o problema ocorre caso o inverso aconteça por conta da concorrência.
c) Dependência de saída - **WAW** (*Write After Write*): a escrita depois da escrita pode ocorrer quando uma variável sofre operação de escrita em uma primeira instrução e imediatamente é atribuída, sofrendo nova operação de escrita. Neste caso, o problema ocorre em atualizações fora da ordem sequencial imposta pela sequência de execução do laço.

Ao realizar a análise de dependência, portanto, é preciso observar-se com atenção variáveis compartilhadas que sofrem operações de leitura e escrita dentro dos tracho a serem paralelizados. Busca-se estas variáveis no código e tenta-se realizar a refatoração a fim de tentar eliminá-las ou protegê-las do acesso concorrente. Ao analisar-se novamente os laços apresentados no algoritmo 2.1 verifica-se que o primeiro (à esquerda) não possui nenhuma dependência, pois as duas posições dos vetores `X` e `A` que estão sendo escritas não aparecem em nenhuma operação de leitura (lado direito das atribuições).

No segundo laço, entretanto, posições acessadas do array `X`, assim como a variável `SUM`, aparecem ao lado esquerdo e direito de atribuições, o que significa que estão sofrendo operações de leitura e escrita no trecho de código que se deseja paralelizar. As operações no vetor `X` configuram **RAW**, pois a escrita deve ser executada antes da leitura. Neste caso, o problema na execução poderá ocorrer se uma *thread* fizer a leitura antes de a referida posição do vetor sofrer a atualização necessária (escrita) na linha anterior. Entretanto, este tipo de dependência é facilmente solucionável se cada *thread* concorrente executar uma iteração completa do laço, evitando a concorrência dentro de cada iteração, criando apenas a concorrência entre as diversas iterações do mesmo laço.

Já as operações na variável `SUM` são mais delicadas, pois tem-se operações de leitura e escrita em uma mesma variável compartilhada. Considerando que as *threads* concorrentes executam iterações completas do laço, conforme citado no parágrafo

acima, as operações concorrentes sobre o acumulador podem gerar situações de **WAW** e **WAR** entre *threads* concorrentes. No caso de WAW, tem-se a ocorrência de *threads* concorrentes sobrescrevendo o valor de `SUM` de forma sucessiva e possivelmente fora de ordem. Para o caso de WAR, pode ocorrer o uso desatualizado da variável `SUM` em uma dada iteração *"i"*, caso o valor da mesma variável na iteração *"i-1"*, em execução por outra *thread*, não tenha sido salvo na posição de memória correspondente. Este caso é bem comum em variáveis como acumuladores, visto que várias *threads* podem estar atualizando estas variáveis de forma intercalada, conforme já mencionado. O tratamento para este tipo de operação será visto em várias ocasiões ao longo deste texto.

A utilização de mecanismos de exclusão mútua serve para evitar os problema acima mencionados, conhecidos também como condição de corrida (*race condition*), onde a leitura e escrita em variáveis compartilhadas podem gerar resultados inconsistentes com o esperado. Mecanismos de exclusão mútua (Ben-Ari, 2005) normalmente fazem parte de disciplinas de Sistemas Operacionais e/ou Sistemas Concorrentes e servem para controlar o acesso a posições de memória compartilhada, as chamadas regiões ou seções críticas, tal que apenas um trecho de código concorrente realize o acesso de cada vez. Embora este tipo de mecanismo traga benefícios em termos da correção na execução do código, eles podem limitar o paralelismo, portanto devem ser evitados sempre que possível. Exemplos de tais mecanismos são os *locks*, semáforos e monitores. As linguagens e bibliotecas de programação paralela costumam oferecer opções de controle de acesso a regiões críticas de código, alguns deles serão vistos ao longo do texto.

2.3.1.2 Detecção de *hot spots*

Inicia-se o trabalho de paralelização do código a partir dos trechos mais demorados, os quais representam no jargão popular o "gargalo" no tempo de processamento, também chamados de *hot spots*. Usualmente os laços são bons candidatos, visto que executam tarefas repetitivas sobre um conjunto de dados homogêneos. Os laços são candidatos naturais, portanto, ao tipo de paralelismo conhecido como paralelismo de dados. Neste tipo de paralelismo é possível recorrer à decomposição de domínio, onde o domínio, representado por uma estrutura de dados homogênea, é dividido entre unidades de processamento concorrentes (*threads* ou processos) que executam o mesmo processamento de forma simultânea sobre dados diferentes. Este tipo de paralelismo se opõe ao paralelismo de tarefas, ou decomposição funcional, onde cada unidade de processamento concorrente executa uma tarefa diferente para a resolução de um problema. Este texto tem o foco na decomposição de domínio.

Assim, de uma maneira geral, escolhe-se os laços candidatos e efetua-se a medição de tempo com o objetivo de detectar aqueles que consomem maior tempo de processamento. Usualmente esta medição é realizada capturando-se o valor do relógio antes e depois do trecho a ser medido e efetuando-se a diferença entre as duas medidas. O algoritmo 2.2 mostra um exemplo de código que pode ser utilizado para esta tarefa, onde se introduziu uma função que permite medir um instante de tempo em intervalos distintos (função `gettimeofday`), e depois calcular a diferença entre esses intervalos.

O processo de introduzir medidas relacionadas a desempenho em código-fonte é conhecido por instrumentalização.

```
#include <stdio.h>
#include <sys/time.h>
#define N 1000000

int main(void) {
   int k;
   double p = 1, x;
   struct timeval inicio, final;
   long long tmili;

   gettimeofday(&inicio, NULL);
   x = 1.0 + 1.0/N;
   for(k=0; k<N; k++)
      p = p*x;
   gettimeofday(&final, NULL);

   tmili = (int) (1000*(final.tv_sec - inicio.tv_sec) +
           (final.tv_usec - inicio.tv_usec) / 1000);

   printf("PI aproximado: %g\n", p);
   printf("tempo decorrido: %lld ms\n",tmili);
   return 0; }
```

Algoritmo 2.2 - Exemplo de código-fonte com instrumentação para medida de tempo de execução de um trecho específico.

A detecção de *hot spots* juntamente com análise de dependência servem para identificar a viabilidade e os possíveis benefícios na paralelização de determinados trechos. A paralelização de código deve ser efetuada em casos críticos onde o tempo de processamento é proibitivo e onde o esforço de paralelização seja recompensado através de ganhos de desempenho (*speedup*). Além disso, a análise de dependência deve indicar possíveis empecilhos que possam tornar a paralelização mais difícil, exigindo uma refatoração mais delicada. Assim, a fase de análise deve trazer indícios para a avaliação de custo e benefício da paralelização de um código.

2.3.2 Projeto e implementação

Após a fase de análise, tem-se a fase de desenvolvimento, onde o projeto e a implementação devem ser executados. A fase de análise indica o tipo de decomposição de domínio a ser empregada, assim como quais são os pontos do código onde se poderá ter maior ganho. No projeto, deve-se levar em conta também qual o hardware disponível e consequentemente qual ou quais os paradigmas de programação a serem empregados.

Num primeiro momento, é comum considerar-se o paradigma de memória compartilhada visto que as arquiteturas mais comuns e presentes na primeira escala de paralelismo são as de *multicore*. Seguindo o mesmo raciocínio, pensa-se em seguida nos aceleradores, tais como as placas gráficas, hoje em dia presentes na maioria das

máquinas utilizadas para fins comuns. Embora as arquiteturas sejam diferentes, em ambos os casos pode-se contar com memória compartilhada e paralelismo entre *threads* - a principal diferença e termos de programação é a granularidade do processamento, como será visto mais adiante. Um terceiro passo seria pensar na escalabilidade, ou seja, na execução do código em centenas ou até milhares de *cores*. Neste ponto, entraria o paradigma de memória distribuída e a execução em *clusters*. O presente texto preocupa-se com a utilização de paralelismo em máquinas mais comumente disponíveis aos desenvolvedores, portanto apenas o paradigma de memória compartilhada é abordado.

2.3.2.1 Padrões de programação paralela

Padrões de código são soluções genéricas e reutilizáveis para problemas comuns em determinados contextos. O projeto de código paralelo pode usar algum padrão de código conhecido para este fim, como os apresentados em Mattson *et al* (2004) e McCool *et al* (2012). Particularmente este último apresenta alguns padrões de forma gráfica que podem ser úteis para que se visualize algumas das construções mais comuns em algoritmos paralelos. Foram selecionados quatro deles (figura 2.2) que serão úteis no decorrer deste texto: *fork-join, map, stencil, reduction*.

O padrão *fork-join* (figura 2.2 (a)) é dividido nestas duas operações que podem ser aninhadas. A operação *fork* (representada pelos círculos escuros na figura) permite que o fluxo de controle seja dividido em múltiplos fluxos de execução paralelos (quadrados). Na operação *join* (círculos mais claros) estes fluxos se reunirão novamente no final de suas execuções, quando apenas um deles continuará.

O padrão *map* (figura 2.2 (b)) replica uma mesma operação sobre um conjunto de elementos indexados. O conjunto de índices pode ser abstrato ou estar diretamente relacionado aos elementos de uma coleção. Este padrão se aplica à paralelização de laços, nos casos onde se pode aplicar uma função independente a todo o conjunto de elementos.

O padrão *stencil* (figura 2.2 (c)) é uma generalização do padrão *map*, onde a função é aplicada sobre um conjunto de vizinhos. Os vizinhos são definidos a partir de um conjunto *offset* relativo a cada ponto do conjunto. Todos podem ser calculados em paralelo, porém alguns cuidados de implementação devem ser tomados para evitar o não-determinismo (assim como no padrão *map*). O não-determinismo pode acontecer porque os elementos são lidos e escritos simultaneamente de forma indiscriminada ocasionando diferença de resultados entre as execuções. Alguns pontos podem estar sendo calculados a partir de dados atualizados, enquanto outros podem estar sendo calculados a partir de dados antigos. A cada execução esta situação pode se alterar e gerar resultados diferentes tanto relativamente à execução sequencial quanto entre as execuções paralelas. Entre as técnicas que podem ser utilizadas para manter o determinismo entre as execuções paralelas está o uso de duas matrizes - uma para leitura e outra para a escrita - que se alternam a cada iteração. Esta técnica será vista num dos exemplos mais adiante neste capítulo.

O padrão *reduction* (figura 2.2 (d)) combina todos os elementos de uma coleção em um único elemento a partir de uma função combinadora associativa. Dada a associatividade da operação, muitas ordens de avaliação podem ser implementadas. Uma operação muito comum em aplicações numéricas é realizar um somatório ou encontrar o máximo de um conjunto de elementos.

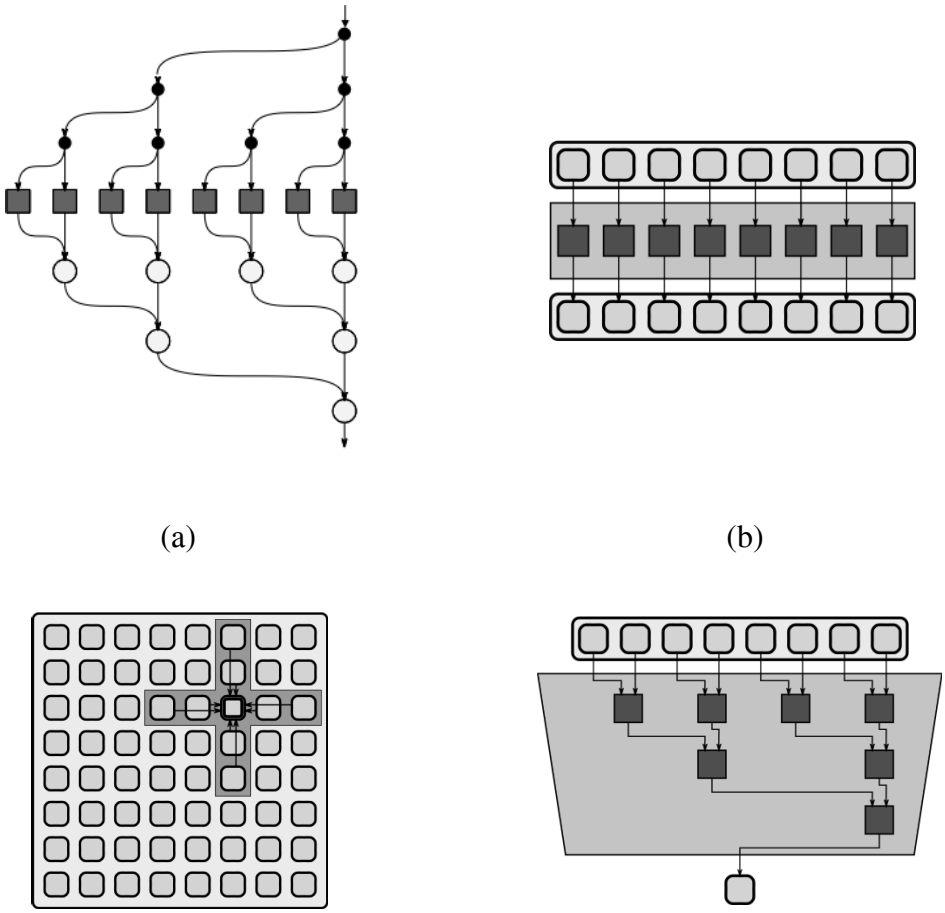

Figura 2.2: Alguns padrões de programação (Fonte: McCool et al, 2012)

Foram descritos alguns padrões básicos de programação paralela que podem ser selecionados na fase de análise. O desenvolvimento do algoritmo paralelo deverá utilizar alguma biblioteca de programação específica e a maioria das bibliotecas possui recursos para a implementação destes e de outros padrões de programação paralela.

Existem algumas opções de programação com *threads* em várias linguagens de programação, como Python, Java e C#. Entretanto, linguagens compiladas como C/C++ ainda apresentam melhor desempenho e são usadas em aplicações onde isto é fundamental. Neste caso, bibliotecas de programação como *Pthreads* (Butenhof, 2006)

e *OpenMP* (Chapman, 2008) são bastante utilizadas. A larga existência de sistemas legados, envolvendo principalmente cálculos numéricos complexos, implica no ainda comum uso da linguagem Fortran em aplicações de alto desempenho.

Este texto aborda duas das bibliotecas de programação mais utilizadas para PAD: OpenMP e OpenACC. A primeira será utilizada para abordar a programação para sistemas paralelos de memória compartilhada (incluindo os *multicores)*, embora algumas funcionalidades voltadas a aceleradores já estejam sendo incorporadas no momento da escrita deste capítulo. Já a segunda possui compilador mais estável para a programação de aceleradores, por isso será utilizada para tal. No caso de se desejar partir para uma implementação voltada à escalabilidade no nível de processos em arquiteturas do tipo *cluster*, a biblioteca de passagem de mensagens MPI - *Message Passing Interface* (Pacheco, 2011) é a mais indicada até o momento, porém foge do escopo deste documento.

OpenMP constitui-se de um padrão que define um conjunto de diretivas de programação juntamente com algumas rotinas de biblioteca (*API - Application Programming Interface* - Interface de Programação de Aplicações) para programação com *threads*. A programação com OpenMP é um dos principais objetivos do curso e será descrita mais adiante, juntamente com a biblioteca OpenACC.

2.3.3 Testes de correção

Este subcapítulo não pretende abordar técnicas e formalismos da área de teste de software, pois fugiria do escopo do texto. A ideia aqui é chamar a atenção para alguns aspectos da execução de programas paralelos que se diferenciam dos sequenciais e que podem ocasionar erros ou inconsistências de execução.

O código *multithreading* é altamente sujeito a erros como condições de corrida (*race conditions*) já abordadas na fase de análise de dependências. Basicamente, estes erros são causados pela alteração concorrente de dados compartilhados, assim como pelo mau-uso dos mecanismos de controle de acesso (por exemplo, *mutex*) e sincronização entre *threads*. Assim, um bom conjunto de testes é necessário, visto que programas concorrentes têm uma tendência a serem não-deterministas, ou seja, uma execução pode ter um resultado diferente de outra ainda que tenham os mesmos dados de entrada.

O método mais comum de teste de correção, embora aparentemente primitivo, ainda é observar a saída dos programas, o que pode ser feito diretamente no console ou num arquivo de saída. Em linguagem C os desenvolvedores de programas paralelos ainda dependem de comandos do tipo *printf* ou *assert* para realizar testes. Um dos principais motivos é que existem poucas ferramentas que permitem algum grau de depuração e a maioria das que existem não são gratuitas.

Assim, caso não haja ferramenta disponível, sugere-se algum teste básico para verificar a coerência da saída da versão sequencial com a da versão paralela. Num primeiro momento, portanto, é recomendável que se tenha uma versão sequencial devidamente testada e com um conjunto ou mais de dados de entrada e saída

conhecidos. Após o processo de paralelização, utiliza-se o mesmo conjunto de entrada para a execução paralela e observa-se se a saída é coerente. Um cuidado extra, porém, deve ser tomado com programas paralelos e sua natureza não-determinística: caso as devidas precauções não sejam tomadas, o resultado pode variar. Assim, existe a possibilidade, por exemplo, de que um primeiro teste seja compatível com o resultado da execução sequencial e os demais não. Assim, é necessário um conjunto de testes bem abrangente para garantir uma coerência constante entre os resultados das execuções, caso contrário dificilmente poderá obter-se certeza da correção da solução.

2.3.4 Análise de desempenho

É absolutamente necessário que o desempenho de uma aplicação paralelizada seja melhor que o de sua versão sequencial, visto que o custo e a complexidade deste processo devem valer à pena. Medidas de tempo e comparações com a versão sequencial do programa são usadas para verificar se houve ganho de desempenho. Caso contrário, o programador deverá verificar pontos de gargalo que estejam atrapalhando o desempenho. Tipicamente, estes pontos são situações onde haja contenção na sincronização de recursos compartilhados (por exemplo, uso sincronizado de variáveis compartilhadas), desbalanceamento de carga de trabalho entre as *threads* e quantidade excessiva de chamadas à biblioteca de *threads*, o que pode causar um *overhead* inexistente na versão sequencial.

Assim, dois dos principais objetivos do projeto de aplicações paralelas consistem em obter-se:

- **Desempenho**: é a capacidade de reduzir o tempo de resolução do problema à medida que os recursos computacionais aumentam;
- **Escalabilidade**: é a capacidade de aumentar ou manter o desempenho à medida que os recursos computacionais aumentam.

A vantagem da escalabilidade está relacionada à possibilidade de se aumentar o *tamanho do problema* e o consequente uso de recursos paralelos para resolvê-lo. O tamanho do problema normalmente corresponde ao tamanho do domínio, ou seja, o volume de dados e de sua estrutura de armazenamento.

Os fatores que limitam o desempenho e a escalabilidade de uma aplicação estão ligados a limites arquiteturais e limites algorítmicos. Entre os limites arquiteturais temos a latência e a largura de banda da camada de interconexão e capacidade de memória da máquina utilizada. Já os limites algorítmicos incluem a própria falta de paralelismo inerente ao algoritmo, usualmente capturado através da análise de dependência de dados. Além disso, tem-se a frequência de comunicação, representada pelo acesso às variáveis compartilhadas ou passagem de mensagens; a frequência de sincronização, normalmente imposta pelo algoritmo, e o escalonamento deficiente, que depende da granularidade das tarefas e do consequente balanceamento de carga a ser efetuado pelo sistema de execução.

A medida básica para a comparação e análise entre versões é o **tempo de execução**. Na fase de análise já utilizou-se medida de tempo semelhante para detectação

dos *hot spots* do programa. Agora, deseja-se saber o quanto o sistema A é mais rápido que o sistema B. Esta medida é dada por: $n = Texec(A) / Texec(B)$.

Baseando-se neste cálculo é possível obter-se o ganho de tempo ou ***speedup*** de um programa paralelo, uma das medidas mais populares para avaliar-se o desempenho de tais aplicações:

Speedup(P) = Texec(1 proc) / Texec(P procs)

- Onde P = número de processadores
- $1 \leq Speedup \leq P$

Apesar do *Speedup* apresentar os limites teóricos de 1 e P, eventualmente podem-se encontrar valores fora desta faixa, como, por exemplo, no caso de uma otimização mal sucedida, onde a nova versão do código apresenta desempenho inferior a versão serial, obtendo assim um *speedup* menor do que 1. Caso a nova versão favoreça o acesso dos dados em memória *cache* (o que é sempre desejável), devido a um melhor dimensionamento do domínio do problema na versão paralela, o *Speedup* pode ser super-linear, e assim atingir valores superiores à P.

Outra medida importante é a eficiência, que indica o uso dos processadores. A eficiência pode ser calculada da seguinte forma:

Eficiência(P) = Speedup(P) / P

- Onde $0 <$ Eficiência ≤ 1

Existe uma limitação clássica para o *speedup* conhecida como Lei de Amdhal (Amdhal, 1967). Basicamente, o conhecido autor dividiu o problema em duas categorias de trechos de código: aqueles que podem ser paralelizados (fração paralela) e aqueles que não podem (fração serial). A fração serial é principalmente limitada em função da análise de dependência de dados. Assim, a fração serial sempre será o limite inferior do tempo de execução, não importando o aumento dos recursos computacionais utilizados pela parte paralela.

Vale salientar que a Lei de Amdhal considera apenas a escalabilidade dos recursos computacionais, sendo o tamanho do problema fixo. Neste caso, efetivamente temos limitação do *speedup* que uma aplicação paralela poderá obter. Em contra partida, temos a Lei de Gustafson-Barsis (Gustafson, 1988) que parte da premissa que toda aplicação tem uma fração inerentemente serial e que para se obter escalabilidade é necessário aumentar o tamanho da aplicação ou do domínio a medida que se aumentam o número de recursos computacionais.

Assim, pode-se definir dois tipos de escalabilidade:

- **Escalabilidade forte**: mantém-se o tamanho do problema e escala-se o número de processadores; é a capacidade de executar aplicações *n* vezes mais rápidas, onde *n* é a quantidade de processadores utilizados (*speedup*).
- **Escalabilidade fraca**: escala-se o tamanho do problema juntamente com o número de processadores; é a capacidade de aumentar a carga de trabalho e a quantidade de processadores por um fator de *n* e manter o tempo de computação.

As próximas subseções apresentarão o paralelismo através de primitivas, além de estudos de caso onde estes conceitos teóricos abordados serão aplicados.

2.4. Programação OpenMP

OpenMP constitui-se de um padrão, composto por um conjunto de diretivas de programação, acrescido de um pequeno conjunto de funções de biblioteca e variáveis de ambiente, que usam como base as linguagens C/C++ e Fortran. Por se tratar de um padrão, várias implementações estão disponíveis. É comum que compiladores já conhecidos, como o popular *gcc (GNU Compiler Collection,* versão *open-source* de compilador C/C++*)*, possuam opções de compilação para OpenMP.

As diretivas em C/C++ para OpenMP estão contidas em diretivas do tipo **`#pragma`**, as quais permitem definir instruções que não existem previamente na linguagem C padrão, mais a palavra chave **`omp`** e o nome da diretiva. O conjunto de diretivas e sua cláusulas é relativamente grande, mas com um pequeno conjunto delas já é possível gerar código paralelo.

A principal diretiva do OpenMP é a **`parallel`** que automaticamente cria as *threads* que devem executar concorrentemente um bloco de instruções que a segue. Vale lembrar que um bloco de instruções com mais de uma linha em linguagem C é definido pelos símbolos `{` e `}` (abre e fecha chaves). Assim, para criar um bloco paralelo basta usar a seguinte combinação de palavras-chave: **`#pragma omp parallel.`** As diretivas podem ser complementadas por cláusulas que têm a função de especificar o modo de execução das diretivas, entre outros atributos.

Um ponto importante da programação paralela em OpenMP é a definição da quantidade de *threads* que será criada, também conhecida como "time de *threads*". A figura 2.3 mostra o modelo de execução do OpenMP, onde times de *threads* podem ser criados em qualquer ponto de um código em linguagem C. No contexto de Sistemas Operacionais, este modelo de execução é conhecido como *fork-join* (a operação *fork* cria processos enquanto a operação *join* realiza a sincronização ao final da execução concorrente). O modelo *fork-join* também foi apresentado entre os padrões de programação apresentados no subcapítulo correspondente.

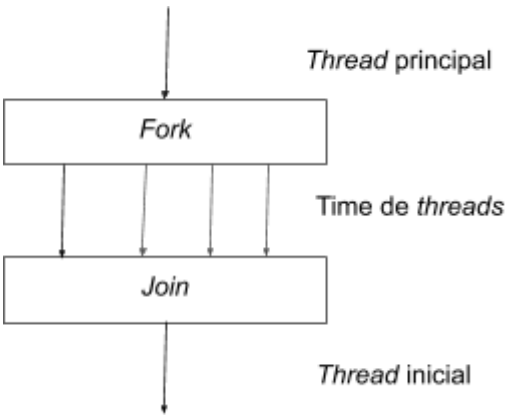

Figura 2.3: Modelo de programação *fork-join* suportado pelo OpenMP

Quanto à quantidade de *threads* criadas em um time existem diferentes métodos para sua definição. Caso não seja especificada de forma explícita no código, o sistema utiliza uma variável de ambiente (que pode ser alterada através de comandos do sistema operacional) chamada **`OMP_NUM_THREADS`**. Outra maneira de alterar a quantidade de *threads* é através de uma função da biblioteca do OpenMP chamada **`omp_set_num_threads()`**. Finalmente, a terceira alternativa é o uso da cláusula **`num_threads`**, juntamente com a diretiva **`parallel`**.

O OpenMP apresenta uma série de diretivas relacionadas ao compartilhamento de trabalho entre as *threads*. A principal delas é a diretiva **`for`**, que automaticamente divide entre as *threads* a execução do laço que a segue. Considerando que normalmente os *hot spots* de programas a serem paralelizados se encontram em laços, a diretiva `for` constitui-se na maneira mais acessível de se obter um programa paralelo com ganhos de desempenho. Esta diretiva, assim como suas principais cláusulas, serão abordadas com mais detalhe no estudo de caso a ser apresentado a seguir.

O padrão OpenMP é amplamente utilizado para paralelização de tarefas visando a obtenção de desempenho em máquinas com memória compartilhada. Também foi o responsável pela popularização da programação paralela por diretivas, a qual tem sido empregada também na programação de aceleradores, através de novas versões do padrão OpenMP e do padrão OpenACC, abordado mais adiante. O estudo de caso a seguir demonstra como aplicar a metodologia de paralelização vista até agora, assim como empregar as principais diretivas do OpenMP para obter-se uma versão paralela e com melhor desempenho da aplicação escolhida.

2.4.1. Estudo de caso em OpenMP: Jogo da Vida

Para exemplificar, vamos tomar um estudo de caso para servir de exemplo de paralelização: O Jogo da Vida (do inglês: *Game of Life* - GOL, Adamatzky 2010), criado por volta de 1960 pelo matemático e professor de Princeton John Conway. O jogo se passa em tabuleiro, similar ao encontrado em tabuleiros de xadrez ou damas, no qual formas geométricas bidimensionais, formada por um conjunto de pontos dispostos no tabuleiro (ou na grade) que replicam-se autonomamente e modificam sua forma a cada evolução (ou iteração do jogo), de acordo com um conjunto de regras pré-definido. O jogo da vida é objeto de estudos para Físicos, Biólogos, Economistas, Matemáticos, Filósofos, entre outros, pois permite observar como padrões complexos podem emergir a partir de implementações de regras muito simples, tal como ocorre em um autômato celular.

As regras de evolução dos elementos que compõem o jogo são muito simples, dado um tabuleiro bidimensional finito ou de bordas infinitas, células desta grade podem ter apenas dois estados, vivo ou morto, sendo que:

a. Cada célula viva com menos de dois vizinhos morre de solidão;
b. Cada célula viva com quatro ou mais vizinhos morre por superpopulação.

c. Cada célula viva com dois ou três vizinhos deve permanecer viva para a próxima geração;
d. Cada célula morta com exatamente 3 vizinhos deve se tornar viva.

A figura 2.4 ilustra um exemplo de configuração para o tabuleiro do jogo, em um determinado instante ou geração.

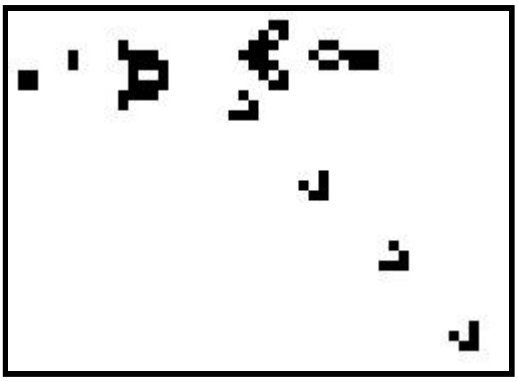

Figura 2.4 - Exemplo de tabuleiro de GOL

O trecho de código-fonte serial que descreve a possível mudança de estado de uma determinada célula no tabuleiro para uma próxima geração, considerando um tabuleiro de bordas periódicas, foi baseado no código-fonte disponível no repositório GitHub sob o endereço: https://github.com/olcf/game_of_life_tutorials, o qual foi desenvolvido pela divisão OLCF/ORNL (*Oak Ridge Leadership Computing Facility no Oak Ridge National Laboratory - TN*), podendo ser conferido no algoritmo 2.3.

```
1. // loop principal para evolucao de geracoes
2. for (iter = 0; iter<maxIter; iter++) {
3.       // Colunas esquerda e direita
4.       for (i = 1; i<=dim; i++) {
5.             // copiar ultima coluna real para coluna
6.             // esquerda de borda (left ghost column)
7.             grid[i][0] = grid[i][dim];
8.             // copiar primeira coluna real para coluna
9.             // direita de borda (right ghost column)
10.            grid[i][dim+1] = grid[i][1];
11.      }
12.      // Linhas superior e inferior
13.      for (j = 0; j<=dim+1; j++) {
14.            grid[0][j] = grid[dim][j];
15.            grid[dim+1][j] = grid[1][j];
16.            }
17.      // Loop sobre as celulas para nova geracao
18.      for (i = 1; i<=dim; i++) {
19.            for (j = 1; j<=dim; j++) {
20.                  // calcula numero de vizinhos
21.                  int numNeighbors = getNeighbors(grid, i, j);
22.
```

```
23.                   // aplicacao das regras do GOL
24.                   if (grid[i][j]==1 && numNeighbors<2)
25.                         newGrid[i][j] = 0;
26.                   else if (grid[i][j]==1 &&
27.                            (numNeighbors==2 || numNeighbors==3))
28.                         newGrid[i][j] = 1;
29.                   else if (grid[i][j]==1 && numNeighbors>3)
30.                         newGrid[i][j] = 0;
31.                   else if (grid[i][j]==0 && numNeighbors == 3)
32.                         newGrid[i][j] = 1;
33.                   else
34.                         newGrid[i][j] = grid[i][j];
35.                   }
36.             }
37.
38.       // troca de arrays para proxima geracao
39.       int **tmpGrid = grid;
40.       grid = newGrid;
41.       newGrid = tmpGrid;
42.       }// Fim do laco principal
```

Algoritmo 2.3 - Principal laço do programa Jogo da Vida (Fonte: OLCF/ORNL)

Este citado programa disponibilizado pela OLCF/ORNL foi avaliado pelos autores do presente texto em um equipamento com a seguinte configuração: *dual* Intel Xeon E5-2660v4 @ 2.00GHz, onde cada CPU conta com 14 núcleos (*cores*). Desta forma a máquina utilizada conta com um total de 28 núcleos homogêneos de processamento, distribuídos em duas CPUs que compartilham o mesmo endereçamento de memória principal de 128 Gb.

O tempo de processamento da versão serial foi avaliado em um tabuleiro bidimensional de **dimensões 2048x2048**, ou seja, com **4.194.304 células,** sem contar as camadas de contorno, necessárias à implementação de uma condição de contorno periódica, de forma que a face esquerda deve ser considerada ligada a face oposta a direita, e a face superior ligada a face inferior (configuração conhecida como malha do tipo *torus*). Executando-se **10.000 iterações**, ou seja, processando 10 mil novas gerações sucessivas do tabuleiro, o tempo de execução medido foi de pouco mais de 5 minutos e 30 segundos (**330,474 segundos**). Vale lembrar ainda que o programa foi compilado com um compilador de linguagem C *PGI community edition* versão 18.10. Este compilador foi escolhido por permitir comparações de desempenho tanto em OpenMP quanto em OpenACC, o qual será mostrado em capítulos seguintes.

Esta versão inicial poderá ser otimizada em seu desempenho computacional ao se utilizar de programação *multithread* por OpenMP. O primeiro passo para introdução das diretivas de OpenMP consiste em analisar o código atual, identificando trechos onde se pode dividir o processamento em sub-tarefas a serem executadas concorrentemente (ou paralelamente). Este passo corresponde à análise de dependência de dados, mencionada anteriormente no texto. Assim, é possível perceber que o laço mais externo (algoritmo 2.3, linha 2), que itera sobre as gerações, não pode ser transformado em uma

região paralela OpenMP, pois há uma dependência entre as operações de cada passo de iteração do laço, ou seja, a próxima geração somente pode ser calculada quando a geração anterior estiver totalmente pronta.

Entretanto, os laços internos, correspondentes às linhas 4, 13 e 19/20 do algoritmo 2.3, realizam tarefas completamente independentes entre as iterações e podem ser paralelizados facilmente. O leitor pode observar que as posições que estão sendo escritas (esquerda das atribuições) não se repetem do lado direito, onde ocorrem as operações de leitura. Isto elimina as dependências do tipo RAW e WAR abordadas anteriormente. Já a dependência do tipo WAW e as que podem ocorrer entre iterações diferentes pode ser eliminada utilizando-se o recurso de duas matrizes: uma para leitura e outra para escrita, explicada mais adiante.

Após a identificação dos citados trechos sem dependência de dados, foi realizada uma tomada de tempo para checar a relevância destes trechos no tempo total de execução, o que permite verificar por *hot spots*. Os três laços citados correspondem por demandas relativas de tempo de computação correspondentes à: 0,19%, 0,02% e 99,75%, respectivamente. Desta forma, percebe-se que há claramente apenas um *hot spot*, que corresponde ao terceiro laço analisado, responsável pelo cálculo das novas gerações do tabuleiro, e que demanda mais de 99% do tempo total de execução (as diferenças de tempo de processamento nos dois primeiros laços, que são muito semelhantes, se devem a questões relacionadas a arquitetura de memória). Entretanto, por questões didáticas neste estudo de caso, todos os laços analisados receberão diretivas OpenMP para geração de laços paralelos, ficando na forma mostrada nos algoritmos 2.4, 2.5 e 2.6.

```
4.       // Colunas esquerda e direita
5.       #pragma omp parallel for
6.       for (i = 1; i<=dim; i++) {
7.             ...
8.             }
```

Algoritmo 2.4 - Laço paralelo para preenchimento de fronteiras esquerda e direita

```
12.      // Linhas superior e inferior
13.      #pragma omp parallel for
14.      for (j = 0; j<=dim+1; j++) {
15.            ...
16.            }
```

Algoritmo 2.5 - Laço paralelo para preenchimento de fronteiras superior e inferior

```
18.       // Laco sobre as celulas para nova geracao
19.       #pragma omp parallel for
20.       for (i = 1; i<=dim; i++) {
21.             for (j = 1; j<=dim; j++) {
22.                   ...
23.                   }
24.             }
```

Algoritmo 2.6 - Laço para cálculo de nova geração do tabuleiro

Cabe ainda citar que, após o laço que percorre as novas gerações do tabuleiro, outro trecho do código poderá ser paralelizado entre as *threads*, ainda que sua relevância em termos de demanda computacional seja de apenas 0,01% do tempo total de execução. Ocorre que ao final de todas as gerações, o número total de células vivas deverá ser exibido como resultado final do programa. Este trecho deverá simplesmente realizar uma contagem (somatório) dos valores de cada célula disposta no tabuleiro, somando o valor um (1) para cada célula viva em uma determinada variável.

Operações como somatórios, produtórios, busca por máximos e mínimos, entre outras, são conhecidas como operações de redução (*reduction*). Por serem muito comuns em códigos numérico-científicos, o OpenMP contempla uma cláusula específica para esta situação, permitindo que o laço paralelo calcule, da maneira eficaz, o resultado de uma operação de redução, conforme já demonstrado anteriormente. A operação paralela de redução foi abordada na seção sobre padrões de programação paralela, onde a figura 2.2(d) demonstra esquematicamente este padrão, o qual combina todos os elementos de uma coleção em um único elemento a partir de uma função combinadora associativa. O trecho que realiza a contagem (somatório) está demonstrado pelo algoritmo 2.7.

```
43.// Somatorio das celulas do tabuleiro
44.int total = 0;
45.   #pragma omp parallel for reduction(+:total)
46.for (i = 1; i<=dim; i++) {
47.       for (j = 1; j<=dim; j++) {
48.             total += grid[i][j];
49.       }
50.}
```

Algoritmo 2.7 - Laço paralelo cálculo de somatória dos valores do tabuleiro por redução

O desempenho da primeira versão OpenMP do código correspondente ao Jogo da Vida foi avaliado com as mesmas configurações e no mesmo equipamento para qual a versão serial gastou 330,474 segundos. Os resultados em relação a tempo de execução, bem como valores de *Speedup* e eficiência paralela podem ser conferidos na tabela 2.1 a seguir, onde, cada *thread* executa em um núcleo da máquina utilizada.

Tabela 2.1 - Desempenho da versão 1 em OpenMP

Nº threads	Tempo(s)	*Speedup*	Eficiência
Serial	330,474	1	100%
2	256,37	~1,289	~64,453%
4	132,531	~2,494	~62,339%
8	70,377	~4,696	~58,697%
16	39.945	~8,273	~51,708%
28 (max)	27.365	~12,076	~43,130%

Conforme se observa na tabela 2.1, a implementação já obteve ganhos de desempenho (*speedup*), apesar deste ganho ter ficado abaixo do *speedup* linear (ideal). Já a eficiência da versão paralela inicia-se com pouco mais de 64%, e reduz-se significativamente até chegar em aproximadamente 43,1%, utilizando-se de todos os 28 núcleos de processamento disponíveis.

As possíveis causas de ineficiência em sistemas paralelos podem ser várias, dentre elas destacam-se:

1. tempo despendido em operações sequenciais (Lei de Amdhal - subcapítulo 3),
2. tempo gasto com comunicações e tarefas de sincronização e,
3. desbalanceamento de carga entre as *threads*/processadores.

Dos itens acima, pode-se descartar o item 3 como possível causa da queda de eficiência observada, pois as tarefas concorrentes presentes nos laços são homogêneas e, portanto, com cargas computacionais idênticas. Para o primeiro item, cabe destacar que a fração serial do código em questão corresponde a toda alocação e iniciação de memória referente ao tabuleiro e seu estado inicial, além do processamento e controle do laço principal entre as gerações de tabuleiro. Esse trecho sequencial corresponde à apenas 0,02% do tempo total de execução da versão serial. Desta forma, na configuração atual, não pode ser considerado um severo fator limitante ao *speedup*. Além deste fato, cabe lembrar que, para a versão paralela, o tempo gasto com abertura e fechamento de *threads* correspondentes às três regiões paralelas implementadas através da diretiva *parallel*, ocorrem 10.000*3 vezes. Assim, este fator limitante corresponde ao item 2, tempo gasto com comunicação e sincronização de processos paralelos. Sua

medida de tempo exata é difícil de ser estimada, pois implica medir tarefas implementadas de forma automática por operações do OpenMP.

Uma possível otimização neste programa poderia ser a paralelização do procedimento que inicia a população do primeiro tabuleiro, entretanto, deve-se lembrar que este trecho apresenta pouca significância no tempo total de computação, conforme citado no parágrafo anterior. Além disso, uma vez que o tabuleiro é iniciado de forma pseudo-aleatória, a partir da geração de números aleatórios de uma semente fixa, a subdivisão desta tarefa em outras subtarefas concorrentes poderia gerar problemas de falta de compatibilidade binária entre a versão serial de referência e as versões paralelas em desenvolvimento, pois, desta forma, o tabuleiro inicial poderia ter seu estado alterado em função da quantidade de *threads*, o que impediria a checagem do resultado final esperado. Assim, durante o desenvolvimento, resolveu-se não alterar a forma da geração do estado inicial, permitindo verificar se o tabuleiro final da versão paralela gera o mesmo resultado da versão serial.

Outra possível otimização consiste em minimizar a quantidade de *threads* criadas e destruídas dentro do laço que controla as gerações. Assim, poderia-se abrir uma determinada quantidade de *threads* paralelas apenas uma única vez. A implementação desta funcionalidade exige, porém, uma maior quantidade de modificações no código serial, ou seja, exigirá do programador mais do que a simples colocação de uma diretiva `parallel for`. A ideia desta refatoração é justamente separar as diretivas `parallel` e `for` e utilizar cláusulas do OpenMP para controlar o escopo das variáveis (compartilhadas ou locais).

Desta forma, as *threads* serão abertas antes do laço que controla as gerações, o qual continuará a ser executado serialmente, porém, por todas as *threads* existentes. Esta mudança exigirá a definição formal do escopo das principais variáveis e *arrays* a serem utilizados na nova macro região paralela a ser criada. Assim, deve-se definir explicitamente quais variáveis/*arrays* terão seus valores compartilhados pelas *threads*, definidas pela cláusula `shared`, e quais serão criadas para serem variáveis locais ou privadas (cláusula `private`) para as *threads* existentes. Os *arrays* `grid` e `newgrid`, além das variáveis apenas de leitura, `maxIter` e `dim`, deverão ser consideradas compartilhadas pelas *threads*, enquanto que as variáveis contadoras de laços `iter`, `i` e `j`, deverão ser privadas, onde em cada *thread* um dado independente será alocado e utilizado.

Outro destaque fica por conta do trecho de código que faz a troca de ponteiros entre a grade antiga e nova, preparando a uma nova geração de população. Esta operação deverá ser efetuada por apenas uma única *thread*, uma vez que a mesma deverá ser atômica. O código resultante, que define a seção paralela com o escopo das variáveis e a operação atômica de troca de ponteiros, pode ser conferido no algoritmo 2.8.

```
1.  // loop principal para evolucao de geracoes
2.  #pragma omp parallel \
3.              shared(grid,newGrid,maxIter,dim) \
4.              private(iter,i,j)
5.  { // principal regiao paralela
```

```
6.  for (iter = 0; iter<maxIter; iter++) {
7.      // Colunas esquerda e direita
8.      #pragma omp for
9.      for (i = 1; i<=dim; i++) {
10.         // copiar ultima coluna real para coluna
11.         // esquerda de borda (left ghost column)
12.         grid[i][0] = grid[i][dim];
13.         // copiar primeira coluna real para coluna
14.         // direita de borda (right ghost column)
15.         grid[i][dim+1] = grid[i][1];
16.     }
17.     // Linhas superior e inferior
18.     #pragma omp for
19.     for (j = 0; j<=dim+1; j++) {
20.         grid[0][j] = grid[dim][j];
21.         grid[dim+1][j] = grid[1][j];
22.         }
23.
24.     // Loop sobre as celulas para nova geracao
25.     #pragma omp for
26.     for (i = 1; i<=dim; i++) {
27.         for (j = 1; j<=dim; j++) {
28.             // calcula numero de vizinhos
29.             int numNeighbors = getNeighbors(grid, i, j);
30.             // aplicacao das regras do GOL
31.             if (grid[i][j]==1 && numNeighbors<2)
32.                newGrid[i][j] = 0;
33.             else if (grid[i][j]==1 &&
34.                      (numNeighbors==2 || numNeighbors==3))
35.                      newGrid[i][j] = 1;
36.             else if (grid[i][j]==1 && numNeighbors>3)
37.                      newGrid[i][j] = 0;
38.             else if (grid[i][j]==0 && numNeighbors == 3)
39.                      newGrid[i][j] = 1;
40.             else
41.                      newGrid[i][j] = grid[i][j];
42.         }
43.     }
44.
45.     // troca de arrays para proxima geracao
46.     #pragma omp single
47.          { // regiao atomica para troca de ponteiros
48.            int **tmpGrid = grid;
49.            grid = newGrid;
50.            newGrid = tmpGrid;
51.          } // fim da regiao atomica
52.     } // Fim do laco principal
53.} // Fim da regiao paralela principal
```

Algoritmo 2.8 - Segunda versão de código OpenMP com otimização na criação de *threads*

Nesta última versão do código com OpenMP percebe-se que os laços paralelos apenas recebem anotações para terem suas iterações subdivididas entre as *threads*, não sendo mais necessário criar a própria *thread*, uma vez que as mesmas já foram criadas anteriormente. Além disso, o trecho que faz a troca de ponteiros recebeu uma cláusula **`single`** indicando que apenas uma *thread* deverá executar o trecho demarcado, e as demais *threads* deverão se abster de executá-lo.

O padrão OpenMP não define explicitamente a forma com que as iterações do laço serão divididas entre as *threads* concorrentes. Divisões em padrões de blocos ou divisões cíclicas, além de combinações de ambas, são igualmente aplicáveis e podem ser explicitamente definidas pelo programador se desejado. No estudo de caso em questão, a divisão de tarefas para os laços paralelos entre as *threads* é deixada a cargo da forma padrão (*default*) do OpenMP. Usualmente, a forma padrão divide as tarefas em blocos contíguos entre *threads* paralelas, de forma a manter uma divisão mais homogênea de carga possível.

Os resultados, comparados com a versão anterior, foram melhores, conforme se pode checar nas figuras 2.5 e 2.6. Percebe-se que a segunda versão consegue atingir, para o número máximo de *threads* (28 *threads*), um ***speedup*** de, aproximadamente, **21,4** em contraste com 12,1 da primeira versão. Já a **eficiência** paralela passou de 43,13% para **76,33%**, atestando a validade da transformação aplicada ao código. Cabe ainda citar que o trecho relativo ao cálculo da somatória dos valores no tabuleiro por redução, previamente mostrado, permanece inalterado nas duas versões. O *speedup* paralelo linear, exibido na figura 2.5, refere-se ao *speedup* exatamente igual a quantidade de *threads* em processamento, e serve como um parâmetro de comparação para estimar a distância do desempenho atingido/medido com o desempenho ótimo teórico, caso a paralelização não enfrentasse nenhum limitante, o que não ocorre na prática, na grande maioria das vezes.

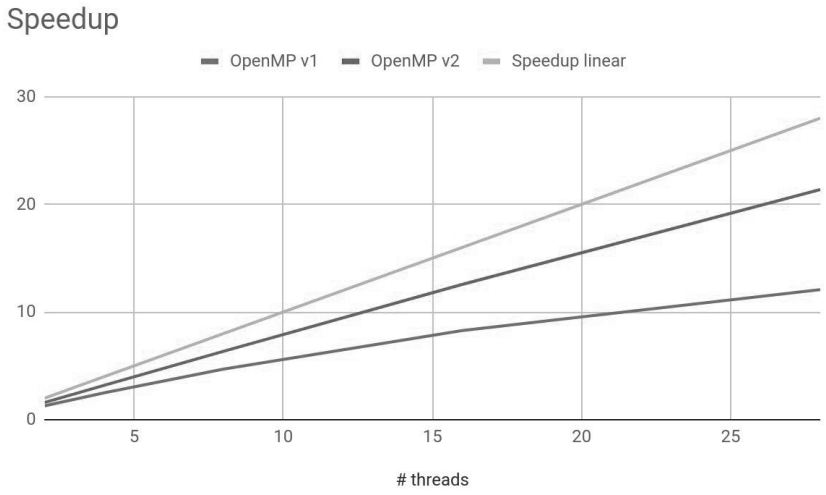

Figura 2.5 - *Speedup* para as versões OpenMP desenvolvidas

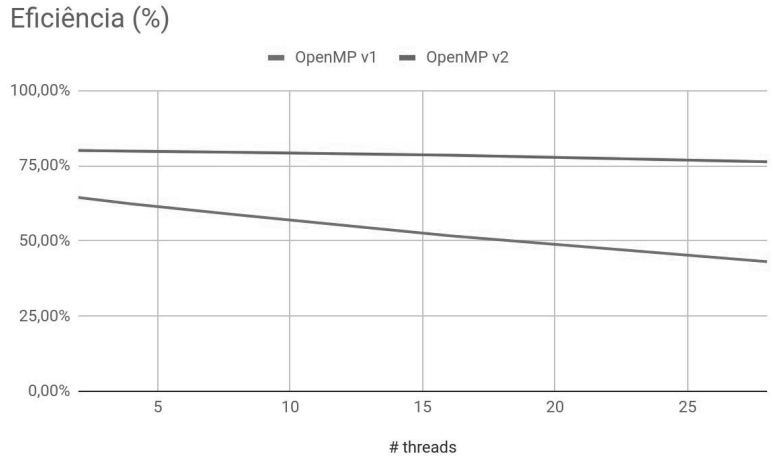

Figura 2.6 - Eficiência paralela para as versões OpenMP desenvolvidas

2.5. Programação para GPUs

As GPUs (*Graphics Processing Units*) estão entre os dispositivos conhecidos por **aceleradores**, os quais, na maioria das vezes, atuam como co-processadores, ou seja, unidades extra de processamento de código dependente de um processador tradicional. As unidades aceleradoras podem estar fisicamente conectadas à um nó de processamento que conta com um processador tradicional, ou representarem um nó computacional por completo.

No momento da escrita deste capítulo, cinco entre as dez primeiras máquinas da lista Top 500 (TOP500, 2018) possuem placas aceleradoras do tipo GPU da NVidia. Além de serem aceleradores eficientes, as GPUs têm um baixo consumo de energia comparados aos processadores *multicores*, por isso também são opções adotadas em "computação verde" (*green computing*) .

Este subcapítulo tem por finalidade apresentar os principais conceitos por parte dos aceleradores do tipo GPU e demonstrar as principais técnicas para programação do mesmo, com detalhamento do padrão OpenACC

As GPUs são compostas por centenas ou até milhares de núcleos (*cores*) simples que, normalmente, executam o mesmo código através de centenas a milhares de *threads* concorrentes. Tal característica se opõe ao modelo tradicional de processadores *multicore*, onde algumas unidades de núcleos complexos são capazes de executar *threads* ou processos independentes. Assim, ao se considerar o processamento paralelo em GPUs, será necessário introduzir o conceito para um modelo de computação conhecido por SIMT (*Single Instruction Multiple Threads*), derivado do clássico termo SIMD (*Single Instruction Multiple Data*). A arquitetura das GPUs não será detalhada neste capítulo por ser altamente especializada. Maiores detalhes das mesmas poderão ser encontrados em Kirk (2010).

Esta seção faz uma breve introdução do modelo de abstração mais conhecido para este tipo de arquitetura, conhecido por CUDA (*Compute Unified Device*

Architecture) (NVIDIA, 2018), a qual ajuda a padronizar a programação e o uso de GPUs ao definir um modelo de programação comum a todos os diferentes tipos de placas que a fabricante disponibiliza. Entretanto, a programação CUDA não será detalhada, servindo apenas de base introdutória para a programação seguindo o padrão OpenACC.

2.5.1 Programação CUDA

CUDA é uma plataforma de computação paralela, livremente distribuído pela empresa NVIDIA, aplicada a computação de aspectos gerais em GPUs. Constitui-se de uma série de extensões para a linguagem C/C++, juntamente com uma API (*Application Programming Interface*) que define algumas funções para manipulação destes dispositivos. O modelo de programação assume que o sistema é composto de um *host* (CPU) e de um dispositivo (*device* ou GPU), que funcionará com o um co-processador.

A programação consiste em definir o código de uma ou mais funções que executarão no dispositivo (conhecidos por *kernels*) e de uma ou mais funções que executarão no *host* (a função principal de um programa C, *main()*, por exemplo). Quando um *kernel* é invocado, centenas ou até milhares de *threads* são iniciadas no dispositivo, executando simultaneamente o código descrito no *kernel*. Os dados utilizados devem estar na memória do dispositivo e CUDA oferece funções para realizar esta transferência.

O Algoritmo 2.9 apresenta um exemplo de código em CUDA que implementa a soma de matrizes no dispositivo. O comando de invocação do *kernel* define a quantidade de *threads* dimensionadas em um bloco e a dimensão de uma grade de blocos. Resumidamente, as *threads* são organizadas em blocos de até três dimensões, e estes blocos compõem uma grade. Este mapeamento por vezes é considerado complexo e necessita de uma atenção maior do programador. O exemplo não aborda características um pouco mais complexas da programação CUDA, tais como o gerenciamento de memória (os dados devem ser transferido para o dispositivo antes da execução e trazidos de volta para a memória principal após o processamento) nem o uso de blocos maiores. Recomenda-se o próprio manual CUDA *Toolkit* disponível em (NVIDIA, 2018).

A partir do exemplo de código apresentado, no entanto, é possível observar algumas das principais características da programação CUDA. São elas:

- uso da palavra-chave `__global__` que indica que a função é um *kernel* e que só poderá ser invocada a partir do código executado no *host*, criando uma grade de *threads* que executarão no dispositivo (linha 02);
- uso das variáveis pré-definidas `threadIdx.x` e `threadIdx.y` que identificam a *thread* dentro do bloco através de suas coordenadas (linhas 6 e 7);
- uso do tipo pré-definido `dim3` (linha 16) para definir um bloco de *threads* com mais de uma dimensão;

- uso dos símbolos `<<<...>>>` (linha 17) para invocar várias instâncias do *kernel* no dispositivo de acordo com a quantidade de blocos e de *threads* por bloco indicada entre eles.

```
01 ...
02 // Kernel definition
03 __global__ void MatAdd(float A[N][N], float B[N][N],
04                         float C[N][N]) {
05     int i = threadIdx.x;
06     int j = threadIdx.y;
07     C[i][j] = A[i][j] + B[i][j];
08 }
09
10 int main() {
11     ...
12  // Kernel invocation with one block of N*N*1 threads
13     int numBlocks = 1;
14     dim3 threadsPerBlock(N, N);
15     MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
16     ...
   }
```

Algoritmo 2.9 - Trecho de código em CUDA somar duas matrizes.
Fonte: (NVIDIA, 2018)

2.5.2. Programação para aceleradores seguindo o padrão OpenACC

Embora o uso de CUDA seja a forma mais direta de programar GPUs, diversas opções têm surgido com o objetivo de tornar a programação mais acessível. Entre elas se destacam o uso de diretivas suportadas pelo OpenMP, na sua versão 4.0 e 4.5 (Van der Pas, 2017) e OpenACC (OPENACC, 2019).

O uso do padrão OpenACC consiste em um modelo para se expressar paralelismo baseado em diretivas aplicadas a linguagem C/C++ ou Fortran. Este modo de se expressar paralelismo reduz consideravelmente a complexidade do código-fonte empregado em comparação com CUDA, por permitir automatização de diversas tarefas comumente executadas em GPUs, como, por exemplo, o gerenciamento de memória. O padrão OpenACC emergiu de iniciativas isoladas de diversas empresas tal como a empresa *Portland Group* (PGI), a qual pode ser encontrada na pesquisa de Lee et al (2009), consistindo de uma proposta de um tradutor automático de código paralelo escrito com diretivas, como no OpenMP, para a linguagem CUDA, bem como nos projetos denominados CUDA-Lite (Ueng *et al*, 2008) e hiCUDA (Han e Abdelrahman, 2009).

Este subcapítulo introduz o modelo de programação OpenACC através de exemplos em linguagem C, incluindo uma aplicação completa conhecida por Jogo da Vida, já apresentada no subcapítulo dedicado ao OpenMP. Destacam-se as principais

vantagens na programação de aceleradores através deste padrão, bem como suas possíveis limitações e as formas de se mitigá-las.

Atualmente, alguns compiladores são disponíveis para uso, alguns com licença proprietária, como o compilador distribuído exclusivamente pela fabricante de supercomputadores Cray, de uso exclusivo dos equipamentos da marca, e outros permitindo o livre uso para finalidades não comerciais, como é o caso do compilador *Portland Group* em sua versão conhecida por *community edition*, o qual será utilizado nos experimentos práticos de programação deste curso.

Convém ainda citar que o padrão OpenACC foi criado com propósitos gerais, não sendo considerado uma ferramenta exclusiva para uso com GPUs. Assim, é possível utilizar um código aderente a este padrão para acelerar programas executando na própria CPU, de certa forma, rivalizando com o padrão OpenMP descrito anteriormente. Entretanto, por entender que o OpenMP já supre as necessidade de programação para sistemas de memória compartilhadas, além de ser um padrão muito bem conhecido na área, este texto limitar-se-á a aplicar códigos em OpenACC para execução em GPUs.

O próprio padrão OpenMP, em suas recentes versões 4, 4.5 e 5, prevê a aceleração de trechos de código para GPUs e demais possíveis aceleradores através do uso de diretivas. Porém, não existe, até o momento da escrita deste capítulo, um compilador que implemente todas as funcionalidades previstas nas últimas versões do padrão OpenMP para execução conhecida por *offloading*, desta forma, essa funcionalidade não será abordada neste capítulo.

2.5.2.1. Criação de regiões paralelas aceleradas com OpenACC

A criação de regiões aceleradas por diretivas do padrão OpenACC em um código-fonte escrito em linguagem C é simples e ocorre de forma muito similar ao bem conhecido padrão OpenMP. Duas diretivas principais podem ser utilizadas para esta finalidade, são elas:

`#pragma acc kernels`	`#pragma acc parallel`

Ambas permitem definir um trecho de código para ser executado em paralelo na GPU, porém, a primeira diretiva `kernels` é usada de forma mais conservativa, demandando análise automática do compilador do trecho delimitado procurando por possíveis dependências de dados, decidindo autonomamente se o trecho deve ou não ser paralelizado; enquanto que a diretiva `parallel` define uma região paralela de forma definitiva, onde a garantia da não existência de dependências deve ser dada pelo programador.

Para os trechos de códigos exibidos a seguir (Algoritmo 2.10), os quais realizam, ambos, uma simples operação de multiplicação seguida de soma (a bem conhecida operação *multiply and add*), temos duas possíveis opções de paralelização por OpenACC.

```
void saxpy(int n, float a,
        float *x,
        float *restrict y) {
     #pragma acc kernels
     #pragma acc loop
     for (int i = 0; i < n; ++i)
         y[i] = a * x[i] + y[i];
     }
```

```
void saxpy(int n, float a,
        float *x,
        float *restrict y) {
     #pragma acc parallel
     #pragma acc loop
     for (int i = 0; i < n; ++i)
         y[i] = a * x[i] + y[i];
     }
```

Algoritmo 2.10 - Duas versões de criação de região paralela acelerada

Nas duas versões do código exibido no algoritmo 2.10, o argumento `float *restrict y` define que o endereço de memória apontado por `*y` será exclusivo e, portanto, não apontará para outras áreas de memória referenciadas por outros ponteiros presentes nos argumentos da função `saxpy`. Assim, o programador garante que não haverá dependência entre os arrays `*x` e `*y`, pois ambos não apontam para uma mesma área de memória, o que poderia impedir a paralelização do laço.

Ambas as versões deverão produzir desempenho semelhante, uma vez que o código utilizando a diretiva `kernels` não deverá identificar dependências que impeçam a paralelização. Assim, o laço será transformado em um código binário a ser executado em GPU, utilizando-se de múltiplas *threads*, em múltiplos blocos, para executar de forma maciçamente paralela.

A cláusula `loop` indica ao compilador para paralelizar um laço imediatamente abaixo da mesma. Pode ser usada também de forma contraída e integrada na mesma linha da cláusula que indica a região acelerada, como em: `#pragma acc parallel loop`. Após a diretiva `loop` pode-se especificar, opcionalmente, outras cláusulas para: i) determinar escopo de variáveis que estão presentes no laço, e a forma de transferência destas para o dispositivo; ii) definir variáveis que irão sofrer processo de redução (`reduction`); iii) colapsar laços aninhados para paralelismo (`collapse`); iv) especificar a configuração lógica de quantidade de *threads* paralelas (`vector`) e de blocos de *threads* (`gang`) em GPUs; e v) indicar que o processamento interno ao laço deve dar-se na forma SIMD (*Single Instruction, Multiple Data*), ou seja, aplicar o mesmo conjunto de instruções para cada *thread* paralela, porém atuando em conjuntos de dados diferentes. Algumas destas características serão vistas a seguir.

A compilação de um código em linguagem C com diretivas no padrão OpenACC, utilizando-se do compilador PGI com chaves usuais, deve ser feito de forma semelhante a linha de compilação do bem conhecido compilador *gcc*, como no exemplo a seguir:

```
pgcc -acc [-Minfo=accel] [-ta=nvidia] [-o exemplo.x] exemplo.c
```

Onde a chave de compilação `-acc` habilita o compilador a reconhecer o padrão OpenACC, enquanto que as chaves `-Minfo` e `-ta` são opcionais, e definem, respectivamente, a exibição de informações sobre o procedimento de aceleração realizado e a especificação do dispositivo para o qual o programa binário deverá ser gerado. Além das chaves opcionais aqui descritas, existem muitas outras que podem ser consultadas no manual ou páginas de ajuda do compilador.

Um procedimento comum em programação paralela consiste em calcular reduções em operações de somatórios, produtórios, extração de máximo valor, etc, o qual também foi previsto em OpenACC, de forma similar ao visto em OpenMP, e que pode ser verificado no algoritmo 2.11.

```
float saxpy_sum(int n, float a, float *x, float *restrict y) {
     float total = 0.;
     #pragma acc parallel loop reduction (+:total)
     for (int i = 0; i < n; ++i) {
          y[i] = a * x[i] + y[i];
          total += y[i]; }
     return total;
     }
```

Algoritmo 2.11 - Exemplo de operação de redução com OpenACC

O padrão OpenACC permite ainda definir nas regiões paralelas o local em que as transferências entre *host* e *device* serão executadas, bem como o escopo dos dados transferidos, ou seja, se são dados apenas utilizados para leitura dentro do dispositivo, se são de leitura e escrita, se são criados no dispositivo e transferidos de volta ao *host* ao final da região, ou ainda se são dados temporários, usados de forma auxiliar pelo algoritmo que executa na GPU, mas que não necessitam ser sincronizados com o *host*. O exemplo no algoritmo 2.12 ilustra o uso de diretivas de transferência de dados, onde a diretiva `copyin` especifica que a transferência do *array* "`x`" será feita no local indicado apenas no sentido CPU→ GPU, pois os dados serão utilizados apenas no modo de leitura pelo *kernel* executando no dispositivo. Entretanto, as informações do *array* "`y`", contidas na diretiva `copy`, devem ser transferidas do *host* para *device* antes da execução do *kernel*, como dado de entrada, e transferida de volta, no sentido GPU→ CPU, ao término da região paralela, como um dado de saída.

Um programa com diretivas OpenACC realiza, por definição, as transferências de dados de forma automática, analisando, no momento da compilação, a região acelerada do código e verificando quais dados são de entrada, saída ou temporários. Entretanto, o compilador toma decisões que podem ser consideradas conservadoras, assim, em muitas oportunidades, o programador deve intervir, alterando a forma de transferência dos dados de maneira a buscar a melhoria no desempenho computacional, uma vez que o tempo despendido neste processo de transferência não pode ser desprezado, podendo afetar negativamente no desempenho do programa. Outro exemplo envolvendo o mesmo assunto será tratado na seção seguinte, para melhorar o entendimento.

```
float saxpy_sum(int n, float a, float *x, float *restrict y) {
     float total = 0.;
     #pragma acc parallel copyin(x[0:n]) copy(y[0:n])
          {
          #pragma acc loop reduction (+:total)
          for (int i = 0; i < n; ++i) {
               y[i] = a * x[i] + y[i];
               total += y[i]; }
          } // fim da região paralela com transf. de dados
     return total;
     }
```

Algoritmo 2.12 - Exemplo de operação de redução com OpenACC

As demais possíveis cláusulas relacionadas a dados em construções paralelas, para códigos em OpenACC, são as seguintes:

- `copy(lista de variáveis e arrays)` - Aloca área de memória para os dados especificados na lista no acelerador, e copia-os do *host* para o *device* (CPU→ GPU) ao entrar na região, e do *device* para o *host* ao sair da região delimitada (GPU→CPU).
- `copyin(lista de variáveis e arrays)` - Aloca área de memória para os dados especificados na lista no acelerador, e copia-os do *host* para o *device* (CPU→GPU) ao entrar na região.
- `copyout(lista de variáveis e arrays)`- Aloca área de memória para os dados especificados na lista no acelerador, e copia-os do *device* para o *host* (CPU→GPU) ao sair da região.
- `create(lista de variáveis e arrays)`- Aloca área de memória para os dados especificados na lista no acelerador, e não realiza nenhuma transferência de dados entre *host* e *device*. Normalmente aplicável a dados temporários.
- `present(lista de variáveis e arrays)` - Indica que os dados citados na lista já estão presentes (ou alocados) no acelerador e devem ser usados. Também não realiza nenhuma transferência de dados entre *host* e *device*.
- `present_or_copy(lista de variáveis e arrays)` - Indica que, caso os dados citados na lista já estejam presentes (ou alocados) no acelerador devem ser usados e nenhuma transferência será realizada. Entretanto, caso os dados da lista não estejam presentes, procede da mesma forma que a cláusula `copy`.
- `present_or_copyin(lista de variáveis e arrays)` - Indica que, caso os dados citados na lista já estejam presentes (ou alocados) no acelerador devem ser usados e nenhuma transferência será realizada. Entretanto, caso os dados da lista não estejam presentes, procede da mesma forma que a cláusula `copyin`.

- `present_or_copyout(lista de variáveis e arrays)` - o mesmo que o anterior, mas para a cláusula `copyout`.
- `present_or_create(lista de variáveis e arrays)` - o mesmo que o anterior, mas para a cláusula `create`.
- `deviceptr(lista de variáveis e arrays)` - A lista deve especificar ponteiros para endereços de memória de dados já alocados no acelerador (tal como usado na função `acc_malloc`).

2.5.2.2. Estudo de caso com o programa Jogo da Vida

Da mesma forma como descrito na seção 2.4.1, será utilizado o programa Jogo da Vida como um estudo de caso para paralelização do código, seguindo o padrão OpenACC, a ser utilizado em um acelerador de hardware do tipo GPU. A versão sequencial a ser tomada como base será a mesma descrita na seção mencionada.

A análise de desempenho foi anteriormente efetuada para OpenMP, e identificou quatro regiões com potencial paralelismo:

a) Laço que atualiza valores de bordas periódicas a esquerda e direita do tabuleiro;
b) Laço que atualiza valores de bordas periódicas superior e inferior;
c) Laço que processa uma nova geração do tabuleiro (com maior relevância de carga computacional);
d) Laço que calcula o valor da somatória dos valores do tabuleiro.

As três primeiras regiões potenciais (ou trechos) estão aninhadas em outro laço, o qual itera sobre as gerações sucessivas do tabuleiro, enquanto o laço descrito pelo item (d) está isolado dos demais, conforme já citado na seção 2.4.1. Para a paralelização por OpenACC as mesmas regiões serão consideradas. Assim, uma primeira versão do algoritmo paralelo em OpenACC consiste em utilizar a diretiva **`parallel`** para paralelizar todos as regiões previamente descritas, com o devido cuidado com a última região que exigirá uma operação de redução, tal qual também ocorreu em OpenMP. Assim, a primeira versão que paraleliza os quatro laços a seguir pode ser conferida nos algoritmos 2.13, 2.14, 2.15 e 2.16.

```
4.       // Colunas esquerda e direita
5.       #pragma acc parallel
6.       #pragma loop
7.       for (i = 1; i<=dim; i++) {
8.             ...
9.             }
```

Algoritmo 2.13 - Laço paralelo em OpenACC para preenchimento de fronteiras esquerda e direita

```
12.       // Linhas superior e inferior
13.       #pragma acc parallel
14.       #pragma loop
15.       for (j = 0; j<=dim+1; j++) {
16.             ... }
```

Algoritmo 2.14 - Laço paralelo em OpenACC para preenchimento de fronteiras superior e inferior

```
18.       // Loop sobre as celulas para nova geracao
19.       #pragma acc parallel
20.       #pragma acc loop
21.       for (i = 1; i<=dim; i++) {
22.             for (j = 1; j<=dim; j++) {
23.                   ...
24.                   }
25.             }
```

Algoritmo 2.15 - Laço paralelo em OpenACC para cálculo de nova geração do tabuleiro

Cabe citar que tanto a diretiva `parallel` quanto a diretiva `kernels` poderiam ser igualmente aplicadas aos laços citados. Deve-se lembrar porém, que a diretiva `kernels` é utilizada em uma abordagem conservadora, deixando para o compilador a tarefa de analisar o trecho de código e decidir ou não pela geração de versão paralela acelerada, enquanto que a diretiva `parallel` força o compilador a paralelizar o trecho demarcado, deixando a responsabilidade para o programador definir sobre a aceleração da região. Nos exemplos aqui utilizados serão utilizados a diretiva `parallel` por conveniência.

O trecho relativo a operação de redução, para cálculo da somatória dos valores do tabuleiro, será paralelizado em OpenACC de acordo com o algoritmo 2.16.

```
43.   // Somatorio das celulas do tabuleiro
44.   int total = 0;
45.   #pragma acc parallel loop reduction(+:total)
46.   for (i = 1; i<=dim; i++) {
47.      for (j = 1; j<=dim; j++) {
48.            total += grid[i][j];
49.            }
50.      }
```

Algoritmo 2.16 - Laço paralelo em OpenACC para cálculo de somatória dos valores do tabuleiro por redução

Entretanto, ao compilar esta primeira versão e analisar as informações exibidas na tela pelo compilador PGI, percebeu-se algo fora do esperado, relacionado a transferência de dados entre CPU e GPU. O compilador implicitamente fez estas transferências para todas as quatro regiões paralelas, porém, na terceira região paralela, a transferência de dados não contemplou todos os elementos das matrizes **`grid`** e **`newGrid`** exigidas, conforme se vê abaixo:

```
80, Accelerator kernel generated
      Generating Tesla code
      82, #pragma acc loop gang /* blockIdx.x */
      83, #pragma acc loop vector(128) /* threadIdx.x */
  80, Generating implicit copyin(grid[1:2048][1:2048])
      Generating implicit copy(newGrid[1:2048][1:2048])
  85, getNeighbors inlined, size=16, file GOL-openacc2-v1.c (8)
       83, Loop is parallelizable
```

No texto acima referente a saída do compilador, a informação `Generating implicit copyin(grid[1:2048][1:2048])` bem como a informação `Generating implicit copy(newGrid[1:2048][1:2048])` significam que o *array* `grid` sofrerá uma transferência, da CPU para a GPU, para os dados que abrangem as células na faixa entre as linhas e colunas que vão de 1 até 2048. O array `newGrid` sofreu processo semelhante, porém, após o processamento no dispositivo, ocorrerá uma transferência no sentido oposto, ou seja, da GPU para CPU, abrangendo a mesma faixa de valores já citada. Ocorre que a aplicação das regras do Jogo da Vida exigirão que se façam leituras nos valores nas posições correspondentes às bordas das matrizes, ou seja, em valores nas linhas 0 e 2050, bem como nas colunas de mesma numeração. Entretanto, o compilador não detectou a necessidade de transferir valores contidos nesta faixa (nas bordas do tabuleiro), desta forma, não foram transferidos para a GPU, implicando em um *array* de tamanho menor a ser alocado no acelerador.

O resultado desta transferência incompleta de dados, que não contempla todos os valores necessários, acabou gerando um erro de execução. Assim, foi necessário corrigir o problema indicando explicitamente para o compilador a faixa de dados a ser utilizada nas transferências. O código que corrige este problema, relativo exclusivamente ao terceiro laço, pode ser conferido no algoritmo 2.17. As cláusulas `copyin` e `copy` especificam a exata faixa de valores das matrizes `grid` e `newgrid`, respectivamente, conforme exigido, iniciando no elemento de índice 0 até o último elemento dos *arrays,* em ambas as dimensões da matriz.

```
18.       // Loop sobre as celulas para nova geracao
19.       #pragma acc parallel \
20.             copyin(grid[0:fullSize][0:fullSize]) \
```

```
21.             copy(newGrid[0:fullSize][0:fullSize])
22.       #pragma acc loop
23.       for (i = 1; i<=dim; i++) {
24.             for (j = 1; j<=dim; j++) {
25.                   ...
26.                   }
27.             }
```

Algoritmo 2.17 - Laço paralelo corrigido em OpenACC para cálculo de nova geração do tabuleiro, contemplando toda a matriz

O desempenho desta primeira versão em OpenACC em comparação com a versão serial e com a melhor versão paralela em OpenMP com 28 threads foi muito abaixo do esperado, e pode ser conferido na tabela 2.2. A execução da versão em GPU por OpenACC se deu em uma GPU NVIDIA *TitanBlack*, a qual conta com 2880 núcleos (chamados de *cuda cores*) executando à 889 MHz, e memória de 6GB.

Tabela 2.2 - Desempenho da primeira versão paralela em GPU por OpenACC

Versão	Tempo (s)	Speedup
Serial	330,474	1
OpenMP v2 - 28 threads	15,463	21,37
OpenACC v1	1208,54	0,27

O desempenho considerado inesperado e pífio, obtido pela primeira versão em OpenACC, pode ser explicado devido ao tempo despendido com transferências de dados entre a CPU e a GPU, pois, uma vez que as três primeiras regiões paralelizadas estão dentro de um laço que se repete 10.000 vezes, essas transferências ocorrem 70.000 vezes durante o tempo de execução do código, sendo 20.000 correspondente ao primeiro laço, devido a 10.000 transferência CPU→ GPU antes do início e 10.000 transferências GPU→ CPU ao término do laço; mais 20.000 vezes correspondente ao segundo laço de forma semelhante, ambos devido ao array `grid`, e 30.000 para o terceiro laço, já que ocorre a transferência CPU→ GPU de `grid` e `newGrid` antes do início do mesmo, e apenas de `newGrid`, no sentido GPU→ CPU, ao término do laço.

Uma segunda versão deste código deverá minimizar a enorme quantidade de transferências de dados que ocorrem entre CPU e GPU. Desta forma, similarmente à técnica aplicada para a segunda versão em OpenMP, o laço principal, que percorre as sucessivas gerações do tabuleiro, receberá uma diretiva em OpenACC que indicará a transferência inicial de dados da CPU para a GPU, e a transferência no sentido oposto (GPU→ CPU) ao término desta seção. Esta nova versão deverá, portanto, delimitar uma região de dados, que compreende todo o laço principal, especificando que o *array* `grid` deverá ser transferido tanto na entrada quanto na saída, nos sentidos CPU→ GPU e

GPU→ CPU: `copy(grid[0:fullSize][0:fullSize])`, enquanto o *array* `newGrid` deverá ser utilizado apenas internamente, portanto podendo ser apenas criado por lá no dispositivo: `create(newGrid[0:fullSize][0:fullSize])`, pois não necessita ser retirado, já que após o seu cálculo, todo o seu conteúdo deverá ser copiado para o *array* `grid`, preparando-se para uma nova possível geração.

Um ressalva importante para esta nova versão do programa em OpenACC refere-se ao trecho de código responsável pela troca de ponteiros entre `grid` e `newGrid`, a qual terá de ser substituída por uma simples cópia entre os dois citados *arrays*, visto que não será possível efetuar a troca de ponteiros do código em um ambiente massivamente paralelo como a GPU, onde os ponteiros estão distribuídos pelos diversos módulos de processamento. Desta forma, o código final resultante pode ser conferido no algoritmo 2.18.

```
1. // loop principal para evolucao de geracoes
2. #pragma acc data copy(grid[0:fullSize][0:fullSize]) \
3.                  create(newGrid[0:fullSize][0:fullSize])
4. { // principal regiao paralela acelerada
5.       for (iter = 0; iter<maxIter; iter++) {
6.             // Colunas esquerda e direita
7.             #pragma acc parallel loop
8.             for (i = 1; i<=dim; i++) {
9.                   // copiar ultima coluna real para coluna
10.                  // esquerda de borda (left ghost column)
11.                  grid[i][0] = grid[i][dim];
12.                  // copiar primeira coluna real para coluna
13.                  // direita de borda (right ghost column)
14.                  grid[i][dim+1] = grid[i][1];
15.                  }
16.            // Linhas superior e inferior
17.            #pragma acc parallel loop
18.            for (j = 0; j<=dim+1; j++) {
19.                  grid[0][j] = grid[dim][j];
20.                  grid[dim+1][j] = grid[1][j];
21.                  }
22.
23.            // Loop sobre as celulas para nova geracao
24.            #pragma acc parallel loop
25.            for (i = 1; i<=dim; i++) {
26.                  for (j = 1; j<=dim; j++) {
27.                    // calcula numero de vizinhos
28.              int numNeighbors = getNeighbors(grid,i, j);
29.                    // aplicacao das regras do GOL
30.                    if (grid[i][j]==1 && numNeighbors<2)
31.                          newGrid[i][j] = 0;
32.                    else if (grid[i][j]==1 &&
33.                         (numNeighbors==2||numNeighbors==3))
34.                          newGrid[i][j] = 1;
35.                    else if (grid[i][j]==1 && numNeighbors>3)
36.                          newGrid[i][j] = 0;
37.                    else if (grid[i][j]==0 && numNeighbors==3)
38.                          newGrid[i][j] = 1;
39.                    else
```

```
40.                         newGrid[i][j] = grid[i][j];
41.                     }
42.             }
43.
44.             // troca de arrays para proxima geracao
45.             #pragma acc parallel loop
46.             for(i = 1; i <= dim; i++) {
47.                   for(j = 1; j <= dim; j++) {
48.                         grid[i][j] = newGrid[i][j];
49.                         }
50.                   } // copia de dados entre grid e newGrid
51.    } // Fim do laco principal
52.} // Fim da regiao paralela principal
```

Algoritmo 2.18 - Segunda versão de código OpenACC com otimização na transferência de dados entre CPU e GPU

O desempenho da segunda versão OpenACC pode ser conferido na tabela 2.3, onde se demonstra um *speedup* de pouco mais de 50 vezes no tempo de execução em relação a versão serial e de 2,37 vezes a melhor versão OpenMP executada.

Tabela 2.3 - Desempenho da primeira versão paralela em GPU por OpenACC

Versão	Tempo (s)	*Speedup*
Serial	330,474	1
OpenMP v2 - 28 threads	15,463	21,37
OpenACC v1	1208,54	0,27
OpenACC v2	6,50	50,84 (2,37 em relação a versão OpenMP v2)

2.6. Conclusão

O capítulo aqui apresentado abordou, de forma prática, através de exemplos em linguagem C, a programação por diretivas para processadores *multicores* e GPUs, utilizando-se de OpenMP e OpenACC, respectivamente. Espera-se que com o texto tutorial e os exemplos aqui demonstrados os leitores tenham capacidade de empregar corretamente as diretivas considerando as características específicas das arquiteturas paralelas de memória compartilhada sugeridas.

Referências bibliográficas

Adamatzky, A. (2010). “Game of Life Cellular Automata”. Springer-Verlag London Limited, 2010.

Amdahl, G. M. (1967). “Validity of the single processor approach to achieving large scale computing capabilities”, AFIPS Spring Joint Computer Conference, 1967.

Ben-Ari, M. (2005). “Principles of Concurrent and Distributed Programming”, Pearson, 2 edition, 2005.

Bernstein, A. J. Analysis of programs for parallel processing. IEEE Trans. on El. Computers, EC-15:757–762, 1966.

Breshears, C. (2009). “The art of concurrency: a thread monkey´s guide to writing parallel applications”. O´Reilly, 2009.

Butenhof, D. R. (2006) “Programming with POSIX Threads”, Addison-Wesley, 2006.

Chapman, B.; Jost, G.; Van der Pas, R. (2008) “Using OpenMP – Portable shared memory parallel programming”. The MIT Press, 2008.

Dowd, K., Severance, C. “High Performance Computing”. 2nd Edition, O'Reilly Media, 1998.

Escobar, Fernando A ; Xin Chang ; Valderrama, Carlos (2016) Suitability Analysis of FPGAs for Heterogeneous Platforms in HPC. IEEE Transactions on Parallel and Distributed Systems, Vol.27(2), pp.600-612. Fevereiro, 2016.

Flynn, Michael J. (1972). "Some Computer Organizations and Their Effectiveness". IEEE Transactions on Computers. C-21 (9): 948–960.doi:10.1109/TC.1972.5009071.

Gilge, M. (2014) IBM System Blue Gene Solution Blue Gene/Q: Application Development, IBM redbooks series, ISBN: 9780738438238, 2014.

Gustafson, J.L. Reevaluating Amdahl's law, Communications of the ACM, Vol. 31, Issue 5, May 1988.

Han, T.D.; Abdelrahman, T.S. hiCUDA: a high-level directive-based language for GPU programming. Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, Washington, D.C., p. 52-61, 2009.

Hassan, Mohamed W.; Helal, Ahmed E.; Athanas, Peter M.; Feng, Wu-chun, Hanaf;, Yasser Y. (2018) Exploring FPGA-specific Optimizations for Irregular OpenCL Applications. Em Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig), Cancun, Mexico, Dezembro de 2018.

Hennessy, D; Patterson, D. (2007) “Computer architecture”. 4ª. Ed, Elsevier, 2007.

Kirk, D. B., Hwu, W.W. (2010) “Programming massively parallel processors: a hands-on approach”. Morgan Kaufman, 2010.

Mattson, T.G., Sanders, B., Massingill, B. “Patterns for Parallel Programming”, Addison-Wesley Professional, 2004.

McCool, M., Reinders, J. and Robinson, A. (2012) Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann, 1st edition.

NVIDIA (2018) "CUDA Toolkit Documentation", NVIDIA. Version 10.0.130, Outubro 2018. Disponível em https://docs.nvidia.com/cuda. Acessado em janeiro de 2019.

OPENACC (2019) OpenACC website disponível em https://www.openacc.org. Acessado em janeiro de 2019.

Pacheco, P.S. "An Introduction to Parallel Programming", Morgan Kaufmann, 2011.

Stallings, W. (2009) "Computer organization and architecture: designing for perfomance", 8ª. Ed., Prentice Hall, 2009.

TOP500 (2018) disponível em http://www.top500.org, acessado em janeiro de 2019.

Ueng, s.; Lathara, M.; Baghsorkhi, S.S.; Hwu, W.; CUDA-lite: Reducing GPU Programming Complexity. Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008.

Van der Pas, Ruud; Stotzer, Eric; Terboven, Christian. (2017) Using OpenMP—The Next Step: Affinity, Accelerators, Tasking, and SIMD. The MIT press, Scientific and Engineering Computation series. Outubro, 2017