Capítulo
5

Programando Aplicações com Diretivas Paralelas

Natiele Lucca, Claudio Schepke
Universidade Federal do Pampa
Alegrete, Brasil

Resumo

Uma das motivações para o desenvolvimento de programas paralelos é acelerar aplicações científicas. Aplicações deste tipo geralmente demandam de um grande tempo de computação para uma versão com um único fluxo de execução, o que pode levar minutos, horas ou até mesmo dias, dependendo do tamanho do domínio ou resolução do problema adotado. Uma das maneiras de gerar paralelismo a partir de um código é inserir diretivas (pragmas), os quais geram fluxos concorrentes de código, que podem ser executados tanto em arquiteturas multi-core como many-core. Neste sentido, este minicurso tem como objetivo apresentar técnicas de exploração de paralelismo em diferentes trechos de código para um conjunto de aplicações científicas usando as interfaces de programação OpenMP e OpenACC. Serão demonstrados exemplos reais do impacto do uso de pragmas no desempenho de códigos, incluindo situações em que a granularidade impede que se obtenha a aceleração do programa.

5.1. Introdução

Este Capítulo aborda a prática de programação com diretivas paralelos. Diretivas de pré-compilação possibilitam a geração de código específico e automatizado. Inicialmente serão vistos alguns conceitos sobre as arquiteturas de programação Multi-core e GPUs. Na sequência também serão introduzidas as interfaces de programação OpenMP e OpenACC, apresentando a estrutura e organização de suas diretivas. Por fim, serão mostrados os códigos-fonte de 2 aplicações científicas, destacando o objetivo de cada uma, bem como as funcionalidades existentes. A partir disso, uma avaliação do custo sequencial de cada função ou rotina de código será então realizada. Trechos de código previamente selecionados serão indicados para que abordagens de paralelização sejam utilizadas, avaliando o impacto na aceleração da aplicação. Com isso, o objetivo é testar e validar diretivas que possam reduzir o tempo total de execução da aplicação.

5.2. Introdução às arquiteturas multi-Core e GPU

O crescente desenvolvimento computacional previsto pela Lei de Moore (MOORE, 1965) possibilitou executar *softwares* cada vez mais rápidos, incluindo melhores funcionalidades e se tornando mais úteis para os usuários. Estes cada vez mais requeriam desta tecnologia, gerando uma fase positiva para a indústria de computadores, que se responsabilizava por manter o contínuo desenvolvimento, praticamente dobrando o número de transistores por processador a aproximadamente cada dois anos ao mesmo tempo em que mantinha os custos.

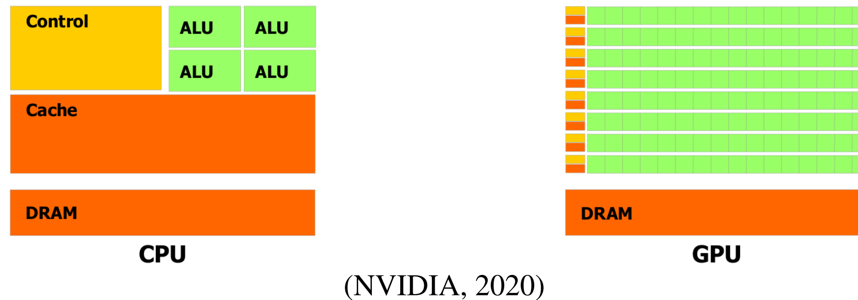
O fato é que essa tendência diminuiu a partir do ano de 2003 devido ao consumo elevado de energia e problemas com a dissipação de calor nos processadores (KIRK; WEN-MEI, 2016). Estes problemas se tornaram fatores limitantes para o aumento da velocidade de *clock* dos processadores, o que levou os fabricantes de processadores a buscarem por novas estratégias para se obter melhor desempenho, tendo um maior cuidado com a eficiência energética e a dissipação de calor. A solução para se obter melhor desempenho considerando os obstáculos mencionados foi a de começar a produzir processadores compostos por dois ou mais *cores*, que são processadores mais simples e com menor frequência de *clock*. Isto fez com que surgissem os processadores *multi-core*, que são processadores que contém múltiplas unidades de processamento, permitindo assim, a execução de instruções de forma paralela. Este fato, de grande importância, foi nomeado por Sutter e Larus (2005) como a revolução da concorrência, para a qual a indústria de *software* deve estar preparada, sendo capaz de produzir aplicações que utilizem os recursos oferecidos pelas arquiteturas paralelas de forma eficiente.

Em contraste com os processadores *multi-core*, existem os chamados processadores *manycore*, também desenvolvidos pensando na execução paralela de instruções. Apesar de que estas duas arquiteturas tenham sido desenvolvidas com o mesmo objetivo, existem diferenças na forma com que são organizadas e as estratégias que utilizam para se obter poder de computação. Enquanto as arquiteturas *multi-core* foram desenvolvidas para manter a velocidade de programas sequenciais enquanto faziam a transição para múltiplos processadores, os dispositivos de arquitetura *manycore* são voltados para a execução de dezenas, centenas ou até milhares de instruções simultâneas (KIRK; WEN-MEI, 2016).

Na Figura 5.2 pode-se perceber os elementos que representam as diferenças entre as duas arquiteturas. À esquerda tem-se a representação de uma CPU *multi-core* e à direita de uma GPU *manycore*.

A arquitetura *multi-core* é otimizada para a performance de código sequencial, contendo uma lógica de controle sofisticada que permite a execução em paralelo de instruções de uma *thread*, somado a existência de uma grande memória *cache* para reduzir a latência de acesso aos dados e instruções necessários por uma aplicação, além de cada ULA também ser otimizada para reduzir a latência. Já as arquiteturas *manycore* possuem um *hardware* especificamente projetado para suportar um grande número de *threads*, com memórias *cache* de pequeno porte, visando uma redução ao acesso à memória principal nos casos em que múltiplas *threads* acessam os mesmo dados, maximizando, desta forma, o espaço dedicado para as ULAs, que também são mais simplificadas.

Figura 5.1. Representação das arquiteturas *multi-core* e *manycore*



(NVIDIA, 2020)

Com isto pode-se perceber que cada arquitetura é especializada em um tipo de tarefa diferente, sendo os processadores *multi-core* voltados tanto para a performance paralela quanto sequencial, dando ênfase para o desempenho individual das *threads*, porém limitado a um número menor de *cores* quando comparado com um processador *manycore*. Arquiteturas *manycore* são totalmente focadas e otimizadas para terem níveis altíssimos de paralelismo, tendo muitos *cores* mais simples e com menos performance individual. Uma arquitetura heterogênea é aquela que combina as duas arquiteturas, aproveitando o melhor das duas estratégias para alcançar máxima eficiência computacional.

Para alcançar este melhor desempenho oferecido pelas arquiteturas paralelas, é necessário expressar o paralelismo nos programas através da programação paralela. Este paradigma de programação permite ao programador especificar as áreas de código que devem ser executadas concorrentemente entre núcleos de um dispositivo paralelo, garantindo assim uma utilização mais eficiente do poder computacional proporcionado por estes dispositivos. Dada a importância do desenvolvimento de aplicações paralelas, atualmente existem diversas *Application Programming Interface* (APIs) voltadas para este fim, a maioria delas já oferece suporte para a programação de GPUs ou foram projetadas para este fim, como é o caso das APIs CUDA, OpenACC e OpenCL.

5.3. A Interface de programação OpenMP

OpenMP é uma API para programação paralela de memória compartilhada e multiplataforma disponível em C/C++ e Fortran (OPENMP, 2020). A API é fundamentada no modelo de execução *fork-join*. Esse modelo possui uma *thread* mestre que inicia a execução e gera *threads* de trabalho para executar as tarefas em paralelo (CHAPMAN; MEHROTRA; ZIMA, 1998). O OpenMP aplica o modelo de execução *fork-join* em segmentos do código que são informados pelo programador (ou usuário). Dessa forma um código sequencial é executado pela *thread* mestre até um bloco ou área de execução paralela.

O inicio da área paralela é demarcado por uma diretiva OpenMP que é responsável por sinalizar que as *threads* de trabalho devem ser lançadas (*fork*). Todo o código seguinte é executado em paralelo pelas *threads* até o fim da área paralela que pode ser demarcado explicitamente como o símbolo de } ou implícito, como por exemplo, em um laço de repetição `for` onde o fim do laço de repetição também é o fim da área paralela. O fim da área paralela implica no encerramento das *threads* de trabalho, sincronização (*fork*) e retorno da *thread* mestre para a execução.

A API OpenMP possui um conjunto de diretivas de compilação, funções e variáveis de ambiente para a programação paralela (TORELLI; BRUNO, 2004). Uma diretiva é precedida obrigatoriamente por `#pragma omp` e seguida por `[atributos]`, sendo que os atributos são opcionais. Seguem algumas diretivas que compõem a API OpenMP (OPENMP, 2020).

- `parallel`: Essa diretiva descreve que a uma área do código será executada por *n threads*, sendo *n* o número de *threads* especificados por um atributo e/ou variável de ambiente.
- `for`: Essa diretiva especifica que as iterações do laço de repetição serão executadas em paralelo por *n threads*.
- `parallel for`: Especifica a construção de um laço paralelo, sendo que o laço será executado por *n threads*.
- `simd`: Essa diretiva descreve que algumas iterações de um laço de repetição podem ser executadas simultaneamente por unidades vetoriais.
- `for simd`: Essa diretiva especifica que um laço pode ser dividido em *n threads* que executam algumas iterações simultaneamente por unidades vetoriais.

Na sequência são apresentados alguns atributos da API OpenMP (OPENMP, 2020). Para todos os casos, `lista` representa uma ou mais variáveis.

- `private (lista)`: Esse atributo informa que o bloco paralelo possui variáveis privadas para cada uma das *n threads*. As variáveis do bloco que não são informadas na lista são públicas.
- `shared (lista)`: O atributo especifica que as variáveis são públicas e compartilhadas entre as *n threads*.
- `num_threads(int)`: Esse atributo determina o número *n* de *threads* utilizadas no bloco paralelo. O valor de *n* é válido apenas para o bloco em que foi definido.
- `reduction (operador: lista)`: A redução é utilizada para executar cálculos em paralelos. Cada *thread* tem seu valor parcial. Ao final da região paralela o valor final da variável é atualizado com os cálculos parciais. O operador pode ser, por exemplo $+$, $-$, $*$, *max* e *min*.
- `nowait`: Uma diretiva OpenMP possui uma barreira implícita ao seu fim, com o objetivo de garantir a sincronização. Entretanto a diretiva `nowait` omite a existência dessa barreira. Dessa forma, as *threads* não ficam em espera até que as demais também terminem o trabalho.

As variáveis de ambiente do OpenMP especificam características que afetam a execução dos programas. Seguem algumas variáveis (OPENMP, 2020):

- `OMP_NUM_THREADS`: Especifica o número *n* de *threads* utilizados nos blocos paralelos do algoritmo.
- `OMP_THREAD_LIMIT`: Descreve o número máximo de *threads*.
- `OMP_NESTED`: Permite ativar ou desativar o paralelismo aninhado.
- `OMP_STACKSIZE`: Especifica o tamanho da pilha para as *threads*.

5.4. A Interface de programação OpenACC

O OpenACC é uma API que contém um conjunto de diretivas de compilação para GPUs, similar às fornecidas por OpenMP para CPU (CHANDRASEKARAN; JUCKELAND, 2017). Enquanto uma área paralela no OpenMP é executada por *threads* de trabalho localizadas na CPU o OpenACC especifica instruções que criam *threads* localizadas na GPU (OPENACC, 2019).

Uma diretiva OpenACC em C/C++ possui o seguinte formato (OPENACC, 2019):

```
#pragma acc diretiva [clausulas [,] clausulas]
```

Em Fortran a diretiva possui o seguinte formato (OPENACC, 2019):

```
!$acc diretiva [clausulas [,] clausulas]
```

O OpenACC possui três níveis de paralelismo, são eles: *gang*, *worker* e *vector*. A *gang* possui um ou mais *workers*, sendo que nos *workers* podem ocorrer operações vetoriais (*vector*).

Seguem algumas diretivas que compõem a API OpenACC (OPENACC, 2019).

- `parallel`: Essa diretiva descreve um bloco em paralelo executado por *n threads* disponíveis.
- `kernels`: Especifica que o compilador dividirá o bloco paralelo em *kernels*.
- `loop`: Paraleliza o(s) loop(s) aninhados imediatamente após a diretiva.
- `serial`: Essa diretiva especifica um bloco de código que será executado sequencialmente.
- `data`: Define um bloco no qual os dados são acessíveis pela GPU.

Seguem algumas cláusulas que compõem a API OpenACC (OPENACC, 2019). Para todos os casos, *lista* representa uma ou mais variáveis.

- `copy (lista)`: Essa cláusula copia os dados do *host* para a GPU e da GPU para o *host*.
- `copyin (lista)`: Essa cláusula copia os dados do *host* para a GPU.
- `copyout (lista)`: Essa cláusula copia os dados da GPU para o *host*.

- `present (lista)`: Essa cláusula informa que os dados utilizados no bloco paralelo já foram copiados para a memória da GPU.

A API OpenACC possui variáveis de ambiente que especificam alterações na execução dos programas. Para as ocorrências, *type* representa o tipo do dispositivo e *size* o tamanho de memória que será alocado. Seguem algumas variáveis (OPENACC, 2019):

- `acc_get_num_devices (type)`: Retorna o número de dispositivos do tipo especificado.
- `acc_set_device_type (type)`: Define o tipo de dispositivo utilizado.
- `acc_get_device_type()`: Retorna o tipo de dispositivo que está sendo usado pela *thread*.
- `acc_wait_all()`: Aguarda até que todas as atividades assíncronas sejam concluídas.
- `acc_malloc(size)`: Retorna o endereço da memória alocada no dispositivo.
- `acc_is_present`: Testa se os dados do *host* já foram copiados para a GPU.

5.5. Exemplos de aplicações científicas

Duas aplicações científicas foram consideradas para a inserção de diretivas paralelas. A primeira aplicação engloba o uso de algoritmos bio-inspirados para a otimização de problemas. A segunda aplicação realiza a simulação numérica de uma câmara de combustão usando as Equações de Navier-Stokes.

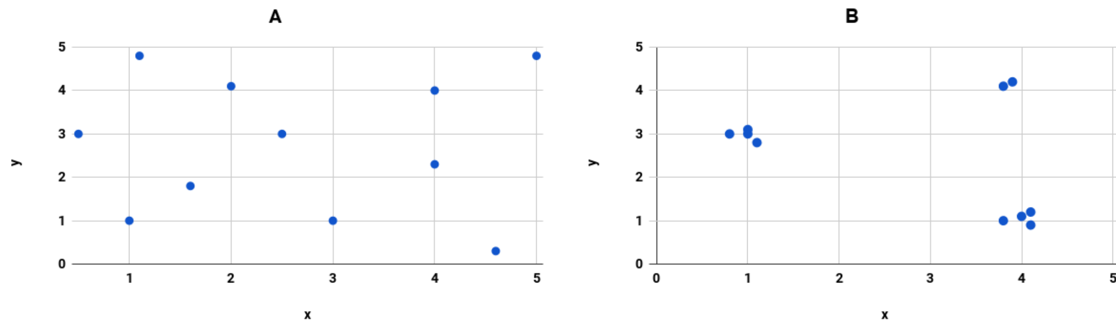
5.5.1. Algoritmos bio-inspirados

Problemas reais de engenharia, ciência e economia por exemplo, não podem ser resolvidos de forma exata e sequencial devido ao elevado tempo de computação para encontrar a solução ótima (IGNÁCIO; FERREIRA, 2002). A capacidade de processamento de um computador não é suficiente para realizar o esforço matemático exigido para encontrar a solução desses problemas. Para isso é necessário que os algoritmos utilizem abordagens que otimizem a execução das tarefas, assim podem solucionar esses problemas de forma eficiente (NIEVERGELT et al., 1995).

A computação natural é uma estratégia de otimização que aplica conceitos da biologia na solução de problemas complexos (CASTRO et al., 2004). Esses problemas possuem um grande número de variáveis ou soluções potenciais, sendo que nem sempre é possível garantir que uma solução encontrada pelo algoritmo seja ótima (CASTRO, 2007).

São exemplos de problemas complexos, os clássicos caixeiro viajante e roteamento de veículos. Outras aplicações são: o trabalho de Lin e Lucas (2015) que apresenta um modelo de evacuação de aeronaves sob emergência com emoções para simular o efeito de passageiros com medo ou pânico; a pesquisa de Diciolla et al. (2015) que classifica e

Figura 5.2. Malha Aleatória x Malha Convergindo



gera uma previsão para pacientes com doença renal terminal (DRC); e o trabalho de Carvalho et al. (2016) que apresenta um algoritmo bio-inspirado para controlar o tráfego em sistemas de elevadores.

Os algoritmos bio-inspirados aplicam técnicas de inteligência de enxames ou inteligência de colônias, ou ainda inteligência coletiva, que simulam o comportamento coletivo de sistemas auto-organizados, distribuídos, autônomos, flexíveis e dinâmicos (SERAPIAO, 2009). Um enxame ou colônia é uma população de elementos que interagem e são capazes de otimizar um objetivo global através da busca colaborativa em um espaço seguindo regras específicas.(KENNEDY; EBERHART, 2001).

A Figura 5.2-*A* mostra uma malha aleatória e a Figura 5.2-*B* uma malha convergindo. A primeira malha representa a inicialização do algoritmo onde os indivíduos ou agentes são dispostos de forma aleatória pelo espaço de busca. Já a malha convergindo expressa a solução quando os indivíduos estão aglomerados em um ou mais grupos que representam as melhores soluções encontradas.

Nesse contexto, a Computação natural ou computação bio-inspirada é o processo de extrair ideias da natureza para desenvolver sistemas computacionais. Os algoritmos inspirados na natureza tem a capacidade de descrever e resolver problemas complexos. Portanto, a computação natural pode ser definida como um campo de pesquisa que, baseado ou inspirado na natureza, permite o desenvolvimento de novas ferramentas para solução de problemas, através da síntese de padrões e de comportamentos (CASTRO, 2007).

A computação inspirada na natureza é subdivida em algoritmos inteligentes que incluem redes neurais artificiais, computação evolutiva, inteligência de enxame, sistemas imunológicos artificiais e outros (ENGELBRECHT, 2007). A inteligência de enxame simula o comportamento de agentes diante da colônia. Esses são inteligentes e autônomos não só realizam tarefas, mas tem a capacidade de tomar decisões (KENNEDY; EBERHART, 2001). São exemplos ACO, ABC, BCO e PSO. A computação evolutiva é baseada na evolução natural das espécies, onde a população se reproduz transferindo e combinando aos outros indivíduos suas características genéticas (BANZHAF et al., 1998). São exemplos de algoritmos dessa área da computação bio-inspirada o GA, DE, SA, ESA

e ES. As redes neurais artificiais simulam a capacidade cognitiva do ser humano, através dos neurônios que interconectados permitem a troca de informação (RAUBER, 2005).

No minicurso é abordado o algoritmo de inteligência de enxame *Particle Swarm Optimization* (Enxame de Partículas - PSO).

5.5.2. Simulação de uma câmara de combustão

Um processo de combustão ocorre quando um material combustível reage com um material reagente. O uso de modelos para a investigação paramétrica de diferentes condições de fluxo relacionadas a problemas de engenharia aeroespacial permite análises rápidas e confiáveis. A camada de mistura compressível serve como um modelo para a análise de problemas de propulsão considerando a passagem de ar em alta velocidade, como a mistura de reagentes em uma câmara de combustão ou a geração de ruído nos bicos de exaustão. Em ambos os casos, duas correntes paralelas a velocidades diferentes podem ser compostas por diferentes espécies químicas ou com gradientes de alta temperatura. Nesses casos, a variação das propriedades pode resultar em diferenças significativas nas características de estabilidade do fluxo, como taxas de crescimento e topologia do fluxo (SILVA et al., 2017).

Para simular uma camada de mistura, tem-se a implementação de uma aplicação. Essa implementação pode ser usada para otimizar o tamanho da dimensão da camada mista e decidir o melhor combustível e condição para um cenário específico. O aplicativo discretiza a equação completa de Navier-Stokes para uma representação bidimensional. A solução numérica considera o método Runge-Kutta usando um esquema de 4ª ordem para o tempo e um esquema de 6ª ordem para o espaço.

No entanto, esse tipo de aplicativo exige muito tempo de execução para qualquer simulação simples. Para reduzir a execução por um tempo aceitável, pode-se explorar a concorrência das instruções nas arquiteturas Multi-Core e GPU. Essas arquiteturas tradicionalmente tem sido adotadas para o desenvolvimento de aplicações numéricas.

A Figura 5.3 apresenta uma relação sistemática entre as funções que compõem a aplicação e suas localizações nos arquivos. O código inicia com o início da execução das chamadas disponíveis no arquivo euler.f90. O primeiro passo a ser feito para iniciar a paralelização é identificar as funções que demandam mais tempo de processamento. Para uma aplicação sequencial, é possível utilizar a ferramenta Gprof (GRAHAM; KESSLER; MCKUSICK, 1982).

5.6. Avaliando a performance de trechos de código

Um código possui trechos que não são paralelizáveis. Dessa forma, é necessário um estudo prévio que realize uma análise e identifique a(s) área(s) que podem ser paralelizadas.

A dependência entre os dados é um fator importante na paralelização de aplicações, uma vez que, pode gerar resultados incorretos devido a acesso de memória com valores ainda não atualizados. Uma característica que deve ser evitada na programação paralela são sincronizações sucessivas em blocos paralelos. Um bloco paralelo realiza o lançamento de *n threads*, entretanto sucessivas pausas para sincronizar os resultados parciais reduzem significativamente a eficiência gerada pelo paralelismo.

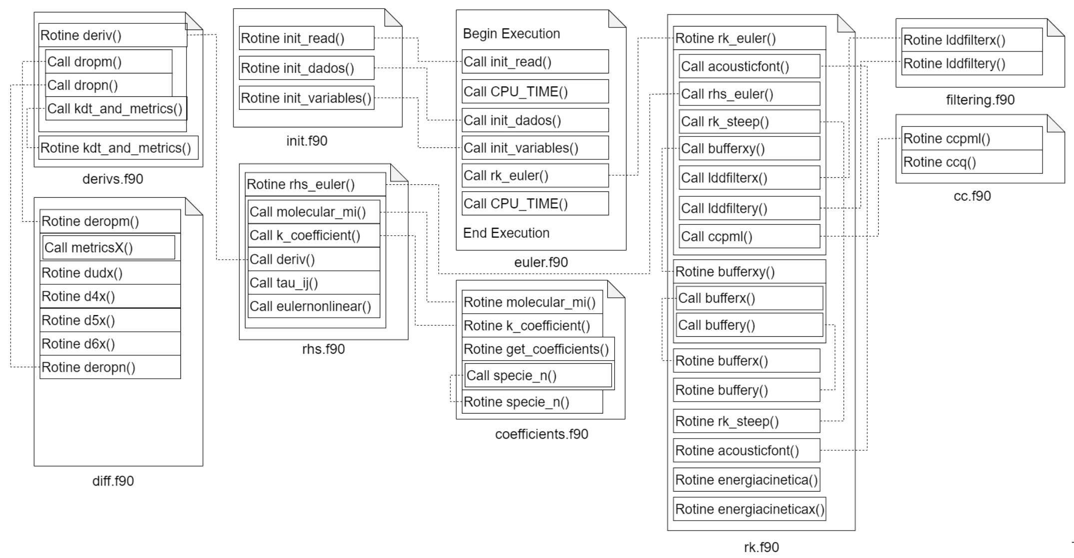


Figura 5.3. Relação esquemática entre as funções

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

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

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

Quando o tempo paralelo é exatamente igual ao tempo sequencial, o *speedup* é igual a 1. Neste caso não há ganho de desempenho. Uma outra forma de mensurar o quanto uma versão paralela é melhor que a versão sequencial é considerar o percentual de ganho de desempenho apresentado na Equação 2.

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

Nessa seção é apresentada uma aplicação que não obtém um ganho desempenho significativo devido as características presentes no domínio e ao volume de dados. A outra aplicação possibilita o ganho expressivo de desempenho em certas funcionalidades do código. Para ambos os casos, o mesmo ambiente de execução foi utilizado conforme

Tabela 5.1. Ambiente de execução: CPU

Características	Xeon E5-2650 (×2)
Frequência	2.00 GHz
Núcleos	8 (×2)
Threads	16 (×2)
Cache L1	32 KB
Cache L2	256 KB
Cache L3	20 MB
Memória RAM	128 GB

Tabela 5.2. Ambiente de execução: GPU

Características	Quadro M5000
Frequência	1.04 GHz
CUDA *cores*	2048
Cache L1	64 KB
Cache L2	2 MB
Memória Global	8 GB

descrito em na Tabela 5.1 e Tabela 5.2.

5.6.1. Aplicando diretivas OpenMP em algoritmos bio-inspirados

Serapiao (2009) define a inteligência de enxames ou inteligência de colônias, ou ainda inteligência coletiva, é um conjunto de técnicas baseadas no comportamento coletivo de sistemas auto-organizados, distribuídos, autônomos, flexíveis e dinâmicos. Os algoritmos que aplicam essa técnica simulam a forma com que o enxame (grupo de agentes) interagem e a forma com que tomam decisões.

A Figura 5.4 apresenta um algoritmo de inteligência de enxame genérico. Inicialmente o algoritmo gera uma população inicial que normalmente é aleatória (critério do desenvolvedor). Em seguida essa população é avaliada de acordo com uma função de qualidade, analisada e em caso de ótimo local armazenada. O critério de parada é uma maneira de controlar a quantidade de iterações que serão realizadas no algoritmo, caso não satisfeito as alterações são realizadas sob a população de acordo com a estratégia de cada algoritmo a fim de melhorar e alcançar a função objetivo; ao fim das iterações a melhor solução é retornada.

O algoritmo otimização por Enxame de Partículas (PSO) foi modelado a partir do comportamento social do bando de aves (ENGELBRECHT, 2007). No PSO, a população de indivíduos ou partículas é agrupada em um enxame (conjunto de soluções). Essas partículas simulam o comportamento de pássaros através do aprendizado próprio (componente cognitivo) e o aprendizado do bando (componente social), ou seja, imitam o seu próprio sucesso e o de indivíduos vizinhos (TAVARES; NEDJAH; MOURELLE, 2015). A partir desse comportamento as partículas podem definir novas posições que as direcionam para soluções ótimas.

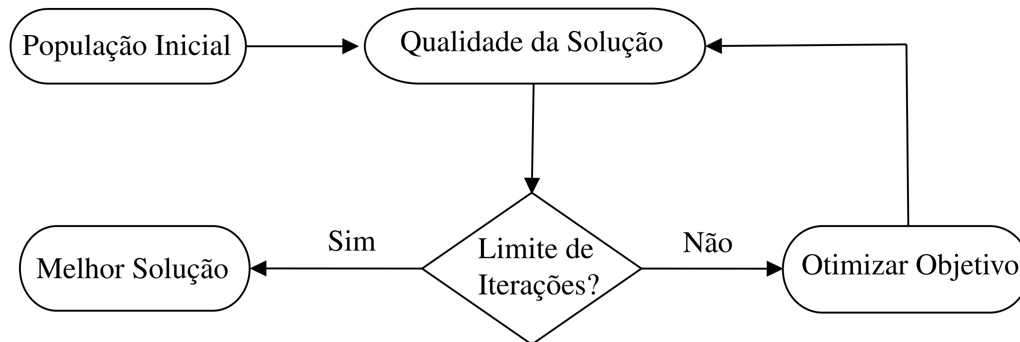


Figura 5.4. Representação da estrutura lógica de um algoritmos de inteligencia de enxame

Algoritmo 1: Pseudocódigo PSO

```
1  início
2      Inicializar as partículas ;                // Soluções iniciais
3      Inicializar a velocidade das partículas ;
4      enquanto critério de parada não for satisfeito faça
5          Calcular a aptidão das soluções;
6          Memorizar a melhor solução local
7          Memorizar a melhor solução global
8          Atualizar a velocidade das partículas ;
9          Atualizar a posição das partículas
10     fim
11     retorna Melhor Solução Encontrada ;
12 fim
```

```
1 double sphere(vector<double> temp){
2   double sum=0;
3   #pragma omp simd reduction(+:sum)
4   for(int i=0; i<temp.size() ;i++){
5     sum += pow(temp[i],2);
6   }
7   return sum;
8 }
```

Figura 5.5. Segmento de código paralelo - PSO

Função	PSO			
	Melhor Solução	**Tempo Sequencial (Desv. Padrão)**	**Tempo Paralelo (Desv. Padrão)**	**Taxa de Desempenho (%)**
1. Alpine	1,95E+00	0,370068 (0,0253)	0,091106 (0,0113)	306,19
2. Booth	1,47E-05	0,012690 (0,0014)	0,001806 (0,0002)	602,84
3. Easom	-9,64E-01	0,104991 (0,0070)	0,026976 (0,0013)	289,20
4. Griewank	1,20E+02	3,792227 (0,3100)	1,167276 (0,1064)	224,88
5. Rastrigin	3,05E+02	3,947903 (0,3863)	1,047876 (0,1123)	276,75
6. Rosenbrock	1,30E+05	2,737766 (0,2457)	0,129914 (0,0711)	2007,38
7. Sphere	1,31E-02	0,085171 (0,0115)	0,006861 (0,0027)	1141,37

Tabela 5.3. Casos de Teste - Melhores Resultados.

As instruções OpenMP no algoritmo PSO foram inseridas na função objetivo e na função de atualização do enxame. As funções Booth e Easom são bidimensionais, logo o laço de repetição foi substituído por uma instrução que compreende o somatório das duas dimensões, as demais funções N-dimensionais possuem um laço de repetição como expresso na Figura 5.5. Esse laço é precedido pela diretiva OpenMP `#pragma omp simd reduction(+: sum)`. O outro segmento de código paralelizado é o método que atualiza a posição de cada partícula e em seguida atualiza a velocidade de cada partícula. A diretiva OpenMP aplicada é `#pragma omp simd`. Para a execução são inseridas *flags* de compilação para sinalizar ao compilador instruções vetoriais do tipo AVX.

O *benchmark* Sphere apresentado na Figura 5.5 foi testado para enxames de diversos tamanhos. A Tabela 5.3 mostra o tamanho da população testado, o tempo sequencial e paralelo com os respectivos desvios padrões, o ganho de desempenho expresso em percentual pela equação 2 e a média das soluções. A média é obtida a partir de 30 execuções.

5.6.2. Aplicando diretivas OpenACC em aplicação de combustão

A aplicação escolhida simula o processo de mistura do combustível de uma aeronave e o material reagente, como detalhado na Seção 5.5.2. O código possui duas versões, uma na linguagem de programação C++ e a outra na linguagem Fortran. A estrutura do código é robusta. Esse código é subdivido em diversos arquivos e métodos como expresso na 5.3.

A Tabela 5.4, apresenta o tempo de execução de cada uma das funções mais expressivas de tempo para uma execução de um domínio de 521 (x) por 481 (y) pontos

Nome da Função	Porcentagem do Tempo Total de Execução
deropn	24,11 %
rhs_euler	19,24 %
deropm	17,46 %
lddfilterx	7,92 %
lddfiltery	7,04 %
rk_euler	5,23 %
eulernonlinear	4,68 %
metricsx	3,72 %
metricsy	2,71 %
molecular_mi	1,93 %
bufferxy	1,48 %
k_coefficient	1,19 %
deriv	1,11 %

Tabela 5.4. Porcentagem do tempo de execução usando GProf

coletados com a ferramenta GProf. Pode-se observar que não há uma função que domina o tempo total de execução e que as funções cujas operações atuam sobre o eixo x demandam um tempo mais expressivo do que as sobre o eixo y, dado que o domínio em x é maior que o em y para o problema em questão. O fato de uma chamada demandar de mais tempo de processamento não significa que esta terá o maior potencial de paralelização. No caso de implementações em GPU por exemplo, o tempo de transferência e sincronização dos dados pode tornar o ganho paralelo pouco efetivo para compensar a transferência de dados.

Inicialmente o código é paralelizado com a API OpenMP, sendo registrado quais modificações foram realizadas e quais os respectivos tempos de execução. O tempo de execução é mensurado a partir da *flag* `time` precedendo o script de execução. Outra informação que deve ser registrada é se o resultado obtido entre a versão sequencial e a paralela é igual. Caso o resultado não for igual o paralelismo está incorreto e não pode ser utilizado, sem o devido tratamento das operações.

O código também pode ser paralelizado com diretivas da API OpenACC. Nem todo segmento de código que contenha um `pragma` de OpenMP pode ser substituído por um `pragma` de OpenACC. Lembrando que os dados utilizados nos métodos paralelizados devem ter sido copiados para a GPU. Através da paralelização do código em OpenACC é possível observar e comparar o tempo de execução da aplicação sobre diferentes APIs.

A Tabela 5.5 apresenta o ganho de desempenho aplicando algumas otimizações em apenas algumas funções da aplicação. Na tabela, pode-se observar a redução do tempo de processamento. Outras funções também podem ser otimizadas para diminuir mais ainda o tempo de processamento.

Tipo	Tempo (s)	Ganho de Desemp (%)
Sequencial	126,6606	-
OpenACC	113,0066	12,08

Tabela 5.5. Taxa de ganhos de desempenho.

5.7. Conclusão

Paralelizar uma aplicação possui desafios. Muitas vezes é necessário reescrever ou realizar adaptações no código sequencial. Posteriormente, deve-se partir para a paralelização do problema. Uma técnica escolhida para o paralelismo pode não ser a mais adequada ao domínio do problema ou as características que ele possui. Neste caso, outras abordagens devem ser testadas. Também é preciso garantir a equivalência numérica dos resultados, ou seja, uma versão paralela não pode resultar em valores inconsistentes da solução do problema.

Nem sempre é possível obter ganho de desempenho com a execução paralela de trechos de código, por mais fino que o paralelismo seja aplicado, uma vez que a granularidade é baixa, ou a cópia de dados envolve um sobrecusto não compensado pelo paralelismo. Este é o caso do algoritmo bio-inspirado avaliado nesse capítulo que serve de exemplo como estudo de caso para situações onde a granularidade paralela é bastante baixa.

Em relação a aplicação de simulação de combustão esta possui um grande conjunto de funções que podem se paralelizadas tanto em OpenMP como em OpenACC. A comparação realizada nas atividades permite perceber o impacto da granularidade das aplicações. Assim como a viabilidade do uso da GPU e custo de alterações. Estas mesmas soluções podem ser aplicadas a outras classes de aplicações.

Referências

BANZHAF, W. et al. *Genetic programming: an introduction*. California: Morgan Kaufmann San Francisco, 1998. v. 1. páginas 7

CARVALHO, G. C. de et al. Otimização com algoritmo bio-inspirado de controle de tráfego em sistemas de grupos de elevadores. *Revista Interdisciplinar de Pesquisa em Engenharia*, v. 2, n. 9, p. 56–76, 2016. páginas 7

CASTRO, L. N. de. Fundamentals of natural computing: an overview. *Physics of Life Reviews*, Elsevier, v. 4, n. 1, p. 1–36, 2007. páginas 6, 7

CASTRO, L. N. de et al. Computação natural: Uma breve visão geral. In: *Workshop em nanotecnologia e Computação Inspirada na Biologia*. Santos, São Paulo: Universidade Católica de Santos (UniSantos), 2004. páginas 6

CHANDRASEKARAN, S.; JUCKELAND, G. *OpenACC for Programmers: Concepts and Strategies*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2017. ISBN 0134694287. páginas 5

CHAPMAN, B.; MEHROTRA, P.; ZIMA, H. Enhancing openmp with features for locality control. In: CITESEER. *Proc. ECWMF Workshop" Towards Teracomputing-The Use of Parallel Processors in Meteorology*. Austrian: PSU, 1998. páginas 3

DICIOLLA, M. et al. Patient classification and outcome prediction in iga nephropathy. *Computers in biology and medicine*, Elsevier, v. 66, p. 278–286, 2015. páginas 6

ENGELBRECHT, A. P. *Computational intelligence: an introduction*. South Africa: John Wiley & Sons, 2007. páginas 7, 10

GRAHAM, S. L.; KESSLER, P. B.; MCKUSICK, M. K. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 17, n. 6, p. 120–126, jun. 1982. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/872726.806987>. páginas 8

IGNÁCIO, A. A. V.; FERREIRA, V. J. M. F. Mpi: uma ferramenta para implementação paralela. *Pesquisa Operacional*, scielo, v. 22, p. 105 – 116, 06 2002. ISSN 0101-7438. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0101-74382002000100007&nrm=iso>. páginas 6

KENNEDY, J.; EBERHART, R. C. *Swarm Intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN 1-55860-595-9. páginas 7

KIRK, D. B.; WEN-MEI, W. H. *Programming massively parallel processors: a hands-on approach*. [S.l.]: Morgan kaufmann, 2016. páginas 2

LIN, J.; LUCAS, T. A. A particle swarm optimization model of emergency airplane evacuations with emotion. *NHM*, v. 10, n. 3, p. 631–646, 2015. páginas 6

MOORE, G. E. *Cramming more components onto integrated circuits, Electronics Magazine*. 1965. páginas 2

NIEVERGELT, J. et al. *All the needles in a haystack: Can exhaustive search overcome combinatorial chaos?* Berlin, Heidelberg: Springer Berlin Heidelberg, 1995. 254–274 p. ISBN 978-3-540-49435-5. Disponível em: <https://doi.org/10.1007/BFb0015248>. páginas 6

NVIDIA. *CUDA C Programming Guide*. 2020. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Accessado em: 2020-02-28. páginas 3

OPENACC. *What is OpenACC?* 2019. [Online; acesso 20 Fevereiro 2020]. Disponível em: <https://www.openacc.org/>. páginas 5, 6

OPENMP. *The OpenMP API specification for parallel programming*. 2020. [Online; accessed at January, 15 2020]. Disponível em: <https://www.openmp.org/>. páginas 3, 4

RAUBER, T. W. Redes neurais artificiais. *Universidade Federal do Espírito Santo*, 2005. páginas 8

SERAPIAO, A. Fundamentos de Otimização por Inteligência de enxames: Uma Visão Geral. *Controle y Automacao*, v. 20, p. 271–304, 07 2009. páginas 7, 10

SILVA, M. et al. Mixing layer stability analysis with strong temperature gradients. In: *17th Brazilian Congress of Thermal Sciences and Engineering (ENCIT 2018)*. [S.l.: s.n.], 2017. páginas 8

SUTTER, H.; LARUS, J. Software and the Concurrency Revolution. *Queue*, ACM, New York, NY, USA, v. 3, n. 7, p. 54–62, set. 2005. ISSN 1542-7730. Disponível em: <http://doi.acm.org/10.1145/1095408.1095421>. páginas 2

TAVARES, Y.; NEDJAH, N.; MOURELLE, L. de M. Utilização de otimização por enxame de partículas e algoritmos genéticos em rastreamento de padrões. In: *12 Congresso Brasileiro de Inteligência Computacional*. Rio de Janeiro, Brasil: Diretoria de Sistemas de Armas da Marinha, 2015. páginas 10

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