Capítulo 3

Programação Paralela e Vetorial em Memória Compartilhada e Distribuída

Matheus S. Serpa – msserpa@inf.ufrgs.br[1]
Claudio Schepke – claudioschepke@unipampa.edu.br[2]
João Vicente Ferreira Lima – jvlima@inf.ufsm.br[3]

Resumo

No passado, o aumento de desempenho das aplicações ocorria de forma transparente aos programadores devido ao aumento do paralelismo a nível de instruções e aumento de frequência dos processadores. Entretanto, isto já não se sustenta mais há alguns anos. Atualmente para se ganhar desempenho nas arquiteturas modernas, é necessário conhecimentos sobre programação paralela, distribuída e vetorial. Todos estes paradigmas são tratados de alguma forma em muitos cursos de computação, mas geralmente não aprofundados. Neste contexto, este minicurso objetiva propiciar um maior entendimento sobre os paradigmas de programação paralela, distribuída e vetorial, de forma que os participantes aprendam a otimizar adequadamente suas aplicações para arquiteturas atuais. Desta forma, os estudantes terão a oportunidade de aprender e praticar conceitos de programação paralela em aplicações de alto desempenho.

[1]Matheus S. Serpa é doutorando no PPGC da Universidade Federal do Rio Grande do Sul (UFRGS). Possui mestrado em Computação pela UFRGS, sendo este feito na modalidade sanduíche na Université de Neuchâtel, Suiça (2017). É bacharel em Ciência da Computação pela Universidade Federal do Pampa (UNIPAMPA), tendo recebido o Prêmio Aluno Destaque da SBC (2016). Tem experiência na área de Ciência da Computação, com ênfase em Arquitetura de Computadores e Projeto de Algoritmos Paralelos.

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

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

3.1. Introdução

A introdução de circuitos integrados, *pipelines*, aumento da frequência de operação, execução fora de ordem e previsão de desvios constituem parte importante das tecnologias introduzidas até o final do século XX. Recentemente, tem crescido a preocupação com o consumo energético, com o objetivo de se atingir a computação em nível *exascale* de forma sustentável. Entretanto, as tecnologias até então desenvolvidas não possibilitam atingir tal objetivo, devido ao alto custo energético de se aumentar a frequência e estágios de *pipeline*, assim como a chegada nos limites de exploração do paralelismo a nível de instrução [Borkar and Chien 2011, Coteus et al. 2011].

A fim de se solucionar tais problemas, arquiteturas paralelas e sistemas distribuídos podem ser utilizadas. A principal característica de arquiteturas paralelas é a presença de vários núcleos de processamento operando concorrentemente, de forma que a aplicação deve ser programada separando-a em diversas tarefas que se comunicam entre si. Em relação a sistemas distribuídos, vários nós compostos por arquiteturas paralelas são distribuídos e interconectados por uma conexão de rede, sendo que, os nós se comunicam através da troca de mensagens.

Além disso, nos últimos anos tem-se visto um grande número de arquiteturas e sistemas computacionais à disposição do mercado, com uma considerável gama de recursos buscando satisfazer as necessidades dos desenvolvedores de *software*. Esse comportamento pode ser visto quando observamos a classificação das 500 máquinas mais rápidas do mundo [J. Dongarra and Strohmaier 2018], as quais são compostas por inúmeras unidades de processamento (processadores e aceleradores) interligados por barramentos e redes especiais destinadas a alto desempenho.

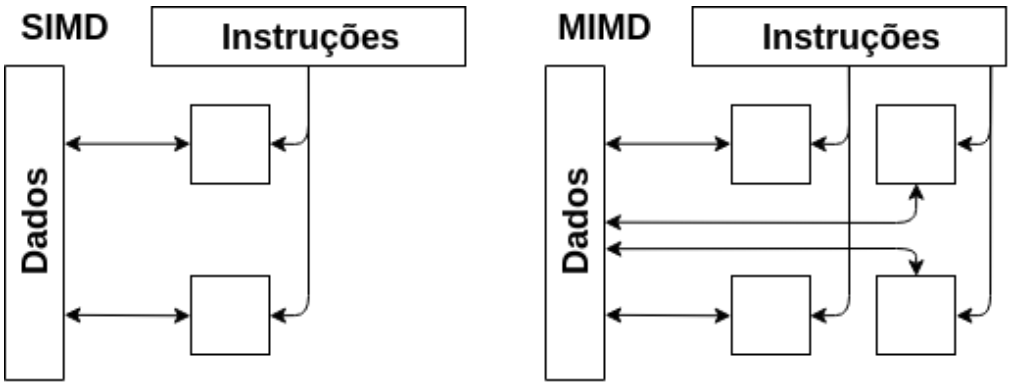
Em paralelo com o desenvolvimento de novas arquiteturas e sistemas computacionais, há uma necessidade de se oferecer recursos de programação que possibilitem o uso desses sistemas. Diante disso, este capítulo apresenta as interfaces de programação paralela OpenMP e MPI, mostrando como é possível desenvolver aplicações eficientes para arquiteturas e sistemas atuais.

O restante deste capítulo está organizado da seguinte forma. A Seção 3.2 discute conceitos sobre arquiteturas paralelas e sistemas distribuídos. A Seção 3.3 apresenta conceitos sobre modelagem e desenvolvimento de aplicações paralelas. A Seção 3.4 descreve conceitos de programação paralela e vetorial em OpenMP. A Seção 3.5 apresenta conceitos de programação distribuída em MPI. A Seção 3.6 mostra um estudo de caso sobre o desempenho de uma aplicação de geofísica utilizando os conceitos apresentado no capítulo e, finalmente, a Seção 3.7 apresenta as conclusões.

3.2. Arquiteturas Paralelas e Sistemas Distribuídas

A computação de alto desempenho tem sido responsável por uma revolução científica. A evolução das arquiteturas de computadores melhorou o poder computacional, aumentando a gama de problemas e a qualidade das soluções que poderiam ser resolvidas no tempo requerido como, por exemplo, a previsão do tempo. Entretanto, devido a limitações de consumo de energia, dissipação de calor, dimensão do processador e uma melhor distribuição das *threads* para processamento, a indústria mudou seu foco para arquiteturas paralelas e

Figura 3.1. Diagramas das classes SIMD (esquerda) e MIMD (direita).



sistemas distribuídas.

A principal característica dessas arquiteturas é a presença de vários núcleos de processamento operando simultaneamente. No entanto, o desenvolvimento de *software* foi afetado por essa mudança de paradigma e diversas aplicações sofreram reengenharias para tornar possível o aproveitamento dos recursos através da execução paralela. Essa seção apresenta conceitos sobre arquiteturas paralelas e sistemas distribuídos.

3.2.1. Classificação de Arquiteturas Paralelas

A classificação de Flynn [Flynn and Rudd 1996] baseia-se no fato de um computador executar uma sequência de instruções sobre uma sequência de dados. As instruções e os dados são separados em um ou vários fluxos de instruções (*instruction stream*) e um ou vários fluxos de dados (*data stream*). Essa classificação possui quatro classes mas vamos nos concentrar apenas nas duas classes que representam as arquiteturas paralelas: *Single Instruction Multiple Data* (SIMD) e *Multiple Instruction Multiple Data* (MIMD). A Figura 3.1 apresenta os diagramas das classes exemplificadas a seguir.

Em uma arquitetura SIMD, uma única instrução é executada ao mesmo tempo sobre múltiplos dados. Esse processamento é controlado por uma única unidade de controle que é alimentada por um único fluxo de instruções. Cada instrução é enviada para todos processadores que executam as instruções em paralelo de forma síncrona sobre diferentes fluxos de dados. Essa arquitetura é encontrada nas unidades MMX/SSE de processadores *multi-core* e nas *Graphics Processing Units* (GPUs).

Em arquiteturas MIMD, cada unidade de controle recebe um fluxo de instruções próprio e repassa-o para seu respectivo processador. Dessa forma, cada processador executa suas instruções em seus dados de forma assíncrona. O princípio dessa classe é bastante genérico, pois se um computador de um grupo de máquinas for analisado separadamente, este pode ser considerado uma máquina MIMD. Nessa classe encontram-se as arquiteturas paralelas *multi-core*

Arquiteturas paralelas e sistemas distribuídos atuais combinam ambas arquiteturas SIMD e MIMD. Por exemplo, um processador *multi-core* possui vários *cores* cada um trabalhando sobre um conjunto de instruções e um conjunto de dados, ou seja, MIMD. Em cada *core* do mesmo processador existe uma unidade especial de ponto flutuante que explora SIMD. Sistemas distribuídos são compostos por várias arquiteturas paralelas,

logo, combinando SIMD e MIMD.

3.2.2. Sistemas de Processamento Distribuído

Um sistema de processamento distribuído pode ser visto como um conjunto de computadores independentes interconectados através de uma rede. Cada computador é formado por uma unidade de processamento e memória próprios, sendo que a memória não pode ser acessada diretamente por outro computador. Consequentemente, a memória é distribuída entre os computadores e cada computador tem seu próprio espaço de endereçamento, que lhe permite acessar somente a sua própria memória. É através de uma rede de interconexão que é possível para os processadores enviar mensagens para outros processos. Estas mensagens podem incluir dados que outros processadores podem requerer para o seu correto processamento.

Um *cluster*, é um exemplo de sistema distribuído, o qual pode ser definido como um conjunto de máquinas ou nós, que cooperam entre si na execução de aplicações utilizando a troca de mensagens pela rede. Esta é uma forma simples de se obter o compartilhamento de recursos para prover um maior desempenho. Geralmente o acesso ao ambiente é restrito e dedicado. Além disso, a arquitetura é escalonável e flexível. Isso significa que máquinas podem ser adicionadas ou removidas, sem a interferência no restante do conjunto.

3.3. Modelagem e Desenvolvimento de Aplicações Paralelas

A programação paralela possibilita utilizar ao máximo os recursos de *hardware*. Com isso, muitos problemas antes impossíveis de serem solucionados podem ser executados sem muito esforço. A demanda de desempenho necessária para a realização de uma tarefa está relacionada com a quantidade de dados ou variáveis envolvidas durante o processamento e quais as operações pelas quais estes terão de passar até o resultado final. Quanto mais eficiente for a implementação de um algoritmo, menor a demanda por desempenho.

Segundo Foster [Foster 1995], a modelagem de um problema de forma paralela passa por quatro fases: o particionamento, a comunicação, o agrupamento e o escalonamento.

1) Particionamento - Primeiramente, os dados são divididos de maneira que cada tarefa possa ser executada independentemente das demais. Com isso obtém-se a menor granularidade possível para cada tarefa.

2) Comunicação - Em um segundo momento, devido ao fato dos dados normalmente estarem inter-relacionados, é necessário que haja a troca de informações entre os processos. Nessa fase é definida a forma de comunicação paralela adotada, caso seja utilizado uma arquitetura multiprocessada.

3) Agrupamento - Em seguida, em uma terceira fase, as operações ou dados são agrupados a fim de realizar um melhor uso dos processadores. O objetivo dessa fase é aumentar a granularidade das operações realizadas por um único processador. Assim, operações que envolvam um conjunto de dados vizinho são executadas em um mesmo processador, diminuindo a interdependência entre os dados.

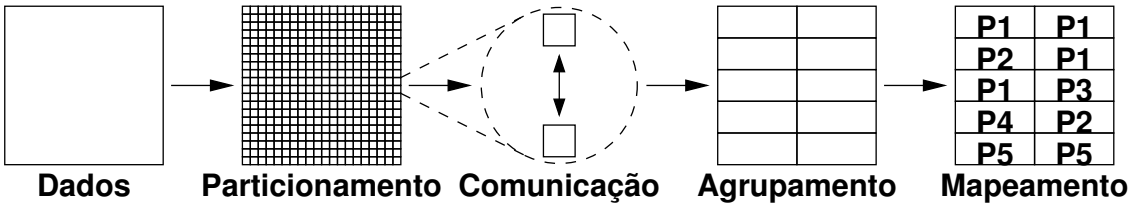


Figura 3.2. Principais etapas na paralelização de um algoritmo.

4) Escalonamento - Por fim, na quarta etapa, ocorre o mapeamento, que é a fase que define como serão distribuídas as tarefas entre os processadores. Essa distribuição busca casar a granularidade das tarefas com a capacidade de processamento dos processadores e a dependência entre os processos que se encontram em processadores distintos.

A Figura 3.2 ilustra cada uma das etapas descritas anteriormente. Inicialmente um conjunto de dados é particionado. Posteriormente são destacadas as interações entre dois pontos vizinhos de granularidade fina. Após o agrupamento entre alguns pontos é feito um mapeamento, que distribui as tarefas entre 5 processadores ($P1, ..., P5$).

3.4. Programação Paralela e Vetorial em OpenMP

OpenMP (*Open Multi-Processing*) [Chapman et al. 2007] é uma API (*Application Programming Interface*) de programação paralela portável para arquiteturas de memória compartilhada. OpenMP surgiu da dificuldade no desenvolvimento de programas paralelos em arquiteturas de memória compartilhada, além da ausência de APIs padronizadas para tais arquiteturas. A interface proporciona diretivas que possibilitam expressar paralelismo de dados, em trechos de código e laço, e paralelismo de tarefas, introduzido em sua versão 3.0 [Ayguadé et al. 2009]. Sua API é constituída de diretivas de compilação, métodos de biblioteca e variáveis de ambiente. Em sua versão 4.0, OpenMP inclui suporte para dependências de dados em tarefas e suporte a aceleradores [OpenMP 2018].

3.4.1. Programando com OpenMP

A API OpenMP é composta basicamente por diretivas de compilação e métodos da biblioteca. As diretivas são anotações no código e os métodos OpenMP dependem da compilação com a biblioteca. As diretivas de compilação, *pragmas* em linguagem C/C++, do OpenMP começam com

```
#pragma omp
```

e são seguidos por construções e cláusulas que se aplicam a um bloco estruturado. As construções descrevem seções paralelas, dividem dados ou tarefas entre *threads* e controlam sincronização. Por sua vez, as cláusulas modificam ou especificam aspectos das construções.

O primeiro exemplo é um *Olá Mundo*. A Figura 3.3 ilustra o primeiro exemplo em OpenMP. A construção `parallel` faz com que o bloco estruturado especificado entre as linhas 8 e 14 seja executado múltiplas vezes. A compilação de tal programa com o `gcc`

necessita da opção `-fopenmp`:

```
$ gcc -fopenmp -o hello hello.c
```

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main(){
5     int myid, nthreads;
6
7     #pragma omp parallel private(myid)
8     {
9        myid = omp_get_thread_num();
10       nthreads = omp_get_num_threads();
11
12       printf("Hello world. I am thread %d of %d\n",
13                  myid, nthreads);
14    }
15    return 0;
16 }
```

Figura 3.3. Exemplo de um *Hello world* em OpenMP.

A execução ocorre da mesma forma que qualquer outro programa em um terminal. Se nenhum argumento é especificado, o programa utilizará todos os processadores disponíveis. Em nosso exemplo, assumindo que a máquina possui quatro processadores, a execução será:

```
$ ./hello
```

Na linha de comando, pode-se alterar o número de *threads* com a variável de ambiente `OMP_NUM_THREADS`:

```
$ OMP_NUM_THREADS=4 ./hello
```

3.4.2. Modelo de Execução

O paralelismo em OpenMP é chamado *fork/join*, ou seja, o programa inicia com uma *thread*, a *thread* inicial. Ao encontrar uma construção `parallel`, o programa cria ou bifurca (*fork*) um grupo de *threads* que executam um bloco estruturado de código. Essas *threads* são então unidas (*join*) ao final do bloco.

A Figura 3.4 mostra um exemplo de execução OpenMP com três regiões paralelas. A *thread* inicial, que encontra a construção `parallel`, é chamada de *thread* **master**. Ela é responsável por criar um grupo de *threads* que executará o bloco paralelo. As regiões sequenciais são aquelas fora da construção `parallel` e são executadas pela *thread* master. Por outro lado, as regiões paralelas executam nos processadores disponíveis e podem

variar o número de *threads* no decorrer da execução. Nesse exemplo (Figura 3.4) existem três regiões paralelas com quatro, seis e três *threads* respectivamente.

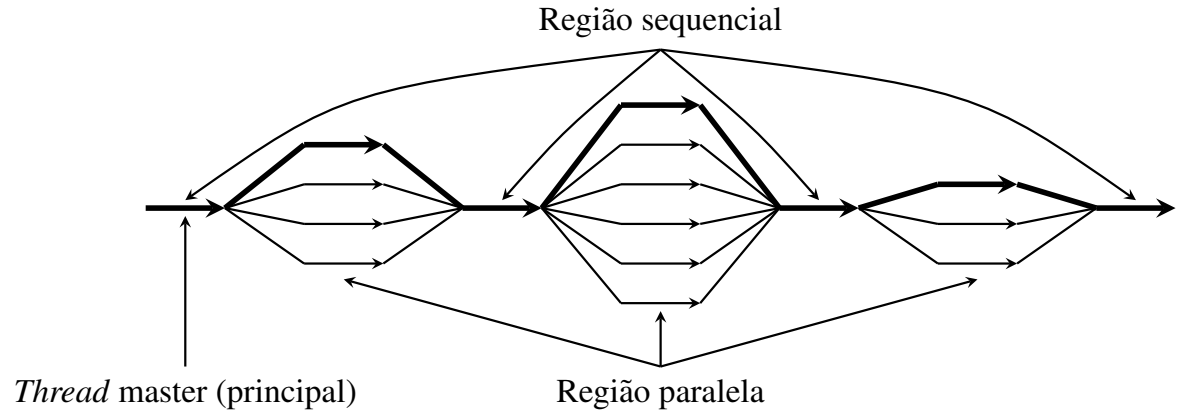


Figura 3.4. Modelo de execução *fork/join* do OpenMP.

A execução dentro de um bloco `parallel` é SPMD (*single program multiple data*), ou seja, as *threads* do grupo executam o mesmo código. A execução em SPMD é amplamente utilizada em alto desempenho e principalmente conhecida por seu uso em programas MPI. Cada *thread* possui um identificador obtido pela função `omp_get_thread_num()`.

3.4.3. Laços Paralelos

Os laços paralelos são uma das principais construções do OpenMP devido a sua popularidade e ocorrência em aplicações paralelas. O laço paralelo distribui as iterações entre as *threads* disponíveis, o que justifica a construção ser chamada **worksharing**.

A Figura 3.5 mostra um exemplo de laço paralelo em OpenMP onde a soma das posições do vetor `v` será dividido entre as *threads* da região paralela.

As construções `parallel` e `for` podem ser combinadas em uma única linha como em

```
#pragma omp parallel for
```

sendo equivalente ao exemplo anterior.

3.4.4. Exclusão Mútua

A sincronização é necessária em programação paralela a fim de coordenar a execução e evitar condições de corrida. Em OpenMP pode-se encontrar diversas formas de sincronização desde controle de ordem de execução até regiões críticas.

Para exclusão mútua, usa-se duas diretivas: `critical` e `atomic`. A diretiva `critical` especifica que o bloco de código é uma região crítica e apenas uma *thread* por vez executa a região. A diretiva `atomic` fornece exclusão mútua para uma região de memória, ou seja, quando a atualização de uma variável precisa ser protegida contra condições de corrida. A Figura 3.5 mostra um exemplo de uso do `atomic` onde um valor

```
1 long long int sum(int *v, long long int N){
2   long long int i, sum_local, sum = 0;
3
4   #pragma omp parallel private(i, sum_local)
5   {
6     sum_local = 0;
7
8     #pragma omp for
9     for(i = 0; i < N; i++)
10      sum_local += v[i];
11
12    #pragma omp atomic
13    sum += sum_local;
14  }
15  return sum;
16 }
```

Figura 3.5. Laço paralelo com OpenMP.

é acumulado. A acumulação é atômica e possivelmente concorrente.

3.4.5. Redução

Em algumas situações as aplicações paralelas precisam reduzir ou acumular um certo valor de forma concorrente dentro de um laço. Tal funcionalidade é suportada em OpenMP com a cláusula `reduction`.

Uma redução em OpenMP possui a sintaxe `reduction (op : list)` onde `op` é a operação e `list` é a lista de variáveis a serem acumuladas. Dentro de um bloco cada variável de `list` gera uma cópia local (por *thread*) e é inicializada de acordo com a operação (ex.: `0` para a operação +). Atualizações por iteração acontecem localmente em cada *thread*, e ao fim do bloco (*join*) as cópias locais são reduzidas em um valor único e combinadas com o valor original. Note que as variáveis em `list` devem ser compartilhadas (`shared`) dentro da região paralela.

Será utilizado como exemplo o cálculo da soma de todos elementos do vetor `v` da Figura 3.6, o qual anteriormente calculamos utilizando somas locais. O exemplo difere do anterior com a adição da construção `parallel for` com a operação de redução + para acumular os resultados na variável `sum`. As operações suportadas pela redução são +, −, ⋆, `min`, `max`, `&`, `|`, `^`, `&&` e `||`.

3.4.6. Vetorização

O paralelismo com execução vetorial se dá de forma diferente do explicado anteriormente. Enquanto na execução normal cada instrução opera em apenas um dado, na instrução vetorial a mesma operação é executada em vários dados de forma independente [Satish et al. 2012]. Considere o seguinte laço, na Figura 3.7 que soma dois vetores e armazena o resultado em um terceiro.

```
1 long long int sum(int *v, long long int N){
2    long long int i, sum = 0;
3
4    #pragma omp parallel for private(i) reduction(+ :  sum)
5    for(i = 0; i < N; i++)
6      sum += v[i];
7
8    return sum;
9 }
```

Figura 3.6. Exemplo do cálculo de soma de vetor com redução em OpenMP.

```
1 void sum(int *a, int *b, int *c, long long int N){
2    long long int i;
3
4    #pragma omp simd
5    for(i = 0; i < N; i++)
6      a[i] = b[i] + c[i]
7 }
```

Figura 3.7. Exemplo do cálculo de soma de dois vetores com SIMD.

Como pode-se perceber, as iterações do laço são independentes. Supondo que há instruções para ler e escrever 8 operandos na memória, e somar 8 operandos, pode-se visualizar o mesmo laço sendo operado vetorialmente 8 operando por unidade de tempo utilizando

```
#pragma omp simd
```

Em cada iteração do laço, carregam-se 8 operandos a partir da posição i dos vetores b e c, soma-se cada par $(b[i], c[i])$ de forma independente, e depois o bloco de 8 operandos é escrito no vetor a a partir da posição i. O número de operandos por unidade de tempo depende tanto do tamanho do dado quanto do tamanho da unidade vetorial do processador alvo.

As instruções vetoriais já estão presentes há muitos anos em processadores x86. A cada nova geração, aumenta-se a quantidade de dados processados por instrução, bem como o número de instruções vetoriais disponíveis. É importante ressaltar que, para maior eficiência, **os endereços acessados no laço em iterações sucessivas devem ser consecutivos**.

3.4.7. Cláusulas de Dados

O OpenMP é uma API de programação paralela para memória compartilhada, então grande parte das variáveis em memória são compartilhadas. Porém, nem todas as varáveis podem ser compartilhadas. Por exemplo, variáveis da pilha de funções e automáticas (de blocos de código) dentro de uma região paralela são privadas.

O OpenMP permite especificar e modificar o modo de acesso dentro de construções por meio de cláusulas. As cláusulas para dados em OpenMP são:

shared - compartilhada entre todas as *threads*.

private - cria uma nova cópia local para cada *thread*.

firstprivate - cria uma nova cópia local com o valor inicial da variável compartilhada.

lastprivate - atualiza o valor da variável compartilhada com o valor da última iteração sequencial.

reduction - descrita anteriormente, ela protege o conteúdo da variável por operação atômica.

3.4.8. Métodos de Biblioteca

Os métodos da biblioteca OpenMP atuam para modificar e monitorar *threads*, processos e a região paralela do programa. Elas são linkadas como funções externas em C. A seguir são listadas as principais funções:

```
void omp_set_num_threads(int N)
```

modifica o número de *threads* da próxima região paralela.

```
int omp_get_num_threads()
```

retorna o número de *threads* da região paralela atual.

```
int omp_get_thread_num()
```

retorna o identificador da *thread* atual.

3.5. Programação Distribuída em MPI

O processo de desenvolvimento de aplicações tem sido simplificado pela existência de mecanismos de programação paralela padronizados e com uma vasta gama de recursos. Um dos recursos amplamente utilizado para tal finalidade é a biblioteca de comunicação (via troca de mensagens) *Message Passing Interface* (MPI) [Gropp et al. 1996].

MPI possui um grande número de funções que podem ser utilizadas, tanto em implementações paralelas, como em implementações distribuídas. Entre os recursos encontram-se mecanismos de comunicação cartesiana, que possibilitam o endereçamento de mensagens aos processos segundo as posições atribuídas aos mesmos. Tais recursos são imprescindíveis para a obtenção de uma boa eficiência paralela, tendo sido recorrente a sua utilização.

3.5.1. Troca de Mensagens

No modelo de troca de mensagens cada processador tem sua memória. A troca de informações ocorre através da comunicação entre os processadores usando uma rede de alta velocidade. Esse modelo introduz um novo problema: como distribuir a tarefa computacional em múltiplas tarefas para múltiplos processadores com diferentes espaços de memória (cada um acessa a sua) e organizar os resultados em uma só solução.

A principal vantagem do modelo de troca de mensagens é a escalabilidade, pois não há limite de processos que podem ser criados, nem o número de processadores que podem ser utilizados. Também há possibilidade de se usar máquinas heterogêneas. No modelo de troca de mensagens, as tarefas geralmente fragmentadas são executadas em processadores distintos e o resultado final normalmente é agrupado em um processo ou em todos os processos.

3.5.2. Message-Passing Interface

Message Passing Interface (MPI) é um padrão para comunicação de dados na computação paralela [Gropp et al. 1996]. Isso garante a portabilidade dos programas paralelos. O principal objetivo de MPI é disponibilizar uma interface que seja largamente utilizada no desenvolvimento de programas baseados em troca de mensagens, onde um conjunto de processos possui acesso a memória local, e para a comunicação entre processos é utilizado o envio e recebimento de mensagens, além da cooperação para transferência de dados.

A especificação de MPI pode ser implementada para diversos tipos de máquinas paralelas. Em geral, é possível escrever programas nas linguagens FORTRAN, C ou C++. MPI define apenas o modelo de troca de mensagens tais como nomes de funções, seqüências de chamadas e resultados de subrotinas, sem se preocupar com a implementação em si. Por isso, existem diversas implementações do padrão MPI, cada qual com suas características e otimizações de código [Open-MPI 2019].

No padrão MPI, uma aplicação é constituída por um ou mais processos que podem ser executados em máquinas distintas, os quais se comunicam através de funções de envio e recebimento de mensagens via interface de rede. Assim, as implementações do padrão oferecem uma infraestrutura para a computação paralela na qual é possível a troca de informações. As rotinas de MPI permitem executar um mesmo fluxo de execução em unidades de processamento distintas (SPMD) ou dividir um fluxo de execução em vários trechos para serem executados em unidades de processamento distintas (MPMD).

MPI disponibiliza diferentes formas de comunicação. Os mecanismos de comunicação mais simples que podem ser utilizados são a comunicação ponto a ponto, onde ocorrem operações de troca de mensagens de um determinado processo com outro. Estruturas mais refinadas de comunicação são obtidas usando um grupo de processos que invocam operações coletivas (*collective*) de comunicação para a execução de operações globais. Além disso, MPI é capaz de suportar comunicação assíncrona e programação modular, através de mecanismos de comunicadores (*communicator*). Os comunicadores permitem ao usuário MPI definir módulos que encapsulem estruturas de comunicação interna (*group communications*).

3.5.3. Programando com MPI

O funcionamento básico de um programa com MPI consiste em lançar um conjunto de processos, os quais atuam sobre um determinado código fonte. Cabe ao programador diferenciar através de um identificador de processo quem é o responsável pela execução de cada parte do código. É através desse mesmo indicador que os demais processos podem acessar esse processo específico. Desta forma, é possível realizar a troca de mensagens, garantindo a comunicação entre os processo.

O primeiro exemplo será um *Hello world* concorrente. A Figura 3.8 ilustra o primeiro exemplo em MPI. A função `MPI_Init` e `MPI_Finalize` definem os limites da execução paralela do **bloco estruturado** especificado na linha 12.

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char **argv){
5    int rank, size;
6
7    MPI_Init(&argc, &argv);
8
9    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10   MPI_Comm_size(MPI_COMM_WORLD, &size);
11
12   printf("Hello world. I am process %d of %d\n", rank, size);
13
14   MPI_Finalize();
15   return 0;
16 }
```

Figura 3.8. Exemplo de um *Hello world* em MPI.

A compilação de tal programa pode ser feita usando o compilador `mpicc`:

```
$ mpicc -o hello hello.c
```

A execução sequencial ocorre da mesma forma que qualquer outro programa em um terminal:

```
$ ./hello
```

Para poder executar paralelamente o programa é preciso usar o comando `mpirun`. Por exemplo, para lançar quatro processos, a execução será através de

```
$ mpirun -np 4 ./hello
```

onde a opção `-np` serve para indicar o número de processos que serão lançados.

A execução anterior é restrita ao computador onde o comando foi executado. Para

a execução em máquinas distintas é necessário passar o nome de um arquivo como argumento, cujo conteúdo identifica os diferentes computadores (nome ou endereço de IP), usando `-hostfile` ou `-machinefile`, como por exemplo:

```
$ mpirun -np 4 -machinefile computadores.txt ./hello
```

A Figura 3.8 também apresenta o uso das funções `MPI_Comm_rank()` e `MPI_Comm_size()`, que permitem descobrir, respectivamente, em tempo de execução, qual o identificador específico de cada processo (*rank*) e o número total de processos em execução (*size*). O número total de processos é igual ao valor passado como argumento no lançamento dos processos (`-np`). O idenficador de cada processo é um número sequencial começando em 0 até $size - 1$. Ambas as funções recebem como primeiro argumento uma rede de comunicação entre os processos. No caso da constante `MPI_COMM_WORLD` a rede de comunicação é formada por todos os processos/computadores envolvidos durante a chamada de `mpirun`.

Através do uso das funções `MPI_Comm_rank` e `MPI_Comm_size` é possível fazer a diferenciação do código que cada processo executa. Na Figura 3.9 é possível ver de forma prática o uso dessas funções em um código de envio e recebimento de mensagens. Neste exemplo, o processo identificado com `rank == MASTER (0)` é responsável por enviar uma mensagem a cada um dos demais processos paralelos envolvidos através da função `MPI_Send`. Os processos destinatários, para poderem receber a mensagem, precisam invocar a função `MPI_Recv`. Tais funções permitem uma comunicação síncrona entre os processos. Para comunicações assíncronas é possível usar as funções `MPI_Isend` e `MPI_Irecv`. Independente dos casos, uma série de argumentos são passados para as funções.

As funções de comunicação síncronas recebem como parâmetros um endereço de memória (tipicamente um vetor), a quantidade de dados que será copiado, de acordo com o tipo de dados que se quer comunicar, para quem os dados são enviados (`MPI_Send`) ou recebidos (`MPI_Recv`), um identificador da mensagem e qual a rede de intercomunicação ao qual os processos comunicantes estão relacionados. No caso de `MPI_Recv` existe ainda um parâmetro de *status* da mensagem.

3.5.4. Operações Coletivas

Através do uso das funções básicas vistas nos exemplos anteriores é possível desenvolver uma quantidade expressiva de códigos. Além disso, o uso das chamadas de comunicação assíncronas, mais o uso de funções coletivas que permitem a difusão ou captação de múltiplas mensagens (`MPI_Bcast()`, `MPI_Gather()` e `MPI_Scatter()`) são o suficientes para uma grande quantidade de desenvolvedores de programas paralelos. A Figura 3.10 demonstra o uso de operações coletivas para envio de uma mensagem para todos processos.

Os recursos encontrados em MPI são muito importantes pois garantem implementações paralelas com mecanismos de comunicação eficientes e uma maior independência entre as execuções dos processos. Além desses, MPI possui ainda mecanismos para a criação de estruturas cartesianas, bem como as funções necessárias para o mapeamento e acesso aos processos. Analisando-se esses recursos, além dos encontrados

```
1  ...
2
3  if(rank == MASTER){
4    strcpy(buffer, "ERAD 2019 - SETREM");
5    for(i = 1; i < size; i++){
6      printf("Process %d: Message send to process %d: ...
           %s\n\n", rank, i, buffer);
7      MPI_Send(buffer, strlen(buffer), MPI_CHAR, i, TAG, ...
           MPI_COMM_WORLD);
8    }
9  }else{
10   memset(buffer, '\0', BUFSIZE);
11   MPI_Recv(buffer, BUFSIZE, MPI_CHAR, MASTER, TAG, ...
       MPI_COMM_WORLD, NULL);
12   printf("Process %d: Message received from process %d: ...
       %s\n\n", rank, MASTER, buffer);
13 }
14
15 ...
```

Figura 3.9. Exemplo de envio e recebimento de mensagens em MPI.

anteriormente, MPI oferece as condições ideais para a programação paralela baseada em particionamento de dados em blocos.

```
1  ...
2
3  if(rank == 0){
4    strcpy(buffer, "ERAD 2019 - SETREM");
5    length = strlen(buffer);
6  }
7
8  MPI_Bcast(&length, 1, MPI_INT, MASTER, MPI_COMM_WORLD);
9  MPI_Bcast(buffer, length + 1, MPI_CHAR, MASTER, ...
     MPI_COMM_WORLD);
10
11 printf("Process %d: Message received from process %d: ...
     %s\n\n", rank, MASTER, buffer);
12
13 ...
```

Figura 3.10. Exemplo do uso de operações coletivas em MPI.

3.6. Estudo de Caso: Aplicação Geofísica

Nesta seção avaliamos o desempenho de uma aplicação de geofísica em um sistema distribuído com arquiteturas paralelas *multi-core*. Primeiro apresentamos a aplicação geofísica, após o ambiente de execução e por fim a discussão dos resultados.

3.6.1. Modelagem Fletcher

A Modelagem Fletcher [Fletcher et al. 2009] simula a propagação de ondas em meio ansinotrópico em um domínio ao longo do tempo. As ondas são emitidas por uma fonte, tipicamente no interior ou na borda do domínio, o qual é um paralelepípedo tridimensional. O código[4] foi escrito em linguagem *C* e a discretização utilizando diferenças finitas.

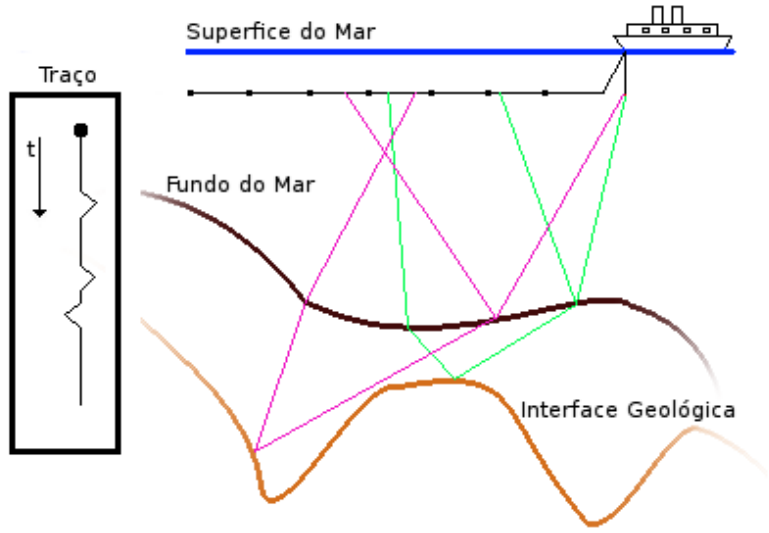


Figura 3.11. Coleta de dados em levantamento sísmico marítimo.

A modelagem simula a coleta de dados em um levantamento sísmico, como na Figura 3.11. De tempos em tempos, equipamentos acoplados ao navio emitem ondas que refletem e refratam em mudanças de meio no subsolo. Eventualmente essas ondas voltam à superfície do mar, sendo coletadas por microfones específicos acoplados a cabos rebocados pelo navio. O conjunto de sinais recebidos por cada fone ao longo do tempo constitui um traço sísmico. Para cada emissão de ondas, gravam-se os traços sísmicos de todos os fones do cabo. O navio continua trafegando e emitindo sinais ao longo do tempo.

3.6.2. Ambiente de Execução

Utilizamos um ambiente distribuído com arquiteturas paralelas para avaliar o desempenho de uma aplicação real. O ambiente é o sistema *hype* do Grupo de Processamento Paralelo e Distribuído (GPPD) da Universidade Federal do Rio Grande do Sul (UFRGS).

O sistema possui 5 nós de computação, sendo que cada nó consiste de 2 processadores Intel Xeon E5-2650 v3 de 20 núcleos. O sistema possui no total 100 núcleos e 200 *threads (Hyper-Threading)*, além de 128 GB DDR4 em cada nó. Os resultados apresentados são a média de 10 experimentos.

[4]Este trabalho foi parcialmente financiado por recursos do projeto Petrobras 2016/00133-9.

3.6.3. Discussão

Quatro versões da Modelagem Fletcher foram escritas tendo como base o material apresentado nesse capítulo. A versão Sequencial implementada em C. A Vetorizada que através da diretiva *simd* tirou proveito das unidades vetoriais. Duas versões paralelas, uma em OpenMP e uma em MPI. A versão em OpenMP utilizou as diretivas *parallel* e *for* e, por fim a versão MPI que utilizou chamadas assíncronas de *MPI_ISEND* e *MPI_IRECV*.

A Tabela 3.1 apresenta os resultados de tempo de execução das diferentes versões utilizando como entrada um cubo de tamanho 1024. A versão vetorial executada em um *core* utilizando unidades vetoriais melhorou o desempenho da aplicação em 1,4× em relação a versão sequencial. A versão paralela em OpenMP foi executada com 40 *threads*, melhorando o desempenho da aplicação em 28,2×. Por fim, desenvolvemos uma versão distribuída em MPI, executando em 5 nós, totalizando 200 processos. Essa versão teve o melhor desempenho, de 39,9× em relação a versão sequencial.

Tabela 3.1. Desempenho da Modelagem Fletcher.

Versão	Tempo de Execução (s)	#
Sequencial	769,4	1 *thread*
Vetorizada	539,8	1 *thread*
OpenMP	27,3	40 *threads*
MPI	19,3	200 processos

3.7. Considerações Finais

O uso de arquiteturas paralelas e sistemas distribuídos é uma solução para incrementar a capacidade de execução, tornando possível a computação eficiente de aplicações com grande demanda de processamento. Para tanto, é preciso fazer uso de interfaces de programação paralela. Neste capítulo foram apresentados detalhes das interfaces *OpenMP* e *MPI*. Tais interfaces são amplamente utilizadas em aplicações de alto desempenho para ambientes de memória distribuída e compartilhada. Com isso, é possível criar processos e *threads* de forma prática. Desta forma, é possível a solução eficiente de problemas que possuem algum tipo de concorrência.

O conhecimento elementar e o uso das duas interfaces de programação apresentadas neste capítulo torna-se imprescindível para o desenvolvimento de aplicações paralelas. Desta forma, como mostramos no estudo de caso de uma aplicação geofísica, é possível maximizar o uso dos recursos computacionais disponíveis nas arquiteturas atuais, com isso, melhorando o desempenho de aplicações sintéticas e reais.

Os *slides*, exemplos e exercícios estão disponíveis em:

`https://gitlab.com/msserpa/prog-paralela-erad`.

Referências

[Ayguadé et al. 2009] Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. (2009). The Design of OpenMP Tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418.

[Borkar and Chien 2011] Borkar, S. and Chien, A. A. (2011). The future of microprocessors. *Communications of the ACM*, 54(5):67–77.

[Chapman et al. 2007] Chapman, B., Jost, G., and van der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, Cambridge, USA.

[Coteus et al. 2011] Coteus, P. W., Knickerbocker, J. U., Lam, C. H., and Vlasov, Y. A. (2011). Technologies for exascale systems. *IBM Journal of Research and Development*, 55(5):14–1.

[Fletcher et al. 2009] Fletcher, R. P., Du, X., and Fowler, P. J. (2009). Reverse time migration in tilted transversely isotropic (tti) media. *Geophysics*, 74(6):WCA179–WCA187.

[Flynn and Rudd 1996] Flynn, M. J. and Rudd, K. W. (1996). Parallel architectures. *ACM Computing Surveys (CSUR)*, 28(1):67–70.

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

[Gropp et al. 1996] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828.

[J. Dongarra and Strohmaier 2018] J. Dongarra, H. M. and Strohmaier, E. (2018). Top500 supercomputer: November 2018. `https://www.top500.org/lists/2018/11/`. [Online; accessed 26-February-2019].

[Open-MPI 2019] Open-MPI (2019). Open MPI: Open Source High Performance Computing. `http://www.open-mpi.org`. [Online; accessed 14-January-2019].

[OpenMP 2018] OpenMP (2018). OpenMP Application Programming Interface - Version 5.0. `http://www.openmp.org`. [Online; accessed 15-January-2019].

[Satish et al. 2012] Satish, N., Kim, C., Chhugani, J., Saito, H., Krishnaiyer, R., Smelyanskiy, M., Girkar, M., and Dubey, P. (2012). Can traditional programming bridge the ninja performance gap for parallel computing applications? In *ACM SIGARCH Computer Architecture News*, volume 40, pages 440–451. IEEE Computer Society.