3

Programação Paralela em Memória Compartilhada e Distribuída

Claudio Schepke – claudioschepke@unipampa.edu.br [1]
João V. F. Lima – jvlima.ufrgs@gmail.com[2]

Resumo:

Programação paralela é a divisão de uma determinada aplicação em partes, de maneira que essas partes possam ser executadas simultaneamente, por vários elementos de processamento. Os elementos de processamento devem cooperar entre si utilizando primitivas de comunicação e sincronização, diferente do paradigma de execução sequencial do fluxo de instruções. Para tanto, existem diversas interfaces de programação paralelas disponíveis, como bibliotecas, extensões de linguagem ou diretivas de compilação. Neste capítulo são apresentadas duas interfaces de programação paralelas existentes e que podem ser utilizadas para a geração de código para as arquiteturas paralelas atuais. Para tanto, são mostradas formas de utilização das interfaces OpenMP e MPI para o desenvolvimento de códigos paralelos. Desta forma, é possível aprender e praticar conceitos elementares de programação paralela.

[1]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). Claudio tem experiência na área de Ciência da Computação, com ênfase em Processamento Paralelo e Distribuído, atuando principalmente nos seguintes temas: processamento de alto desempenho, aplicações numéricas e programação paralela. Desde a graduação trabalha com aplicações paralelas, utilizando diferentes interfaces de programação paralela. Nos 2 últimos anos foi responsável pelo minicurso de Programação Paralela na ERAD/RS.

[2]João V. F. Lima possui graduação em Ciência da Computação pela Universidade Federal de Santa Maria (2007), mestrado em Computação pela Universidade Federal do Rio Grande do Sul (2009) e doutorado em co-tutela entre a Universidade Federal do Rio Grande do Sul e o MSTII da Universidade de Grenoble (2014). Atualmente é Professor Adjunto do Departamento de Linguagens e Sistemas de Computação da Universidade Federal de Santa Maria. João 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 e aplicações paralelas. Desde a graduação trabalha com interfaces de programação paralela. Nos 3 últimos anos foi desenvolvedor da ferramenta de programação paralela XKaapi.

3.1. Introdução

O uso de ambientes de Processamento de Alto Desempenho (PAD) tem sido recorrente para a execução de aplicações que exijam uma significativa capacidade de processamento de dados [Dongarra et al. 1991, Buyya 1999, Sims et al. 2000, Shameem and Roberts 2005]. São inúmeras as aplicações encontradas no cenário atual que fazem uso de PAD. Além disso, muitas outras estão em desenvolvimento, sob forma de pesquisa, especialmente na área da saúde, biologia, engenharia do petróleo, fluxo de gases e climatologia [Xavier et al. 2007, Panetta et al. 2007, Souto et al. 2007, Schepke et al. 2009, Osthoff et al. 2011, Schepke et al. 2013]. Tais aplicações, sob o ponto de vista computacional, são diferentes entre si em termos de implementação, execução, e necessidade de recursos computacionais e de software. A todas essas observações adiciona-se ainda a forma como os fluxos de execução de uma aplicação são processados, uma vez que estes dependem de uma série de fatores, tais como parâmetros de entrada e leitura de dados durante a execução do código, ou seja, dinamicamente.

No transcorrer dos últimos anos tem-se visto um grande número de sistemas computacionais à disposição no mercado, com uma considerável gama de recursos que satisfaça as necessidades dos desenvolvedores de software. Isto pode ser claramente visto quando se observa a composição das 500 máquinas com maior capacidade de processamento do mundo [TOP 500 2014], as quais são compostas por inúmeros unidades de processamento (processadores) interligadas, seja através de um barramento comum, seja através de redes especiais, as quais são destinadas a uma infinidade de aplicações. Geralmente, tais soluções estão baseadas no desenvolvimento de arquiteturas paralelas [Foster 1995, Wilkinson and Allen 1998, Andrews 2001, Dongarra et al. 2002, Foster and Kesselman 2003]. O uso de máquinas vetoriais, multiprocessadas e, atualmente, de sistemas *multi-core* tem sido algumas das alternativas. Tais tecnologias podem ainda ser combinadas através da formação de *clusters* e *grids*. Chega-se, dessa forma, a composição de plataformas com múltiplos níveis de paralelismo.

Ao mesmo tempo em que há um progresso no desenvolvimento de hardware, especialmente de arquiteturas paralelas, há uma necessidade de se oferecer recursos de programação compatíveis com os diferentes ambientes computacionais disponíveis. Diante disso, este capítulo apresenta algumas interfaces de programação eficientes para ambientes de memória compartilhada e distribuída, mostrando como é possível desenvolver aplicações usando as bibliotecas OpenMP e MPI. Para tanto, são abordados inicialmente aspectos relacionados a arquiteturas paralelas e técnicas de paralelização de algoritmos.

3.2. Arquiteturas Paralelas

Um dos motivos pelos quais a evolução dos sistemas computacionais tem conseguido manter seu ritmo de crescimento nos últimos anos deve-se à exploração e à concepção de arquiteturas paralelas. Arquiteturas paralelas são baseadas na utilização de múltiplas unidades de processamento. Atualmente estão sendo explorados diferentes níveis de paralelismo, que vão desde múltiplos núcleos de processamento das arquiteturas *multi-core* e multi-processadores, passando pelo uso de multicomputadores, através do uso de *clusters*, até a organização de um *grid* em escala mundial. Com isso, pode-se obter um desempenho muito mais elevado do que o estimado em arquiteturas mais sim-

ples. Devido à relevância do assunto, nesta seção são revistas as principais arquiteturas paralelas em uso no momento, relatando as suas características básicas.

3.2.1. Classificação de *Flynn*

Para melhor entender a concepção das diferentes arquiteturas paralelas pode-se utilizar uma técnica de classificação. Uma das maneiras de se realizar isso é utilizar a classificação de *Flynn*, a qual é amplamente adotada [Wilkinson and Allen 1998]. Esta classificação está baseada na relação que existe entre o fluxo de instruções e o fluxo de dados. Neste sentido quatro classes podem existir:

- *Single Instruction Single Data* (SISD) - Onde para cada instrução há um único dado sendo operado. Geralmente computadores pessoais possuem essa configuração.

- *Multiple Instruction Single Data* (MISD) - Neste caso, hipoteticamente haveria várias instruções sendo operadas sobre o mesmo dado.

- *Single Instruction Multiple Data* (SIMD) - São as máquinas vetoriais, ou seja, uma única instrução é executada sobre múltiplos dados de forma paralela.

- *Multiple Instruction Multiple Data* (MIMD) - Aqui múltiplas instruções são executadas sobre múltiplos dados. Tais execuções paralelas podem ocorrer através de multiprocessadores ou multicomputadores.

A seguir serão vistos detalhes de como as arquiteturas paralelas podem ser organizadas.

3.2.2. Multicomputadores

Uma arquitetura Multicomputador pode ser vista como um conjunto de computadores independentes interconectados através de uma rede [Andrews 2001]. Cada computador é formado por unidade 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.

Multicomputadores podem ser vistos como um sistema baseado em **troca de mensagens**. As mensagens trocadas entre os processos são ditadas pelo código do programa. A ideia, com isso, é dividir o problema em várias partes e, durante a execução, realizar a troca de mensagens entre os processos quando necessário.

Uma comparação entre arquiteturas multiprocessador e multicomputador pode ser vista na Figura 3.1. Nesta ilustração, os processos P de uma máquina multiprocessada acessam a memória (M) através de um barramento comum. Já em uma arquitetura multicomputador, cada processador tem sua memória local e só pode acessar os dados de outra máquina através de uma interconexão de rede.

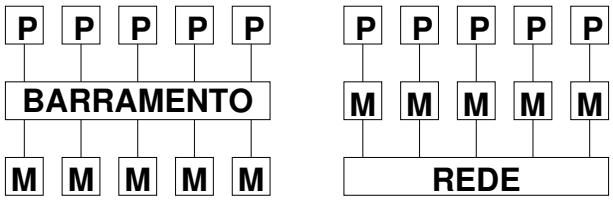


Figura 3.1: Esquema das arquiteturas multiprocessador e multicomputador.

3.2.3. Multiprocessadas

Uma forma simples de formar uma máquina paralela é agregar processadores a um barramento comum. Desta forma, todos os processadores poderiam ter acesso à memória. Esta solução é a base para a concepção de uma máquina multiprocessada. Uma máquina multiprocessada é, portanto, um conjunto de processadores que opera sobre um conjunto de memória, sendo que a memória pode ser acessada uniformemente (*Uniform Memory Access* - UMA), ou seja, mesma latência de acesso, ou não uniformemente (*Non-Uniform Memory Access* - NUMA) [Wilkinson and Allen 1998].

Uma das características de uma máquina multiprocessada é de que a escalabilidade é limitada. Assim, uma vez concebida a arquitetura, não é possível adicionar um maior número de processadores. Além disso, o uso de um número elevado de processadores torna mais difícil o gerenciamento da arquitetura como a coerência de *cache* e o controle do acesso ao barramento comum.

3.2.4. *Multi-core*

Arquiteturas *Multi-core* são formadas por processadores com múltiplos núcleos de processamento [Shameem and Roberts 2005, Gepner and Kowalik 2006]. Atualmente é possível encontrar no mercado processadores com diversos *cores* para uma série de modelos e fabricantes. Além disso, já estão previstos os lançamentos de processadores com um número maior de núcleos de processamento e a tendência é de que o número de núcleos em um mesmo *chip* aumente. No caso da arquitetura *Xeon Phi* da *Intel* é possível encontrar modelos com 57, 60 e 61 *cores*.

Em arquiteturas de processadores tradicionais, com apenas um núcleo de processamento, existe apenas um conjunto de recursos (unidade de controle, operações aritméticas e lógicas, etc) concebido para a execução das instruções. Logicamente, esta estrutura pode ser bastante complexa, com a existência de *pipelines* e de replicações de pequenas unidades de processamento. Já em uma arquitetura *multi-core*, todo esse conjunto de recursos é replicado, como se vários processadores estivessem integrados em um único *chip*. Com isso, múltiplos programas ou fluxo de execuções podem ser executados simultaneamente, de acordo com o número de núcleos existente.

Um aspecto particularmente importante em arquiteturas *multi-core* é o fato de que recursos de *cache* podem ser compartilhados entre os diferentes núcleos de processamento. Com isso, diferentes fluxos de instrução podem ser executados sobre os mesmos dados em *cache*. Atualmente existem múltiplos níveis de *cache*, buscando agilizar o processo de acesso de instruções e de dados em memória por parte das instruções.

Arquiteturas *multi-core* exploram o paralelismo em nível de fluxos de execução. Isto significa que um programa tem uma maior eficiência quando este é formado por

diversos processos leves (*threads*). Um processo leve pode ser visto como a decomposição de um programa sem que o mesmo deixe de ser um único processo sob ponto de vista do sistema operacional. A implementação de programas para esse tipo de arquitetura precisa levar em consideração este fato, pois alto desempenho deixa de ser sinônimo de computação com multiprocessadores ou multicomputadores, mas passa a ser viável em um único *chip*.

3.2.5. *Cluster*

A arquitetura de *cluster* pode ser definida como um conjunto de máquinas ou nós, que cooperam entre si na execução de aplicações utilizando a troca de mensagens pela rede [Buyya 1999]. 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. Na maioria das vezes as máquinas de um *Cluster* são homogêneas, possuindo os mesmos recursos de memória, processamento, *cache* e rede.

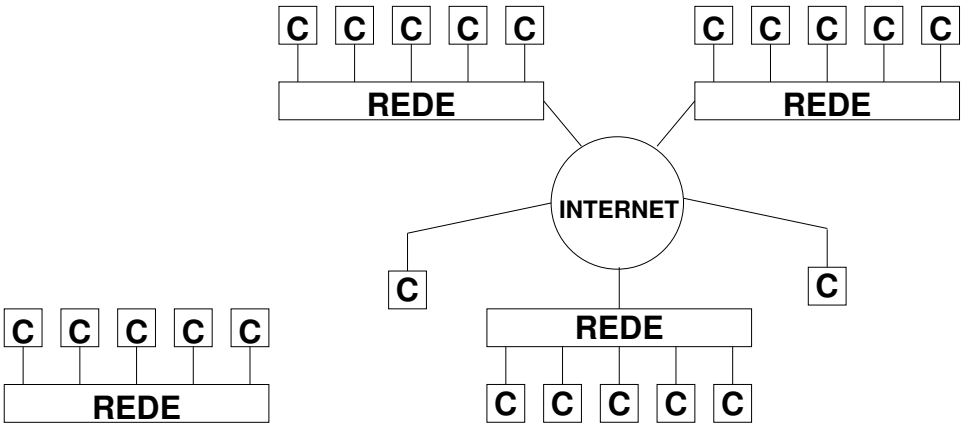
Um dos principais aspectos referentes à arquitetura de *cluster* é o gerenciamento dos recursos. O gerenciamento de *clusters* envolve diversos fatores, desde a instalação do sistema operacional, até a definição de ferramentas para a configuração, manutenção, monitoramento e escalonamento de tarefas.

Do ponto de vista da aplicação, o escalonamento de tarefas pode ser considerado a parte principal do gerenciamento de um *cluster*. O escalonamento define como são utilizados os nós de um *cluster*, fornecendo, mediante requisição, a possibilidade de uso do mesmo por parte dos usuários. Os objetivos de um escalonador são maximizar a utilização do *cluster*, maximizar a quantidade de aplicações executadas, reduzir o tempo de resposta, mesclar requisições dos usuários com as ordens administrativas e oferecer ao usuário a ilusão de uma máquina única e dedicada. Para tanto, existem algumas ferramentas disponíveis, tais como PBS, CCS, Condor e OAR [Henderson 1995, Keller and Reinefeld 1998, Thain et al. 2005, Capit et al. 2005].

3.2.6. *Grid*

Um *Grid* é uma infraestrutura de *hardware* e *software* capaz de prover uma grande capacidade de processamento (TeraFlops), a qual é necessária para a resolução de aplicações que necessitam dessa soma de recursos [Foster and Kesselman 2003, Bernholdt et al. 2005]. Em um primeiro momento, essa arquitetura foi proposta para possibilitar o compartilhamento de recursos computacionais de instituições científicas. No entanto, com a popularização das redes, a computação em *grid* também passou a ser utilizada como um ambiente de execução de aplicações que necessitam de uma capacidade computacional geralmente indisponível.

Atualmente a computação em *grid* é uma das áreas que mais recebe atenção na Computação em termos de pesquisa. O ambiente de *grid* possui várias propriedades que contribuem para o aumento de sua complexidade. Por isso, existem diversos projetos em várias áreas que se esforçam para tornar a infraestrutura de *grid* popular, assim como é o caso de *clusters*. Entre as dificuldades para a construção de uma infraestrutura de grid, pode-se destacar o gerenciamento dos recursos, a grande heterogeneidade de máquinas e



(a) Cluster (b) Grid

Figura 3.2: Ilustração do funcionamento de uma estrutura de cluster e de grid.

de conexões de rede, a grande escalabilidade, o controle de acesso aos administradores e usuários e as interfaces de programação [Foster and Kesselman 2003].

Na literatura existem diversos projetos de pesquisa em todas essas áreas. O mais popular e utilizado é o sistema Globus [Foster and Kesselman 2003]. Globus define um conjunto de serviços para facilitar a computação em *grids*, tanto para o desenvolvimento de aplicações, quanto para o de ferramentas. Além do Globus, também existem outros projetos com expressiva utilização, como o sistema MyGrid, EGEE, DataGrid, Grid5000, DAS2, PlanetLab e Naregi [Quétier and Cappello 2005].

O que difere um *cluster* de um *grid* é basicamente a infraestrutura da rede. Ilustrativamente, a Figura 3.2.6. descreve o funcionamento de um *cluster* e de um *grid*. Percebe-se que um *cluster* tem uma rede de dimensão local (e restrita), enquanto um *grid* atinge proporções globais. Consequentemente pode-se imaginar que um *grid* pode envolver um conjunto de *clusters* de diversas configurações, ou mesmo um conjunto muito grande de máquinas geograficamente distribuídas.

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.

3.3.1. Modelagem Paralela

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

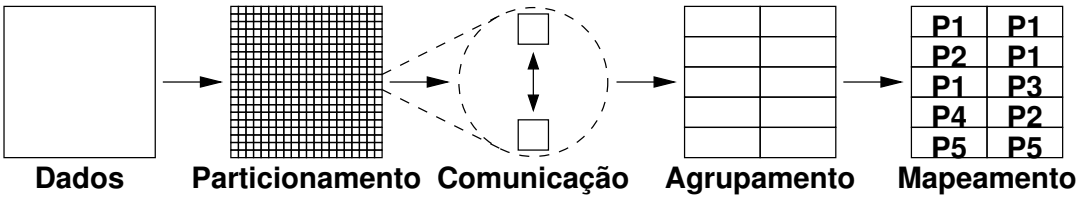


Figura 3.3: Principais etapas na paralelização de um algoritmo.

mento.

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.

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 3.3 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 ($P1, ..., P5$).

3.3.2. Modelos de Programação Paralela

Existem diversos modelos de programação paralela. Uma algoritmo pode ser decomposto essencialmente em **Nível de Dados** e em **Nível de Tarefas**.

No paralelismo em Nível de Dados, as mesmas operações são efetuadas sobre diversos subconjuntos de dados. Neste caso, um mesmo conjunto de operações é executada concorrentemente sobre partes distintas do domínio do problema.

Já no paralelismo em Nível de Tarefas, os procedimentos executados sobre os dados são realizados em processos e momentos distintos. Caso a ordem das execuções influencie no resultado é necessário que a mesma seja respeitada, formando uma espécie de *pipeline* de processamento.

As Figuras 3.4 e 3.5 ilustram respectivamente a paralelização em Nível de Dados e em Nível de Tarefas de um conjunto de dados utilizando 4 processos ($P1, ..., P4$). Em

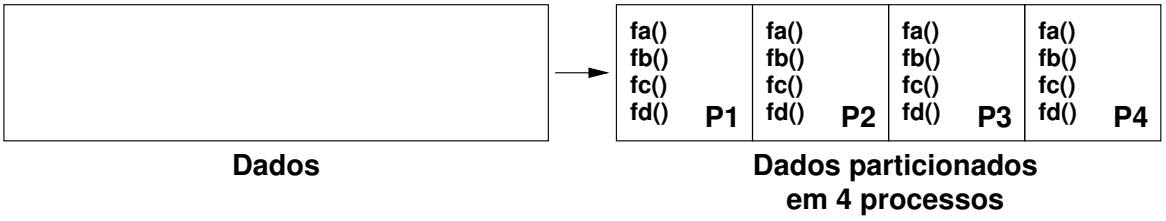


Figura 3.4: Paralelização em Nível de Dados.

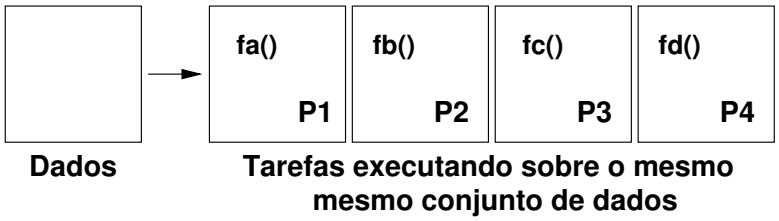


Figura 3.5: Paralelização em Nível de Tarefas.

ambos os casos 4 funções são aplicadas ($fa(), fb(), fc()\ e\ fd()$). No paralelismo em Nível de Dados as 4 funções operam em cada um dos 4 processos em conjunto de dados distintos. No paralelismo em Nível de Tarefas cada um dos 4 processos executa uma das funções sobre todo o conjunto de dados.

A paralelização de métodos de decomposição de domínios geralmente é feita através da exploração do paralelismo de dados, uma vez que é possível particionar os dados operados pelos métodos de maneira simples. Pelo fato dos dados particionados estarem inter-relacionados, faz-se necessária a sincronização ou comunicação entre os fluxos de execução. Desta forma, todos os fluxos de execução terão os valores necessários para que as operações sejam efetuadas corretamente.

3.4. Programação em Memória Compartilhada com 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 2013].

3.4.1. Programando com OpenMP

A API OpenMP é composta basicamente por diretivas de compilação e métodos da biblioteca. As diretivas são comentários no código que podem ser ignorados pelo com-

pilador, enquanto que 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 será um *Hello world* concorrente. A Figura 3.6 ilustra o primeiro exemplo em OpenMP. A construção `parallel` faz com que o **bloco estruturado** especificado entre as linhas 4 e 8 seja executado múltiplas vezes.

```
1 #include <omp.h>
2 int main(void) {
3 #pragma omp parallel
4  {
5    int id = omp_get_thread_num();
6    int nthreads = omp_get_num_threads();
7    printf("Hello␣world␣from␣thread␣%d␣of␣%d\n", id,
        nthreads);
8  }
9 }
```

Figura 3.6: Exemplo de um *Hello world* em OpenMP.

A compilação de tal programa com o `GCC` necessita da opção `-fopenmp`:

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

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=2 ./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 3.7 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 time 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 3.7) existem três regiões paralelas com quatro, seis e três *threads* respectivamente.

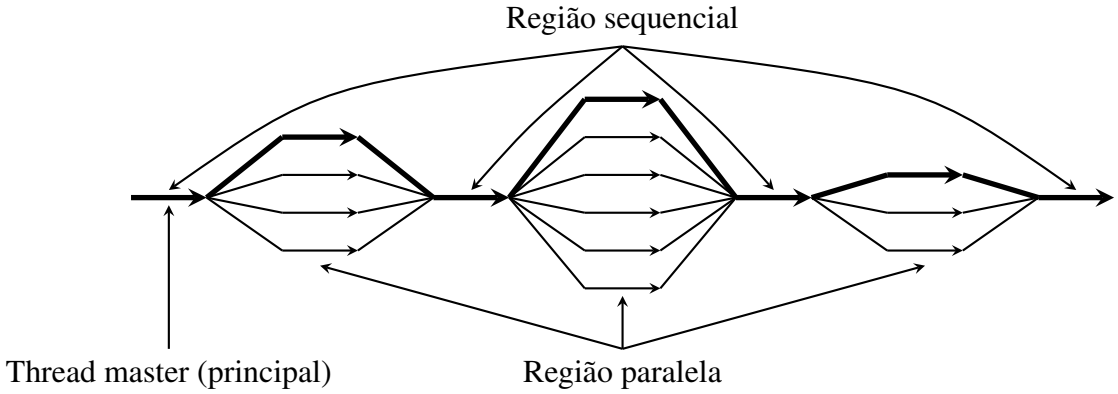


Figura 3.7: Modelo de execução *Fork/Join* do OpenMP.

A execução dentro de um bloco `parallel` é SPMD, 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()`.

Pode-se alterar o número de threads criadas em grupo de duas formas: função do OpenMP ou cláusula de compilação. A Figura 3.8 exemplifica o uso das duas para um trecho de código paralela. A primeira é através da função `omp_set_num_threads()` na linha 1 que recebe como argumento o número de threads que deverá ser utilizado na próxima região paralela. A outra forma utiliza a cláusula `num_threads` na linha 5 que especifica que a região paralela em questão executará com duas threads.

```
1 omp_set_num_threads(4);
2 #pragma omp parallel
3 { /* quatro threads */ }
4 /* parte sequencial */
5 #pragma omp parallel num_threads(2)
6 { /* duas threads */ }
```

Figura 3.8: Diferentes formas de mudar o número de threads OpenMP.

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 3.9 mostra um exemplo de laço paralelo em OpenMP onde a soma das posições do vetor `a` e `b` em `a` será dividido entre as threads da região paralela.

```
1 #pragma omp parallel
2 #pragma omp for
3   for(i = 0; i < N; i++) {
4     a[i] = a[i] + b[i];
5   }
```

Figura 3.9: Laço paralelo com OpenMP.

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

```
#pragma omp parallel for
```

sendo equivalente ao exemplo anterior.

3.4.3.1. Distribuição das Iterações entre Threads

Pode-se especificar a política de distribuição de iterações entre threads OpenMP por meio da cláusula `schedule` com sintaxe

```
#pragma omp for schedule(kind[,chunk])
```

A cláusula recebe a política de mapeamento (`kind`) e o número de iterações atribuídas de cada vez (`chunk`), sendo esse argumento opcional. As possíveis políticas são:

static - distribui blocos de iterações iguais para todas as threads e não altera essa configuração durante a execução do laço.

dynamic - cada thread remove um bloco de iterações de uma lista durante a execução do laço até que todas tenham sido executadas.

guided - as threads removem iterações dinamicamente. O tamanho do bloco de iterações inicia grande e diminui até o tamanho `chunk`.

runtime - política e bloco de iterações são definidos por funções da biblioteca ou pela variável de ambiente `OMP_SCHEDULE`.

auto - deixa a cargo da implementação do OpenMP escolher a política de escalonamento.

A escolha da política adequada e desempenho resultante podem variar de acordo com a aplicação e comportamento do laço. A política `static` é recomendada quando o trabalho é regular e altamente previsível. Porém, o uso de `dynamic` deve ser feito quando o trabalho é irregular e varia a cada iteração.

3.4.3.2. Laços Aninhados

OpenMP também suporta a paralelização de laços aninhados com a cláusula `collapse`. A cláusula recebe como argumento o número de laços a serem paralelizados, como no exemplo

```
#pragma omp parallel for collapse(2)
for(i = 0; i < N; i++) {
  for(j= 0; j < M; j++) {
    /* */
  }
}
```

Essa cláusula forma um laço simples de tamanho $N \times M$ e então paraleliza ele. O `collapse` é recomendado quando o laço externo `N` tem tamanho $O(th)$ onde th é o número de threads. Caso contrário, pode ser adequado paralelizar somente o laço externo para evitar sobrecusto com excesso de tarefas.

3.4.3.3. 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 média de `N` números da Figura 3.10. O exemplo difere do anterior com a adição da construção `parallel for` com a operação de redução `+` para acumular os resultados na variável `media`.

```
1 double media = 0.0f, A[N]; int i;
2 #pragma omp parallel for reduction (+:media)
3 for(i = 0; i < N; i++) {
4   media += A[i];
5 }
6 media = media / N;
```

Figura 3.10: Exemplo do cálculo de média com redução em OpenMP.

As operações suportadas são `+`, `-`, `*`, `min`, `max`, `&`, `|`, `^`, `&&` e `||`.

3.4.4. Sincronização

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.

Uma das diretivas básicas do OpenMP é `barrier` que insere uma barreira onde todas as threads de um grupo esperam até chegar nesse ponto. A Figura 3.11 mostra um exemplo simples de barreira onde o segundo cálculo na linha 6 depende do resultado de todas as threads na linha 4. Um programa OpenMP possui diversas barreiras implícitas como no final de uma região paralela (`parallel`) ou de um laço (`for`).

```
1 #pragma omp parallel
2 {
3  int id = omp_get_thread_num();
4  A[id] = calculo1(id);
5 #pragma omp barrier
6  B[id] = calculo2(id, A);
7 }
```

Figura 3.11: Exemplo do uso de barreira em OpenMP.

A diretiva `master` permite que apenas a thread principal (master) ou inicial execute um bloco de código. As demais threads ignoram o bloco e continuam a execução, o que significa que a diretiva não inclui uma barreira implícita. Outra diretiva semelhante é `single` que permite a execução de um bloco por apenas uma thread, não sendo necessariamente a thread principal (master). As outras threads ignoram o bloco e esperam que o bloco seja executado, ou seja, há uma barreira implícita ao final do bloco `single`. A Figura 3.12 mostra dois exemplos que usam a diretiva `master` e em seguida a diretiva `single`. Note que se utiliza a cláusula `nowait` para evitar a barreira implícita ao final do bloco `single`.

```
1 #pragma omp parallel
2 {
3 #pragma omp master
4  { /* a thread principal executa */ }
5 #pragma omp single nowait
6  { /* uma thread executa sem barreira ao final */ }
7 }
```

Figura 3.12: Exemplo de diretivas `single` e `master` em OpenMP.

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 3.13 mostra um exemplo de uso de ambas diretivas onde um valor é removido e em seguida acumulado. A remoção do valor é feito em uma região crítica, e executado por apenas uma thread por vez, e a acumulação é atômica e possivelmente concorrente.

```
1 #pragma omp parallel
2 {
3 #pragma omp critical
4  valor = remove(A);
5 #pragma omp atomic
6  total += valor;
7 }
```

Figura 3.13: Exemplo de diretivas `critical` e `atomic` para exclusão mútua.

3.4.5. 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.

threadprivate - definida na versão 4.0, cria uma cópia da variável para cada thread.

default - determina por padrão se as variáveis serão `shared` ou `none`, enquanto que `private` se aplica somente em Fortran.

A Figura 3.14 mostra um exemplo do cálculo de Pi com construções combinadas `parallel for` e de dados. Nesse exemplo especificou-se que `x` é privado em cada thread e a soma dos resultados usa `reduction` para acumular os resultados parciais de cada thread. Note que a variável `i` é privada por padrão.

```
1 #include <omp.h>
2 static long num_steps = 100000; double step;
3
4 void main(void) {
5  int i; double x, pi, sum = 0.0;
6  step = 1.0/(double)num_steps;
7 #pragma omp parallel for private(x) reduction(+:sum)
8  for(i= 0; i < num_steps; i++) {
9    x = (i+0.5)*step;
10   sum = sum + 4.0/(1.0+x*x);
11  }
12  pi = step * sum;
13 }
```

Figura 3.14: Exemplo do cálculo de Pi com atributos de dados.

3.4.6. 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:

`omp_set_num_threads(int)` - modifica o número de threads da próxima região paralela.

`omp_get_num_threads()` - retorna o número de threads do grupo atual.

`omp_get_thread_num()` - retorna o identificador da thread atual.

`omp_get_max_threads()` - máximo de threads para um novo grupo de threads.

`omp_in_parallel()` - retorna *true* se está em uma região paralela, *false* caso contrário.

`omp_set_dynamic(int)` - se o argumento é *true*, habilita o ajuste do número de threads dinamicamente se a implementação do OpenMP suportar.

`omp_get_dynamic()` - verifica se o ajuste dinâmico do número de threads está habilitado.

`omp_num_procs()` - retorna o número de processadores disponíveis.

3.4.7. Variáveis de Ambiente

As variáveis de ambiente aparecem em maiúsculo com valores recebidos em caixa baixa ou alta (não verificado pela implementação) ou em números. As principais variáveis são:

`OMP_NUM_THREADS(int)` - especifica o número de threads a serem usados nas regiões paralelas.

`OMP_STACKSIZE(size[B|K|M|G])` - tamanho da pilha criada para cada thread pela implementação do OpenMP usada.

`OMP_WAIT_POLICY(active | passive)` - define a política de espera em threads ociosas em barreiras e *locks*, sendo `active` para espera ativa (*busy wait*) e `passive` para espera passiva.

`OMP_PROC_BIND(true | false)` - determina se as threads poderão mover para diferentes processadores em tempo de execução. O valor `true` define que as threads não mudam, enquanto `false` permite as migrações.

3.5. Programação em Memória Distribuída com 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) [MPI FORUM 1994], [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 (embora degrade a performance geral) 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 interface MPI foi definida em 1994 pelo Fórum MPI [Gropp et al. 1996]. Em 1996 foi estabelecido o padrão MPI-2 [Gropp et al. 1999]. Ele incluía novos recursos como o de entrada e saída paralela e controle dinâmico de processos, além de suportar as linguagens ANSI C, Fortran 90 e ANSI C++. Algumas melhorias adicionais foram incluidas no padrão MPI-3, no ano de 2012 [MPI-FORUM 2012].

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 2014], [MPICH2 2014].

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 3.15 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 6.

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

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

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

```
1 #include <mpi.h>
2 #include <stdio.h>
3 int main(int argc, char **argv)
4 {
5  MPI_Init(&argc, &argv);
6  printf("Hello_World!\n");
7  MPI_Finalize();
8  return 0;
9 }
```

Figura 3.15: Exemplo de um *Hello world* em MPI.

```
$ ./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
```

Um segundo exemplo oferece mais alguns métodos essenciais para o desenvolvimento de aplicações paralelas. A Figura 3.16 apresenta o uso das funções `MPI_Comm_rank()` e `MPI_Com_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_COM_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 3.17 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 == 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.

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

Figura 3.16: Exemplo aprimorado de um *Hello world* em MPI.

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.

Na Figura 3.18 há um exemplo de uso da função `MPI_Barrier()`, que permite a sincronização dos processos através de uma barreira. Os processos esperam até que todos cheguem o mesmo ponto de execução do código, podendo prosseguir posteriormente.

3.5.4. Considerações

Através do uso das 7 funções básicas vistas nos 4 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.

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

```
1 #include <mpi.h>
2 #include <stdio.h>
3 #include <string.h>
4 int main(int argc, char **argv){
5   int rank, size, tag, i;
6   MPI_Status status;
7   char msg[20];
8   MPI_Init(&argc, &argv);
9   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10  MPI_Comm_size(MPI_COMM_WORLD, &size);
11  if(rank == 0) {
12    strcpy(msg, "Hello␣World!\n");
13    for(i=1; i < size; i++)
14    MPI_Send(msg, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
15  } else {
16    MPI_Recv(msg, 13, MPI_CHAR, 0, tag, MPI_COMM_WORLD,
        &status);
17    printf("Process␣%d:␣Message␣received:␣%s\n", rank,
        msg);
18  }
19  MPI_Finalize();
20  return 0;
21 }
```

Figura 3.17: Exemplo de envio e recebimento de mensagens em MPI.

3.6. Programação Híbrida com MPI e OpenMP

Atualmente, grande parte das arquiteturas de alto desempenho é híbrida com paralelismo de memória compartilhada e distribuída como em um *cluster*. Nessa caso, pode-se explorar o paralelismo de duas formas distintas. A primeira consiste em utilizar somente processos MPI, ou seja, em processadores *multi-core* a execução teria um processo por *core*. A segunda alternativa é utilizar processos MPI com threads OpenMP. Dessa forma, assumindo uma arquitetura com M processadores e N *cores* em cada, uma execução poderá ter um processo por nó com $N * M$ threads OpenMP ou M processos por nó com N threads por processador.

A programação nesse tipo de arquiteturas depende, principalmente, do suporte a chamadas concorrentes na implementação MPI. Os principais passos de um programa MPI com threads OpenMP são:

1. Inicializar o MPI com `MPI_Init_thread` para requisitar o suporte a chamadas concorrentes;

2. Iniciar a região paralela do OpenMP;

3. Dentro de uma região paralela, realizar trocas de mensagem MPI com apenas uma

```
1 #include <mpi.h>
2 #include <stdio.h>
3 int main(int argc, char **argv) {
4   int rank, size;
5   MPI_Init(&argc, &argv);
6   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7   MPI_Comm_size(MPI_COMM_WORLD, &size);
8   printf("I'm␣%d␣of␣%d\n", rank, size);
9   if(rank == 0) {
10    printf("(%d)␣->␣Primeiro␣a␣escrever!\n", rank);
11    MPI_Barrier(MPI_COMM_WORLD);
12  } else {
13    MPI_Barrier(MPI_COMM_WORLD);
14    printf("(%d)␣->␣Agora␣posso␣escrever!\n", rank);
15  }
16  MPI_Finalize();
17  return 0;
18 }
```

Figura 3.18: Exemplo de envio e recebimento de mensagens em MPI.

thread;

4. Finalizar o MPI.

Dentro do que foi abordado anteriormente com relação a MPI e OpenMP, deve-se discutir agora a inicialização e troca de mensagens em uma região paralela do OpenMP.

A inicialização do MPI para uso de threads deve requisitar o suporte a chamadas MPI concorrentes no começo do programa por meio da função `MPI_Init_thread`. Essa função especifica o nível de suporte requisitado pelo programa e retorna qual nível a implementação MPI pode fornecer. Assume-se que uma única thread do processo MPI fará as comunicações pois nem todas as implementações MPI suportam chamadas concorrentes. Para tanto, a implementação MPI deverá suportar pelo menos dois tipos de níveis:

- `MPI_THREAD_FUNNELED` - o processo pode ser multi-thread, mas apenas a *thread* principal (`master` do OpenMP) faz chamadas MPI.
- `MPI_THREAD_SERIALIZED` - o processo pode ser multi-thread e qualquer *thread* pode fazer chamadas MPI, mas não de forma concorrente.

As Figuras 3.19 e 3.20 mostram exemplos dos dois níveis de suporte a threads. No primeiro caso, com `MPI_THREAD_FUNNELED` (Figura 3.19) duas barreiras foram inseridas, antes e depois de uma construção `master`. Isso deve-se ao fato dessa construção não ter barreira implítica ao fim do bloco. Enquanto que no segundo caso com `MPI_THREAD_SERIALIZED` (Figura 3.20) a barreira foi inserida apenas antes da construção `single` pois ela possui uma barreira implícita ao fim.

```
1 MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &r);
2 #pragma omp parallel
3 {
4   #pragma omp barrier
5   #pragma omp master
6   { /* Chamadas MPI_* permitidas */ }
7   #pragma omp barrier
8 }
```

Figura 3.19: Utilização de MPI e OpenMP com comunicação na *thread* principal..

```
1 MPI_Init_thread(&argc,&argv, MPI_THREAD_SERIALIZED,&r);
2 #pragma omp parallel
3 {
4   #pragma omp barrier
5   #pragma omp single
6   { /* Chamadas MPI_* permitidas */ }
7 }
```

Figura 3.20: Utilização de MPI e OpenMP com comunicação em qualquer *thread*..

3.7. Conclusão

O uso de arquiteturas paralelas 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.

No caso de *MPI*, não existe uma alternativa, uma vez que PVM deixou de ser usado e *Remote Procedure Call* - RPC é uma alternativa para sistemas distribuídos. Portanto, pode-se dizer que *MPI* é um padrão *de facto*. Já para ambientes de memória compartilhada, além de *OpenMP*, o próprio MPI pode ser utilizado. Além disso existem as interfaces: *PThread*, voltada para a programação de sistemas operacionais; *Cilk*, como uma ótima alternativa para a exploração de paralelismo de tarefas; e *TBB*, indicado para ambientes Intel, usando C++. Independente das alternativas existentes, *OpenMP* oferece um paralelismo de forma bastante simples.

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, é possível maximizar o uso dos recursos computacionais paralelos

disponíveis nas arquiteturas atuais.

3.8. Bibliografia

[Andrews 2001] Andrews, G. R. (2001). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, Reading, Massachusetts.

[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.

[Bernholdt et al. 2005] Bernholdt, D., Bharathi, S., Brown, D., Chanchio, K., Chen, M., Chervenak, A., Cinquini, L., Drach, B., Foster, I., Fox, P.; Garcia, J., and Kesselman, C. (2005). The earth system grid: supporting the next generation of climate modeling research. In *Proceedings of the IEEE*, volume 93, pages 485–495. Loas Alamitos UKA, IEEE Computer Society.

[Buyya 1999] Buyya, R. (1999). *High-Performance Cluster Computing: Architectures and Systems*. Prentice Hall, USA.

[Capit et al. 2005] Capit, N., Costa, G. D., Georgiou, Y., Huard, G., Martin, C., Mounie, G., Neyron, P., and Richard, O. (2005). A batch scheduler with high level components. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 776–783, Washington, DC, USA. IEEE Computer Society.

[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.

[Dongarra et al. 2002] Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., and White, A., editors (2002). *The Sourcebook of Parallel Computing*. Elsevier, San Francisco, CA, USA.

[Dongarra et al. 1991] Dongarra, J. J., Duff, I. S., Sorensen, D. C., and van der Vorst, H. A. (1991). *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA.

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

[Foster and Kesselman 2003] Foster, I. and Kesselman, C. (2003). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, CA, USA, second edition.

[Gepner and Kowalik 2006] Gepner, P. and Kowalik, M. F. (2006). Multi-core processors: New Way to Achieve High System Performance. In *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, pages 9–13, Washington, DC, USA. Institute of Electrical and Electronics Engineers (IEEE).

[Gropp et al. 1999] Gropp, W., Ewing, L., and Thakur, R. (1999). Using MPI-2 - Advanced Features of the Message-Passing Interface. *The Mit Press*.

[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.

[Henderson 1995] Henderson, R. L. (1995). Job scheduling under the portable batch system. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294, London, UK. Springer-Verlag.

[Keller and Reinefeld 1998] Keller, A. and Reinefeld, A. (1998). Ccs resource management in networked hpc systems. In *HCW '98: Proceedings of the Seventh Heterogeneous Computing Workshop*, page 44, Washington, DC, USA. IEEE Computer Society.

[MPI FORUM 1994] MPI FORUM (1994). MPI: A Message-Passing Interface Standard. *Journal of Supercomputing Applications*, 8(3-4):165–414.

[MPI-FORUM 2012] MPI-FORUM (2012). MPI: A Message-Passing Interface Standard Version 3.0. Chapter author for Collective Communication, Process Topologies, and One Sided Communications. Available at: <http://www.unixer.de/ htor/publications/>. Last access: November, 2014.

[MPICH2 2014] MPICH2 (2014). MPICH High-Perfomance Portable MPI. Available at: <http://www.mpich.org>. Last access: November, 2014.

[Open-MPI 2014] Open-MPI (2014). Open MPI: Open Source High Performance Computing. Available at: <http://www.open-mpi.org>. Last access: November, 2014.

[OpenMP 2013] OpenMP (2013). OpenMP Application Programming Interface - Version 4.0. Available at: <http://www.openmp.org>. Last access: November, 2014.

[Osthoff et al. 2011] Osthoff, C., Grunmann, P., Boito, F., Kassick, R., Pilla, L., Navaux, P., Schepke, C., Panetta, J., Maillard, N., Silva Dias, P., and Walko, R. (2011). Improving Performance on Atmospheric Models through a Hybrid OpenMP/MPI Implementation. In *Proc. of the 9th ISPA*, pages 69–74.

[Panetta et al. 2007] Panetta, J., de Souza Filho, P., da Cunha Filho, C., da Motta, F., Pinheiro, S., Pedrosa, I., Rosa, A., Monnerat, L., Carneiro, L., and de Albrecht, C. (2007). Computational Characteristics of Production Seismic Migration and its Performance on Novel Processor Architectures. In *Proc. of the 19th SBAC-PAD*, pages 11–18.

[Quétier and Cappello 2005] Quétier, B. and Cappello, F. (2005). A Survey of Grid Research Tools: Simulators, Emulators and Real Life Platforms. In *17th IMACS World Congress (IMACS 2005)*, Paris, France.

[Schepke et al. 2009] Schepke, C., Maillard, N., and Navaux, P. O. A. (2009). Parallel lattice boltzmann method with blocked partitioning. *Int. J. Parallel Program.*, 37(6):593–611.

[Schepke et al. 2013] Schepke, C., Maillard, N., Schneider, J., and Heiss, H.-U. (2013). Online Mesh Refinement for Parallel Atmospheric Models. *International Journal of Parallel Programming*, 41(4):552–569.

[Shameem and Roberts 2005] Shameem, A. and Roberts, J. (2005). *Multi-Core Programming - Increasing Performance through Software Multithreading*. Intel, Hillsboro, OR.

[Sims et al. 2000] Sims, J. S., Hagedorn, J. G., Ketcham, P. M., and Satterfield, S. G. (2000). Accelerating Scientific Discovery Through Computation and Visualization. *Journal of Research of the National Institute of Standards and Technology*, 105(6):875–894.

[Souto et al. 2007] Souto, R., Avila, R., Navaux, P., Py, M., Maillard, N., Diverio, T., Velho, H., Stephany, S., Preto, A., Panetta, J., Rodrigues, E., and Almeida, E. (2007). Processing Mesoscale Climatology in a Grid Environment. In *Proc. of the 7th CCGrid.*, pages 363–370.

[Thain et al. 2005] Thain, D., Tannenbaum, T., and Livny, M. (2005). Distributed Computing in Practice: The Condor Experience. *Concurrency - Practice and Experience*, 17(2-4):323–356.

[TOP 500 2014] TOP 500 (2014). Top 500 Supercomputing Site. Available at: <http://www.top500.org>. Last access: November, 2014.

[Wilkinson and Allen 1998] Wilkinson, B. and Allen, M. (1998). *Parallel Programming: Using Networked Workstations and Parallel Computers*. Prentice Hall, New Jersey.

[Xavier et al. 2007] Xavier, C., Sachetto, R., Vieira, V., dos Santos, R. W., and Jr., W. M. (2007). Multi-level Parallelism in the Computational Modeling of the Heart. In *SBAC-PAD 2007*, pages 3–10, Gramado. IEEE.