Execução de Aplicações em Ambientes Concorrentes

Prof. Philippe O. A. Navaux

Marcos E. Barreto, Rafael B. Ávila, Fábio A. D. de Oliveira

Instituto de Informática
Universidade Federal do Rio Grande do Sul
E-mail: mcluster-l@scliar.inf.ufrgs.br

1. Introdução

A concorrência é uma técnica freqüentemente empregada no desenvolvimento de aplicações e sistemas computacionais que objetivam tempos de execução baixos e o uso eficiente dos recursos providos pela arquitetura na qual executam. A idéia de dois ou mais processos independentes disputando o acesso a recursos compartilhados, tais como processadores, unidades de disco, interfaces de redes, servidores, etc., provê as duas principais características de um ambiente concorrente:

- alto desempenho: é obtido através da sobreposição de computação e comunicação, garantindo que enquanto um processo realiza alguma operação de entrada e saída (geralmente de natureza bloqueante), um segundo processo possa utilizar o processador para a realização de algum cálculo, reduzindo, dessa forma, o tempo total de execução.
- eficiência: este aspecto refere-se ao uso "ótimo" de uma determinada arquitetura, de modo que uma aplicação possa utilizar todos os recursos providos pela arquitetura (dois processadores, por exemplo) buscando atingir alto desempenho e aumentar o número de tarefas realizadas dentro de um intervalo de tempo (*throughput*).

Um ambiente concorrente pode ser visto em diferentes níveis, desde o *hardware* utilizado até as aplicações executadas. Em termos de *hardware*, a arquitetura pode prover dois ou mais processadores, discos e interfaces de redes redundantes, ou até mesmo um conjunto de nodos multiprocessados interligados por redes de comunicação de alta velocidade – comumente referenciados como arquiteturas de agregados (*clusters*). Tanto o sistema operacional quanto as linguagens e bibliotecas de programação devem ser capazes de explorar eficientemente os recursos fornecidos pela arquitetura, provendo ao programador um conjunto de primitivas de programação que permitam-lhe expressar características concorrentes em suas aplicações – por exemplo, a criação de diferentes fluxos de execução (*threads*).

No nível mais alto estão as aplicações, as quais apresentam diferentes modelos de solução de problemas e, conseqüentemente, diferentes requisitos computacionais. Dessa forma, um ambiente concorrente deve ser capaz de suportar, através de seu

sistema operacional e ferramentas de desenvolvimento, uma variada gama de aplicações, cada uma com suas características particulares.

De uma maneira geral, observa-se que um ambiente concorrente pode ser visto então como um conjunto de níveis, cada qual provendo diferentes recursos voltados à obtenção de desempenho e eficiência. Mesmo que a arquitetura-alvo não seja multiprocessada, é possível ao sistema operacional e às linguagens de programação simular a concorrência entre diversas aplicações – neste caso, o tempo de execução tende a aumentar – através de políticas de escalonamento de processos que procuram sobrepor comunicação e computação. Ainda, uma rede de computadores pode ser facilmente utilizada para simular uma máquina virtual provida de diversos processadores e, dessa forma, executar aplicações de caráter distribuído.

O presente tutorial objetiva a apresentação de conceitos ligados à programação e execução de aplicações concorrentes. Serão abordados os diferentes modelos de programação de aplicações, ressaltando as necessidades computacionais de cada um; bem como serão apresentados os dois principais modelos de comunicação – troca de mensagens e memória compartilhada – utilizados para permitir que diferentes processos de uma aplicação concorrente possam trocar dados durante suas execuções. Por último, serão abordados alguns exemplos de aplicações concorrentes comumente utilizadas para a avaliação de desempenho de arquiteturas e ambientes de programação paralela e distribuída.

2. Paradigmas de Sistemas Paralelos

Como mencionado na seção anterior, a concorrência num sistema é encontrada em diversos níveis deste. Pode-se caracterizar, como linha geral, que existem três a quatro níveis de paradigmas de concorrência: o nível de **algoritmos**, o nível de **implementação** e o nível de **arquitetura**. Alguns autores [5] dividem a implementação em modelos de programação e modelos de gerenciamento.

2.1. Modelos de algoritmos paralelos

Os **modelos de algoritmos paralelos** podem ser resumidos em cinco paradigmas principais:

Divisão e Conquista – O paradigma de divisão e conquista paralelo é semelhante ao de um sistema seqüencial em suas partes. Seu princípio é dividir uma tarefa em diversas partes menores e atribuí-las a processos filhos. Os filhos processarão as tarefas em paralelo e retornarão os resultados para seus pais, que as integrarão. Esta ação de divisão e integração será executada de forma recursiva em cima da tarefa até sua execução completa. É um dos paradigmas mais simples de implementar, porém sua desvantagem é a dificuldade de obter um balanceamento de carga na divisão das tarefas (ver figura 1.a).

Pipeline – No paradigma pipeline um número de processos forma um pipeline virtual. Um fluxo contínuo de dados entra no primeiro estágio do pipeline e os

processos são executados nos demais estágios complementares, de forma simultânea (ver figura 1.b).

Mestre/Escravo (*process farm*) – Neste paradigma, um processo mestre executa as tarefas essenciais do programa paralelo e divide o resto das tarefas para processos escravos. Quando um processo escravo termina sua tarefa, ele informa o mestre que atribui uma nova tarefa para o escravo. Este paradigma é bastante simples, visto que o controle está centralizado num processo mestre; sua desvantagem é que o mestre torna-se um gargalo na comunicação (ver figura 1.c).

***Pool* de Trabalho** – Neste modelo, um *pool* (conjunto) de tarefas é disponibilizado por uma estrutura de dados global e um determinado número de processos é criado para executar esse conjunto de tarefas. No início só existe um único pedaço de tarefa; gradativamente os processos buscam pedaços da tarefa e imediatamente passam a executá-los, espalhando o processamento. O programa paralelo termina quando o *pool* de trabalho fica vazio. Este tipo de modelo facilita o balanceamento da carga; por outro lado é difícil obter um acesso eficiente e homogêneo aos múltiplos processos (ver figura 1.d).

Fases Paralelas – Neste modelo, a aplicação consiste num número de etapas, onde cada etapa é dividida em duas fases: uma fase de computação, quando os múltiplos processos executam processamentos independentes, seguida de uma fase de interação, quando os processos executam uma ou mais operações de interação síncrona, tais como barreiras ou comunicações bloqueantes (ver figura 1.e).

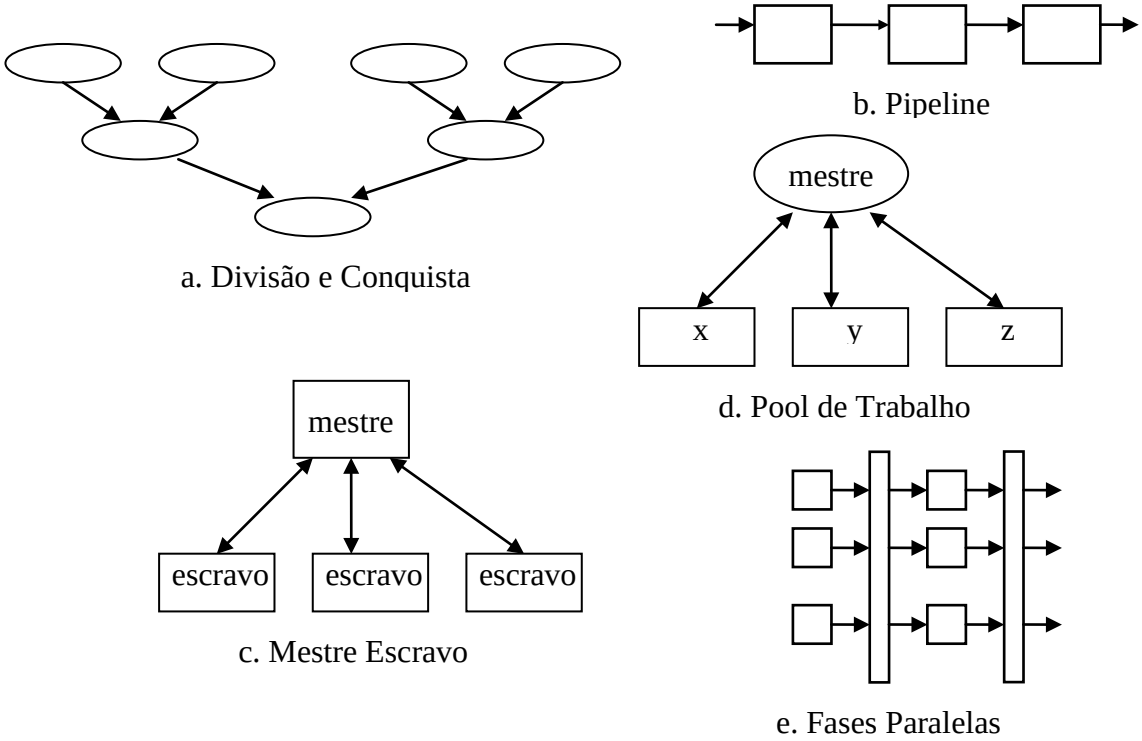

Figura 1: Modelos de Algoritmos Paralelos.

2.2 Modelos de programação paralela

Os **modelos de programação paralela** dividem-se em um modelo de programação implícita e três modelos de programação explícita: Passagem de Mensagens, Paralelismo de Dados e Variáveis Compartilhadas.

Paralelismo Implícito – O paralelismo implícito acontece quando o programador não explicita o paralelismo através de comandos, deixando para o compilador e os sistemas de execução a responsabilidade de explorar automaticamente o paralelismo existente no programa. Normalmente, o compilador detecta os trechos do programa que poderão ser executados em paralelo, através da análise de dependências. São exemplos o Kap e o Forge.

Paralelismo de Dados – Este modelo é aplicável para modos de execução SIMD ou SPMD. A idéia é executar a mesma instrução ou trecho de programa com conjuntos de dados diferentes em nodos de processamento diferentes. Exemplos de linguagens são o Fortran 90 e o HPF.

Passagem de Mensagens – Um programa que emprega passagem de mensagens consiste em múltiplos processos, cada qual com seu fluxo de controle, executados de forma assíncrona. Este modelo pode suportar paralelismo de controle (MPMD) e paralelismo de dado (SPMD) .Os processos possuem espaços de endereçamento separados e o usuário especifica de forma explícita a carga e os dados para cada um deles. Exemplos de bibliotecas são o PVM e o MPI.

Variáveis Compartilhadas – O modelo de memória compartilhada é similar ao modelo de paralelismo de dados, onde existe um espaço de endereçamento único; e é parecido ao de passagem de mensagens, o qual é multiprogramado e assíncrono. A diferença é que os dados residem num espaço de endereçamento único compartilhado. A comunicação é feita através da escrita e leitura de variáveis compartilhadas e a sincronização é explícita. Um exemplo é o modelo de programação X3H5.

2.3 Modelos de arquiteturas paralelas

Os **modelos de arquiteturas paralelas** normalmente são classificados em dois tipos principais: SIMD (*Single Instruction Stream, Multiple Data Stream*) e MIMD (*Multiple Instruction Stream, Multiple Data Stream*).

Máquinas SIMD – Este tipo de máquina normalmente é dirigida para aplicações específicas. Nesta arquitetura, uma única instrução, através de uma unidade de controle, atua de forma síncrona sobre um conjunto de dados diferentes, distribuídos ao longo de processadores elementares (ver figura 2.a). Os próximos modelos de arquiteturas são do tipo MIMD.

Máquinas Vetoriais (PVP) – Estas máquinas têm como característica básica o fato de possuírem processadores compostos de vários pipelines vetoriais de alto poder de processamento, capazes de fornecerem alguns GFlops de desempenho. Estas poucas unidades processadoras estão interligadas através de chaves de alta velocidade a uma

memória comum compartilhada, formando uma estrutura MIMD (ver figura 2.b). Exemplos destas máquinas são os computadores Cray e NEC.

Multiprocessadores Simétricos (**SMP)** – A estrutura destas máquinas é semelhantes às anteriores, diferenciando-se no fato de empregarem doi sou mais microprocessadores como unidades de processamento. Estes microprocessadores simétricos estarão interligados, em geral, por um barramento de alta velocidade a uma memória compartilhada (ver figura 2.c). Exemplos destas máquinas são os DEC Alpha Server 8400 e o SGI Power Challenge.

Máquinas Massivamente Paralelas (MPP) - Já os MPPs são máquinas com diversos microprocessadores interligados através de uma rede de interconexão normalmente proprietária. Cada nó de processamento da malha de interconexão pode possuir mais de um processador e podem existir máquinas com milhares destes nós. A grande diferença em relação aos dois últimos modelos de máquinas é que estas não possuem uma memória compartilhada (ver figura 2.d). Exemplos destas máquinas são o Ncube e o Paragon, da Intel e a máquina TFLOP.

Multiprocessadores com Memória Compartilhada Distribuída (DSM) – Estas máquinas são semelhantes às SMP possuindo um conjunto de microprocessadores interligados através de uma rede de interconexão de alta velocidade. A diferença está no fato de que a memória global compartilhada na verdade está fisicamente distribuída entre os nós; porém, para o usuário é como se ele estivesse acessando um espaço de endereçamento único (ver figura 2.f). Um exemplo destas máquinas é o T3D da Cray.

Clusters – Sob este nome estão máquinas cujo princípio básico é o emprego de uma rede de custo baixo, porém de alto desempenho, interligando nodos que podem possuir mais de um processador. Estas máquinas são geralmente classificadas em dois tipos: aquelas em que os nodos são estações de trabalho, conhecidas por COW (*Clusters of Workstations*); e aquelas em que os nodos são computadores comuns do tipo Pentium, em geral no formato Dual ou Quad (ver figura 2.e).

Uma característica importante destas máquinas é que os processadores são máquinas comuns e completas, às vezes com todos os periféricos. Exemplos destas máquinas são o Berkeley NOW e o Beowulf, as duas do tipo COW; assim como vários *clusters* oferecidos pelos principais fabricantes através da agregação de computadores PCs da sua linha de máquinas. Na evolução dos *clusters* de PCs encontram-se configurações em que são empregados redes baseadas em chaveamento para a interconexão dos nodos, tais como a Myrinet [2], ou mecanismos de comunicação em anel tais como a SCI [6]. Os sistemas operacionais empregados muitas vezes são o Linux para PCs ou Unix para estações.

Na tabela 1 são indicadas formas de acesso à memória, divididas em UMA (acesso uniforme à memória), NUMA (acesso não uniforme à memória) e NORMA (acesso não remoto à memória), ou seja, através de mensagens.

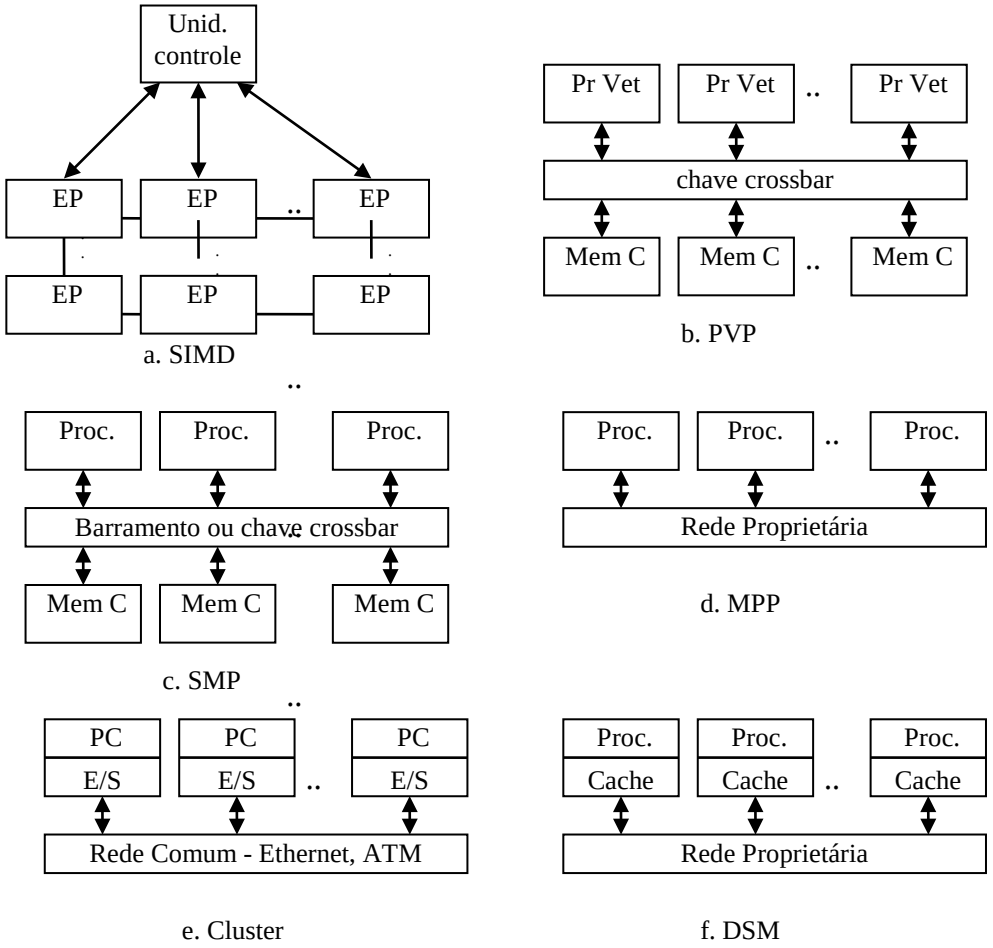

Figura 2: Modelos de Arquiteturas Paralelas.

Tipo	**SIMD**	**PVP**	**SMP**	**MPP**	**DSM**	**Cluster**
Estrutura	SIMD	MIMD	MIMD	MIMD	MIMD	MIMD
Comunicação	direta	memória compart.	memória compart.	troca de mensagem	memória compart.	troca de mensagem
Interconexão	chave específica	chave crossbar	Barramento ou crossbar	rede específica	rede específica	rede comum
número de nós	100 a 1000	10	10	100 a 1000	10 a 1000	10 a 1000
Endereçamento	único	único	único	múltiplo	único	múltiplo
Processador	específico	específico	específico	específico	específico ou comum	comum
Acesso à memória	UMA	UMA	UMA	NORMA	NUMA	NORMA

Tabela 1: Modelos de Arquiteturas Paralelas.

3. Modelos de Comunicação

Para a construção de um programa paralelo, há que se pensar no modo de comunicação entre os processadores envolvidos na sua execução. Existem, basicamente, dois paradigmas de comunicação: troca de mensagens e compartilhamento de memória.

3.1 Comunicação por troca de mensagens

O paradigma de troca de mensagens tem sido tradicionalmente empregado em sistemas fracamente acoplados, representados pelas arquiteturas baseadas em memória distribuída, em que cada processador tem acesso somente à sua memória local e, por conseguinte, a comunicação, necessariamente, dá-se por meio de uma rede de interconexão.

Conceitualmente, a idéia de troca de mensagens é totalmente independente de hardware, sistema operacional, linguagens de programação e bibliotecas. Quando do desenvolvimento de um programa paralelo por troca de mensagens, o programador deve distribuir os dados explicitamente entre os processadores. Visto que os espaços de endereçamento dos processos que compõem o programa paralelo são distintos, concebeu-se a abstração de **mensagem**, que pode ser enviada de um processo a outro por um **canal de comunicação**. Tal é denotado no programa através de primitivas do tipo *send* e *receive*, as quais supõem um processo que pretende enviar (*send*) uma mensagem a outro, que espera recebê-la (*receive*). Entre os processos comunicantes diz-se que existe um canal de comunicação.

Na prática, os programadores dispõem de bibliotecas de comunicação com primitivas à semelhança de *send* e *receive*. As bibliotecas de comunicação que obtiveram maior aceitação foram, indubitavelmente, MPI [8] e PVM [4], ambas com suporte às linguagens de programação C e Fortran.

MPI (*Message Passing Interface*) é, na verdade, um padrão estabelecido pelo MPI *forum*, especificando a sintaxe e semântica de um conjunto de funções de comunicação que, em um dado momento, julgou-se atender às necessidades de aplicações paralelas típicas. Existem diversas implementações do padrão MPI, para as mais variadas arquiteturas.

O padrão MPI incorporou os modelos de comunicação por troca de mensagens habitualmente descritos na literatura, prevendo tanto a comunicação síncrona quanto a assíncrona, permitindo, em qualquer dos casos, o uso de primitivas do tipo *send* e *receive* bloqueantes ou não-bloqueantes. Ademais, MPI destaca-se pelo excelente suporte à comunicação coletiva, que facilita o desenvolvimento de aplicações que necessitam de efetuar freqüentes operações sobre matrizes. Além das tradicionais primitivas *MPI_Send*, *MPI_Recv* e suas variantes, sobressaem-se funções para a comunicação coletiva entre processos, tais como *MPI_Scatter*, *MPI_Gather*, *MPI_Reduce*, *MPI_Alltoall* e generalizações destas, bem como a primitiva *MPI_Bcast*. Para a sincronização entre um grupo de processos MPI, a abstração de **barreira** é representada pela primitiva *MPI_Barrier*.

No que concerne ao modelo de programação, MPI segue a filosofia SPMD (*Single Program Multiple Data*), de modo que os processos componentes do programa paralelo apresentam exatamente o mesmo código; adicionalmente, não é permitida a criação ou destruição de processos durante a execução da aplicação. Dito de outra forma, o modelo de processos é estático.

Outra biblioteca de comunicação notoriamente difundida na área de Processamento Paralelo e Distribuído é PVM (*Parallel Virual Machine*). Diferentemente de MPI, PVM não é um padrão formal, embora as implementações mais recentes estejam efetivamente convergindo.

PVM é mais do que uma biblioteca de comunicação. Trata-se de um software básico com suporte próprio ao gerenciamento de processos e centrado no conceito de máquina virtual. O usuário do ambiente PVM pode adicionar máquinas físicas — e.g., PCs de uma rede local ou *cluster* — à máquina virtual PVM. Após proceder às devidas configurações da máquina virtual, pode-se disparar processos que fazem uso das primitivas da biblioteca de comunicação PVM.

A grande diferença entre PVM e MPI está no modelo de processos. PVM suporta o modelo de programação MPMD (*Multiple Program Multiple Data*), possibilitando a criação e destruição de processos em tempo de execução da aplicação paralela. Em contrapartida, os serviços de comunicação ponto a ponto e comunicação coletiva são pífios, quando cotejados com os da biblioteca MPI. Os grupos de processos PVM são dinâmicos e abertos, ao passo que em MPI os grupos são eminentemente estáticos.

Desde o projeto inicial, em 1989, PVM sempre favoreceu mecanismos de interoperabilidade, ensejando a construção de uma máquina virtual única, possivelmente composta por computadores de diferentes arquiteturas. Neste contexto, os *daemons* de comunicação PVM exercem um papel fundamental. Ao contrário, o padrão MPI procurou promover um equilíbrio entre facilidade de uso e desempenho para aplicações paralelas.

As tendências atuais apontam para a união das bibliotecas MPI e PVM. Recentemente, foi proposto o padrão MPI-2, agregando ao original o suporte ao modelo MPMD, além de primitivas para o compartilhamento de memória, mesmo em ambientes fisicamente distribuídos. De modo análogo, PVM também está evoluindo na direção de MPI, com a introdução de melhores serviços de comunicação coletiva. Em sendo assim, há correntes que advogam a junção de ambas as bibliotecas em um ambiente de programação único e abrangente em termos de aplicações e serviços fornecidos.

3.2 Comunicação por memória compartilhada

A evidente desvantagem que acomete o paradigma de comunicação por troca de mensagens para a paralelização de código é a dificuldade de programação. Programadores afeitos ao modelo seqüencial podem intimidar-se ante uma abordagem

radicalmente diferente, que requer a distribuição explícita de dados, originalmente contíguos, entre os diferentes processadores que executarão o programa paralelo.

O modelo seqüencial baseia-se essencialmente na utilização de um único espaço de endereçamento, pertencente ao fluxo de execução do programa. Deste modo, o compartilhamento de memória estendido à prática de programação paralela visa a agregar o melhor de dois mundos: um paradigma de fácil assimilação e a busca de alto desempenho. A noção de memória compartilhada faz parte da cultura dos desenvolvedores de aplicações seqüenciais.

A construção de aplicações paralelas norteada pela comunicação por memória compartilhada enquadra-se, basicamente, em duas categorias: *multithreading* ou DSM. Assim como a comunicação por troca de mensagens é adequada a máquinas com memória distribuída, a utilização de múltiplos fluxos de execução (*threads*), que compartilham um espaço de endereçamento, é a abordagem mais natural a ser empregada em arquiteturas baseadas em vários processadores que acessam uma memória comum, tais como SMPs.

No caso de arquiteturas com memória fisicamente distribuída, a programação por compartilhamento de memória exige uma camada de software que forneça a abstração de memória compartilhada, a exemplo da biblioteca TreadMarks [1]. Isto se denomina memória compartilhada distribuída — DSM (Distributed Shared Memory) —, ou seja, a memória é fisicamente distribuída mas logicamente compartilhada.

Em se tratando da biblioteca TreadMarks, tem-se a implementação do mecanismo de DSM totalmente em software. Há implementações de DSM em hardware, como por exemplo, as arquiteturas baseadas em conjuntos de SMPs interconectados por uma tecnologia como SCI (Scalable Coherent Interface).

Quando do desenvolvimento de aplicações, pode-se lançar mão de primitivas para gerenciamento de *theads* ou, ainda, de abstrações que encerram a idéia de segmentos de memória compartilhados, acessáveis por primitivas que permitem escrever ou ler dados. No que tange à sincronização no acesso simultâneo a regiões de memória compartilhada, é necessário o uso de semáforos ou *mutexes*.

Conforme mencionou-se anteriormente, o próprio padrão MPI-2 já prevê a incorporação de primitivas que atuam sobre abstrações de memória compartilhada.

4. Aplicações

Vistas as diversas técnicas utilizadas na paralelização e distribuição de algoritmos, esta seção apresenta exemplos de aplicações matemáticas e científicas onde tais técnicas são comumente aplicadas.

4.1. Gerador de fractais de Mandelbrot

Os fractais de Mandelbrot [7] consistem em imagens abstratas geradas através de operações matemáticas aplicadas sucessivamente sobre os pontos de uma figura.

Formalmente, os fractais de Mandelbrot são calculados no plano dos números complexos, onde cada número $c = a + bi$ equivale, em termos computacionais, a um pixel de uma figura. Sendo c um ponto qualquer no plano complexo, iniciando-se por um valor $z_0 = 0 + 0i$ (ou seja, $z_0 = 0$), pode-se gerar uma seqüência de pontos, chamada de *órbita*, utilizando-se a seguinte fórmula

$$z_{n+1} = z_n^2 + c$$

Se algum ponto da órbita possui módulo (distância à origem) maior que 2.0, diz-se que a órbita "escapou". O *conjunto* de Mandelbrot é, então, definido como o conjunto de pontos cujas órbitas nunca escapam.

Quando da implementação do algoritmo de geração de fractais de Mandelbrot, entretanto, duas restrições se fazem necessárias. A primeira é que é necessário discretizar os pontos a serem calculados de modo a corresponderem a pixels de uma figura. A segunda é que, para que o algoritmo tenha um fim, deve-se estabelecer um limite para o número de iterações efetuadas sobre um mesmo ponto. Se a órbita não escapa até o número máximo de iterações ser alcançado, considera-se que o ponto faz parte do conjunto. Essas restrições fazem com que a imagem gerada seja apenas uma aproximação do conjunto real; a qualidade da figura é, assim, determinada pela resolução utilizada e pelo número máximo de iterações efetuadas sobre um ponto.

Para obter-se o efeito estético normalmente empregado para representar visualmente o fractal, define-se a cor de cada ponto da figura em função do número de iterações necessário para calculá-lo, repetindo as cores de forma circular, se necessário.

Dado o limite de tamanho 2.0 para o módulo de um ponto, a aplicação da fórmula sobre os pontos pertencentes à região compreendida entre os pontos (-2, -2) e (2, 2) é suficiente para se obter o conjunto de Mandelbrot completo. A Figura 3 ilustra esse conjunto utilizando uma paleta de 8 cores.

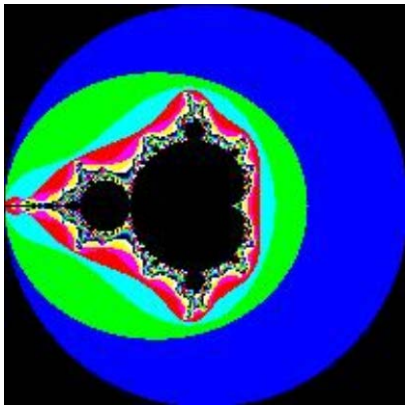

Figura 3: Conjunto completo de Mandelbrot.

É usual, entretanto, limitar o cálculo a uma sub-região dessa área, de modo a obter-se um efeito de *zoom* que proporciona a geração de figuras diferentes. Do ponto de vista computacional, esse procedimento altera o custo de processamento do algoritmo, pois diferentes pontos do fractal escapam em mais ou menos iterações, quando não exigem a execução do número máximo. A escolha de uma ou outra região e um maior ou menor efeito de *zoom* podem provocar mudanças significativas no tempo de processamento do algoritmo.

Feitas estas considerações, podem ser identificados três fatores que influem, primariamente, no custo de processamento de um fractal de Mandelbrot:

- as dimensões (ou resolução) da figura que se deseja obter pois, naturalmente, uma figura maior contém um número maior de pontos a serem calculados;
- a região do fractal a ser calculada;
- número de iterações sobre cada ponto.

Na prática, o número de iterações deve representar um equilíbrio entre o tempo de processamento e a qualidade da figura que se deseja obter, em função das dimensões e da região de cálculo. De forma geral, pontos mais internos ao fractal necessitam de um número maior de iterações para serem alcançados; portanto, um efeito de *zoom* muito grande implica no aumento do número de iterações necessário para que se obtenha uma imagem de qualidade razoável.

Algoritmo seqüencial O algoritmo seqüencial para geração do fractal de Mandelbrot, na forma mais intuitiva, consiste em realizar, para cada ponto da imagem, o número de iterações necessárias, definindo sua cor final. Para se ter uma idéia do custo de processamento do algoritmo, uma imagem de 400×400 pixels leva em torno de 19s para ser gerada em uma máquina Pentium II 266MHz, considerando-se o conjunto completo — de (-2, -2) a (2, 2) — e um número máximo de 500 iterações.

Algoritmo paralelo A idéia mais natural de paralelização do algoritmo de Mandelbrot é dividir a figura que se deseja calcular em sub-regiões que podem ser calculadas concorrentemente. Como não existe dependência entre o cálculo de um ponto e de outro, essa idéia pode ser implementada de maneira relativamente simples.

A simples divisão de um número pré-determinado de sub-regiões para cada processador, entretanto, não garante um bom desempenho para a aplicação; a variação de tempo de processamento entre os pontos de uma ou outra região pode rapidamente levar um processador a ficar ocioso, enquanto que outros tenham que calcular o número total de iterações.

Por esse motivo, torna-se mais eficiente dividir a figura em um número maior de regiões (ex. 20 vezes o número de processadores) e fazer os processadores calcularem-nas por demanda, requisitando novas regiões de cálculo à medida em que completam regiões recebidas anteriormente.

A implementação paralela traz, assim, outros dois fatores que influenciam no desempenho do algoritmo:

- número de processadores utilizados no cálculo;
- tamanho das sub-regiões nas quais a figura é dividida.

O segundo item é um fator que precisa ser cuidadosamente escolhido; as regiões não podem ser muito pequenas, pois levariam os processadores a dispenderem mais tempo com comunicação do que com o cálculo em si, e nem muito grandes, o que recairia na abordagem inicial. É comum tomar-se uma linha inteira de pixels como unidade básica das regiões de cálculo, sendo, portanto, o tamanho de uma região determinado pelo número de linhas que a compõe. Tomando-se o mesmo exemplo dado anteriormente (imagem de 400×400 pixels), obtém-se os melhores resultados com regiões compostas por 10 a 15 linhas.

Uma terceira abordagem, semelhante à primeira, também pode ser utilizada, produzindo bons resultados. Consiste em fazer os processadores calcularem a imagem de forma entrelaçada, de modo que o primeiro processador calcule a linha #0, o segundo a linha #1, e assim por diante. Ao terminar o cálculo, cada processador passa à próxima linha que corresponde à sua numeração; por exemplo, sendo n o número de processadores, o primeiro calcularia inicialmente a linha #0 e, na seqüência, as linhas #(0 + n), #(0 + n + n), etc. A grande vantagem desta abordagem é que, baseando-se somente em sua numeração, cada processador sabe exatamente quais linhas deve calcular, dispensando qualquer espécie de comunicação durante o cálculo. Uma única mensagem de cada processador é necessária, ao final do processo, para que os valores calculados possam ser reunidos de modo a compor a imagem final.

4.2. Difusão de calor pela equação de Laplace

Uma aplicação tradicionalmente empregada na implementação de métodos de paralelização é o cálculo de difusão de calor através da equação de Laplace [9], informalmente formulado como segue:

– Suponha uma placa retangular de metal, na qual a temperatura, em qualquer ponto de sua superfície, seja igual a 0º; em um instante t_0, uma fonte externa de calor é aplicada homogeneamente a um dos lados da placa, de modo que a temperatura nesse lado passa instantaneamente a 100º. Considerando que, a partir desse instante, a temperatura nas bordas da placa é mantida constante, o problema é determinar a temperatura em seus pontos internos.

Este problema enquadra-se na categoria dos problemas “de borda” e pode ser resolvido pela equação de Laplace:

$$\frac{\partial^2 \Theta}{\partial x^2} + \frac{\partial^2 \Theta}{\partial y^2} = 0$$

Para ser resolvida algoritmicamente, a equação tem que ser convertida em um método iterativo, o qual deve ser processado até convergir à solução do problema. Para isso, o problema deve ser adaptado a uma abordagem computacional.

O primeiro passo é discretizar a superficie da placa em um conjunto de pontos, formando uma grade (ou *mesh*, como o problema é geralmente caracterizado), a qual é representada por uma matriz $\Theta(n,n)$, onde n é o número de divisões em cada linha/coluna. Naturalmente, quanto maior esse número, mais próximo se chega da solução verdadeira, porém maior o custo de processamento do algoritmo.

Uma vez discretizado o problema, pode-se aplicar um método iterativo. Dois métodos comumente usados, e de implementação relativamente simples, são os de *Jacobi* e de *Gauss-Seidel*. Em ambos, a temperatura de um determinado ponto da grade, $\Theta_{x,y}$, é determinada em função de seus vizinhos. Pelo método de Jacobi, cada ponto é determinado por

$$\Theta^{i+1}_{x,y} = \frac{1}{4}(\Theta^{i}_{x-1,y} + \Theta^{i}_{x,y-1} + \Theta^{i}_{x+1,y} + \Theta^{i}_{x,y+1})$$

ou seja, pela média dos quatro pontos adjacentes. Note-se que, para uma matriz $n \times n$, x e y variam de 2 a $n - 1$, pois os valores das bordas devem ser mantidos constantes.

Pelo método de Jacobi, o cálculo de um ponto na iteração $i + 1$ utiliza os valores dos vizinhos na iteração i, ou seja, de um passo anterior. Para se obter uma convergência mais rápida, pode-se usar o método de Gauss-Seidel, que utiliza dois valores da iteração corrente:

$$\Theta^{i+1}_{x,y} = \frac{1}{4}(\Theta^{i+1}_{x-1,y} + \Theta^{i+1}_{x,y-1} + \Theta^{i}_{x+1,y} + \Theta^{i}_{x,y+1})$$

Algoritmo seqüencial Assim como o gerador de fractais de Mandelbrot, a implementação seqüencial da equação de Laplace consiste em percorrer todos os pontos um a um, repetindo esse procedimento várias vezes até que a diferença entre o valor de um ponto em duas iterações subseqüentes seja inferior a um limite de erro pré-estabelecido (ex. 10^{-3}). Empiricamente, pode-se de imediato perceber que este algoritmo apresenta alto custo computacional.

Algoritmo paralelo A solução paralela para o algoritmo da equação de Laplace, ao contrário do gerador de fractais, pode ser eficientemente implementada utilizando-se a solução mais intuitiva: dividir a matriz Θ pelo número de processadores disponíveis. Isso porque, neste caso, não há diferença entre o custo de processamento de uma ou outra região da matriz.

A resolução paralela da equação de Laplace, entretanto, apresenta outra particularidade que influi diretamente no desempenho do algoritmo: a dependência entre os valores de pontos adjacentes. Quando a matriz é dividida entre vários processadores, os valores dos pontos nas bordas precisam ser comunicados aos processadores vizinhos a cada iteração, para que as equações sejam plenamente satisfeitas. Dependendo do tamanho de cada região, esse procedimento pode acarretar em excesso de comunicação, mascarando todo o ganho em desempenho conseguido pelo paralelismo do cálculo.

Nota-se que o problema é semelhante ao que ocorria no gerador de fractais, sendo causado pelo tamanho das regiões de cálculo. Como este é, em última análise, determinado pelo número de processadores disponíveis, pode ser necessário utilizar um número menor de processadores para obter-se o melhor desempenho.

4.3. Resolução da equação *Shallow water*

As duas aplicações apresentadas anteriormente são comumente utilizadas, na área de paralelismo, para a exemplificação de algoritmos paralelos. A aplicação apresentada aqui, resolução da equação Shallow Water, é um exemplo mais ligado ao mundo real. Essa equação [3] é utilizada para modelar a dinâmica de águas em meios naturais como rios e córregos. Os grupos de Matemática na Computação e Processamento Paralelo e Distribuído do Instituto de Informática da UFRGS desenvolvem uma aplicação de modelagem das águas do Rio Guaíba, o qual banha a cidade de Porto Alegre, para efeito de estudos de poluição.

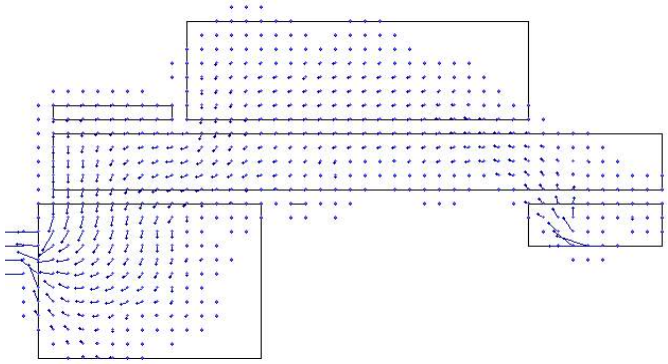

Figura 4: Modelagem das águas do Rio Guaíba.

De forma semelhante ao problema da equação de Laplace, a área de estudos banhada pelo rio é discretizada e modelada através de um conjunto de matrizes, necessárias para abranger-se a forma irregular da região. A Figura 4 mostra o particionamento adotado. Percebe-se nos lados esquerdo e direito da figura os pontos de saída e entrada de águas, respectivamente.

Uma vez definidas as matrizes e estabelecidos os valores iniciais, os pontos internos são calculados de forma iterativa, podendo convergir para a solução ou não, caso em que o algoritmo é interrompido por força de um limite máximo de iterações. Como se pode prever, as matrizes devem comunicar-se nas bordas, trocando dados mais novos a cada iteração.

O algoritmo apresenta ainda, além das matrizes que modelam o fluxo das águas, uma série de vetores que contém dados relativos à aplicação como um todo; tais elementos são mantidos de forma distribuída no algoritmo, e o acesso a eles deve ser feito de forma sincronizada.

5. Considerações finais

Após esta breve introdução aos principais conceitos ligados à execução de aplicações em ambientes concorrentes, percebe-se a diversidade de métodos existentes para atingir esse fim. Comparada à programação seqüencial, a programação paralela apresenta uma série de novos detalhes que, somados à diversidade recém mencionada, podem agir como inibidores aos iniciantes nesta área. Entretanto, uma vez que se absorvam os principais fundamentos do processamento paralelo, o vislumbramento de soluções paralelas para aplicações matemáticas e científicas passa a surgir de forma natural, de modo que, por vezes, a solução pode mesmo parecer mais simples do que a seqüencial. Em contribuição a esse fato, grande parte das pesquisas em processamento paralelo e distribuído da atualidade são direcionadas a reduzir o impacto inicial dos estudos nessa área, tendo gerado inúmeros modelos e extensões de linguagens de programação que procuram embutir o paralelismo em mecanismos e estruturas já conhecidos. Todos esses motivos levam a crer que é cada vez mais significativa a importância do processamento paralelo e distribuído na Ciência da Computação, e sua presença é quase que imprescindível no mundo atual.

Referências bibliográficas

[1] AMZA, C. et al. TreadMarks: shared memory computing on networks of workstations. **IEEE Computer**, v.29, n.2, 1996. p.18-28.

[2] BODEN, N. et al. Myrinet: a gigabit-per-second local-area network. **IEEE Micro**, v.15, n.1, 1995. p.29-36.

[3] CASULLI, V. Semi-implicit finite difference methods for the two-dimensional shallow water equations. **Journal of Computational Physics**, v.86, 1990. p.56-74

[4] GEIST, Al et al. **PVM**: Parallel Virtual Machine. Cambridge: MIT Press, 1994. 279p.

[5] HWANG, Kai & ZHIWEI, Xu. **Scalable parallel computing**: technology, architecture, programming. Boston: Wcb- Mc Graw-Hill, 1998. 802p.

[6] IEEE. IEEE Standard for Scalable Coherent Interface (SCI), IEEE 1596-1992, 1992.

[7] MANDELBROT, Benoit. **The fractal geometry of nature**. New York: W. E. Freeman and Company, 1982.

[8] MPI FORUM. **The MPI message passing interface standard**. Knoxville: University of Tenessee, 1994. Relatório técnico.

[9] PRESS, W. H et al. **Numerical recipes in C++**: the art of scientific computing. New York: Cambridge University Press, 1988. 735p.