# 1
# Princípios da Programação Concorrente

**Gerson Geraldo Homrich Cavalheiro**[1] *(UNISINOS, gersonc@exatas.unisinos.br)*

**Resumo:**

Neste capítulo é abordada a programação em ambientes dotados de múltiplos recursos de processamento. O objetivo é oferecer ao leitor uma introdução às diferentes questões envolvidas nesta tarefa. Na primeira parte do texto são apresentadas uma visão geral de modelos de programação concorrente encontrados na bibliografia e noções para composição de programas concorrentes em termos de tarefas e manipulação de informações. A seqüência possui uma abordagem mais prática, enfocando a problemática da programação em aglomerados de computadores e duas populares ferramentas de programação: *threads* POSIX e Message Passing Interface (MPI).

[1]Doutor em Informática: Sistemas e Comunicações pelo Instituto Nacional Politécnico de Grenoble, Franç a, Mestre em Cência da Computação pela Universidade Federal do Rio Grande do Sul. Professor no Programa Interdisciplinar de Pós-Graduação em Computação Aplicada da Universidade do Vale do Rio dos Sinos.

# 1.1. Introdução

Como o capítulo anterior apresentou, existem diversas arquiteturas de computadores possibilitando a programação paralela. Diferentes técnicas são oferecidas aos programadores para explorá-las, permitindo a estes implementar programas capazes de executar atividades simultâneas sobre arquiteturas com várias unidades de processamento. A questão que se coloca neste momento, e que é abordada neste capítulo, é como obter programas concorrentes para estas arquiteturas.

Neste capítulo é apresentada uma visão geral de técnicas de construção de programas concorrentes. Especial atenção será dada aos métodos e ferramentas utilizadas para exploração de aglomerados de computadores. Para tanto, o texto encontra-se organizado como segue.

A seção 1.2. apresenta, de forma bastante resumida, uma visão geral de diferentes modelos de programação concorrente. É tecido um paralelo da aplicação destes modelos a diferentes arquiteturas de máquinas paralelas.

Na seqüência, a seção 1.3. define, no contexto do estudo conduzido, o significado da concorrência e de atividades concorrentes. É discutida também a forma como a informação é manipulada por estas atividades em um programa em execução.

Uma abordagem prática dos conceitos é apresentada a partir da seção 4, onde são discutidas questões envolvidas na programação em aglomerados de computadores. A seção 1.5. discute a programação concorrente em arquiteturas com memória compartilhada, utilizando como base para discussão a ferramenta de programação baseadas no padrão *threads* POSIX. A programação em arquiteturas com memória distribuída, através de uma biblioteca de comunicação MPI, é discutida na seção 1.6..

# 1.2. Modelos para Programação Concorrente

Modelos são comumente propostos em diversas áreas das ciências para descrever um item da realidade. A razão de ser de um modelo é permitir uma visão sistemática e, não raramente simplificada, do objeto modelado. Alguns modelos de programação concorrente são discutidos na presente seção, o texto no entanto não tem a intenção de esgotar o assunto. Ao leitor interessado em aprofundar estes assuntos, recomenda-se a consulta do livro escrito por C. Leopold [25], a coletânea editada por Zomaya [41] e o também o texto de A. Goldman [22].

## 1.2.1. Níveis de granulosidade de concorrência

A concorrência em um programa em execução pode ser encontrada em diferentes *níveis de granulosidade*. O termo granulosidade diz respeito à razão entre o tempo necessário ao cálculo de uma determinada operação e os custos envolvidos nas trocas de dados entre esta operação e as demais operações de programa em execução. Em linhas gerais, a granulosidade de um programa pode ser relacionada ao tamanho médio das operações envolvidas neste programa. Assim, um programa pode ser classificado em função de sua granulosidade: *grossa, média, fina* e todas as variantes possíveis (*muito-fina, muito-grossa*, ...).

A Tabela 1.1 apresenta diferentes níveis de granulosidade que podem ser explorados em arquiteturas de computadores. Nesta tabela encontram-se identificadas, para cada nível, arquiteturas típicas para o suporte a execução a cada classe de granulosidade.

**Tabela 1.1: Níveis de granulosidade oferecidos por diferentes arquiteturas de computadores.**

| Granulosidade | Classe | Ocorrência |
|---|---|---|
| Muito fina | Intra-instrução | Processadores VLIW e super-escalares |
| Fina | Entre instruções | Processadores vetoriais |
| Fina/Média | Blocos | UMA |
| Média | Procedimentos | UMA/NUMA |
| Grossa | Processos | SC-NUMA/NORM |
| Muito grossa | Aplicações | Qualquer arquitetura |

Neste texto, não está sendo considerada a categoria de granulosidade *muito grossa*, uma vez que não representa de fato um problema ligado a programação propriamente dito. Nos demais casos, é importante que o programador observe os níveis de granulosidade oferecidos pela arquitetura para qual está sendo desenvolvido um programa. A aplicação sistemática de técnicas para explorá-los permitirá obter um programa com bom desempenho. Isto se aplica mesmo quando existam mecanismos, sejam estes de hardware ou de um compilador paralelizante, capazes de extrair a concorrência de forma automática. Este caso é comum na exploração da granulosidade fina ou muito fina.

Os casos mais típicos onde ocorre exploração automática de concorrência são arquiteturas VLIW e superescalares. Nas arquiteturas VLIW (*Very Large Instruction Word*), a palavra descrevendo uma instrução é grande o suficiente para conter várias operações a serem executadas por unidades distintas do processador – a concorrência existe internamente a instrução pela execução simultânea destas operações. No caso das arquiteturas super-escalares a idéia é viabilizar a execução de um conjunto de $n$ instruções de forma concorrente pela replicação em $n$ conjuntos de unidades do processador e ativação destas instruções sobre cada um destes conjuntos. Nestes casos a mão do programador está na seleção correta das instruções (VLIW) ou na definição de seqüências de instruções que explorem de forma eficiente os recursos duplicados no processador (superescalares), evitando dependência entre as instruções.

Menos sutis para exploração automática, a concorrência identificada nos níveis médio ou grosso deve ser explicitadas dentro de um programa. Nestes níveis se colocam as questões abordadas na seqüência deste texto, uma vez que dizem respeito a arquiteturas dotadas de vários processadores. As decisões tomadas para exploração da concorrência média e/ou grossa têm impacto direto no desempenho final obtido pela execução de um programa.

### 1.2.2. Classificações de modelos de programação concorrente

Em se tratando de modelo de programação para arquiteturas seqüenciais, a proposta de von Neumann pode ser considerada praticamente a única, sendo largamente utilizada por toda a comunidade associada aos computadores. Em se tratando de arquiteturas

dotadas de vários processadores, o número de modelos propostos é maior. Uma justificativa simples a este fato: arquiteturas dotadas de vários processadores podem assumir diversas configurações.

Dentre as classificações, talvez a mais conhecida seja a classificação de Flynn [18]. Esta classificação é baseada em fluxos de execução e fluxos de dados, os quais podem ser simples ou múltiplos. Assim, são definidas quatro classes de modelos: SISD (um único fluxo de instrução, um único fluxo de dados), SIMD (um único fluxo de instrução, múltiplos fluxos de dados), MISD (múltiplos fluxos de instrução, um único fluxo de dados) e MIMD (múltiplos fluxos de instrução, um único fluxo de dados). Note que este modelo esta comumente associado à descrição de arquiteturas de computadores, mas que se aplica claramente a modelos de programação.

Em uma clara extensão ao modelo de Flynn, surgem os termos SPMD e MPMD, respectivamente um único programa, múltiplos dados e múltiplos programas múltiplos dados. Esta extensão leva a estruturas de execução onde existe *i* um único código sendo executado por vários processadores e *ii* vários códigos distintos sendo executados por vários processadores. Em ambos os casos manipulando conjunto de dados distintos.

Já o modelo de El-Rewini and Lewis [17] é bidimensional, sendo baseado no grau de acoplamento provido pela arquitetura (de fortemente acoplada à base de dados compartilhadas) e pela granulosidade da aplicação (fina, media, grossa).

Outra classificação bastante conhecida é a de Skillicorn e Talia [34], a qual refere-se exclusivamente a questões de programação. Como na proposta anterior, são utilizadas duas dimensões para classificar os diferentes modelos. A primeira diz respeito ao grau de abstração que o programador pode contar no desenvolvimento de seu código concorrente. Esta primeira dimensão possui as seguintes possibilidades:

- Modelos com paralelismo implícito;
- Modelos com paralelismo explícito, mas decomposição implícita;
- Modelos com decomposição explícita, mas com mapeamento implícito;
- Modelos com mapeamento implícito, mas com comunicações explícitas;
- Modelos com comunicação explícita, mas com sincronizações implícitas; e,
- Modelos onde o programador é responsável por todas as tarefas.

Como é possível ver, a primeira dimensão distingue os modelos de programação concorrente em função do grau de abstração que o programador pode ter no desenvolvimento de seu código. Já a segunda dimensão caracteriza os modelos em função das comunicações realizadas.

- Número dinâmico de processos envolvidos no cálculo e volume de comunicação ilimitado;
- Número fixo de processos e volume de comunicação ilimitado; e,
- Número fixo de processos e volume de comunicação limitado.

Outros esquemas para classificação ainda podem ser encontrados na bibliografia. Este texto limitou-se a apresentar alguns poucos.

## 1.2.3. Modelos de programação

Esta seção apresenta alguns dos modelos utilizados para desenvolvimento de programas concorrentes. Estes modelos são na verdade descritos segundo *o ponto de vista do programador* [25]. Isto quer dizer que estão de fato associados ao uso prático no desenvolvimento de aplicações. Dada a esta característica, não é de se surpreender caso um mesmo modelo seja encontrado em diferentes ambientes e/ou linguagens – ao contrário, esta situação é bastante comum.

### 1.2.3.1. Paralelismo de dados *vs.* Paralelismo de tarefa

A primeira distinção entre modelos que é apresentada diz respeito à identificação do elemento delimitador da concorrência em uma aplicação. Uma vez identificado este elemento, o programa concorrente pode ser projetado.

*Paralelismo de tarefa.* O primeiro modelo apresentado trata do paralelismo de execução de tarefa. Este modelo tem como característica a execução paralela de diferentes atividades sobre conjuntos distintos de dados. Neste caso, o programador dedica seu esforço em identificar as atividades concorrentes da aplicação e deve ser observada a distribuição destas tarefas nos recursos de hardware disponíveis. Embora este modelo possa parecer um tanto óbvio, ele contrasta com o modelo de paralelismo de dados.

*Paralelismo de dados.* Com uma abordagem completamente ortogonal ao modelo de paralelismo de tarefa, o modelo de paralelismo de dados é caracterizado pela execução paralela de uma mesma atividade sobre diferentes partes de uma mesma coleção de dados. Neste caso, são os dados que determinam a concorrência da aplicação e a forma como o cálculo deve ser distribuído na arquitetura.

### 1.2.3.2. Memória compartilhada *vs.* troca de mensagens

Uma vez de posse do projeto de aplicação concorrente, cabe ao programador selecionar o conjunto de recursos de programação que irão suportar sua implementação. Embora diversos modelos possam ser empregados [25, 41], tais cliente/servidor, computação em grade, orientação a objetos, os mais clássicos são compartilhamento de memória e trocada de mensagens.

*Compartilhamento de memória.* Neste modelo, as tarefas em execução compartilham um mesmo espaço de memória. A comunicação entre estas pode ser efetivada através do simples acesso a esta memória. Variáveis compartilhadas são portanto o meio de comunicação utilizado.

*Troca de mensagens.* Caso não exista um espaço de endereçamento comum, a opção para efetivar a troca de dados entre tarefas recai em utilizar troca de mensagens de forma explicita. Ou seja, uma rede de interconexão é explorada para o envio e recebimento de mensagens.

## 1.2.4. Modelos abstratos

Enquanto os modelos de programação são voltados a auxiliar o programador a desenvolver seu código, os modelos abstratos têm por objetivo oferecer dados para melhor compreensão dos algoritmos concorrentes e tecer previsões sobre desempenho. Estes modelos são também conhecidos como *bridge models*, pois tentam aproximar a descrição

de hardware paralelo para auxiliar no projeto de software concorrente. Observe-se, no entanto, que até o momento não foi encontrado um modelo *bridge* aceito universalmente.

Nesta seção são descritos, brevemente, três dos mais populares modelos: PRAM, LogP e BSP. O conteúdo desta seção pode ser estendido com consulta à [25, 22, 41].

*PRAM.* A sigla PRAM denota *Parallel Random Access Machine*. O modelo prevê a execução de programas concorrentes em uma máquina paralela composta por um número não limitado de processadores compartilhando um espaço de endereçamento comum. Um programa executado sobre uma arquitetura PRAM tem suas instruções sincronizadas, isto é, o avanço da execução é cadenciado pela execução das instruções de forma coordenada. Variantes deste modelo foram propostas com vistas a torná-lo mais realista. Entre estas variantes encontram-se as que introduzem limitações no acesso concorrente a memória e no número de processadores disponíveis.

*BSP.* O modelo BSP *Bulk-Synchronous Parallel* pode ser considerado uma extensão do modelo PRAM, utilizando múltiplos fluxos de execução executando em processadores interconectados por uma em rede. A característica principal deste modelo é a execução, pelos fluxos de execução, de *super-steps*: cada *super-step* consiste em uma fase de cálculo, na qual todos os processadores efetuam suas operações (as quais não têm necessariamente a mesma duração) seguida de uma fase de comunicação, podendo esta também ser executada por todos os fluxos. Um *super-step* termina com uma sincronização entre todos os fluxos.

*LogP.* O modelo LogP (Latência, *Overhead*, *Gap* entre as comunicações e Processadores) descreve arquiteturas paralelas em função de determinados parâmetros, os quais constam no próprio nome do modelo. Com LogP é possível realizar análises mais próximas a que serão obtidas de fato no comportamento de aplicações, uma vez que são considerados dados mensuráveis em arquiteturas reais. Embora possibilite dados de desempenho mais consistente, LogP não é adequado enquanto modelo de programação.

## 1.3. A Programação Concorrente

O termo concorrência é comumente aplicado para definir sistemas onde é possível compartilhar o uso dos recursos de processamento de um computador – tipicamente o processador – para suportar o fluxo de execução de dois ou mais programas independentes [36] [33]. Nestes casos, a concorrência visa obter ganho de desempenho no sistema e/ou prover um ambiente *time-sharing*. Um ambiente *time-sharing* fornece um mecanismo de compartilhamento dos recursos de processamento de um computador entre processos submetidos por diferentes usuários, desta forma, cada usuário tem o sentimento de que possui um computador próprio para execução de seu programa. Já o uso da programação concorrente para ganho de desempenho, vem de uma constatação de que enquanto um programa realiza operações de E/S, o processador permanece inativo. Considerando que os programas são independentes entre si, a concorrência é empregada para não desperdiçar o tempo de uso da unidade de processamento: assim, quando um programa necessita realizar uma operação de E/S, um novo programa é selecionado para ocupar o processador e avançar na execução de suas instruções.

Seguindo o modelo de von Neumann, a arquitetura dos computadores consiste em uma máquina de cálculo que possui um processador, uma área de memória para dados e código executável e de dispositivos de E/S. Este modelo prevê a execução seqüencial das

instruções de um programa na ordem especificada pelo programa fonte dentro de um fluxo de execução – implicitamente existe um controle que não permite que uma instrução execute antes que a anterior tenha terminado. Já programas concorrentes possuem diversos fluxos de execução, executando cada um uma seqüência própria de instruções. Quando em execução sobre uma arquitetura mono-processada não existe nenhum controle implícito das relações de ordem que possa existir para a execução das instruções entre os diferentes fluxos. Além disto, os fluxos competem pelo tempo de ocupação do processador e dividindo entre si o espaço de memória disponível e o acesso aos dispositivos de E/S.

Assim temos a primeira definição para concorrência: disputa. Esta definição é retirada do fato de existir um conjunto de fluxos de execução independentes que concorrem pelo uso dos recursos de arquitetura para suprir as necessidades de execução de suas instruções. Esta disputa reflete-se no compartilhamento do tempo de uso do processador e de espaço de armazenamento na memória.

A abordagem adotada pela programação concorrente [40] possui uma diferença importante: neste caso, busca-se a implementação de uma única aplicação, utilizando-se do recurso de decompô-la em diversos fluxos de execução. Cada um destes fluxos é responsável pela execução de uma parte da solução do problema, o que implica que eles não sejam verdadeiramente independentes. Sendo cada fluxo de execução responsável por encontrar uma parte da solução do problema, é necessário que, ao ser obtido um resultado por um fluxo, este resultado possa ser comunicado a outro fluxo de forma a compor a solução final, ou seja, a resposta a ser apresentada pelo programa.

Assim agrega-se uma nova característica ao termo concorrência: colaboração. Muito embora os fluxos estejam competindo por recursos, estes se encontram interagindo de forma a evoluir a execução do programa no sentido de encontrar a solução final. Esta interação é manifesta na troca de informações entre os diferentes fluxos.

Desta forma, programar de forma concorrente implica detectar na aplicação as atividades que não possuem restrições temporais e os pontos onde restrições temporais existem. Ou seja, para uma aplicação desejada, identificar as seqüências de instruções que podem ser executadas de forma independente uma das outras e quais são as relações de produção e consumo de dados entre estas seqüências.

O termo concorrência passa assim a agregar um novo significado, aplicando-se a fluxos de execução onde determinados trechos de instruções não possuem restrições temporais de execução – independentemente do fato de compartilharem ou competirem por recursos de execução –, intercalados por trocas de dados entre estes fluxos. Esta forma de programação permite que diferentes fluxos de execução colaborem entre si, empregando mecanismos de sincronização [32], para atingir um objetivo comum: o resultado esperado.

Um programa concorrente é, desta forma, composto por um conjunto de fluxos de execução que, de alguma forma ordenada, trocam informações. Os fluxos de execução consistem no suporte à execução das tarefas definidas na aplicação e as sincronizações oferecem os mecanismos sobre os quais são implementadas as trocas de dados entre estas tarefas. Um programa concorrente será corretamente executado se todas tarefas definidas forem executadas respeitando as sincronizações definidas entre elas.

### 1.3.1. Concorrência *vs.* paralelismo *vs.* distribuição

A atenção desta seção encontra-se toda voltada à concorrência. Cabe neste momento distinguir a aplicação dos termos concorrência, paralelismo e distribuição na progra-

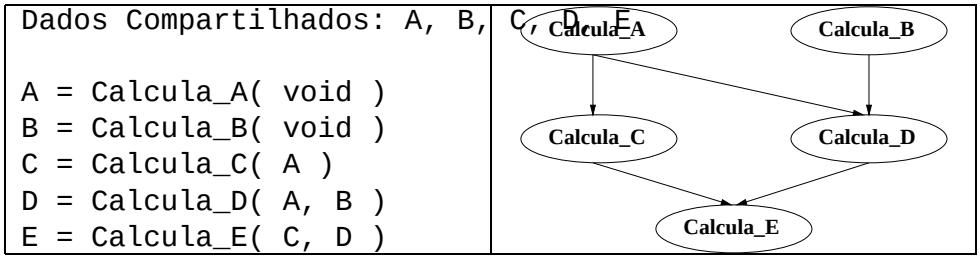


**Figura 1.1: Grafo gerado pela execução de um algoritmo concorrente..**

mação segundo os critérios deste texto.

A *programação concorrente* é aquela onde as atividades envolvidas na resolução de um programa encontram-se decompostas em atividades independentes e as relações de trocas de dados encontram-se igualmente identificadas. Em um *programa paralelo*, além das atividades e das relações de trocas de dados estarem identificadas, estas precisam ser, de fato, efetivadas com vistas a explorar eficientemente recursos de uma arquitetura [31].

A *programação distribuída* é outro caso particular de concorrência. A particularidade é que as atividades de um programa encontram-se em execução sobre arquiteturas que não possuem memória comum.

Assim, pode-se dizer que a programação concorrente é a forma mais genérica de especificar programas para o processamento de alto desempenho. Tanto a programação paralela como a distribuída consistem em especializações desta, em termos dos recursos disponíveis para execução, muito embora o termo paralelismo esteja fortemente associado ao processamento de alto desempenho.

### 1.3.2. Tarefa e sincronização em programas concorrentes

Em um programa concorrente, uma tarefa é definida por uma atividade da aplicação capaz de produzir parte da solução do problema. Ela é composta por conjunto de instruções que devem ser executadas de forma seqüencial sob um fluxo de execução. Uma tarefa necessita de um conjunto de parâmetros de entrada para poder executar e, ao seu final, produz um conjunto de dados de saída. Com os parâmetros de entrada e os dados de saídas das tarefas especificados, ficam definidas as relações de cooperação para troca de dados entre as tarefas em execução. Durante a execução de um programa, duas tarefas são consideradas independentes quando a saída de uma não for entrada de outra, não havendo, portanto, restrições temporais para execução entre elas. Caso contrário, existe uma comunicação: as duas tarefas devem então ser executadas de forma síncrona, garantindo a correta comunicação entre elas.

O leitor deve estar atento ao fato que, se uma tarefa produz como resultado um dado que é tomando como entrada por uma outra tarefa, existe uma relação de produção entre as duas tarefas. A ordem de execução deverá, portanto ser respeitada.

A sincronização é o mecanismo que permite o controlar a evolução da execução de um programa. Através da sincronização do acesso das tarefas aos dados gerados por outras tarefas é realizada a coordenação das trocas de dados (comunicações) entre as tarefas.

Tomando em consideração as dependências entre as tarefas, um programa concorrente pode ser representado através de um grafo dirigido, onde cada vértice representa

uma tarefa e uma aresta dirigida, uma sincronização. O algoritmo de um programa concorrente e o grafo gerado pela sua execução são apresentados na Figura 1.1. Uma análise deste grafo permite concluir verificar que as tarefas possui um ciclo de vida composto por quatro estados [20]:

- **Aguardando:** a tarefa foi definida, porém os dados de entrada não encontram-se disponíveis. É o caso da tarefa `Calcula_C`, antes do término da execução da tarefa `Calcula_A`.
- **Pronta:** a tarefa pode ser executada, pois os dados que necessita para ser executada encontram-se disponíveis. Será o caso, no exemplo, da tarefa `Calcula_C` após o término da execução da tarefa `Calcula_A`.
- **Executando:** quando uma tarefa é atribuída a um fluxo de execução, tendo suas instruções executadas seqüencialmente.
- **Concluída:** uma tarefa é considerada concluída ao ter produzido seus dados de saída, liberando o fluxo de execução ao qual estava associada.

Os mecanismos de sincronização empregados para realizar as comunicações de dados permitem controlar a evolução no ciclo de vida das tarefas, garantido a execução segundo a ordem definida pelo programa.

#### 1.3.2.1. Modelo de programa concorrente

Na abordagem realizada neste texto, o caso prático da programação implica que o programador deva, explicitamente, identificar no seu programa as atividades concorrentes e as sincronizações de controle de troca de dados. Para auxiliar a compreensão, é introduzido um modelo oferecendo uma abstração para um programa concorrente. Antes de apresentar o modelo de programa concorrente, é apresentada a abstração considerada para programas seqüenciais.

O modelo de programa seqüencial parte do modelo de von Neumann. Um programa seqüencial $\mathcal{A}$ é lançado a execução com um conjunto de dados $X$, resultando em uma execução representada por $\mathcal{A}(X)$. Esta execução é resultado ativação de um conjunto $\mathcal{I}$ de $n$ instruções, $\mathcal{I} = \{\iota_1, \iota_2, ..., \iota_n\}$, de forma que a execução de uma instrução $\iota_i, 1 < i \leq n$ produz um dado de saída[2]. O mecanismo de sincronização é implícito, onde uma instrução $\iota_i$ precede, no tempo, a execução de uma instrução $\iota_{i+1}$, ou seja, $\iota_i \prec \iota_{i+1}$. Esta relação de precedência é explicitada pelo programador no momento em que dispõe as instruções no programa, no entanto o controle é realizado de forma implícita: o próprio modelo de execução de programas seqüenciais garante que uma instrução $\iota_{i+1}$ não vai ser executada antes que uma instrução $\iota_i$ termine.

No modelo de programa concorrente adotado, as premissas adotadas são bastante próximas a este modelo seqüencial. Seja $\mathcal{G}(X)$ a descrição de um programa concorrente. Este programa é composto por um conjunto de tarefas $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$ e por um conjunto de dados $\mathcal{X} = \{x_1, x_2, ..., x_n\}$. Portanto, $\mathcal{G}(X) = \mathcal{T} \cup \mathcal{X}$. Uma tarefa $\tau_i$ é descrita por uma função $x_i = \mathcal{F}(x_j), 1 \leq j < i$, que tem como entrada um conjunto de dados $x_i$ e produz como resultado um dado $x_j$.

[2]No caso da execução seqüencial, esta saí daé representada pela alteração do conteúdo de uma posição de memória.

Nesta representação sob a forma de grafo de fluxo de dados ([24, 20]), fica explicita a existência de relações de dependência entre tarefas. Esta relação é representada por $\tau_j \prec \tau_i$, significando que a tarefa $\tau_j$ deve terminar *antes* que a tarefa $\tau_i$ seja lançada, pois $\tau_j$ produz como resultado um parâmetro atendido como parâmetro de entrada pela tarefa $\tau_i$.

A prática da programação vai se traduzir, portanto, na definição das tarefas e da sua ativação para execução de forma concorrente, na identificação dos dados trocados entre estas e na introdução de mecanismos de sincronização. Estes aspectos explicitam, respectivamente, o grau de concorrência da aplicação e sua granulosidade, a comunicação entre as atividades concorrentes e pontos onde a concorrência deva ser limitada. Todas as ferramentas de programação concorrente oferecem recursos para realização destas operações. A seqüência do texto apresenta como estes aspectos são tratados, considerando tanto arquiteturas com memória compartilhada e como as com memória distribuída.

## 1.4. Exploração da concorrência em aglomerados

Como visto anteriormente, na prática da programação concorrente, o termo concorrência é aplicado a diferentes níveis na execução de uma aplicação e, de acordo com os recursos de hardware disponíveis, ela é expressa de uma forma diferente. Sendo tomado um aglomerado de computadores como arquitetura de suporte a execução de uma aplicação, os níveis de concorrência que podem ser explorados são: a concorrência intra-nó e a concorrência entre nodos. A principal diferença entre estes está na forma de como os fluxos de execução interagem, ou seja, como os resultados das atividades são comunicados de uma a outra. Ora, esta interação é fortemente baseada na arquitetura de hardware disponível nos dois casos: em a concorrência intra-nó explora o fato de diferentes fluxos de execução compartilharem uma mesma área de endereçamento, enquanto na entre nodos o recurso para interação é tipicamente baseado em trocas de mensagens.

### 1.4.1. Interação em arquiteturas com memória compartilhada

A concorrência intra-nó é definida por um conjunto de atividades concorrentes executadas em um mesmo nó de um aglomerado (ou de um computador independente). A concorrência intra-nó pode ainda ser real ou temporal dependendo do número de fluxos de execução ativos em um determinado momento e do número de recursos de processamento – processadores – disponíveis no nó. Caso haja um número de processadores pelo menos igual ao de fluxos de execução ativos, existe a concorrência real, também denominado paralelismo. Caso contrário, um número de fluxos de execução ativos em um determinado instante de tempo maior que o número de processadores disponíveis, existe a concorrência clássica, onde há o compartilhamento de recursos.

A forma de interação mais natural, e menos onerosa, para o compartilhamento de informações nesta forma de concorrência é através do espaço de armazenamento provido pela memória do próprio nó. O mecanismo de comunicação explora o fato de que os dados manipulados pelas instruções dos diferentes fluxos de execução são variáveis armazenadas em memória cujo espaço de endereçamento é compartilhado. O acesso a estas variáveis se dá através de triviais operações de leitura e escrita (tipo *load* e *store*). Sendo a memória um recurso pertencente ao nó e compartilhado pelos fluxos em execução, um

dado escrito por uma instrução em um fluxo de execução pode ser lido por uma instrução em outro fluxo.

Mesmo que a comunicação entre fluxos de execução se apóie em instruções de leitura e escrita, portanto idênticas às utilizadas em programas seqüenciais, deve ser tomado cuidados adicionais para garantir a sincronização ao acesso a dados compartilhados.

Em uma execução seqüencial, o resultado de uma instrução é comunicado a outra instrução através de uma escrita em memória, que reflete o efeito colateral da execução da instrução: a própria alteração do estado do dado na memória, que no futuro servirá de entrada para uma outra instrução. A comunicação entre duas instruções se dá através do compartilhamento de certas posições de memória e a sincronização entre duas instruções é realizada implicitamente, através da garantia que uma instrução será executada somente após o término da instrução que a precede. Com isto garante-se que as alterações de estado da memória foram realizadas e que todo dado lido pela instrução em execução está atualizado.

Em uma execução concorrente, o sincronismo implícito entre instruções pertencentes a diferentes fluxos de execuções é inexistente. Portanto, toda instrução em um fluxo de execução que acesse um dado que também possa ser acessado por uma outra instrução em um outro fluxo consiste em uma *instrução crítica*. Na verdade, são instruções críticas todas as instruções, sobre quaisquer dos fluxos de execução, que possam vir a acessar uma mesma posição de memória compartilhada.

Mesmo que uma instrução crítica possa acessar um dado compartilhado da mesma forma que uma instrução executada em um fluxo de execução de execução seqüencial, um mecanismo externo deve prover a sincronização entre a instrução que produz o valor para o dado compartilhado e a instrução que necessita deste dado como entrada para sua própria execução. O correto emprego da sincronização é a única garantia de que a comunicação entre as tarefas vai ser realizada como esperado.

Nos diversos mecanismos de sincronização, o princípio das técnicas empregadas é o mesmo: um grupo de instruções em um fluxo de execução que manipulam dados em memória que servem para comunicar informações entre dois ou mais fluxos consiste em uma *seção crítica*. Os mecanismos mais utilizados são os que garantem exclusão mútua no acesso a memória (mutexes) e os que realizam controle no avanço de execução tais criação de novos fluxos de execução e bloqueio aguardando um o término da execução de um fluxo (*create* e *joins*).

As ferramentas que possibilitam a exploração da concorrência intra-nó mais clássicas baseiam-se no padrão POSIX [1] para processos leves (ou *threads*). *Threads* POSIX são disponíveis em diferentes sistemas, tais como Linux, AIX, Minix, Solaris e mesmo na família Windows. Algumas características das ferramentas de programação baseadas neste padrão são apresentadas neste texto (seção 1.5.).

### 1.4.2. Desenvolvimento sobre arquitetura com memória compartilhada

Como visto no final da seção 1.3.2.1., os elementos de destaque em um programa concorrente são: as *tarefas* concorrentes, a *comunicação* para troca de dados entre estas tarefas e a *sincronização* entre as tarefas para garantir que os dados trocados entre estas sejam realizados de forma a respeitar a definição do programa. O ferramental para programação em ambientes com memória compartilhada deve oferecer recursos

para compor um programa com estes elementos. As características destes recursos são descritas nos parágrafos seguintes.

**Fluxos de execução** são os recursos básicos para programação concorrente, uma vez que são estes que oferecem suporte para a execução de seqüências de instruções. O programador deve, explicitamente criar e destruir os fluxos de execução em seu programa. Sobre os fluxos criados o programador é capaz de ativar a execução de diversas **tarefas**, onde uma tarefa é delimitada por dois pontos de sincronização: iniciando no momento em que recebe seus dados de entrada e finalizando no momento em que resultados são disponibilizados. O número de fluxos de execução definidos por um programa define o maior o número possível de atividades executando de forma concorrente. Normalmente um fluxo de execução é destruído no momento em que tenha terminado a seqüência de tarefas definidas para ele.

A **comunicação** entre instruções é executada através da escrita e leitura de dados em variáveis armazenadas na memória compartilhada. Esta comunicação é efetivada sempre da mesma forma, não importando se as instruções serão ou não executadas sobre o mesmo fluxo de execução.

A **sincronização** em programas concorrentes ocorre de forma implícita somente entre as instruções executadas em um mesmo fluxo de execução. Entre instruções executando em fluxos distintos, dois métodos são possíveis. O primeiro destes utiliza um mecanismo semelhante a uma barreira, onde uma mesma barreira é utilizada para *sincronizar* o acesso a dados compartilhados por seções críticas. Neste mecanismo, o programador deve introduzir instruções no seu código que sejam específicas para manipular estas barreiras de forma a garantir que apenas um dos fluxos de execução execute instruções pertencentes a uma seção crítica. Note que este mecanismo não introduz relação de ordem para execução de seções críticas pertencentes a fluxos distintos, apenas garante que apenas um estará ativo em um determinado instante de tempo.

Pode parecer estranho um mecanismo de controle de troca de dados que não determine ordem de execução entre as tarefas, de certa forma, indo de encontro ao que foi apresentado na seção 1.3.2.1.. No entanto, esta característica ocorre, e muitas vezes é desejada em muitas aplicações. Considere um caso genérico onde existam quatro tarefas $\tau_a$, $\tau_b$, $\tau_c$ e $\tau_d$. É sabido que existem as seguintes relações de precedência: $\tau_a \prec \tau_b$, $\tau_a \prec \tau_c$, $\tau_b \prec \tau_d$ e $\tau_c \prec \tau_d$. Entre $\tau_b$ e $\tau_c$ não existe uma precedência, mas sim uma situação de concorrência, a qual será representada por $\|$, tal que $\tau_b \parallel \tau_c$.

Esta situação pode ser compreendida por um exemplo bastante prosaico: a manipulação do saldo em uma conta em banco, supondo a existência de três operações básicas: consulta saldo, deposita(valor) e saca(valor). Caso uma conta possua saldo inicial de R$ 100,00 e sejam realizadas as seguintes operações: saca(R$ 1,00) e deposita(R$ 1,00), o saldo final deve continuar sendo de R$ 100,00. Esta situação é representada na Figura 1.2. Nesta figura é representado o grafo de dependência entre as operações sobre o saldo de uma conta em banco e uma possível execução sobre dois fluxos concorrentes.

A segunda forma de sincronização é realizada através do controle sobre a ativação e término dos fluxos de execução. A criação de um novo fluxo define a criação de duas novas tarefas, uma que se executará sobre o novo fluxo criado e a outra sobre o mesmo fluxo onde foi solicitada a criação do novo fluxo. De forma ortogonal, o resultado produzido pela última tarefa executada sobre um fluxo de execução pode ser considerado o resultado *deste* fluxo. Assim, uma tarefa pode sincronizar com a destruição de um fluxo de execução para obter resultados de uma tarefa.

Esta segunda forma de sincronização é ilustrada na Figura 1.3, onde é representado

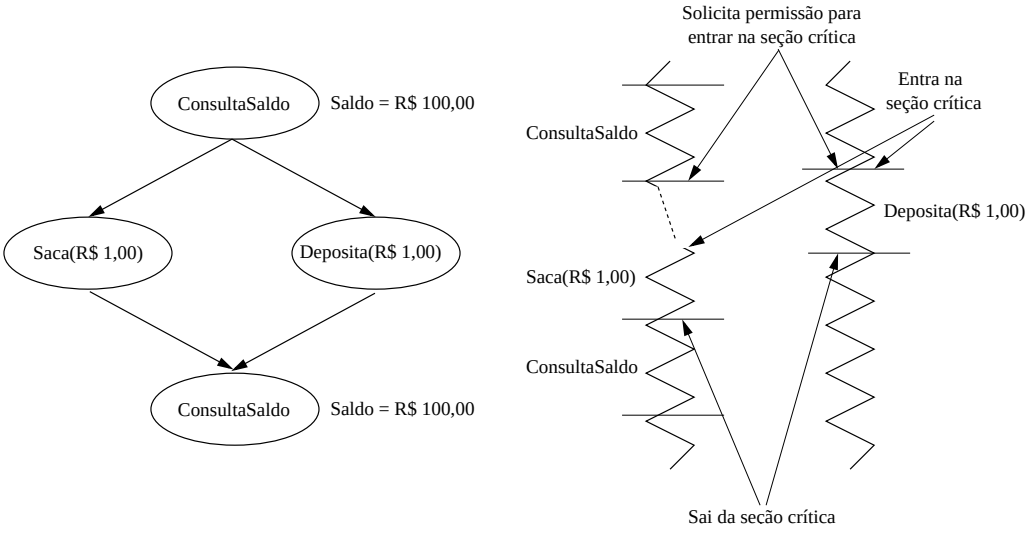


**Figura 1.2: Tarefas concorrendo no acesso a dados em memória.**

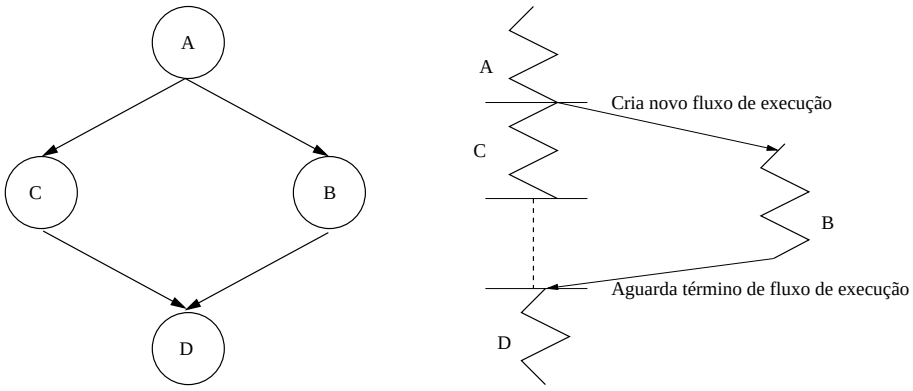


**Figura 1.3: Sincronização de tarefas sobre fluxos de execução.**

o grafo de dependências entre as tarefas e a respectiva execução deste sobre dois fluxos de execução concorrentes.

### 1.4.3. Interação em arquiteturas com memória distribuída

Em um aglomerado, cada nó possui seu próprio conjunto de recursos de processamento: processadores e memória. A concorrência é portanto obtida pela execução de tarefas de forma que diferentes fluxos de execução suportados por diferentes nodos não competem por processador ou memória. Assim sendo, tarefas que não necessitem de sincronizações mútuas podem ser executadas ao mesmo tempo, uma em cada nó, explorando o paralelismo real da arquitetura.

A característica da arquitetura mais marcante é a inexistência de uma área de memória compartilhada para troca de informações entre as tarefas. E como para execução de sincronizações e trocas de dados entre as tarefas é condição necessária à evolução de

um programa, o único recurso é explorar a rede de comunicação de forma a prover mecanismos para comunicar as tarefas. Costuma-se denominar *programa distribuído* um programa em execução sobre uma arquitetura não dotada de memória comum que explora a rede de interconexão entre os nodos para permitir a cooperação entre as tarefas. O termo *distribuída* reflete a configuração da memória, distribuída entre os nodos.

Utilizar uma rede para prover a troca de dados entre fluxos de execução implica no uso de primitivas do tipo *Envia* e *Recebe* (*Send* e *Receive*) ou suas variantes (algumas das quais detalhadas nas próximas seções). Essa forma de interação é consideravelmente mais complexa (e mais onerosa em tempo de processamento) que o mecanismo empregado quando todos processadores dispõem do acesso a uma memória comum. Esta maior complexidade advém do fato que é uma tarefa ”enviadora”necessita conhecer o endereço da tarefa ”destino”para poder realizar a operação *Envia*. De forma análoga, a tarefa ”receptora”deve explicitamente se preparar para receber um dado.

A utilização das primitivas de *Envia* e *Recebe* permite a colaboração e a sincronização entre as tarefas definidas pela aplicação através de troca de mensagens. O princípio desta colaboração é empregar uma primitiva *Recebe* para receber um e *Envia* para enviar um dado. Durante a execução de um programa, chamadas a estas primitivas determinam o término e a criação de novas tarefas. No caso de uma chamada a uma primitiva *Envia*, a tarefa *enviadora* é considerada terminada e o resultado produzido enviado a tarefa *receptora*. No fluxo de execução onde foi realizado o envio dos dados, uma nova tarefa é iniciada, tendo como dados de entrada a própria memória do fluxo de execução. Na chamada a uma primitiva *Recebe*, a tarefa em execução é considerada terminada e a seqüência de instruções após o *Recebe* define a próxima tarefa, a qual estará **Pronta** para ser executada somente após o recebimento do respectivo dado. A entrada desta tarefa são os dados recebidos na mensagem e a própria memória do fluxo de execução corrente.

Apesar de produzir algoritmos com lógicas mais complexas, o uso do compartilhamento de dados via troca de mensagens é bastante popular e são encontradas na bibliografia diversas ferramentas disponibilizando estes recursos de programação. Entre estas ferramentas, podemos citar MPI (*Message Passing Interface*), PVM (*Portable Virtual Machine*) e até mesmo bibliotecas construídas segundo o padrão sockets [35]. As duas primeiras permitem a criação de uma máquina virtual, composta de nodos virtuais, cada nó executando um único fluxo de execução (uma componente da aplicação) e são voltadas exclusivamente para aglomerados de computadores. Bibliotecas de sockets são populares em ambientes do tipo Unix e seu uso pode ser expandido a grandes redes de computadores.

#### 1.4.3.1. Desenvolvimento sobre arquitetura com memória distribuída

A exploração da concorrência por programas em arquiteturas com memória distribuída deve buscar soluções às questões básicas da programação concorrente (conforme 1.3.2.1.), ou seja, definição de grau concorrência a ser explorado (número de *tarefas*), definição da forma como dados serão trocados (*comunicação*) entre as tarefas e como deve ser garantida a correta ordem de troca de dados (*sincronização*). Algumas abordagens sobre estes itens são tratadas na seqüência.

Por se tratar de uma arquitetura distribuída, a primeira questão a ser tratada é a criação de processos sobre os diferentes nodos da rede de computadores que serve de apoio à computação. Sobre estes processos, **fluxos de execução** são capazes de oferecer

suporte a execução de tarefas.[3] O conjunto de instruções determinado para este fluxo de execução é comumente determinado por dois modelos: SPMD ou MPMD, ou seja, *Single* ou *Multiple Program Multiple Data*, respectivamente um único código executando em cada processo ou cada processo executando um código diferente, em ambos os casos, cada processo contendo seu próprio conjunto de dados. Algumas soluções também prevêem a possibilidade de alterar dinamicamente o número de processos envolvidos no cálculo, outras não.

Sobre os fluxos de execução disponíveis, as *tarefas* em execução são delimitadas por dois pontos de sincronização: um determinando seu início, quando do recebimento dos dados necessário à sua computação, e outro determinando seu término, quando o seu resultado é disponibilizado a uma outra tarefa. Observe-se que, ao contrário dos fluxos de execução associados a *threads*, o ciclo de vida de um fluxo de execução se estende durante a execução de toda o programa.

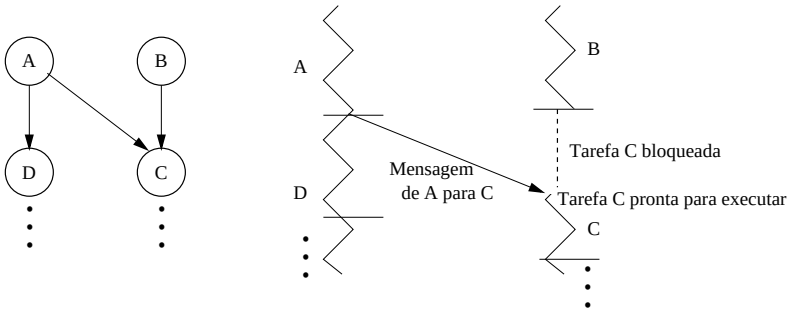


**Figura 1.4: Identificação de tarefas em uma estratégias de comunicação síncrona.**

Internamente ao fluxo de execução, a *comunicação* entre as instruções se dá através da memória do processo. Entre fluxos de execução distintos a comunicação é realizada de forma explícita, através de primitivas de envio e recebimento de dados. Existem muitas variantes para estas primitivas, por exemplo, uma tarefa pode enviar dados a diversas outras tarefas (comunicação em grupo) ou ainda uma tarefa pode ter duas versões de execução, considerando o recebimento ou não de dados no momento da invocação de uma primitiva de recebimento de dados (no caso de uma comunicação assíncrona no recebimento): uma caso o dado tenha sido recebido e outra caso a comunicação ainda não tenha sido efetuada.

Por ser realizada de forma explícita, a comunicação muitas vezes envolve a *sincronização* das tarefas. Deve se notar que, apesar de que tanto a comunicação como a sincronização entre tarefas serem consideradas operações mais complexas em programas distribuídos em relação às equivalentes em programas *multithread*, uma vez que existe a necessidade de endereçamento, elas não requerem controle de acesso a seções críticas. Toda operação de troca transfere um dado, ou realiza uma cópia, de uma tarefa para outra. O cuidado a ser tomado é que cada envio de mensagem possua um ponto de recepção, e vice-versa. A Figura 1.4 exemplifica este caso, apresentando um grafo de tarefas e uma possível execução. Note que no caso apresentado pela figura, a execução da tarefa C é retardada enquanto a mensagem atendida não é recebida.

[3]Nada impede que estes processos executem múltiplos fluxos de execuç ão, no entanto, nesta seção estamos restringindo a discussão ao modelo de processos comunicantes.

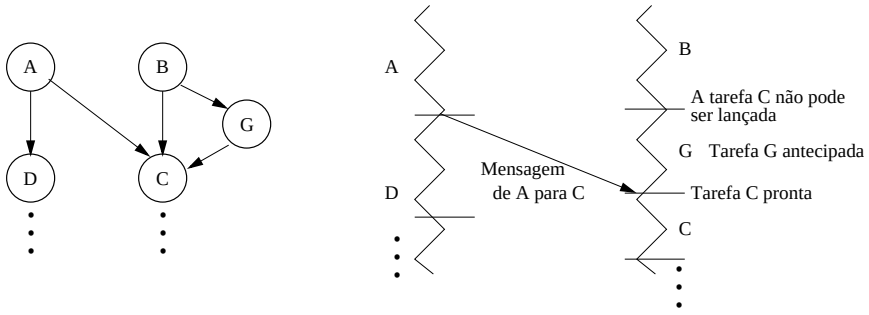


**Figura 1.5: Identificação de tarefas em uma estratégias de comunicação assíncrona.**

Outro caso é representado na Figura 1.5. Neste segundo caso está sendo considerado que a recepção da mensagem é realizada através de uma operação assíncrona. Como a composição do grafo de dependências é realizada de forma dinâmica, podem ocorrer duas situações, dependendo da forma como as mensagens são encaminhadas na rede. A primeira situação encontra-se representada na figura: é verificado se os dados necessários à execução da tarefa C encontram-se disponíveis; como não é o caso, é criada uma tarefa alternativa, a tarefa G, que irá executar uma outra computação, tendo como entrada apenas os dados produzidos por B e a memória do processo. Após o término de G, uma nova verificação é realizada – no exemplo apresentado, a mensagem esperada já foi recebida, estando os dados disponíveis. A segunda situação ocorre quando os dados de entrada de C encontram-se disponíveis logo na primeira verificação, o que implicaria que a tarefa G não seria criada.

A estratégia de executar uma tarefa alternativa pode ser empregada com diversos fins. Por exemplo, pode ser *adiantada* a execução de uma tarefa da aplicação, com vistas a aumentar o desempenho do programa.

### 1.4.4. Concorrência intra e entre nodos

Para uma completa exploração do paralelismo em um aglomerado, é necessário empregar tanto a concorrência intra-nó como a entre nodos. Porém o casamento entre estes dois mecanismos não é uma tarefa simples [8]. O ganho que pode ser obtido na exploração desses dois níveis de concorrência é o recobrimento de parte do tempo gasto em comunicações pela execução de cálculo útil [38], princípio equivalente ao utilizado para compartilhar o uso do processador quando um processo realiza uma operação de E/S.

Diversas ferramentas propõem uma solução disponibilizando os recursos de exploração de ambos níveis de concorrência, como por exemplo podem ser citadas as bibliotecas Anahy [12], Athapascan-0 [8] e Nexus [19] e a linguagem de programação Java [15].

## 1.5. Multiprogramação Leve

A ferramenta típica de exploração da concorrência intra-nó faz uso da multiprogramação leve (*multithreading*), ou seja, permite a criação de vários fluxos de execução

no interior de um processo. Cada um destes fluxos de execução é chamado de processo leve ou *threads*. O termo "leve"faz referência ao fato de que os recursos de processamento alocados a um processo são compartilhados por todas suas *threads* ativas [37], não sendo necessário que cada *thread* possua sua própria descrição de recursos, a manipulação de *threads* é menos onerosa (mais leve) ao sistema operacional.

Dentre os recursos compartilhados pelas *threads*, a memória alocada ao processo desempenha um papel especial: é através desta memória que as *threads* compartilham dados e se comunicam.

Na próxima seção são apresentados os modelos básicos que definem o comportamento de execução de *threads*. Na seqüência são apresentadas algumas funcionalidades do padrão POSIX para *threads*, em termos de seus recursos de programação. O texto propõe uma introdução, tendo como base de discussão a biblioteca LinuxThreads, disponibilizada pelos sistemas Gnu-Linux. O texto oferece uma visão geral sobre estes recursos, o leitor poderá encontrar mais sobre *threads* em [26, 9, 13, 10].

## 1.5.1. Implementações para *threads*

Muitos sistemas operacionais, como Solaris, AIX e Linux, disponibilizam bibliotecas oferecendo recursos para a manipulação de *threads*. Em outros ambientes, como Minix e Windows 95/98, *threads* são disponíveis através de bibliotecas não integradas ao sistema operacional. O fato de serem ou não disponibilizadas diretamente pelo sistema operacional influi diretamente no modelo de *thread* oferecido. Os três modelos básicos, que podem ser identificados pelo mecanismo de escalonamento de *threads* ao processador, são [8]: 1:1 (*one-to-one*), N:1 (*many-to-one*) e M:N (*many-to-many*).

### 1.5.1.1. Estratégia 1:1

Este modelo provê o que se convencionou chamar *threads* sistema (ou *kernel*). Estas *threads* são suportadas diretamente pelo sistema operacional, possuindo os mesmos direitos que processos no escalonamento do processador. Assim, um processo composto por n *threads* sistema recebe n vezes mais o processador que um processo composto por apenas uma única *thread*.

As vantagens do modelo *one-to-one* refletem o fato que *threads* sistemas são manipuladas individualmente pelo sistema operacional. Uma arquitetura multiprocessadora pode, desta forma, ser explorada eficientemente: num instante de tempo, cada processador pode estar executando uma das *threads* da aplicação, provendo o paralelismo na execução das atividades da aplicação. Pelo mesmo princípio, no momento em que uma *thread* sistema bloqueia suas atividades para executar uma operação de E/S, as demais continuam suas respectivas execuções sem nenhum prejuízo.

### 1.5.1.2. Estratégia N:1

O modelo *many-to-one* é normalmente oferecido por bibliotecas no intuito de prover o recurso de *threads* quando estas não são oferecidas pelo sistema operacional. Neste caso as *threads* são denominadas *threads* usuário, executando ao mesmo nível da aplicação. Isto significa que as *threads* são escalonadas no interior do processo, quando este obtiver acesso ao processador. O fato de *threads* usuário serem escalonadas no interior de um processo implica que arquiteturas multiprocessadoras não sejam exploradas

por completo e que todo o conjunto de *threads* usuário de um processo seja bloqueado quando uma destas *thread* iniciar uma operação de E/S.

Em contrapartida, a manipulação de *threads* usuário é ainda menos onerosa que a manipulação de *threads* sistema, o que possibilita o programador utilizar um número superior de *threads* em sua aplicação.

#### 1.5.1.3. Estratégia M:N

Finalmente o modelo *many-to-many* permite que as características de ambos modelos anteriores sejam mescladas. Neste modelo, no interior de cada processo podem existir N *threads* sistema, sobre cada uma das quais é suportada a execução de um subconjunto das M *threads* usuário definidas na aplicação.

Desta forma, o benefício da estrutura mais leve de *thread* usuário, em geral M é muito superior a N, reflete no desempenho de execução e o uso de *thread* sistema aporta as vantagens de um mesmo programa dispor de várias unidades de escalonamento. Outra vantagem é que o programador não precisa restringir o grau de concorrência de seu programa em função dos recursos de hardware disponíveis, bastando achar a relação entre *threads* sistema e *threads* usuário que oferece um bom compromisso de desempenho.

Solaris [29] oferece este modelo de *threads*, utilizando *light weight processes*, as LWPs.

### 1.5.2. Biblioteca LinuxThreads

A biblioteca LinuxThreads é uma biblioteca de *threads* que segue o padrão POSIX. Esta biblioteca é distribuída junto ao sistema operacional Linux, podendo ser utilizada em programas escritos em C/C++. Sua utilização implica na utilização, em um programa, do arquivo de *header* e link-editar a biblioteca de funções libpthread.h.

### 1.5.3. Criação e destruição de fluxos de execuç ão

Um fluxo de execução definido é criado através da criação de uma nova *thread*. No momento da criação desta *thread* é informado o conjunto de instruções que ela deve executar e os dados iniciais para o processamento destes. As instruções são agrupadas no contexto de uma função, sendo que os parâmetros requisitados por esta função que determinam quais dados são necessários para sua execução.

Na interface de programação POSIX, a função a ser executada por uma *thread* pode receber um único parâmetro, sendo este do tipo `void *`. Da mesma forma, o retorno final de uma *thread* é um único valor, de tipo também `void *`. Desta forma, o protótipo desta função é dado por:

```
void* func( void * args );
```

Dada a flexibilidade do tipo `void *`, o programador tem liberdade para construir tipos de dados complexos, tais registros e vetores e/ou matrizes.

A criação de uma nova *thread* se dá através da invocação da primitiva `pthread_create` dentro de um bloco qualquer de comandos.Esta primitiva interage com a biblioteca de manipulação de *threads*, permitindo a criação e a manipulação de um novo fluxo de execução. A primitiva `pthread_create` possui o seguinte cabeçalho:

```
int pthread_create( pthread_t *thid,
 const pthread_attr_t *atrib,
        void *(*funcao) (void *),
 void *args );
```

Cada invocação a `pthread_create` cria uma nova *thread* responsável pela execução da função `funcao`. Nesta criação, a biblioteca de *thread* pode ser instruída para manipular a *thread* com alguns atributos especiais, atributos estes fornecidos especificados pelo programador em `atrib` (se `atrib` é `NULL`, utiliza atributos default[4]. Eventuais parâmetros podem enviados para a nova *thread* através do ponteiro void `args`. O primeiro parâmetro requisitado, `thid`, recebe na execução da primitiva `pthread_create` um identificador da nova *thread* criada; este identificador é representado por um valor numérico, o que permite identificar as *threads* individualmente. O retorno da primitiva permite verificar se a operação ocorreu normalmente ou sem foi verificado algum erro, se o retorno for 0 (zero) não houve erro e a nova *thread* foi criada corretamente.

Observe-se que, no momento da criação de uma *thread* a função indicada para ser executada está apta a executar. Assim, o conjunto de dados que compõem o parâmetro de entrada da função deve estar disponível para leitura. Assim como os dados de entrada de um programa definem a ordem de execução das suas instruções, os dados recebidos pela função determinam a seqüência de execução das instruções no corpo da *thread* – ou, no contexto da programação concorrente, das tarefas. É importante notar, no entanto, que interferem na ordem de execução o acesso a dados compartilhados.

Após ter sido criada, uma *thread* tem um tempo de vida efêmero, sendo este referente ao tempo necessário à execução da função. Desta forma, no momento em que a função termina a execução de sua última tarefa, a *thread* é destruída, disponibilizando o resultado produzido pela última tarefa executada. Note-se que este resultado pode ser considerado o resultado da execução da própria *thread*, de forma análoga que o resultado obtido pela última instrução de um programa seqüencial pode ser interpretado como o resultado da final da execução.

Caso uma tarefa, em execução, necessite dados produzidos por uma outra tarefa em execução sobre uma outra *thread*, é possível evidenciar esta dependência através da primitiva `pthread_join`. Através desta primitiva é possível aguardar que uma *thread* específica termine e recuperar os dados por ela produzidos. O protótipo desta função é:

```
pthread_t pthread_join( pthread_t thid,
        void **ret );
```

O uso do `pthread_join` permite que uma *thread* bloqueie, impedindo que sua execução avance sobre uma tarefa que ainda não pode ser executada, uma vez que esta tarefa necessita, como entrada, os dados produzidos por outra tarefa em execução sobre uma outra *thread*. A *thread* permanecera bloqueada enquanto não ocorrer o término da computação da *thread* identificada por `thid`. Uma vez `thid` tenha terminado sua computação, portanto produzido seus resultados, a tarefa em espera de seus parâmetros de entrada tem sua condição de execução satisfeita, sendo a *thread* é liberada para execução.

A recuperação do retorno de dados da *thread* `thid` é possível através de `ret`, caso a *thread* não retorne nenhum valor, `NULL` pode ser empregado para este parâmetro.

Um exemplo da utilização deste primeiro conjunto de primitivas é apresentado a seguir, onde `pthread_self()` retorna um valor identificando a *thread* corrente.

[4]Consulte o material já referenciado para obter mais informações sobre atributos de criação de *threads* e demais funcionalidades.

```
1. #include <stdio.h>   //E/S em C
2. #include <stdlib.h>  //exit
3. #include <pthread.h> //biblioteca de threads
4.
5. void * OiMundo( void * str ) {
6.  printf( (char *} str );
7.  printf( "Eu sou a thread %d!!!\n", (int) pthread_self() );
8. }
9.
10. void main() {
11.  pthread_t thid;
12.  char *str = "Oi Mundo !!!!";
13.
14.  if( pthread_create( &thid, NULL, OiMundo, NULL ) != 0 ) {
15.    printf( "Ocorreu um erro!!!\n" );
16.    exit(0);
17.  }
18.  printf( "Foi criada a thread %d.\n", (int) thid );
19.  pthread_join( thid, NULL );
20.  printf( "A thread %d ja terminou.\n", (int) thid );
21. }
```

### 1.5.4. Compartilhamento de memória

A comunicação entre *threads* não se limita apenas aos parâmetros de entrada e do retorno da função executada por uma *thread*. A própria memória do processo serve de base de comunicação e, como já foi dito, o acesso à memória se dá pela simples execução de instruções de escrita e leitura. O problema é garantir o correto acesso às informações pelas *threads*, a solução emprega algum mecanismo de sincronização: exclusão mútua ou coordenação (em [32] competição e cooperação, respectivamente).

A sincronização no acesso à memória é necessária para limitar o indeterminismo na execução de programas concorrentes. Neste caso, a função da sincronização é controlar a execução de conjuntos de instruções que acessam uma área de dados compartilhada. A este conjunto de instruções é dada a denominação de seção crítica[5].

Na abstração utilizada neste texto, uma tarefa termina no momento em que solicita entrada em uma seção crítica. Os dados de saída desta tarefa são as informações produzidas e armazenadas em memória até o momento. A nova tarefa inicia assim que for possível a entrada na seção crítica, tendo como entrada não só os dados produzidos pela tarefa que a antecedeu no próprio fluxo de execução, mas também dados na memória compartilhada. De forma análoga, a saída de uma *thread* de uma seção crítica define o término de uma tarefa e o início de outra. Note-se que o término da tarefa que executa a seção crítica permite iniciar a tarefa que a sucede no próprio fluxo de execução e também de outra tarefa que esteja sobre uma outra *thread* bloqueada aguardando a sincronização com o término desta.

[5]Outra forma de sincronismo foi vista anteriormente, com a primitiva `pthread_join`.

#### 1.5.4.1. Mutex

Mutex é um construtor de sincronização que permite o controle de execução de instruções em seções críticas de *threads* concorrentes. Um mutex possui dois estados possíveis: aberto e fechado. A manipulação interna do estado dos mutexes é realizada empregando instruções *test-and-set*, a qual realiza a leitura de um valor em uma posição de memória e a escrita nesta mesma posição em uma única operação, sem que ocorra nenhuma outra interrupção entre a leitura e a escrita (mais sobre estas instruções em [2]).

Através do uso deste recurso, é possível permitir que uma tarefa tenha acesso exclusivo a uma área de dados (o termo mutex é originário do inglês *mutual exclusion*). Tendo exclusividade no acesso, garante-se que uma seção crítica pode ser executada sem que uma outra *thread* tenha alguma instrução que manipule a mesma área de dados executada, interferindo no resultado.

O funcionamento do mutex é bastante simples, e baseia-se em operações de lock e unlock. Ao entrar em uma seção crítica, é fechada uma ”porta”impedindo que outras *threads* avancem pela seção crítica. Ao sair de uma seção crítica, abre-se a porta, permitindo que outras *threads* avancem pelas suas respectivas seções críticas. A operação lock permite solicitar a ”chave”para abrir a porta e unlock devolve esta chave. A aquisição da chave implica em obter a permissão para avançar sobre a seção crítica. A operação unlock devolve a chave, informando a saída da seção crítica.

Uma *thread* restará bloqueada aguardando a liberação do mutex caso realize uma operação lock enquanto uma outra *thread* esteja executando sua seção crítica. Neste caso o lock já foi pego, na execução da operação unlock uma das *threads* bloqueadas no lock será selecionada para ”pegar”o mutex.

Na biblioteca Pthread, os mutex são disponibilizados da seguinte através de um tipo de dado: `pthread_mutex_t`. As funções de manipulação de mutex são as seguintes:

Observe-se que um mutex é uma variável como outra qualquer em um programa, devendo ser instanciada como um dado ordinário no programa. O tipo desta variável é `pthread_mutex_t`, e as operações permitidas (entre outras) são:

```
pthread_mutex_init( pthread_mutex_t *_m,
    pthread_mutexattr_t *atrib );
int pthread_mutex_lock( pthread_mutex_t *m );
int pthread_mutex_unlock( pthread_mutex_t *m );
```

A primitiva `pthread_mutex_init(...)` permite inicializar um mutex, informando se seu estado inicial é aberto ou fechado. A opção default (aberto) pode ser selecionada com o valor `NULL` para o argumento `atrib`. As primitivas `pthread_mutex_lock` e `pthread_mutex_unlock` Permitem manipular o mutex de forma a ”pegar”e ”soltar”o mutex.

A seguir é apresentado um exemplo de controle de acesso a seções críticas. Neste exemplo existe uma variável global `saldo`, acessada de forma concorrente por duas *threads*: `Deposita` e `Saca`. A execução ao final deste código garante que o `saldo`, ao final, terá um valor consistente.

```
int saldo = 100; // saldo é uma variável global, inicializado com R$ 100,00
pthread_mutex_t m; // m é um mutex associado a variável saldo
pthread_mutex_init( &m, NULL );
```

```
// Thread Deposita(1) // deposita R$ 1,00
pthread_mutex_lock( &m );
a = saldo; // lê dado compartilhado
a = a + 1; // realiza a operação
saldo = a; // efetua o depósito
pthread_mutex_unlock( &m );
```

```
// Thread Saca(1)      // saca R$ 1,00
pthread_mutex_lock( &m );
b = saldo; // lê dado compartilhado
b = b - 1; // realiza a operação
saldo = b; // efetua o saque
pthread_mutex_unlock( &m );
```

Conforme ilustra o exemplo apresentado, o mutex `m` é uma variável como outra qualquer. Seu uso depende da lógica adotada no algoritmo implementado. Neste exemplo, o algoritmo associou o mutex m à variável `x`. Cada seção crítica que acessara variável `x` deve explicitar as operações *lock* e *unlock*. Caso existissem outras variáveis compartilhadas, outros mutex poderiam ser criados e manipulados dentro do programa.

O não uso de mutex no caso apresentado poderia incorrer em erro de resultado do programa. O uso do mutex permite que as instruções de uma seção crítica executem de forma atômica. Caso o uso do mutex tivesse sido omitido, esta atomicidade não seria obtida, o que permitiria a execução intercalada das instruções das duas *threads* e um resultado não consistente.

#### 1.5.4.2. Variáveis de condição

Em muitos algoritmos concorrentes, uma *thread* deve entrar em uma seção crítica somente se ela obtém tanto o direito ao acesso exclusivo, utilizando *lock* em um mutex, como também obter a satisfação de uma determinada condição, por exemplo, o valor da variável `x` é `y`). Caso uma destas condições não estiver satisfeita, o código da seção crítica não deve ser executado. Para não ser necessário empregar um algoritmo que teste a intervalos regulares a variável `x`, as *threads* contam com um segundo mecanismo de sincronização, as variáveis de condição.

As variáveis de condição, associadas a um mutex, permitem sincronizar duas (ou mais) *threads* em uma alteração de memória. Um uso típico é na sincronização de *threads* em um algoritmo do tipo produtor/consumidor.

Na biblioteca Linux Threads, uma variável de condição pode ser construída a partir do tipo: `pthread_cond_t` e, como no caso do mutex, a inicialização prevê que uma variável de condição tenha sido previamente declarada. As operações envolvidas na manipulação de variáveis de condição são:

```
pthread_cond_init( pthread_cond_t *c,
   pthread_condattr_t *atrib );
pthread_cond_wait( pthread_cond_t *c,
   pthread_mutex_t *m );
pthread_cond_signal( pthread_cond_t *c );
pthread_cond_broadcast( pthread_cond_t *c );
```

Uma chamada primitiva `pthread_cond_init` inicializa uma variável de condição, informando o valor inicial a ser assumido: condição (satisfeita ou não). A opção default é não satisfeita, podendo ser selecionada informando o valor `NULL` para `atrib`.

A primitiva `pthread_cond_wait` permite que uma *thread* seja bloqueada na espera de uma sinalização. Observe que sempre que uma *thread* ao invocar esta primitiva ela será bloqueada aguardando um sinal na condição `c`. As duas outras primitivas `pthread_cond_signal` e `pthread_cond_broadcast` permitem sinalizar que uma condição foi satisfeita. A particularidade da primitiva *signal* é que o sinal é enviado a apenas uma das *bloqueadas*, enquanto que a primitiva *broadcast* envia o sinal a todas as *threads* bloqueadas.

Um aspecto importante a considerar é que a condição tratada consiste em uma posição de memória compartilhada com, no mínimo, duas *threads*: a *thread* dependente da condição e a *thread* liberadora. Sendo assim, a presença de um mutex é obrigatória, e todas as primitivas de manipulação de uma variável de condição devem estar inseridas

em uma seção crítica protegida por lock e unlock. Para evitar uma situação de *deadlock*, no momento em que uma *thread* é bloqueada em *wait*, o mutex associado a condição é liberado automáticamente (por isto o parâmetro `m` na primitiva *wait*) e, no momento em que a *thread* bloqueada recebe uma sinalização, o mutex deve ser recuperado. Para isto, são executados de forma implícita pelo *wait* uma chamada a uma primitiva unlock no momento em que um *wait* é iniciado e a uma primitiva lock quando o *wait* for satisfeito.

Caso seja a sinalização provenha de um *broadcast*, apenas uma das *threads* obtém o mutex e prossegue a execução, as demais aguardam a liberação do mutex, mesmo tendo a condição satisfeita. Esta característica das variáveis de condição faz com que seja necessário um teste extra, para verificar se a condição continua *estando* satisfeita.

Abaixo um exemplo de um algoritmo produtor/consumidor utilizando variáveis de condição para sincronizar a produção e o consumo de itens em um *buffer*.

```
// Area de memória compartilhada entre o Produtor e o Consumidor
Buffer b;           // Buffer de armazenamento temporário
int nb_itens = 0;   // Contador de ítens no buffer
pthread_mutex_t mb; // Proteção do buffer e do contador
pthread_cond_t c;   // Sincronização entre produtor e consumidor
```

```
void Produtor() {
 Item it;
 for( ; ; ) {
  it = ProduzItem();
  pthreads_mutex_lock( &mb );
  ArmazenaBuffer( b, it );
  nb_itens++;
  pthread_cond_signal( &c );
  pthreads_mutex_unlock( &mb );
 }
}
```

```
void Consumidor() {
 Item it;
 for( ; ; ) {
  pthreads_mutex_lock( &mb );
  while( nb_item <= 0 )
   pthread_cond_wait( &c, &mb );
  it = LeBuffer();
  nb_itens--;
  pthreads_mutex_unlock( &mb );
 }
}
```

É importante observar que uma sinalização em uma variável de condição não é memorizada. Somente as *threads* em estado de *wait* recebem o sinal.

### 1.5.5. Fluxos de Execução e Tarefas

Cabe salientar que uma *thread* não é necessariamente uma tarefa, conforme foi visto pela discussão dos mecanismos utilizados para sincronização de tarefas com primitivas definidas pelo padrão POSIX de threads. Uma *thread* deve ser vista como um suporte à execução do conjunto de instruções pertencentes a diversas tarefas. Esta distinção permite diferenciar a concorrência existente entre as atividades de uma aplicação da concorrência real que pode ser obtida em uma determinada arquitetura [5]. O número de atividades concorrentes é uma característica própria da aplicação, sendo estas codificas em tarefas em um programa. Os fluxos de execução são definidos em termos de recursos de processamento disponíveis pela arquitetura disponível.

A forma clássica de implementar o mapeamento de tarefas sobre *threads* é a utilização de mecanismos de sincronização clássicos como mutexes e variáveis de condição[6]. No entanto, outras formas são possíveis.

Uma técnica de mapeamento pode utilizar um mecanismo de fila de forma a armazenar tarefas prontas (cf. seção 1.3.2.). Esta fila mantém as tarefas enquanto aguardam uma *thread* disponível para serem executas [32, 11]. As *threads* são neste caso vistas como processadores virtuais, oferecendo simplesmente capacidade de processamento as tarefas.

[6]Outros recursos de sincronização, tais semáforos, não foram apresentados neste texto.

*Thread* passa ser sinônimo de tarefa em certos programas em que um fluxo de execução é criado para suportar a execução de uma única tarefa. Ao final desta tarefa, este fluxo é destruído. Esta situação pode ocasionar aumento no custo da aplicação, existindo alguns trabalhos que tem por objetivo reduzir o impacto deste custo adicional, tal Cilk [7] e Anahy [12, 14].

## 1.5.6. Outras ferramentas

Embora o seja o mais clássico e um dos mais utilizados, o padrão POSIX tem dividido seu espaço com outras ferramentas para programação em arquiteturas com memória compartilhada. Dentre estas ferramentas, citamos Anahy, Java e OpenMP.

### 1.5.6.1. Anahy

Anahy [12, 14] é uma ferramenta de programação concorrente modelada para permitir a construção de programas para exploração de processamento de alto desempenho em arquiteturas com memória distribuída. O modelo de arquitetura explorado por Anahy é de uma arquitetura com memória compartilhada por processadores independentes.

A interface de programação de Anahy é composta por duas primitivas básicas, permitindo que tarefas sejam descritas em termos de sincronização de fluxos de execução: criação e sincronização com término. Esta interface foi definida com as primitivas `athread_create(...)` e `athread_join(...)`, possuindo as mesmas funcionalidades (e os mesmos parâmetros) que as funções homólogas da interface POSIX. Outros mecanismos de sincronização não foram introduzidos com vistas a permitir a análise das dependências de dados entre as tarefas por mecanismos de escalonamento. Portanto, a única forma de comunicar dados entre tarefas é através dos parâmetros de entrada e na produção de resultados de saída.

Através da programação segundo a abstração de modelo concorrente proposta por Anahy, o programador pode descrever a concorrência de sua aplicação sem mapear esta concorrência nos recursos disponíveis pela arquitetura. Um núcleo executivo explora as relações de dependência entre as tarefas de forma a evitar sobrecarga de execução.

### 1.5.6.2. Java

O mercado tem mostrado um forte crescimento na utilização da linguagem Java. Uma das razões de sua popularização é devido ao fato dela agregar diversas funcionalidades, entre elas, a multiprogramação leve [27]. O uso de *threads* em Java reflete a natureza orientada a objetos da linguagem. Para poder contar com objetos que possuam *threads* independentes, o programador deve definir classes que herdem da classe `Thread`. Desta classe são herdados três métodos cujo uso é associado à manipulação de *threads*: `run`, `start` e `join`.

O método `run` é um método abstrato definido pela na classe `Thread`. A semântica associada a este método é semelhante a da função a ser executada por uma *thread* criada por `pthread_create(...)`. No entanto, a implementação deste método é de responsabilidade da classe que herda a classe `Thread`. O método deve explicitar a seqüência de instruções a ser executada pela *thread* a ser criada. Os métodos `start` e `join` são implementados pela própria classe `Thread`. Eles permitem, respectivamente, iniciar a execução de uma nova *thread* e sincronizar com seu término.

O contexto da execução dos métodos acima é referente ao contexto de um objeto. Também os dados compartilhados entre *threads* consistem de objetos. O mecanismo de sincronização utilizado é baseado em mecanismo de monitores ([2]), podendo o programador indicar métodos em um objeto que não podem executar de forma simultânea.

#### 1.5.6.3. OpenMP

Diferente das *threads* POSIX, OpenMP privilegia a descrição de uma aplicação em termos de paralelismo de dados[7]. O modelo de execução é considerado *estruturado*, uma vez que viabiliza a criação de *threads* de forma hierárquica [25]. Não é possível, como nas ferramentas vistas anteriormente, manipular *threads* individualmente: elas são tratadas sempre em grupos.

A programação com OpenMP faz uso de diretivas de compilação, e permite que uma *thread*, denominada *thread* mestre, defina um trecho de código como pertencente a uma região paralela. Este código é executado por diversas *threads* concorrentes sobre conjunto de dados distintos. A criação de regiões paralelas pode ser feita de forma recursiva.

A troca de dados é possível através da memória compartilhada, sendo possível definir dados locais às *threads* e compartilhados. A principal forma sincronização de OpenMP é na entrada e na saída de regiões paralelas, e é realizada entre a *thread* mestre e as *threads* filhas. Outros mecanismos de sincronização, tais seções críticas, podem ser utilizados para controlar o acesso a dados compartilhados.

## 1.6. Processos Comunicantes

Um grande número das atuais arquiteturas paralelas de computadores é representado por arquiteturas compostas por com um conjunto de nodos interligados por uma rede de comunicação. Cada nodo desta rede dispõe de um (ou um grupo de) processador(es), um módulo de memória privado e possui um identificar único nesta rede de forma a poder ser identificado. Neste tipo de arquitetura, a malha de interconexão consiste no único recurso que pode ser explorado para a cooperação entre os nodos: placas de rede conectadas a cada nó são responsáveis por (*i*) ao comando de um nó, enviar um conjunto de dados que se encontram em uma determinada região da memória através da rede e (*ii*) receber dados da rede e disponibilizá-los em alguma região da memória para futuramente serem recuperados por algum programa em execução.

Na camada aplicativa (software) esta estrutura reaparece ao se tratar de processos: um processo, em um sistema de computação, consiste em uma unidade de execução autônoma que possui uma área de memória própria para armazenamento de dados e um conjunto de instruções. Este conjunto de instruções define o serviço a ser realizado, e é executado sobre um fluxo de execução próprio ao processo[8] acessando unicamente os dados que possui na sua memória privada. Dois ou mais processos podem vir a cooperar empregando mecanismos explícitos de troca de mensagens, primitivas do tipo *Envia* e *Recebe*, para envio dos dados, independentemente de estarem ou não sendo executados

[7] O site oficial de OpenMP é `www.openmp.org`, tendo sido acessado em 2/nov/2003.

[8] Na seção 1.5. é mostrado que, na realidade, um processo pode possuir vários fluxos de execução, cada um suportando a execução de diferentes seqüências de instruções.

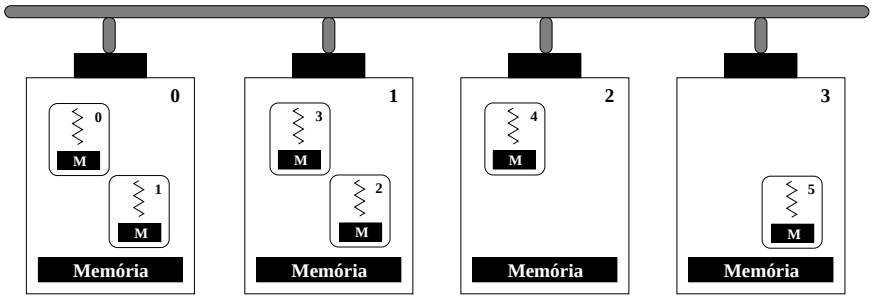


**Figura 1.6: Máquina virtual de seis nodos sobre um aglomerado de 4 nodos..**

sobre um mesmo nó físico. O uso conjunto destas primitivas permite acesso a dados em uma posição de memória remota.

Quando considerado apenas o aspecto físico da comunicação, o problema da troca de mensagens é de compreensão relativamente simples. No entanto, se as necessidades de uma determinada aplicação são consideradas, em se tratando do programa concorrente a ser desenvolvido, questões mais complexas se colocam. Estas questões dizem respeito à sincronização e ao compartilhamento de dados entre as diferentes partes da aplicação.

Uma tarefa, em tal estrutura de aplicação paralela, pode ser definida por uma seqüência de instruções entre duas primitivas de comunicação, as quais permitem a sincronização entre tarefas. Assim, os dados recebidos através de uma primitiva *Recebe* consistem nos parâmetros de entrada da tarefa (uma vez recebidos os dados, a tarefa está pronta para ser executada) e os dados enviados consistem nos parâmetros de saída da tarefa.

Uma aplicação para construir uma estrutura de processos que executem em computadores de uma rede, colaborando entre si através de mensagens, deve dispor de recursos para criação remota de processos e de comunicação [40]. Estes recursos são discutidos nesta seção, fazendo uso de ferramentas que seguem o padrão MPI [28] – *Message Passing Interface*. A discussão é conduzida tendo como base a implementação oferecida pela biblioteca LAM – *Local Area Machine*. O estudo introdutório desta seção pode ser aprofundado consultando [28, 9, 13].

## 1.6.1. Máquina virtual

A solução proposta por MPI para oferecer fluxos de execução para suporte à execução é através da construção de uma *máquina virtual*. Esta arquitetura virtual é composta de *nodos virtuais*, os quais consistem em processos executando sobre nodos (reais) de uma arquitetura (igualmente real). O processo em execução no nó virtual é responsável por executar código referente à aplicação. Não existe limite no número de nodos virtuais que um nó real pode suportar.

Na Figura 1.6 é apresentada uma configuração de uma máquina virtual contando com seis nodos; cada nó virtual representado por um retângulo com bordas arredondadas. As formas quadradas na Figura representam os 4 nodos da máquina real sobre a qual a máquina virtual esta sendo executada. Nos nodos virtuais estão representados o fluxo de execução suportado e sua área de memória privada.

O processo que executa o nó virtual consiste em uma unidade de execução independente, oferecendo suporte à execução de tarefas e podendo receber e enviar mensagens de/a outro nó virtual.

```
main() {
 int id;

 código comum
 id = fork();

 if( id != 0 ) { código pai }
 else { código filho }
}
```

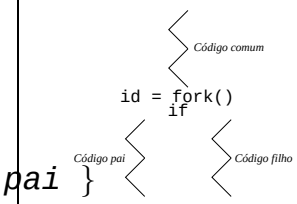


**Figura 1.7: Uso de primitiva `fork` em ambientes Unix..**

#### 1.6.1.1. Programação SPMD

Antes de iniciar o estudo do MPI propriamente dito, é necessário conhecer o modelo de programação empregado, neste caso, o modelo SPMD – *Single Program, Multiple Data*. Neste modelo, a aplicação define um conjunto de processos que deverão executar o mesmo código de forma concorrente. Note que, devido ao fato de diferentes processos manipularem diferentes conjuntos de dados, a porção de código sendo executada por um processo não é necessariamente a mesma que a executada por um outro processo.

Este modelo lembra a utilização de primitivas do tipo `fork` em Unix (exemplo na Figura 1.7 para o código e fluxos de execução gerados). A chamada de uma primitiva *fork* por um processo força sua duplicação em dois processos independentes: um processo *pai* e um processo *filho*. Apesar de cada processo conter uma cópia completa do código da aplicação, instruções de controle de fluxo, como o `if` no exemplo, decidem qual o trecho a ser executado em cada um.

Na programação SPMD, não apenas dois, mas um grupo de $n$ processos executa um mesmo programa. De forma semelhante ao `id` no `fork`, cada processo tem acesso à identificação de sua posição no grupo, ou seja, saber que ele é o $i$-ésimo nó de uma máquina virtual de $n$ nodos; em função da posição de seu nó, o processo pode selecionar a porção do código a ser executado.

Durante a execução dos processos, não existe nenhuma forma de sincronização implícita entre os processos; a introdução de pontos de sincronização entre é de responsabilidade da aplicação: o programador deve, explicitamente, utilizar mecanismos de troca de mensagens para possibilitar a cooperação entre as tarefas. A única exceção diz respeito ao controle do início e no término da execução dos nodos virtuais, bem verdade que ainda assim em pontos informados explicitamente pelo programador.

Outra diferença fundamental entre o `fork` e a programação SPMD é que, no momento da execução do serviço `fork` o processo filho consiste em uma cópia do processo original, implicando que a área de dados seja igualmente duplicada, enquanto processos SPMD consistem em instâncias totalmente autônomas desde o momento em que a execução é iniciada. No caso do `fork`, o processo *filho* executa-se a partir da instrução seguinte à chamada ao `fork` no *mesmo nó* do processo original e contém na sua área de dados uma imagem do estado da memória do processo pai. Os processos na programação SPMD são iniciados todos a partir do mesmo ponto, o início do programa, cada um responsável por inicializar sua própria área de dados.

#### 1.6.1.2. Utilizando LAM

LAM oferece um conjunto de aplicativos para auxiliar no desenvolvimento de aplicações. São oferecidos facilitadores para compilação, como `mpicc` e `mpiCC` para compilar em C ou C++, e para manipulação da arquitetura virtual, tais `lamboot` e `wipe`. O primeiro destes permite iniciar o suporte da máquina virtual sobre uma arquitetura real e o segundo para ”desmontar”uma máquina virtual e seu suporte. O aplicativo `mpirun` permite lançar a execução de um programa, criando a respectiva máquina virtual dedicada a sua execução.

### 1.6.2. Exemplo de programa MPI/LAM

Nesta seção é apresentado um primeiro exemplo de programa em MPI. Neste programa o nó virtual 0 imprime na tela um texto informando o número de nodos da máquina virtual envolvida no processamento.

```
#include <stdio.h>
#include <mpi.h>

int main( int argc, char **argv ) {

 int myrank, size;

 MPI_Init( &argc, &argv );
 MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
 MPI_Comm_size( MPI_COMM_WORLD, &size );

 if( myrank == 0 ) printf( "Foram criados %d nodos\n", size );

 MPI_Finalize();
}
```

Observando que todas primitivas MPI iniciam pelo prefixo `MPI_`, identificamos no exemplo acima à chamada a quatro serviços MPI. A primeira invocação, realizada através de `MPI_Init`, permite inicializar o nó virtual, os parâmetros passados são o `argc` e o `argv` recebidos pelo programa principal. Obrigatoriamente este é o primeiro serviço MPI a ser invocado.

A seguir, as primitivas `MPI_Comm_size` e `MPI_Comm_rank` permite que o processo saiba quantos nodos existem na máquina virtual e qual sua posição neste grupo. No programa apresentado, estas informações são utilizadas para imprimir uma simples mensagem; no caso do nó número `0`, é impressa também uma mensagem informando o número total de nodos virtuais criados. O primeiro parâmetro destas duas primitivas identificam o grupo de nodos virtuais que deseja-se manipular, no caso `MPI_COMM_WORLD`, todos os nodos da máquina virtual.

Finalmente, o uso de MPI é encerrado através de uma invocação à `MPI_Finalize`. Observe que, ao executar uma invocação a `MPI_Finalize`, o processo é bloqueado e permanece neste estado enquanto aguarda que todos os outros processos executem este serviço, sendo então a máquina virtual destruída.

### 1.6.3. Compartilhamento de dados

MPI oferece diversas primitivas de comunicação, as quais permitem a troca de mensagens entre dois (ou mais) processos. Uma mensagem em MPI pode ser representada como no esquema da Figura 1.8 onde é possível observar que ela é composta por quatro campos: a identificação da **origem** e do **destino** da mensagem, os **dados** transmitidos e um **tag**. Este tag permite que as mensagens sejam rotuladas, viabilizando sua filtragem na recepção.


**Figura 1.8: Diferentes componentes de uma mensagem MPI..**

Dentre as primitivas de comunicações disponibilizadas por MPI, encontram-se as tradicionais `MPI_Send` e `MPI_Recv`, para envio e recebimento de mensagens. Note que mensagens recebidas em um nó virtual são armazenadas em uma fila enquanto aguardam a operação de recebimento correspondente; nesta fila, as mensagens são armazenadas na ordem em que foram recebidas, não sendo garantida a ordem temporal de envio das mensagens provenientes de diferentes nodos. A sintaxe destas primitivas é:

```
int MPI_Send( void *buff, int cont, MPI_Datatype tipo,
              int dest, int tag, MPI_Comm grupo );
int MPI_Recv( void *buff, int cont, MPI_Datatype tipo,
              int origem, int tag, MPI_Comm grupo,
      MPI_Status *status );
```

onde:

- `buff` corresponde à área de dados a ser transmitida ou onde os dados recebidos devem ser armazenados. A gerência desta área de memória, alocação e liberação, é responsabilidade do programador.
- `tipo` descreve o tipo do dado que a mensagem contém. Alguns tipos primitivos são definidos por MPI, conforme apresentado na Tabela 1.2. Outros tipos podem ser introduzidos pelo usuário, maiores

  informações em [28].
- `cont` o número de elementos a serem enviados ou recebidos, sendo cada elemento do tipo `tipo`. Em outras palavras, `buff` é um *array* contendo `cont` elementos do tipo `tipo`.
- `origem` e `dest` identificam, respectivamente o nó origem e destino da mensagem. Na recepção, o parâmetro `origem` permite que as mensagens presentes na fila de recepção sejam filtradas, sendo *recebida* procurada a mensagem mais antiga na fila originada de um determinado nó; caso seja informado o valor `MPI_ANY_SOURCE` para este parâmetro, não ocorrerá este filtro, sendo recebida a mensagem mais antiga na fila.
- `tag` aplica um novo nível de filtro, possibilitando a classificação de mensagens através de um rótulo: somente uma mensagem que possua o *tag* informado será recebida. Caso não haja necessidade de filtro a este nível, a este parâmetro deve ser passado o valor `MPI_ANY_TAG`.

**Tabela 1.2: Principais de dados MPI predefinidos..**

| Tipo de dado MPI | Correspondente em C |
|---|---|
| `MPI_CHAR` | `char` |
| `MPI_INT` | `int` |
| `MPI_LONG` | `long` |
| `MPI_FLOAT` | `float` |
| `MPI_DOUBLE` | `double` |
| `MPI_PACKED` | tipo a ser informado |

- `grupo` informa o grupo a que pertencem os nodos `origem` e `destino`, para considerar o grupo composto por todos os nodos, deve ser utilizado `MPI_COMM_WORLD`.

- `status` permite que o receptor tenha acesso a uma série de informações a respeito da mensagem recebida (por exemplo, tamanho em bytes).

O par de primitivas `MPI_Send` e `MPI_Recv` permite realizar comunicações síncronas entre processos. Isto quer dizer que um processo ao invocar uma primitiva `MPI_Send` fica bloqueado até que a comunicação seja concluída. De forma semelhante, ao invocar `MPI_Recv`, o processo fica bloqueado até que a mensagem desejada esteja presente na fila de mensagens. Caso o sincronismo a este nível (bloqueamento do processo) não seja desejado, é possível optar por primitivas de comunicação assíncronas (não bloquantes), tipo `MPI_Isend` e `MPI_Irecv`, cujos parâmetros são os mesmos das suas homólogas síncronas.

No caso de uma operação de envio assíncrono, a mensagem é postada na rede e o processo é desbloqueado, podendo continuar suas operações. Na recepção assíncrona, caso a mensagem já tenha sido recebida, ela é lida para o *buffer* de recepção e o processo pode continuar suas operações; caso a mensagem ainda não tenha sido recebida, o processo é também liberado para continuar sua execução, devendo em um momento posterior tentar uma nova leitura.

#### 1.6.3.1. Comunicação de grupo

Outra possibilidade de comunicação em MPI é a comunicação de grupo, onde um processo pode enviar, ou receber, mensagens para, ou de, todos outros processos. Exemplos destas primitivas são `MPI_Bcast` e `MPI_Reduce`, que oferecem de mecanismos de *broadcast* e redução. Observe-se que nestes serviços de comunicação é necessário identificar o grupo de processos envolvidos (`MPI_Comm`). Neste grupo, um dos nodos é identificado como raiz da comunicação; quando da ocorrência de um *broadcast* o processo raiz é responsável pelo envio da mensagem, e no caso de uma redução, pelo recebimento e tratamento das mensagens.

A sintaxe para o *broadcast* e para o *reduce* em MPI é apresentada. Note que existe apenas uma primitiva para cada uma destas operações. Em ambos os casos, tanto no *broadcast* como na redução, a mesma primitiva deve ser invocada, tanto pelo processo raiz, que envia a mensagem, como pelos processos que a receberão.

```
int MPI_Bcast( void *buff, int cont, MPI_Datatype tipo,
               int raiz, MPI_Comm grupo );
int MPI_Reduce( void *operando, void* resultado, int cont,
        MPI_Datatype tipo, MPI_Op operador,
        int raiz, MPI_Comm grupo );
```

Ao utilizar `MPI_Bcast`, a semântica associada ao parâmetro `buff` é obtida pelo valor fornecido ao parâmetro `raiz`. Caso o valor informado para `raiz` corresponda a própria posição do processo no `grupo` (seu *rank*), `buff` contém os dados a serem enviados: este processo é considerado a raiz da comunicação. Nos demais processos, `buff` corresponde a área de dados onde deve ser armazenado os dados recebidos.

A redução é um mecanismo inverso ao *broadcast*, em que vários nodos enviam mensagens (uma mensagem por nó) a um nó raiz. A quantidade de parâmetros necessários a uma operação de redução é maior que em um *broadcast* pois é necessário especificar a *ação de redução* a ser tomada quando os dados forem recebidos pelo nó raiz. Esta operação é especificada em `operador`, que, no nó raiz é realizada sobre o valor recebido em `operando` e em `resultado`, armazenando o resultado em `resultado`. Em um outro nó que não seja o raiz, esta operação não é realizada, sendo o dado armazenado em `operando` transmitido na mensagem.

Existem várias operações pré-definidas para redução, tais `MPI_SUM` que produz a soma de `operando` e `resultado` e `MPI_MAX` mantém o maior valor entre `operando` e `resultado`.

## 1.6.4. Outras ferramentas

MPI como ferramenta para o processamento de alto desempenho tornou-se popular por ser um padrão de fato e disponibilizar uma quantidade razoável de recursos, tais as primitivas para comunicação em grupo. O número de opções, no entanto, é razoável. Entre estas citamos PVM, sockets, RPC e Java.

### 1.6.4.1. PVM

Assim como MPI, PVM (*Parallel Virtual Machine*) [21, 13], oferece uma infraestrutura para execução em aglomerados de computadores, explorando o conceito de máquina virtual. No entanto o PVM não possui uma interface de serviços padronizada, o que não impediu que viesse, de fato, tornar-se popular.

A opção entre PVM e MPI deve ser uma decisão própria do programador. Sugere-se consultar dados sobre funcionalidades, suporte a heterogeneidade e desempenho para decisão de qual ferramenta é a mais apta a um determinado fim.

### 1.6.4.2. Sockets

Uma interface de programação com um número bastante reduzido de primitivas é a oferecida por bibliotecas sockets [35], intoduzidas por Berkeley Unix. Conceitualmente, sockets podem ser definidos como uma porta sobre a qual um processo pode enviar ou receber mensagens através de uma rede.

Através das bibliotecas sockets, é possível utilizar dois tipos de protocolo para as comunicações: TCP e UDP. O protocolo TCP é orientado a conexão, garantindo uma maior segurança na entrega das mensagens. Em uma conexão TCP existe o papel do

servidor e do cliente, a diferença entre estes dois papéis é que o servidor aguarda que um cliente conecte para iniciar a troca de mensagens. Já o protocolo UDP baseia-se em datagrama, oferecendo uma menor confiabilidade na entrega de mensagens, mas, por outro lado, não força o programador utilizar uma estrutura cliente/servidor e oferece maior desempenho.

#### 1.6.4.3. RPC

RPC (*Remote Procedure Call*) [4, 2] é uma alternativa à programação em ambientes com memória distribuída, a qual oferece um maior nível de abstração ao programador. RPC oferece ao programador uma abstração bastante próxima à chamada de um procedimento ordinário para invocar um procedimento a ser executado em um outro nodo.

A estrutura de suporte é baseada em um servidor, responsável pelo cálculo, um cliente, que invoca os serviços do servidor, e um *stub*. O *stub* é executado junto ao cliente e consiste em um procedimento responsável por capturar as invocações ao serviço remoto. Uma vez capturada a invocação, na realidade é realizada uma chamada local ao *stub*, uma mensagem indicando uma solicitação de serviço é composta e enviada ao nodo servidor.

Na sua origem, RPC consiste em invocações síncronas, ou seja, o cliente aguarda o retorno, via *stub* do procedimento invocado. Novas versões deste mecanismo permitem invocações assíncronas.

#### 1.6.4.4. Java

Como citado anteriormente, a linguagem Java tem se popularizado pelos seus recursos de programação, entre eles a possibilidade de desenvolvimento explorando arquiteturas com memória distribuída [23]. Por se tratar de uma linguagem orientada a objetos, os recursos para programação distribuída são introduzidos segundo o modelo da programação orientada a objetos.

Um destes recursos é a exploração de comunicação segundo o padrão sockets. A interface aplicativa de programação de Java oferece um conjunto de classes com funcionalidades de servidores e clientes para conexões TCP e também para comunicações datagrama.

Outra possibilidade de Java é explorar uma estrutura de programa próxima a de RPC, o uso de RMI (Remote Method Invocation). Neste caso não são funções que são invocadas, mas sim métodos pertencentes a objetos.

## 1.7. A Programação Concorrente no Processamento de Alto Desempenho

A programação concorrente por si só não garante níveis de desempenho na execução de programas em arquiteturas paralelas. Embora as questões que envolvam o projeto do programa concorrente sejam importantes, outros aspectos necessitam ser explorados. Dentre eles técnicas de escalonamento e de troca de mensagens eficientes.

### 1.7.1. Escalonamento de tarefas

A exploração de arquiteturas paralelas, dotadas ou não de memória comum, é realizada a mais de duas décadas para solução de problemas oriundos das mais diversas áreas [6, 34]. No entanto, as interfaces de programação para o processamento concorrente oferecem pouco conforto de programação. Ao contrário da programação seqüencial, na programação concorrente não é possível abstrair as características dos recursos de hardware disponíveis, em particular o número de processadores e a distribuição dos módulos de memória. Para obter um programa concorrente, o programador necessita, em um primeiro momento, decompor sua aplicação em diversas atividades concorrentes e estabelecer os critérios de sincronização entre estas atividades para em seguida poder mapeá-las sobre os recursos disponíveis.

Em se tratando de programação para o processamento de alto desempenho, é de se observar que se faz necessário introduzir alguma estratégia de distribuição da carga gerada pelo programa em execução entre os recursos de processamento. Nas ferramentas apresentadas nas seções anteriores (*threads* e MPI), este compartilhamento de carga deve ser provido no próprio programa, pois é o programador que deve mapear dados e tarefas sobre os módulos de memória e sobre os processadores disponíveis.

Na bibliografia são encontrados diversos ambientes para o processamento paralelo dotados de recursos de escalonamento com capacidade de gerir a carga computacional gerada por um programa. São exemplos destes ambientes Anahy [12], Athapascan-1 [20], Cilk [7] e Jade [30]. Estas ferramentas são dotadas de um núcleo de escalonamento que toma para si a responsabilidade de distribuir a carga computacional, ou seja as tarefas, dos programas em execução sobre os recursos de processamento, respeitando as sincronizações e dependência de dados entre elas.

### 1.7.2. Mensagens ativas

A comunicação de dados entre tarefas executando em nodos distintos é uma das operações mais custosas em se tratando de processamento concorrente em arquiteturas com memória distribuída. Diversas alternativas têm surgido, como protocolos de comunicação mais eficientes e hardwares para comunicação com melhor desempenho [3].

Outra estratégia utilizada emprega *mensagens ativas*. Mensagens ativas [16] permitem realizar comunicações sem introduzir grande quantidade de sobrecustos de execução [39]. Diferente dos mecanismos tradicionais de envio e recebimento de mensagens, realizados por operações explícitas de *send*/*receive*, mensagens ativas permitem enviar uma mensagem contendo não só o dado a ser transmitido mas também a operação a ser realizada. No momento da recepção de uma mensagem ativa, um procedimento é acionado para recuperar a mensagem da rede e invocar o serviço desejado.

A implicação deste método é que simplifica a implementação de aplicações e aumenta o desempenho de execução. Programas que utilizam mensagens ativas são menos complexos pois não necessitam da introdução de pontos de sincronização explícitos para recepção de dados. O aumento de desempenho é garantido pela própria estrutura de suporte às mensagens ativas e pelo fato que os dados enviados através da rede podem ser tratados com uma menor latência.

## 1.8. Bibliografia

[1] American National Standards Institute. *IEEE standard for information technology: Portable Operating Sytem Interface (POSIX). Part 1, system application program interface (API) — amendment 1 — realtime extension [C language]*. IEEE Computer Society, Silver Spring, 1994.

[2] G. R. Andrews. *Foundations of Multithreaded, Parallel and Distributed Programming*. Addison Wesley, Reading, 2000.

[3] A. M. P. Barcellos and L. P. Gaspary. Tecnologias de rede para processamento de alto desempenho. In G. G. H. Cavalheiro and M. Pasin, editors, *Escola Regional de Alto Desempenho*, III *ERAD*, chapter 3. SBC/UNISINOS/UFSM/UNILASALLE, Santa Maria, 2003.

[4] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computers Systems*, 2(1):39–59, February 1984.

[5] David L. Black. Scheduling support for concurrency and parallelism in the mach operating system. *Computer*, 23(5):35–43, May 1990.

[6] G. E. Blelloch. Programming parallel algorithms. *Comm. of the ACM*, 39(3):85–97, March 1996.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. C. E. Zhou. Cilk: an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, 30(8):207–216, August 1995.

[8] A. S. Carissimi. *Le noyau exécutif Athapascan-0 et l'exploitation de la multiprogrammation légère sur les grappes de stations multiprocesseurs*. Thèse de doctorat, Institut National Polytechnique de Grenoble, Grenoble, France, September 1999.

[9] G. G. H. Cavalheiro. Introdução à programação paralela e distribuída. In P. Navaux and T. Diverio, editors, *Escola Regional de Alto Desempenho*, I *ERAD*, chapter 2. SBC/II-UFRGS/PUCRS/UNISINOS, Gramado, 2001.

[10] G. G. H. Cavalheiro. Multiprogramação leve para o processamento de alto desempenho. In *Escola de Informática Norte*, IV *EIN*, chapter 5. SBC, Belém-Macapá-Palmas, 2002.

[11] G. G. H. Cavalheiro, Y. Denneulin, and J.-L. Roch. A general modular specification for distributed schedulers. In *Proceedings of Europar'98*, Southampton, England, September 1998. Springer Verlag, LNCS 980.

[12] G. G. H. Cavalheiro and E. C. Dall'Agnol L. C. Villa Real. Uma biblioteca de processos leves para a implementação de aplicações altamente paralelas. In *Workshop de Sistemas Computacionais de Alto Desempenho*, IV *WSCAD*, São Paulo, November 2003. SBC.

[13] C. M. da Costa, D. Stringhini, and G.G. H. Cavalheiro. Programação concorrente: Threads, mpi e pvm. In T. Diverio and G. G. H. Cavalheiro, editors, *Escola Regional de Alto Desempenho*, II *ERAD*, chapter 2. SBC/II-UFRGS/UNISINOS/ULBRA, São Leopoldo, 2002.

[14] E. C. Dall'Agnol, L. C. Villa Real, E. D. Benitez, and G. G. H. Cavalheiro. Portabilidade na programação para o processamento de alto desempenho. In *Workshop de Sistemas Computacionais de Alto Desempenho*, IV *WSCAD*, São Paulo, November 2003. SBC.

[15] H. M. Deitel and P. J. Deitel. *Java: como programar*. Bookman, Porto Alegre, 2000.

[16] T. V. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *19th Int. Symp. on Computer Architecture*, Gold Coast, May 1992.

[17] H. El-Rewini and T. Lewis. *Distributed and Parallel Computing*. Manning Publications, Greenwich, 1997.

[18] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, September 1972.

[19] Ian Foster, Carl Kesselman, and Steve Tuecke. The Nexus approach to integrating multithreading and communications. *Journal of Parallel and Distributed Computing*, 37(1):70–82, August 1996.

[20] F. Galilée, J.-L. Roch, G. G. H. Cavalheiro, and M. Doreille. Athapascan-1: online building data flow graph in a parallel language. In *Pact'98*, Paris, France, October 1998.

[21] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*. MIT, Massachusetts, 1994.

[22] A. Goldman. Modelos para computação paralela. In G. G. H. Cavalheiro and M. Pasin, editors, *Escola Regional de Alto Desempenho*, III *ERAD*, chapter 2. SBC/UNISINOS/UFSM/UNILASALLE, Santa Maria, 2003.

[23] E. R. Harold. *Java Network Programming*. O'Reilly Nutshell, Cambridge, 2000.

[24] R. Jagannathan. Dataflow models. In A. Y. H. Zomaya, editor, *Parallel and Distibuted Computing Handbook*, chapter 8. John Wiley and Sons, New York, 1996.

[25] C. Leopold. *Parallel and Distributed Computing*. John Wiley and Sons, New York, 2001.

[26] B. Nichols, D. Buttlar, and J. P. Farrel. *Pthreads Programming*. O'Reilly Nutshell, Cambridge, 1996.

[27] Scott Oaks and Henry Wong. *Java threads*. O'Reilly, Sebastopol, 1999.

[28] Peter S. Pacheco. *Programming parallel processors using MPI*. Morgan Kaufmann, San Francisco, 1995.

[29] M. L. Powell, Steve R. Kleiman, Steve Barton, Devang Sshah, Dan Stein, and Mary Weeks. Sunos multi-thread architecture. In LNCS 980 Springer Verlag, editor, *Proceedings of the Winter 1991 USENIX Technical Conference and Exhibition*, pages 65–80, Dallas, January 1991.

[30] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, May 1998.

[31] F. B. Schneider. *On Concurrent Programming*. Springer, New York, 1997.

[32] Robert W. Sebesta. *Conceitos de linguagens de programação*. Bookman, Porto Alegre, 4. ed., 2000.

[33] A. Silberschatz and P. Galvin. *Operating systems concepts*. John Wiley & Sons, New York, 5 edition, 1997.

[34] D. B. Skillicorn and D. Talia. Models and languages for parallel computations. *ACM Computing Surveys*, 30(2):123–169, 1998.

[35] W. R. Stevens. *Unix Network Programming*, volume 1. Prentice Hall, Upper Saddle River, 2 edition, 1998.

[36] Andrew S. Tanenbaum. *Modern operating systems*. Prentice Hall, New Jersey, 1992.

[37] Uresh Vahalia. *UNIX Internals*. Prentice Hall, Englewood Cliffs, 1976.

[38] L. G. Valiant. A bridging model for parallel computation. *Comm. of the ACM*, 33(8):103–111, August 1990.

[39] D. A. Wallach, W. C. Hsieh, K. L. Johnson, and M. F. Kaashoek. Optimistic active message: a mechanism for scheduling communications with computation. In *5th ACM SIGPlan Symp. on Principles & Practice of Parallel Programming*, Santa Barbara, August 1995. ACM SIGPlan.

[40] Barry Wilkinson and Michael Allen. *Parallel programming: tecniques and applications using networked workstations and parallel computers*. Prentice-Hall, Upper Saddle River, 1999.

[41] A. Y. H. Zomaya, editor. *Parallel and Distibuted Computing Handbook*. John Wiley and Sons, New York, 1996.