5

Introdução à programação para arquiteturas multicore

Professores:

Mauricio Lima Pilla[1]
(pilla@ufpel.edu.br)
Rafael Ramos dos Santos[1]
(rafael.santos@ceitec.org.br)
Gerson Geraldo H. Cavalheiro[2]
(gerson.cavalheiro@ufpel.edu.br)

Resumo:

A rápida popularização de processadores multicores possibilitou o ingresso do processamento paralelo em praticamente todos os sistemas computacionais. Como resultado, aumentou consideravelmente a demanda pelo desenvolvimento de aplicativos que explorem efetivamente esta capacidade de processamento. Neste contexto, o modelo de programação e execução proposto pela multiprogramação leve é o que possibilita obter maiores vantagens no desenvolvimento de software para este tipo de arquitetura. O presente capítulo caracteriza as arquiteturas de processadores multicores, bem como os recursos oferecidos pela multiprogramação leve. O texto é ilustrado com a descrição de três das mais populares ferramentas para multiprogramação leve, Pthreads, .NET e OpenMP, visando oferecer um panorama de diferentes domínios de aplicação da multiprogramação leve sobre processadores multicores. Por fim, são apresentadas técnicas de programação focadas na exploração eficiente das características dos processadores multicores.

[1] Graduação e doutorado em Ciência da Computação na Universidade Federal do Rio Grande do Sul. Atualmente, é Professor na Universidade Federal de Pelotas. Seus temas de pesquisa incluem: Arquiteturas de Computadores, Processamento Paralelo e Distribuído, Computação Pervasiva.

[1] Graduado em Informática pela Pontíficia Universidade Católica do Rio Grande do Sul. Mestrado e doutorado em Ciência da Computação pela Universidade Federal do Rio Grande do Sul. Atualmente, é Líder de Grupo no CEITEC. Seus principais temas de pesquisa são: Arquiteturas de Computadores, Processamento Paralelo e Distribuído.

[2] Graduado em Informática pela Pontíficia Universidade Católica do Rio Grande do Sul. Mestre em Ciência da Computação pela Universidade Federal do Rio Grande do Sul. Doutor em Informatique, Systèmes et Communications pela Institut National Polythecnique de Grenoble. Atualmente, é Professor da Universidade Federal de Pelotas. Atua principalmente nos seguintes temas: Arquiteturas Paralelas e Distribuídas, Programação Paralela, Escalonamento, Regulação de Carga, Fluxo de Dados.

5.1. Introdução

O surgimento da programação concorrente se deu por volta de 1960 [Andrews 2000], com o surgimento de dispositivos dedicados a operações de entrada e saída de dados (E/S). O emprego da concorrência tinha como objetivo obter melhor aproveitamento do tempo de processamento do processador principal, pois, uma vez que dispositivos especializados responsabilizavam-se por executar as operações de E/S, não havia necessidade do processador permanecer ocioso aguardando o término das requisições. A estratégia adotada permitia a execução de fluxos de instruções de diferentes aplicações, sobrepondo computação do fluxo de execução de uma aplicação com operações de E/S de outras. Passados quase 50 anos, o compartilhamento de tempo ainda é uma técnica muito empregada. Hoje, no entanto, o horizonte de aplicação da programação concorrente é mais amplo, permitindo que seja empregada em praticamente todos os recursos computacionais existentes com apoio de diferentes ferramentas de programação.

Processadores multicores refletem os mais recentes avanços na arquitetura de processadores, trazendo o poder do processamento paralelo aos computadores domésticos e estendendo a barreira da supercomputação. Basicamente, um processador é dito multicore quando possui dois ou mais núcleos completos de processamento (processadores) no mesmo chip. A arquitetura básica segue o modelo da arquitetura SMP (Symmetric Multi-Processors). Como nesta arquitetura, os diferentes processadores (ou *cores*) são capazes de executar fluxos de instruções de forma independente, e compartilham um espaço de endereçamento provido por uma área de memória comum.

A multiprogramação leve permite que uma aplicação seja descrita em termos de diversos fluxos de execução, ou *threads*, capazes de executar de forma concorrente. Cada fluxo possui uma área de endereçamento própria, mas os threads compartilham acesso a um espaço de endereçamento global. A concorrência, neste caso, expressa não somente a disputa de recursos de processamento, como tempo de processador e espaço de memória, mas também a disputa dos threads de um programa pelo acesso aos dados armazenados em memória compartilhada.

Como consequência natural da popularização dos multicores, o desenvolvimento de novos aplicativos, desde os programas multimídia até elaboradas aplicações de Computação Científica, deve incorporar técnicas de multiprogramação leve [Akhter and Robert 2006].

O restante deste texto introduz conceitos e ferramentas para multiprogramação leve em processadores multicores. A próxima seção descreve as características principais dos processadores multicores, sendo também apresentados aspectos históricos da evolução tecnológica que resultou nesta arquitetura e produtos de mercado. Na Seção seguinte, Seção 5.3, são apresentados conceitos relacionados à multiprogramação leve e questões relacionadas ao desenvolvimento de programas. A sequência é dada pela Seção 5.4, onde três ferramentas para multiprogramação leve são apresentadas: .Net, POSIX threads e OpenMP. A Seção 5.5 discute técnicas e recursos de programação aplicáveis a arquiteturas multicores. Por fim, a Seção 5.6 tece comentários finais sobre este texto.

5.2. Arquiteturas Multiprocessadas

Os avanços na área de circuitos integrados permitiram o aumento de desempenho dos microprocessadores nos últimos 25 anos graças às frequências de *clock* cada vez maiores, e mais oportunidades para inovações microarquiteturais. Esta evolução, ao longo dos anos, permitiu que o aumento de desempenho fosse observado pelo usuário, sem que este tivesse que se preocupar necessariamente com a forma com que o software era escrito. As características do software existente são na maioria das vezes projetadas em função da aplicação em si, e não em função do resultado em termos do desempenho de sua execução. Isto se dá porque historicamente observou-se um aumento de desempenho na ordem de 40% ao ano transparentes para o software. Ou seja, cada nova geração de microprocessador permitiu que o software se tornasse mais complexo, sem preocupação com o desempenho.

Explorando frequências de *clock* mais rápidas, os microprocessadores disponíveis comercialmente até poucos anos atrás exploravam essencialmente o paralelismo no nível de instruções, executando de forma paralela instruções definidas por um único fluxo de execução, thread ou processo. Com esta estratégia, programas sequenciais poderiam se beneficiar das inovações microarquiteturais, sem necessariamente serem reescritos, ou seja, de forma transparente. Este cenário infelizmente contribuiu para que pouco fosse explorado comercialmente no que diz respeito à paralelização do software convencional, aquele usado no dia-a-dia, incluindo, nesta categoria, sistemas operacionais.

A frequência de *clock* não é a única medida de desempenho, nem mesmo é necessariamente uma boa medida, mas no entanto é uma medida representativa. Nas útimas três décadas observou-se um incremento significativo nas frequências de *clock* atingindo, atualmente, a faixa dos 3 a 4 GHz. No entanto, pode-se perceber que nos últimos cinco anos houve uma redução nesta escala de incremento.

De forma similar, os mecanismos até então empregados para explorar o paralelismo no nível de instrução também encontraram seus limites, já que o custo para incrementar o desempenho a partir de inovações microarquiteturais não justifica maiores esforços nesta área, pois em geral estão limitados a um único thread.

O que se percebe atualmente é que uma nova abordagem deve ser considerada para que novos índices de desempenho sejam alcançados. Todas as inovações advindas de melhor e mais eficiente tecnologia e de mecanismos agressivos implementados na microarquitetura resultaram em um aumento significativo da complexidade e custo de projeto, sem que resultados em escalas proporcionais fossem propiciados. Tornou-se comercialmente inviável e tecnicamente quase impossível aumentar o desempenho das aplicações convencionais apenas com o aumento da frequência de *clock*, ou com o uso de técnicas que exploram o paralelismo limitado a um único thread.

Nesta Seção, são discutidas as características das arquiteturas multicores. Os aspectos históricos e a evolução destas arquiteturas são apresentados como forma de contextualizar a sua inserção no mercado como tecnologia de ponta e, principalmente, como forma de enfatizar o impacto desta arquitetura no projeto e desenvolvimento de software.

5.2.1. Modelo de Von Neumann

A arquitetura de von Neumann é um modelo de computação que emprega uma estrutura comum de memória que armazena dados e intruções. O modelo apresentado por John von Neumann (1945) foi considerado uma revolução na época ao especificar o modelo de referência para arquiteturas sequenciais.

Os computadores até então possuíam um programa fixo e não permitiam a execução de outros programas, cujo código já não se encontrava no seu projeto. A proposta de von Neumann introduziu um novo conceito, permitindo que se modificasse o conteúdo de uma memória introduzindo ou modificando instruções como forma de se reprogramar o computador. Surge também neste momento a ideia de conjunto de instruções e de detalhamento da computação, a partir de uma sequência de instruções (programas).

Esta sequência de instruções tem como finalidade especificar os passos que devem ser executados pelo processador, para que um determinado algoritmo seja realizado. Cada instrução ocupa uma ou mais posições de memória e é representada por uma sequência específica de bits (código de máquina). Para que o programa possa ser executado, o processador realiza uma série de etapas, independentes do algoritmo.

Apesar de ser um modelo simples, porém, revolucionário, o modelo de von Neumann introduziu um problema que permanece até hoje e que John Backus chamou de **gargalo de von Neumann** [Backus 1978], também conhecido na literatura por *memory gap*. A comunicação entre processador e memória representa um fator limitante, pois todo o processamento depende de dados e instruções que estão armazenados na memória. Neste modelo, o acesso à memória determina a velocidade em que o processador pode realizar o processamento propriamente dito. Outro fato historicamente importante é que a velocidade dos processadores e o tamanho das memórias têm aumentado muito mais rapidamente do que a velocidade de acesso dos processadores a estas memórias.

No que diz respeito ao software, a maioria das linguagens de programação convencionais está baseada no modelo de computação de von Neumann, em particular as linguagens do paradigma imperativo. Nestas linguagens, variáveis correspondem a células de armazenamento, comandos de atribuição correspondem a leitura, comandos de escrita, operações artiméticas e comandos de controle correspondem a instruções de teste e salto. Segundo Backus, o símbolo "=" representa linguisticamente o gargalo de von Neumann. Com este modelo reforça-se a maneira sequencial de pensar e escrever programas.

5.2.2. Lei de Moore

Gordon Moore, fundador da Intel, previu empiricamente que o avanço na tecnologia seria tal que o número de transistores em um circuito integrado dobraria a cada 24 meses. Esta previsão passou a ser chamada de *Lei de Moore*, e os avanços da indústria comprovam esta previsão até os dias atuais. Porém, o fato mais importante em relação a isso não é a previsão em si, mas sim que os projetistas de microprocessadores se depararam com um problema singular: o que fazer com tantos transistores [Burger and Goodman 1997]?

O número cada vez maior de transistores disponíveis permitiu o incremento das inovações microarquiteturais. Entende-se por inovação microarquitetural toda técnica nova que melhora ou explora de maneira mais eficiente (ou, pode-se dizer, mais agres-

siva) a execução de instruções no interior de um microprocessador.

Por outro lado, o aumento do número de transistores é resultado da diminuição do tamanho do transistor em si, que, por sua vez, é resultado das melhorias na tecnologia de fotolitografia capaz de "imprimir" transistores cada vez menores em uma pastilha de silício.

Transistores menores permitem frequências de *clock* mais elevadas, que, associadas a técnicas mais agressivas de execução de instruções, oferecem melhor desempenho. No entanto, microprocessadores atuais, e em especial os da família x86, atingiram uma barreira técnica no que diz respeito ao consumo de potência a frequências muito elevadas.

Segundo [Koch 2005], em 1993 um microprocessador Intel Pentium tinha por volta de 3 milhões de transistores, enquanto hoje um microprocessador dual core Itanium 2 tem mais de 1 bilhão de transistores.

Por fim, a velocidade das memórias não cresce na mesma proporção que a velocidade da lógica de processamento. No final da década de 1980, um microprocessador `i486` necessitava de 6 a 8 ciclos de *clock* de processador para acessar a memória. Já o microprocessador Pentium necessita por volta de 224 ciclos de *clock* para a mesma tarefa, ou seja, um aumento de quase 20 vezes.

Os fatores apresentados fazem crer que não serão vistos aumentos drásticos de frequência enquanto os limites físicos não forem atingidos, embora ainda exista espaço para aumentar o número e a velocidade dos transistores.

5.2.3. Evolução das arquiteturas de microprocessadores

O aumento de desempenho observado a cada nova geração de microprocessador foi obtido principalmente pelas otimizações realizadas a partir das pesquisas em três frentes distintas:

1. Frequência de *clock*: o aumento da frequência de *clock* pode ser resultado da diminuição dos atrasos de propagação dos sinais, da diminuição do tempo de chaveamento dos transistores ou da simplificação e/ou particionamento da lógica. O aumento de desempenho resultante desse tipo de otimização pode não ser proporcional ao aumento de frequência obtido pela aplicação da otimização; em alguns casos pode até mesmo ocorrer perda de desempenho. O resultado prático e direto do aumento da frequência é a realização de mais ciclos por segundo.

2. Fluxo de execução: as otimizações no fluxo de execução trazem como resultado a possibilidade de executar um maior número de operações por ciclo. Otimizações desse tipo são obtidas pelo emprego de técnicas de *pipelining* (escalar ou superescalar), previsão de desvios, execução fora de ordem, multithreading em hardware e diversas outras em geral aplicadas no nível microarquitetural.

3. Emprego de memórias cache: praticamente todo processador apresenta pelo menos um nível de memória cache. Como são mais rápidas que a memória principal, embora menores, permitem reduzir o impacto do acesso a dados e instruções.

Os primeiros microprocessadores que implementaram o modelo de von Neumann utilizavam um modelo baseado em ciclo único único de execução [Patterson and Hennessy 1997]. Neste modelo, toda e qualquer instrução deve ser executada em um único ciclo de *clock*. Para que isto seja possível, o tempo de ciclo deve ser determinado em função da instrução mais lenta, ou seja, da instrução que requer mais tempo para ser executada.

Como forma de tornar o processamento mais eficiente e aumentar o desempenho, surgiram as arquiteturas baseadas em múltiplos ciclos de execução. Com este modelo, foram introduzidos registradores intermediários (barreiras) que têm como finalidade separar fisicamente cada etapa do ciclo de instrução. Dessa forma, com uma unidade de controle mais complexa é possível determinar quais etapas são efetivamente necessárias para a execução de cada instrução. Cada etapa consome necessariamente um ciclo de *clock*, porém, cada instrução pode requerer um número diferente de etapas. Dessa forma, é possível executar somente as etapas necessárias para cada instrução, tornando o processamento mais eficiente.

A inserção de registradores intermediários como barreiras permitiu o isolamento das etapas, e deu abertura para a implementação de uma técnica conhecida como *pipelining*. Esta técnica, amplamente empregada nos dias de hoje, tem como objetivo sobrepor a execução de múltiplas instruções.

A implementação de *pipelining* (de instruções) pode seguir duas abordagens: escalar ou superescalar. A primeira permite a execução de múltiplas instruções, de forma que cada instrução ativa esteja em uma etapa diferente do ciclo de instrução. Em outras palavras, pode-se dizer que cada estágio pode receber somente uma instrução por vez. Como resultado, ao contrário da abordagem multiciclo, onde uma instrução leva necessariamente mais de um ciclo de *clock* para ser executada, na implementação com *pipeline* escalar pode-se chegar ao desempenho máximo de um ciclo por instrução, ou uma instrução por ciclo (máximo teórico) – em uma implementação multiciclo, o número de instruções por ciclo (IPC) é sempre menor do que um [Patterson and Hennessy 1997].

Historicamente, o aumento considerável na complexidade dos microprocessadores se deu devido a dois fatores: *(i)* disponibilidade de um número cada vez maior de transistores, e *(ii)* forte pressão do mercado para o oferecimento de microprocessadores mais rápidos – rapidez aqui denota desempenho e não somente frequência de *clock*. Seguindo essa tendência, surgiram então as arquiteturas superescalares. Na implementação de um *pipeline* superescalar, o objetivo é atingir IPCs maiores que um.

A única forma de alcançar esse objetivo é permitir a execução de múltiplas instruções por estágio de *pipeline*. Observe que a intenção é executar instruções efetivamente em paralelo. Nesse nível, o paralelismo é chamado de ILP (*Instruction Level Parallelism*), e acontece entre instruções de um mesmo thread.

Na prática, é necessária replicação de cada unidade interna do *pipeline* (estágio) de forma proporcional ao número de instruções que se pretende executar por ciclo. Em uma arquitetura superescalar, o IPC máximo (ideal) é proporcional ao número de instruções que podem ser executadas de forma paralela.

Duas observações importantes devem ser realizadas neste ponto. Primeiro, todos os objetivos em termos de desempenho, tanto das arquiteturas com *pipeline* escalar, como superescalar, estão limitados à resolução de dependências de dados, de controle e contenção de recursos. A existência de dependências entre instruções faz com que o processador não consiga preencher todas as unidades funcionais internas, resultando em desempenho abaixo do ideal (máximo teórico) [Wall 1991].

Segundo, os mecanismos para tratamento das dependências entre instruções adicionam complexidade significativa à unidade de controle e, para a exploração mais eficiente do ILP, são necessárias técnicas cada vez mais agressivas. No entanto, nas últimas gerações de microprocessadores observou-se que o custo destas implementações mais complexas não se justifica, devido ao fato de que o paralelismo existente nas aplicações é limitado. Portanto, os retornos em termos de desempenho não são proporcionais à complexidade adicionada e, não menos importante, as questões relacionadas ao consumo, custo e tempo de projeto passaram a receber mais atenção dos projetistas.

5.2.4. Arquiteturas paralelas

A idéia de usar múltiplos processadores com a finalidade de melhorar o desempenho e/ou a disponibilidade (redundância, tolerância a falhas) não é recente. Flynn [Flynn 1966, Flynn 1972], propôs uma taxonomia simples para categorização das arquiteturas baseando-se no número de fluxos de instruções (I) e de dados (D) observados. Os fluxos são caracterizados por serem únicos (*Single, ou S*) ou múltiplos (*Multiple, ou M*). Esta taxonomia, ainda atualmente empregada, é composta por quatro categorias: SISD, SIMD, MISD e MIMD.

As arquiteturas SISD apresentam um fluxo único de instruções operando sobre um fluxo único de dados (uniprocessadores). As arquiteturas SIMD apresentam fluxos adicionais de dados, sobre os quais as mesmas instruções são executadas em paralelo (processadores vetoriais e matriciais). As arquiteturas MISD possuem vários fluxos de instruções agindo sobre um mesmo fluxo de dados, não sendo encontrados comercialmente [Hennessy and Patterson 2007] (embora alguns autores caracterizem processadores *pipeline* como MISD). Finalmente, arquiteturas MIMD possuem múltiplos fluxos de instruções que executam sobre fluxos de dados. Nesta última categoria, encontram-se os multicores, os SMPs (*Symmetric Multi-Processors*), os SMTs (*Simultaneous Multi-Threaded*) e multicomputadores em geral.

Atualmente, o paralelismo é explorado em diferentes níveis em praticamente todas as arquiteturas. Assim, o paralelismo é explorado deste sua granulosidade mais fina, no nível de instrução em uniprocessadores, até a exploração do paralelismo no nível de threads em máquinas MIMD comunicando-se via troca de mensagens [De Rose and Navaux 2003]. Além disso, é importante também ressaltar que as formas de exploração de paralelismo podem ser combinadas. Por exemplo, um cluster pode ser implementado com base em processadores superescalares com instruções SIMD. Neste caso, o paralelismo de threads, dados e instruções é explorado em diferentes níveis do sistema.

A exploração do paralelismo no nível de instrução (ILP) em arquiteturas superescalares foi, durante muitos anos, foco das otimizações na maioria dos microprocessadores

de propósito geral [Santos and Santos 2005]. A razão está no fato de que o aumento de desempenho obtido é percebido no software de maneira transparente, ou seja, sem que este precise ser modificado. No entanto, esta técnica atingiu níveis de complexidade elevados e não apresenta mais os mesmos retornos em termos de desempenho.

5.2.5. Arquiteturas multicores

Ganhos consideráveis de desempenho foram obtidos pela exploração do paralelismo no nível de instrução em um único thread de controle. Estes ganhos foram obtidos com a aplicação de técnicas microarquiteturais complexas, como execução fora de ordem, previsão de desvios, renomeação de registradores, caches *on-chip*, *superpipelining*, entre outras. Estudos mais recentes e a própria experiência da indústria apontam que este caminho está saturado [Borkar 2004, Kongetira et al. 2005, Hennessy and Patterson 2007]. Além disso, consumo de energia e altas latências de acesso à memória têm representado obstáculos difíceis de serem transpostos, levando em consideração o desempenho de um único thread.

Durante anos foram discutidas alternativas e técnicas para aumento de desempenho dos processadores e, no final da década de 1990, parecia haver um consenso de que o foco das otimizações deveria voltar-se para a exploração do paralelismo de threads.

Naquele momento diversas implementações de processadores superescalares já estavam no mercado, por exemplo, Pentium III, AMD K6-III, UltraSparc I, Alpha 21264 e outros. Estes processadores implementam as complexas estruturas necessárias para explorar o paralelismo no nível de instrução, mas, em eficiência, estes estavam muito aquém do que poderiam alcançar.

A eficiência com relação ao desempenho ideal é reduzida devido às limitações impostas pelas dependências de dados e de controle que fazem com que os recursos sejam subutilizados. Uma alternativa para aumentar a eficiência seria inserir uma lógica capaz de gerenciar instruções provindas de múltiplos threads em execução. Um dos trabalhos pioneiros nesta direção foi realizado por Nemirovsky [Nemirovsky et al. 1991] e depois ampliado por Tullsen [Tullsen et al. 1995], que introduziu o termo SMT (*Simultaneous Multi-Threading*).

Um processador SMT é capaz de executar instruções em paralelo, tal como um processador superescalar, porém, pode simultaneamente extrair instruções de múltiplos threads de controle. Este modelo de computação chamou a atenção dos fabricantes, pois estes tinham conhecimento de que muitos dos recursos já existentes não estavam sendo utilizados eficientemente, em vista das limitações da exploração de paralelismo no interior de um único thread. O fator motivante teve por origem no fato de que estes recursos poderiam ser melhor utilizados, se houvesse outras fontes de onde se pudesse extrair instruções independentes.

Para implementar um processador SMT, no entanto, seria necessário garantir que instruções provenientes de um determinado thread não causariam nenhum efeito negativo para instruções de outros threads. A solução envolve replicar o banco de registradores e marcar as instruções, de forma que os resultados produzidos por instruções de um thread nunca se misturassem aos resultados de instruções dos demais.

Porém, este não era o único problema. A maior disponibilidade de instruções causava também uma maior pressão no sistema de memória, uma vez que a frequência de acessos à memória era maior. O termo SMT é também conhecido como *Hyperthreading*, e foi empregado pela Intel nos processadores Intel Pentium 4 HT, com duas threads. Houve poucas implementações de máquinas puramente SMT, mas alguns exemplos podem ser citados: IBM POWER5, MIPS MT e RMI MIPS SOC.

Embora as implementações SMT/HT superassem em parte o problema original (limitação de paralelismo ILP), a complexidade do processador em si continuava crescendo. Olukotun [Olukotun et al. 1996, Hammond et al. 1997] foi um dos pioneiros nas pesquisas relacionadas às arquiteturas multicores.

Por multicore entende-se todo processador que possui dois ou mais *cores*, ou seja, processadores completos no mesmo chip que trocam informações via uma memória compartilhada. Esta forma de TLP é frequentemente chamada também de CMP (*Chip-level Multiprocessing*).

5.2.6. Multicores e os novos desafios para o projeto de software concorrente

Uma série de vantagens dos chips multicores, tais como uso eficiente de energia, custo de desenvolvimento e as possibilidades de exploração do paralelismo de threads e processos [Hennessy and Patterson 2007] estão fazendo com que todos os fabricantes de microprocessadores de propósito geral apresentem as suas versões de CMP. No entanto, nem tudo é perfeito. O problema de eficiência das arquiteturas superescalares resulta das limitações do paralelismo de instruções, ou seja, da forma com que o software era descrito. Em alguns casos, o compilador pode ajudar, mas nem sempre é possível mudar drasticamente a natureza sequencial de algumas aplicações.

No caso das arquiteturas multicores, a eficência e melhor desempenho podem ser percebidas ou não dependendo do sistema operacional e, principalmente, da aplicação. É provável que um usuário comum perceba a diferença ao mudar para um processador de dois *cores*. No entanto, à medida em que o número de *cores* aumenta, fica mais difícil perceber melhorias no desempenho, se as aplicações não forem devidamente adaptadas para explorar e expor mais o paralelismo de threads. Considerando a evolução na tecnologia de fabricação de circuitos integrados, não demorará muito tempo para que um número bem maior de *cores* por chip esteja disponível. Exemplo é o processador UltraSparc T2, que possui oito *cores*, sendo que cada um é capaz de executar até oito threads. Outro exemplo é o processador Vega 2 da Azul Systems, que possui 48 *cores*, sendo que a máquina 3220 apresenta 4 chips ou seja, um total de 192 *cores* otimizados para aplicações Java.

Com estas perspectivas, pode-se dizer que mais uma vez as plataformas de desenvolvimento de software, bem como a cultura de desenvolvimento de software paralelo, estão bem atrás do potencial do hardware disponível.

5.3. Multiprogramação leve

Esta Seção introduz a multiprogramação leve. Inicialmente, são apresentados conceitos de processo leve e programa concorrente. A sequência se dá introduzindo as principais facilidades de programação oferecidas por ferramentas de programação multithread, e

caracteriza os principais modelos para implantação de programas concorrentes. A seção finaliza com uma discussão sobre erros comuns observados na programação concorrente.

5.3.1. Noção de thread

Um programa escrito em uma linguagem imperativa sequencial, ao ser lançado, é manipulado pelo sistema operacional sob a forma de um processo. Este processo é formado por um fluxo de execução, também chamado thread, uma área de memória e um registro descritor de processo (ou PCB, para o termo em inglês *Process Control Block*). O fluxo executa sequencialmente as instruções definidas pelo programa. Complementa este fluxo uma área de memória própria, manipulada como uma pilha, na qual os dados temporários são amazenados e que contém o registro da invocação de funções e o estado dos registradores. A memória abriga os dados manipulados pelo programa e o próprio código em execução. O PCB contém informações relevantes ao processo, como prioridade, ocupação de memória e manipuladores de arquivos em uso. O grupo de processos em execução simultânea é gerenciado pelo sistema operacional, o qual é responsável pelo escalonamento e pelo gerenciamento dos recursos disponíveis.

O lançamento de um programa multithread também implica na criação de um processo. A diferença se dá que, ao invés de apresentar um único fluxo de execução, verifica-se a existência de múltiplos threads, cada thread executando uma determinada sequência de instruções, possuindo sua própria pilha e a manutenção do estado dos registradores. Não raro o número de threads ativos varia durante a execução do programa. Também de forma semelhante ao thread único do processo sequencial, cada thread necessita de escalonamento para ocupar o processador. Neste ponto, se apresenta uma característica fundamental de thread, que lhe vale o usual sinônimo de *processo leve* em muitos textos da literatura. Uma vez que a quantidade de informação de contexto relativa a um thread é menor que a quantidade de informação associada de um processo, o custo de manipulação de um thread pelas ações do sistema operacional é menor que para um processo convencional [Tanenbaum and Woodhull 2006].

5.3.2. Programa concorrente

Um programa é a solução computacional para uma aplicação. Programas são normalmente vistos como implementações de algoritmos que, ao executar, tendo como entrada um conjunto de dados, os transformam, produzindo como resultado final um novo conjunto de dados. Embora esta visão também se aplique a programas concorrentes, em uma análise mais detalhada podem ser identificados dois níveis nos algoritmos destes programas. Em um dos níveis encontra-se a resolução algorítmica da aplicação propriamente dita. Este nível é composto por *tarefas*, sendo que cada tarefa tem por objetivo oferecer a solução computacional para parte do problema da aplicação. De forma análoga a um programa, cada tarefa possui um conjunto de dados de entrada e, no final de sua computação, produz resultados. Os resultados produzidos por uma tarefa podem servir como entrada para outras tarefas ou, ao final da execução, comporão parte da solução final. Uma propriedade particular destas tarefas é que elas podem ser submetidas à execução concorrente. O segundo nível propõe a estratégia de controle de evolução do programa. Em outras palavras, este nível oferece o algoritmo que coordena a execução das tarefas, tendo como atribuição principal garantir a correta ordem de execução das tarefas e comunicação de

dados entre estas, de forma a atingir o resultado final esperado.

Embora em muitas implementações o código para estes dois níveis encontre-se mesclado, o programador precisa ter consciência de que ele possui dois problemas para resolver. O primeiro diz respeito à implementação da solução do problema definido pela aplicação. O segundo, ao uso das técnicas de programação concorrente para exploração eficiente de um determinado hardware paralelo. Outro aspecto a ser considerado é que, embora um programa possa contar com diversas tarefas aptas a ser executadas em um determinado instante de tempo, apenas um subconjunto destas estarão executando simultaneamente. A concorrência expressa o paralelismo potencial da aplicação, mas a execução paralela, de fato, depende da quantidade de recursos disponíveis. Por exemplo, em um computador com quatro processadores haverá, no máximo, quatro tarefas executando paralelamente.

No contexto da multiprogramação leve para arquiteturas multiprocessadas, a natureza concorrente do programa é reflexo do número de tarefas que o compõem. Na maioria dos casos, as aplicações contêm um grau de paralelismo muito maior que aquele suportado por um determinado hardware paralelo. Assim, as tarefas, quando em execução, competem pelos recursos disponíveis, em particular por tempo de acesso aos processadores. No entanto, ressalta-se que estas tarefas compõem partes de um algoritmo maior, cujo objetivo é realizar a transformação de dados, de forma a atingir o resultado final. Nesta perspectiva, as tarefas possuem uma relação de colaboração, na qual dados armazenados em memória devem ser compartilhados, e sobre os quais cada tarefa opera seu algoritmo de transformação próprio. Assim, a concorrência entre tarefas reflete, tanto a competição pelo acesso aos recursos de hardware, quanto a disputa no acesso aos dados compartilhados. O controle da primeira forma de concorrência é normalmente delegado ao sistema operacional, que, munido de suas estratégias de escalonamento, garante acesso aos recursos disponíveis pelas partes componentes do programa. A concorrência no acesso aos dados, por sua vez, é a estratégia empregada no nível de controle do algoritmo. Esta segunda forma de concorrência deve ainda ser entendida pelo programador, não tanto pela idéia de disputa, mas sim de coordenação, pois as tarefas necessitam acessar dados para colaborar entre si, comunicando resultados de suas computações.

5.3.3. Recursos de programação

Na multiprogramação leve, os recursos de programação são voltados, basicamente, às operações de controle dos fluxos de execução. Estas operações envolvem desde a criação de novos fluxos até bloqueio temporário da execução de um fluxo. De alguma forma, todas estas operações envolvem a sincronização entre diferentes fluxos. Estas sincronizações têm por objetivo fornecer aos fluxos de execução as informações sobre o estado da execução de outros fluxos.

As operações de sincronização permitem que fluxos de execução publiquem informações sobre seus estados de execução, ou façam leituras sobre os estados de execução de outros fluxos. Estas sincronizações permitem introduzir a coordenação dos threads no acesso aos dados compartilhados. A própria operação de criação de thread pode ser considerada uma sincronização, pois ela indica que o thread original produziu e armazenou na memória compartilhada dados que podem ser lidos pelo thread criado. No entanto,

outros operadores garantem outras formas de sincronização.

Talvez o mais elementar dos recursos de sincronização disponíveis seja a *barreira*. Uma barreira define um ponto no código de um grupo de threads. Cada thread deste grupo, ao alcançar este ponto, suspende sua execução, até que todos os threads do grupo tenham alcançado este ponto. Outro mecanismo similar é o operador *join*, que suspende a execução do thread que o executou, até que um determinado thread, especificado como parâmetro deste operador, termine sua execução.

Outros recursos populares de programação permitem controle de acesso a seções críticas e regulagem de avanço de fluxos pelo uso de semáforos e variáveis de condição. Uma *seção crítica* corresponde a um trecho de código do corpo de um thread que acessa pelo menos um dado na memória compartilhada, dado este cujo acesso também pode ser requisitado por instruções executando em outro thread. A seção é dita crítica pela necessidade de cuidados para manuntenção da integridade do dado manipulado. Para tanto, deve ser garantida exclusividade no acesso aos dados ao referido trecho de código, sob pena de ocorrer incoerência no acesso aos dados pela manipulação simultânea deste por dois ou mais threads. O mecanismo normalmente utilizado para controle de seções críticas é o *mutex*, acrônimo para a expressão em inglês *mutual exclusion*, exclusão mútua. De forma similar ao mutex, também conhecido como semáforo binário, operam os semáforos.

É importante observar que tanto mutex como semáforo oferecem duas operações de sincronização: uma indicando a intenção de um fluxo de execução entrar em um determinado trecho de código, e a segunda informando a sua saída. É importante que estas duas operações sejam sempre realizadas de forma que as seções de código de fluxo controlado sejam monitoradas, e as execuções dos diferentes threads ocorram de forma coordenada.

Com uma concepção um pouco diferente se apresenta o mecanismo de *variável de condição*. Uma variável de condição permite o controle do avanço de threads sobre determinados pontos do programa, considerando o valor de um dado presente na aplicação. Na prática, a condição é considerada *satisfeita* ou *não satisfeita* considerando o valor deste dado no momento em que a condição é avaliada. Caso a condição seja considerada não satisfeita, o mecanismo de variável de condição é empregado para suspender a execução de um fluxo, para que este aguarde a sinalização da mudança de valor do dado. Este sinal é enviado quando um outro thread modificar o valor do referido dado.

5.3.4. Modelos de concorrência

A onipresença de sistemas computacionais requer agilidade nos processos de desenvolvimento de software. A comunidade de desenvolvimento de software, em particular sua parcela ligada ao paradigma de orientação a objeto, já há alguns anos emprega técnicas de padrões de projeto (*design patterns*). Na computação paralela, padrões de programação também ocorrem. Embora haja esforço de pesquisa em padrões de projeto para computação paralela, a atenção é aqui dedicada a padrões recorrentes encontrados para representar a decomposição das tarefas em aplicações. São diversos os padrões encontrados [Skillicorn and Talia 1998], mas uma discussão sobre as estruturas mais gerais, representadas pelo paralelismo de tarefas, paralelismo de dados, fluxo de dados e tarefas comunicantes, permite oferecer uma visão bastante clara das questões de projeto de um código concorrente.

O *paralelismo de tarefas* é bastante empregado em aplicações onde a concorrência é definida em termos de atividades que atuam sobre conjuntos disjuntos de dados. O esforço de programação está relacionado ao controle das trocas de informações entre estas atividades concorrentes. Aplicações ditas trivialmente paralelizáveis são um dos casos onde o paralelismo de tarefas torna-se interessante. Estas aplicações são caracterizadas por possuírem reduzidas, ou mesmo inexistentes, necessidades de trocas de dados entre as atividades em execução. Desta forma, a concepção de algoritmos concorrentes, em geral, é simplificada pela reduzida necessidade de sincronizações entre as atividades.

A principal propriedade que uma aplicação deve apresentar para ser implementada segundo o modelo de paralelismo de dados é possuir uma grande coleção de dados, devendo os elementos deste conjunto sofrer, de forma independente, uma série de manipulações. Bons exemplos são encontrados em aplicações que manipulam vetores e matrizes de grandes dimensões. As abstrações de programação permitem gerar e controlar a concorrência na manipulação dos elementos destas estruturas de dados.

O modelo de *fluxo de dados* representa a execução de um programa sob a forma de um grafo dirigido. Neste grafo, os vértices correspondem às tarefas, e as arestas às dependências de dados (comunicações) entre estas tarefas. Nesta representação, as arestas são dirigidas, partindo da tarefa que produz um dado em direção à tarefa que o consome. A mais conhecida implementação do modelo de fluxo de dados segue o esquema de linha de produção, conhecido como *pipeline*, embora esteja presente em implementações de algoritmos mais complexos, como aplicações da programação dinâmica [Gusfield 1997].

Finalmente, o modelo de *tarefas comunicantes* é composto por tarefas e canais de comunicação, sendo também representado por um grafo. Como no modelo de fluxo de dados, os vértices deste grafo representam tarefas. As arestas, porém, não são dirigidas e representam canais de comunicação entre tarefas. Esta forma de comunicação indica dependência mútua das tarefas por alguma informação. Em uma implementação multithread é comum o uso desta estrutura, particularmente considerando que canais de comunicação podem ser representados por dados armazenados em memória.

Embora os modelos de paralelização tenham sido apresentados de forma independente, é comum que implementações reais sejam compostas por combinações destes, ou mesmo por uma variante de uma estrutura básica.

5.3.5. Cuidados com a programação

Além de descrever sua aplicação em termos de um programa concorrente, o programador deve ainda atentar para não cometer erros de programação. No uso dos recursos de programação multithread, os erros mais comuns são as condições de corrida e situações de impasse.

As condições de corrida (*data-races*) são situações em que não há a devida sincronização dos threads para executar instruções em uma seção crítica. Nestes casos, as instruções de duas ou mais seções críticas podem ser executadas de forma intercalada, e não em regime de exclusão mútua. A falha refletida por tal erro de programação é a inconsistência no valor de um dado compartilhado, pois não há garantias de como os acessos serão realizados.

Uma situação de impasse (*deadlock*) ocorre quando um thread permanece bloqueado aguardando um sinal – que nunca é enviado – de outro thread. Um erro de programação típico é um thread informar sua entrada em uma seção crítica e não informar sua saída. A falha que ocorre é suspensão da execução de parte do programa.

Outro cuidado que o programador deve ter é de não presumir ordem de execução dos threads. Embora esquemas de prioridade sejam especificados em diferentes interfaces de programação, não é confiável delegar a credibilidade do programa desenvolvido a características de implementação de suportes de execução. No mínimo, o programador teria o ônus de verificar a compatibilidade de seu programa a cada nova versão do suporte utilizado. Na prática, seria comum verificar em um mesmo suporte comportamentos anômalos de execução.

5.4. Ferramentas de programação

Existem diversas ferramentas para programação multithread. Dentre estas, nesta Seção são apresentadas Pthreads, .NET Framework e OpenMP. Estas ferramentas são hoje bastante populares nos seus próprios nichos de aplicação: Pthreads é bastante popular na construção de sistemas de softwares básicos, em particular em ambientes UNIX-like; .NET, por sua vez, define uma interface de programação portável para diferentes linguagens em diversos ambientes; e OpenMP abre a possibilidade de exploração de processamento de alto desempenho em aplicações com grão fino de concorrência.

Dadas as restrições de espaço e o escopo deste curso, esta Seção se limita a uma visão geral destas ferramentas.

5.4.1. Threads POSIX – Pthreads

O padrão POSIX (Portable Operating System Interface for uniX), proposto pela IEEE, define um conjunto de interfaces de programação para o sistema UNIX. O padrão para threads (IEEE POSIX 1003.1c de 1995) é normalmente referenciado como Pthreads [Nichols et al. 1998, Lewis and Berg 1998]. Nos sistemas UNIX-like, Pthreads são disponibilizadas na forma de uma biblioteca. O padrão caracteriza a interface de serviços para uso conjunto com programas C/C++, embora existam esforços de padronização desta interface para a linguagem FORTRAN.

A importância de Pthreads está intimamente associada à abrangência deste padrão. Como o padrão POSIX encontra-se disponível na maioria dos sistemas operacionais, seu potencial de portabilidade é bastante alto. Uma das principais características desta interface de programação multithread é que ela oferece grande liberdade ao programador para implementação de seu programa. Como resultado, diversas estruturas de programa concorrente podem ser empregadas.

Nesta seção considera-se a utilização de Pthreads em ambiente GNU/Linux. Nestes ambientes, o pacote Pthreads conta com dois arquivos principais: o arquivo de *header* e a biblioteca de funções. O arquivo de header, `pthread.h`, deve ser incluido no programa-fonte que utiliza os serviços da biblioteca. A biblioteca propriamente dita deve ser ligada ao programa do usuário, o que pode ser feito passando o parâmetro `-lpthread` para o compilador GCC.

5.4.1.1. Criação e destruição de threads

Em Pthreads, o lançamento de um novo thread é associado à execução da sequência de instruções definidas no corpo de uma função. Esta função deve receber como parâmetro um endereço para uma área de memória, correspondendo ao local onde encontram-se os parâmetros de entrada da função, retornando outro endereço de memória, onde encontram-se os resultados produzidos pela função. O retorno deve encontrar-se no heap ou alocados globalmente, porque variáveis alocadas automaticamente pela função na pilha são descartadas no fim da execução da função. Ambos os endereços são do tipo `void`.

A criação e término de threads são realizadas de forma explícita. A invocação da primitiva `pthread_create` por um thread implica a criação de um novo fluxo. Um thread termina sua execução quando executar toda a sequência de instruções definida em seu corpo, ou quando executar o comando `return` convencional de C – alternativamente, invocando o serviço `pthread_exit`. A assinatura destas funções é dada por:

```
void *func( void *args );
int pthread_create( pthread_t *thid, pthread_attr_t *atrib,
                    void *(*func) (void *), void *args );
int pthread_join( pthread thid, void **ret );
int pthread_exit( void *retval );
```

Em `pthread_create`, o novo thread a ser criado deve executar o código especificado pela função `func`. Este thread poderá ser referenciado futuramente por meio do valor retornado no parâmetro `thid`. Pthread permite ainda aplicar atributos de execução à nova thread. Estes atributos são passados à biblioteca pelo parâmetro `atrib`. O inteiro retornado corresponde a um código de erro, com zero normalmente associado ao sucesso na execução.

O padrão não define quando o novo thread será disparado, tão pouco como se dará a sobreposição do tempo de uso do processador por este novo thread com os threads já em curso de execução. Em contrapartida, há o serviço `pthread_join`, que sincroniza um thread com o thread identificado por `thid`, garantindo que um thread continue apenas quando o outro thread tenha terminado sua execução. Adicionalmente, o parâmetro `ret` retorna o endereço de memória que contém os dados retornados pelo thread sincronizado quando este terminar. Caso o thread sincronizado já tenha terminado, ocorre simplesmente a recuperação do retorno. Caso contrário, o thread que necessita a sincronização permanecerá bloqueado até que esta seja satisfeita.

Cada thread pode sofrer apenas uma operação de *join*. Caso o retorno do thread seja útil a dois ou mais pontos do programa, o programador deve usar estratégias de programação próprias. Outro cuidado a ser tomado é em relação à utilização da primitiva `pthread_exit` em programas C++: a execução desta primitiva termina o fluxo de execução corrente, mas não invoca os destrutores dos objetos armazenados na área de memória local ao thread.

Um exemplo da utilização deste primeiro conjunto de primitivas é apresentado a seguir. Neste exemplo, uma função `oiMundo` é executada por um thread. O programa principal, função `main`, cria o thread e aguarda seu término.

```
void *oiMundo( void *str ) {
  printf("Oi do thread %s",
         (char *) str );
  free(str);
  return NULL;
}
```

```
void main() {
  pthread_t thid;
  for( int i = 0 ; i < 5 ; i++ )
   if( !pthread_create(&thid, NULL,
                       oiMundo, itoa(i)) ){
     printf( "Ocorreu um erro!!!\n" );
     exit(EXIT_FAILURE);
   }
  pthread_join( thid, NULL );
}
```

Neste exemplo, foi utilizado o valor `NULL` para os *atributos* do thread criado, fazendo com que assumam os valores *default*. Os atributos desejados são descritos em uma variável do tipo `pthread_attr_t`. O padrão POSIX para threads define um conjunto de serviços para manipular variáveis deste tipo, entre eles:

```
int pthread_attr_init(pthread_attr_t *atrib );
int pthread_attr_destroy(pthread_attr_t *atrib );
int pthread_attr_setdetachstate(pthread_attr_t *atrib, int val);
int pthread_attr_setschedpolicy(pthread_attr_t *atrib, int val);
int pthread_attr_setstacksize(pthread_attr_t *atrib, size_t val);
```

Enquanto o serviço `init` inicializa uma variável com os valores *default* de atributos, o serviço `destroy` torna uma variável de atributo inutilizável. Os demais serviços `set*` permitem configurar diferentes propriedades para atributos de threads. O primeiro apresentado `setdetachstate` pode receber dois valores: `PTHREAD_CREATE_JOINABLE`, para indicar que o thread deve sofrer uma sincronização por *join* ou `PTHREAD_CREATE_DETACHED`, indicando o contrário.

5.4.1.2. Sincronização

A comunicação entre threads não se limita apenas aos parâmetros de entrada e do retorno da função executada por um thread. A própria memória do processo serve de base de comunicação, sendo que o acesso à memória se dá pela simples execução de instruções de escrita e leitura. O problema é garantir o correto acesso às informações pelos threads, evitando condições de corrida e situações de impasse. A biblioteca Pthreads oferece os recursos de mutex e variável de condição.

Em Pthreads o mutex é oferecido pelo tipo `pthread_mutex_t`. Instâncias deste tipo são usualmente chamadas de *mutexes* e são manipuladas pelos seguintes serviços:

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

De forma análoga ao que ocorre com os atributos de thread, anteriormente descritos, os serviços `init` e `destroy` preparam para o uso e inutilizam um mutex. O atributo default de inicialização é indicado por `NULL`, fazendo com que inicialmente o mutex esteja disponível. Os serviços `lock` e `unlock` permitem manipular o mutex. A invocação do serviço `lock` indica o desejo do thread corrente em executar instruções em uma seção crítica. A invocação ao `unlock` indica a saída do thread de uma seção crítica. Enquanto a operação do serviço `unlock` é simples, apenas liberando o mutex, a operação de aquisição pode resultar em um thread bloqueado, caso outro thread já tenha adquirido o mutex. A seguir, um exemplo do uso de mutex para sincronizar dois threads é apresentado.

```
int x;   //variável global
pthread_mutex_t m; // m é um mutex associado à variável x
pthread_mutex_init( &x, NULL );

// Thread A                          // Thread B
pthread_mutex_lock( &m );            pthread_mutex_lock( &m );
a = x;                               b = x;
a = a + 1;                           b = b - 1;
x = a;                               x = b;
pthread_mutex_unlock( &m );          pthread_mutex_unlock( &m );
```

Observe que o mutex `m` é uma variável de acesso global, e seu uso depende da lógica adotada no algoritmo implementado. No exemplo, o algoritmo associa o mutex `m` à variável `x`. Cada seção crítica que necessita acesso a `x` deve explicitar as operações `lock` e `unlock`. Caso existissem outras variáveis compartilhadas, outros mutex poderiam ser criados e manipulados dentro do programa. O serviço `trylock`, por sua vez, não é bloqueante. Ele é utilizado para adquirir um mutex, caso este esteja disponível.

Outro mecanismo disponível na biblioteca Pthreads é a *variável de condição*. Este recurso permite controlar o avanço de threads sobre seções críticas, considerando o valor de algum dos dados manipulados pelo algoritmo da aplicação. Os serviços disponíveis são os seguintes:

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

Como mostram as assinaturas das funções, variáveis de condição também são manipuladas com apoio de variáveis construídas a partir de um tipo de dado especial: `pthread_cond_t`. Os serviços `init` e `destroy` permitem inicializar (o valor default de atributo é `NULL`) e inutilizar uma variável deste tipo. A invocação do serviço `wait` por um thread faz com que este suspenda sua execução, até que receba um sinal indicando que a condição foi satisfeita. Os serviços `signal` e `broadcast` podem ser executados por threads para sinalizar que uma condição encontra-se satisfeita – `signal` envia o sinal de condição satisfeita a apenas um dos threads suspensos na condição indicada; `broadcast` sinaliza todos os threads suspensos na variável de condição.

Note-se, no entanto, que variáveis de condição são utilizadas para coordenar acesso a uma seção crítica. Assim, seu uso é associado, necessariamente, a um mutex. Isto é caracterizado no parâmetro do tipo mutex indicado no serviço `wait`. Segue um exemplo de um produtor/consumidor utilizando variáveis de condição para sincronizar a produção e o consumo de itens em um *buffer*.

```
// Área de memória compartilhada entre o Produtor e o Consumidor
Buffer b;            // Buffer de armazenamento temporário
int nb_itens = 0;    // Contador de itens no buffer
pthread_mutex_t mb;  // Proteção do buffer e do contador
pthread_cond_t c;    // Sincronização entre produtor e consumidor
void Produtor() {                    void Consumidor() {
 Item it;                             Item it;
 for( ; ; ) {                         for( ; ; ) {
   it = ProduzItem();                   pthreads_mutex_lock(&mb);
   pthreads_mutex_lock(&mb);            while(nb_item <= 0)
   ArmazenaBuffer(b, it);                 pthread_cond_wait(&c, &mb);
   nb_itens++;                          it = LeBuffer();
   pthread_cond_signal(&c);             nb_itens--;
   pthreads_mutex_unlock(&mb);          pthreads_mutex_unlock(&mb);
 }                                    }
}                                    }
```

É importante observar que uma sinalização em uma variável de condição não é memorizada, ou seja, somente os threads em estado de *wait* recebem o sinal. Outro aspecto importante a observar é que tanto o serviço `wait` como `signal` (ou `broadcast`) devem ser executados *dentro* das seções críticas que manipulam os dados compartilhados. Não existe possibilidade de ocorrer uma situação de impasse, pois a implementação do `wait`, na biblioteca, faz com que a primeira ação executada seja a liberação do mutex recebido como segundo parâmetro e, assim que o sinal de condição satisfeita for recebido, faz com que o thread entre novamente na disputa pelo mutex.

Outro mecanismo de sincronização de fluxos de execução são os semáforos. O padrão Pthreads não define serviços para sua utilização, sendo sua especificação apresentada na Seção IEEE 1003.1b. No entanto é comum que serviços para manipulá-los sejam disponibilizados. Por exemplo, no GNU/Linux semáforos são disponibilizados pela inclusão de `semaphore.h`. Alguns dos serviços disponibilizados são:

```
int sem_init( sem_t *sem, int comp, unsigned int val);
int sem_destroy( sem_t *sem );
int sem_wait( sem_t *sem );
int sem_post( sem_t *sem );
```

A manipulação de semáforos se dá com apoio de variáveis do tipo `sem_t`. Enquanto o serviço `destroy` inutiliza o uso de um semáforo, `init` permite inicializá-lo: `val` deve informar o valor inicial do semáforo e `comp` permite indicar se o uso do semáforo será restrito aos threads do processo corrente ou se deverá operar entre processos. O valor zero para `comp` indica uso local. As operações `wait` e `post` correspondem, respectivamente, às tradicionais operações *P* e *V* de semáforos. O algoritmo básico para um semáforo pode ser encontrado

Há ainda outros mecanismos de sincronização disponíveis, tais como barreiras, os quais não são apresentados neste texto.

5.4.2. .NET Framework

Em 2002, foi lançado pela Microsoft o CLR (*Common Language Runtime*), um ambiente de execução desenvolvido para permitir uma plataforma de execução única para diferentes linguagens. Similar a Java, este ambiente opera como uma máquina virtual, oferecendo uniformidade de operação. Junto com CLR, foi também apresentado um conjunto de recursos de programação conhecidos como .NET Framework[4] [Thai and Lam 2003]. Os recursos de .NET podem ser utilizados em diversas linguagens, como C#, C++, Visual Basic .NET e JScript.

.NET propõe uma interface de programação de alto nível, adotando características da programação orientada a objetos. A popularidade desta interface de programação pode ser associada ao grau de abstração proposto, embora a compatibilidade da interface entre as linguagens suportadas pelo CLR não deva ser desconsiderada. Outro fator ainda a ser considerado é o grande número de computadores que rodam sob o sistema operacional da Microsoft, e a implementação multiplataforma do ambiente Mono[5].

[4] `http://msdn2.microsoft.com/en-us/library/aa139615.aspx`
[5] `http://www.mono-project.com`

Neste texto, os exemplos .NET foram apresentados em C#. Para informar que utilizará os serviços de thread em c#, deve-se incluir o pacote `System.Threading`.

5.4.2.1. Criação e destruição de threads

O corpo do código de um thread deve ser especificado em um método estático de uma classe qualquer. A manipulação de um fluxo de execução se dá com auxílio de um objeto da classe `Thread`. Um objeto desta classe é criado explicitamente pela invocação do serviço `ThreadStart`. Este serviço necessita como parâmetro um *delegate* para o método a ser executado pelo thread. O termo *delegate* segue a terminologia adotada por .NET, podendo ser considerado equivalente ao endereço do método. Por exemplo:

```
public class Exemplo {
 public static void corpoThread() { ... }
}
...
Thread t = new Thread( new ThreadStart( corpoThread ) );
```

A criação de um objeto da classe `Thread` não implica na ativação de um novo fluxo de execução. O lançamento do novo thread deve ser realizado de forma explícita, com invocação do método `Start` no objeto `Thread`. De forma equivalente, o método `Join` pode ser empregado para sincronizar a execução de um thread com o término de outro. Com a sobrecarga de método, o método `Join` também aceita um valor (inteiro) indicando o tempo em milisegundos que o thread corrente pode aguardar a sincronização. Caso a sincronização seja satisfeita é retornado o valor `true`, caso contrário `false` .

Uma outra possibilidade em C#/.NET é a de interromper, pela invocação do método `Abort`, a execução de um thread. A chamada deste método implica o envio de um sinal que pode ser capturado pelo thread, possibilitando o abandono de sua execução. Este recurso faz uso do mecanismo de tratamento de exceções, conforme ilustra o seguinte exemplo:

```
public class Exemplo {
 public static void umThread() {
  try {
   ... código do thread
  } catch( ThreadAbortException e ) {
      //término induzido
  } finally{ //término padrão
  }
```

```
 public static void Main() {
  Thread t = new
    Thread(new ThreadStart( umThread ));
  t.Start();
  Thread.Sleep(1000);
  t.Abort();
  t.Join();
 }
}
```

Além do método `Abort`, dois outros métodos podem ter aplicações interessantes na manipulação de threads: `Suspend` e `Resume`. Com estes métodos, é possível interromper e retomar a execução de threads.

5.4.2.2. Sincronização

Em C#, o recurso mais simples, mas nem por isso pouco eficiente, é a primitiva `lock`, operando sobre um objeto. O caso típico de uso é não permitir acesso simultâneo de dois ou mais métodos sobre um atributo de um objeto. Um exemplo desta situação é dado no trecho de código seguinte, onde a primitiva `lock` é aplicada sobre um objeto `ctrLock` para proteger a região crítica onde o saldo é incrementado.

```
public void deposita( int val ) {
  lock( ctrLock ) {saldo = saldo + val; }
}
```

Uma possibilidade seria utilizar a própria referência `this` já existente para o objeto ao invés de criar um objeto especialmente para este fim. No entanto, sabendo que o objeto é público, a criação de um atributo privado permite maior robustez no código.

As operações atômicas em C# usam os serviços da classe `Interlocked`. Desta classe, `Increment` e `Decrement` permitem incrementar e decrementar um valor armazenado em uma variável inteira. O método `Exchange` permite a inicialização atômica de uma variável. O exemplo na sequência manipula o atributo inteiro `contador`.

```
Interlocked.Exchange( ref contador, 0 ); // contador recebe 0
...
Interlocked.Increment( ref contador ); // Conta uma passagem
...
Interlocked.Decrement( ref contador ); // Subtrai uma passagem
```

Os monitores, por sua vez, são bastante similiares, em sua estrutura, ao mecanismo oferecido por `lock`, sendo que o programador deve usar os métodos estáticos oferecidos na classe `Monitor`. Monitores garantem execução de seções críticas em regime de exclusão mútua com apoio de um objeto válido. A entrada em uma seção crítica é requisitada pela invocação do método `Enter` e a saída informada pela invocação do método `Exit`. Também é possível utilizar o serviço não-bloqueante `TryEnter`, que retorna o valor booleano `false`, caso algum outro thread detenha o direito de uso do monitor. O exemplo de uso do monitor apresentado manipula um objeto `ctrMonitor` para controlar o acesso às seções críticas.

```
Monitor.Enter( ctrMonitor );
try { ... }
finally { Monitor.Exit( ctrMonitor ); }
```

O uso das cláusulas `try/finally` não são obrigatórias, no entanto elas aumentam a robustez do código, garantindo que, na ocorrência de alguma exceção que impossibilite a execução do código na seção crítica, o monitor será liberado. Este procedimento previne eventuais situações de impasse.

Outro recurso disponível para manipulação de monitores é a liberação temporária do direito de execução sobre a seção crítica. A invocação ao método `Wait`, após ter sido adquirido o direito de uso do monitor, faz com que o thread corrente *libere* o acesso ao monitor em uso e, imediatamente, entre na disputa por ele novamente. Utilizar este serviço favorece a implementação de algoritmos em que ocorrem constantes trocas de dados entre os threads.

O mecanismo de .NET para mutex possui uma estrutura diferente do empregado nos monitores. Um mutex é manipulado por meio de um objeto *mutex*, instanciado a partir da classe `Mutex`. Os métodos de tentativa de acesso às seções críticas e de saída são, respectivamente, `WaitOne` e `ReleaseMutex`. Um exemplo de uso de mutex é dado como segue. Como o exemplo caracteriza, o mutex `m` é um objeto estático, de forma a ser compartilhado por todas as instâncias da classe `Contador`.

```
class Contador {
  private static Mutex m = new Mutex();
  private static int contador;
  public void passagem() {
    m.WaitOne();
    try { contador = contador + 1; }
    finally { m.RealeaseMutex(); }
  }
}
```

Semáforos em .NET são bastante semelhantes ao mutex, sendo disponibilizados pela classe `Semaphore`. A diferença básica é que um objeto mutex é binário, e um semáforo pode ter qualquer valor positivo. Assim, a sincronização garante a passagem de um conjunto de threads, de acordo com o valor inteiro positivo do semáforo. O valor inicial é dado quando da criação do objeto, e os métodos `WaitOne` e `Release` o manipulam. `WaitOne` é bloqueante caso o valor associado ao semáforo seja igual a zero – isto significa que já foi atingido o número permitido de passagens de threads especificados pelo semáforo. Caso contrário, o valor do semáforo é decrementado e o thread prossegue sua execução. O método `Realease` incrementa o valor associado ao semáforo, permitindo que um outro thread adquira direito de passagem.

5.4.3. OpenMP – Open Multi-Processing

OpenMP [Chandra et al. 2001] é um acrônimo para *Open Multi-Processing*[6]. Em 1997, OpenMP foi proposto como uma interface de programação multithread para a linguagem FORTRAN. Atualmente, além de FORTRAN, está também disponível na linguagem C/C++ em diferentes plataformas UNIX-like e Windows. Esta interface de programação é fortemente apoiada por processos de compilação. Programas OpenMP são desenvolvidos principalmente por meio de diretivas de compilação, podendo ainda incorporar chamadas a serviços de biblioteca em tempo de execução. Outra forma de interação possível é via variáveis de ambiente.

O grande apelo desta plataforma é seu alto grau de portabilidade. Para tanto, colabora o fato de diversos compiladores incluírem mecanismos de apoio a esta interface. Somado a isto, a facilidade de paralelização de código também tem papel crescente na sua popularidade. Os recursos de programação disponibilizados permitem incorporar trechos paralelos a programas sequenciais de forma incremental, sem alterar o algoritmo da aplicação.

Dentre as variáveis de ambientes utilizadas para influenciar a execução de programas OpenMP, `OMP_NUM_THREADS` determina o número de threads default para executar trechos de código em regiões paralelas. Outras variáveis de ambiente permitem determinar as demais condições de execução, como estratégia de escalonamento de threads e tamanho da pilha de dados dos threads.

Para compilar programas OpenMP nas versões mais atuais de GCC, usa-se a flag `-fopenmp`, enquanto que o compilador Intel necessita do parâmetro `-openmp`.

5.4.3.1. Criação e destruição de threads

A estrutura de programas OpenMP segue o modelo *fork*/*join*. Uma operação *fork* identifica, implícita ou explicitamente, atividades a ser executadas concorrentemente. A ope-

[6] `http://www.openmp.org`

ração *join* sincroniza o término da execução de todas as atividades disparadas por um fork. As atividades concorrentes definidas pelo programa não são threads de fato, mas sim unidades de trabalho a serem executadas por threads de serviço, implementados no ambiente de execução. No entanto, neste texto, o termo thread é utilizado como sinônimo de atividade concorrente, de forma a manter a nomenclatura compatível com a empregada pelas demais ferramentas vistas.

O número de atividades concorrentes efetivamente instanciadas depende, tanto do código a ser executado, quanto de decisões de escalonamento tomadas pelo ambiente de execução. A estratégia considera, entre outros critérios, o número de threads de serviço disponíveis, identificado na variável de ambiente `OMP_NUM_THREADS`, e a execução inicia atribuindo a um destes threads a responsabilidade da execução da função `main`. Este thread de serviço é identificado como *mestre* do *time* de threads de serviço.

A interface de programação de OpenMP emprega diretivas de compilação (*pragmas*) como abstração para as operações *fork*/*join*. Chamadas a serviços de biblioteca permitem interação do programa em execução com o ambiente de execução.

As diretivas de compilação OpenMP são compostas por sentinelas, diretivas e cláusulas:

```
#pragma omp { diretiva [cláusula] }
 corpo
```

Nesta sintaxe, `#pragma omp` é chamado *sentinela*, sendo dependente da linguagem de programação utilizada. A *diretiva* OpenMP é identificada em `diretiva`, opcionalmente acompanhada de uma ou mais *cláusulas*. O corpo para os threads é definido em `corpo`, composto por um bloco de instruções ou por um comando de iteração por bloco. Nas instruções do bloco não podem ser incluídas instruções de salto (`goto`) para qualquer posição do programa, mesmo dentro da própria região paralela. As operações fork e join encontram-se implícitas nesta estrutura: o fork é especificado na própria linha onde o pragma se encontra e o join no final do bloco que especifica o corpo das atividades concorrentes. Um primeiro exemplo é dado pela utilização da diretiva `sections`:

```
#pragma omp sections
 {
#pragma omp section
     {
       foo(x);
     }
#pragma omp section
     {
       bar(x);
     }
 }
```

Neste exemplo, foram definidas duas seções, cada uma identificada por um bloco específico. Estas duas seções definem o corpo para a execução de dois threads do programa, o primeiro executa a função identificada por `foo` e a segunda a função `bar`. O conjunto dos dados de entrada dos threads criados é a memória acessível ao thread corrente, no exemplo representado pela variável inteira `x`. O comportamento padrão define que o acesso à memória é compartilhado pelos fluxos criados. Assim, os threads criados acessam a mesma instância de `x`, podendo ocorrer situações de corrida caso algum dos threads acesse `x` em escrita.

Outro aspecto interessante é que as execuções de todas as seções paralelas sincronizam no seu final de forma implícita, ou seja, o trecho de código que segue a definição de seções paralelas somente será habilitado a executar quando todos os threads completarem suas respectivas execuções.

Outra diretiva para expressar paralelismo é `parallel`. Esta diretiva implica a criação de threads, que deverão ser executados pelos threads de serviço do ambiente de execução. Como o número de threads de serviço corresponde ao valor associado à variável de ambiente `OMP_NUM_THREADS`, o número de threads em execução concorrente será dependente deste número. Assim, se esta variável possuir o valor `4`, quatro threads de serviço executarão, de forma paralela, quatro threads do programa de aplicação. O uso desta diretiva é exemplificado por:

```
int x = 313, y = 617;
#pragma omp parallel private(x)
  x = y + 1;
```

Caso `OMP_NUM_THREADS` = `4`, quatro instâncias do código especificado serão executadas. Tal como ocorre nas seções paralelas, o final da região paralela determina uma barreira para sincronização entre os threads.

No trecho de código apresentado também é exemplificada a *comunicação de dados* entre os threads, no caso as variáveis inteiras `x` e `y`. O comportamento-padrão implica que todos os dados locais ao thread original sejam acessíveis aos threads criados. No exemplo, esta situação é observada pelo uso da variável inteira `y`. O comportamento-padrão é alterado para uso da variável `x`, onde a cláusula `private` indica que cópias locais devem ser instanciadas para cada thread, sendo que o valor inicial de `x` local é *indefinido*. Caso seja necessário, é possível *herdar* o último valor de uma variável antes do disparo dos threads, fazendo uso da cláusula `firstprivate`. O comportamento-padrão, compartilhamento de instância de dados, pode ser identificado pelo uso da cláusula `shared`.

No sentido inverso, dos threads criados para o thread original, podem ser aplicadas duas outras cláusulas: `lastprivate` e `reduce`. A primeira faz com que o valor computado pelo último thread do programa seja o valor utilizado pelo thread original após a sincronização. Uma variável pode ser, ao mesmo tempo, `first` e `lastprivate`. A cláusula `reduce` é mais elaborada, permitindo a combinação de resultados parciais dos threads ao final. Esta combinação pode ser realizada aplicando um dos operadores disponíveis, aritméticos ou lógicos . O uso da cláusula `reduce` é exemplificado na sequência. Neste exemplo, a variável inteira `soma` é inicializada com `100` e são criados quatro threads, cada uma produzindo o valor `1`. No final, o valor acumulado corresponde a `104`, ou seja, o valor inicial de `soma` adicionado de uma unidade para cada thread executado. Note-se o uso de `set_num_threads` para definir o número de threads a serem criados.

```
int soma = 100;
 omp_set_num_threads(4);
#pragma omp parallel reduction(+:soma)
  soma += 1;
```

Outro importante recurso disponível em OpenMP é a possibilidade de realizar a paralelização de laços. A diretiva empregada é `parallel for`. O código a seguir apresenta o uso desta primitiva para acumular os valores de um vetor.

```
int vet[100], i, somatorio = 0;
#pragma omp parallel for
 for( i = 0 ; i < 100 ; i++ ) vet[i] = i;

#pragma omp parallel for reduction(+:somatorio)
 for( i = 0 ; i < 100 ; i++ ) somatorio += vet[i];
```

Algumas regras devem ser seguidas para construção do laço a ser paralelizado. A variável de iteração necessita ser do tipo inteiro com sinal (`int` em C/C++). Da mesma forma, devem ser valores inteiros com sinal o teste da condição de término do laço e o valor a ser adicionado ou subtraído da variável de iteração a cada iteração realizada. Estes dois valores não podem variar durante toda execução do laço. O teste de continuação pode empregar, apenas, os seguintes operadores relacionais: `<`, `<=`, `>` e `>=`, incrementando ou decrementando, conforme o caso, a variável de iteração a cada iteração. Outro aspecto importante é que o corpo do laço possui apenas uma entrada e uma saída – o comando `break` de C/C++ não pode, portanto, ser utilizado.

Embora de aparência e utilização simples, na prática a paralelização de laços não é uma tarefa trivial. Um conjunto de regras deve ser seguido para eficiência na utilização deste recurso. O programador deve evitar situações nas quais a passagem por uma iteração qualquer dependa da execução de uma passagem anterior, como no caso onde a computação da passagem i depende dos resultados da computação da passagem $i-1$. A discussão de estratégias para detecção e remoção de dependências entre instruções foge ao escopo deste texto, sendo aconselhado ao leitor buscar maiores informações sobre o assunto em literatura especializada, como [Chandra et al. 2001].

5.4.3.2. Sincronização

O substrato para comunicação de dados entre os threads providos por OpenMP é o espaço de endereçamento único provido pela memória compartilhada. Não são oferecidos mecanismos de troca de dados via parâmetros e retorno de funções, como em Pthreads, nem abstrações como monitores e objetos como em C#/.Net. Mecanismos de controle explícito do fluxo de execução podem ser realizados por meio de diretivas e cláusulas específicas para coordenar a execução dos threads ativos.

O mais básico destes recursos é oferecido pela diretiva `barrier`. O uso desta primitiva implica a criação de um ponto de sincronização entre todos os threads criados a partir de uma diretiva `parallel` comum. Este mecanismo garante que, no escopo da execução de cada thread, a instrução seguinte à barreira somente será executada após todos os threads participantes terem satisfeito a condição de sincronização, ou seja, terem atingido a barreira.

Um mecanismo mais elaborado que a barreira é possível com o uso da diretiva `critical`. O uso desta diretiva permite controlar a execução de blocos de comandos em um regime de exclusão mútua. Considere o código apresentado na sequência. Este código encontra-se inserido em uma região paralela, e tem por objetivo manter a variável compartilhada `maior` atualizada com o maior valor encontrado em um vetor.

```
int vet[100], i, maior, lmaior
  maior = vet[0];
#pragma omp parallel for private(lmaior)
  for( i = 0 ; i < 100 ; i++ ) {
#pragma omp critical
    lmaior = maior;
    if( lmaior < vet[i] ) lmaior = vet[i];
  }
#pragma omp critical
  if( lmaior > maior ) maior = lmaior;
}
```

A barreira implementada pela diretiva `critical` tem abrangência global, ou seja, é aplicada a todos os threads executando coletivamente. Para evitar contenção desnecessária, é possível *nomear* seções críticas, identificando claramente o recurso a ser utilizado em exclusão mútua. Assim, no exemplo anterior, poderia ser informado `critical(MAXVAL)` em ambos os pragmas, identificando, na lógica do programa, que o recurso compartilhado é a variável global `maior`.

Em algumas situações, dentro de uma região paralela, pode ser necessário executar um trecho de código apenas por um dos threads ativos. Um exemplo típico é dado pelo envio de saídas para o console (impressão de mensagens na tela). A diretiva `single` permite definir um trecho de código que deve ser executado apenas por um dos threads.

O código na sequência exemplifica o uso da diretiva `single` considerando duas situações: na primeira, todos os threads permanecem bloqueados aguardando que o usuário seja informado que diferentes threads serão criados para execução de `Trabalho`; na segunda, o usuário é informado que a execução de uma destas instâncias foi concluída com auxílio da cláusula `nowait`. Esta última relaxa a condição de sincronização dos threads, permitindo que threads avancem sem a necessidade de sincronização no final do bloco `single`.

```
#pragma omp parallel
 {
  #pragma omp single
    print("Iniciando %d instancias de Trabalho.\n",
          omp_get_num_threads());

  Trabalho();

  #pragma omp single nowait
    print("Uma instancia de Trabalho foi terminada.\n");
}
```

O padrão não define, para a diretiva `single`, qual thread é responsável pela execução do referido trecho de código. É garantido apenas que o código será executado uma única vez, e que todos os threads devem permanecer bloqueados no final do corpo do código da seção `single` aguardando seu término, exceto se for utilizada a cláusula `nowait`. Opcionalmente a diretiva `master` pode ser utilizada. Esta diretiva garante que o código será executado pelo thread mestre de um time.

5.5. Prática de programação

Nesta Seção, são apresentados alguns aspectos que devem ser considerados quando se usa ferramentas de multiprogramação leve. Em particular, são destacados aspectos relacionados ao desempenho – embora nem todas as implementações sejam focadas em processamento de alto desempenho, é desejável evitar onerar a execução de aplicações com implementações mal projetadas.

Inicialmente, esta Seção apresenta os principais cuidados a serem tomados no momento do projeto de um programa multithread. A sequência aborda questões associadas ao desempenho, tais como estratégias para exploração da hierarquia de memória disponível em arquiteturas multicores e associação de afinidade de threads a processadores.

5.5.1. Modelagem da concorrência

Quando se trata de programação concorrente, a primeira preocupação do programador deve ser identificar, na sua aplicação, quais são as atividades concorrentes que podem ser definidas e quais as dependências de dados existentes entre estas. Esta etapa visa realizar a *decomposição* paralela do programa. De posse desta informação, é possível caracterizar a decomposição como *funcional* ou de *dados*.

A *decomposição funcional* consiste em mapear as atividades concorrentes de um programa em unidades de execução independentes (os threads). Normalmente estas atividades concorrentes são implementadas como uma função, podendo receber e retornar dados, como parâmetros ou retorno de resultados.

A *decomposição de dados*, por outro lado, explora a propriedade de fraco acoplamento de dados que certas aplicações possuem, e permite dividir um dado de entrada, o domínio inicial, em um conjunto de domínios independentes, atribuindo cada subdomínio a uma computação passível de execução paralela por um thread. O resultado final é obtido com a agregação dos resultados parciais obtidos a cada computação realizada. Esta forma de decomposição é, normalmente, bem representativa em compiladores paralelizantes e em OpenMP.

O programador deve também estar atento para identificar as duas formas de paralelismo em uma determinada aplicação. Assim, em um primeiro momento, o programa pode exprimir atividades concorrentes caracterizadas pela decomposição funcional da aplicação. Internamente, cada atividade concorrente pode ainda determinar etapas de sua computação, que podem ser executadas segundo o modelo de decomposição de dados.

Outro aspecto fundamental é a determinação da *granularidade* da aplicação. A granularidade expressa a relação entre a quantidade de cálculo e o número de sincronizações efetuadas por um programa paralelo em execução. O tamanho do grão pode ser facilmente entendido como a quantidade de computação existente entre duas sincronizações (em termos médios).

Por fim, deve-se considerar a portabilidade do programa não apenas em relação ao seu código. Deve também ser considerada a sua *portabilidade de desempenho*. A portabilidade de código diz respeito à possibilidade de migrar código entre duas plataformas distintas. Já a portabilidade de desempenho está relacionada ao potencial de escalabilidade de execução do programa. Em outras palavras, é a propriedade que o programa possui de ter seu desempenho de execução melhorado à medida em que aumenta o poder de processamento paralelo de uma determinada arquitetura. As linguagens e bibliotecas apresentadas neste texto possuem um bom grau de portabilidade, permitindo a execução em ambientes UNIX-like e Windows através de diferentes implementações.

Portabilidade com o sentido de escalabilidade é uma nova questão que tem cada vez mais despertado atenção dos desenvolvedores de programas paralelos. Atualmente,

processadores multicores dispõem de dois ou quatro *cores* de processamento. No entanto, o histórico na evolução dos sistemas computacionais permite inferir que este número irá crescer, podendo facilmente atingir dezenas de *cores*. A implicação imediata é que diferentes configurações de plataforma estarão disponíveis, com até 80 (!) *cores*.

5.5.2. Alinhamento de variáveis na memória

Para minimizar o impacto do tempo de acesso à memória principal no tempo total de processamento de programas, as arquiteturas de processadores modernos fazem uso de hierarquias de memória com caches. A exploração da localidade de referência de dados e instruções permite diminuir o número de acessos à memória principal, concentrando acessos nas caches, mais rápidas e mais próximas do processador. Estratégias de programação podem ser empregadas para melhorar o aproveitamento das caches. Em ambientes multiprocessados como os processadores multicores, o impacto do emprego ou não de tais estratégias no desempenho é ainda maior. As técnicas mais comuns visam aumentar a localidade de referência de dados em instruções próximas.

Uma técnica bastante simples consiste em observar como os elementos de um arranjo multidimensional, como uma matriz, são alocados em memória, e fazer com que laços aninhados percorram este arranjo explorando a vizinhança entre os dados. Por exemplo, em C/C++ os elementos de uma matriz bidimensional são ordenados por linha. Assim, em C/C++, o elemento da primeira coluna de uma linha i qualquer ocupa a posição de memória seguinte à ocupada pelo elemento da última coluna da linha $i-1$. Dessa forma, sendo necessário percorrer todos os elementos desta matriz, é mais interessante organizar o acesso a cada linha, percorrendo todas as suas colunas, conforme o seguinte código:

```
int m[LINHAS][COLUNAS];
for( int i = 0 ; i < LINHAS ; i++ )
  for( int j = 0 ; j < COLUNAS ; j++ )
   m[i][j] = foo(i,j);
```

Outra estratégia para explorar a localidade de dados é menos evidente ao programador, considerado o mecanismo empregado pelo processador para alinhar dados em memória. Este alinhamento é definido pela arquitetura do processador, de forma a otimizar o desempenho de sua operação. Um endereço de variável de um tipo que possua n bytes é dito *alinhado* caso n seja potência de dois e o endereço seja múltiplo de n. Dessa forma, uma variável do tipo `char` é considerada alinhada em qualquer posição de memória, e uma variável `int` somente nas posições de memória múltiplas de quatro. Apesar do ganho de desempenho na manipulação dos dados, podem ocorrer espaços vazios de memória, o que pode ser ilustrado pela execução do seguinte código:

```
char x;
int y;
printf("&y-&x = %d\n", (int)&x - (int)&y );
```

No caso do código apresentado, é pouco provável que a diferença entre os endereços corresponda a um (1) byte – isto aconteceria em situações onde a variável `x` pudesse ser alocada em uma posição imediatamente anterior a um endereço alinhado para `y`. Como experiência, substitua a linha `char x;` por `int x;` e depois por `char a, b, c, x;`.

Nestes casos, a diferença entre os endereços de `x` e `y` são quatro bytes – o tamanho necessário para armazenar o inteiro `y` em um processador de 32 bits.

Como regra geral, o alinhamento dos dados deve ser observado para não disperdiçar espaços de memória. Em particular deve ser evitado intercalar a definição de variáveis de tipos de dados diferentes, principalmente quando o tamanho em bytes dos seus respectivos tipos de dados diferir substancialmente.

5.5.3. Afinidade de processador

Em arquiteturas multiprocessadas e multicores, uma outra estratégia para exploração da memória cache é a associação de *afinidade* de threads a processadores. Esta estratégia pode ser explorada quando o programador dispõe de informações para realizar o balanceamento de carga de sua aplicação. Suponha-se um chip com dois cores e um programa onde quatro threads disputam recursos de processamento. Neste programa, dois threads são dedicados a cálculo intensivo e dois a operações de entrada e saída. Um escalonamento satisfatório deveria atribuir um thread de cálculo e um thread de entrada e saída a cada um dos cores disponíveis.

Importante observar que, caso um programador incorpore no seu código instruções para definir afinidades de threads a processadores, significa que ele está retirando do mecanismo de escalonamento do sistema operacional a atribuição de definir em qual processador um determinado thread deverá executar. Isto também caracteriza o uso de recursos de programação de baixo nível, acessando diretamente serviços providos pelo sistema operacional. Atualmente, OpenMP não incorpora facilidades para definição de afinidades de threads.

Em .NET, é possível atribuir afinidade de todos os threads de um processo a um processador pelo método `set_ProcessorAffinity`, do pacote `Process`. Este método recebe como parâmetro um inteiro, cujo valor representa uma *máscara* com a qual é representada a afinidade estabelecidada entre o thread e processadores. O valor para a máscara depende do número de processadores reais disponíveis. Cada bit nesta máscara corresponde a um processador, devendo ser colocado com valor 1 o bit correspondendo ao processador habilitado a executar os threads do processo. Dessa forma, o valor default é $2^n - 1$, onde n é o número de processadores do sistema. Considerando um processador com quatro *cores*, o valor 0x0001 habilita o *core* 1 a executar os threads, enquanto o valor 0x0003 habilita os *cores* 1 e 2. O método `get_ProcessorAffinity` permite obter a afinidade determinada para os threads do processo corrente.

Em ambiente GNU/Linux, a chamada de sistema `sched_setaffinity` permite registrar a afinidade, enquanto que `sched_getaffinity` consulta a afinidade atualmente associada. No caso do GNU/Linux, estes serviços são aplicados individualmente a threads, não ao conjunto de threads de um processo. Processos criados com a chamada de sistema `fork` herdam do processo-pai sua afinidade.

5.5.4. Redução de contenção e spin locks

Durante a execução de um programa multithread, o uso de mecanismos de sincronização como barreiras e seções críticas restringe o paralelismo de execução. Em outras palavras, a execução de uma primitiva de sincronização implica, de uma forma ou outra, seriali-

zação de instruções pertencentes a dois fluxos de execução distintos. Caso um programa introduza um grande número de sincronizações, o efeito que pode ser observado é a contenção de execução: além de introduzir tempo de processamento extra no processamento da sincronização propriamente dita (não raro envolvendo chamadas de sistema), pode ocorrer o bloqueio de diversos threads pela espera de um mesmo recurso. Como resultado, observa-se um baixo potencial de portabilidade. As alternativas que se apresentam tendem a incorporar nas implementações soluções algorítmicas que eliminem a necessidade de sincronizações ou, pelo menos, diminuam o impacto destas no tempo total de processamento.

Uma prática de programação bastante empregada é reduzir ao máximo o número de instruções nas seções críticas, de forma a minimizar o tempo de espera – e, consequentemente, o número de threads na fila de espera – pelo uso de um determinado recurso. O uso desta técnica, no entanto, pode implicar adição de sobrecustos extras de processamento, não somente em função da frequência de requisição e liberação de recursos, mas também do esforço extra do mecanismo de escalonamento em trocas de contextos a cada requisição bem/mal sucedida de recursos de sincronização. Como alternativa, aponta-se o uso de *spin locks*.

Spin locks são mecanismos para controle de acesso a seções críticas no qual o thread invocador da operação *lock* não libera o processador, caso o recurso solicitado não esteja disponível. A idéia é evitar a troca de contexto do thread solicitante, apostando que o tempo de espera pelo recurso não seja muito longo. A biblioteca Pthreads oferece este recurso (padrão POSIX 1003.1j):

```
int pthread_spin_init( pthread_spinlock_t *lock, int pshared );
int pthread_spin_destroy( pthread_spinlock_t *lock );
int pthread_spin_lock( pthread_spinlock_t *lock );
int pthread_spin_trylock( pthread_spinlock_t *lock );
int pthread_spin_unlock( pthread_spinlock_t *lock );
```

A interface de serviços oferecida é bastante semelhante à empregada para uso de mutex convencional. Destaca-se o parâmetro `pshared` na inicialização de um *spin lock*, podendo indicar que o escopo a ser empregado é limitado ao contexto do processo corrente (valor `PTHREAD_PROCESS_PRIVATE`) ou compartilhado entre processos (`PTHREAD_PROCESS_SHARED`).

É importante observar que o uso de *spin lock* pode favorecer a obtenção de desempenho. No entanto, o tempo de espera no laço deve ser calibrado de acordo com o sistema em que o programa encontra-se executando: uma espera muito curta pode implicar em fracasso na tentativa de obter o recurso e uma consequente troca de contexto torna-se inevitável; já uma espera muito longa pode comprometer a estratégia. Outro ponto é a relação entre o número de threads em execução e o número de processadores disponíveis. Se esta relação indicar que o número de processadores é baixo, o uso deste recurso torna-se desinteressante: em um caso extremo, com um único processador, não é interessante usar *spin locks*.

Outras técnicas podem ser utilizadas, tais como o uso de instruções especiais que executam de forma atômica. No entanto, a portabilidade dos programas fica comprometida.

5.6. Considerações finais

A programação concorrente e ferramentas para programação multithread são temas obrigatórios nos cursos de graduação na área de Computação. Com a recente popularização de processadores multicore, observa-se a ampliação do interesse nestes assuntos, uma vez que o processamento paralelo passa a ser quase uma necessidade para exploração eficiente de tais arquiteturas. A demanda atual é por utilização consciente dos conceitos associados à multiprogramação leve para obter das arquiteturas multicore o máximo de sua capacidade.

A arquitetura multicore representa um dos mais recentes avanços na tecnologia de implementação de arquiteturas de processadores. Diferente das técnicas que a precederam, processadores multicore explicitam seu potencial para execução paralela na forma de várias unidades de processamento independentes. Como consequência, no desenvolvimento de um programa para este tipo de arquitetura, deve ser prevista a execução concorrente de atividades. Neste contexto, a multiprogramação leve se apresenta como uma importante aliada do programador, oferecendo recursos de programação orientados a arquiteturas multiprocessadas.

Além de apresentar a arquitetura dos processadores multicore, pontuando aspectos históricos que marcaram a evolução tecnológica em direção a esta solução, o texto apresentou uma visão geral da multiprogramação leve, considerando seu modelo de programação e execução. A maior parte do capítulo foi dedicada à prática da programação, sendo apresentadas três ferramentas de programação multithread: Pthreads, .NET Framework e OpenMP. Estas ferramentas foram selecionadas tendo em vista critérios como popularidade e representação em diferentes domínios de aplicação, considerando diferentes necessidades das aplicações e dos ambientes de desenvolvimento.

Por fim, foram apresentadas questões específicas de desenvolvimento de programas multithread em processadores multicore. Em particular, foram caracterizados aspectos relacionados à exploração eficiente dos recursos de processamento disponíveis e estratégias de programação para reduzir contenção de execução dos threads.

Referências bibliográficas

[Akhter and Robert 2006] Akhter, S. e Robert, J. (2006). *Multi-Core Programming: Increasing Performance through Software Multithreading*. Intel Press, Hillsboro.

[Andrews 2000] Andrews, G. R. (2000). *Foundations of multithreaded, parallel, and distributed programming*. Addison-Wesley, Reading.

[Backus 1978] Backus, J. (1978). Can programming be liberated from the neumann style? A functional style and its algebra of programs. *ACM Turing Award Lecture*, 21(8):613–641.

[Borkar 2004] Borkar, S. (2004). Microarchitecture and design challenges for gigascale integration. In *MICRO 37*, p. 3, Washington, DC, USA. IEEE CS.

[Burger and Goodman 1997] Burger, D. e Goodman, J. R. (1997). Billion-transistor architectures - guest editors' introduction. *IEEE Computer*, 30(9):46–49.

[Chandra et al. 2001] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., e Menon, R. (2001). *Parallel Programming in OpenMP*. Morgan Kaufmann, San Francisco.

[De Rose and Navaux 2003] De Rose, C. e Navaux, P. O. A. (2003). *Arquiteturas Paralelas*. Sagra–Luzzato, Porto Alegre.

[Flynn 1966] Flynn, M. J. (1966). Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909.

[Flynn 1972] Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Trans. on Computing*, C-21(9):948–960.

[Gusfield 1997] Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. CUP.

[Hammond et al. 1997] Hammond, L., Nayfeh, B. A., e Olukotun, K. (1997). A single-chip multiprocessor. *Computer*, 30(9):79–85.

[Hennessy and Patterson 2007] Hennessy, J. L. e Patterson, D. A. (2007). *Computer architecture (4th ed.): a quantitative approach*. Morgan Kaufmann, San Francisco.

[Koch 2005] Koch, G. (2005). Discovering multi-core: Extending the benefits of moore law. *Technology@Intel Magazine*.

[Kongetira et al. 2005] Kongetira, P., Aingaran, K., e Olukotun, K. (2005). Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29.

[Lewis and Berg 1998] Lewis, B. e Berg, D. J. (1998). *Multithreaded programming with Pthreads*. Sun Microsystems / Prentice Hall, New Jersey.

[Nemirovsky et al. 1991] Nemirovsky, M. D., Brewer, F., e Wood, R. C. (1991). Disc: dynamic instruction stream computer. In *MICRO'91*, p. 163–171, New York. ACM Press.

[Nichols et al. 1998] Nichols, B., Buttar, D., e Farrell, J. P. (1998). *Pthreads Programming*. O'Reilly, Cambridge, 2 edition.

[Olukotun et al. 1996] Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K., e Chang, K. (1996). The case for a single-chip multiprocessor. In *VII ASPLOS*, p. 2–11, New York. ACM Press.

[Patterson and Hennessy 1997] Patterson, D. A. e Hennessy, J. L. (1997). *Computer Organization and Design: the hardware/software interface*. Morgan Kaufmann, San Francisco.

[Santos and Santos 2005] Santos, T. G. S. e Santos, R. R. (2005). *Projeto e Implementação de Arquiteturas Superescalares*, cap. 3. SBC, Porto Alegre.

[Skillicorn and Talia 1998] Skillicorn, D. B. e Talia, D. (1998). Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169.

[Tanenbaum and Woodhull 2006] Tanenbaum, A. S. e Woodhull, A. S. (2006). *Operating Systems Design and Implementation,*. Prentice Hall, Essex.

[Thai and Lam 2003] Thai, T. L. e Lam, H. Q. (2003). *.NET Framework Essential*. O'Reilly, Cambridge.

[Tullsen et al. 1995] Tullsen, D. M., Eggers, S., e Levy, H. M. (1995). Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of the 22th Annual Intl. Symp. on Computer Architecture*, p. 392–403.

[von Neumann 1945] von Neumann, J. (1945). *First Draft of a Report on the EDVAC*. Moore School of Electrical Engineering, Univ. of Pennsylvania.

[Wall 1991] Wall, D. W. (1991). Limits of instruction-level parallelism. In *Proc. of the 4th Intl. Conf. on Architectural Support for Programming Languages and Operating System (ASPLOS)*, vol. 26, p. 176–189, New York, NY. ACM Press.