

7

*Existem, no mar, peixes melhores
do que os que já foram pescados.*

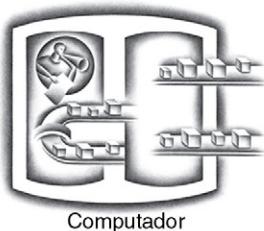
Provérbio irlandês

Multicores, multiprocessadores e clusters

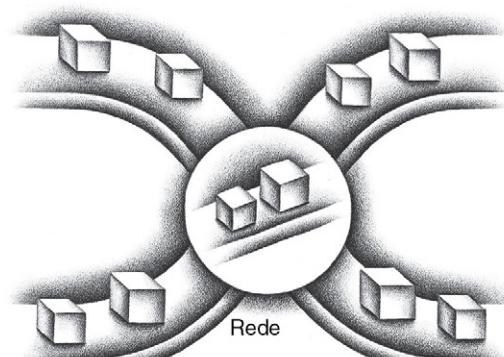
- 7.1 **Introdução** 510
- 7.2 **A dificuldade de criar programas
com processamento paralelo** 512
- 7.3 **Multiprocessadores de memória
compartilhada** 515
- 7.4 **Clusters e outros multiprocessadores
de passagem de mensagens** 517

- 7.5 Multithreading do hardware** 521
7.6 SISD, MIMD, SIMD, SPMD e vetor 524
7.7 Introdução às unidades de processamento de gráficos 528
7.8 Introdução às topologias de rede multiprocessador 534
7.9 Benchmarks de multiprocessador 537
7.10 Roofline: um modelo de desempenho simples 539
7.11 Vida real: benchmarking de quatro multicores usando o modelo roofline 546
7.12 Falácia e armadilhas 552
7.13 Comentários finais 553
7.14 Perspectiva histórica e leitura adicional 555
7.15 Exercícios 555

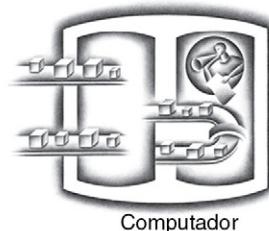
Organização de multiprocessador ou cluster



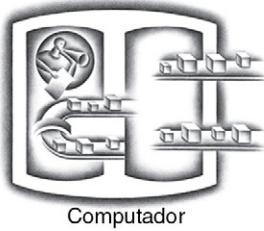
Computador



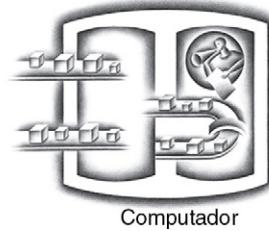
Rede



Computador



Computador



Computador

“Sobre as montanhas da lua, pelo vale das sombras, cavalgue, cavalgue corajosamente.” Respondeu a sombra: “Se você procurar o Eldorado!”

Edgar Allan Poe, “Eldorado”, stanza 4, 1849

multiprocessador Um sistema de computador com pelo menos dois processadores. Isso é o contrário do **processador**, que tem apenas um.

paralelismo em nível de tarefa ou paralelismo em nível de processo Utilizar vários processadores executando programas independentes simultaneamente.

programa de processamento paralelo Um único programa que é executado em vários processadores simultaneamente.

cluster Um conjunto de computadores conectados por uma rede local (LAN) que funciona como um único e grande multiprocessador.

microprocessador multicore Um microprocessador contendo vários processadores (“cores”) em um único circuito integrado.

7.1

Introdução

Há muito tempo, os arquitetos de computadores têm buscado o Eldorado do projeto de computadores: criar computadores poderosos simplesmente conectando muitos computadores menores existentes. Essa visão dourada é a origem dos **multiprocessadores**. O cliente pede tantos processadores quantos seu orçamento permitir e recebe uma quantidade correspondente de desempenho. Portanto, o software para multiprocessadores precisa ser projetado para trabalhar com um número variável de processadores. Como dissemos no Capítulo 1, a potência tornou-se o fator limitante para centros de dados e microprocessadores. Substituir grandes processadores ineficazes por muitos processadores eficazes e menores pode oferecer melhor desempenho por watt ou por joule tanto no grande quanto no pequeno, se o software puder utilizá-los com eficiência. Assim, a melhor eficiência de potência se junta ao desempenho escalável no caso para os multiprocessadores.

Como o software multiprocessador é escalável, alguns projetos podem suportar operar mesmo com a ocorrência de quebras no hardware; ou seja, se um único processador falhar em um multiprocessador com n processadores, o sistema fornece serviço continuado com $n - 1$ processadores. Portanto, os multiprocessadores também podem melhorar a disponibilidade (veja Capítulo 6).

Alto desempenho pode significar alta vazão para tarefas independentes, chamado **paralelismo em nível de tarefa**, ou **paralelismo em nível de processo**. Essas tarefas paralelas são aplicações independentes, e são um uso importante e comum dos computadores paralelos. Essa técnica é contrária à execução de uma única tarefa em vários processadores. Usamos o termo **programa de processamento paralelo** para indicar um único programa que é executado em vários processadores simultaneamente.

Há muito tempo existem problemas científicos que precisam de computadores muito mais rápidos, e essa classe de problemas tem sido usada para justificar muitos computadores paralelos novos no decorrer das últimas décadas. Veremos vários deles neste capítulo. Alguns desses problemas podem ser tratados de forma simples, usando um **cluster** composto de microprocessadores abrigados em muitos servidores ou PCs independentes. Além disso, os clusters podem servir a aplicações igualmente exigentes fora das ciências, como mecanismos de busca, servidores Web e bancos de dados.

Como dissemos no Capítulo 1, os multiprocessadores ganharam destaque porque o problema da potência significa que aumentos futuros no desempenho aparentemente virão de mais processadores por chip em vez de taxas de clock mais altas e CPI melhorado. Eles são chamados **microprocessadores multicore** e não microprocessadores multiprocessador, provavelmente para evitar redundância de nomeação. Logo, os processadores normalmente são chamados **cores** em um chip multicore. O número de cores deverá dobrar a cada dois anos. Assim, os programadores que se preocupam com o desempenho precisam se tornar programadores paralelos, pois programas sequenciais significam programas lentos.

O grande desafio enfrentado pela indústria é criar hardware e software que facilite a escrita de programas de processamento paralelo, que sejam eficientes no desempenho e potência à medida que o número de cores por chip aumenta geometricamente.

Essa mudança repentina no projeto do microprocessador apanhou muitos de surpresa, de modo que ainda existe muita confusão sobre a terminologia e o que ela significa. A **Figura 7.1** tenta esclarecer os termos serial, paralelo, sequencial e concorrente. As colunas dessa figura representam o software, que é inherentemente sequencial ou concorrente. As linhas da figura representam o hardware, que é serial ou paralelo. Por exemplo, os programadores de compiladores pensam neles como programas sequenciais: as etapas são análise léxica, parsing, geração de código, otimização e assim por diante. Ao contrário, os programadores de sistemas operacionais normalmente pensam neles como programas concorrentes: processos em cooperação tratando de eventos de E/S devido a tarefas independentes executando em um computador.

		Software	
		Seqüencial	Concorrente
Hardware	Serial	Multiplicação de matriz escrita em MatLab executando em um Intel Pentium 4	Sistema operacional Windows Vista executando em um Intel Pentium 4
	Paralelo	Multiplicação de matriz escrita em MatLab executando em um Intel Xeon e5345 (Clovertown)	Sistema operacional Windows Vista executando em um Intel Xeon e5345 (Clovertown)

FIGURA 7.1 Categorização e exemplos de hardware/software do ponto de vista da concorrência e do paralelismo.

O motivo desses dois eixos da [Figura 7.1](#) é que o software concorrente pode ser executado no hardware serial, como os sistemas operacionais para o processador Intel Pentium 4, ou no hardware paralelo, como um SO no mais recente Intel Xeon e5345 (Clovertown). O mesmo acontece para o software sequencial. Por exemplo, o programador MATLAB escreve uma multiplicação de matriz pensando nela sequencialmente, mas poderia executá-la serialmente no hardware do Pentium 4 ou em paralelo no hardware do Xeon e5345. Você poderia supor que o único desafio da revolução paralela é descobrir como fazer com que o software naturalmente sequencial tenha alto desempenho no hardware paralelo, mas também fazer com que os programas concorrentes tenham alto desempenho nos multiprocessadores, à medida que o número de processadores aumenta. Com essa distinção, no restante deste capítulo usaremos *programa de processamento paralelo* ou *software paralelo* para indicar o software sequencial ou concorrente executando em hardware paralelo.

A próxima seção descreve por que é difícil criar programas eficientes para processamento paralelo. As Seções 7.3 e 7.4 descrevem as duas alternativas de uma característica fundamental do hardware paralelo, que é se todos os processadores nos sistemas contam ou não com um único endereço físico. As duas versões comuns dessas alternativas são chamadas *microprocessadores de memória compartilhada* e *clusters*. A Seção 7.5 descreve então o *multithreading*, um termo que geralmente é confundido com multiprocessamento, em parte porque se baseia em uma concorrência semelhante nos programas. A Seção 7.6 descreve um esquema de classificação mais antigo que na [Figura 7.1](#). Além disso, ela descreve dois estilos de arquiteturas de conjunto de instruções que dão suporte à execução de aplicações sequenciais em hardware paralelo, a saber, *SIMD* e *votor*. A Seção 7.7 descreve um estilo de computador relativamente novo, da comunidade de hardware gráfico, chamado *unidade de processamento de gráficos* (GPU — Graphics Processing Unit). O Apêndice A descreve as GPUs com mais detalhes. Em seguida, discutimos a dificuldade de se achar benchmarks paralelos, na Seção 7.9. Essa seção é seguida por uma descrição de um modelo de desempenho novo e simples, porém introspectivo, que ajuda no projeto de aplicações e também de arquiteturas. Na Seção 7.11, usamos esse modelo para avaliar quatro computadores multicore recentes em dois kernels de aplicação. Terminamos com falácias e armadilhas e nossas conclusões sobre o paralelismo.

Antes de prosseguirmos ainda mais até o paralelismo, não se esqueça das nossas incursões iniciais dos capítulos anteriores:

- Capítulo 2, Seção 2.11: Paralelismo e sincronização de instruções.
- Capítulo 3, Seção 3.6: Paralelismo e aritmética computacional: associatividade.
- Capítulo 4, Seção 4.10: Paralelismo e paralelismo avançado em nível de instrução.
- Capítulo 5, Seção 5.8: Paralelismo e hierarquias de memória: coerência de cache.
- Capítulo 6, Seção 6.9: Paralelismo e E/S: Redundant Arrays of Inexpensive Disks.

Verdadeiro ou falso: para que se beneficie de um multiprocessador, uma aplicação precisa ser concorrente.

Verifique você mesmo

7.2

A dificuldade de criar programas com processamento paralelo

A dificuldade com o paralelismo não está no hardware; é que muito poucos programas de aplicação importantes foram escritos para completar as tarefas mais cedo nos multiprocessadores. É difícil escrever software que usa processadores múltiplos para completar uma tarefa mais rápido, e o problema fica pior à medida que o número de processadores aumenta.

Mas por que isso acontece? Por que os programas de processamento paralelo devem ser tão mais difíceis de desenvolver do que os programas sequenciais?

A primeira razão é que você *precisa* obter um bom desempenho e eficiência do programa paralelo em um multiprocessador; caso contrário, você usaria um programa sequencial em um processador, já que a programação é mais fácil. Na verdade, as técnicas de projeto de processadores, como execução superescalar e fora de ordem, tiram vantagem do paralelismo em nível de instrução (ver Capítulo 4), normalmente sem envolvimento do programador. Tais inovações reduzem a necessidade de reescrever programas para multiprocessadores, já que os programadores poderiam não fazer nada e ainda assim seus programas sequenciais seriam executados mais rapidamente nos novos computadores.

Por que é difícil escrever programas de multiprocessador que sejam rápidos, especialmente quando o número de processadores aumenta? No Capítulo 1, usamos a analogia de oito repórteres tentando escrever um único artigo na esperança de realizar o trabalho oito vezes mais rápido. Para ter sucesso, a tarefa precisa ser dividida em oito partes de mesmo tamanho, pois senão alguns repórteres estariam ociosos enquanto esperam que aqueles com partes maiores terminem. Outro perigo do desempenho seria que os repórteres gastariam muito tempo se comunicando entre si em vez de escrever suas partes do artigo. Para essa analogia e para a programação paralela, os desafios incluem escalonamento, balanceamento de carga, tempo para sincronização e overhead para a comunicação entre as partes. O desafio é ainda maior quando aumenta o número de repórteres para um artigo do jornal e quanto aumenta o número de processadores para a programação paralela.

Nossa discussão no Capítulo 1 revela outro obstáculo, conhecido como a Lei de Amdahl. Ela nos lembra que mesmo as pequenas partes de um programa precisam estar em paralelo para que o programa faça bom proveito dos muitos cores.

EXEMPLO

DESAFIO do speed-up

Suponha que você queira alcançar um speed-up 90 vezes mais rápido com 100 processadores. Que fração da computação original pode ser sequencial?

A Lei de Amdahl (Capítulo 1) diz que:

$$\text{Tempo de execução após melhoria} = \frac{\text{Tempo de execução afetado pela melhoria}}{\text{Quantidade de melhoria}} + \text{Tempo de execução não afetado}$$

Podemos reformular a lei de Amdahl em termos de speed-up *versus* o tempo de execução original:

$$\text{Speed-up} = \frac{\text{Tempo de execução antes}}{(\text{Tempo de execução antes} - \text{Tempo de execução afetado}) + \frac{\text{Tempo de execução afetado}}{100}}$$



Essa fórmula normalmente é reescrita considerando-se que o tempo de execução antes é 1 para alguma unidade de tempo, e o tempo de execução afetado pela melhoria é considerado a fração do tempo de execução original:

$$\text{Speed up} = \frac{1}{(1 - \text{Fração de tempo afetada}) + \frac{\text{Fração de tempo afetada}}{100}}$$

Substituindo pela meta de um speed-up de 90 na fórmula anterior:

$$90 = \frac{1}{(1 - \text{Fração de tempo afetada}) + \frac{\text{Fração de tempo afetada}}{100}}$$

Então, simplificando a fórmula e resolvendo para a fração de tempo afetada:

$$90 \times (1 - 0,99 \times \text{Fração de tempo afetada}) = 1$$

$$90 - (90 \times 0,99 \times \text{Fração de tempo afetada}) = 1$$

$$90 - 1 = 90 \times 0,99 \times \text{Fração de tempo afetada}$$

$$\text{Fração de tempo afetada} = 89/89,1 = 0,999$$

Portanto, para obter um speed-up de 90 com 100 processadores, a porcentagem sequencial só poderá ser 0,1%.

Entretanto, existem aplicações com um substancial paralelismo.

DESAFIO do speed-up, ainda maior

Suponha que você queira realizar duas somas: uma é a soma de duas variáveis escalares e outra é uma soma matricial de um par de arrays bidimensionais, com dimensões 10×10 . Que speed-up você obtém com 10 *versus* 100 processadores? Em seguida, calcule os speed-ups supondo que as matrizes crescem para 100 por 100.

EXEMPLO

Se considerarmos que o desempenho é uma função do tempo para uma adição, t , então há 10 adições que não se beneficiam dos processadores paralelos e 100 adições que se beneficiam. Se o tempo para um único processador é $110t$, o tempo de execução para 10 processadores é

RESPOSTA

$$\begin{aligned} \text{Tempo de execução após melhoria} &= \\ \frac{\text{Tempo de execução afetado pela melhoria}}{\text{Quantidade de melhoria}} + \text{Tempo de execução não afetado} & \end{aligned}$$

$$\text{Tempo de execução após melhoria} = \frac{110t}{10} + 10t = 20t$$

Então, o speed-up com 10 processadores é $110t/20t = 5,5$. O tempo de execução para 100 processadores é

$$\text{Tempo de execução após melhoria} = \frac{100t}{100} + 10t = 11t$$

de modo que o speed-up com 100 processadores é $100t/11t = 10$.

Assim, para o tamanho deste problema, obtemos cerca de 55% do speed-up em potencial com 10 processadores, mas somente 10% com 100. Veja o que acontece quando aumentamos a matriz. O programa sequencial agora utiliza $10t + 10.000t = 10.010t$. O tempo de execução para 10 processadores é

$$\text{Tempo de execução após melhoria} = \frac{10.000t}{10} + 10t = 1010t$$

de modo que o speed-up com 10 processadores é $10.010t/1010t = 9,9$. O tempo de execução para 100 processadores é

$$\text{Tempo de execução após melhoria} = \frac{10.000t}{100} + 10t = 110t$$

de modo que o speed-up com 100 processadores é $10.010t/110t = 91$. Assim, para esse tamanho de problema maior, obtemos cerca de 99% do speed-up em potencial com 10 processadores e mais de 90% com 100.

Esses exemplos mostram que obter um bom speed-up em um multiprocessador enquanto se mantém o tamanho do problema fixo é mais difícil do que conseguir um bom speed-up aumentando o tamanho do problema. Isso nos permite apresentar dois termos que descrevem maneiras de expandir. **Expansão forte** significa medir o speed-up enquanto se mantém o tamanho do problema fixo. **Expansão fraca** significa que o tamanho do problema cresce proporcionalmente com o aumento no número de processadores. Vamos supor que o tamanho do problema, M, seja o conjunto de trabalho na memória principal, e que temos P processadores. Então, a memória por processador para a expansão forte é aproximadamente M/P , e para a expansão fraca ela é aproximadamente M.

Dependendo da aplicação, você pode argumentar em favor de qualquer uma dessas técnicas de expansão. Por exemplo, o benchmark de banco de dados débito-crédito TPC-C (Capítulo 6) requer que você aumente o número de contas de cliente para conseguir um maior número de transações por minuto. O argumento é que não faz sentido pensar que determinada base de clientes de repente começará a usar caixas eletrônicos 100 vezes por dia só porque o banco adquiriu um computador mais rápido. Em vez disso, se você for demonstrar um sistema que pode funcionar 100 vezes o número de transações por minuto, deverá fazer uma experiência com 100 vezes a quantidade de clientes.

Este exemplo final mostra a importância do balanceamento de carga.

EXEMPLO

DESAFIO do speed-up: balanceamento DE CARGA

Para conseguir o speed-up de 91 no problema maior, mostrado anteriormente, com 100 processadores, consideramos que a carga foi balanceada perfeitamente. Ou seja, cada um dos 100 processadores teve 1% do trabalho a realizar. Em vez disso, mostre o impacto sobre o speed-up se a carga de um processador for maior que todo o restante. Calcule em 2% e 5%.

RESPOSTA

Se um processador tem 2% da carga paralela, então ele precisa realizar $2\% \times 10.000$ ou 200 adições, e os outros 99 compartilharão as 9800 restantes. Como eles estão operando simultaneamente, podemos simplesmente calcular o tempo de execução como um máximo

$$\text{Tempo de execução após melhoria} = \text{Max}\left(\frac{9.800t}{99}, \frac{200t}{1}\right) + 10t = 210t$$

O speed-up cai para $10.010t/210t = 48$. Se um processador tem 5% da carga, ele precisa realizar 500 adições:

$$\text{Tempo de execução após melhoria} = \text{Max}\left(\frac{9.500t}{99}, \frac{500t}{1}\right) + 10t = 510t$$

O speed-up cai ainda mais para $10.010t/510t = 20$. Esse exemplo demonstra o valor do balanceamento de carga, pois apenas um único processador com o dobro da carga dos outros reduz o speed-up quase ao meio, e cinco vezes a carga em um processador reduz o speed-up por quase um fator de cinco.

Verdadeiro ou falso: a expansão forte não está ligada à lei de Amdahl.

Verifique você mesmo

7.3

Multiprocessadores de memória compartilhada

Dada a dificuldade de reescrever programas antigos para que funcionem bem em hardware paralelo, uma pergunta natural é o que os projetistas de computador podem fazer para simplificar a tarefa. Uma resposta para isso foi oferecer um único espaço de endereços físico que todos os processadores possam compartilhar, de modo que os programas não precisem se preocupar com o local onde são executados, apenas que podem ser executados em paralelo. Nessa técnica, todas as variáveis de um programa podem ficar disponíveis a qualquer momento para qualquer processador. A alternativa é ter um espaço de endereços separado por processador, o que requer que o compartilhamento seja explícito; vamos descrever essa opção na próxima seção. Quando o espaço de endereços físico é comum — que normalmente acontece para chips multicore —, então o hardware normalmente oferece coerência de cache para dar uma visão consistente da memória compartilhada (veja Seção 5.8 do Capítulo 5.)

Um **multiprocessador de memória compartilhada** (**SMP – shared memory multiprocessor**) é aquele que oferece ao programador um *único espaço de endereços físico* para todos os processadores, embora um termo mais preciso teria sido multiprocessador de *endereço* compartilhado. Observe que esses sistemas ainda podem executar tarefas independentes em seus próprios espaços de endereços virtuais, mesmo que todos compartilhem um espaço de endereços físico. Os processadores se comunicam por meio de variáveis compartilhadas na memória, com todos os processadores capazes de acessar qualquer local da memória por meio de loads e stores. A [Figura 7.2](#) mostra a organização clássica de um SMP.

Microprocessadores com único espaço de endereços podem ser de dois tipos. O primeiro leva aproximadamente o mesmo tempo para acessar a memória principal, não importa qual processador o solicite e não importa qual palavra é solicitada. Essas máquinas são chamadas multiprocessadores de **acesso uniforme à memória** (**UMA**). No segundo estilo, alguns acessos à memória são muito mais rápidos do que outros, dependendo de qual processador pede qual palavra. Essas máquinas são chamadas multiprocessadores de **acesso não uniforme à memória** (**NUMA**). Como você poderia esperar, os desafios de programação são mais difíceis para um multiprocessador NUMA do que para um multiprocessador UMA, mas as máquinas NUMA podem expandir para tamanhos maiores, e as NUMAs podem ter latência inferior para a memória próxima.

multiprocessador de memória compartilhada (SMP) Um processador paralelo com um único espaço de endereços, implicando na comunicação implícita com loads e stores.

acesso uniforme à memória (UMA) Um multiprocessador em que os acessos à memória principal levam aproximadamente a mesma quantidade de tempo não importa qual processador acessa e não importa qual palavra é solicitada.

acesso não uniforme à memória (NUMA) Um tipo de multiprocessador com espaço de endereços único em que alguns acessos à memória são muito mais rápidos do que outros, dependendo de qual processador solicita qual palavra.

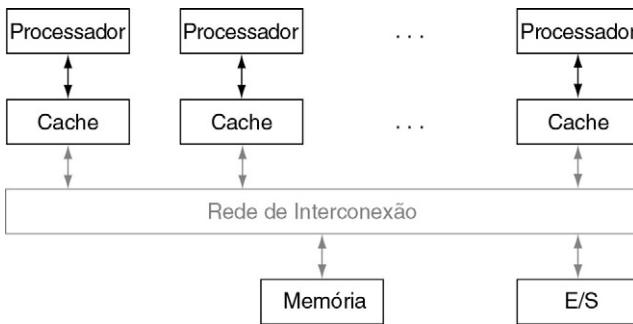


FIGURA 7.2 Organização clássica de um multiprocessador de memória compartilhada.

sincronização O processo de coordenar o comportamento de dois ou mais processos, que podem estar sendo executados em diferentes processadores.

lock Um dispositivo de sincronização que permite o acesso aos dados somente por um processador de cada vez.

Como os processadores operando em paralelo normalmente compartilharão dados, eles também precisam coordenar quando operarão sobre dados compartilhados; caso contrário, um processador poderia começar a trabalhar nos dados antes que outro tenha terminado. Essa coordenação é chamada de **sincronização**. Quando o compartilhamento tem o suporte de um único espaço de endereços, é preciso haver um mecanismo separado para sincronização. Uma técnica utiliza um **lock** para uma variável compartilhada. Somente um processador de cada vez pode adquirir o lock, e outros processadores interessados nos dados compartilhados precisam esperar até que o processador original libere a variável. A Seção 2.11 do Capítulo 2 descreve as instruções para o locking no MIPS.

EXEMPLO

RESPOSTA

Um programa de processamento paralelo simples para um espaço de endereços compartilhado

Suponha que queremos somar 100.000 números em um computador com multiprocessador com tempo de acesso à memória uniforme. Vamos considerar que temos 100 processadores.

A primeira etapa novamente seria dividir o conjunto de números em subconjuntos do mesmo tamanho. Não alocamos os subconjuntos a um espaço de memória diferente, já que existe uma única memória para essa máquina; apenas atribuímos endereços iniciais diferentes a cada processador. P_n é o número que identifica o processador, entre 0 e 99. Todos os processadores começam o programa executando um loop que soma seu subconjunto de números:

```

sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)
    sum[Pn] = sum[Pn] + A[i]; /* soma as áreas atribuídas*/
    
```

redução Uma função que processa uma estrutura de dados e retorna um único valor.

A próxima etapa é fazer essas muitas somas parciais. Essa etapa se chama **redução**. Dividimos para conquistar. A metade dos processadores soma pares de somas parciais, depois, um quarto soma pares das novas somas parciais e assim por diante, até que tenhamos uma única soma final. A Figura 7.3 ilustra a natureza hierárquica dessa redução.

Neste exemplo, os dois processadores precisam ser sincronizados antes que o processador “consumidor” tente ler o resultado do local da memória escrito pelo

processador “produtor”; caso contrário, o consumidor pode ler o valor antigo dos dados. Queremos que cada processador tenha sua própria versão da variável contadora de loop i , de modo que precisamos indicar que ela é uma variável “privada”. Aqui está o código (half também é privada):

```
half = 100; /* 100 processadores no multiprocessador*/
repeat
    synch(); /* espera a conclusão da soma parcial*/
    if (half%2 != 0 && Pn == 0)
        sum[0] = sum[0] + sum[half-1];
        /* soma condicional necessária quando half é
           ímpar; Processor0 obtém elemento ausente */
    half = half/2; /* linha divisora sobre quem soma */
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1); /* sai com soma final em Sum[0] */
```

Verdadeiro ou falso: multiprocessadores de memória compartilhada não podem tirar proveito do paralelismo em nível de tarefa.

**Verifique
você mesmo**

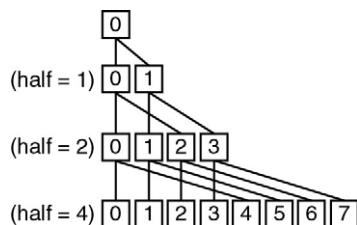


FIGURA 7.3 Os quatro últimos níveis de uma redução que soma os resultados de cada processador, de baixo para cima. Para todos os processadores cujo número i é menor que $half$, adicione a soma produzida pelo processador número ($i+half$) à sua soma.

Detalhamento: Uma alternativa ao compartilhamento do espaço de endereço físico seria ter espaços de endereços físicos separados, mas compartilhar um espaço de endereços virtuais comum, deixando para o sistema operacional a tarefa de cuidar da comunicação. Essa técnica tem sido experimentada, mas possui um alto overhead para oferecer uma abstração de memória compartilhada prática ao programador.

7.4

Clusters e outros multiprocessadores de passagem de mensagens

A técnica alternativa ao compartilhamento de um espaço de endereços é que cada processador tenha seu próprio espaço privado de endereços físicos. A Figura 7.4 mostra a organização clássica de um multiprocessador com múltiplos espaços de endereços privados. Esse multiprocessador alternativo precisa se comunicar por meio da **passagem de mensagens** explícita, que tradicionalmente é o nome desse estilo de computadores. Desde que o sistema tenha rotinas para **enviar** e **receber mensagens**, a coordenação é embutida na passagem da mensagem, pois um processador sabe quando uma mensagem é enviada, e o processador receptor sabe quando uma mensagem chega. Se o emissor precisar de confirmação de que a mensagem chegou, o

passagem de mensagens
Comunicação entre vários processadores enviando e recebendo informações explicitamente.

rotina para enviar mensagem Uma rotina usada por um processador em máquinas com memórias privadas para passar uma mensagem a outro processador.

rotina para receber mensagem Uma rotina usada por um processador em máquinas com memórias privadas para aceitar uma mensagem de outro processador.

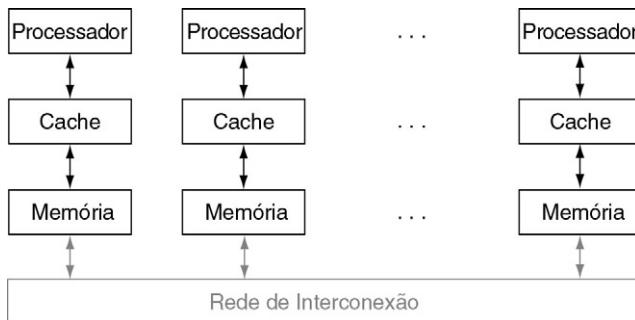


FIGURA 7.4 Organização clássica de um multiprocessador com múltiplos espaços de endereços privados, tradicionalmente chamado de multiprocessador de passagem de mensagens. Observe que, diferente do SMP da Figura 7.2, a rede de interconexão não está entre as caches e a memória, mas entre os nós processador-memória.

processador receptor poderá então enviar uma mensagem de confirmação para o emissor.

Algumas aplicações concorrentes funcionam bem em hardware paralelo, não importa se ele oferece endereços compartilhados ou passagem de mensagens. Em particular, paralelismo e aplicações em nível de tarefa com pouca comunicação — como busca na web, servidores de correio e servidores de arquivo — não exigem que o endereçamento compartilhado funcione bem.

Houve várias tentativas de construir computadores de alto desempenho com base em redes de passagem de mensagens de alto desempenho, e eles ofereceram melhor desempenho de comunicação absoluta do que os clusters criados por meio de redes locais. O problema foi que eles eram muito mais caros. Poucas aplicações poderiam justificar o desempenho de comunicação mais alto, dados os custos muito mais altos. Logo, os **clusters** se tornaram o exemplo mais divulgado atualmente do computador de passagem de mensagens. Os clusters geralmente são coleções de computadores básicos que são conectados entre si por sua interconexão de E/S, através de switches e cabos de rede padrão. Cada um executa uma cópia distinta do sistema operacional. Praticamente cada serviço da internet conta com clusters de servidores e switches básicos.

Uma desvantagem dos clusters foi que o custo de administrar um cluster de n máquinas é cerca do mesmo que o custo de administrar n máquinas independentes, enquanto o custo de administrar um multiprocessador de memória compartilhada com n processadores é aproximadamente o mesmo que administrar uma única máquina.

Esse ponto fraco é um dos motivos para a popularidade das máquinas virtuais (Capítulo 5), pois as VMs tornam os clusters mais fáceis de administrar. Por exemplo, as VMs possibilitam parar ou iniciar programas atomicamente, o que simplifica as atualizações de software. As VMs podem ainda migrar um programa de um computador em um cluster para outro sem interromper o programa, permitindo que um programa migre a partir de um hardware defeituoso.

Outra desvantagem dos clusters é que os processadores em um cluster normalmente são conectados por meio da interconexão de E/S de cada computador, enquanto os cores em um multiprocessador normalmente são conectados na interconexão de memória do computador. A interconexão de memória tem largura de banda mais alta e latência mais baixa, permitindo um desempenho de comunicação muito melhor.

Um último ponto fraco é o overhead na divisão de memória: um cluster de n máquinas tem n memórias independentes e n cópias do sistema operacional, mas um multiprocessador de memória compartilhada permite que um único programa utilize quase toda a memória no computador, e só precisa de uma única cópia do SO.

clusters Coleções de computadores conectados por E/S por switches de rede padrão para formar um multiprocessador de passagem de mensagens.



Eficiência da memória

Suponha que um único processador de memória compartilhada tenha 20GB de memória principal, cinco computadores em cluster com 4GB cada, e o SO ocupe 1GB. Quanto espaço a mais existe para os usuários com memória compartilhada?

EXEMPLO

A razão da memória disponível para os programas do usuário no computador de memória compartilhada *versus* o cluster seria

$$\frac{20-1}{5 \times (4-1)} = \frac{19}{15} \approx 1,25$$

de modo que os computadores com memória compartilhada têm cerca de 25% mais espaço.

RESPOSTA

Vamos refazer o exemplo de soma da seção anterior para ver o impacto das memórias privadas múltiplas e a comunicação explícita.

Um programa simples de processamento paralelo para passagem de mensagens

Suponha que queremos somar 100.000 números em um multiprocessador por passagem de mensagens com 100 processadores, cada um com múltiplas memórias privadas.

EXEMPLO

Como esse computador possui múltiplos espaços de endereçamento, o primeiro passo é distribuir os 100 subconjuntos para cada uma das memórias locais. O processador contendo os 100.000 números envia os subconjuntos para cada um dos 100 nós de memória de processador.

RESPOSTA

A próxima etapa é obter a soma de cada subconjunto. Essa etapa é simplesmente um loop que toda unidade de execução segue: ler uma palavra da memória local e adicioná-la a uma variável local:

```
sum = 0;
for (i = 0; i<1000; i = i + 1) /* Loop em cada array */
    sum = sum + AN[i]; /* Soma os arrays locais */
```

A última etapa é a redução que soma essas 100 somas parciais. A parte difícil é que cada soma parcial está localizada em uma unidade de execução diferente. Consequentemente, precisamos usar a rede de interconexão para enviar somas parciais e acumular a soma final. Em vez de enviar todas as somas parciais a um único processador, o que resultaria em acrescentar sequencialmente as somas, mais uma vez dividimos para conquistar.

Primeiro, metade dos processadores envia suas somas parciais para a outra metade dos processadores, em que duas somas parciais são feitas. Depois, um quarto das unidades de execução (metade da metade) envia essa nova soma parcial para o outro quarto dos processadores (a metade da metade restante) para a próxima rodada de somas. Essas divisões, envios e recepções continuam até haver uma única soma

de todos os números. Seja Pn o número da unidade de execução, $send(x, y)$ uma rotina que envia pela rede de interconexão para a unidade de execução número x o valor y e $receive()$ a função que recebe um valor da rede para esse processador :

```
limit = 100; half = 100; /* 100 processadores */
repeat
    half = (half+1)/2; /* linha divisória entre send e receive*/
    if (Pn >= half && Pn < limit) send(Pn - half, sum);
    if (Pn < (limit/2)) sum = sum + receive();
    limit = half; /* limite superior dos emissores */
until (half == 1); /* sai com a soma final */
```

Esse código divide todos os processadores em emissores ou receptores, e cada processador receptor recebe apenas uma mensagem, de modo que podemos presumir que um processador receptor estará suspenso até receber uma mensagem. Portanto, `send` e `receive` podem ser usados como primitivas para sincronização e para comunicação, já que os processadores estão cientes da transmissão dos dados.

Se houver um número ímpar de nós, o nó central não participa da emissão/recepção. O limite, então, é definido de modo que esse nó seja o nó mais alto na próxima iteração.

Detalhamento: Este exemplo considera implicitamente que a passagem de mensagens é tão rápida quanto a adição. Na realidade, a emissão e recepção de mensagens é muito mais lenta. Uma otimização para balancear melhor o cálculo e a comunicação poderia ser o uso de menos nós recebendo muitas somas de outros processadores.

Interface hardware/software

Computadores que contam com a passagem de mensagens para a comunicação, em vez da memória compartilhada coerente com a cache, são muito mais fáceis para os projetistas de hardware (veja Seção 5.8 do Capítulo 5). A vantagem para os programadores é que a comunicação é explícita, o que significa que existem menos surpresas de desempenho do que com a comunicação implícita nos computadores de memória compartilhada coerentes com a cache. A desvantagem para os programadores é que é mais difícil transportar um programa sequencial para um computador com passagem de mensagens, pois cada comunicação precisa ser identificada antecipadamente, ou o programa não funcionará. A memória compartilhada coerente com a cache permite que o hardware descubra quais dados precisam ser comunicados, o que facilita o transporte. Existem diferenças de opinião quanto ao caminho mais curto para o alto desempenho, dados os prós e contras da comunicação implícita.

A limitação de memórias separadas para memória do usuário torna-se uma vantagem na disponibilidade do sistema. Como um cluster consiste em computadores independentes conectados por meio de uma rede local, é muito mais fácil substituir uma máquina sem paralisar o sistema em um cluster do que em um SMP. Fundamentalmente, o endereçamento compartilhado significa que é difícil isolar um processador e substituí-lo sem um árduo trabalho por parte do sistema operacional. Já que o software de cluster é uma camada que roda sobre o sistema operacional executado em cada computador, é muito mais fácil desconectar e substituir uma máquina defeituosa.

Como os clusters são construídos por meio de computadores inteiros e redes independentes e escaláveis, esse isolamento também facilita expandir o sistema sem paralisar a aplicação que executa sobre o cluster.

Menor custo, alta disponibilidade, maior eficiência de energia e a rápida e gradual expansibilidade tornam os clusters atraentes para provedores de serviços para a world wide web. Os mecanismos de busca que milhões de nós utilizamos todos os dias dependem dessa tecnologia. eBay, Google, Microsoft, Yahoo e outros possuem múltiplos centros de dados, cada um com clusters de dezenas de milhares de processadores. Logicamente, o uso de múltiplos processadores nas empresas de serviço de internet tem sido muito bem-sucedido.

Detalhamento: Outra forma de computação em grande escala é a *computação em grade*, em que os computadores são espalhados por grandes áreas, e depois os programas que executam neles precisam se comunicar por redes de longa distância. A forma mais comum e exclusiva de computação em grade foi promovida pelo projeto SETI@home. Observou-se que milhões de PCs ficam ociosos em determinado momento, sem realizar nada de útil, e eles poderiam ser apanhados e ter boa utilidade se alguém desenvolvesse software que pudesse rodar nesses computadores e depois dar a cada PC uma parte independente do problema para atuar. O primeiro exemplo foi o Search for ExtraTerrestrial Intelligence (SETI). Mais de 5 milhões de usuários de computador em mais de 200 países se inscreveram para o SETI@home e contribuíram coletivamente com mais de 19 bilhões de horas de tempo de processamento de computador. Ao final de 2006, a grade SETI@home operava em 257 TeraFLOPS.

1. Verdadeiro ou falso: assim como os SMPs, os computadores com passagem de mensagens contam com locks para a sincronização.
2. Verdadeiro ou falso: diferentemente dos SMPs, os computadores com passagem de mensagens precisam de múltiplas cópias do programa de processamento paralelo e do sistema operacional.

Verifique você mesmo

7.5

Multithreading do hardware

O **multithreading** do hardware permite que várias threads compartilhem as unidades funcionais de um único processador de um modo sobreposto. Para permitir esse compartilhamento, o processador precisa duplicar o estado independente de cada thread. Por exemplo, cada thread teria uma cópia separada do banco de registradores e do PC. A memória em si pode ser compartilhada por meio de mecanismos de memória virtual, que já suportam multiprogramação. Além disso, o hardware precisa suportar a capacidade de mudar para uma thread diferente com relativa rapidez. Em especial, uma troca de thread deve ser muito mais eficiente do que uma troca de processo, que normalmente exige centenas a milhares de ciclos de processador, enquanto uma troca de thread pode ser instantânea.

Existem dois métodos principais de multithreading do hardware. O **multithreading fine-grained** comuta entre threads a cada instrução, resultando em execução intercalada de várias threads. Essa intercalação normalmente é feita de forma circular, saltando quaisquer threads que estejam suspensas nesse momento. Para tornar o multithreading fine-grained prático, o processador precisa ser capaz de trocar threads a cada ciclo de clock. Uma importante vantagem do multithreading fine-grained é que ele pode ocultar as perdas de vazão que surgem dos stalls curtos e longos, já que as instruções de outras threads podem ser executadas quando uma thread é suspensa. A principal desvantagem do multithreading fine-grained é que ele torna mais lenta a execução das threads individuais, já que uma thread que está pronta para ser executada sem stalls será atrasada por instruções de outras threads.

multithreading do hardware Aumentar a utilização de um processador trocando para outra thread quando uma thread é suspensa.

multithreading fine-grained Uma versão do multithreading do hardware que sugere a comutação entre as threads após cada instrução.

multithreading

coarse-grained Uma versão do multithreading do hardware que sugere a comutação entre as threads somente após eventos significativos, como uma falha de cache.

simultaneous multithreading

(SMT) Uma versão do multithreading que reduz o custo do multithreading, utilizando os recursos necessários para a microarquitetura de despacho múltiplo, escalonada dinamicamente.

O **multithreading coarse-grained** foi criado como uma alternativa para o multithreading fine-grained. Esse método de multithreading comuta threads apenas em stalls onerosos, como as falhas de cache de nível 2. Essa mudança reduz a necessidade de tornar a comutação de thread essencialmente gratuita e tem muito menos chance de tornar mais lenta a execução de uma thread individual, visto que só serão despachadas instruções de outras threads quando uma thread encontrar um stall oneroso. Entretanto, o multithreading coarse-grained sofre de uma grande desvantagem: é limitado em sua capacidade de sanar perdas de vazão, especialmente de stalls mais curtos. Essa limitação surge dos custos de inicialização de pipeline do multithreading coarse-grained. Como um processador com multithreading coarse-grained despacha instruções por meio de uma única thread, quando ocorre um stall, o pipeline precisa ser esvaziado ou congelado. A nova thread que começa a ser executada após o stall precisa preencher o pipeline antes que as instruções consigam ser concluídas. Devido a esse overhead de inicialização, o multithreading coarse-grained é muito mais útil para reduzir a penalidade dos stalls de alto custo, em que a reposição de pipeline é insignificante comparada com o tempo de stall.

O **simultaneous multithreading (SMT)** é uma variação do multithreading do hardware que usa os recursos de um processador de despacho múltiplo escalonado dinamicamente para explorar paralelismo em nível de thread ao mesmo tempo em que explora o paralelismo em nível de instrução. O princípio mais importante que motiva o SMT é que os processadores de despacho múltiplo modernos normalmente possuem mais paralelismo de unidade funcional do que uma única thread efetivamente pode usar. Além disso, com a renomeação de registradores e o escalonamento dinâmico, diversas instruções de threads independentes podem ser despachadas sem considerar as dependências entre elas; a resolução das dependências pode ser tratada pela capacidade de escalonamento dinâmico.

Como estamos contando com os mecanismos dinâmicos existentes, SMT não troca de recurso a cada ciclo, mas sempre está executando instruções de múltiplas threads, deixando para o hardware a associação de *slots* de instrução e registradores renomeados com suas threads apropriadas.

A [Figura 7.5](#) ilustra conceitualmente as diferenças na capacidade de um processador de explorar recursos superescalares para as configurações de processador a seguir. A parte superior mostra como quatro threads seriam executadas de forma independente em um superescalar sem suporte a multithreading. A parte inferior mostra como as quatro threads poderiam ser combinadas para serem executadas no processador de maneira mais eficiente usando três opções de multithreading:

- Um superescalar com multithreading coarse-grained.
- Um superescalar com multithreading fine-grained.
- Um superescalar com simultaneous multithreading.

No superescalar sem suporte a multithreading do hardware, o uso dos slots de despacho é limitado por uma falta de paralelismo em nível de instrução. Além disso, um importante stall, como uma falha de cache de instruções, pode deixar o processador inteiro ocioso.

No superescalar com multithreading coarse-grained, os longos stalls são parcialmente ocultados pela comutação para outra thread que usa os recursos do processador. Embora isso reduza o número de ciclos de clock completamente ociosos, o overhead de inicialização do pipeline ainda produz ciclos ociosos, e as limitações do paralelismo em nível de instrução significam que nem todos os slots de despacho serão utilizados. Em um processador com multithreading coarse-grained, a intercalação de threads elimina quase todos os slots totalmente vazios. Porém, como apenas uma thread despacha instruções em um determinado ciclo de clock, as limitações do paralelismo em nível de instrução ainda geram um número significativo de slots ociosos dentro de alguns ciclos de clock.

No caso SMT, o paralelismo em nível de thread e o paralelismo em nível de instrução são explorados simultaneamente, com múltiplas threads usando os slots de despacho

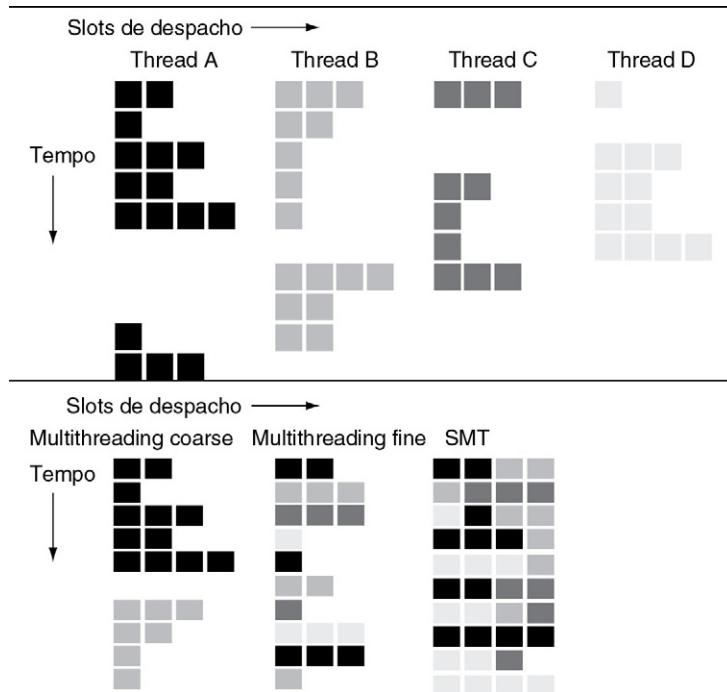


FIGURA 7.5 Como quatro threads usam os slots de despacho de um processador superescalar em diferentes métodos. As quatro threads no topo mostram como cada uma seria executada em um processador superescalar padrão sem suporte a multithreading. Os três exemplos embaixo mostram como elas seriam executadas juntas em três opções de multithreading. A dimensão horizontal representa a capacidade de despacho de instrução em cada ciclo de clock. A dimensão vertical representa uma sequência dos ciclos de clock. Uma caixa vazia (branco) indica que o slot de despacho correspondente está vago nesse ciclo de clock. Os tons de cinza e preto correspondem a quatro threads diferentes nos processadores multithreading. Os efeitos de inicialização de pipeline adicionais para multithreading coarse, que não estão ilustrados nessa figura, levariam a mais perda na vazão para multithreading coarse.

em um único ciclo de clock. O ideal é que o uso de slots de despacho seja limitado por desequilíbrios nas necessidades e na disponibilidade de recursos entre múltiplas threads. Na prática, outros fatores podem restringir o número de slots usados. Embora a Figura 7.5 simplifique bastante a operação real desses processadores, ela ilustra as potenciais vantagens de desempenho em potencial do multithreading em geral e do SMT em particular. Por exemplo, o recente multicore Intel Nehalem suporta SMT com duas threads para melhorar a utilização do core.

Vamos concluir com três observações. Primeiro, pelo Capítulo 1, sabemos que a barreira da potência está forçando um projeto em direção a processadores mais simples e mais eficientes em termos de potência em um chip. Pode ser que os recursos subutilizados dos processadores fora de ordem possam ser reduzidos e, portanto, formas mais simples de multithreading sejam utilizadas. Por exemplo, o microprocessador Sun UltraSPARC T2 (Niagara 2) na Seção 7.11 é um exemplo de um retorno a microarquiteturas mais simples e, portanto, ao uso do multithreading fine-grained.

Segundo, um desafio de desempenho importante é tolerar a latência decorrente das falhas de cache. Computadores fine-grained, como o UltraSPARC T2, trocam para outra thread em caso de falha, o que provavelmente é mais eficaz para esconder a latência da memória do que tentar preencher slots de despacho não usados, como em um SMT.

Uma terceira observação é que o objetivo do multithreading do hardware deve usar o hardware com mais eficiência compartilhando os componentes entre diferentes tarefas. Os projetos multicore também compartilham recursos. Por exemplo, dois processadores poderiam compartilhar uma unidade de ponto flutuante ou uma cache L3.

Esse compartilhamento reduz alguns dos benefícios do multithreading em comparação com o oferecimento de mais cores não multithreaded.

Verifique você mesmo

1. Verdadeiro ou falso: tanto o multithreading quanto o multicore contam com o paralelismo para obter mais eficiência de um chip.
2. Verdadeiro ou falso: o multithreading simultâneo utiliza threads para melhorar a utilização de recursos de um processador fora de ordem, escalonado dinamicamente.

7.6

SISD, MIMD, SIMD, SPMD e vetor

Outra categorização do hardware paralelo proposta na década de 1960 ainda está em uso atualmente. Ela foi baseada no número de fluxos de instruções e no número de fluxos de dados. A [Figura 7.6](#) mostra as categorias. Assim, um processador convencional tem um único fluxo de instruções e um único fluxo de dados, e um multiprocessador convencional possui fluxos de instruções e fluxos de dados múltiplos. Essas duas categorias são abreviadas como **SISD** e **MIMD**, respectivamente.

SISD ou Single Instruction stream, Single Data stream. Um processador único.

MIMD ou Multiple Instruction streams, Multiple Data streams. Um multiprocessador.

		Fluxos de Dados	
		Único	Múltiplo
Fluxos de instrução	Único	SISD: Intel Pentium 4	SIMD: instruções SSE do x86
	Múltiplo	MISD: não há exemplos atuais	MIMD: Intel Xeon e5345 (Cloverdale)

FIGURA 7.6 Categorização de hardware e exemplos baseados no número de fluxos de instruções e fluxos de dados: SISD, SIMD, MISD e MIMD.

SPMD Single Program, Multiple Data streams. O modelo de programação MIMD convencional, em que um único programa é executado em todos os processadores.

SIMD ou Single Instruction stream, Multiple Data streams. Um multiprocessador. A mesma instrução é aplicada a muitos fluxos de dados, assim como em um processador de vetor ou de array.

Embora seja possível escrever programas separados que são executados em diferentes processadores em um computador MIMD e ainda trabalharem juntos para um objetivo grandioso e coordenado, os programadores normalmente escrevem um único programa que executa em todos os processadores de um computador MIMD, contando com instruções condicionais quando diferentes processadores deveriam executar diferentes seções de código. Esse estilo é chamado **Single Program Multiple Data (SPMD)**, mas é apenas o modo normal de programar um computador MIMD.

Embora seja difícil oferecer exemplos de computadores úteis que sejam classificados como múltiplos fluxos de instruções e fluxo de instrução único (MISD), o inverso faz muito mais sentido. Computadores **SIMD** operam sobre vetores de dados. Por exemplo, uma única instrução SIMD poderia acrescentar 64 números enviando 64 fluxos de dados a 64 ALUs, para formar 64 somas dentro de um único ciclo de clock.

As virtudes do SIMD são que todas as unidades de execução paralelas são sincronizadas e todas elas respondem a uma única instrução que emana de um único contador de programa (PC). Do ponto de vista de um programador, isso é próximo do já conhecido SISD. Embora cada unidade esteja executando a mesma instrução, cada unidade de execução tem seus próprios registradores de endereço, e portanto cada unidade pode ter diferentes endereços de dados. Assim, em termos da [Figura 7.1](#), uma aplicação sequencial poderia ser compilada para executar em hardware serial organizado como um SISD ou em hardware paralelo que foi organizado como um SIMD.

A motivação original por trás do SIMD foi amortizar o custo da unidade de controle por dezenas de unidades de execução. Outra vantagem é o tamanho reduzido da memória do programa — SIMD só precisa de uma cópia do código que está sendo executado simultaneamente, enquanto os MIMDs com passagem de mensagem podem precisar de uma cópia em cada processador, e o MIMD com memória compartilhada precisará de múltiplas caches de instrução.

SIMD funciona melhor quando lida com arrays em loops *for*. Logo, para o paralelismo funcionar no SIMD, é preciso haver muitos dados estruturados de forma idêntica, o que é chamado de **paralelismo em nível de dados**. SIMD é mais fraco em instruções *case* ou *switch*, em que cada unidade de execução precisa realizar uma operação diferente sobre seus dados, dependendo de quais dados ela tenha. As unidades de execução com os dados errados são desativadas, de modo que as unidades com dados corretos possam continuar. Essas situações basicamente executam em desempenho $1/n$, onde n é o número de casos.

Os chamados processadores de array que inspiraram a categoria SIMD desapareceram na história (veja  Seção 7.14 no site), mas duas interpretações do SIMD permanecem ativas hoje.

paralelismo em nível de dados Paralelismo obtido operando-se sobre dados independentes.

SIMD no x86: extensões de multimídia

A variação mais utilizada do SIMD encontra-se em quase todo microprocessador de hoje, e é a base das centenas de instruções MMX e SSE do microprocessador x86 (veja Capítulo 2). Elas foram acrescentadas para melhorar o desempenho dos programas de multimídia. Essas instruções permitem que o hardware tenha muitas ALUs operando simultaneamente ou, de modo equivalente, particione uma ALU única e larga em muitas ALUs menores paralelas, que operam simultaneamente. Por exemplo, você poderia considerar que um único componente de hardware seja uma ALU de 64 bits ou duas ALUs de 32 bits ou quatro ALUs de 16 bits ou oito ALUs de 8 bits. Loads e stores simplesmente possuem a largura da ALU mais larga, de modo que o programador pode pensar na mesma instrução de transferência de dados como transferindo um único elemento de dados de 64 bits, ou dois elementos de dados de 32 bits, ou quatro elementos de dados de 16 bits, ou oito elementos de dados de 8 bits.

Esse paralelismo a um custo muito baixo para dados inteiros estreitos foi a inspiração original das instruções MMX do x86. À medida que a lei de Moore continuava, mais hardware era acrescentado a essas extensões de multimídia, e agora o SSE2 admite a execução simultânea de um par de números de ponto flutuante de 64 bits.

A largura da operação e dos registradores é codificada no opcode dessas instruções de multimídia. Enquanto a largura de dados de registradores e operações crescia, o número de opcodes para instruções de multimídia explodia, e agora existem centenas de instruções SSE para realizar as combinações úteis (veja Capítulo 2).

Vetor

Uma interpretação mais antiga e mais elegante do SIMD é a chamada arquitetura de vetor, que tem sido identificada de perto com os Cray Computers. Essa é novamente uma grande combinação com os problemas com muito paralelismo em nível de dados. Em vez de ter 64 ALUs realizando 64 adições simultaneamente, como os antigos processadores de array, as arquiteturas de vetor colocaram a ALU em pipeline para obter bom desempenho com custo reduzido. A filosofia básica da arquitetura de vetor é coletar elementos de dados da memória, colocá-los em ordem em um grande conjunto de registradores, operar sobre eles sequencialmente nos registradores e depois escrever os resultados de volta para a memória. Um recurso importante das arquiteturas de vetor é um conjunto de registradores de vetor. Assim, uma arquitetura de vetor poderia ter 32 registradores de vetor, cada um com 64 elementos de 64 bits.

EXEMPLO**Comparando código de vetor com código convencional**

Suponha que estendamos a arquitetura do conjunto de instruções MIPS com instruções de vetor e registradores de vetor. As operações de vetor utilizam os mesmos nomes das operações MIPS, mas com a letra “V” acrescentada. Por exemplo, addv.d soma dois vetores de precisão dupla. As instruções de vetor apanham como entrada um par de registradores de vetor (addv.d) ou um registrador de vetor e um registrador escalar (addvs.d). No segundo caso, o valor no registrador escalar é usado como a entrada para todas as operações — a operação addvs.d somará o conteúdo de um registrador escalar a cada elemento em um registrador de vetor. Os nomes l_v e s_v indicam load de vetor e store de vetor, e carregam ou armazenam um vetor inteiro de dados de precisão dupla. Um operando é o registrador de vetor a ser carregado ou armazenado; o outro operando, que é um registrador MIPS de uso geral, é o endereço inicial do vetor na memória. Dada essa descrição curta, mostre o código MIPS convencional *versus* o código MIPS de vetor para

$$Y = a \times X + Y$$

onde X e Y são vetores de 64 números de ponto flutuante com precisão dupla, inicialmente residentes na memória, e a é uma variável escalar de precisão dupla. (Esse exemplo é o chamado loop DAXPY, que forma o loop interno do benchmark Linpack; DAXPY significa Double precision $a \times X$ Plus Y .) Suponha que os endereços iniciais de X e Y estejam em $\$s0$ e $\$s1$, respectivamente.

RESPOSTA

Aqui está o código MIPS convencional para o DAXPY:

```

    l.d      $f0,a($sp)      ;carrega escalar a
    addiu   r4,$s0,#512      ;limite superior do que carregar
loop:  l.d      $f2,0($s0)      ;carrega x(i)
        mul.d   $f2,$f2,$f0      ;a x x(i)
        l.d      $f4,0($s1)      ;carrega y(i)
        add.d   $f4,$f4,$f2      ;a x x(i) + y(i)
        s.d      $f4,0($s1)      ;armazena em y(i)
        addiu   $s0,$s0,#8       ;incrementa índice a x
        addiu   $s1,$s1,#8       ;incrementa índice a y
        subu   $t0,r4,$s0        ;calcula limite
        bne    $t0,$zero,loop     ;verificar se terminou

```

Aqui está o código MIPS de vetor para o DAXPY:

```

    l.d      $f0,a($sp)      ;carrega escalar a
    lv      $v1,0($s0)        ;carrega vetor x
    mulvs.d $v2,$v1,$f0      ;multiplica vetor escalar
    lv      $v3,0($s1)        ;carrega vetor y
    addv.d  $v4,$v2,$v3      ;soma y ao produto
    sv      $v4,0($s1)        ;armazena o resultado

```

Existem algumas comparações interessantes entre os dois segmentos de código neste exemplo. A mais impressionante é que o processador de vetor reduz bastante a largura de banda de instrução dinâmica, executando apenas seis instruções contra quase 600 para o MIPS. Essa redução ocorre tanto porque as operações de vetor trabalham sobre 64 elementos quanto porque as instruções de overhead que constituem quase metade do

loop no MIPS não estão presentes no código de vetor. Como você poderia esperar, essa redução nas instruções buscadas e executadas economiza energia.

Outra diferença importante é a frequência dos hazards de pipeline (Capítulo 4). No código MIPS direto, cada `add . d` precisa esperar por um `mul . d`, e cada `s . d` precisa esperar pelo `add . d`. No processador de vetor, cada instrução de vetor só gerará stall para o primeiro elemento em cada vetor, e depois os elementos subsequentes fluirão tranquilamente pelo pipeline. Assim, os stalls do pipeline só são necessários uma vez por operação de vetor, em vez de uma vez por elemento de vetor. Neste exemplo, a frequência de stall do pipeline no MIPS será de aproximadamente 64 vezes maior do que no VMIPS. Os stalls do pipeline podem ser reduzidos no MIPS usando desdobramento de loop (veja Capítulo 4). Porém, a grande diferença na largura de banda de instrução não pode ser reduzida.

Detalhamento: O loop no exemplo anterior combinou exatamente com o tamanho do vetor. Quando os loops são mais curtos, as arquiteturas de vetor utilizam um registrador que reduz o tamanho das operações de vetor. Quando os loops são maiores, acrescentamos código de contabilidade para percorrer operações de vetor de tamanho total e tratar do restante. Esse último processo é conhecido como *strip mining* (ou *mineração a céu aberto*).

Vetor versus escalar

As instruções de vetor possuem várias propriedades importantes em comparação com os arquivos convencionais de conjunto de instruções, que são chamadas *arquiteturas escalares* nesse contexto:

- Uma única instrução de vetor especifica muito trabalho — isso é equivalente a executar um loop inteiro. A largura de banda de busca e decodificação de instrução necessária é bastante reduzida.
- Usando uma instrução de vetor, o compilador ou programador indica que o cálculo de cada resultado no vetor é independente do cálculo de outros resultados no mesmo vetor, de modo que o hardware não tem de verificar hazards de dados dentro de uma instrução de vetor.
- Arquiteturas e compiladores de vetor têm uma reputação de tornar muito mais fácil que os multiprocessadores MIMD escrever aplicações eficientes quando elas contêm paralelismo em nível de dados.
- O hardware só precisa verificar hazards de dados entre duas instruções de vetor uma vez por operando de vetor, e não uma vez para cada elemento dentro dos vetores. A redução de verificações pode economizar potência.
- Instruções de vetor que acessam a memória possuem um padrão de acesso conhecido. Se os elementos do vetor forem todos adjacentes, então buscar o vetor de um conjunto de bancos de memória bastante intervalados funciona muito bem. Assim, o custo da latência para a memória principal é visto apenas uma vez para o vetor inteiro, e não uma vez para cada palavra do vetor.
- Como um loop inteiro é substituído por uma instrução de vetor cujo comportamento é predeterminado, os hazards de controle que normalmente surgiram do desvio do loop são inexistentes.
- A economia na largura de banda da instrução e verificação de hazard mais o uso eficaz da largura de banda da memória dão às arquiteturas de vetor vantagens em potência e energia contra as arquiteturas escalares.

Por esses motivos, as operações de vetor podem se tornar mais rápidas que uma sequência de operações escalares sobre o mesmo número de itens de dados, e os projetistas são motivados a incluir unidades de vetor se o domínio da aplicação puder usá-las com frequência.

Vetor versus extensões de multimídia

Assim como as extensões de multimídia encontradas nas instruções SSE do x86, uma instrução de vetor especifica múltiplas operações. Porém, as extensões de multimídia normalmente especificam algumas poucas operações, enquanto o vetor especifica dezenas de operações. Diferente das extensões de multimídia, o número de elementos em uma operação de vetor não está no opcode, mas em um registrador separado. Isso significa que diferentes versões da arquitetura de vetor podem ser implementadas com um número diferente de elementos apenas mudando o conteúdo desse registrador e retendo, portanto, a compatibilidade binária. Ao contrário, um novo grande conjunto de opcodes é acrescentado toda vez que o tamanho do “vetor” muda na arquitetura da extensão de multimídia do x86.

Também diferente das extensões de multimídia, as transferências de dados não precisam ser contíguas. Os vetores admitem os acessos “strided”, em que o hardware carrega cada n -ésimo elemento de dados na memória, e acessos indexados, em que o hardware encontra os endereços dos itens a serem carregados em um registrador de vetor.

Assim como as extensões de multimídia, o vetor facilmente captura a flexibilidade nas larguras de dados, de modo que é fácil fazer uma operação funcionar em 32 elementos de dados de 64 bits ou em 64 elementos de dados de 32 bits ou em 128 elementos de dados de 16 bits ou 256 elementos de dados de 8 bits.

Geralmente, as arquiteturas de vetor são um meio muito eficaz de executar programas de processamento paralelo de dados; elas combinam melhor com a tecnologia de compilador do que extensões de multimídia; são mais fáceis de evoluir com o tempo do que as extensões de multimídia na arquitetura x86.

Verifique você mesmo

Verdadeiro ou falso: conforme exemplificamos no x86, as extensões de multimídia podem ser consideradas como uma arquitetura de vetor com vetores curtos que suportam apenas transferências sequenciais de dados de vetor.

Detalhamento: Dadas as vantagens do vetor, por que eles não são mais comuns fora da computação de alto desempenho? Havia preocupações sobre o custo maior para registradores de vetor aumentando o tempo de troca de contexto e a dificuldade de tratar das falhas de página nos loads e stores de vetor, e as instruções SIMD conseguiram alguns dos benefícios das instruções de vetor. Porém, os anúncios recentes da Intel sugerem que os vetores desempenharão um papel mais importante. A *Advanced Vector Instructions (AVI)* da Intel, disponível em 2010, expandirá a largura dos registradores SSE de 128 bits para 256 bits imediatamente, e permitirá a eventual expansão para 1024 bits. Essa última largura é equivalente a 16 números de ponto flutuante de precisão dupla. Se houver instruções load e store de vetor, isso ainda não está claro. Além disso, a entrada da Intel no mercado discreto de GPU para 2010 — apelidado de “Larrabee” — supostamente terá instruções de vetor.

Detalhamento: Outra vantagem das extensões de vetor e multimídia é que é relativamente fácil estender uma arquitetura de conjunto de instruções escalar com essas instruções para melhorar o desempenho das operações paralelas com dados.

7.7

Introdução às unidades de processamento de gráficos

Uma justificativa importante para o acréscimo de instruções SIMD às arquiteturas existentes foi que muitos microprocessadores eram conectados a telas gráficas em PCs e estações de trabalho, de modo que uma fração cada vez maior do tempo de processamento era usada para os gráficos. Daí, quando a lei de Moore aumentou o número de transistores disponíveis nos microprocessadores, fez sentido melhorar o processamento gráfico.

Assim como a lei de Moore permitiu que a CPU melhorasse o processamento gráfico, também permitiu que os chips do controlador gráfico de vídeo acrescentassem funções para acelerar gráficos 2D e 3D. Além do mais, na ponta estavam as placas gráficas caras, geralmente da Silicon Graphics, que poderiam ser acrescentadas às estações de trabalho, para permitir a criação de imagens com qualidade fotográfica. Essas placas gráficas de alto nível foram comuns na criação de imagens geradas por computador, que mais tarde entraram nos anúncios de televisão e depois nos filmes. Assim, os controladores gráficos de vídeo tinham um alvo direcionado quando os recursos de processamento cresceram, assim como os supercomputadores ofereceram um rico recurso de ideias para os microprocessadores na busca por maior desempenho.

Uma força motriz importante para melhorar o processamento gráfico foi a indústria de jogos de computador, tanto em PCs quanto em consoles de jogos dedicados, como o PlayStation da Sony. O mercado de jogos em rápido crescimento encorajou muitas empresas a fazerem investimentos cada vez maiores no desenvolvimento de hardware gráfico mais rápido, e esse feedback positivo levou o processamento gráfico a melhorar em um ritmo mais rápido do que o processamento de uso geral nos microprocessadores centrais.

Dado que a comunidade de gráficos e jogos teve objetivos diferentes da comunidade de desenvolvimento de microprocessador, ela evoluiu seu próprio estilo de processamento e terminologia. Quando os processadores gráficos aumentaram sua potência, eles ganharam o nome *Graphics Processing Units*, ou *GPUs*, para distingui-los das CPUs. Aqui estão algumas das principais características de como as GPUs se distinguem das CPUs:

- GPUs são aceleradores que complementam uma CPU, de modo que não precisam ser capazes de realizar todas as tarefas de uma CPU. Esse papel lhes permitiu dedicar todos os seus recursos aos gráficos. Não importa se as GPUs realizam algumas tarefas mal ou que não realizem, visto que, em um sistema com uma CPU e uma GPU, a CPU pode realizá-las se for preciso. Assim, a combinação CPU-GPU é um exemplo de *multiprocessamento heterogêneo*, em que nem todos os processadores são idênticos. (Outro exemplo é a arquitetura IBM Cell da Seção 7.11, que também foi projetada para acelerar gráficos 2D e 3D.)
- As instruções de programação das GPUs são interfaces de programação de aplicação (APIs) de alto nível, como OpenGL e Microsoft's DirectX, junto com linguagens de sombreamento gráfico de alto nível, como C for Graphics (Cg) da NVIDIA e High Level Shader Language (HLSL) da Microsoft. Os compiladores de linguagem são voltados para linguagens intermediárias padrão da indústria, em vez de instruções de máquina. O software de driver GPU gera instruções de máquina otimizadas, específicas para GPU. Embora essas APIs e linguagens evoluam rapidamente para abranger novos recursos de GPU habilitados pela lei de Moore, a liberdade da compatibilidade com a instrução binária permite que os projetistas de GPU explorem novas tecnologias sem temer que sejam seladas para sempre com a implementação de experimentos falhos. Esse ambiente leva à inovação mais rápida em GPUs do que em CPUs.
- O processamento gráfico envolve o desenho de vértices de primitivas de geometria 3D, como linhas e triângulos, e *sombreamento* ou renderização de fragmentos de pixels de primitivas geométricas. Os *video games* por exemplo, desenham 20 a 30 vezes mais pixels que vértices.
- Cada vértice pode ser desenhado independentemente, assim como na renderização de cada fragmento de pixel. Para renderizar milhões de pixels por frame rapidamente, a GPU evoluiu para executar muitos threads de programas sombreadores de vértice e pixel em paralelo.
- Os tipos de dados gráficos são vértices, consistindo em coordenadas (x, y, z, w), e pixels, consistindo em componentes de cor (vermelho, verde, azul, alfa). (Veja o Apêndice A para descobrir mais sobre vértices e pixels.) GPUs representam cada

componente do vértice como um número de ponto flutuante de 32 bits. Cada um dos quatro componentes de pixel foi originalmente um inteiro não sinalizado de 8 bits, mas as GPUs recentes agora representam cada componente como um número de ponto flutuante de precisão simples, entre 0,0 e 1,0.

- O conjunto de trabalho pode ter centenas de megabytes, e ele não mostra a mesma localidade temporal que os dados nas principais aplicações. Além do mais, existe muito paralelismo em nível de dados nessas tarefas.

Essas diferenças têm levado a diferentes estilos de arquitetura:

- Talvez a maior diferença seja que as GPUs não contam com caches multinível para contornar a longa latência para a memória, como nas CPUs. Em vez disso, as GPUs contam em ter threads suficientes para ocultar a latência para a memória. Ou seja, entre o momento de uma solicitação de memória e o momento em que os dados chegam, a GPU executa centenas ou milhares de threads que são independentes dessa solicitação.
- As GPUs contam com um extenso paralelismo para obter alto desempenho, implementando muitos processadores paralelos e muitos threads concorrentes.
- A memória principal da GPU é assim orientada para largura de banda, em vez de latência. Existem até mesmo chips de DRAM separados para GPUs que são mais largas e possuem largura de banda mais alta que os chips de DRAM para as CPUs. Além disso, as memórias da GPU tradicionalmente têm tido memória principal menor que os microprocessadores convencionais. Em 2008, as GPUs normalmente tinham 1GB ou menos, enquanto as CPUs tinham de 2 a 32GB. Finalmente, lembre-se de que, para a computação de uso geral, você precisa incluir o tempo para transferir os dados entre a memória da CPU e a memória da GPU, pois a GPU é um coprocessador.
- Dada a confiança em muitos threads para oferecer boa largura de banda de memória, as GPUs podem acomodar muitos processadores paralelos, além de muitos threads. Logo, cada processador de GPU é altamente multithreaded.
- No passado, as GPUs contavam com processadores heterogêneos de uso especial para oferecer o desempenho necessário a aplicações gráficas. GPUs recentes estão voltadas para processadores idênticos de uso geral, de modo a oferecer mais flexibilidade na programação, tornando-os mais semelhantes aos projetos multicore encontrados na computação principal.
- Dada a natureza de quatro elementos dos tipos de dados gráficos, as GPUs historicamente possuem instruções SIMD, como as CPUs. Contudo, as GPUs recentes estão focalizando mais as instruções escalares para melhorar a facilidade de programação e a eficiência.
- Diferente das CPUs, não tem havido suporte para a aritmética de ponto flutuante com precisão dupla, pois ela não é necessária nas aplicações gráficas. Em 2008, foram anunciadas as primeiras GPUs a ter suporte para precisão dupla no hardware. Apesar disso, as operações de precisão simples ainda serão de oito a dez vezes mais rápidas que a precisão dupla, mesmo nessas novas GPUs, enquanto a diferença no desempenho para as CPUs seja limitada a benefícios na transferência de menos bytes no sistema de memória, devido ao uso de dados estreitos.

Embora as GPUs fossem projetadas para um conjunto mais estreito de aplicações, alguns programadores questionaram se poderiam especificar suas aplicações em uma forma que lhes permitisse aproveitar o alto desempenho em potencial das GPUs. Para distinguir esse estilo de uso das GPUs, alguns a chamam de *General Purpose GPUs*, ou *GPGPUs*. Depois de cansar de tentar especificar seus problemas usando as APIs gráficas e linguagens de sombreado de gráficos, eles desenvolveram linguagens de

programação inspiradas em C para permitir que escrevam programas diretamente às GPUs. Um exemplo é Brook, uma linguagem de streaming para GPUs. O próximo passo na facilidade de programação do hardware e da linguagem de programação é a CUDA (Compute Unified Device Architecture) da NVIDIA, que permite que o programador escreva programas em C para execução nas GPUs, embora com algumas restrições. O uso de GPUs para a computação paralela está aumentando com sua crescente facilidade de programação.

Introdução à arquitetura de GPU NVIDIA

O Apêndice A contém muito mais detalhes sobre GPUs e apresenta minuciosamente a arquitetura de GPU da NVIDIA, chamada Tesla. Como as GPUs evoluíram em seu próprio ambiente, elas não apenas têm arquiteturas diferentes, conforme sugerimos anteriormente, mas também têm um conjunto de termos diferente. Quando você aprender os termos da GPU, verá as semelhanças nas técnicas apresentadas nas seções anteriores, como multithreading fine-grained e vetores.

Para ajudá-lo com essa transição ao novo vocabulário, apresentamos uma rápida introdução aos termos e às ideias na arquitetura de GPU Tesla e no ambiente de programação CUDA.

Um chip GPU discreto se encontra em uma placa separada conectada a um PC padrão através da interconexão PCI-Express. As chamadas placa-mãe de GPU são integradas ao chip set da placa mãe, como uma north bridge ou uma south bridge (Capítulo 6).

As GPUs geralmente são oferecidas como uma família de chips em diferentes pontos de desempenho de preço, com todas sendo compatíveis em software. Os chips de GPUs baseadas em Tesla são oferecidas com algo entre 1 e 16 nós, que a NVIDIA chama de *multiprocessadores*. No início de 2008, a maior versão é chamada de GeForce 8800 GTX, que tem 16 multiprocessadores e uma taxa de clock de 1,35GHz. Cada multiprocessador contém oito unidades de ponto flutuante de precisão simples multithreaded e unidades de processamento de inteiros, que a NVIDIA chama de *processadores streaming*.

Como a arquitetura inclui uma instrução de multiplicação-adição de ponto flutuante com precisão simples, o desempenho máximo da multiplicação-adição de precisão simples do chip 8800 GTX é:

$$\begin{aligned} & 16 \text{ MPs} \times \frac{8 \text{ SPs}}{\text{MP}} \times \frac{2 \text{ FLOPs/instr.}}{\text{SP}} \times \frac{1 \text{ instr.}}{\text{clock}} \times \frac{1,35 \times 10^9 \text{ clocks}}{\text{seg.}} \\ & = \frac{16 \times 8 \times 2 \times 1,35 \text{ GFLOPs}}{\text{seg.}} \\ & = \frac{345,6 \text{ GFLOPs}}{\text{seg.}} \end{aligned}$$

Cada um dos 16 multiprocessadores do GeForce 8800 GTX tem um store local gerenciado por software com uma capacidade de 16KB mais 8192 registradores de 32 bits. O sistema de memória do 8800 GTX consiste em seis partições de 900MHz Graphics DDR3 DRAM, cada uma com 8 bytes de largura e com 128 MB de capacidade. O tamanho de memória local é, portanto, 768MB. A largura de banda de pico da memória GDDR3 é

$$6 \times \frac{8 \text{ Bytes}}{\text{transf.}} \times \frac{2 \text{ transf.}}{\text{clock}} \times \frac{0,9 \times 10^9 \text{ clocks}}{\text{seg.}} = \frac{6 \times 8 \times 2 \times 0,9 \text{ GB}}{\text{seg.}} = \frac{86,4 \text{ GB}}{\text{seg.}}$$

Para ocultar a latência da memória, cada processador de streaming tem threads com suporte do hardware. Cada grupo de 32 threads é chamado de *warp*. Um warp é a unidade de escalonamento, e as threads ativas em um warp — até 32 — executam em

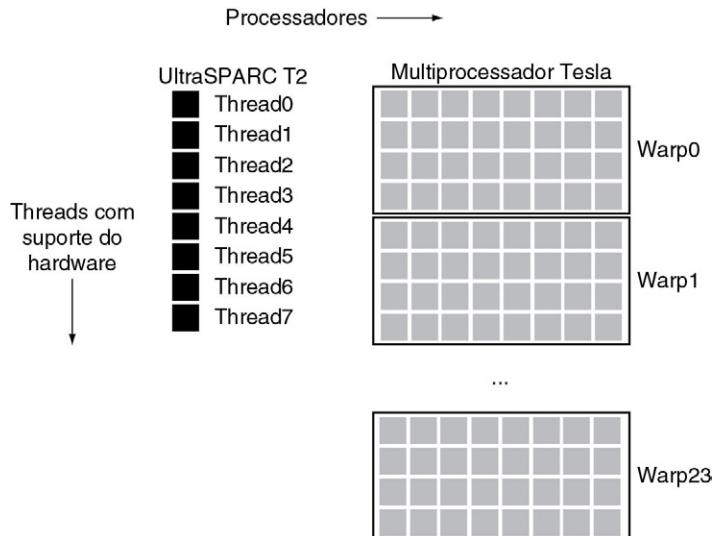


FIGURA 7.7 Comparando o único core de um Sun UltraSPARC T2 (Niagara 2) com um único multiprocessador Tesla. O core T2 é um único processador e usa multithreading com suporte do hardware, com oito threads. O multiprocessador Tesla contém oito processadores streaming e usa multithreading com suporte do hardware com 24 warps de 32 threads (oito processadores vezes quatro ciclos de clock). O T2 pode comutar a cada ciclo de clock, enquanto o Tesla pode comutar apenas a cada dois ou quatro ciclos de clock. Um modo de comparar os dois é que o T2 só pode realizar multithread do processador com o tempo, enquanto o Tesla pode realizar o multithread com o tempo e espaço; ou seja, pelos oito processadores de streaming e também segmentos de quatro ciclos de clock.

paralelo no padrão SIMD. Contudo, a arquitetura multithreaded lida com condições, permitindo que os threads tomem caminhos de desvio diferentes. Quando as threads de um warp tomam caminhos divergentes, o warp executa sequencialmente os caminhos de código com algumas threads inativas, o que faz com que as threads ativas sejam executadas de forma mais lenta. O hardware junta as threads de volta em um warp totalmente ativo assim que os caminhos condicionais são concluídos. Para obter o melhor desempenho, todos as 32 threads de um warp precisam ser executadas juntas em paralelo. Em um estilo semelhante, o hardware também examina os fluxos de endereço vindo de diferentes threads para tentar mesclar as solicitações individuais em menos transferências de bloco de memória, porém maiores, no sentido de aumentar o desempenho da memória.

A Figura 7.7 combina todos esses recursos e compara um multiprocessador Tesla com um core Sun UltraSPARC T2, descrito nas Seções 7.5 e 7.11. Ambos são multithreaded por hardware escalonando-se threads com o tempo, como mostra o eixo vertical. Cada multiprocessador Tesla consiste em oito processadores streaming, cada um executando oito threads paralelas por clock, mostradas horizontalmente. Como dissemos, o melhor desempenho vem quando todas as 32 threads de um warp são executadas juntas em um padrão tipo SIMD, que a arquitetura Tesla chama de Single-Instruction Multiple-Thread (SIMT). SIMT descobre dinamicamente quais threads de um warp podem executar a mesma instrução juntas, e quais threads independentes estão ociosas nesse ciclo. O core T2 contém apenas um único processador multithreaded. Cada ciclo executa uma instrução para uma thread.

O multiprocessador Tesla usa multithreading de hardware fine-grained para escalar 24 warps com o tempo, que aparecem verticalmente em blocos de quatro ciclos de clock. De modo semelhante, o UltraSPARC T2 escala 8 threads com suporte do hardware com o tempo, uma thread por ciclo, mostrado verticalmente. Dessa forma, assim como o hardware T2 comuta entre threads para manter o core T2 ocupado, o hardware Tesla alterna entre warps para manter o multiprocessador Tesla ocupado. A principal diferença é que o core T2 tem um processador que pode alternar as threads a cada ciclo de clock,

enquanto a unidade de comutação mínima dos warps no microprocessador Tesla é de dois ciclos de clock por oito cores streaming. Como Tesla é voltado para programas com muito paralelismo em nível de dados, os projetistas acreditaram que existe pouca diferença de desempenho entre a comutação a cada dois ou quatro ciclos de clock, contra cada ciclo de clock, e o hardware se tornou muito mais simples restringindo a frequência de comutação.

O ambiente de programação CUDA também possui sua própria terminologia. Um programa CUDA é um programa C/C++ unificado para um sistema de CPU e GPU heterogêneo. Ele é executado na CPU e despacha o trabalho paralelo para a GPU. Esse trabalho consiste em uma transferência de dados da memória principal e um *despacho de thread*. Uma thread é um pedaço do programa para a GPU. Os programadores especificam o número de threads em um *bloco de threads*, e o número de blocos de threads que eles querem começar a executar na GPU. O motivo para os programadores se importarem com os blocos de threads é que todos os threads no bloco de threads são escalonados para serem executados no mesmo multiprocessador, de modo que todos compartilham a mesma memória local. Assim, eles podem se comunicar por meio de loads e stores, em vez de mensagens. O compilador CUDA aloca registradores a cada thread, sob a restrição de que os registradores por thread vezes threads por bloco de threads não ultrapasse os 8192 registradores por multiprocessador.

Um bloco de threads pode ter até 512 threads. Cada grupo de 32 threads em um bloco de threads é empacotado em warps. Grandes blocos de threads possuem melhor eficiência do que os pequenos, e eles podem ser tão pequenos quanto uma única thread. Como dissemos, os blocos de threads e warps com menos de 32 threads operam de modo menos eficiente do que os completos.

Um escalonador de hardware tenta escalar múltiplos blocos de threads por multiprocessador quando for possível. Se fizer isso, o escalonador também particiona o store local de 16KB dinamicamente entre os diferentes blocos de threads.

Colocando as GPUs em perspectiva

GPUs como a arquitetura Tesla da NVIDIA não se encaixam muito bem nas classificações anteriores dos computadores, como a [Figura 7.6](#). Claramente, o GeForce 8800 GTX, com 16 multiprocessadores Tesla, é um MIMD. A questão é como classificar cada um dos multiprocessadores Tesla e os oitos processadores principais que compõem um multiprocessador Tesla.

Lembre-se de que já dissemos que o SIMD funcionou melhor com loops for e pior com instruções case e switch. O Tesla visa o alto desempenho para o paralelismo em nível de dados, enquanto facilita para os programadores lidarem com cases paralelos independentes em nível de thread. O Tesla permite que o programador pense que o multiprocessador é um MIMD multithreaded de oito processadores streaming, mas o hardware tenta reunir os oito processadores streaming para que atuem no padrão SIMT quando múltiplos threads do mesmo warp podem estar executando juntos. Quando os threads operam independentemente e seguem um caminho de execução independente, eles são executados mais lentamente do que no padrão SIMT, pois todos os 32 threads de um warp compartilham uma única unidade de busca de instrução. Se todos os 32 threads de um warp estivessem executando instruções independentes, cada thread operaria em 1/16 do desempenho máximo de um warp completo de 32 threads executando em oito processadores streaming por quatro clocks.

Assim, cada thread independente tem seu próprio PC efetivo, de modo que os programadores podem pensar no multiprocessador Tesla como MIMD, mas os programadores precisam ter o cuidado de escrever instruções de fluxo de controle que permitem que o hardware SIMT execute programas CUDA no padrão SIMD para oferecer o desempenho desejado.

Ao contrário das arquiteturas de vetor, que contam com um compilador de vetorização para reconhecer o paralelismo em nível de dados em tempo de compilação e gerar

instruções de vetor, as implementações de hardware da arquitetura Tesla descobrem o paralelismo em nível de dados entre os threads em tempo de execução. Assim, GPUs Tesla não precisam de compiladores de vetorização, e tornam mais fácil para o programador lidar com as partes do programa que não possuem paralelismo em nível de dados. Para explicar melhor essa técnica exclusiva, a [Figura 7.8](#) coloca as GPUs em uma classificação que compara o paralelismo em nível de instrução com o paralelismo em nível de dados e se ele é descoberto em tempo de compilação ou execução. Essa categorização é uma indicação de que a GPU Tesla está ganhando terreno na arquitetura do computador.

	Estático: descoberto em tempo de compilação	Dinâmico: descoberto em tempo de execução
Paralelismo em nível de instrução	VLIW	Superescalar
Paralelismo em nível de dados	SIMD ou Vetor	Multiprocessador Tesla

FIGURA 7.8 Categorização de hardware das arquiteturas de processador e exemplos baseados em estático versus dinâmico e ILP versus DLP.

Verifique você mesmo

Verdadeiro ou falso: GPUs contam com chips de DRAM gráficos para reduzir a latência da memória e, portanto, aumentar o desempenho em aplicações gráficas.

7.8

Introdução às topologias de rede multiprocessador

Os chips multicore exigem que as redes nos chips conectem os cores. Esta seção revisa os prós e os contras de diferentes redes de multiprocessadores.

Os custos de rede incluem o número de switches, o número de links em um switch que se conectam à rede, a largura (número de bits) por link, o tamanho dos links quando a rede é mapeada no chip. Por exemplo, alguns cores podem ser adjacentes e outros podem estar no outro lado do chip. O desempenho da rede também tem muitas faces. Ele inclui a latência em uma rede não carregada para enviar e receber uma mensagem, a vazão em termos do número máximo de mensagens que podem ser transmitidas em determinado período de tempo, atrasos causados pela disputa por uma parte da rede, e desempenho variável dependendo do padrão de comunicação. Outra obrigação da rede pode ser tolerância a falhas, pois os sistemas podem ter de operar na presença de componentes defeituosos. Finalmente, nesta era de chips de potência limitada, a eficiência de potência das diferentes organizações pode superar outros aspectos.

As redes normalmente são desenhadas como gráficos, com cada arco do gráfico representando um link da rede de comunicação. O nó processador-memória aparece como um quadrado preto, e o switch aparece como um círculo colorido. Nesta seção, todos os links são *bidirecionais*; ou seja, a informação pode fluir em qualquer direção. Todas as redes consistem em *switches* cujos links vão para os nós processador-memória e para outros switches. A primeira melhoria em relação a um barramento é uma rede que conecta uma sequência de nós:



Essa topologia é chamada de *anel*. Como alguns nós não são conectados diretamente, algumas mensagens terão um salto por nós intermediários até que cheguem ao destino final.

Diferente de um barramento, um anel é capaz de realizar muitas transferências simultâneas. Como existem diversas topologias para escolher, métricas de desempenho são necessárias a fim de distinguir esses projetos. Duas são comuns. A primeira é a **largura de banda de rede total**, que é a largura de banda de cada link multiplicado pelo número de links, e representa o melhor caso. Para a rede de anel apresentada, com P processadores, a largura de banda de rede total seria P vezes a largura de banda do link; a largura de banda de rede total de um barramento é a largura de banda desse barramento, ou duas vezes a largura de banda desse link.

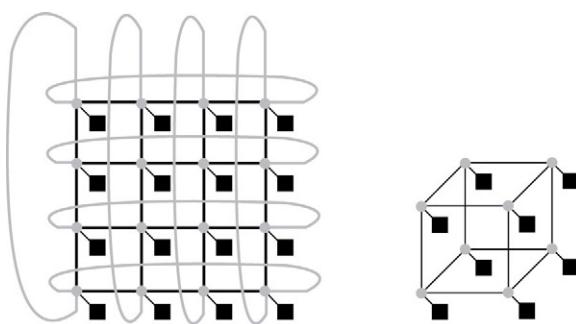
Para balancear esse melhor caso, incluímos outra métrica que é mais próxima do pior caso: a **largura de banda da corte**. Esta é calculada dividindo-se a máquina em duas partes, cada uma com metade dos nós. Depois você soma a largura de banda dos links que cruzam essa linha divisória imaginária. A largura de banda de corte de um anel é duas vezes a largura de banda do link, e é uma vez a largura de banda de link para o barramento. Se um único link for tão rápido quanto o barramento, o anel tem apenas o dobro da velocidade de um barramento no pior caso, mas é P vezes mais rápido no melhor caso.

Como algumas topologias de rede não são simétricas, surge a questão de onde desenhar a linha imaginária quando fizer o corte da máquina. Essa é uma métrica do pior caso, de modo que a resposta é escolher a divisão que gera o desempenho de rede mais pessimista. Em outras palavras, calcule todas as larguras de banda de corte e escolha a menor. Tomamos a visão pessimista porque programas paralelos normalmente são limitados pelo elo mais fraco na cadeia de comunicação.

No outro extremo de um anel está a **rede totalmente conectada**, em que cada processador tem um link bidirecional com cada outro processador. Para as redes totalmente conectadas, a largura de banda de rede total é $P \times (P - 1)/2$, e a largura de banda de corte é $(P/2)^2$.

A tremenda melhoria no desempenho das redes totalmente conectadas é anulada pelo enorme aumento no custo. Isso inspira os engenheiros a inventarem novas topologias que estão entre o custo dos anéis e o desempenho das redes totalmente conectadas. A avaliação do sucesso depende em grande parte da natureza da comunicação na carga de trabalho de programas paralelos executados na máquina.

O número de topologias diferentes que foram discutidas nas diversas publicações seria difícil de contar, mas somente uma minoria foi utilizada em processadores paralelos comerciais. A [Figura 7.9](#) ilustra duas das topologias mais comuns. As máquinas reais constantemente acrescentam links extras a essas topologias simples para melhorar o desempenho e a confiabilidade.



a. Grade 2-D ou malha de 16 nós

b. Árvore de cubo n com 8 nós ($8 = 2^3$, de modo que $n = 3$)

FIGURA 7.9 Topologias de rede que apareceram nos processadores paralelos comerciais. Os círculos coloridos representam switches, e os quadrados pretos representam nós processador-memória. Embora um switch tenha muitos links, geralmente apenas um vai para o processador. A topologia de cubo n booleana é uma interconexão n -dimensional com 2^n nós, exigindo n links por switch (mais um para o processador) e, portanto, n nós com o vizinho mais próximo. Constantemente, essas topologias básicas têm sido suplementadas com arcos extras para melhorar o desempenho e a confiabilidade.

largura de banda de rede Informalmente, a taxa de transferência de pico de uma rede; pode se referir à velocidade de um único link ou a taxa de transferência coletiva de todos os links na rede.

largura de banda de corte A largura de banda entre duas partes iguais de um multiprocessador. Essa medida é para uma divisão do multiprocessador no pior caso.

rede totalmente conectada Uma rede que conecta nós processador-memória fornecendo um link de comunicação dedicado entre cada nó.

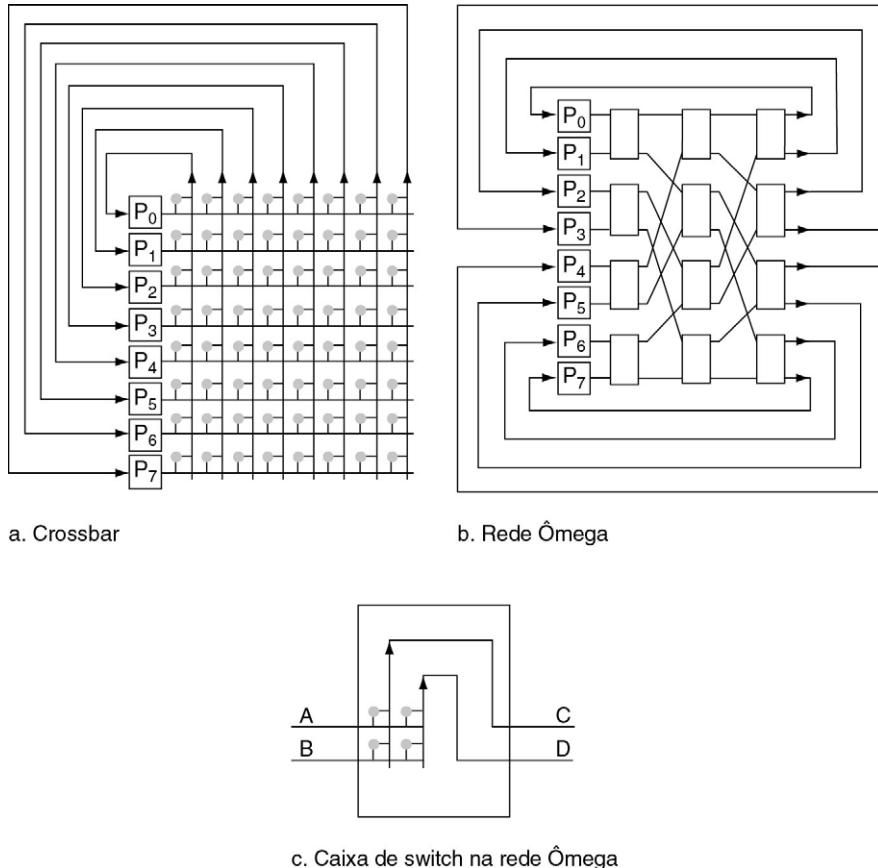


FIGURA 7.10 Topologias comuns de rede multiestágio para oito nós. Os switches nesses desenhos são mais simples do que nos desenhos anteriores, pois os links são unidirecionais; os dados entram na parte de baixo e saem pelo link da direita. A caixa de switch em c pode passar A para C e B para D ou B para C e A para D. O crossbar usa n^2 switches, em que n é o número de processadores, enquanto a rede Ômega usa $2n \log_2 n$ das caixas de switch grandes, cada uma composta logicamente por quatro dos switches menores. Nesse caso, o crossbar utiliza 64 switches contra 12 caixas de switch, ou 48 switches, na rede Ômega. O crossbar, porém, pode aceitar qualquer combinação de mensagens entre os processadores, enquanto a rede Ômega não pode.

rede multiestágio Uma rede que fornece um pequeno switch em cada nó.

rede totalmente conectada Uma rede que conecta nós processadores de memórias por meio do fornecimento de um link de comunicação dedicado entre cada nó.

rede crossbar Uma rede que permite que qualquer nó se comunique com qualquer outro nó em uma passada pela rede.

Uma alternativa a colocar um processador em cada nó de uma rede é deixar apenas o switch em alguns desses nós. Os switches são menores que os nós processador-memória-switch, e assim podem ser compactados de forma mais densa, reduzindo assim a distância e aumentando o desempenho. Essas redes normalmente são chamadas **redes multiestágio** para refletir as múltiplas etapas em que uma mensagem pode trafegar. Os tipos de redes multiestágio são tão numerosos quanto as redes de único estágio; A Figura 7.10 ilustra duas das organizações multiestágio mais comuns. Uma **rede totalmente conectada** ou **crossbar** permite que qualquer nó se comunique com qualquer outro nó em uma passada pela rede. Uma **rede Ômega** usa menos hardware do que a rede crossbar ($2n \log_2 n$ contra n^2 switches), mas pode ocorrer disputa entre as mensagens, dependendo do padrão de comunicação. Por exemplo, a rede Ômega na Figura 7.10 não pode enviar uma mensagem de P₀ a P₆ ao mesmo tempo em que envia uma mensagem de P₁ a P₇.

Implementando topologias de rede

Esta análise simples de todas as redes nesta seção ignora considerações práticas importantes na construção de uma rede. A distância de cada link afeta o custo de comunicação em uma alta taxa de clock — geralmente, quanto maior a distância, mais dispendioso é trabalhar em uma taxa de clock alta. Distâncias mais curtas também facilitam a atribuição de mais fios

no link, pois a potência para conduzir por muitos fios a partir de um chip é menor se os fios forem curtos. Fios mais curtos também são mais baratos do que os mais longos. Outra limitação prática é que os desenhos tridimensionais precisam ser mapeados nos chips que são basicamente mídia bidimensional. A preocupação final é com a potência. Problemas de potência podem forçar chips multicore a contarem com topologias de grade simples, por exemplo. A conclusão é que as topologias que parecem ser elegantes quando esboçadas em um quadro podem ser impraticáveis quando construídas em silício.

7.9

Benchmarks de multiprocessador

Como vimos no Capítulo 1, sistemas de benchmarking sempre é um assunto delicado, pois é uma forma altamente visível de tentar determinar qual sistema é melhor. Os resultados afetam não apenas as vendas de sistemas comerciais, mas também a reputação dos projetistas desses sistemas. Logo, os participantes querem ganhar a competição, mas eles também querem ter certeza de que, se alguém mais ganhar, eles mereçam ganhar porque possuem um sistema genuinamente melhor. Esse desejo leva a regras para garantir que os resultados do benchmark não sejam simplesmente truques de engenharia para esse benchmark, mas, em vez disso, avanços que melhoraram o desempenho das aplicações reais.

Para evitar possíveis truques, uma boa regra é que você não pode mudar o benchmark. O código-fonte e os conjuntos de dados são fixos, e existe uma única resposta apropriada. Qualquer desvio dessas regras torna os resultados inválidos.

Muitos benchmarks de multiprocessador seguem essas tradições. Uma exceção comum é ser capaz de aumentar o tamanho do problema de modo que você possa executar o benchmark em sistemas com um número bem diferente de processadores. Ou seja, muitos benchmarks permitem pouca facilidade de expansão, em vez de exigir muita facilidade, embora você deva ter cuidado ao comparar resultados para programas executando problemas com diferentes tamanhos.

A [Figura 7.11](#) é um resumo de vários benchmarks paralelos, também descritos a seguir:

- *Linpack* é uma coleção de rotinas de álgebra linear, e as rotinas para realizar a eliminação Gaussiana constituem o que é conhecido como benchmark Linpack. A rotina DAXPY no exemplo da Seção 7.6 representa uma pequena fração do código fonte do benchmark Linpack, mas é responsável pela maior parte do tempo de execução do benchmark. Ele permite expansão fraca, deixando que o usuário escolha qualquer tamanho de problema. Além do mais, ele permite que o usuário reescreva o Linpack em qualquer formato e em qualquer linguagem, desde que calcule o resultado apropriado. Duas vezes por ano, os 500 computadores com o desempenho Linpack mais rápido são publicados em www.top500.org. O primeiro nessa lista é considerado pela imprensa como o computador mais rápido do mundo.
- *SPECrate* é uma métrica de vazão baseada nos benchmarks SPEC CPU, como SPEC CPU 2006 (veja Capítulo 1). Em vez de relatar o desempenho dos programas individuais, SPECrate executa muitas cópias do programa simultaneamente. Assim, ele mede o paralelismo em nível de tarefa, pois não há comunicação entre as tarefas. Você pode executar tantas cópias dos programas quantas desejar, de modo que essa novamente é uma forma de expansão fraca.
- *SPLASH* e *SPLASH 2* (Stanford Parallel Applications for Shared Memory) foram esforços realizados por pesquisadores na Stanford University na década de 1990 para reunir um conjunto de benchmarks paralelo, semelhante em objetivos ao conjunto de benchmarks SPEC CPU. Ele inclui kernels e aplicações, além de muitos da comunidade de computação de alto desempenho. Esse benchmark requer expansão forte, embora venha com dois conjuntos de dados.

Benchmark	Expansão?	Reprograma?	Descrição
Linpack	Fraca	Sim	Matriz densidade linear em álgebra [Dongarra, 1979]
SPECrate	Fraca	Não	Trabalho independente de paralelismo [Henning, 2007]
Stanford Parallel Applications for Shared Memory SPLASH 2 [Woo et al., 1995]	Forte (embora ofereça dois tamanhos de problema)	Não	Complexo 1D FFT Decomposição LU bloqueada Fatorização de Cholesky esparsa bloqueada Radix Sort inteiro Barnes-Hut Multipole Rápida e Adaptativa Simulação de oceano Radiosidade hierárquica Traçador de raios Renderizador de Volume Simulação de água com estrutura de dados espaciais Simulação de água sem estrutura de dados espaciais
NAS Parallel Benchmarks [Bailey et al., 1991]	Fraca	Sim (C ou Fortran somente)	EP: embarcaçosamente paralelo MG: multigrid simplificado CG: grid não estruturado para um método gradiente conjugado FT: equação diferencial parcial 3-D utilizando FFTs IS: maior sort de inteiro
PARSEC Benchmark Suite [Bienia et al., 2008]	Fraca	Não	Blackscholes – Opção de especificação com o equação diferencial parcial (PDE) Black-Scholes Bodytrack – Rastreamento do corpo de uma pessoa Canneal – Recozimento simulado cache-aware para otimizar o roteamento Dedup – Compressão de próxima geração com deduplicação de dados. Facesim – Simulação dos movimentos de um rosto humano Ferret – Servidor de busca de conteúdos similares Fluidanimate – Dinâmicas de fluidos para uma animação com método SPH Freqmine – Mineração frequente de conjunto de itens Streamcluster – Agrupamento on-line de um input stream Swaptions – Precificação de um portfolio de swaptions Vips – Processamento de imagem x264 – Codificação de vídeo H.264
Berkeley Design Patterns [Asanovic et al., 2006]	Forte ou fraca	Sim	Máquina em estado finito Lógica Combinatória Gráfico Transversal Grid estruturado Matriz densidade Matriz esparsa Métodos espetrais (FFT) Programação dinâmica N-Body MapReduce Backtrack/Desvio e Limite Inferência modelo gráfica Grid não estruturado

FIGURA 7.11 Exemplos de benchmarks paralelos.

Pthreads Uma API do UNIX para criar e manipular threads. Ela vem com uma biblioteca.

OpenMP Uma API para multiprocessamento de memória compartilhada em C, C++ ou Fortran, que é executada em plataformas UNIX e Microsoft. Ela inclui diretivas de compilador, uma biblioteca e diretivas de runtime.

- Os *benchmarks paralelos NAS* (*NASA Advanced Supercomputing*) foram outra tentativa da década de 1990 de realizar o benchmark em multiprocessadores. Tomados da dinâmica de fluidos computacional, eles consistem em cinco kernels, e permitem expansão fraca, definindo alguns poucos conjuntos de dados. Assim como o Linpack, esses benchmarks podem ser reescritos, mas as regras exigem que a linguagem de programação só possa ser C ou Fortran.
- O recente *conjunto de benchmarks PARSEC* (*Princeton Application Repository for Shared Memory Computers*) consiste em programas multithreaded que usam **Pthreads** (POSIX threads) e **OpenMP** (Open MultiProcessing). Eles focalizam mercados emergentes e consistem em nove aplicações e três kernels. Oito contam com paralelismo de dados, três contam com paralelismo em pipeline, e um com paralelismo não estruturado.

O lado negativo dessas restrições tradicionais dos benchmarks é que a inovação é limitada principalmente a arquitetura e compilador. Estruturas de dados melhores, algoritmos, linguagens de programação e assim por diante geralmente não podem ser usados, pois isso geraria um resultado ilusório. O sistema poderia ganhar, digamos, por causa do algoritmo, e não por causa do hardware ou do compilador.

Embora essas orientações sejam compreensíveis quando os alicerces da computação são relativamente estáveis — como eram na década de 1990 e na primeira metade desta década —, elas são indesejáveis no início de uma revolução. Para que essa revolução tenha sucesso, precisamos encorajar a inovação em todos os níveis.

Uma técnica recente foi defendida pelos pesquisadores na Universidade da Califórnia em Berkeley. Eles identificaram 13 padrões de projeto que afirmam que será parte das aplicações do futuro. Esses padrões de projeto são implementados por frameworks ou kernels. Alguns exemplos são matrizes esparsas, grade estruturada, máquinas de estados finitos, redução de mapa e travessia de gráfico. Mantendo as definições em um alto nível, elas esperam encorajar inovações em qualquer nível do sistema. Assim, o sistema com o solucionador de matriz esparsa mais rápido está livre para usar qualquer estrutura de dados, algoritmo e linguagem de programação, além de novas arquiteturas e compiladores. Veremos exemplos desses benchmarks na Seção 7.11.

Verdadeiro ou falso: a principal desvantagem com as técnicas convencionais de benchmarks para computadores paralelos é que as regras que garantem justiça também suprimem a inovação.

**Verifique
você mesmo**

7.10

Roofline: um modelo de desempenho simples

Esta seção é baseada em um artigo de Williams e Patterson [2008]. No passado, a sabedoria convencional em arquitetura de computador levou a projetos de microprocessador semelhantes. Quase todo computador desktop e servidor utilizava caches, pipelining, emissão de instrução superescalar, previsão de desvio e execução fora de ordem. Os conjuntos de instruções variavam, mas os microprocessadores eram todos da mesma escola de projeto.

A passagem para multicore provavelmente significa que os microprocessadores se tornarão mais diversos, pois não existe sabedoria convencional sobre qual arquitetura tornará mais fácil escrever programas de processamento paralelo corretos, que executem de forma eficiente e se expandam à medida que o número de cores aumenta com o tempo. Além do mais, à medida que o número de cores por chip aumenta, um único fabricante provavelmente oferecerá diferentes números de cores por chip em diferentes pontos de preço ao mesmo tempo.

Dada a diversidade crescente, seria especialmente útil se tivéssemos um modelo simples que oferecesse ideias para o desempenho de diferentes projetos. Ele não precisa ser perfeito, apenas criterioso.

O modelo 3Cs do Capítulo 5 é uma analogia. Ele não é um modelo perfeito, pois ignora fatores potencialmente importantes, como tamanho de bloco, diretiva de alocação de bloco e diretiva de substituição de bloco. Além do mais, ele possui algumas esquisitices. Por exemplo, uma falha pode ser atribuída à capacidade em um projeto e a uma falha de conflito em outra cache do mesmo tamanho. Mesmo assim, o modelo 3Cs tem sido popular há 20 anos, pois oferece ideias para o comportamento dos programas, ajudando arquitetos e programadores a melhorarem suas criações com base em concepções desse modelo.

Para descobrir esse modelo, vamos começar com os 13 padrões de projeto de Berkeley, na Figura 7.9. A ideia dos padrões de projeto é que o desempenho de determinada aplicação é na realidade a soma ponderada de vários kernels que implementam esses padrões de projeto. Vamos avaliar os kernels individuais aqui, mas lembre-se de que as aplicações reais são combinações de muitos kernels.

Embora haja versões com diferentes tipos de dados, ponto flutuante é comum em várias implementações. Logo, o desempenho de pico em ponto flutuante é um limite sobre a velocidade desses kernels em determinado computador. Para chips multicore, o desempenho de pico em ponto flutuante é o desempenho de pico coletivo de todos os cores no chip. Se houvesse múltiplos microprocessadores no sistema, você multiplicaria o pico por chip pelo número total de chips.

As demandas no sistema de memória podem ser estimadas dividindo-se esse desempenho de pico em ponto flutuante pelo número médio de operações de ponto flutuante por byte acessado:

$$\frac{\text{Operações de PF/Seg}}{\text{Operações de PF/Byte}} = \text{Bytes/Seg}$$

intensidade aritmética A razão entre as operações de ponto flutuante em um programa e o número de bytes de dados acessados por um programa a partir da memória principal.

A razão entre operações de ponto flutuante por byte de memória acessada é chamada de **intensidade aritmética**. Ela pode ser calculada apanhando-se o número total de operações de ponto flutuante para um programa dividido pelo número total de bytes de dados transferidos para a memória principal durante a execução do programa. A [Figura 7.12](#) mostra a intensidade aritmética de vários dos padrões de projeto de Berkeley da [Figura 7.11](#).

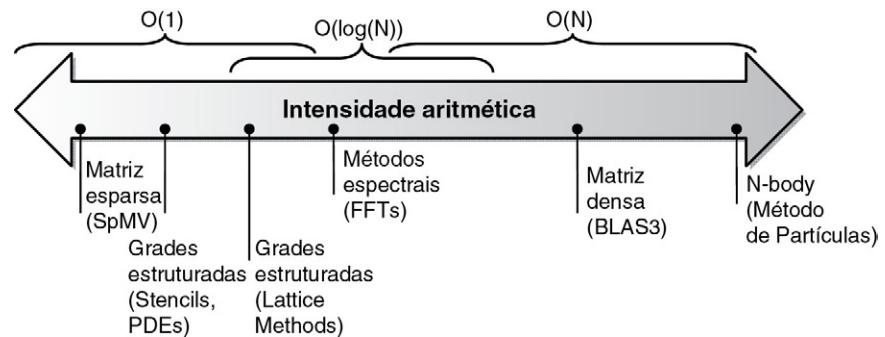


FIGURA 7.12 Intensidade aritmética, especificada como o número de operações de ponto flutuante para executar o programa dividido pelo número de bytes acessados na memória principal [Williams, Patterson, 2008]. Alguns kernels possuem uma intensidade aritmética que se expande com o tamanho do problema, como Matrizes Densas, mas existem muitos kernels com intensidades aritméticas independentes do tamanho do problema. Para os kernels nesse primeiro caso, a expansão fraca pode levar a diferentes resultados, pois coloca muito menos demanda sobre o sistema de memória.

O modelo roofline

O modelo simples proposto reúne desempenho de ponto flutuante, intensidade aritmética e desempenho da memória em um gráfico bidimensional [Williams, Patterson, 2008]. O desempenho de pico em ponto flutuante pode ser encontrado usando as especificações de hardware mencionadas anteriormente. O conjunto dos kernels que consideramos aqui não se encaixa em caches no chip, de modo que o desempenho de pico da memória pode ser definido pelo sistema de memória por trás das caches. Um modo de encontrar o desempenho de pico da memória é o benchmark Stream. (Veja a seção *Detalhamento* na Seção “Projetando o sistema de memória para suportar caches”, no Capítulo 5.)

A [Figura 7.13](#) mostra o modelo, que é feito uma vez para um computador, e não para cada kernel. O eixo-Y vertical é o desempenho de ponto flutuante alcançável de 0,5 a 64,0 GFLOPs/segundo. O eixo-X horizontal é a intensidade aritmética, variando de 1/8 FLOPs/DRAM acessados por byte a 16 FLOPs/DRAM acessados por byte. Observe que o gráfico é uma escala log-log.

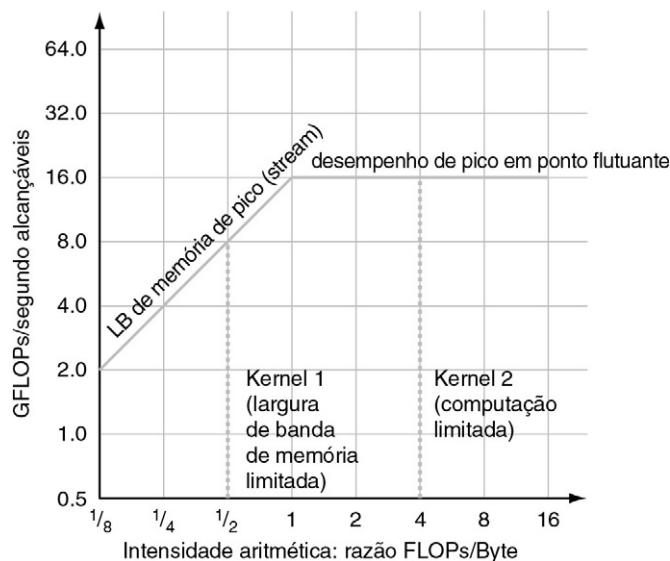


FIGURA 7.13 Modelo Roofline [Williams, Patterson, 2008]. Este exemplo tem um desempenho de pico de 16 GFLOPs/seg e uma largura de banda de memória de pico de 16GB/seg do benchmark Stream. (Como o Stream na realidade tem quatro medições, essa linha é a média das quatro.) A linha vertical pontilhada à esquerda representa o Kernel 1, que tem uma intensidade aritmética de 0,5 FLOPs/byte. Ela é limitada pela largura de banda de memória a não mais que 8 GFLOPs/seg nesse Opteron X2. A linha vertical pontilhada à direita representa o Kernel 2, que tem uma intensidade aritmética de 4 FLOPs/byte. Ela é limitada apenas computacionalmente a 16 GFLOPs/seg. (Esses dados são baseados no AMD Opteron X2 (Revision F) usando dual cores executando a 2GHz em um sistema dual socket.)

Para determinado kernel, podemos encontrar um ponto no eixo X com base em sua intensidade aritmética. Se desenhássemos uma linha vertical passando por esse ponto, o desempenho do kernel nesse computador teria de ficar em algum lugar nessa linha. Podemos desenhar uma linha horizontal mostrando o desempenho de pico em ponto flutuante do computador. Obviamente, o desempenho real em ponto flutuante não pode ser maior que a linha horizontal, pois esse é um limite do hardware.

Como poderíamos desenhar o desempenho de pico da memória? Como o eixo X é FLOPs/byte e o eixo Y é FLOPs/segundo, bytes/segundo é simplesmente uma linha diagonal em um ângulo de 45 graus nessa figura. Logo, podemos desenhar uma terceira linha que mostre o desempenho máximo em ponto flutuante que o sistema de memória desse computador pode suportar para determinada intensidade aritmética. Podemos expressar os limites como uma fórmula para desenhar a linha no gráfico da Figura 7.13:

$$\text{GFLOPs/seg alcançável} = \text{Min}(\text{LB memória de pico} \times \text{Intensidade aritmética}, \text{Desempenho de pico em ponto flutuante})$$

As linhas horizontal e diagonal dão nome a esse modelo simples e indicam seu valor. A “roofline” define um limite superior no desempenho de um kernel, dependendo de sua intensidade aritmética. Se pensarmos na intensidade aritmética como um poste que atinge o telhado, ou ele atinge a parte plana do telhado, o que significa que o desempenho é computacionalmente limitado, ou atinge a parte inclinada do telhado, o que significa que por fim está limitado pela largura de banda da memória. Na Figura 7.13, o kernel 2 é um exemplo do primeiro, e o kernel 1 é um exemplo do segundo. Dada uma roofline de um computador, você pode aplicá-la repetidamente, pois ela não varia por kernel.

Observe que o “ponto de cumeeira”, em que os telhados diagonal e horizontal se encontram, oferece uma percepção interessante para o computador. Se for muito longe à direita, então somente os kernels com intensidade aritmética muito alta podem alcançar o desempenho máximo desse computador. Se for muito à esquerda, então quase todo kernel poderá potencialmente atingir o desempenho máximo. Veremos exemplos de ambos em breve.

Comparando duas gerações de Opterons

O AMD Opteron X4 (Barcelona) com quatro cores é o sucessor do Opteron X2 com dois cores. Para simplificar o projeto da placa, eles usam o mesmo soquete. Logo, eles possuem os mesmos canais de DRAM e, portanto, a mesma largura de banda de memória de pico. Além de dobrar o número de cores, o Opteron X4 também tem o dobro do desempenho de pico em ponto flutuante por core: os cores do Opteron X4 podem emitir duas instruções SSE2 de ponto flutuante por ciclo de clock, enquanto os cores do Opteron X2 emite no máximo uma. Como os dois sistemas que estamos comparando possuem taxas de clock semelhantes — 2,2GHz para o Opteron X2 *versus* 2,3GHz para o Opteron X4 — o Opteron X4 tem mais de quatro vezes do desempenho de ponto flutuante de pico do Opteron X2 com a mesma largura de banda de DRAM. O Opteron X4 também tem uma cache L3 de 2MB, que não é encontrada no Opteron X2.

A Figura 7.14 compara os modelos roofline para ambos os sistemas. Como poderíamos esperar, o ponto de cumeeira passa de 1 no Opteron X2 para 5 no Opteron X4. Logo, para ver um ganho de desempenho na próxima geração, os kernels precisam de uma intensidade aritmética maior que 1, ou seus conjuntos de trabalho terão de caber nas caches do Opteron X4.

O modelo roofline oferece um limite superior para o desempenho. Suponha que seu programa esteja muito abaixo desse limite. Que otimizações você deverá realizar, e em que ordem?

Para reduzir os gargalos computacionais, as duas otimizações a seguir podem ajudar a quase todo kernel:

1. *Mix de operações de ponto flutuante.* O desempenho de pico em ponto flutuante para um computador normalmente exige um número igual de adições e multiplicações quase simultâneas. Esse equilíbrio é necessário ou porque o computador admite uma instrução multiplicação-adição unificada (veja a seção *Detalhamento* na Seção “Aritmética de precisão”, no Capítulo 3) ou porque a unidade de ponto flutuante tem um número igual de somadores de ponto flutuante e multiplicadores de ponto flutuante. O melhor desempenho também requer que uma fração significativa do mix de instruções seja operações de ponto flutuante, e não instruções de inteiros.

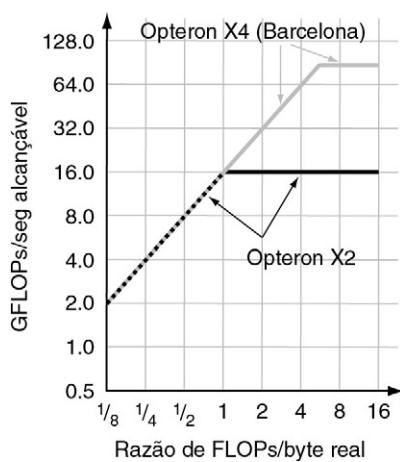


FIGURA 7.14 Modelos roofline de duas gerações de Opterons. A roofline do Opteron X2, que é a mesma que na Figura 7.11, está em preto, e a roofline do Opteron X4 está colorida. O ponto de cumeeira maior do Opteron X4 significa que os kernels que eram computacionalmente limitados no Opteron X2 poderiam ser limitados pelo desempenho da memória no Opteron X4.

2. *Melhore o paralelismo em nível de instrução e aplique SIMD.* Para arquiteturas superescalares, o desempenho mais alto surge com a busca, execução e commit de três a quatro instruções por ciclo de clock (veja Capítulo 4). O objetivo aqui é melhorar o código do compilador para aumentar o ILP. Uma forma é desdobrando loops. Para as arquiteturas x86, uma única instrução SIMD pode operar sobre pares de operandos de precisão dupla, de modo que elas devem ser usadas sempre que possível.

Para reduzir os gargalos da memória, as duas otimizações a seguir podem ajudar:

1. *Pré-busca do software.* Normalmente, o desempenho mais alto exige manter muitas operações da memória no ato, que é mais fácil de se fazer executando instruções de pré-busca do software, em vez de esperar até que os dados sejam exigidos pela computação.
2. *Afinidade de memória.* A maioria dos microprocessadores de hoje inclui um controlador de memória no mesmo chip com o microprocessador. Se o sistema tiver múltiplos chips, isso significa que os mesmos endereços vão para a DRAM que é local a um chip, e o restante requer que os acessos pela interconexão do chip acessem a DRAM que é local a outro chip. O segundo caso reduz o desempenho. Essa otimização tenta alocar dados e as threads encarregadas de operar sobre esses dados no mesmo par memória-processador, de modo que os processadores raramente precisam acessar a memória dos outros chips.

O modelo roofline pode ajudar a decidir quais dessas otimizações serão realizadas e em que ordem. Podemos pensar em cada uma dessas otimizações como um “teto” abaixo da roofline apropriada, significando que você não pode ultrapassar um teto sem realizar a otimização associada.

A roofline computacional pode ser encontrada nos manuais, e a roofline de memória pode ser encontrada executando-se o benchmark Stream. Os tetos computacionais, como o equilíbrio de ponto flutuante, também vêm dos manuais desse computador. O teto de memória exige a execução de experimentos em cada computador, para determinar a lacuna entre eles. A boa notícia é que esse processo só precisa ser feito uma vez por computador, pois quando alguém caracterizar os tetos de um computador, todos poderão usar os resultados a fim de priorizar suas otimizações para esse computador.

A Figura 7.15 acrescenta tetos ao modelo roofline da Figura 7.13, mostrando os tetos computacionais no gráfico superior e os tetos da largura de banda de memória no gráfico inferior. Embora os tetos mais altos não sejam rotulados com as duas otimizações, isso está implícito nessa figura; para ultrapassar o teto mais alto, você já deverá ter ultrapassado todos os tetos abaixo.

A espessura da lacuna entre o teto e o próximo limite mais alto é a recompensa por tentar essa otimização. Assim, a Figura 7.15 sugere que a otimização 2, que melhora o ILP, tem um grande benefício para melhorar a computação nesse computador, e a otimização 4, que melhora a afinidade de memória, tem um grande benefício para melhorar a largura de banda da memória nesse computador.

A Figura 7.16 combina os tetos da Figura 7.15 em um único gráfico. A intensidade aritmética de um kernel determina a região de otimização, que, por sua vez, sugere quais otimizações tentar. Observe que as otimizações computacionais e as otimizações de largura de banda da memória se sobrepõem para grande parte da intensidade aritmética. Três regiões são sombreadas de formas diferentes na Figura 7.16 para indicar as diferentes estratégias de otimização. Por exemplo, o Kernel 2 cai no trapezóide azul à direita, que sugere trabalhar apenas nas otimizações computacionais. O Kernel 1 cai no paralelogramo azul-cinza no meio, que sugere tentar os dois tipos de otimização. Além do mais, ele sugere começar com as otimizações 2 e 4. Observe que as linhas verticais do Kernel 1 caem abaixo da otimização de desequilíbrio de ponto flutuante, de modo que a otimização 1 pode ser desnecessária. Se um kernel caísse no triângulo cinza no canto inferior esquerdo, isso sugeriria tentar apenas otimizações de memória.

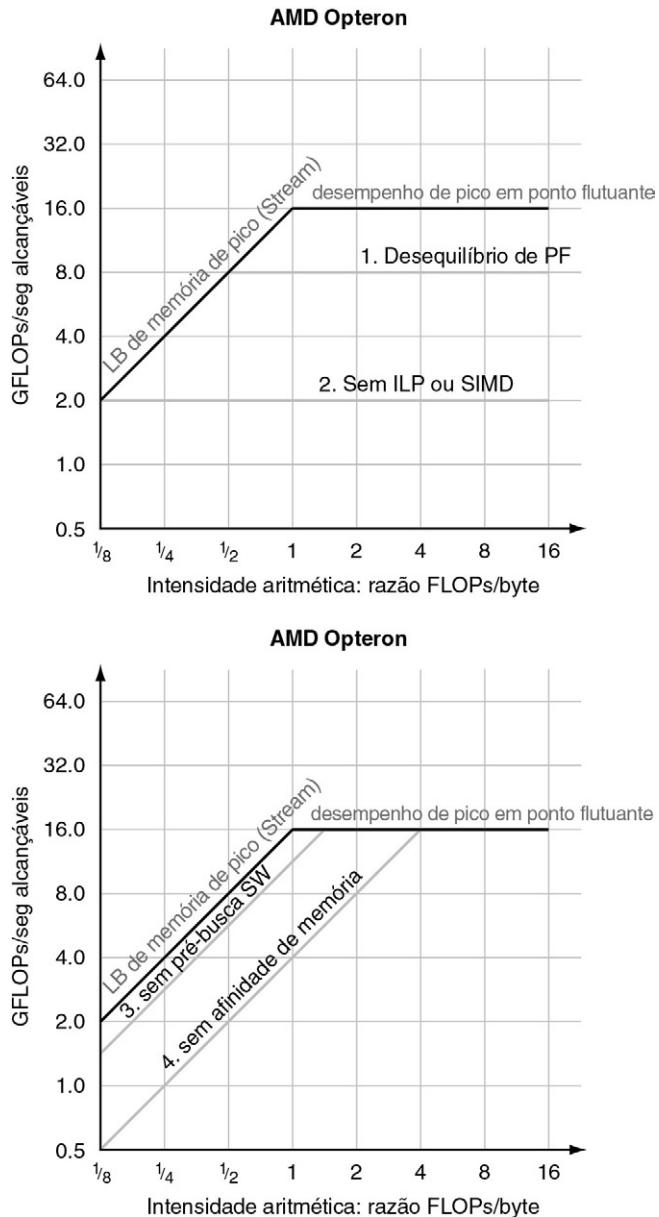


FIGURA 7.15 rooffline com tetos. O gráfico superior mostra os “tetos” computacionais de 8 GFLOPs/seg se o mix de operações de ponto flutuante estiver desequilibrado e 2 GFLOPs/seg se as otimizações para aumentar o ILP e o SIMD também estiverem faltando. O gráfico inferior mostra os tetos de largura de banda da memória de 11GB/seg sem pré-busca de software e 4,8GB/seg se as otimizações de afinidade de memória também estiverem faltando.

Até aqui, estivemos supondo que a intensidade aritmética é fixa, mas esse não é realmente o caso. Primeiro, existem kernels cuja intensidade aritmética aumenta com o tamanho do problema, como para os problemas Matriz Densa e N-body (veja Figura 7.12). Na realidade, esse pode ser o motivo para os programadores terem mais sucesso com a expansão fraca do que com a expansão forte. Segundo, as caches afetam o número de acessos que vão para a memória, de modo que as otimizações que melhoraram o desempenho da cache também melhoraram a intensidade aritmética. Um exemplo é melhorar a localidade temporal desdobrando loops e depois agrupando instruções com endereços semelhantes. Muitos computadores possuem instruções de cache especiais, que alocam dados em uma cache, mas não preenchem primeiro os dados da memória nesse endereço, pois eles logo

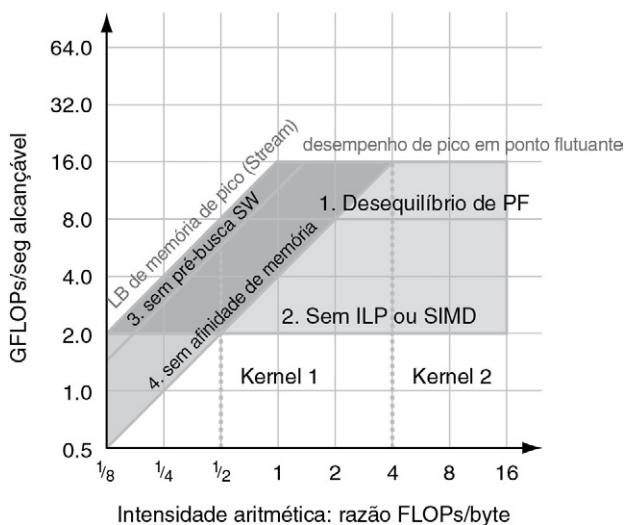


FIGURA 7.16 Modelo roofline com tetos, áreas sobrepostas sombreadas e os dois kernels da Figura 7.13. Os kernels cuja intensidade aritmática se encontra no trapezoide azul à direita deverão focalizar otimizações de computação, e os kernels cuja intensidade aritmática se encontra no triângulo cinza no canto inferior esquerdo devem focalizar otimizações de largura de banda de memória. Aqueles que se encontram no paralelogramo azul-cinza no meio precisam se preocupar com ambos. Quando o Kernel 1 cai no paralelogramo do meio, tente otimizar ILP e SIMD, afinidade de memória e pré-busca de software. O Kernel 2 cai no trapezoide à direita; portanto, tente otimizar ILP e SIMD e o equilíbrio das operações de ponto flutuante.

serão modificados. Essas duas otimizações reduzem o tráfego da memória, movendo assim o poste da intensidade aritmática para a direita por um fator de, digamos, 1,5. Esse deslocamento para a direita poderia colocar o kernel em uma região de otimização diferente.

A próxima seção usa o modelo roofline para demonstrar a diferença em quatro microprocessadores multicore recentes, para dois kernels de aplicação reais. Embora os exemplos anteriores mostrem como ajudar os programadores a melhorarem o desempenho, o modelo também pode ser usado por arquitetos para decidir onde eles otimizariam o hardware para melhorar o desempenho dos kernels que acreditam que serão importantes.

Detalhamento: Os tetos são ordenados de modo que os mais baixos são mais fáceis de otimizar. Logicamente, um programador pode otimizar em qualquer ordem, mas ter essa sequência reduz as chances de desperdiçar esforço em uma otimização que não possui benefício devido a outras restrições. Assim como o modelo 3Cs, desde que o modelo roofline ofereça percepções, um modelo pode ter esquisitices. Por exemplo, ele supõe que o programa tem平衡amento de carga entre todos os processadores.

Detalhamento: Uma alternativa ao benchmark Stream é usar a largura de banda bruta da DRAM como roofline. Enquanto as DRAMs definem um limite rígido, o desempenho real da memória normalmente está tão distante desse limite que não é tão útil como um limite superior. Ou seja, nenhum programa pode chegar perto desse limite. A desvantagem de usar o Stream é que uma programação muito cuidadosa pode exceder os resultados do Stream, de modo que a roofline da memória pode não ser um limite tão rígido quanto a roofline computacional. Ficamos com o Stream porque menos programadores serão capazes de oferecer mais largura de banda de memória do que o Stream descobre.

Detalhamento: Os dois eixos usados anteriormente eram operações de ponto flutuante por segundo e intensidade aritmética dos acessos à memória principal. O modelo roofline poderia ser usado para outros kernels e computadores cujo desempenho foi uma função de diferentes métricas de desempenho.

Por exemplo, se o conjunto de trabalho couber na cache L2 do computador, a largura de banda desenhada na roofline diagonal poderia ser largura de banda de cache L2, em vez da largura de banda da memória principal, e a intensidade aritmética no eixo X seria baseada em FLOPs por byte da cache L2 acessado. A linha de desempenho L2 diagonal subiria, e o ponto de cumeeira provavelmente se moveria para a esquerda.

Como um segundo exemplo, se o kernel fosse classificado, os registros classificados por segundo poderiam substituir as operações de ponto flutuante por instrução no eixo X e a intensidade aritmética se tornaria registros por byte de DRAM acessado.

O modelo roofline poderia ainda funcionar para um kernel com uso intenso de E/S. O eixo Y seria operações de E/S por segundo, o eixo X seria o número médio de instruções por operação de E/S, e a roofline mostraria a largura de banda de E/S de pico.

Detalhamento: Embora o modelo roofline apresentado seja para processadores multicores, ele certamente também funcionaria para um processador.

7.11

Vida real: benchmarking de quatro multicores usando o modelo roofline

Dada a incerteza sobre a melhor maneira de proceder nessa revolução paralela, não é surpresa que vejamos tantos projetos diferentes quantos chips multicore. Nesta seção, vamos examinar quatro sistemas multicore para dois kernels dos padrões de projeto da [Figura 7.11](#): matriz esparsa e grade estruturada. (As informações nesta seção são de [Williams, Oliker, et al., 2007], [Williams, Carter, et al., 2008], [Williams and Patterson, 2008].)

Quatro sistemas multicore

A [Figura 7.17](#) mostra a organização básica dos quatro sistemas, e a [Figura 7.18](#) lista as principais características dos exemplos desta seção. Estes são todos sistemas de soquete dual. A [Figura 7.19](#) mostra o modelo de desempenho roofline para cada sistema.

O Intel Xeon e5345 (apelidado de “Clovertown”) contém quatro cores por soquete, empacotando dois chips dual core em um único soquete. Esses dois chips compartilham um barramento front side que é conectado a um chip set north bridge separado (ver Capítulo 6). Esse chip set north bridge admite dois barramentos front side e, portanto, dois soquetes. Ele inclui o controlador de memória para as Fully Buffered DRAM DIMMs (FBDIMMs) de 667MHz. Esse sistema de soquete dual usa uma taxa de clock do processador de 2,33GHz e tem o mais alto desempenho de pico dos quatro exemplos: 75 GFLOPs. Porém, o modelo roofline na [Figura 7.19](#) mostra que isso só pode ser obtido com intensidades aritméticas de 8 e acima. O motivo é que os barramentos front sideiais interferem um com o outro, gerando largura de banda de memória relativamente baixa aos programas.

O AMD Opteron X4 2356 (Barcelona) contém quatro cores por chip, e cada soquete tem um único chip. Cada chip tem um controlador de memória na placa e seu próprio caminho para a DRAM DDR2 de 667MHz. Esses dois soquetes se comunicam por links Hypertransport separados, dedicados, o que possibilita a criação de um sistema multichip “sem cola”. Esse sistema de soquete dual utiliza uma taxa de clock de processador de 2,30GHz e tem um desempenho de pico de aproximadamente 74GFLOPs. A [Figura 7.19](#) mostra que o ponto de cumeeira no modelo roofline está à esquerda do Xeon e5345 (Clovertown), em uma intensidade aritmética de aproximadamente 5 FLOPs por byte.

O Sun UltraSPARC T2 5140 (apelidado de “Niagara 2”) é muito diferente das duas microarquiteturas x86. Ele usa oito cores relativamente simples por chip, com uma taxa de clock muito mais baixa. Também oferece multithreading fine-grained com oito threads

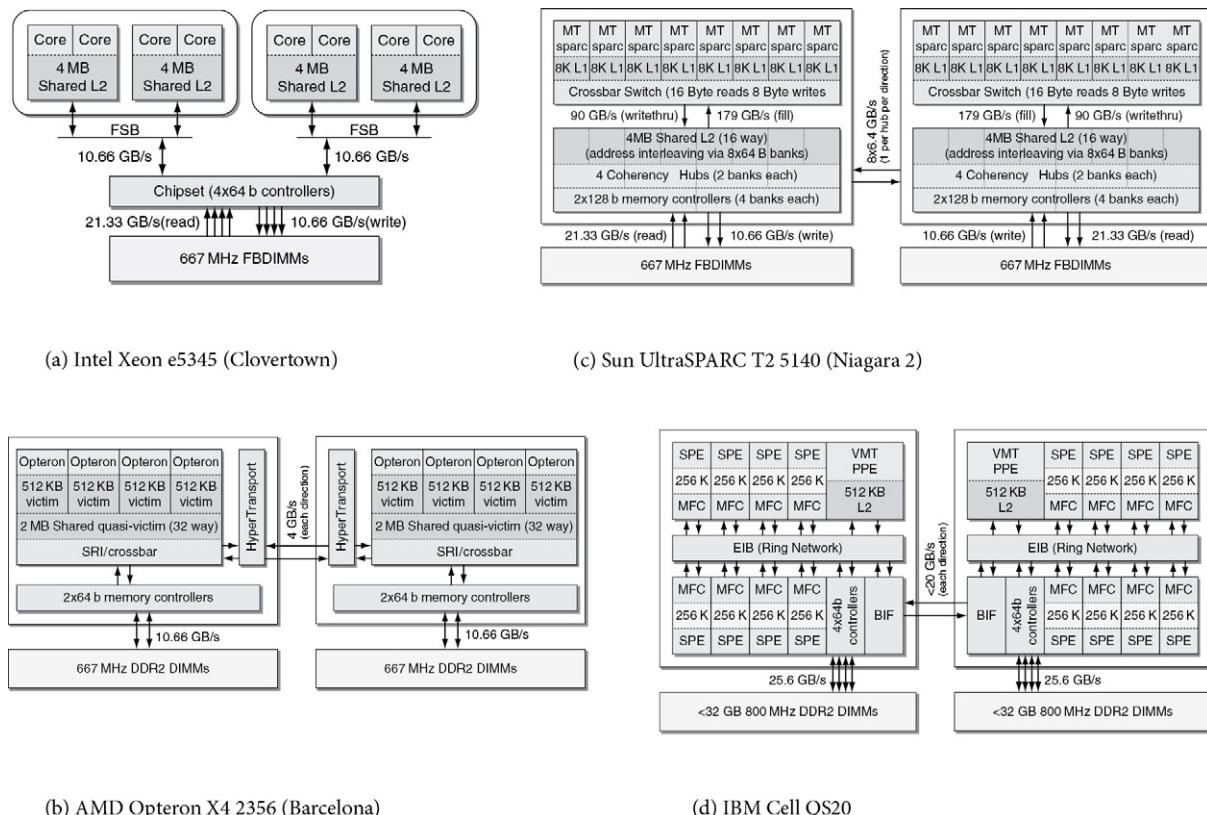


FIGURA 7.17 Quatro multiprocessadores recentes, cada um usando dois soquetes para os processadores. Começando com o canto superior esquerdo, os computadores são: (a) Intel Xeon e5345 (Clovertown), (b) AMD Opteron X4 2356 (Barcelona), (c) Sun UltraSPARC T2 5140 (Niagara 2) e (d) IBM Cell QS20. Observe que o Intel Xeon e5345 (Clovertown) tem um chip north bridge separado, não encontrado nos outros microprocessadores.

Tipo de Microprocessador	ISA	Número de Threads	Número de Cores	Número de Soquetes	Clock GHz	GFLOP/s de Pico	DRAM: GB/s de pico, Taxa de clock, Tipo
Intel Xeon e5345 (Clovertown)	x86/64	8	8	2	2,33	75	FSB: 2 x 10,6 667 MHz FBDIMM
AMD Opteron X4 2356 (Barcelona)	x86/64	8	8	2	2,30	74	2 x 10,6 667 MHz DDR2
Sun UltraSPARC T2 5140 (Niagara 2)	Sparc	128	16	2	1,17	22	2 x 21,3 (read) 2 x 10,6 (write) 667 MHz FBDIMM
IBM Cell QS20	Cell	16	16	2	3,20	29	2 x 25,6 XDR

FIGURA 7.18 Características dos quatro multicores recentes. Embora o Xeon e5345 e o Opteron X4 tenham DRAMs com a mesma velocidade, o benchmark Stream mostra uma largura de banda de memória prática mais alta devido a ineficiências do barramento front side no Xeon e5345.

por core. Um único chip tem quatro controladores de memória que poderiam impulsionar quatro conjuntos de FBDIMMs de 667MHz. Para juntar dois chips UltraSPARC T2, dois dos quatro canais de memória são conectados, deixando dois canais de memória por chip. Esse sistema de soquete dual tem um desempenho de pico de cerca de 22 GFLOPs, e o ponto de cumeeira é uma intensidade aritmética incrivelmente baixa, com apenas 1/3 FLOPs por byte.

O IBM Cell QS20 novamente é diferente das duas microarquiteturas x86 e do UltraSPARC T2. Esse é um projeto heterogêneo, com um core PowerPC relativamente simples e com oito SPEs (Synergistic Processing Elements) que têm seu próprio conjunto de instruções exclusivo em estilo SIMD. Cada SPE também tem sua própria memória local, em

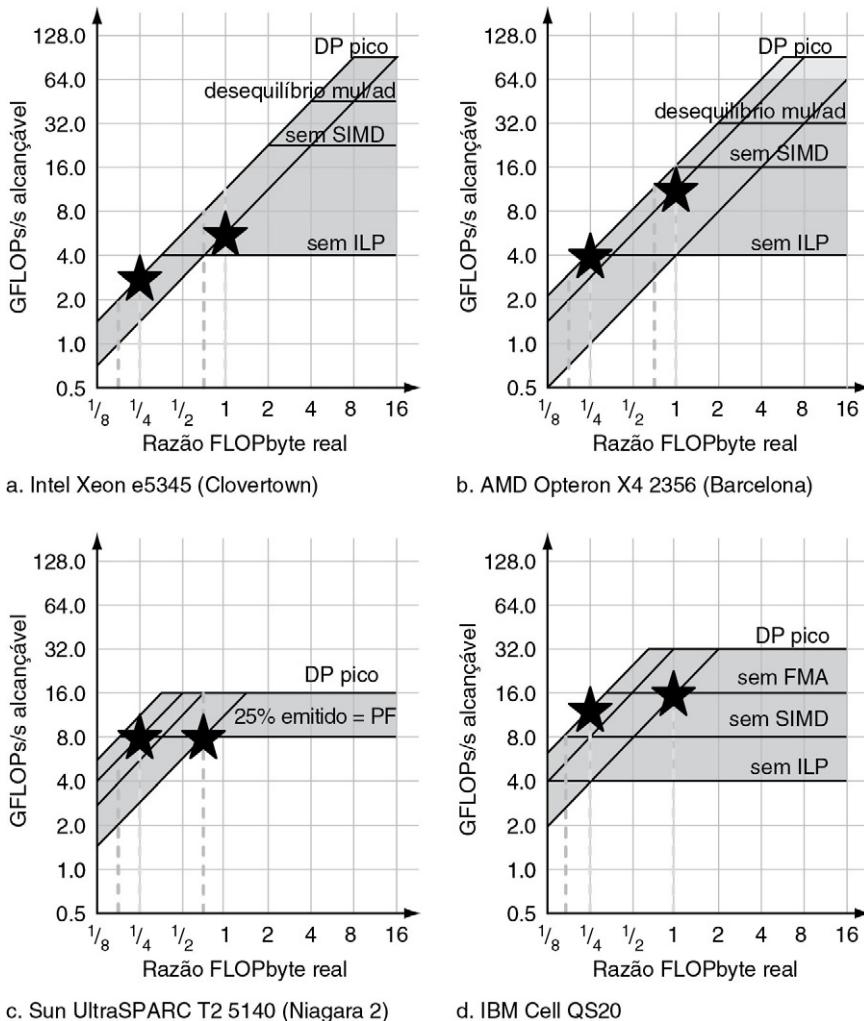


FIGURA 7.19 Modelo roofline para os multiprocessadores multicore na Figura 7.15. Os tetos são os mesmos que na Figura 7.13. Começando com o canto superior esquerdo, os computadores são: (a) Intel Xeon e5345 (Clovertown), (b) AMD Opteron X4 2356 (Barcelona), (c) Sun UltraSPARC T2 5140 (Niagara 2) e (d) IBM Cell QS20. Observe que os pontos de cumeeira para os quatro microprocessadores cruzam o eixo X nas intensidades aritméticas de 6, 4, 1/3 e 3/4, respectivamente. As linhas verticais tracejadas são para os dois kernels desta seção e as estrelas marcam o desempenho obtido para esses kernels após todas as otimizações. SpMV é o par de linhas verticais tracejadas à esquerda. Ele tem duas linhas porque sua intensidade aritmética melhorou de 0,166 para 0,255 com base nas otimizações de bloqueio de registrador. LBHMD são as linhas verticais tracejadas à direita. Ele tem um par de linhas em (a) e (b) porque uma otimização de cache pula o preenchimento do bloco de cache em uma falha quando o processador escreveria novos dados no bloco inteiro. Essa otimização aumenta a intensidade aritmética de 0,70 para 1,07. Essa é uma única linha a 0,70 em (c) porque o UltraSPARC T2 não oferece a otimização da cache. Essa é uma única linha a 1,07 em (d) porque o Cell possui store local carregado pelo DMA, de modo que o programa não busca dados desnecessários, como fazem as caches.

vez de uma cache. Um SPE deve transferir dados da memória principal para a memória local, a fim de operar sobre ela e depois de volta à memória principal, quando é concluído. Ele usa DMA, que tem alguma semelhança com a pré-busca de software. Os dois soquetes são conectados por meio de links dedicados a comunicações multichip. A taxa de clock desse sistema é a mais alta dos quatro multicores em 3,2GHz, e usa chips de DRAM XDR, que normalmente são encontrados em consoles de jogos. Eles possuem muita largura de banda, mas pouca capacidade. Dado que a aplicação principal do Cell era de gráficos, ele tem desempenho em precisão simples muito mais alto do que o desempenho em precisão dupla. O desempenho de pico em precisão dupla dos SPEs no sistema de soquete dual é 29 GFLOPs, e o ponto de cumeeira da intensidade aritmética é 0,75 FLOPs por byte.

Embora as duas arquiteturas x86 tivessem muito menos cores por chip que as ofertas da IBM e Sun no início de 2008, é exatamente aí que elas estão hoje. À medida que se espera que o número de cores dobre a cada geração da tecnologia, será interessante ver se as arquiteturas x86 fecharão a “lacuna de core” ou se IBM e Sun poderão sustentar um número maior de cores, dado que seu foco principal está nos servidores e não no desktop.

Observe que essas máquinas utilizam técnicas muito diferentes para o sistema de memória. O Xeon e5345 usa uma cache L1 privada convencional e então pares de processadores compartilham uma cache L2. Estes são conectados por meio de um controlador de memória fora do chip a uma memória comum por dois barramentos. Ao contrário, o Opteron X4 tem um controlador de memória separado e memória por chip, e cada core tem caches L1 e L2 privados. UltraSPARC T2 tem o controlador de memória no chip e quatro canais de DRAM separados por chip, e todos os cores compartilham a cache L2, que tem quatro bancos para melhorar a largura de banda. Seu multithreading fine-grained no topo de seu projeto multicore permite que ele mantenha muitos acessos à memória no ato. O mais radical é o Cell. Ele tem memórias privadas locais por SPE e usa DMA para transferir dados entre a DRAM conectada a cada chip e memória local. Ele sustenta muitos acessos à memória no ato tendo muitos cores e depois muitas transferências de DMA por core.

Vejamos como esses quatro multicores em contraste funcionam nos dois kernels.

Matriz esparsa

O primeiro kernel de exemplo do padrão de projeto computacional de matriz esparsa é o Sparse Matrix-Vector multiply (SpMV). SpMV é comum na computação científica, modelagem econômica e recuperação de informações. Infelizmente, as implementações convencionais normalmente executam em menos de 10% do desempenho de pico dos processadores. Um motivo é o acesso irregular à memória, que você poderia esperar de um kernel trabalhando com matrizes esparsas. O cálculo é

$$y = A \times x$$

onde A é uma matriz esparsa e x e y são vetores densos. Quatorze matrizes esparsas tomadas de uma série de aplicações reais foram usadas para avaliar o desempenho do SpMV, mas somente o desempenho mediano é relatado aqui. A intensidade aritmética varia de 0,166 antes de uma otimização de bloqueio de registrador para 0,250 FLOPs por byte depois disso.

O código primeiro foi paralelizado para utilizar todos os cores. Dado que a intensidade aritmética baixa do SpMV esteve abaixo do ponto de cumeeira de todos os multicores na Figura 7.19, a maioria das otimizações envolveu o sistema de memória:

- *Pré-busca.* Para obter o máximo dos sistemas de memória, as pré-buscas de software e hardware foram utilizadas.
- *Afinidade de memória.* Essa otimização reduz os acessos à memória DRAM conectada ao outro soquete nos três sistemas que têm memória DRAM local.
- *Compactação de estruturas de dados.* Como a largura de banda de memória provavelmente limita o desempenho, essa otimização usa estruturas de dados menores para aumentar o desempenho — por exemplo, usando um índice de 16 bits em vez de um índice de 32 bits, e usando representações dos não zeros com uso mais eficiente do espaço nas linhas de uma matriz esparsa.

A Figura 7.20 mostra o desempenho no SpMV para os quatro sistemas em comparação com o número de cores. (Os mesmos resultados são encontrados na Figura 7.19, mas é difícil comparar o desempenho quando se usa uma escala logarítmica.) Observe que, apesar de ter o desempenho de pico mais alto na Figura 7.18 e o desempenho de core isolado mais alto, o Intel Xeon e5345 tem o desempenho oferecido mais baixo dos quatro multicores. O Opteron X4 dobra seu desempenho. O gargalo do Xeon e5345 são os barramentos front side duais. Apesar da taxa de clock mais baixa, o número maior de cores simples do Sun

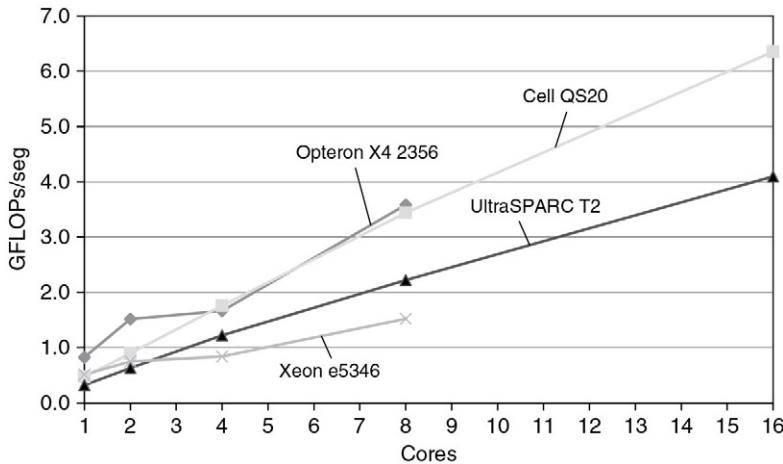


FIGURA 7.20 Desempenho do SpMV nos quatro multicores.

UltraSPARC T2 supera os dois processadores x86. O IBM Cell tem o desempenho mais alto dos quatro. Observe que todos menos o Xeon e5346 se expandem bem com o número de cores, embora o Opteron X4 se expanda mais lentamente com quatro ou mais cores.

Grade estruturada

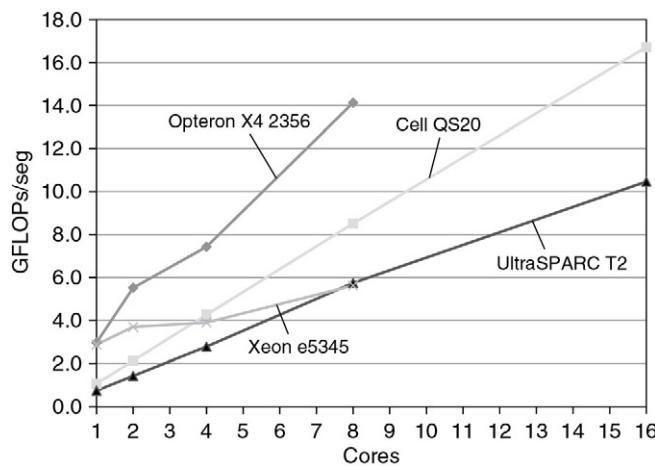
O segundo kernel é um exemplo do padrão de projeto de grade estruturada. Lattice-Boltzmann Magneto-Hydrodynamics (LBMHD) é comum para a dinâmica de fluido computacional; ele é um código de grade estruturada com uma série de etapas de tempo.

Cada ponto envolve leitura e escrita de aproximadamente 75 números de ponto flutuante de precisão dupla e cerca de 1300 operações de ponto flutuante. Assim como SpMV, LBMHD tende a conseguir uma pequena fração do desempenho de pico nos processadores, devido à complexidade das estruturas de dados e da irregularidade dos padrões de acesso à memória. A razão entre FLOPs e bytes é muito maior, 0,70, em comparação com menos de 0,25 no SpMV. Não preenchendo o bloco de cache da memória em uma perda de escrita quando o programa for sobrescrever o bloco inteiro, a intensidade sobe para 1,07. Todos os multicores menos o UltraSPARC T2 (Niagara 2) oferecem essa otimização de cache.

A Figura 7.19 mostra que a intensidade aritmética do LBMHD é tão alta que as otimizações de largura de banda computacional e da memória fazem sentido em todos os multicores, menos o UltraSPARC T2, cujo ponto de cumeeira da roofline está abaixo do ponto do LBMHD. O UltraSPARC T2 pode alcançar a roofline usando apenas as otimizações computacionais.

Além de colocar o código em paralelo, de modo que possa usar todos os cores, as otimizações a seguir foram usadas para LBMHD:

- *Afinidade de memória:* Essa otimização novamente é útil pelos mesmos motivos mencionados anteriormente.
- *Minimização de perda de TLB:* Para reduzir as perdas de TLB significativamente no LBMHD, use uma estrutura de arrays e combine alguns loops juntos em vez da técnica convencional de usar um array de estruturas.
- *Desdobramento e reordenação de loop:* Para expor paralelismo suficiente e melhorar a utilização de cache, os loops foram desdobrados e depois reordenados para agrupar as instruções com endereços semelhantes.
- *SIMD:* Os compiladores dos dois sistemas x86 não poderiam gerar bom código SSE, de modo que estes tiveram de ser escritos à mão em linguagem assembly.


FIGURA 7.21 Desempenho do LBMHD nos quatro multicores.

Tipo Microprocessador	Kernel	GFLOPs/s base	GFLOPs/s otimizado	% simples do otimizado
Intel Xeon e5345 (Clovertown)	SpMV	1,0	1,5	64%
	LBMHD	4,6	5,6	82%
AMD Opteron X4 2356 (Barcelona)	SpMV	1,4	3,6	38%
	LBMHD	7,1	14,1	50%
Sun UltraSPARC T2 (Niagara 2)	SpMV	3,5	4,1	86%
	LBMHD	9,7	10,5	93%
IBM Cell QS20	SpMV	-	6,4	0%
	LBMHD	-	16,7	0%

FIGURA 7.22 Desempenho de base versus totalmente otimizado dos quatro cores nos dois kernels. Observe a alta fração de desempenho totalmente otimizado oferecido pelo Sun UltraSPARC T2 (Niagara 2). Não existe coluna de desempenho de base para o IBM Cell, pois não existe como portar o código para os SPEs sem caches. Embora você possa executar o código no core do Power, ele tem um desempenho com uma ordem de grandeza a menos que o SPES, e por isso o ignoramos nesta figura.

A Figura 7.21 mostra o desempenho dos quatro sistemas em comparação com o número de cores para LBMHD. Assim como o SpMV, o Intel Xeon e5345 tem a pior escalabilidade. Desta vez, os cores mais poderosos do Opteron X4 são superiores aos cores simples do UltraSPARC T2, apesar de ter metade do número de cores. Mais uma vez, o IBM Cell é o sistema mais rápido. Tudo menos o Xeon e5345 se expande bem com o número de cores, embora T2 e Cell se expandam mais tranquilamente do que o Opteron X4.

Produtividade

Além do desempenho, outra questão importante para a revolução da computação paralela é a produtividade, ou a dificuldade da programação de alcançar o desempenho. Para ilustrar as diferenças, a Figura 7.22 compara o desempenho simples com o desempenho totalmente otimizado para os quatro cores nos dois kernels.

O mais fácil foi o UltraSPARC T2, devido à sua grande largura de banda de memória e seus cores fáceis de entender. O conselho para esses dois kernels no UltraSPARC T2 é simplesmente tentar obter código de bom desempenho do compilador e depois usar o máximo de threads possível. O único cuidado para outros kernels é que o UltraSPARC T2 pode cair na armadilha sobre garantir que a associatividade de conjunto combina com o número de threads de hardware (veja Seção 5.11, no Capítulo 5). Cada chip admite 64 threads de hardware, enquanto o cache L2 é associativo em conjunto com quatro vias. Essa divergência pode exigir a reestruturação de loops para reduzir as falhas por conflito.

O Xeon e5346 era considerado complexo porque não era fácil entender o comportamento de memória dos barramentos front side duais, era difícil entender como funcionava a pré-busca do hardware, assim como obter um bom código SIMD do compilador. O código C para ele e para o Opteron X4 são repletos de instruções intrínsecas envolvendo instruções SIMD para obter um bom desempenho.

O Opteron X4 beneficiou-se da maioria dos tipos de otimizações, de modo que precisou de mais esforço que o Xeon e5345, embora o comportamento da memória do Opteron X4 fosse mais fácil de entender que o do Xeon e5345.

O Cell forneceu dois tipos de desafios. Primeiro, as instruções SIMD do SPE eram difíceis de compilar, de modo que às vezes você precisava ajudar o compilador, inserindo instruções intrínsecas com instruções em linguagem assembly no código C. Segundo, o sistema de memória era mais interessante. Como cada SPE tem memória local em um espaço de endereço separado, você não poderia simplesmente transportar o código e começar a executar no SPE. Portanto, não há uma coluna de código base para o IBM Cell na Figura 7.22, e você precisava mudar o programa para emitir comandos de DMA e transferir dados entre o armazenamento local e a memória. A boa notícia é que o DMA desempenhou o papel de pré-busca de software nas caches, e o DMA é muito mais fácil de usar e conseguir um bom desempenho da memória. O Cell foi capaz de oferecer quase 90% da “roofline” de largura de banda de memória para esses kernels, em comparação com 50% ou menos para os outros multicores.

Por mais de uma década os analistas anunciam que a organização de um único computador alcançou seus limites e que avanços verdadeiramente significantes só podem ser feitos pela interconexão de uma multiplicidade de computadores de tal modo que permita solução cooperativa... Demonstrou-se a continuada validade do método de processador único...

Gene Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities”, Spring Joint Computer Conference, 1967

7.12

Falácia e armadilhas

Os muitos ataques ao processamento paralelo revelaram inúmeras falácias e armadilhas. Veremos três delas aqui.

Falácia: a Lei de Amdahl não se aplica aos computadores paralelos.

Em 1987, o diretor de uma organização de pesquisa afirmou que a Lei de Amdahl tinha sido quebrada por uma máquina de multiprocessador. Para tentar entender a base dos relatos da mídia, vejamos a citação que nos trouxe a Lei de Amdahl [1967, p. 483]:

Uma conclusão bastante óbvia que pode ser tirada nesse momento é que o esforço despendido em conseguir altas velocidades de processamento paralelo é desperdiçado se não for acompanhado de conquistas de mesmas proporções nas velocidades de processamento sequencial.

Essa afirmação ainda deve ser verdadeira; a parte ignorada do programa deve limitar o desempenho. Uma interpretação da lei leva ao seguinte princípio: partes de cada programa precisam ser sequenciais e, portanto, precisa haver um limite superior lucrativo para o número de processadores – digamos, 100. Mostrando speed-up linear com 1.000 processadores, esse princípio se torna falso e, então, a Lei de Amdahl foi quebrada.

O método dos pesquisadores foi mudar a entrada para o benchmark: em vez de ir 1.000 vezes mais rápido, eles calcularam 1.000 vezes mais trabalho em tempo comparável. Para o algoritmo deles, a parte sequencial do programa era constante, independente do tamanho da entrada, e o restante era totalmente paralelo – daí, speed-up linear com 1.000 processadores.

Não vemos razão para que a Lei de Amdahl não se aplique aos processadores paralelos. O que essa pesquisa salienta é um dos principais usos de computadores rápidos é executar problemas grandes, mas ter ciência de como os algoritmos escalam com o crescimento do tamanho do problema.

Falácia: o desempenho de pico segue o desempenho observado.

Por exemplo, a Seção 7.11 mostra que o Intel Xeon e5345, o microprocessador com o desempenho de pico mais alto, foi o mais lento dos quatro microprocessadores multicore para dois kernels.

A indústria de supercomputadores usou essa métrica no marketing, e a falácia é enfatizada com as máquinas paralelas. Não apenas os marketeiros estão usando o desempenho de pico quase inatingível de um único processador, mas eles também estão multiplicando pelo número total de processadores, considerando speed-up perfeito! A lei de Amdahl sugere como é difícil alcançar qualquer um desses picos; multiplicar os dois multiplica os pecados. O modelo roofline ajuda a entender melhor o desempenho de pico.

Armadilha: não desenvolver o software para tirar proveito de (ou otimizar) uma arquitetura de multiprocessador.

Há um longo histórico de software ficando para trás nos processadores paralelos, possivelmente porque os problemas do software são muito mais difíceis. Temos um exemplo para mostrar a sutileza dessas questões, mas existem muitos exemplos que poderíamos escolher!

Um problema encontrado com frequência ocorre quando o software projetado para um processador único é adaptado a um ambiente multiprocessador. Por exemplo, o sistema operacional SGI protegia originalmente a tabela de página com um único lock, supondo que a alocação de página é pouco frequente. Em um processador único, isso não representa um problema de desempenho, mas em um multiprocessador, pode se tornar um gargalo de desempenho importante para alguns programas. Considere um programa que usa um grande número de páginas que são inicializadas quando começa a ser executado, o que o UNIX faz para as páginas alocadas estaticamente. Suponha que o programa seja colocado em paralelo, de modo que múltiplos processos aloquem as páginas. Como a alocação de página requer o uso da tabela de página, que é bloqueada sempre que está em uso, até mesmo um kernel do SO que permita múltiplas threads no SO será colocado em série se todos os processos tentarem alocar suas páginas ao mesmo tempo (que é exatamente o que poderíamos esperar no momento da inicialização!).

Essa serialização da tabela de página elimina o paralelismo na inicialização e tem um impacto significativo sobre o desempenho paralelo geral. Esse gargalo de desempenho persiste até mesmo para o paralelismo em nível de tarefa. Por exemplo, suponha que dividimos o programa de processamento paralelo em tarefas separadas e as executemos, uma tarefa por processador, de modo que não haja compartilhamento entre elas. (É exatamente isso o que um usuário fez, pois acreditava que o problema de desempenho era devido ao compartilhamento não intencional ou interferência em sua aplicação.) Infelizmente, o lock ainda coloca todas as tarefas em série — de modo que, até mesmo o desempenho da tarefa independente é fraco.

Essa armadilha indica os tipos de bugs de desempenho sutis, porém significativos, que podem surgir quando o software é executado em multiprocessadores. Assim como muitos outros componentes de software essenciais, os algoritmos do SO e as estruturas de dados precisam ser repensadas em um contexto de multiprocessador. Colocar locks em partes menores da tabela de página efetivamente elimina o problema.

7.13

Comentários finais

O sonho de construir computadores apenas agregando processadores existe desde os primeiros dias da computação. No entanto, o progresso na construção e no uso de processadores paralelos eficientes tem sido lento. Essa velocidade de progresso foi limitada pelos difíceis problemas de software bem como por um longo processo de evolução da arquitetura dos multiprocessadores para melhorar a usabilidade e a eficiência. Discutimos muitos dos problemas de software neste capítulo, incluindo a dificuldade de escrever programas que obtêm bom speed-up devido à Lei de Amdahl. A grande variedade de métodos arquitetônicos diferentes e o sucesso limitado e a vida curta de muitas arquiteturas até agora se juntam às dificuldades de software. Abordaremos a história do desenvolvimento desses multiprocessadores na  Seção 7.14, no site.

Estamos dedicando todo o nosso desenvolvimento de produto futuro aos projetos multicore. Acreditamos que esse seja um ponto de inflexão importante para a indústria. ... Essa não é uma corrida, é uma mudança de mares na computação.

Paul Otellini, Presidente da Intel, Intel Developers Forum, 2004.

Como dissemos no Capítulo 1, apesar desse longo e sinuoso passado, a indústria da tecnologia de informação agora tem seu futuro ligado à computação paralela. Embora seja fácil apontar fatos para que esse esforço falhe como muitos no passado, existem motivos para termos esperança:

- Claramente, o **software como um serviço** está ganhando mais importância, e os clusters provaram ser um modo muito bem-sucedido de oferecer tais serviços. Oferecendo redundância em um nível mais alto, incluindo centros de dados geograficamente distribuídos, esses serviços têm oferecido disponibilidade $24 \times 7 \times 365$ para os clientes no mundo inteiro. É difícil não imaginar que o número de servidores por centro de dados e o número de centros de dados continuarão a crescer. Certamente, esses centros de dados abraçarão projetos multicore, pois eles já podem usar milhares de processadores em suas aplicações.
- O uso de processamento paralelo em domínios como a computação científica e de engenharia é comum. Esse domínio de aplicação possui uma necessidade quase ilimitada de mais computação. Ele também possui muitas aplicações com uma grande quantidade de paralelismo natural. Mais uma vez, os clusters dominam essa área de aplicação. Por exemplo, usando o relatório do Linpack 2007, os clusters representam mais de 80% dos 500 computadores mais rápidos. Entretanto, isso não tem sido fácil: programar processadores paralelos até para essas aplicações continua sendo um desafio. Ainda assim, esse grupo certamente também acolherá os chips multicore, pois novamente eles têm experiência com centenas a milhares de processadores.
- Todos os fabricantes de microprocessador de desktop e servidor estão construindo multiprocessadores para alcançar o desempenho mais alto, de modo que, diferente do passado, não existe um caminho fácil para o desempenho mais alto para aplicações sequenciais. Logo, os programadores que precisam de desempenho mais alto precisam colocar seus códigos em paralelo ou escrever novos programas de processamento paralelo.
- Processadores múltiplos no mesmo chip permitem uma velocidade de comunicação muito diferente dos projetos de múltiplos chips, oferecendo latência muito mais baixa e largura de banda muito mais alta. Essas melhorias podem facilitar a oferta de bom desempenho.
- No passado, os microprocessadores e os multiprocessadores estavam sujeitos a diferentes definições de sucesso. Ao escalar o desempenho do processador único, os arquitetos de microprocessador ficavam felizes se o desempenho com uma única thread subisse pela raiz quadrada da área de silício aumentada. Assim, eles ficavam felizes com um desempenho sublinear em termos de recursos. O sucesso do multiprocessador era definido como um speed-up linear como função do número de processadores, supondo que o custo da compra ou o custo da administração de n processadores era n vezes o custo de um processador. Agora que o paralelismo está acontecendo no chip via multicore, podemos usar o microprocessador tradicional com sucesso na melhoria do desempenho sublinear.
- O sucesso da compilação em tempo de execução just-in-time torna viável pensar no software adaptando-se para tirar proveito do número cada vez maior de cores por chip, o que oferece uma flexibilidade que não está disponível quando limitado a compiladores estáticos.
- Diferente do passado, o movimento do código-fonte aberto tornou-se uma parte fundamental da indústria de software. Esse movimento é uma meritocracia, em que melhores soluções de engenharia podem ganhar a fatia de desenvolvedores em relação a questões legadas. Ele também alcança a inovação, convidando a mudança no software antigo e recebendo novas linguagens e produtos de software. Essa cultura aberta poderia ser extremamente útil nessa época de mudança rápida.

Essa revolução na interface de hardware/software talvez seja o maior desafio que esse campo encarou nos últimos 50 anos. Ela oferecerá muitas novas oportunidades de pesquisa e negócios dentro e fora do campo de TI, e as empresas que dominam a era do multicore podem não ser as mesmas que dominaram a era do processador único. Talvez você será um dos inovadores que aproveitará as oportunidades que certamente aparecerão nos tempos de incerteza mais adiante.

7.14

Perspectiva histórica e leitura adicional

Esta seção no site oferece um histórico rico e geralmente desastroso dos multiprocessadores nos últimos 50 anos.

7.15

Exercícios¹

Exercício 7.1

Primeiro, escreva uma lista das atividades diárias que você realiza normalmente em um fim de semana. Por exemplo, você poderia levantar da cama, tomar um banho, se vestir, tomar o café, secar seu cabelo, escovar os dentes etc. Lembre-se de distribuir sua lista de modo que tenha um mínimo de dez atividades.

7.1.1 <7.2> Agora considere qual dessas atividades já está explorando alguma forma de paralelismo (por exemplo, escovar vários dentes ao mesmo tempo em vez de um de cada vez, carregar um livro de cada vez para a escola em vez de colocar todos eles na sua mochila, e depois carregá-los “em paralelo”). Para cada uma de suas atividades, discuta se elas já estão sendo executadas em paralelo, mas se não, por que não estão.

7.1.2 <7.2> Em seguida, considere quais das atividades poderiam ser executadas simultaneamente (por exemplo, tomar café e escutar as notícias). Para cada uma das suas atividades, descreva qual outra atividade poderia ser emparelhada com essa atividade.

7.1.3 <7.2> Para o Exercício 7.1.2, o que poderíamos mudar sobre os sistemas atuais (por exemplo, banhos, roupas, TVs, carros) de modo que pudéssemos realizar mais tarefas em paralelo?

7.1.4 <7.2> Estime quanto tempo a menos seria necessário para executar essas atividades se você tentasse executar o máximo de tarefas em paralelo possível.

Exercício 7.2

Muitas aplicações de computador envolvem a pesquisa por um conjunto de dados e a classificação dos dados. Diversos algoritmos eficientes de busca e classificação foram criados para reduzir o tempo de execução dessas tarefas tediosas. Neste problema, vamos considerar como é melhor colocar essas tarefas em paralelo.

¹ Contribuição de David Kaeli, da Northeastern University

7.2.1 [10] <7.2> Considere o seguinte algoritmo de busca binária (um algoritmo clássico do tipo dividir e conquistar) que procura um valor X em um array de N elementos A e retorna o índice da entrada correspondente:

```
BinarySearch(A[0..N-1], X) {
    low = 0
    high = N - 1
    while (low <= high) {
        mid = (low + high) / 2
        if (A[mid] > X)
            high = mid - 1
        else if (A[mid] < X)
            low = mid + 1
        else
            return mid // encontrado
    }
    return -1 // não encontrado
}
```

Suponha que você tenha Y cores em um processador multicore para executar BinarySearch. Supondo que Y seja muito menor que N, expresse o fator de speed-up que você poderia esperar obter para os valores e Y e N. Desenhe isso em um gráfico.

7.2.2 [5] <7.2> Em seguida, suponha que Y seja igual a N. Como isso afetaria suas conclusões na sua resposta anterior? Se você estivesse encarregado de obter o melhor fator de speed-up possível (ou seja, expansão forte), explique como poderia mudar esse código para obter isso.

Exercício 7.3

Considere o seguinte trecho de código em C:

```
for (j=2;j<1000;j++)
    D[j] = D[j-1]+D[j-2];
```

O código MIPS correspondente a esse fragmento é:

```
DADDIU r2,r2,999
loop: L.D f1, -16(f1)
      L.D f2, -8(f1)
      ADD.D f3, f1, f2
      S.D f3, 0(r1)
      DADDIU r1, r1, 8
      BNE r1, r2, loop
```

As instruções têm as seguintes latências associadas (em ciclos):

ADD.D	L.D	S.D	DADDIU
4	6	1	2

7.3.1 [10] <7.2> Quantos ciclos são necessários para que todas as instruções em uma única iteração do loop anterior sejam executadas?

7.3.2 [10] <7.2> Quando uma instrução em uma iteração posterior de um loop depende do valor de dados produzido em uma iteração anterior do mesmo loop, dizemos que existe uma *dependência carregada pelo loop* entre as iterações do loop. Identifique as dependências carregadas pelo loop no código anterior. Identifique a variável de programa dependente e os registradores em nível de assembly. Você pode ignorar a variável de indução de loop j.

7.3.3 [10] <7.2> O desdobramento de loop foi descrito no Capítulo 4. Aplique o desdobramento de loop a esse loop e depois considere a execução desse código em um sistema de passagem de mensagem com memória distribuída com 2 nós. Suponha que usaremos a passagem de mensagem conforme descrito na Seção 7.4, na qual apresentamos uma nova operação send(x,y) que envia ao nó x o valor y, e uma operação receive() que espera pelo valor sendo enviado a ele. Suponha que as operações send gastem um ciclo para emitir (ou seja, outras instruções no mesmo nó podem prosseguir para o próximo ciclo), mas gastem 10 ciclos para serem recebidas no nó receptor. Operações receive provocam stall da execução no nó em que são executadas até que recebam uma mensagem. Produza um schedule para os dois nós; considere um fator de desdobramento de 4 para o corpo do loop (ou seja, o corpo do loop aparecerá quatro vezes). Calcule o número de ciclos necessários para que o loop seja executado no sistema de passagem de mensagens.

7.3.4 [10] <7.2> A latência da rede de interconexão desempenha um papel importante na eficiência dos sistemas de passagem de mensagens. Que velocidade a interconexão precisa ter a fim de obter qualquer speed-up com o uso do sistema distribuído descrito no Exercício 7.3.3?

Exercício 7.4

Considere o seguinte algoritmo mergesort recursivo (outro algoritmo clássico para dividir e conquistar). Mergesort foi descrito inicialmente por John von Neumann em 1945. A ideia básica é dividir uma lista não classificada x de m elementos em duas sublistas de aproximadamente metade do tamanho da lista original. Repita essa operação em cada sublista e continue até que tenhamos listas de tamanho 1. Depois, começando com sublistas de tamanho 1, faça o “merge” das duas sublistas em uma única lista classificada.

```
Mergesort(m)
    var list left, right, result
    if length(m) = 1
        return m
    else
        var middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = Mergesort(left)
        right = Mergesort(right)
        result = Merge(left, right)
    return result
```

A etapa do merge é executada pelo seguinte código:

```

Merge(left,right)
    var list result
    while length(left) > 0 and length(right) > 0
        if first(left) = first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
        if length(left) > 0
            append rest(left) to result
        if length(right) > 0
            append rest(right) to result
    return result

```

7.4.1 [10] <7.2> Suponha que você tenha Y cores em um processador multicore para executar o MergeSort. Supondo que Y seja muito menor que length(m), expresse o fator de speed-up que você poderia esperar obter para os valores de Y e length(m). Desenhe isso em um gráfico.

7.4.2 [10] <7.2> Em seguida, considere que Y é igual a length(m). Como isso afetaria suas conclusões na sua resposta anterior? Se você estivesse encarregado de obter o melhor fator de speed-up possível (ou seja, expansão forte), explique como poderia mudar esse código para obtê-lo.

Exercício 7.5

Você está tentando preparar três tortas de mirtilo. Os ingredientes são os seguintes:

- 1 xícara de manteiga
- 1 xícara de açúcar
- 4 ovos grandes
- 1 colher de chá de extrato de baunilha
- 1/2 colher de chá de sal
- 1/4 colher de chá de noz moscada
- 1 1/2 xícaras de farinha de trigo
- 1 xícara de mirtilos

A receita para uma única torta é a seguinte:

Passo 1: pré-aqueça o forno a 160 °C. Unte e polvilhe farinha na forma.

Passo 2: em uma bacia grande, bata com a batedeira a manteiga e o açúcar em velocidade média até que a massa fique leve e macia. Acrescente ovos, baunilha, sal e noz moscada. Bata até que tudo fique totalmente misturado. Reduza a velocidade da batedeira e acrescente farinha de trigo, 1/2 xícara por vez, batendo até ficar bem misturado.

Passo 3: inclua os mirtilos aos poucos. Espalhe uniformemente na forma da torta. Leve ao forno 60 minutos.

7.5.1 [5] <7.2> Sua tarefa é cozinhar três tortas da forma mais eficiente possível. Supondo que você só tenha um forno com tamanho suficiente para conter uma torta, uma bacia

grande, uma forma de torta e uma batedeira, prepare um plano para fazer as três tortas o mais rapidamente possível. Identifique os gargalos para completar essa tarefa.

7.5.2 [5] <7.2> Suponha agora que você tenha três bacias, três formas de torta e três batedeiras. O quanto o processo fica mais rápido, agora que você tem esses recursos adicionais?

7.5.3 [5] <7.2> Agora suponha que você tem dois amigos que o ajudarão a cozinhar, e que você tem um forno grande, que possa acomodar todas as três tortas. Como isso mudará o plano que você preparou no Exercício 7.5.1?

7.5.4 [5] <7.2> Compare a tarefa de preparação da torta com o cálculo de três iterações de um loop em um computador paralelo. Identifique o paralelismo em nível de dados e o paralelismo em nível de tarefa no loop de preparação da torta.

Exercício 7.6

A multiplicação de matriz desempenha um papel importante em diversas aplicações. Duas matrizes só podem ser multiplicadas se o número de colunas da primeira matriz for igual ao número de linhas na segunda.

Vamos supor que tenhamos uma matriz $m \times n A$ e queiramos multiplicá-la por uma matriz $n \times p B$. Podemos expressar seu produto como uma matriz $m \times p$ indicada por AB (ou $A \cdot B$). Se atribuirmos $C = AB$, e c_{ij} indicar a entrada em C na posição (i, j) , então

$$c_{i,j} = \sum_{r=1}^n a_{i,r} b_{r,j} = a_{i,1} b_{1,j} + a_{i,2} b_{2,j} + \dots + a_{i,n} b_{n,j}$$

para cada elemento i e j com $1 \leq i \leq m$ e $1 \leq j \leq p$. Agora, queremos ver se podemos fazer o cálculo de C em paralelo. Suponha que as matrizes estejam dispostas na memória sequencialmente da seguinte forma: $a_{1,1}, a_{2,1}, a_{3,1}, a_{4,1}, \dots$, etc.

7.6.1 [10] <7.3> Suponha que iremos calcular C em uma máquina de memória compartilhada de único core e uma máquina com memória compartilhada de 4 cores. Calcule o speed-up que esperaríamos obter em uma máquina de 4 cores, ignorando quaisquer problemas de memória.

7.6.2 [10] <7.3> Repita o Exercício 7.6.1, supondo que as atualizações em C incorrem em uma falha de cache, devida ao compartilhamento falso quando os elementos consecutivos que estão em sequência (ou seja, índice i) são atualizados.

7.6.3 [10] <7.3> Como você consertaria o problema de compartilhamento falso que pode ocorrer?

Exercício 7.7

Considere as seguintes partes de dois programas diferentes rodando ao mesmo tempo em quatro processadores em um processador multicore simétrico (SMP). Suponha que, antes que esse código seja executado, tanto x quanto y sejam 0.

Core 1: $x = 2;$

Core 2: $y = 2;$

Core 3: $w = x + y + 1;$

Core 4: $z = x + y;$

7.7.1 [10] <7.3> Quais são todos os valores resultantes possíveis de w, x, y e z? Para cada resultado possível, explique como poderíamos chegar a esses resultados. Você precisará examinar todas as intercalações possíveis das instruções.

7.7.2 [5] <7.3> Como você poderia tornar a execução mais determinística, de modo que somente um conjunto de valores seja possível?

Exercício 7.8

Em um sistema de memória compartilhada CC-NUMA, as CPUs e a memória física são divididas entre os nós de computação. Cada CPU possui caches locais. Para manter a coerência da memória, podemos acrescentar bits de status em cada bloco de cache, ou podemos introduzir diretórios de memória dedicados. Usando diretórios, cada nó oferece uma tabela de hardware dedicada para gerenciar o status de cada bloco de memória que seja “local” a esse nó. O tamanho de cada diretório é uma função do tamanho do espaço compartilhado CC-NUMA (uma entrada é fornecida para cada bloco de memória local a um nó). Se armazenarmos informações de coerência na cache, aumentamos essa informação a cada cache em cada sistema (ou seja, a quantidade de espaço de armazenamento é uma função do número de blocos de cache disponíveis em todas as caches).

7.8.1 [15] <7.3> Se tivermos P CPUs no sistema com T nós no sistema CC-NUMA, com cada CPU tendo C blocos de memória, e mantivermos um byte de informação de coerência em cada bloco de cache, elabore uma equação que expresse a quantidade de memória que estará presente nas caches em um único nó do sistema para manter a coerência. Não inclua o espaço de armazenamento de dados real consumido nessa equação, só considerando o espaço usado para armazenar informações de coerência.

7.8.2 [15] <7.3> Se cada entrada de diretório mantiver um byte de informação para cada CPU, se nosso sistema CC-NUMA tiver um total de S blocos de memória e o sistema tiver T nós, elabore uma equação que expresse a quantidade de memória que estará presente em cada diretório.

Exercício 7.9

Considerando o sistema CC-NUMA descrito no Exercício 7.8, suponha que o sistema tenha quatro nós, cada um com uma CPU de único core (cada CPU tem sua própria cache de dados L1 e cache de dados L2). A cache de dados L1 é store-through, embora a cache de dados L2 seja write-back. Suponha que o sistema tenha uma carga de trabalho na qual uma CPU escreve em um endereço e todas as outras CPUs leiam os dados que são escritos. Suponha também que o endereço em que os dados são escritos esteja inicialmente apenas na memória, e não em qualquer cache local. Além disso, depois da escrita, suponha que o bloco atualizado só esteja presente nas caches L1 do core para formar a escrita.

7.9.1 [10] <7.3> Para um sistema que mantém coerência usando status de bloco baseado em cache, descreva o tráfego entre nós que será gerado à medida que cada um dos quatro cores escreve em um endereço exclusivo, após o qual cada endereço escrito é lido por cada um dos três cores restantes.

7.9.2 [10] <7.3> Para um mecanismo de coerência baseado em diretório, descreva o tráfego entre nós gerado quando o mesmo padrão de código é executado.

7.9.3 [20] <7.3> Repita os Exercícios 7.9.1 e 7.9.2 supondo que cada CPU agora seja uma CPU multicore, com quatro cores por CPU, cada uma mantendo uma cache de dados L1, mas provido de uma cache de dados L2 compartilhada pelos quatro cores. Cada core realizará a escrita, seguida por leituras por cada um dos 15 outros cores.

7.9.4 [10] <7.3> Considere o sistema descrito no Exercício 7.9.3, agora assumindo que cada core escreve em dois bytes diferentes armazenados no mesmo bloco de cache. Como isso afeta o tráfego do barramento? Explique.

Exercício 7.10

Em um sistema CC-NUMA, o custo de acessar a memória não local pode limitar nossa capacidade de utilizar o multiprocessamento com eficiência. A tabela a seguir mostra os custos associados aos dados de acesso na memória local *versus* memória não local e a localidade da nossa aplicação expressa como a proporção de acesso que é local.

Load/store local (ciclo)	Load/store não local (ciclos)	% acessos locais
25	200	20

Responda às perguntas a seguir supondo que os acessos à memória sejam distribuídos uniformemente pela aplicação. Além disso, suponha que somente uma única operação da memória possa estar ativa durante qualquer ciclo. Indique todas as suposições sobre a ordenação das operações de memória local *versus* não local.

7.10.1 [10] <7.3> Se, na média, precisamos acessar a memória uma vez a cada 75 ciclos, qual é o impacto sobre nossa aplicação?

7.10.2 [10] <7.3> Se, na média, precisamos acessar a memória uma vez a cada 50 ciclos, qual é o impacto sobre nossa aplicação?

7.10.3 [10] <7.3> Se, na média, precisamos acessar a memória uma vez a cada 100 ciclos, qual é o impacto sobre nossa aplicação?

Exercício 7.11

O problema do jantar dos filósofos é um problema clássico de sincronização e concorrência. O problema geral é enunciado como filósofos sentados em volta de uma mesa redonda fazendo uma de duas coisas: comendo ou pensando. Quando eles estão comendo, não estão pensando, e quando estão pensando, não estão comendo. Há uma tigela de macarrão no centro. Um garfo é colocado entre cada filósofo. O resultado é que cada filósofo tem um garfo à sua esquerda e um garfo à sua direita. Devido à forma como se come macarrão, o filósofo precisa de dois garfos para comer, e só pode usar os garfos do seu lado esquerdo e direito. Os filósofos não conversam entre si.

7.11.1 [10] <7.4> Descreva o cenário em que nenhum dos filósofos consegue comer (ou seja, inanição). Qual é a sequência de eventos que leva a esse problema?

7.11.2 [10] <7.4> Descreva como podemos solucionar esse problema introduzindo o conceito de uma prioridade. Mas podemos garantir que trataremos de todos os filósofos de forma justa? Explique.

Agora, suponha que contratemos um garçom encarregado de atribuir garfos aos filósofos. Ninguém pode pegar um garfo até que o garçom lhe diga que pode. O garçom tem conhecimento global de todos os garfos. Além disso, se impusermos a diretriz de que os filósofos sempre solicitarão para apanhar seu garfo da esquerda antes de apanhar seu garfo da direita, então podemos garantir que o impasse (deadlock) será evitado.

7.11.3 [10] <7.4> Podemos implementar as solicitações ao garçom como uma fila de solicitações ou como uma retentativa periódica de uma solicitação. Com uma fila, as solicitações são tratadas na ordem em que são recebidas. O problema com o uso da

fila é que podemos nem sempre ser capazes de atender ao filósofo cuja solicitação está no início da fila (devido à indisponibilidade de recursos). Descreva um cenário com cinco filósofos, em que uma fila é fornecida, mas o serviço não é concedido mesmo que haja garfos disponíveis para outro filósofo (cuja solicitação está mais profunda na fila) utilizar.

7.11.4 [10] <7.4> Se implementarmos solicitações ao garçom repetindo periodicamente nossa solicitação até que os recursos estejam disponíveis, isso solucionará o problema descrito no Exercício 7.11.3? Explique.

Exercício 7.12

Considere as três organizações de CPU a seguir:

CPU SS: Um microprocessador superescalar de dois cores que oferece capacidades de emissão fora de ordem em duas unidades funcionais (FUs). Somente uma única thread pode ser executada em cada core de cada vez.

CPU MT: Um processador multithreaded fine-grained que permite que instruções de duas threads sejam executadas simultaneamente (ou seja, existem duas unidades funcionais), embora somente instruções de uma única thread possam ser emitidas em cada ciclo.

CPU SMT: Um processador SMT que permite que instruções de duas threads sejam executadas simultaneamente (ou seja, existem duas unidades funcionais), e as instruções de qualquer uma ou ambas as threads podem ser emitidas para executar em qualquer ciclo.

Suponha que tenhamos duas threads, X e Y, para executar nessas CPUs, o que inclui as seguintes operações:

Thread X	Thread Y
A1 – leva 3 ciclos para executar	B1 – leva 2 ciclos para executar
A2 – sem dependências	B2 – conflitos para uma unidade funcional com B1
A3 – conflitos para uma unidade funcional com A1	B3 – depende do resultado de B2
A4 – depende do resultado de A3	B4 – sem dependências e leva 2 ciclos para executar

Suponha que todas as instruções utilizem um único ciclo para serem executadas, a menos que observado de outra forma ou que encontrem um hazard.

7.12.1 [10]<7.5> Suponha que você tenha uma CPU SS. Quantos ciclos são necessários para executar essas duas threads? Quantos slots de emissão são desperdiçados devido a hazards?

7.12.2 [10]<7.5> Agora, suponha que você tenha uma CPU MT. Quantos ciclos são necessários para executar essas duas threads? Quantos slots de emissão são desperdiçados devido a hazards?

7.12.3 [10]<7.5> Suponha que você tenha uma CPU SMT. Quantos ciclos são necessários para executar essas duas threads? Quantos slots de emissão são desperdiçados devido a hazards?

Exercício 7.13

O software de virtualização está sendo agressivamente implantado para reduzir os custos de gerenciamento dos servidores de alto desempenho de hoje. Empresas como VMWare,

Microsoft e IBM desenvolveram diversos produtos de virtualização. O conceito geral, descrito no Capítulo 5, é que uma camada hipervisora pode ser introduzida entre o hardware e o sistema operacional para permitir que vários sistemas operacionais compartilhem o mesmo hardware físico. A camada hipervisora é então responsável por alocar recursos de CPU e memória, além de tratar os serviços normalmente tratados pelo sistema operacional (por exemplo, E/S).

A virtualização oferece uma visão abstrata do hardware subjacente ao sistema operacional host e ao software de aplicação. Isso exigirá que repensemos como os sistemas multicore e multiprocessador serão projetados no futuro para dar suporte ao compartilhamento das CPUs e memórias por diversos sistemas operacionais simultaneamente.

7.13.1 [30] <7.5> Selecione dois hipervisores no mercado hoje e compare como eles virtualizam e gerenciam o hardware subjacente (CPUs e memória).

7.13.2 [15] <7.5> Discuta quais mudanças podem ser necessárias nas plataformas de CPU multicore do futuro a fim de que sejam mais coerentes com as demandas de recursos impostas sobre esses sistemas. Por exemplo, o multithreading pode desempenhar um papel eficaz para reduzir a competição por recursos de computação?

Exercício 7.14

Gostaríamos de executar o loop a seguir da forma mais eficiente possível. Temos duas máquinas diferentes, uma máquina MIMD e uma máquina SIMD.

```
for (i=0; i < 2000; i++)
    for (j=0; j<3000; j++)
        X_array[i][j] = Y_array[j][i] + 200;
```

7.14.1 [10] <7.6> Para uma máquina MIMD com quatro CPUs, mostre a sequência de instruções MIPS que você executaria em cada CPU. Qual é o speed-up para essa máquina MIMD?

7.14.2 [20] <7.6> Para uma máquina SIMD de largura 8 (ou seja, oito unidades funcionais SIMD paralelas), escreva um programa em assembly usando suas próprias extensões SIMD para o MIPS para executar o loop. Compare o número de instruções executadas na máquina SIMD com a máquina MIMD.

Exercício 7.15

Um array sistólico é um exemplo de uma máquina MISD. Um array sistólico é uma rede de pipeline ou “wavefront” de elementos de processamento de dados. Cada um desses elementos não precisa de um contador de programa, pois a execução é disparada pela chegada de dados. Os arrays sistólicos com clock são computados em “lock-step”, com cada processador realizando fases alternadas de computação e comunicação.

7.15.1 [10] <7.6> Considere as implementações propostas de um array sistólico (você pode encontrá-las na internet ou em publicações técnicas). Depois, tente programar o loop fornecido no Exercício 7.14 usando esse modelo MISD. Discuta quaisquer dificuldades que você encontrar.

7.15.2 [10] <7.6> Discuta as semelhanças e diferenças entre uma máquina MISD e SIMD. Responda a essa pergunta em termos de paralelismo em nível de dados.

Exercício 7.16

Suponha que queiramos executar o loop DAXP mostrado na Seção 7.6 em assembly MIPS na GPU NVIDIA 8800 GTX descrita neste capítulo. Nesse problema, vamos supor que todas as operações matemáticas sejam realizadas em números de ponto flutuante com precisão simples (vamos mudar o nome do loop para SAXP). Suponha que as instruções utilizem o seguinte número de ciclos para serem executadas.

Loads	Stores	Add.S	Mult.S
5	2	3	4

7.16.1 [20] <7.7> Descreva como você construirá warps para o loop SAXP explorar os oito cores fornecidos em um único multiprocessador.

Exercício 7.17

Faça o download do CUDA Toolkit e SDK em www.nvidia.com/object/cuda_get.html. Lembre-se de usar a versão “emurelês” (Emulation Mode) do código (você não precisará do hardware NVIDIA real para esse trabalho). Crie os programas de exemplo fornecidos no SDK e confirme se eles rodarão no emulador.

7.17.1 [90] <7.7> Usando o “template” de exemplo de SDK como ponto de partida, escreva um programa CUDA para realizar o seguinte vetor de operações:

1. $a - b$ (subtração vetor-vetor)
2. $a \cdot b$ (produto pontual de vetor)

O produto pontual de dois vetores $a = [a_1, a_2, \dots, a_n]$ e $b = [b_1, b_2, \dots, b_n]$ é definido como:

$$a \cdot b = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$$

Submeta o código para cada programa que demonstra cada operação e verifica a exatidão dos resultados.

7.17.2 [90] <7.7> Se você tiver hardware GPU disponível, complete uma análise de desempenho do seu programa, examinando o tempo de computação para a GPU e uma versão de CPU do seu programa para uma faixa de tamanhos de vetor. Explique quaisquer resultados que você encontrar.

Exercício 7.18

A AMD recentemente anunciou que estarão integrando uma unidade de processamento gráfico com seus cores x86 em um único pacote, embora com diferentes clocks para cada um dos cores. Este é um exemplo de um sistema multiprocessador heterogêneo que esperamos ver produzido comercialmente no futuro próximo. Um dos principais pontos de projeto será permitir a comunicação de dados rápida entre a CPU e a GPU. Atualmente a comunicação deve ser realizada entre chips de CPU e GPU discretos. Mas isso está mudando na arquitetura Fusion da AMD. Atualmente, o plano é usar múltiplos canais PCI express (pelo menos, 16) para facilitar a intercomunicação. A Intel também está saltando para essa arena com seu chip Larrabee. A Intel está considerando o uso de sua tecnologia de interconexão QuickPath.

7.18.1 [25] <7.7> Compare a largura de banda e latência associadas a essas duas tecnologias de interconexão.

Exercício 7.19

Consulte a [Figura 7.9b](#), que mostra uma topologia de interconexão de cubo n de ordem 3, que interconecta oito nós. Um recurso atraente de uma topologia de rede de interconexão de cubo n é sua capacidade de sustentar links partidos e ainda oferecer conectividade.

7.19.1 [10] <7.8> Desenvolva uma equação que calcule quantos links no cubo n (onde n é a ordem do cubo) podem falhar e ainda podemos garantir que um link não partido existirá para conectar qualquer nó no cubo n .

7.19.2 [10] <7.8> Compare a resiliência a falha do cubo n com uma rede de interconexão totalmente conectada. Desenhe uma comparação da confiabilidade como uma função do número de links que podem falhar para as duas topologias.

Exercício 7.20

O benchmarking é um campo de estudo que envolve identificar cargas de trabalho representativas para rodar em plataformas de computação específicas a fim de poder comparar objetivamente o desempenho de um sistema com outro. Neste exercício, vamos comparar duas classes de benchmarks: o benchmark Whetstone CPU e o pacote de benchmark PARSEC. Selecione um programa do PARSEC. Todos os programas deverão estar disponíveis gratuitamente na internet. Considere a execução de múltiplas cópias do Whetstone contra a execução do benchmark PARSEC em qualquer um dos sistemas descritos na Seção 7.11.

7.20.1 [60] <7.9> O que é inherentemente diferente entre essas duas classes de carga de trabalho quando executadas nesses sistemas multicore?

7.20.2 [60] <7.9, 7.10> Em termos do modelo roofline, que dependência terão os resultados que você obtiver ao executar esses benchmarks na quantidade de compartilhamento e sincronização presente na carga de trabalho utilizada?

Exercício 7.21

Ao realizar cálculos sobre matrizes esparsas, a latência na hierarquia de memória torna-se um fator muito importante. As matrizes esparsas não possuem a localidade espacial no fluxo de dados, normalmente encontrada nas operações de matriz. Como resultado, novas representações de matriz foram propostas.

Uma das representações de matriz esparsa mais antigas é o Yale Sparse Matrix Format. Ele armazena uma matriz esparsa inicial $m \times n$, M , em formato de linha usando três arrays unidimensionais. Suponha que R indique o número de entradas diferentes de zero em M ; podemos construir um array A de tamanho R que contém todas as entradas diferentes de zero de M (na ordem da esquerda para a direita e de cima para baixo). Também construímos um segundo array IA de tamanho $m + 1$ (ou seja, uma entrada por linha, mais um). $IA(i)$ contém o índice de A do primeiro elemento diferente de zero da linha i . A linha i da matriz original se estende de $A(IA(i))$ até $A(IA(i + 1) - 1)$. O terceiro array, JA , contém o índice de coluna de cada elemento de A , de modo que também tem tamanho R .

7.21.1 [15] <7.9> Considere a matriz esparsa X a seguir e escreva o código C que armazenaria esse código no Yale Sparse Matrix Format.

```

Linha 1[1,2,0,0,0,0]
Linha 2[0,0,1,1,0,0]
Linha 3[0,0,0,0,9,0]
Linha 4[0,0,3,3,0,7]
Linha 5[1,3,0,0,0,1]

```

7.21.2 [10] <7.9> Em termos do espaço de armazenamento, supondo que cada elemento na matriz X tenha formato de ponto flutuante com precisão simples, calcule a quantidade de armazenamento usada para armazenar a matriz acima no Yale Sparse Matrix Format.

7.21.3 [15] <7.9> Realize a multiplicação de matriz da Matriz X pela Matriz Y mostrada a seguir.

$$[2, \quad 4, \quad 1, \quad 99, \quad 7, \quad 2]$$

Coloque esse cálculo em um loop e meça o tempo de sua execução. Não se esqueça de aumentar o número de vezes que esse loop é executado para obter uma boa resolução na sua medição do tempo. Compare o tempo de execução do uso de uma representação simples da matriz e do Yale Sparse Matrix Format.

7.21.4 [15] <7.9> Você consegue achar uma representação de matriz esparsa mais eficiente (em termos de overhead de espaço e computacional)?

Exercício 7.22

Nos sistemas do futuro, esperamos ver plataformas de computação heterogêneas construídas a partir de CPUs heterogêneas. Começamos a ver algumas aparecendo no mercado de processamento embutido nos sistemas que contêm DSPs de ponto flutuante e CPUs de microcontrolador em um pacote de módulo multichip.

Suponha que você tenha três classes de CPU:

CPU A — Uma CPU multicore de velocidade moderada (com uma unidade de ponto flutuante) que pode executar múltiplas instruções por ciclo.

CPU B — Uma CPU inteira de único core rápida (ou seja, sem unidade de ponto flutuante) que pode executar uma única instrução por ciclo.

CPU C — Uma CPU de vetor lenta (com capacidade de ponto flutuante) que pode executar múltiplas cópias da mesma instrução por ciclo.

Suponha que nossos processadores executem nas seguintes frequências:

CPU A	CPU B	CPU C
1 GHz	3 GHz	250 MHz

A CPU A pode executar 2 instruções por ciclo, a CPU B pode executar 1 instrução por ciclo, e a CPU C pode executar 8 instruções (embora a mesma instrução) por ciclo. Suponha que todas as operações possam concluir sua execução em um único ciclo de latência sem quaisquer hazards.

Todas as três CPUs possuem a capacidade de realizar aritmética com inteiros, embora a CPU B não possa realizar aritmética de ponto flutuante diretamente. As CPUs A e B têm um conjunto de instruções semelhante a um processador MIPS. A CPU C só pode realizar operações de soma e subtração de ponto flutuante, assim como loads e stores de

memória. Suponha que todas as CPUs tenham acesso à memória compartilhada e que a sincronização tenha custo zero.

A tarefa em mãos é comparar duas matrizes X e Y, que contêm cada uma 1024×1024 elementos de ponto flutuante. A saída deverá ser uma contagem dos índices numéricos em que o valor em X foi maior que o valor em Y.

7.22.1 [10] <7.11> Descreva como você particionaria o problema nas três CPUs diferentes para obter o melhor desempenho.

7.22.2 [10] <7.11> Que tipo de instrução você acrescentaria à CPU de vetor C para obter o melhor desempenho?

Exercício 7.23

Suponha que um sistema de computador quad-core possa processar transações de banco de dados em uma taxa de estado constante de solicitações por segundo. Suponha também que cada instrução leve, na média, uma quantidade de tempo fixa para ser processada. A tabela a seguir mostra pares de latência de transação e taxa de processamento.

Latência de transação média	Taxa de processamento de transação máxima
1 ms	5000/seg
2 ms	5000/seg
1 ms	10.000/seg
2 ms	10.000/seg

Para cada um dos pares na tabela, responda às seguintes perguntas:

7.23.1 [10] <7.11> Na média, quantas solicitações estão sendo processadas em determinado instante?

7.23.2 [10] <7.11> Se você passasse para um sistema de 8 cores, na forma ideal, o que aconteceria com a vazão do sistema (ou seja, quantas transações/segundo o computador processará)?

7.23.3 [10] <7.11> Discuta por que raramente obtemos esse tipo de speed-up simplesmente aumentando o número de cores.

§7.1: Falso. O paralelismo em nível de tarefa pode ajudar as aplicações sequenciais e as aplicações sequenciais podem ser criadas para executar em hardware paralelo, embora isso seja mais difícil.

§7.2: Falso. A expansão *fraca* pode compensar uma parte serial do programa que, de outra forma, limitaria a expansão.

§7.3: Falso. Como o endereço compartilhado é um endereço *físico*, múltiplas tarefas em seus próprios espaços de endereço *virtual* podem executar bem em um multiprocessador de memória compartilhada.

§7.4: 1. Falso. Enviar e receber uma mensagem é uma sincronização implícita, além de um modo de compartilhar dados. 2. Verdadeiro.

§7.5: 1. Verdadeiro. 2. Verdadeiro.

§7.6: Verdadeiro.

§7.7: Falso. DIMMs de DRAM gráfica são apreciadas por sua largura de banda mais alta.

§7.9: Verdadeiro. Provavelmente precisamos de inovação em todos os níveis da pilha de hardware e software para vencer a aposta da indústria na computação paralela.

**Respostas das
Seções “Verifique
você mesmo”**