Explorando o Paralelismo no nível de *Threads*

Márcio Machado Pereira
IC/UNICAMP
RA 780681
mpereira@ic.unicamp.br

Luiz Augusto Biazotto Filho
IC/UNICAMP
RA 121493
biazotto@gmail.com

SUMÁRIO

Ao longo das últimas décadas o crescimento no desempenho dos computadores foi resultado de uma combinação de dois fatores: (a) aumento na velocidade dos circuitos, e (b) adoção de técnicas de micro-arquitetura capazes de detectar paralelismo em nível de instruções (ILP). Isto permitiu dobrar o desempenho dos computadores a cada 18-24 meses (Lei de Moore). Infelizmente, a redução no tamanho dos transistores resultou em um aumento na densidade de potência dos processadores, inviabilizando o projeto de um único superprocessador. A solução foi então espalhar os transistores em vários *cores*, que dissipassem menos potência, resultando nas arquiteturas *multicore* modernas. Fabricantes de processadores têm adotado arquiteturas *multicores* diferentes. Nota-se, por exemplo, uma tendência de se combinar CPUs e GPUs (Graphics Processing Units) ou Engines. Este é o caso da Intel (SandyBridge), AMD (Fusion), NVIDIA (Tegra) e IBM (Power7). Por outro lado, independente da arquitetura, extrair paralelismo de programas tornou-se uma questão central com vistas ao aumento no desempenho dos programas executando nestes processadores. Uma vez que laços são responsáveis pela maior parte do tempo de execução de um programa, detectar e extrair paralelismo de laços tornou-se uma das tarefas mais importantes nesta área. Este trabalho apresenta duas arquiteturas que exploram o paralelismo no nível de *threads* (TLP) e as principais técnicas utilizadas pelos compiladores modernos para a extração automática destas threads.

Categorias e Descritores de Assunto

C.1.2 [Arquiteturas de Processador]: Arquiteturas Paralelas, Multicores.

D.3.4 [Linguagens de Programação]: Paralelização automática, Compiladores, Geração de Código.

Termos Gerais

Desempenho, *Projeto*.

Palavras Chaves

Paralelismo no nível de *threads*, multiprocessadores, *Chip-Multithreading* (CMP), *Simultaneous Multithreading* (SMT), DOALL, DOACROSS, DOPIPE, DSWP.

Este trabalho foi apresentado no curso de Arquitetura de Computadores MO401 ministrado pelo prof. Dr. Paulo Centoducatte, no primeiro semestre de 2011. O mesmo é requisito de avaliação da disciplina pelo Instituto de Computação da UNICAMP (IC-Unicamp) e foi elaborado pelos alunos Pereira, M. M. (e-mail: mpereira@ic.unicamp.br) e Biaznotto, L. A. (email: biazotto@gmail.com).

1. INTRODUÇÃO

Na grande maioria das aplicações, as melhorias de desempenho têm sido principalmente resultado de dois fatores: Em primeiro lugar, e o mais importante, os processadores tornaram-se cada vez mais rápidos devido às melhorias tecnológicas. A velocidade de clock dos processadores melhorou de 5 KHz em 1940 (ENIAC) para mais de 3 GHz com a introdução do Pentium 4. O segundo fator de aumento no desempenho foi a melhoria constante da micro-arquitetura dos processadores (por exemplo, *caches*, *pipelining* e previsão de uso) acompanhado também da evolução no processo de compilação. Juntos, os efeitos destas melhorias representaram um aumento estimado de 40% no desempenho por ano [1].

Este cenário, de longa duração, de constante melhoria no desempenho está mudando. Recentemente, a indústria de microprocessadores atingiu duramente os limites físicos, incluindo aqui a dissipação de calor e o consumo de energia. Além disso, a introdução de técnicas mais agressivas na micro-arquitetura para uma maior exploração do paralelismo no nível de instrução (ILP) [2] tem afetado negativamente estas questões físicas, como por exemplo, a introdução de mais componentes de hardware, tornando a lógica de execução das instruções mais complexa, além de aumentar o consumo de potência. Portanto, nem o aumento nas freqüências de *clock*, nem os avanços na exploração de ILP estão contribuindo de forma significativa para melhorar o desempenho. Como resultado, o desempenho da maioria dos aplicativos não tem aumentado no mesmo ritmo que costumava acontecer há várias décadas. Uma vez que estas eram as duas principais fontes de aumento no desempenho, a industria e o meio acadêmico voltam-se agora para a exploração do paralelismo no nível de *threads* (TLP), que foi a terceira e muito menos importante fonte de desempenho ao longo das últimas décadas.

A boa notícia é que o paralelismo está se tornando mais barato. Com o aumento contínuo no número de transistores por chip, os processadores de vários núcleos (*multicores*) se tornaram um padrão. Os processadores *multicores* são encontrados hoje em *desktops*, *laptops* e até em telefones celulares. Estas máquinas "paralelas" estão disponíveis na maioria dos sistemas, mesmo quando os usuários ou aplicativos não a utilizam. Além disso, comparado aos tradicionais processadores multi-simétricos (SMP), a natureza fortemente integrada dos processadores em um único chip traz o potencial para explorar o paralelismo em um nível de granularidade mais fina.

Apesar da recente disponibilidade destas máquinas paralelas, as maiorias das aplicações existentes não conseguem se beneficiar deste poder computacional. Este fato se deve a duas razões principais: a falta de aplicações escritas em linguagens paralelas,

e a aplicabilidade restrita de técnicas de paralelização automática.

Aplicações paralelas podem ser obtidas fazendo com que os programadores escrevam aplicações em linguagens de programação paralela (por exemplo, *Erlang*[14]). No entanto, a maioria das aplicações ainda são escritas em linguagens seqüenciais pela facilidade de programação. Em paradigmas de programação paralela, o programador precisa raciocinar sobre concorrência, custos de comunicação, localidade de dados e se preocupar com novos problemas causados por um grande número de *interleavings* possíveis de múltiplas linhas (*threads*) de execução. Falhas comuns de concorrência incluem *data races*, *deadlocks* e *livelocks*. O grande interesse em instrumentos para ajudar a detectar esses erros [3, 4] é um testamento à sua freqüência, dificuldade de detecção e custo. Dados os custos elevados de desenvolvimento e manutenção de programas paralelos, a partir de uma perspectiva de engenharia de software, definitivamente é melhor utilizar os modelos de programação seqüenciais.

Mesmo escritas no paradigma seqüencial, as aplicações podem, potencialmente, serem traduzidas em códigos paralelos. Isto pode ser conseguido com o auxílio de ferramentas automáticas, especialmente pelos compiladores. Infelizmente, a paralelização automática só foi bem sucedida até agora em domínios de aplicação restrita, principalmente onde o paralelismo de dados é abundante. Em particular, a paralelização automática tem sido eficaz para aplicações científicas e de multimídia. Em domínios de aplicação mais geral, estas técnicas não têm sido capazes de extrair um paralelismo útil. Particularmente, a presença irregular de acessos à memória (como em estruturas de dados ligadas) e um controle de fluxo arbitrário e intenso, a maioria das técnicas de paralelização proposta ao longo das últimas décadas e implementadas nos modernos compiladores não se mostraram eficazes.

Nas próximas seções, faremos uma breve descrição de duas das arquiteturas mais importantes que exploram o paralelismo no nível de *threads* (TLP) e discorreremos sobre alguns dos trabalhos científicos sobre extração automática de *threads* nas granularidades grossa e fina. Esta visão ajudará a compreender tanto os resultados bem como as limitações destas técnicas, colocando assim o leitor no contexto, motivações e desafios que envolvem os recentes trabalhos acadêmicos. Em seguida, concluiremos nossa apresentação procurando apontar direções onde possam surgir contribuições para uma extração de *threads* mais efetiva, visando uma melhor exploração do paralelismo.

2. ARQUITETURAS MULTITHREADING

Apesar da técnica de exploração de ILP ser, muitas vezes, transparente ao programador, na prática ela tem-se mostrado limitada. No entanto, observa-se que muitas aplicações costumam apresentar naturalmente um paralelismo de mais alto nível. São exemplos, as transações *online* onde há grande quantidade de requisições e respostas, mas quase sempre independentes. Aplicações científicas também são candidatas a explorar paralelismo em alto nível. A este paralelismo em alto nível, damos o nome de TLP, porque é logicamente estruturado como tarefas ou linhas de execução (*threads)* distintos. Uma *thread* pode representar um processo, que é parte de um programa paralelo contendo vários processos, ou ainda pode representar o próprio programa independente. Cada *thread* tem todo o estado (dados de instruções, contador de programa, estado de registradores, etc.) necessários para permitir sua execução. Diferentemente do ILP, que explora implicitamente operações paralelas dentro de laços e blocos de um programa, TLP é explicitamente representado pelo uso de várias *threads* de execução que são essencialmente paralelas. TLP tem-se mostrado uma alternativa interessante ao ILP por ter um custo menor. Uma outra motivação para a exploração do TLP surgiu da observação que um *data path* construído para explorar ILP costuma apresentar unidades funcionais livres em virtude dos *stalls* e das dependências no código. A técnica *Multithreading* permite então que múltiplas *threads* compartilhem as unidades funcionais de um único processador de maneira intercalada. Para isso, o processador deve duplicar o estado independente de cada *thread*, como cópia separada do arquivo de registradores, contador de programa e uma tabela de página individual. A memória pode ser compartilhada através de mecanismo de memória virtual, que já suporta multi-programação. O hardware tem que suportar que a troca de *threads* seja feita mais rapidamente que a troca de processos, que pode levar muitos ciclos.

Quando falamos em *Multithreading*, podemos dizer que há duas principais formas para realizar a troca entre *threads*, qual seja, a granularidade fina e a grossa. Na granularidade fina, a troca entre *threads* ocorre a cada instrução, e para que isso seja possível, a CPU tem que trocar de *thread* a cada ciclo. A vantagem deste tipo de troca é que ela permite ocultar a perda em largura de banda para casos de *stalls* curtos e longos, uma vez que, assim que uma *thread* entra em *stall*, instruções de outra thread podem ser executadas. A desvantagem da granularidade fina é que ela atrasa a execução de *threads* de forma individual, uma vez que uma thread que estava preparada para ser executada sem *stall* terá sua execução atrasada por instruções de outras threads. Na granularidade grossa, apenas há troca de *threads* quando acontece um *stall* que costuma ter uma penalidade grande, como, por exemplo, um *miss* na *cache* de nível 2. Em conseqüência desta característica, a performance de uma *thread* individual é menos afetada. Em contrapartida, a granularidade grossa apresenta um problema ainda maior que a fina, pois tem um *pipeline* limitado para despachar instruções de uma única *thread*. Isto significa que a cada troca de *threads*, o *pipeline* precisa ser esvaziado, acarretando em um custo em termos de performance para preenchê-lo. Logo, só faz sentido utilizá-la em casos onde o *stall* é tão longo, que vai demorar mais tempo que o tempo do preenchimento do *pipeline*. Nas sub-seções a seguir, iremos descorrer sobre duas das principais abordagens de *multithreading* presentes nos processadores comerciais, como a abordagem *Simultaneous Multithreading* (SMT) e *Chip Multiprocessor* (CMP).

2.1 SMT

A abordagem SMT pode ser entendida como uma variação sobre a abordagem da granularidade fina, que utiliza os recursos de um processador de emissão múltipla dinamicamente escalonado para explorar o TLP ao mesmo tempo em que explora o ILP [2]. A motivação para SMT é que os processadores modernos de emissão múltipla freqüentemente têm maior paralelismo de unidades funcionais disponíveis do que uma única *thread* pode utilizar efetivamente. Em um processador SMT [23], são feitas modificações no hardware para que seja possível manipular mais de uma entidade computacional (programa, tarefa, processo ou *thread*), aumentando o *throughput*. Uma vez que há muita dependência entre instruções de uma única entidade computacional, (como, por exemplo, uma espera de uma operação *load* para que uma adição seja realizada sobre a informação carregada), e essas dependências limitam o ILP, o processador terá unidades de execução livres. Para ocupar essas unidades, SMT propõe que instruções possam ser executadas de múltiplas entidades computacionais, uma vez que não há

dependência entre estas instruções, exceto as físicas, que são facilmente resolvidas. O mecanismo que permite fazer o *fetch* de instruções das entidades computacionais é dependente de implementação, mas, em geral, em cada ciclo, uma instrução de uma entidade diferente é escolhida. A maior vantagem do conceito SMT é que ele requer apenas a adição de 5-25% de transistores comparando com uma tecnologia não SMT. Estes transistores adicionais armazenam informações separadamente para todas as entidades computacionais, de maneira que há múltiplos ponteiros de instrução, contadores de programa e registradores. É responsabilidade do arquiteto do processador definir quais recursos serão replicados além dos recursos necessários, como também quais recursos serão igualmente divididos ou compartilhados entre as entidades computacionais. Um problema a ser gerenciado é que os processadores SMT devem ser iniciados em um modo de *thread* único e, depois, migrados para SMT, o que requer alterações no sistema operacional e na BIOS.

Comercialmente, a Intel implementou SMT nos processadores Xeon, para uso em servidores corporativos, e nos processadores Pentium 4, para uso em estações de trabalho, batizando est novaa tecnologia de *Hyper-Threading*[24]. Na tecnologia Hyper-Threading, desenvolvida pela Intel, um único processador físico funciona como dois processadores lógicos, que compartilham recursos do núcleo de processamento. Cada processador lógico mantém um conjunto completo do estado da arquitetura, que consiste dos registradores de propósito geral, registradores de controle, registradores do controle avançado de interrupções programáveis (APIC) e alguns registradores de controle da máquina. Os processadores lógicos compartilham quase todos os recursos do processador físico, como as *caches*, unidades de execução e barramentos. Como cada processador lógico possui seus próprios registradores APIC, interrupções enviadas a um processador lógico específico são atendidas somente por este. A execução contínua e independente de instruções dos dois processadores lógicos é assegurada através do gerenciamento das filas de espera de recursos compartilhados (*buffers*), de modo que nenhum processador lógico utilize todas as entradas desta quando duas *threads* estão ativas no mesmo processador físico. Há uma política de gerenciamento flexível desses recursos, de maneira que, se houver apenas uma *thread* ativa no processador físico, este deve executar na mesma velocidade de um processador não SMT. A Intel voltou a usar o conceito de *Hyper-Threading* em sua nova micro-arquitetura *Nehalem*.

2.2 CMP

Um *Chip Multiprocessor* (CMP) é simplesmente um grupo de processadores integrados em um mesmo chip, de forma que possam atuar como um time, ocupando a área que originalmente seria preenchida com um único e grande processador, agora, com vários *cores* menores [25]. Também pode ser chamado de *multicores*, ou ainda, *manycore multiprocessor*. Do ponto de vista de engenharia, é mais fácil de ser implementado, pois é uma multiplicação de uma mesma família de processadores, com pequenos ajustes de demanda de banda e latência no *pipeline*. Do ponto de vista de conectividade externo, o processador parece o mesmo que um *single core*, de forma que poderia ser utilizado nas mesmas placas, desde que estas aumentem a capacidade de gerenciamento de largura de banda para entrada e saída de dados e memória.

Os *cores* são, agora, vistos pelos programadores como entidades separadas, de maneira que temos que substituir o modelo computacional de Von Neumann por um novo modelo de programação paralela. Com este tipo de modelo, programadores são incitados a dividir suas aplicações em partes independentes ou semi-independentes (*threads*), de forma que estas partes possam operar simultaneamente nos *cores*, ou seus programas não aproveitarão o poder de processamento disponível no modelo CMP. Uma vez que *threads* são criados, programas podem aproveitar as vantagens do TLP rodando as *threads* separadamente e em paralelo, além de explorar ILP entre instruções individuais dentro de cada *thread*. Como na maioria de aplicações servidoras há bastante TLP, pouco ILP e alta taxa de *miss* em cache, existe motivação suficiente para a utilização de *multithreading* também em CMP, adicionando recurso em *hardware* para cada *core* de modo que estes consigam despachar instruções de múltiplas *threads*, sejam elas simultâneas ou não, sem a necessidade de intervenção do sistema operacional ou de outro software para fazer a troca entre *threads*. Dá-se o nome a esta abordagem de CMT (*Chip Multithreading*), ou seja, os processadores CMP também possuem recursos em hardware para despachar instruções para múltiplos threads (*multicore* e *multithreaded*).

Um exemplo é o processador Sun T1 (Niagara), que utiliza a granularidade fina como abordagem para *multithreading* [2]. O processador Niagara [26] despacha apenas uma instrução por ciclo, mas consegue gerenciar até quatro *threads* individuais de execução. Cada *thread* tem seu próprio contador de programa, *buffer* de instruções one-line e arquivo de registro. As quatros *threads* compartilham a *cache* de instruções, *cache* de dados e vários outros recursos de execução no *core*. Para trocar entre *threads*, utiliza mecanismo *least recently used* (LRU). Quando as quatro *threads* estão disponíveis, o *scheduler* utiliza o mecanismo de *round robin*, executando uma instrução de cada *thread*. Se alguma *thread* entra em *stall*, ela é retirada da lógica de escolha de *threads* até que esteja novamente disponível para executar suas instruções. O processador Niagara 2 utiliza a técnica SMT, ou seja, une CMP capaz de explorar TLP via SMT.

3. EXTRAÇÃO AUTOMÁTICA DE THREADS

Nesta seção, discorreremos sobre os principais problemas na área de detecção e extração de paralelismo de laços de programas e as técnicas e métodos mais utilizados para paralelizar um programa.

3.1 *Coarse-Grained* TLP

A grande maioria das técnicas de paralelização automática focam em extrair o paralelismo de "maior granularidade" (*coarse-grained*) no nível de laços (*loops*). Estas técnicas normalmente exploram o paralelismo entre as diferentes iterações de um laço cujas iterações são todas independentes. Esses laços são conhecidos como laços DOALL [5]. Os desafios para a exploração de paralelismo DOALL vêm de duas alternativas complementares: (a) analisar um laço aninhado para provar que suas iterações são independentes, e (b) transformar um laço aninhado não DOALL em um equivalente DOALL. Muitas técnicas têm sido propostas para responder às alternativas, incluindo os testes GCD e Ômega, *polyhedral methods, loop skewing*, e *loop distribution* [5]. Compiladores com otimizações baseados nestas técnicas geralmente podem encontrar quantidades razoáveis de paralelismo em aplicações científicas. Neste domínio de aplicação, os programas normalmente contêm laços sem dependências entre as iterações (dependencias de malha ou *loop-carried dependence*), poucos controles de fluxo e acessos à memória de forma regular (baseada em vetores). No

entanto, fora do domínio da computação científica, essas técnicas são raramente aplicáveis. As razões para isso são principalmente a grande quantidade de acessos à memória de forma irregular (baseadas em estruturas ligadas, ou ponteiros), muitos fluxos de controle e laços com dependencias entre as iterações, cacterísticas estas que são muito difíceis, se não impossíveis de eliminar.

Na presença de laços com dependências de malha (*loop-carried dependence*), uma técnica de paralelização notável tem sido proposta, denominado DOACROSS [5]. Com DOACROSS, as iterações do laço são executados em vários processadores na forma de rodízio (*round-robin fashion*). Para respeitar as dependências de malha, primitivas de sincronização são inseridas no código, de modo que as instruções em uma iteração aguardem suas instruções dependentes nas iterações anteriores para prosseguir. Apesar de algum sucesso, os benefícios da técnica DOACROSS são geralmente limitadas por dois fatores. Primeiro, o número e posição das dependências de malha levam a sincronizações no código que limitam a quantidade de paralelismo. Em segundo lugar, dividindo-se as iterações do laço entre os processadores, DOACROSS insere sincronizações no caminho crítico de execução do loop. Em outras palavras, o custo da sincronização é pago entre cada par de iterações, essencialmente, multiplicando o custo de sincronização com o número de iterações do loop. No entanto, DOACROSS tem encontrado aplicabilidade quando combinada com técnicas de especulação conhecidas como *Thread-level Speculation* (TLS) [6, 7]. Infelizmente, mesmo com o apoio da especulação, a quantidade de paralelismo obtido por estas técnicas tem sido limitado, dificilmente justificando o suporte complexo de hardware necessário para manter o overhead destas técnicas em níveis aceitáveis.

3.2 *Fine-Grained* TLP

Dada a dificuldade de encontrar um paralelismo *coarse-grained* na maioria das aplicações, pesquisadores têm investigado a possibilidade de extrair paralelismo em granularidades mais finas (*fine-grained*). O potencial de exploração de paralelismo *fine-grained* é motivado pelos processadores multicore. Comparativamente com os tradicionais *symmetric multi-processors* (SMPs), existem duas diferenças principais que permitem o paralelismo de granularidade fina em processadores multicore. Primeiro, há a possibilidade de se comunicar diretamente entre os núcleos, sem ter que "sair" do chip. Isto pode fornecer uma largura de banda (*bandwidth*) de comunicação muito mais elevada. Segundo, existe a possibilidade de se explorar os mecanismos especializados de hardware, de modo a diminuir o overhead de comunicação entre os núcleos (*inter-core communication)*. Nesta seção, damos uma visão geral dos mecanismos de comunicação, bem como as técnicas de compilação propostas para explorá-los.

3.2.1 Scalar Operand Network

Um exemplo notável de mecanismo de comunicação entre núcleos do mesmo chip (*on-chip inter-core)* é o ***Scalar Operand Network*** presente no processador RAW [8]. Este mecanismo consiste de malhas interligadas entre os núcleos. Associado a cada núcleo, há um co-processador de roteamento. Para o software, o ***Scalar Operand Network*** pode ser abstraído como um conjunto de filas em hardware para comunicação escalar *inter-core*. Cada núcleo se comunica com os outros através do seu roteador. Os acessos para as filas de comunicação são codificados como registros especializados de leitura e gravação. Por esta razão, as filas do hardware RAW são chamadas de filas mapeada em registro (*register-mapped queues*).

3.2.2 Synchronization Array

Outro mecanismo de hardware para implementar a comunicação escalar *inter-core* é a matriz de sincronização (*synchronization array*) [9]. Nesse mecanismo, o conjunto de instruções do processador (ISA) é estendido com duas novas instruções, produzir (***produce***) e consumir (***consume***), que, respectivamente, enviam e recebem um valor escalar por uma fila específica do hardware. Em [9], este mecanismo de hardware foi utilizado para apoiar a paralelização de laços que percorrem (atravessam) estruturas de dados recursiva ou ligadas. Estes laços foram divididos em dois segmentos, um de execução do percurso da estrutura de dados, e o outro de execução do corpo do laço. A idéia básica era esconder a latência de acesso à memória, que pode ser bastante cara no percurso de estruturas de dados ligadas devido à falta de localidade. *Speedups* razoáveis para uma seleção de benchmarks com uso intensivo de ponteiros foram obtidos em [9]. A chave para seu sucesso foi que, criando um fluxo unidirecional de dependências entre as *threads*, foi alcançada .uma execução realmente dissociada (*truly decopled execution*). Este esquema de particionamento de *threads*, que produz *pipelined multi-threading*, foi denominado ***Decoupled Software Pipelining*** (DSWP).

3.2.3 Memória Compartilhada

Várias alternativas para os mecanismos de ***Synchronization Array*** foram investigadas com o objetivo de reduzir os custos de hardware [10]. Observou-se, por exemplo, que as filas de comunicação podem ser implementadas através de **memória compartilhada**, por uma pequena penalidade no desempenho. Isto não só reduz os custos de hardware, mas também facilita a virtualização. Para obter um bom desempenho através de memória compartilhada, no entanto, duas alterações de hardware ainda são necessários. Primeiro, as instruções, bloqueantes ***produce*** e ***consume*** são necessários para evitar os custos de implementação de sincronização em software. Em segundo lugar, é necessário alterar o protocolo de coerência de cache para realizar o encaminhamento das linhas de cache (*forwarding of cache lines*) usadas para implementar as filas. Ter uma linha de cache transmitida do núcleo produtor, quando esta ficar cheia, elimina-se os *stalls* para acessar as filas no núcleo consumidor .

3.3 Paralelização Semi-automática

A maioria das abordagens discutidas até aqui possuem limitações em sua aplicabilidade, seja pela natureza das aplicações reais, sobre como elas foram construídas ou, pela forte presença de dependências. Uma alternativa é a técnica de paralelização semi-automática.

A paralelização de código é dita ser semi-automática quando o compilador recebe marcações em trechos de código que indicam a possibilidade de paralelização daquele trecho. Em geral, a paralelização por meio de anotações é aplicada a domínios específicos, inibindo algumas oportunidades alcançadas pela paralelização manual. Em contrapartida, exige um menor esforço por parte do desenvolvedor por apenas explicitar o paralelismo. Com isso, o programa está menos suscetível a erros – uma vez que o processo de paralelização em si é automatizado. Outra vantagem desse tipo de abordagem é a escalabilidade do código gerado. Hoje, para laços mais simples, já existem pré-processadores que automaticamente anotam o código para o compilador [11].

Um exemplo de padrão que pode ser classificado nesse grupo é o OpenMP [12]. Esse padrão foi projetado para prover, de maneira simples, o paralelismo no nível de *threads*. Nele, são disponibilizadas além de bibliotecas de funções em Fortran, C e C++, um conjunto de diretivas que indicam ao compilador quais blocos de código devem ser paralelizados e como os dados devem ser compartilhados entre as *threads*. Mas a paralelização semi-automática não surgiu com o OpenMP. Ela já vem sendo utilizada pelo High Performance Fortran (HPF) [13], por exemplo, com sua diretiva HPF$ INDEPENDENT, que marca laços que não possuem dependências de dados entre iterações e, portanto, podem ser executados em paralelo.

3.4 Programação Funcional

Outra novidade é o uso de linguagens de programação funcional [28]. Peyton Jones[1] nos apresenta quatro argumentos que apoiam esta afirmação. Primeiro, linguagens funcionais não necessitam de construções explícitas na linguagem para representar paralelismo, sincronização ou comunicação entre as tarefas (*threads*). O paralelismo está implícito no algoritmo que o programador escolhe, e quaisquer cálculos independentes podem ser executados simultaneamente, se os recursos estão disponíveis; a implementação da linguagem pode manipular automaticamente a sincronização e comunicação entre as *threads*. Segundo, não são necessários mecanismos especiais para proteger os dados que são compartilhados entre threads concorrentes. Qualquer *thread* que faz referência a uma sub-expressão comum pode proceder sua avaliação, e qualquer outra thread pode usar o resultado avaliado. Em terceiro lugar, o mesmo raciocínio formal que se aplica a programas funcionais seqüenciais também se aplica aos programas paralelos. Avaliação paralela é uma característica da implementação, mas a semântica da linguagem é definida em um nível alto de abstração e se aplica a ambas as implementações paralelas e seqüenciais. Finalmente, os resultados de um programa funcional são deterministicos. Qualquer implementação correta de uma linguagem funcional deve preservar a transparência referencial, o que significa que o próprio programa deve fornecer um mapeamento consistente entre entradas e saídas. Em particular, nenhum fator externo ou dependente de implementação, tais como o agendamento de *threads* individuais, podem influenciar esse mapeamento.

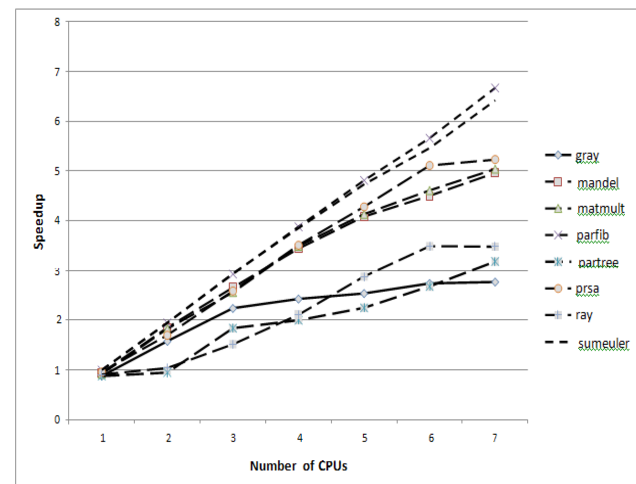
Apesar desses argumentos, tem-se revelado difícil perceber esse potencial na prática. Em geral, o paralelismo obtido a partir da transparência referencial em linguagens funcionais puras é de granularidade muito fina, não apresentando bom desempenho. Compiladores que exploram paralelismo implícito têm enfrentado dificuldades para promover o balanceamento de carga entre os processadores e manter os custos de comunicação baixo.

Embora um compilador possa extrair paralelismo de forma automatica de um programa funcional, há alguns indícios que sugerem que um controle explícito por parte do programador faz-se necessário. Novamente, a forma de se obter um desempenho mais confiável em arquiteturas paralelas é incluir "anotações" no programa para indicar quando a computação paralela pode ser benéfica. Algumas experiências com Haskell em um ambiente de programação real, o Glasgow Haskell Compiler (GHC) tem-se mostrado bastante promissoras.

[1] Peyton Jones é pesquisador da Microsoft Research em Cambridge, Inglaterra e também professor honorário do Departamento de Ciência da Computação na Universidade de Glasgow, onde foi professor durante os anos de 1990-1998.

Pelo menos em teoria, GHC tem estado em vantagem na corrida para encontrar uma maneira eficaz de explorar o TLP. Linguagem puramente funcional por definição, Haskell traz uma riqueza de paralelismo inerente no seu código. O modelo de programação semi-explícito adotado em Haskell tem se mostrado extremamente eficaz [29]. A semântica do programa permanece completamente deterministica, e o programador não é obrigado a identificar threads, bem como o uso de mecanismos de comunicação ou sincronização. Eles simplesmente anotam as sub-computações que podem ser avaliadas em paralelo, deixando a escolha de realmente fazê-lo para o sistema de execução. As *threads* então são criadas pelo *runtime* de forma dinâmica, e sua granularidade e tamanho variam muito.

A figura abaixo resume alguns benchmarks realizados por Marlow e outros em [30]. Os valores de referência consistem de uma seleção de programas pequenos e médios. No artigo, pode-se ver os resultados de cada programa do benchmark após as melhorias com o uso de “anotações”, em relação ao desempenho da versão seqüencial. Por "seqüencial" significa que a versão *single-threaded* do sistema de execução foi utilizado, na qual não há *overhead* de sincronização.



4. CONCLUSÃO

Nós vimos que a exploração do paralelismo no nível de threads não é um problema trivial. Isto se deve em parte à falta de abstrações para expressar o paralelismo no nível das linguagens de programação; à falta de modelos de programação paralela de fácil uso; às limitações impostas aos compiladores na paralelização automática (a presença de dependências em laços é o principal limitante na extração automática de *threads)* e muitas outras limitações práticas presentes na arquitetura como *threading overhead*, *destructive cache interference* (entre as *threads*) e o dimensionamento de recursos como a largura de banda de memória. Recentemente, tem havido um grande corpo de trabalhos abordando estas questões, discutidas a seguir:

Software: Tem havido um crescente impulso no desenvolvimento de novas linguagens de programação para capturar de forma eficiente o paralelismo inerente no início do ciclo de desenvolvimento de software. Exemplos incluem as linguagens X10 da IBM[15], Fortress da Sun[16] e Chapel da Cray[17]. Por outro lado, novos modelos de programação, tais como o OpenMP[18] e PGAS[19] estão sendo desenvolvidos ou estendidos para facilitar a programação paralela. Além disso,

estruturas de dados paralelas[20] e bibliotecas[21], estão sendo desenvolvidos para ajudar o desenvolvimento de software baseados em componentes, chave para alcançar alta produtividade.

Hardware: Nós vimos a importância do suporte da arquitetura para a exploração do TLP de granularidade fina (e.g., *synchronization array*). Adicionalmente, especulação no nível de *threads* (TLS)[2] e memória transacional (TM)[22] têm sido propostos como um meio para extrair concorrência otimista de regiões de código potencialmente paralelas.

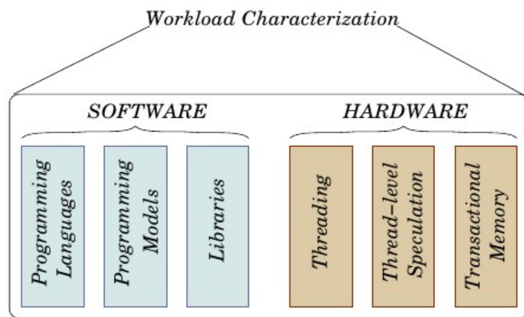


Figure 1. The role of workload characterization

A figura acima mostra que esta caracterização da carga de trabalho (workload) desempenha um papel fundamental na orientação da pesquisa e desenvolvimento de software e hardware para a exploração do TLP. Isso decorre do fato de que a introdução de qualquer nova idéia no software ou no hardware é altamente dependente da sua aplicabilidade para workloads existentes e emergentes.

De fato, com base nesta caracterização da carga de trabalho, tem havido uma ênfase crescente nos projetos de linguagens baseadas em PGAs como o *X10* e *Chapel* citados anteriormente, além de novas linguagens de programação de domínio específico, como para a computação científica (e.g. MATLAB), e versões paralelas do SQL para aplicações de banco de dados.

Nós vimos também que a utilização de programas puramente funcionais, como Haskell, são alternativas viáveis para a exploração do TLP em arquiteturas multicore. De fato, nos últimos anos Haskell (GHC, em particular) ganhou suporte expressivo com melhorias significativas no desempenho do seu sistema de execução. No entanto, isto apenas endereça metade do problema. A outra metade depende de uma maior sintonia dos programas para uma execução eficaz em paralelo. O programador ainda precisa exercer um certo controle sobre a granularidade das tarefas, a dependência de dados, a especulação e, até certo ponto na ordem de avaliação. A obtenção errada destas informações pode ser desastrosa para o desempenho. Por exemplo, a granularidade não deve ser nem muito fina nem muito grossa. Muito grossa e o sistema de execução não será capaz de fazer efetivamente um balanceamento de carga para manter todas os *cores* constantemente ocupados; muito fina e os custos de criação e programação das *threads* superam os benefícios de executá-las em paralelo. Os métodos atuais para estes ajustes dependem ainda em grande parte da tentativa e erro, da experiência, e um olho para a compreensão das estatísticas produzidas ao final da execução de um programa.

[2] *Speculative Multithreading* (SpMT), também conhecida como *thread level speculation* (TLS), é uma técnica de paralelização dinâmica que depende de execução fora-de-ordem para conseguir aumento na velocidade de execução em multi-cores. É uma espécie de execução especulativa, que ocorre no nível de threads, em oposição ao nível de instrução.

5. AGRADECIMENTOS

Nosso agradecimento ao Prof. Paulo Centoducatte por nos incentivar neste trabalho de pesquisa do estado da arte, ampliando a nossa visão de escopo do curso.

6. REFERENCIAS

[1] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August. Revisiting the sequential programming model for multi-core. In Proceedings of the 40th Annual ACM/IEEE International Symposium on Microarchitecture, pages 69–81, December 2007.

[2] Hennessy, J. L. and Patterson, D. A. 2007. Computer Architecture: A Quantitative Approach..Morgan Kaufmann Publishers, Inc. San Mateo, CA. Forth edition, 1995.

[3] G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva. Deadlock detection in MPI programs. Concurrency and Computation: Practice and Experience, 14(11):911–932, 2002.

[4] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems, 15(4):391–411, 1997.

[5] R. Allen and K. Kennedy. Optimizing compilers for modern architectures: A dependence-based approach. Morgan Kaufmann Publishers Inc., 2002.

[6] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures, pages 99–108, 2002.

[7] J. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The superthreaded processor architecture. IEEE Transactions on Computers, 48(9):881–902, 1999.

[8] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar operand networks. IEEE Transactions on Parallel and Distributed Systems, 16(2):145–162, February, 2005.

[9] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pages 177–188, September 2004.

[10] R. Rangan, N. Vachharajani, A. Stoler, G. Ottoni, D. I. August, and G. Z. N. Cai. Support for high-frequency streaming in CMPs. In Proceedings of the 39th International Symposium on Microarchitecture, pages 259–269, December 2006.

[11] M. Ishihara, H. Honda, and M. Sato. Development and implementation of an interactive parallelization assistance tool for openmp: ipat/omp. IEICE – Trans. Inf. Syst., E89-D(2):399–407, 2006.

[12] Parallel programming in OpenMP, Chandra, Rohit, San. : Francisco, Calif Morgan Kaufmann; London: Harcourt, 2000.

[13] D. B. Loveman. High performance fortran. IEEE Parallel Distrib. Technol., 1(1):25–42, 1993.

[14] Hedqvist, P. A parallel and multithreaded ERLANG Implementation. Master's dissertation, Computer Science Department, Uppsala University, Uppsala, Sweden, June 1998.

[15] The X10 Programming Language. http://x10-lang.org/

[16] Project Fortress Overview. http://research.sun.com/projects/plrg/Fortress/overview.html

[17] Chapel. http://chapel.cray.com/

[18] The OpenMP Application Program Interface. http://openmp.org/wp/about-openmp/

[19] PGAS – Partitioned Global Address Space. http://www.pgas.org

[20] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In Proceedings of the SIGPLAN'07 Conference on Programming Language Design and Implementation, pages 211–222, San Diego, CA, 2007.

[21] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 48–57, New York, NY, 2006.

[22] M. Herlihy and E. Moss. Transactional memory: Architectural support for lock-free data structures. In Proceedings of the 20th International Symposium on Computer Architecture, pages 289–300, San Diego, CA, May 1993.

[23] "Chip Multithreading and Multiprocessing", Erik Fischer, Technical Ambassador, Sun Microsystems, Hungary, December 14 2005, acessado em 18 de Junho de 2011, em http://developers.sun.com/solaris/articles/cmtmp.html

[24] Ungerer, T., Robic, B., Silic, J. A Survey of Processors with Explicit Multithreading, *ACM Computing Surveys*, 35, 1 (march 2003), 29-63.

[25] Olukotun, Oyekunle & Kunle, Hammond, L., Laudon, J.; Chip multiprocessor architecture: techniques to improve throughput and latency, Morgan & Claypool Publishers, 2007

[26] McGhan, Harlan. Niagara 2 Opens the Floodgates: Niagara 2 Design is Closest Thing Yet to a True Server on a Chip, acessado em 15 de junho de 2011, http://bnrg.eecs.berkeley.edu/~randy/Courses/CS294.F07/SunNiagara2.pdf

[27] Jinghe, Zhang. Chip Multi-threading and Chip Multi-threading and Sun's Niagara-series, acessado em 15 de junho de 2011, http://www.cs.wm.edu/~kemper/cs654/slides/niagara.pdf

[28] James Larus, D. Gannon, "Multicore computing and scientific discovery", The Fourth Paradigm: Data-Intensive Scientific Discovery. Part 3: Scientific Infrastructure, Microsoft Research, 2009.

[29] H.W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Pe na, S. Priebe, Á J. Rebón, and P. W. Trinder. Comparing parallel functional languages: Programming and performance. Higher Order Symbol. Comput., 16(3):203–251, 2003.

[30] S. Marlow, S. Peyton Jones and S. Singh, "Runtime Support for Multicore Haskell", ICFP'09, Proceedings of the 14th ACM SIGPLAN international conference on Functional programming, Edinburgh, Scotland, UK, 2009.