Paralelização Automática de Aplicações

Cristianno Martins
Instituto de Computação
Caixa Postal 6.176
Campinas, Brasil
cristianno.martins@students.ic.unicamp.br

RESUMO

Nos últimos anos, a tecnologia presente nos processadores tem evoluído a passos largos. Como resultado da busca por um maior desempenho, processadores *multicore* surgem trazendo a computação paralela do nicho científico para as demais aplicações computacionais.

A paralelização de aplicações, contudo, não é uma tarefa trivial. Com o objetivo de minimizar o impacto do uso de paralelismo sobre os modelos de programação, várias abordagens para paralelização automática de código foram desenvolvidas. Algumas dessas abordagens são discutidas neste trabalho, bem como suas limitações.

Categorias e Descritores de Assunto

H.4 [**Aplicação de Sistemas de Informação**]: Variado; D [**Computação Paralela**]: Aplicações—*Paralelização automática*

Termos Gerais

Auto-paralelização

Palavras-chave

Níveis de paralelismo, Abordagens, Limitações

1. INTRODUÇÃO

O interesse em resolver problemas cada vez mais complexos tem levado à necessidade de computadores cada vez mais poderosos. Entretanto, limitações econômicas e físicas sempre impediram que supercomputadores fossem aplicados a todos os problemas. Para contornar essas limitações, aplicações que demandavam muito processamento normalmente eram divididas em pedaços – chamados tarefas – que poderiam executar em paralelo. Essas tarefas, então, eram executadas simultaneamente em máquinas distintas, ou em máquinas com múltiplas unidades de processamento. A essa abordagem, deu-se o nome de processamento em paralelo.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

Mas, ao contrário do que possa parecer, a busca por paralelismo não teve seu início com o uso de várias unidades processantes trabalhando cooperativamente. Muito antes, quando os microprocessadores começaram a ser desenvolvidos, o paralelismo já era buscado como forma de aumentar a velocidade das máquinas: o paralelismo no nível de bits (discutido na Seção 4.1), que nada mais é que o aumento do tamanho do barramento de dados do processador, pode ser visto como pioneiro no campo da computação paralela.

Apesar de paralelismo em níveis mais baixos, tais como o de bits ou de instruções (também discutido na Seção 4.1), alcançarem ganhos consideráveis em desempenho, as oportunidades presentes em níveis mais altos não foram completamente ignoradas. Elas têm sido exploradas há décadas, mesmo sendo as soluções encontradas para simplificar suas aplicações bastante rudimentares [2]. Isso se deve ao fato de que a paralelização firmou seu nicho em aplicações científicas, apresentando-se timidamente em poucas aplicações com outros propósitos (em Sistemas Operacionais de propósito geral, por exemplo).

Nos últimos anos, o crescimento constante da complexidade dos circuitos e a redução no tamanho dos transistores e na distância entre eles, assim como o crescimento (até então elevado) na velocidade do processador, estabilizaram. Buscando ainda ganhos de desempenho, uma nova técnica de aperfeiçoamento começou a ser aplicada: múltiplos núcleos de processamento por chip. A idéia é simples: ao invés de criar processadores mais velozes e instáveis por conta do aumento da temperatura e da inexperiência com transistores menores, núcleos mais simples e confiáveis são encapsulados numa mesma pastilha de silício e o limite de desempenho passa a ser proporcional ao limite de paralelismo das aplicações sendo executadas. Por conta disso, pesquisadores têm desenvolvido várias técnicas para automaticamente paralelizar aplicações [10] [15] [12] [28] [30]. Algumas delas serão discutidas nas próximas seções.

As seções seguintes estão organizadas como segue: na Seção 2, os níveis de paralelização alcançáveis pelo desenvolvedor são discutidos; na Seção 3, rapidamente são comentadas formas de paralelização semi-automática (que necessita do auxílio do desenvolvedor); na Seção 4, formas de paralelização automática são examinadas; paralelização híbrida (combinando abordagens discutidas em seções anteriores) são comentadas na Seção 5; e, finalmente, as conclusões se encontram na Seção 6.

2. PARALELIZAÇÃO MANUAL

A princípio, toda a responsabilidade sobre a administração

de recursos (divisão de tarefas, escalonamento, comunicação, etc.) encontrava-se nas mãos de programadores. E a paralelização manual de aplicações não é um problema trivial: requer considerável parcela de tempo para o projeto e equipe especializada para o desenvolvimento [11]. Não apenas isso: muitas vezes a versão paralela da aplicação não é portável ou, mesmo, escalável.

Para amenizar o problema da portabilidade, padrões para interfaces de bibliotecas foram desenvolvidas, tais como MPI (*Message Passing Interface*) [24] e POSIX Threads [7]. No entanto, mesmo nos casos em que uma ou outra biblioteca é aplicável, a escalabilidade e o verdadeiro ganho de desempenho ainda dependem do projeto de paralelização.

A despeito de tudo isso, paralelizar manualmente uma aplicação é freqüentemente a única maneira de paralelizá-la completa e eficientemente. Isso porque, na paralelização manual, o desenvolvedor pode extrair informações específicas do programa, às quais o compilador pode não ter acesso, ou simplesmente não as aproveita em suas transformações.

Os níveis alcançados ao se paralelizar manualmente uma aplicação são discutidos a seguir.

Paralelização no nível de arrays

A maioria das aplicações comportam-se de maneira regular no acesso aos dados; e não é raro o uso de uma mesma operação sobre um conjunto de dados, com resultados independentes entre si. Essa operação poderia, teoricamente, ser aplicada a todos os dados de uma só vez, se houvesse recursos para tal. Assim, todos os resultados para o conjunto seriam obtidos como se a operação não ocorresse de forma escalar (um único dado sendo processado por vez). À aplicação desse conceito dá-se o nome de vetorização.

Na prática, não existem infinitos registradores em processadores vetoriais; tampouco o tamanho desses registradores é ilimitado. Nesses processadores, os registradores utilizados para operações vetoriais são maiores que o dado com o qual se deseja trabalhar. A vetorização, então, é aplicada a subconjuntos do conjunto de dados, de tamanho igual (ou menor) ao tamanho do registrador de vetorização. Assim, se o registrador tiver tamanho quatro vezes maior que os demais registradores, os subconjuntos do conjunto de dados têm tamanho quatro e a operação pode ser realizada sobre quatro dados do conjunto de uma só vez.

A paralelização no nível de *arrays* é o mais baixo nível que o programador pode ter acesso diretamente. Por conseqüência, trata-se da paralelização com maiores possibilidades de ganho no desempenho, porém também a mais complexa de ser implementada. Ademais, nem sempre os dados encontram-se organizados de forma a facilitar a visualização de oportunidades de vetorização.

Como exemplo, seja uma aplicação qualquer que trabalha com n pontos, e cada ponto possui d dimensões. Assuma que esse conjunto de pontos encontra-se armazenado numa estrutura linear na memória. Seja também um trecho de código que opera sobre apenas uma dimensão dos pontos, somando a ela uma constante c. Se a estrutura, inicialmente não projetada para suportar operações em paralelo, está organizada como um *array* de pontos, a vetorização não será aplicável para este problema. Por outro lado, se as dimensões dos pontos são alocadas seqüencialmente, com um único ponto formado por *arrays* de dimensões, é possível vetorizar este trecho de código.

Paralelização no nível de processos

Num nível mais alto, existe o paralelismo provido por processos. Aqui, o paralelismo se dá pela execução de mais que um processo para realizar uma determinada tarefa. Esse é o nível provido por bibliotecas que suportam os padrões MPI ou PVM (*Parallel Virtual Machine*) [27], que simplificam o gerenciamento do ambiente e a comunicação entre os processos.

Utilizando processos, é possível aplicar basicamente paralelismo espacial ou temporal de dados ou paralelismo de tarefas, dependendo da natureza do programa e de projeto de sua paralelização. Paralelismo espacial de dados, segundo [4], consiste na divisão de um grande conjunto de dados em partes menores, sendo cada uma dessas partes independentes entre si quando operadas. Por outro lado, paralelismo temporal de dados, diz respeito à divisão tanto dos dados quanto da própria operação em partes menores; cada processo é responsável pela execução de uma parte da operação e, ao realizar a operação sobre cada subconjunto de dados, um processo repassa seus resultados para o processo que executará a sub-operação seguinte no curso serial do programa. No paralelismo de tarefas, os dados e as operações são completamente independentes e, portanto, podem ser executadas em paralelo.

O paralelismo no nível de processos é um dos mais utilizados, principalmente para ambientes distribuídos, onde bibliotecas de comunicação auxiliam na abstração da localidade dos processos. Mesmo assim, esse nível de paralelismo pode ser aplicado a ambientes multiprocessados. No entanto, em ambos os casos, é desejável que a comunicação entre os processos seja minimizada ou sobreposta à computação, pois toda a sincronização realizada não existiria na versão serial do programa e, portanto, representa um custo adicional à aplicação.

Paralelização no nível de threads

Threads são os núcleos operacionais de um processo nos sistemas operacionais modernos. Várias threads podem coexistir em um mesmo processo, sendo que recursos como bibliotecas e o código do programa em execução são compartilhados por todas as threads de um processo. Assim, um processo provê às suas threads um ambiente básico de execução, enquanto cada thread executa sua tarefa.

Em ambientes multiprocessados, o uso de threads é favorável por simplificar a comunicação e o compartilhamento de dados. Primeiro, porque as threads já compartilham um ambiente de execução comum, logo sua criação, sincronização e gerenciamento são mais simples quando comparadas a processos; segundo, porque os dados armazenados na memória estão acessíveis por qualquer parte o processo, bastando uma thread possuir o endereço dos dados de outra para acessá-los.

Assim como no nível de processos, esse nível de paralelismo também possui padrões para bibliotecas, como o padrão POSIX Threads, mas requer maior atenção ao ser utilizado em implementações. Isso porque, se o compartilhamento de recursos por threads não for devidamente projetado, dados podem ser facilmente corrompidos por acessos simultâneos, ou mesmo errôneos, a regiões de memória. Paralelismo de dados e de tarefas também são aplicáveis neste nível.

Paralelização baseada em modelos

Não mais tratando de níveis de paralelismo, algumas aplicações possuem comportamento parecido, ou mesmo adaptável, a alguns modelos de problemas já resolvidos. A aplicação desses modelos tende a poupar tempo de projeto, por definirem representações para determinadas classes de problemas. É importante notar que tanto o paralelismo no nível de threads quanto no nível de processos, ou ambos, podem ser aplicados a esses modelos. Algumas das classes de problemas mais conhecidas são discutidas a seguir.

- **Divisão e Conquista** → consiste na redução da tarefa principal em tarefas menores, que possam ser resolvidas diretamente, e na conseqüente combinação dos resultados de cada uma das partes para alcançar o resultado global. Os algoritmos de ordenação clássicos MergeSort e QuickSort são implementados seguindo esse modelo.
- **Map e Reduce** → aplicação de uma mesma operação sobre dados na memória. A diferença principal entre *map* e *reduce* está no fato de que a primeira aplica a operação sobre cada dado do conjunto individualmente, resultando em um novo conjunto de dados; o segundo, por outro lado, aplica a operação cumulativamente em todos os dados do conjunto, resultando em um só dado. Assim sendo, a vetorização pode ser vista como uma aplicação de *map* sobre dados contíguos na memória. *Maps* são intrinsecamente paralelizáveis, desde que a operação sobre um elemento do conjunto independa de outros elementos do mesmo conjunto, enquanto *reduces* só podem ser paralelizados quando a operação a ser aplicada é, ao menos, associativa – se for também comutativa, pode-se alcançar granularidades muito finas.
- **Produtor e Consumidor** → uma ou mais unidades processantes são responsáveis por suprir outras unidades com dados. Geralmente a produção é mais veloz que o consumo, o que torna necessária a presença de *bufferes* para as saídas dos produtores.
- **Pipeline** → caso especial do Produtor e Consumidor, é exatamente a aplicação do paralelismo de dados temporal. Aqui, unidades processantes são conectadas em série e a saída de cada uma é a entrada da próxima, sendo a saída da última, o resultado global.

3. PARALELIZAÇÃO SEMI-AUTOMÁTICA DE CÓDIGO

A paralelização de aplicações é dita ser semi-automatizada quando o compilador recebe marcações em trechos de código que indicam a possibilidade de paralelização daquele trecho. Pelo geral, a paralelização por meio de anotações pode ser 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, porém não encará-lo – e 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. Apesar de o programa paralelo ser gerado estaticamente, a quantidade de unidades processantes pode ser determinada dinamicamente na maioria dos compiladores ou bibliotecas com suporte a anotações. Ainda, para laços mais simples, já existem pré-processadores que automaticamente anotam o código para o compilador [16].

Um exemplo de padrão que pode ser classificado nesse grupo é o OpenMP [19]. Esse padrão foi projetado para prover, de maneira simples, o paralelismo no nível de threads e, portanto, deve ser utilizado para plataformas multiprocessadas. Nele, são disponibilizadas diretivas que indicam ao compilador quais blocos de código devem ser paralelizados e como os dados devem ser compartilhados entre as threads, além de uma biblioteca com funções para configuração e coleta de informações sobre o ambiente de execução. Mas paralelização semi-automática de código não surgiu com OpenMP: ela já era usada em *High Performance Fortran* (HPF) [18], por exemplo, com seus comandos FORALL e HPF_INDEPENDENT, que marcavam laços que não possuíssem dependências de dados entre iterações e, portanto, pudessem executar em paralelo. Basicamente, HPF aplica-se a paralelismo de dados.

4. PARALELIZAÇÃO AUTOMÁTICA

A paralelização automatizada de código surgiu como uma tentativa de simplificar o processo de paralelização. Para tal, a automatização objetivava alcançar um maior paralelismo, com o mínimo de esforço por parte do desenvolvedor. No entanto, as dependências existentes nos programas – sejam de dados ou controle – restringem oportunidades de paralelismo [8]. Como apontado nas próximas subseções, técnicas de paralelização automática geralmente podem ser aplicadas a um subconjunto limitado de problemas.

Em alguns casos, porém, a paralelização automática provoca pouco ou, mesmo, nenhum impacto sobre os modelos de programação utilizados. Esse é o caso ideal, quando o mesmo programa que pode executar numa plataforma serial, pode tirar vantagem de plataformas paralelas. Algumas dessas técnicas também focam na melhor utilização do hardware disponível, evitando que unidades funcionais permaneçam ociosas, por exemplo.

Nas próximas seções são discutidas técnicas que vão desde auto-paralelização de baixo nível até o uso de especulação.

4.1 Paralelizações em baixo nível

Basicamente, no nível mais baixo, a paralelização é realizada diretamente no hardware. Nenhuma mudança no código da aplicação é exigida, nem no modelo de programação utilizado, porém a aplicação deve ser compilada para a arquitetura-alvo para aproveitar-se dos recursos presentes naquela arquitetura. O desenvolvedor da aplicação, na maioria das vezes, não alcança esse nível diretamente, ficando a cargo do compilador ou do próprio hardware a paralelização da aplicação.

As principais formas de paralelismo em baixo nível são discutidas a seguir.

Paralelismo no nível de bits

A forma mais básica de paralelismo – e tão discreta que muitas vezes é ignorada – é o paralelismo de bits. Esse tipo de paralelismo é alcançado pelo aumento do tamanho do barramento de dados do processador. Como principal vantagem, o aumento da largura do barramento permite o endereçamento de uma maior quantidade de memória e a

utilização de uma menor quantidade de ciclos de processador para executar operações mais complexas.

Os primeiros microprocessadores fabricados pela Intel, modelo 4004, possuíam barramentos com largura de apenas 4 bits. Logo processadores de 8 bits chegaram ao mercado, devido às muitas limitações do modelo anterior. Esses novos processadores poderiam endereçar até 640 KB de memória.

Algum tempo depois, processadores de 16, 32 e 64 bits foram desenvolvidos, mas o aumento no tamanho do barramento possui outras implicações (não só positivas) sobre o sistema. Como prova disso, antes do lançamento dos processadores de 16 bits, a Intel fabricava processadores do modelo 8088, que eram 8 bits apenas externamente, mas os barramentos internos eram de 16 bits. Além disso, apesar de haver algum ganho no desempenho provido pelo paralelismo de bits, as larguras dos barramentos mantêm-se estáveis por longos períodos de tempo. Como motivos para isso, tem-se que o aumento físico na largura do barramento acarreta um maior gasto de energia e ocupa maior espaço. Além disso, todo o hardware ligado a algum barramento deverá suportar o barramento de tamanho maior, o que implica em hardware mais complexo, ocupando mais espaço, e em uma maior geração de calor, por conseqüência.

Além disso, uma aplicação apenas utiliza todo o potencial de processadores com um barramento maior se necessita de uma grande quantidade de memória para ser executada ou se realiza instruções muito complexas constantemente durante a execução.

Paralelismo no nível de instruções

Nos primeiros microprocessadores, apenas uma instrução era executada por vez. As instruções mais complexas, que gastavam vários ciclos para serem executadas, deveriam ser concluídas para que outras instruções pudessem começar a executar. Se somente instruções simples (que sejam completadas em apenas um ciclo) fizessem parte do programa, sua execução não teria muitos limitantes: num ciclo, uma instrução finalizaria sua execução e, no próximo ciclo, outra instrução começaria a executar (Figura 1). Entretanto, se um conjunto de instruções complexas deve ser executado, cada instrução só poderá começa sua execução após a finalização da instrução anterior.

Para alcançar o paralelismo no nível de instruções, a execução de uma instrução é dividida em estágios (Figura 1). Esses estágios aparecem em seqüência, sendo que a saída de um estágio é a entrada para o próximo (denominado *pipeline*), e cada estágio realiza uma tarefa para a execução da instrução. No caso ideal, a cada ciclo há uma instrução sendo carregada, uma instrução sendo executada e uma instrução sendo finalizada no *pipeline*. E, mesmo para instruções complexas, há um ganho com o paralelismo: ainda que a instrução complexa C_1 tenha iniciado sua execução, a instrução seguinte, C_2, já estará carregada e pronta para a execução no mesmo ciclo em que houver a finalização da instrução C_1.

No entanto, existem certas limitações para aplicações desse nível de paralelismo. A presença de instruções de desvio, por exemplo, impedem que a próxima instrução seja executada; na verdade, não se sabe qual será a próxima instrução até que a condição do desvio tenha sido avaliada. Além disso, instruções que tenham sido carregadas, mas dependem do resultado da instrução imediatamente anterior deve aguardar a finalização da execução daquela instrução para iniciar sua própria execução. Variações em implementações do paralelismo no nível de instruções, tais como especulação de instruções (discutida nas próximas seções) e adiantamento de dados, podem resolver os problemas citados acima, com a introdução de certo grau de complexidade ao hardware.

Execução fora de ordem

No estágio de execução do pipeline, encontram-se as unidades funcionais (UF) do processador. Geralmente, cada uma dessas unidades possui um propósito específico: operações com inteiros, operações com decimais, *loads*/*stores*, etc. Teoricamente, uma vez que uma instrução esteja carregada no processador, os dados para sua execução disponíveis e a UF necessária livre, essa instrução poderia ser executada. Entretanto, mesmo que todas essas condições sejam respeitadas, o processador discutido até aqui pode somente executar as instruções na ordem em que elas aparecem no programa. Se a condição de ordem for dispensada, instruções prontas podem ser alocadas às UF ociosas, maximizando o paralelismo no nível de instruções. Esse é o princípio básico da execução de instruções fora de ordem.

Tanto o despacho quanto o término de uma instrução podem se dar fora de ordem. Desse modo, as arquiteturas que implementam execução fora de ordem podem ser classificadas em um dos grupos listados à seguir.

- Despacho em ordem e término em ordem → uma instrução só disputa por UF quando todas as instruções anteriormente despachadas já foram executadas. Se houver algum conflito por uma UF ou alguma delas exige mais de um ciclo para gerar o resultado, o despacho é suspenso;

- Despacho em ordem e término fora de ordem → o despacho é feito, mesmo que instruções anteriores ainda não tenham finalizado sua execução. O despacho ainda pode ser suspenso caso haja conflito por uma UF ou haja uma dependência verdadeira de dados;

- Despacho fora de ordem e término fora de ordem → utiliza uma janela de despacho para selecionar as instruções que possam ser despachadas, ainda que existam dependências verdadeiras ou conflitos de recurso em instruções anteriores.

Para que seja possível o despacho fora de ordem, um estágio pode ser adicionado após o estágio de decodificação do pipeline. Nesse novo estágio, os operandos das instruções são aguardados, e as instruções só são despachadas quando seus operandos estiverem disponíveis.

Especulação de instruções

Uma das instruções mais presente nos códigos de máquina é o desvio. Desvios são úteis principalmente para reuso de código, por exemplo no caso de laços, mas também para ditar o fluxo de instruções que deve ser executado. Desvios incondicionais não são vistos como um problema: eles apenas mudam o valor do registrador PC (*Program Counter*) durante a execução. O desvio condicional, no entanto, impede que instruções sejam carregadas até que sua condição tenha sido avaliada, e o desvio deva ser tomado ou não.

A especulação de instruções pretende tirar proveito do *pipeline* presente no processador, ainda que desvios condicionais sejam carregados. Para isso, várias abordagens podem ser utilizadas, tais como:

- Assume-se que o desvio possui sempre o mesmo comportamento, seja tomado ou não-tomado;
- É utilizada uma tabela de bits que marcam os últimos comportamentos de um dado desvio para, baseando-se nessa tabela, especular o próximo comportamento;
- É utilizada uma máquina de estados que pode mudar a cada acerto ou erro do previsor.

Caso a especulação tenha sido incorreta, os dados devem retornar a um estado consistente. Na prática, isso é feito através do uso de um circuito que realiza o re-ordenamento das instruções ao finalizá-las. As modificações provenientes de instruções de um desvio incorretamente especulado são simplesmente descartadas.

Processador superescalar

Unidades funcionais são os recursos mais importantes presentes no pipeline. Dado que a complexidade das UF é diretamente proporcional à complexidade da operação realizada, operações diferentes podem gastar diferentes quantidades de ciclos para executar. Se muitas instruções complexas (que utilizem a mesma UF) estão presentes no programa sendo executado, a concorrência pela mesma UF irá limitar bastante o paralelismo alcançado pelo *pipeline*.

Para superar essa limitação, muitos processadores possuem mais de uma UF para cada operação. Todavia, constantemente algumas UF estarão ociosas durante a execução. Com o objetivo de minimizar o tempo ocioso dessas UF, processadores superescalares podem carregar mais de uma instrução por ciclo.

Assim, processadores superescalares podem ser vistos como uma evolução dos processadores vetoriais, que, por sua vez, provêem melhorias aos processadores escalares. Esses processadores mais simples, chamados escalares, executam uma instrução com um ou dois operandos por ciclo; os vetoriais, executam uma mesma instrução sobre um conjunto de dados, em apenas um ciclo; nos superescalares, cada instrução opera sobre um ou dois operandos, porém, graças à presença de UF redundantes, várias instruções independentes podem executar no mesmo ciclo.

No entanto, o paralelismo provido por processadores superescalares é limitado pelo grau de paralelismo intrínseco no conjunto de instruções do programa e na complexidade e, portanto, no tempo gasto pelo estágio de despacho na checagem de dependências entre as instruções.

Processadores VLIW (*Very Long Instruction Word*)

Mesmo podendo executar mais de uma instrução por ciclo, processadores superescalares apenas podem carregar uma instrução por vez para o processador. Isso significa que o uso desses processadores só será vantajoso a partir do momento em que várias instruções já foram carregadas e seus operandos tornaram-se disponíveis para execução num mesmo ciclo – se duas ou mais instruções aguardam o resultado de uma mesma instrução como um de seus operandos, por exemplo. Enquanto essa condição não ocorre, o processador se comporta como um processador escalar com execução fora de ordem.

Ademais, processadores superescalares devem detectar dinamicamente as dependências entre as instruções. Isso exige a presença de circuitos para verificar as dependências, além de esses circuitos consumirem ciclos para realizar essa verificação. Dessa forma, processadores VLIW vêm como uma alternativa à execução de várias instruções por ciclo.

Processadores VLIW carregam um conjunto de instruções (palavra), ao invés de apenas uma instrução por ciclo. Também, esses processadores dispensam a presença de circuitos para verificação de independência entre as instruções, pois toda a verificação e escalonamento de instruções são realizados estaticamente pelo compilador. Geralmente, existe um conjunto de restrições sobre a forma das instruções, tais como:

- O tamanho da palavra é fixo, mas pode variar entre processadores. Processadores com palavras de tamanho pequeno – duas instruções, por exemplo – são denominados apenas LIW ($LongInstructionWord$);
- As instruções de uma palavra são obrigatoriamente independentes entre si, podendo todo o conjunto executar em paralelo. Isso garante que, a cada ciclo, uma palavra seja carregada e cada UF receba uma instrução dessa palavra;
- A ordem em que as instruções devem aparecer na palavra é predeterminada, dependente do processador e de quantas UF ele possui.

Essas restrições permitem que o hardware necessário para o processador VLIW seja simplificado, diminuindo o tempo necessário para a execução das instruções. Primeiro porque, com palavras de tamanho fixo, o carregamento das instruções sempre é realizado da mesma forma. Segundo, a independência entre as instruções numa palavra garantida pelo compilador evita a necessidade de verificação de dependências entre instruções. Terceiro, a ordem das instruções na palavra é diretamente mapeável nas UF, o que permite enviá-las diretamente à UF correspondente.

Processadores VLIW, no entanto, possuem desempenho fortemente dependente da qualidade do código gerado pelo compilador. Mesmo que a complexidade do hardware diminua, fator este que contribui para o aumento de desempenho na execução, a complexidade dos compiladores cresce em proporções exponenciais. Além disso, nem sempre dependências entre instruções podem ser analisadas estaticamente; a presença de apontadores no código, por exemplo, pode causar a presença de dependências irregulares no código [13] [5], somente detectadas no momento da execução do programa.

4.2 Paralelização automática básica

Não se pode deixar de mencionar laços e arrays quando se trata de auto-paralelização de código [23]; o que não é surpresa, visto que as instruções contidas em laços são, na maioria das aplicações, responsáveis por um grande consumo de tempo de processamento e, freqüentemente, elementos em arrays são processados de maneira regular (uma operação sobre todo o conjunto de dados, com resultados independentes entre si).

Tendo em vista esse fato, laços e acessos a arrays podem ser paralelizados, se for garantida a independência entre suas instruções. Essa abordagem se assemelha àquela utilizada em HPF, discutida na Seção 3. Para garantir a independência, essencialmente são seguidos três passos:

- É realizada uma análise sobre o programa para detectar as dependências entre as instruções;

- Trechos de código são selecionados para a paralelização;

- Transformações são aplicadas sobre os trechos selecionados no código original.

Auto-vetorização e paralelização de laços aninhados são o principal foco desse tipo de análise, mas sua aplicação não é prática. Primeiramente, a análise de dependências não é um problema trivial; como mencionado anteriormente, a presença de indireções no programa torna essa análise complexa e diminui sua precisão. Ademais, dependências entre instruções são comuns em programas reais; sendo necessária a ausência de dependências no laço (seja em instruções presentes na mesma iteração, ou de iterações diferentes) ou nos dados gerados a partir dos elementos do array, as aplicações dessa técnica são bastante limitadas.

4.3 Paralelização baseada em modelos teóricos

Para melhorar a precisão da auto-paralelização, algumas técnicas baseiam-se em modelos teóricos, fundamentados sobre os passos discutidos na Seção 4.2. Esses modelos, em sua maioria, especificam uma sub-classe de problemas aos quais a técnica pode ser aplicada, com garantia de resultados próximos ao ótimo. Eles são ditos teóricos porque modelam os programas em objetos matemáticos e realizam transformações sobre esses objetos – não mais diretamente sobre o programa. Duas dessas abordagens são discutidas abaixo.

Modelo Poliédrico

O modelo poliédrico é um modelo matemático para otimização de laços internos. Essa abordagem opera sobre os laços na forma de um sistema de equações de recorrência (ER). Os valores que essas equações assumem podem ser inferidos estaticamente pelo compilador, sem ter que executar o programa. Os laços são, então, encapsulados em objetos matemáticos chamados poliedros, que podem sofrer transformações e, após isso, são traduzidos novamente para laços equivalentes aos iniciais, porém com maior paralelismo explicitado. Gráficos que expressam o comportamento dos laços podem ser vistos na Figura 2.

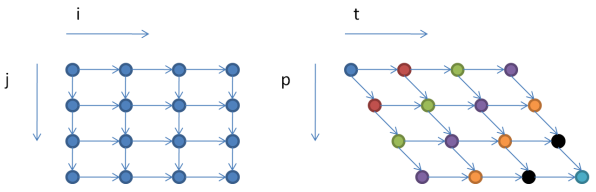

Figure 1: Dependências entre os elementos com escalonamento entre os laços. À esquerda, o laço original; à direita, o resultado da aplicação de modelos poliédricos, com possibilidade de execução paralela de elementos com mesma coloração.

São duas as principais vantagens em se utilizar modelos poliédricos: a técnica é tolerante à reutilização de memória, uma vez que é não existem dependências falsas nas ER; as ER auxiliam na detecção de reduções, que aparecem serializadas por limites impostos por linguagens de programação.

Apesar disso, modelos poliédricos sofrem de duas limitações. Primeiramente, há uma restrição própria do modelo sobre a classe de programas aos quais essa técnica pode ser aplicada: somente aos programas nos quais os passos discutidos na Seção 4.2 possam ser seguidos sistemática e completamente. Ademais, nem a análise, tampouco as transformações no programa, podem respeitar de forma satisfatória restrições externas, como a comunicação, por exemplo.

Modelos de transformações de matrizes

Muitas otimizações podem ser aplicadas ao programa antes da geração de código de máquina. Cada otimização realiza modificações no código, sendo que algumas delas possuem objetivos conflitantes.

A ordem em que as otimizações são aplicadas gera o resultado ótimo, dependendo da aplicação [29]. Isso porque, a depender da ordem das otimizações, pode-se obter vantagem da hierarquia de memória, graças à localidade dos dados, ou ela pode se tornar um gargalo. Alguns compiladores, por simplicidade, impõem uma ordem que deve ser seguida ao aplicar as otimizações disponíveis; outros, utilizam a técnica conhecida por *generate and test*, na qual muitos arranjos das otimizações são aplicados em busca da melhor ordem para aquele programa. Nota-se que a primeira abordagem simplesmente ignora o fato de que possa haver uma ordenação ótima para um dado programa; a segunda, por outro lado, resulta num processo bastante custoso, pois vai em busca da melhor ordem, verificando muitas possibilidades.

Uma técnica razoável para a escolha da melhor ordem das otimizações consiste em utilizar vetores de distâncias para representar dependências em laços internos, e vetores de direção para representar dependências entre dois ou mais laços internos ao mesmo laço. Dessa forma, duas matrizes representam todos os laços internos do programa, e as transformações podem ser aplicadas sobre essas matrizes – processo menos custoso que modificar diretamente o código do programa. Note que apenas os laços internos são considerados, uma vez que sua estrutura é mais simples – contém apenas instruções, sem desvios – e, quando o programa possui vários laços aninhados, as instruções dos laços mais internos executam uma maior quantidade de vezes que as dos laços mais externos.

Essa técnica pode ser aplicada objetivando maximizar o paralelismo ou a localidade dos dados, otimizar o programa para uma dada granularidade – seja nas tarefas ou nos dados, e é possível unificar algumas otimizações em apenas uma transformação nas matrizes, como é o caso das otimizações *loop interchange*, *reversal* e *skewing*, cuja transformação é discutida em [29].

Essa abordagem, contudo, é somente aplicável para laços com estruturas internas simples, o que limita sua aplicação a laços que não contenham outros laços.

4.4 Paralelização por escalonamento de tarefas

Uma outra abordagem para alcançar paralelismo é por reescalonamento de tarefas. O escalonamento de tarefas tem sido pesquisado há anos [3], pois, ao organizar as instruções de maneira diferente, a localidade de dados ou a independência entre as instruções pode ser aumentada. Pelo geral, o reescalonamento aproveita-se de características presentes no hardware (como hierarquias de memória, múltiplas execuções por ciclo ou *pipeline*) para aumentar a eficiência

do código. Três abordagens para reescalonamento são discutidas a seguir.

Loop unrolling

Loop unrolling é uma técnica que objetiva a redução do custo de instruções de controle em laços. Para isso, o corpo do laço é aumentado de forma que mais de uma iteração do laço original possa ser realizada em apenas uma iteração do novo laço. Durante o aumento do corpo do laço, deve-se manter a mesma semântica do programa original. Assim, laços podem ser desenrolados se o total de iterações é conhecido *a priori*, ou se uma cópia do laço original é mantida para possível ajuste no número de iterações.

A aplicação de *loop unrolling* dá ao compilador maiores oportunidades de otimização, visto que a quantidade de instruções no corpo do laço foi aumentada, além de minimizar a quantidade de instruções inúteis que ocupam UF e gastam ciclos – em verdade, as instruções necessárias para o controle dos laços não são úteis para alcançar o resultado do programa. Infelizmente, a aplicação dessa técnica tende a aumentar o tamanho do código, principalmente se todas as iterações do laço forem desenroladas (o que elimina a necessidade de controle do laço). Apesar disso, essa é uma técnica bastante simples, implementada pela maioria dos compiladores, e que serve de base para outras técnicas, como *software pipelining*, por exemplo.

Software pipelining

Software pipelining é outra técnica útil para expor paralelismo em estruturas de repetição. Sua abordagem assemelha-se à execução de instruções fora de ordem (discutida na Seção 4.1), exceto que é implementada pelo compilador, e não diretamente pelo hardware.

A aplicação de *software pipelining* consiste na reestruturação do laço de forma que todas as instruções internas sejam independentes entre si numa mesma iteração. Parte das instruções pode ser movida para blocos anterior e posterior ao laço. Como geralmente sua aplicação é combinada a *loop unrolling*, instruções de diferentes iterações podem partilhar o corpo do mesmo laço. O uso dessa técnica expõe maiores oportunidades para o paralelismo no nível de instruções (Seção 4.1), e é especialmente aplicável à compilação para processadores VLIW (Seção 4.1). Em contrapartida, como algumas instruções originalmente do laço passam a ser executadas em blocos fora do laço, deve-se exigir um número mínimo de iterações para que aquele laço seja executado. Esse limite mínimo é dependente do laço reestruturado e obriga a conservação de uma cópia do código original para laços nos quais não exista garantia daquele mínimo.

Paralelismo inter-blocos

Uma outra forma de escalonar tarefas, não mais focando em estruturas de repetição, é a busca de paralelismo entre blocos no programa. Blocos, segundo [14], podem ser conjuntos de instruções, objetos, processos, expressões, ou seja, qualquer unidade de processamento num programa.

Ao se definir essa unidade (geralmente dependente da linguagem de programação), uma análise das dependências é executada e suas informações são armazenadas como um DAG (*Directed Acyclic Graph*). A partir do DAG, os blocos são formados pela união de nós de maneira a aproveitar o paralelismo e minimizar a comunicação. Essa técnica, entretanto, é dependente de restrições externas, o que exige recompilação para plataformas com quantidade de processadores diferentes, ou com diferentes tempos de comunicação entre eles, por exemplo.

4.5 Paralelização especulativa

A presença de dependências é o principal limitante de desempenho nas aplicações; e a detecção dinâmica de dependências é um recurso caro, pois envolve aumento do hardware e requer ciclos para detectá-las. Por outro lado, algumas vezes as dependências detectadas estaticamente não existem realmente no momento da execução. Mais que isso: existem estruturas de laços as quais não deixam claro quantas iterações serão executadas, nem mesmo no momento da execução. Nesses laços é impossível aplicar qualquer desses conceitos vistos até então [9].

Especulação nada mais é que a execução de uma tarefa sem saber se seu resultado será útil, ou o uso de um dado sem tê-lo ainda calculado. Através da execução especulativa, é possível aproveitar recursos ociosos no sistema para pré-calcular resultados de instruções e somente utilizá-los se for necessário, bem como prever possíveis valores de um dado e utilizá-lo antes do fim de sua computação – especialmente útil para dados de fácil previsão, como variáveis de indução (contadores) em laços. Quando algo for erroneamente especulado, devem-se desfazer as tarefas realizadas baseadas naquela especulação para retornar a um estado consistente do programa.

Algumas técnicas que aplicam especulação são discutidas abaixo.

Especulação no nível de threads

Especulação no nível de threads (ENT) trata a computação de iterações de laços como tarefas, sem que suas dependências sejam consideradas [17] [22] [25]. Sua aplicação é semelhante ao uso de execução fora de ordem, com término em ordem (segundo Seção 4.1): para garantir que nenhuma violação de dependências ocorreu, os resultados são enfileirados, e as modificações aplicadas em ordem. Caso alguma violação tenha ocorrido, a iteração será executada novamente para os novos dados.

Também, para minimizar as violações por dependências entre iterações, ENT não inicia todas as threads no mesmo ciclo: há um atraso entre as threads da primeira e da segunda iterações, da segunda e da terceira, e assim por diante. Com isso, iterações anteriores podem gerar os dados necessários para as posteriores antes que as posteriores necessitem dos dados – no caso das dependências ocorrerem nas mesmas instruções do laço.

Entretanto, caso as dependências existentes no laço estejam em instruções diferentes, podem haver muitas iterações especuladas de forma errada. Além disso, esse tipo de especulação assume a presença de processadores homogêneos para minimizar as violações, não sendo aplicável no caso de processadores heterogêneos. E mais: o custo da comunicação envolvida na sincronização entre threads é alto; por isso existem estudos que mostram abordagens que minimizem esse custo [26].

Especulação no nível de trace

Fundamentalmente, um *trace* de execução é um caminho dentro do código que pode ser seguido até que o resultado esteja disponível. Logo, ele possui definidas as condições para todos os desvios executados [20].

Um *trace* pode ser dito *hot* se suas instruções são executadas na maioria das vezes [6]. Para a detecção de *hot traces*, uma abordagem baseada em *feedback* pode ser utilizada, tais como uso de *profiling* ou anotações do desenvolvedor.

Ao se detectar *hot traces*, uma entre duas abordagens pode ser seguida:

1. O *trace* mais executado do programa pode ser otimizado como um grande bloco sem desvios. Caso algum dos desvios devesse ser tomado, o código original seria executado a partir do primeiro desvio incorreto ou, por simplicidade, totalmente [21].

2. A cada desvio, seu *hot trace* pode ser tomado especulativamente.

A primeira abordagem permite a aplicação de muitas otimizações no bloco gerado, mas exige que a versão original do código esteja presente na aplicação. Além disso, se a checagem da violação de um desvio é feita somente no final da computação do *hot trace*, muitas instruções deverão ser descartadas.

A segunda abordagem, como aplicada em [1], recomenda a utilização de duas threads para a computação: uma especulativa e outra não. Essencialmente, a primeira thread executa os *hot traces* para cada desvio e a segunda, verifica se o desvio deveria ou não ser tomado. Seu principal limite é a sua dependência de *feedback*. Caso os dados utilizados durante o *profiling* ou o *trace* indicado pelo desenvolvedor não seja muito executado na prática, no melhor dos casos não haverá perda de desempenho com relação à versão sem especulação.

Execução de multi-caminhos

Mesmo o melhor esquema de especulação sofre com algumas predições incorretas. Cada uma delas, causa um rompimento no fluxo de execução, quando o processador precisa eliminar instruções erroneamente especuladas e começar a executar o caminho correto. Teoricamente, sendo possível executar especulativamente ambos os caminhos – a ser tomado e não-tomado, descartando-se os resultados obtidos a partir da especulação errada, a previsão alcançaria 100% de predição correta. Essa técnica é conhecida como execução de multi-caminhos (EM).

Muitas vantagens podem ser alcançadas a partir de uma implementação de EM em hardware. Porém, muitos recursos seriam necessários para manter os estados de todos os desvios que pudessem estar ao mesmo tempo sendo especulados. Ademais, o crescimento no número de desvios especulados tende a ser exponencial. Para diminuir esse crescimento, EM pode ser utilizada juntamente com alguma técnica de previsão de caminhos, como discutidas na Seção 4.1. Somente para os caminhos onde a previsão obtivesse baixas taxas de probabilidade de acerto (abaixo de 60%, por exemplo), EM seria aplicada.

5. TÁTICAS HÍBRIDAS

Combinações entre as abordagens vistas também são aplicáveis. Algumas das possíveis combinações são mencionadas abaixo.

- *Threaded Software Pipelining* → sendo que é garantida a independência entre as instruções numa mesma iteração, instruções diferentes no corpo do laço podem ser executadas por diferentes threads. Essencialmente, o laço é dividido entre as threads, e a menor granularidade do paralelismo é dada pela granularidade do laço: se o laço possui poucas instruções, dificilmente será possível utilizar muitas threads para paralelizá-lo;

- Execução *multi-threaded* de multi-caminhos → uma forma de organizar as execuções de múltiplos caminhos é utilizar threads. Uma thread, por si só, provê contextualização, o que é útil no caso da EM. Processadores SMT (*Simultaneous Multithreading*) são ambientes quase ideais: várias threads sendo executadas ao mesmo tempo, com instruções sendo carregadas para todas no mesmo ciclo. Se pequenas modificações no hardware forem aplicadas, como a criação de um barramento para cópia de registradores entre os contextos, o problema da comunicação também pode ser minimizado.

- paralelização baseada em *feedback* → uma combinação possível entre a paralelização automática e a manual, é o uso de *feedback* para o desenvolvedor. Esse tipo de abordagem é utilizado quando a paralelização automática não pode ser aplicada a um trecho do programa, e o compilador não pode realizar transformações no código sem a garantia de manter a semântica original. O compilador, então, aponta aquele trecho como passível de paralelização– por exemplo, o programa pode utilizar estruturas de dados às quais uma abordagem de paralelização não se aplique. Para que a paralelização automática seja possível, o desenvolvedor deve intervir e realizar modificações que permitam ao compilador paralelizar aquele trecho.

6. CONCLUSÃO

Se, por um lado, computadores pessoais passaram a ser paralelos e com maiores potencialidades, por outro, a indústria de software não está pronta para o desenvolvimento massivo de aplicações paralelas.

Mesmo através da combinação de técnicas, podem ser exigidas mudanças no hardware ou conhecimento do desenvolvedor sobre a exploração do paralelismo provido pela arquitetura.

Ainda, a maioria das abordagens discutidas possui limitações em suas aplicações: seja sobre a natureza da aplicação, sobre como a aplicação foi construída ou sobre a presença de dependências. Também, a possibilidade de encapsular núcleos heterogêneos num mesmo processador aumenta exponencialmente a complexidade de programação. Mesmo com todo o esse esforço, dificilmente a paralelização poderá ocorrer de forma genérica e satisfatória.

7. REFERENCES

[1] Trace-level speculative multithreaded architecture. In *ICCD '02: Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*, page 402, Washington, DC, USA, 2002. IEEE Computer Society.

[2] Keynote 4: Can parallel software catch up with parallel hardware? trends in automatic parallelization. In *ICITA '05: Proceedings of the Third International Conference on Information Technology and Applications (ICITA'05) Volume 2*, page 25, Washington, DC, USA, 2005. IEEE Computer Society.

[3] K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson. Adaptive scheduling with parallelism feedback. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 100–109, New York, NY, USA, 2006. ACM.

[4] T. Akenine-Moller and E. Haines. *Real-Time Rendering, ed. 2nd*. AK Peters Ltd, July 2002.

[5] M. Arenaz, J. Tourino, and R. Doallo. Compiler support for parallel code generation through kernel recognition. *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 79–, April 2004.

[6] B. J. Bradel and T. S. Abdelrahman. Automatic trace-based parallelization of java programs. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 26, Washington, DC, USA, 2007. IEEE Computer Society.

[7] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[8] M. Clialabine and C. Kessler. A survey of reasoning in parallelization. *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, 3:629–634, July 30 2007-Aug. 1 2007.

[9] J.-F. Collard. Space-time transformation of while-loops using speculative execution. *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 429–436, May 1994.

[10] B. Di Martino and C. Kessler. Two program comprehension tools for automatic parallelization. *Concurrency, IEEE [see also IEEE Parallel & Distributed Technology]*, 8(1):37–47, Jan-Mar 2000.

[11] M. Frumkin, M. Hribar, H. Jin, A. Waheed, and J. Yan. A comparison of automatic parallelization tools/compilers on the sgi origin 2000. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–22, Washington, DC, USA, 1998. IEEE Computer Society.

[12] M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. PhD thesis, University of Passau, Germany, 2004.

[13] M. Guo. Automatic parallelization and optimization for irregular scientific applications. *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 228–, April 2004.

[14] L. Guodong, Z. Defu, and H. Jie. An automatic compiler for extracting inter-block parallelism. *Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on*, pages 502–506, Aug. 2003.

[15] M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. *Int. J. Parallel Program.*, 28(6):537–562, 2000.

[16] 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.

[17] D. Llanos, D. Orden, and B. Palop. Meseta: a new scheduling strategy for speculative parallelization of randomized incremental algorithms. *Parallel Processing, 2005. ICPP 2005 Workshops. International Conference Workshops on*, pages 121–128, June 2005.

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

[19] T. G. Mattson. How good is openmp. *Sci. Program.*, 11(2):81–93, 2003.

[20] C. Molina, A. Gonzalez, and J. Tubella. Compiler analysis for trace-level speculative multithreaded architectures. In *INTERACT '05: Proceedings of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures*, pages 2–10, Washington, DC, USA, 2005. IEEE Computer Society.

[21] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity: An effective abstraction for reliable software speculation. *IEEE Micro*, 28(1):21–31, 2008.

[22] D. Orden and B. Palop. New scheduling strategies for randomized incremental algorithms in the context of speculative parallelization. *IEEE Trans. Comput.*, 56(6):839–852, 2007. Member-Diego R. Llanos.

[23] L. Ricci. Automatic loop parallelization: an abstract interpretation approach. *Parallel Computing in Electrical Engineering, 2002. PARELEC '02. Proceedings. International Conference on*, pages 112–118, 2002.

[24] J. M. Squyres. Definitions and fundamentals – the message passing interface (MPI). *ClusterWorld Magazine, MPI Mechanic Column*, 1(1):26–29, December 2003.

[25] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 2, Washington, DC, USA, 1998. IEEE Computer Society.

[26] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 65, Washington, DC, USA, 2002. IEEE Computer Society.

[27] V. S. Sunderam. PVM: A framework for parallel distributed computing. 2(4):315–339 (or 315–340??), Dec. 1990.

[28] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, 1991.

[29] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, 1991.

[30] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.