Trebuchet: Explorando *TLP* com Virtualização DataFlow

Tiago A. O. Alves, Leandro A. J. Marzulo, Felipe M. G. França
Universidade Federal do Rio de Janeiro
Programa de Engenharia de Sistemas e Computação, COPPE
Rio de Janeiro, RJ, Brasil
{tiagoaoa, lmarzulo, felipe}@cos.ufrj.br

Vítor Santos Costa
Universidade do Porto
Departamento de Ciência de Computadores
Porto, Portugal
vsc@dcc.fc.up.pt

Resumo

No modelo DataFlow *as instruções são executadas tão logo seus operandos de entrada estejam disponíveis, expondo, de forma natural, o paralelismo em nível de instrução (ILP). Por outro lado, a exploração de paralelismo em nível de thread (TLP) passa a ser também um fator de grande importância para o aumento de desempenho na execução de uma aplicação em máquinas* multicore*. Este trabalho propõe um modelo de execução de programas, baseado nas arquiteturas* DataFlow*, que transforma ILP em TLP. Esse modelo é demonstrado através da implementação de uma máquina virtual* multi-threaded*, a* Trebuchet*. A aplicação é compilada para o modelo* DataFlow *e suas instruções independentes (segundo o fluxo de dados) são executadas em Elementos de Processamento (EPs) distintos da* Trebuchet*. Cada EP é mapeado em uma* thread *na máquina hospedeira. O modelo permite a definição de blocos de instruções de diferentes granularidades, que terão disparo guiado pelo fluxo de dados e execução direta na máquina hospedeira, para diminuir os custos de interpretação. Como a sincronização é obtida pelo modelo* DataFlow*, não é necessária a introdução de locks ou barreiras nos programas a serem paralelizados. Um conjunto de três* benchmarks *reduzidos, compilados em oito* threads *e executados por um processador* quadcore *Intel® Core™i7 920, permitiu avaliar:* (i) *o funcionamento do modelo;* (ii) *a versatilidade na definição de instruções com diferentes granularidades (blocos);* (iii) *uma comparação com o* OpenMP*. Acelerações de 4,81, 2,4 e 4,03 foram atingidas em relação à versão sequencial, enquanto que acelerações de 1,11, 1,3 e 1,0 foram obtidas em relação ao* OpenMP*.*

1. Introdução

Extrair desempenho de processadores com um só núcleo vem se tornando uma tarefa cada vez mais complexa nos últimos tempos. Segundo Olukotun [11], a complexidade da lógica adicional necessária para encontrar instruções paralelas dinamicamente é, aproximadamente, proporcional ao quadrado do número de instruções que podem ser disparadas simultaneamente. Os mais recentes processadores de um único núcleo apresentam ganho de desempenho inferior e desproporcional à evolução da tecnologia de silício, além de consumirem cada vez mais energia.

Máquinas *multicore* se mostraram como uma alternativa para ganho de desempenho [12], com maior eficiência no uso de energia [7], em relação aos processadores de um só núcleo e, por isso, elas dominam o cenário atual na produção de processadores. Algumas classes de aplicação, como aplicações de bancos de dados, são beneficiadas imediatamente pelo aumento no número de núcleos. No entanto, muitas aplicações são programadas sequencialmente e é necessário reescrever o código para expor suficiente paralelismo.

Infelizmente, desenvolver aplicações *multi-thread* não é uma tarefa trivial. O programador precisa dividir seu programa em partes que possam ser executadas em *threads* distintas. Comunicação entre as *threads* requer o uso de *locks*, ou de técnicas como Memória Transacional [6], para garantir o correto acesso aos recursos compartilhados. Além disso, espera-se que a aplicação escale para processadores com mais núcleos. *OpenMP* [5] e MPI [1] são opções populares para programação paralela baseadas, respectivamente, nos modelos de memória compartilhada e de troca de mensagens. Um problema comum a ambas as aborda-

gens é a necessidade do uso de barreiras para realizar a sincronização entre as diversas *threads* em execução, o que pode deixar núcleos de processamento ociosos. Além disso, a execução guiada pelo fluxo de controle faz com que as *threads* fiquem ociosas, aguardando a produção de dados por outras. No entanto, podem existir operações em um ponto adiantado do código da *thread* ociosa, cujos dados de entrada (o conjunto de leitura) já estejam prontos, o que permitiria a sua imediata execução.

Balakrishnan e Sohi endereçam este problema com a criação do *Program Demultiplexing* [2], um paradigma de execução onde métodos ou funções são demultiplexados para que possam executar concorrentemente com o restante do programa. A execução dos métodos demultiplexados é disparada segundo o fluxo de dados, isto é, eles iniciam a execução assim que seus conjuntos de leitura estejam disponíveis, o que pode ocorrer antes de serem efetivamente chamados, segundo a ordem do programa.

A ideia de executar operações quandos os operandos estejam disponíveis, ao invés de seguir a ordem do programa, é o principio fundamental dos modelos *Dataflow*. O *ILP* é naturalmente exposto por este modelo e o mesmo pode ser convertido em *TLP*. Tendo em vista essas observações, apresentamos um modelo de execução de programas, baseado em arquiteturas *DataFlow*, que permite a extração de *TLP* de programas sequenciais para execução em sistemas com múltiplos núcleos de processamento. Este modelo é demonstrado através da implementação de uma máquina virtual *multi-threaded*, a *Trebuchet*. A idéia chave é compilar o programa alvo (sequencial) para a linguagem de máquina da *Trebuchet*, cujos Elementos de Processamento (EPs) estão mapeados em *threads* na máquina hospedeira (estilo Von Neumann). Com isso, o programa executado na máquina virtual é paralelizado de forma implícita.

O principal objetivo deste trabalho é fazer uma análise dos principais problemas encontrados e resolvidos na construção de tal sistema, e apresentar resultados comparativos com soluções e alternativas já existentes. O restante deste trabalho está organizado da seguinte forma: na Seção 2 é apresentada a *Trebuchet*; os experimentos e resultados são exibidos na Seção 3; na Seção 4 são descritos os trabalhos relacionados; na Seção 5 é feita a conclusão e a indicação de possíveis trabalhos futuros.

2. Trebuchet

A maioria dos processadores atuais segue o modelo Von Neumann, isto é, a execução é baseada no fluxo de controle. Mesmo que sejam utilizados mecanismos de execução fora-de-ordem baseados no fluxo de dados, como Tomasulo [14], a emissão das instruções é sequencial. O modelo *DataFlow* expõe paralelismo em nível de instrução (*ILP*) de forma natural. A máquina virtual *Trebuchet* tira proveito das vantagens do *DataFlow* para transformar *ILP* em *TLP*, de forma que aplicações sequenciais possam se beneficiar dos múltiplos núcleos de processamento disponíveis.

2.1. Conjunto de Instruções

Sendo a *Trebuchet* uma máquina virtual, é importante a existência de um formato de instrução para viabilizar a tarefa de interpretação. Além disso, o formato deve permitir a definição de instruções com diferentes números de operandos de entrada e de saída.

A Figura 1 mostra o formato de uma instrução da *Trebuchet*. Os primeiros 32 bits contêm o *Opcode*, bem como a quantidade de *Resultados* produzidos e de operandos de *Origem* (quantas portas de entrada a instrução possui). A informação no campo `Opcode` indicará a presença ou não do campo `Imediato` nos 32 bits seguintes. Cada porta de entrada de uma instrução será preenchida com um operando que pode ser proveniente de mais de uma instrução. Em tempo de execução, o operando será enviado apenas por uma das instruções, de acordo com o fluxo seguido no grafo. Sendo assim, para cada porta de entrada teremos um campo de 32 bits indicando quantas instruções podem produzir este operando e em seguida uma lista com linhas de 32 bits com as instruções produtoras (*ID* com 27 bits) e o número da porta de saída na instrução produtora (com 5 bits, como no campo `#Resultados`).

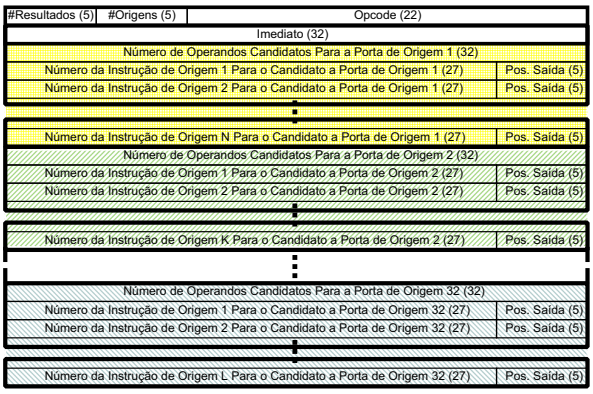

Figura 1. Formato de uma Instrução.

A *Trebuchet* implementa as principais instruções aritméticas e lógicas (`add`, `sub`, `and`, `or`, `mult`, `div`, além de suas variações com operando imediato). Vale notar que instruções que implementam desvios de controle, laços de repetição e chamadas de função são diferentes em máquinas *DataFlow*, merecendo um destaque nesta Seção. Como não há contador de programa em máquinas *DataFlow*, desvios de controle em um programa devem alterar o fluxo dos dados de acordo com o caminho selecionado.

Isto é feito através da instrução `steer` que recebe um operando O e um seletor Booleano S, fazendo com que O seja enviado para um de dois caminhos possíveis, dependendo do valor de S.

Na execução de laços de repetição, porções independentes de instruções contidas no laço podem executar mais rapidamente, abrindo a possibilidade de disparar a execução destas mesmas instruções para iterações seguintes. As outras porções mais lentas poderiam, portanto, receber operandos provenientes de iterações futuras. Para que operandos de iterações distintas não se misturem existem duas soluções: *(i)* o *DataFlow* estático, onde uma iteração de um laço só pode iniciar a sua execução quando a iteração anterior tiver terminado; *(ii)* o *DataFlow* dinâmico, onde cada instrução possui diversas instâncias, uma para cada iteração, e os operandos trocados entre as instruções são rotulados com o número da instância à qual são destinados. A *Trebuchet* adota o *DataFlow* dinâmico. Quando um operando passa de uma iteração para a seguinte, seu rótulo deve ser incrementado para que seja corretamente encaminhado à nova iteração. A instrução *Increment Iteration Tag* (`inctag`) é responsável por esta ação.

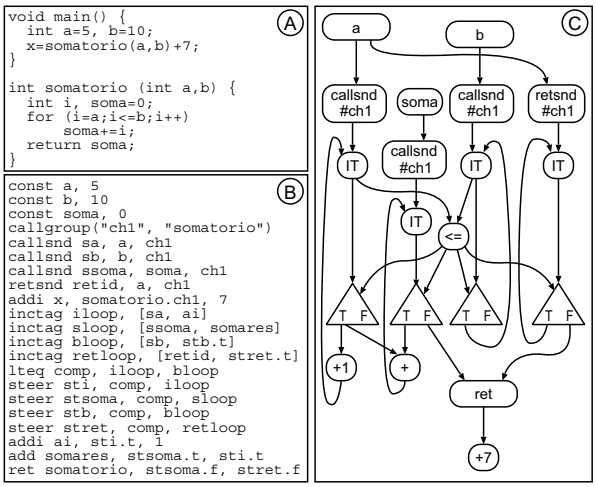

Figura 2. Exemplo de Assembly Trebuchet.

Em chamadas de função, operandos (parâmetros da chamada) podem ser enviados a partir de diferentes pontos de chamada no código (estáticos). Além disso, podem haver múltiplas chamadas a partir de um mesmo ponto, como em um laço ou recursão (dinâmicos). Para que operandos de chamadas diferentes, estáticas ou dinâmicas, não se misturem, são necessários mais dois itens no rótulo dos operandos: o grupo da chamada (`GrpCh`) e o número da instância da chamada (`#Chamada`). O primeiro é utilizado para diferenciar operandos de pontos de chamada estáticos diferentes para uma mesma função e é definido durante o processo de montagem, com o uso da macro `callgroup`. O segundo serve para diferenciar instâncias de um mesmo ponto de chamada, sendo atualizado dinamicamente, com base em um contador local (`ContCh`), pela instrução `callsnd`. O rótulo `#Chamada` da função chamadora é passado para a função chamada com a instrução `retsnd` e é usado para o retorno da função, que é feito com a instrução `ret`.

A Figura 2 exemplifica o uso das seguintes instruções: *(i)* `callsnd` e `ret` para chamada de função; *(ii)* `inctag` para manutenção dos rótulos em iterações de um laço; *(iii)* `steer` para implementar desvios de controle em *DataFlow*. Em A é exibido o código em alto nível que realiza o somatório dos números inteiros no intervalo de $[a, b]$, utilizando a função `somatório`. Em B é mostrado o código de montagem *DataFlow* para este programa e em C o grafo associado.

2.2. Usando a Trebuchet

Para paralelizar programas sequenciais usando a *Trebuchet* é necessário compilá-los para a sua linguagem de montagem *DataFlow* e executá-los. A *Trebuchet* está implementada na forma de uma máquina virtual. Cada um de seus Elementos de Processamento (EPs) está associado a uma *thread* na máquina hospedeira. Quando um programa é executado na *Trebuchet*, as instruções são alocadas aos EPs e executadas segundo o fluxo de dados. O modelo *DataFlow* permite que instruções independentes executem em paralelo. Se tais instruções estiverem associadas a EPs distintos, cada EP sendo uma *thread*, elas executarão paralelamente na máquina hospedeira, se esta contar com núcleos de processamento disponíveis.

Como os custos de interpretação de cada instrução podem ser altos, a execução de programas a partir da compilação completa para *DataFlow* pode apresentar um desempenho insatisfatório. Sendo assim, blocos de código Von Neumann podem ser transformados em instruções mais complexas ou *superinstruções*. Esses blocos são então compilados para a máquina hospedeira e a interpretação das superinstruções associadas se resume apenas a chamadas para execução direta dos blocos. Vale lembrar que a execução de cada superinstrução continua sendo disparada pela máquina virtual segundo o modelo *DataFlow*. Esta é uma das contibuições originais introduzidas pela *Trebuchet*: permitir a definição de blocos *Von Neumann* com diferentes granularidades (não apenas funções, como feito no *Program Demultiplexing*), além de não demandar suporte de *hardware* adicional para o seu funcionamento.

A Figura 3 mostra o fluxo de trabalho para a paralelização de um programa e sua execução na *Trebuchet*. Inicialmente são definidos os blocos que formarão as superinstruções através de uma transformação do código fonte. Ferramentas de *profiling* podem ajudar a determi-

nar quais porções do código são candidatas interessantes à paralelização. O código transformado é compilado na forma de uma biblioteca dinâmica. É preciso então definir o fluxo de dados entre os blocos e escrever o código de montagem *DataFlow* associado, que pode conter superinstruções ou instruções simples provenientes da compilação completa de parte do código para *DataFlow*. É feita então a transformação do código de montagem *DataFlow* em código alvo. A alocação das instruções aos EPs é determinada e o código alvo é carregado para execução. A execução das instruções simples envolve interpretação completa. No caso das superinstruções o que ocorre é um desvio para o trecho do código alvo ("binário") da máquina hospedeira (contido na biblioteca).

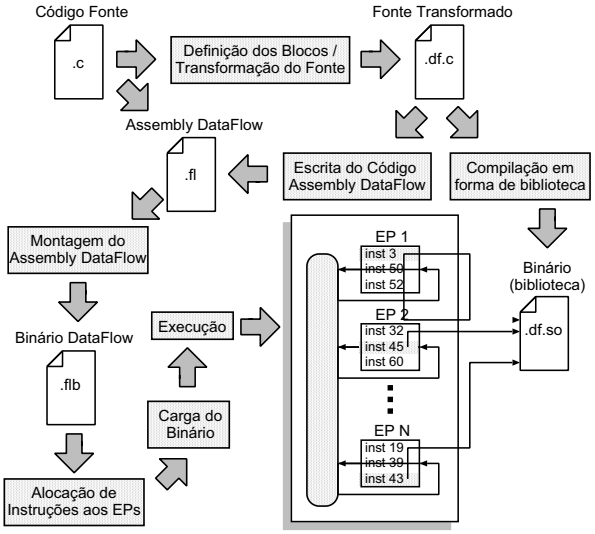

Figura 3. Fluxo de Trabalho.

Os processos de montagem do código *DataFlow* e carga do binário gerado são automatizados. Os processos de definição de superinstruções, tranformações de código para geração da biblioteca, geração do código de montagem *DataFlow* e alocação de instruções aos EPs, no presente momento são realizados de forma manual. A automatização destes processos faz parte de pesquisa em andamento.

A Figura 4 mostra um exemplo de paralelização de uma aplicação de integração numérica usando a *Trebuchet*. Em A é exibido o código fonte sequencial. O uso da macro `superinst` ilustrada no código de montagem *DataFlow* (exibido em D) permite a extensão do conjunto de instruções da *Trebuchet*, com a criação de superinstruções. No exemplo é criada a superinstrução `sP` que corresponde à execução de uma porção do *loop* descrito em A. Cada instância de `sP` é informada da porção do *loop* que irá executar através dos operandos `tID` e `nT`. Neste exemplo temos quatro instâncias (`nT=4`) e cada uma pode ser associada a um EP e, portanto, a uma *thread*. Note que em `super0` são definidos operandos de entrada que serão enviados por outras instruções no grafo *DataFlow*, além dos resultados produzidos. O código da biblioteca é compilado gerando o binário (C). O grafo *DataFlow* associado a D é exibido em E. Note que todas as instruções do exemplo são instruções simples e totalmente interpretadas na *Trebuchet*, com exceção de `sP`, cuja interpretação se resumirá a um desvio para a biblioteca (função `super0`).

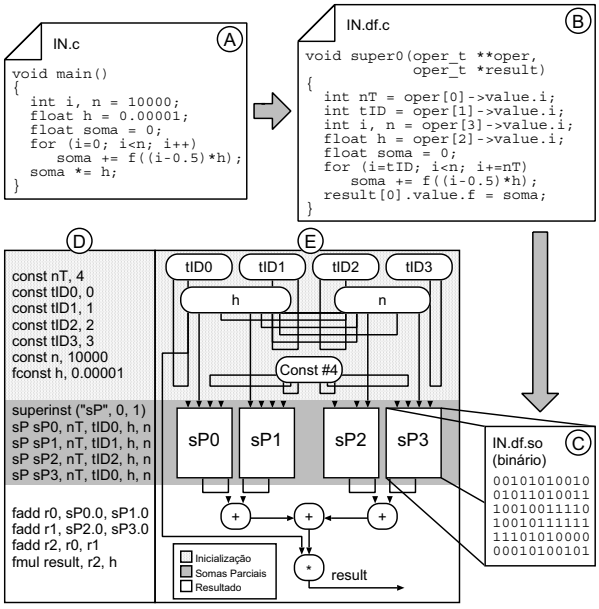

Figura 4. Exemplo de uso da Trebuchet.

2.3. Arquitetura da Trebuchet

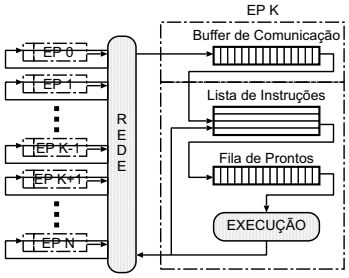

Figura 5. Arquitetura da Trebuchet.

Na Figura 5 é exibida a Arquitetura da *Trebuchet*. Ela é formada por um conjunto de Elementos de Processa-

mento (EPs) idênticos e conectados via uma rede de interconexão. Os EPs são responsáveis pela interpretação e execução de instruções, segundo as regras do modelo *DataFlow*. Eles possuem um *buffer* de comunicação visível aos outros EPs para o recebimento de mensagens com operandos. Uma lista de instruções armazena as informações de cada instrução estática mapeada no EP. Além disso, cada instrução possui uma lista de operandos pertencentes às suas instâncias dinâmicas. O casamento de operandos com mesmo rótulo, ou seja, pertencentes à mesma instância de uma instrução, é feito nesta lista. A instrução na qual ocorreu um casamento é enviada, junto com os operandos associados, para a fila de prontos onde aguardará a interpretação e execução.

O modelo proposto neste trabalho é baseado em troca de mensagens, sendo as rotinas `Envia()` e `Recebe()` responsáveis pela troca de operandos entre instruções alocadas a EPs diferentes. Como a *Trebuchet* é implementada como uma máquina virtual *multithread*, inicialmente a ser hospedada em máquinas *multicore*, a passagem de mensagens entre os EPs é realizada através de acessos a regiões de memória compartilhada. Neste caso, o *overhead* de comunicação é dado pelo custo dos *locks* para o acesso a essas regiões. Daí a importância da alocação de instruções independentes em EPs diferentes visando tirar proveito do paralelismo e, ao mesmo tempo, de tentar manter instruções relacionadas no mesmo EP, ou minimizar a distância entre elas na rede de interconexão.

A Figura 6 (a) detalha uma instrução da lista de instruções. Ela contém o `Opcode`, `Imediato`, e listas encadeadas para armazenar os operandos das até 32 portas de entrada. Os operandos são ordenados por rótulo em suas listas e o casamento é feito quando para um determinado rótulo exista um operando correspondente em cada lista. Os operandos produzidos com a execução da instrução são enviados para os destinos indicados na *Matriz de Destinos* onde cada linha especifica os N destinos de um operando. Cada elemento desta matriz contém o ponteiro para a instrução de destino e a `Posição` (número da porta de entrada) na instrução destino. Além disso, cada instrução possui também o número do EP ao qual a instrução em questão está mapeada. Vale lembrar que operandos trocados entre instruções localizadas no mesmo EP não passam pela rede de interconexão, sendo apenas encaminhadas para a instrução destino na lista de instruções. O campo `ContCh` é usado pela instrução `callsnd` para armazenar o valor do contador de chamadas, mencionado na Seção 2.1. Neste caso, o campo `Imediato` é usado para armazenar o `GrpCh`, explicado na mesma Seção.

Na Figura 6 (b) é mostrada uma mensagem com operando trocada entre EPs. Ela contém o ponteiro para a instrução à qual o operando se destina, a posição (porta) de entrada na instrução destino e o operando em si. Na Figura 6 (c) é exibido um operando, que possui o rótulo, composto pelo `#Chamada` e número da iteração, o seu valor e um ponteiro para o próximo operando (usado quando o operando está inserido em uma lista encadeada da instrução).

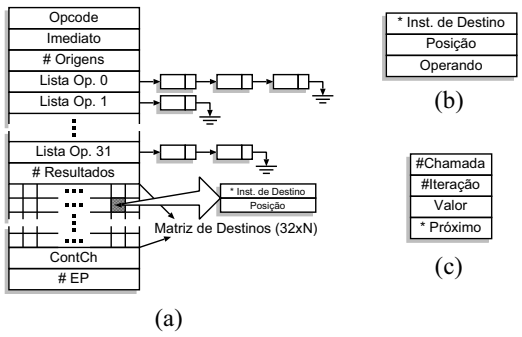

Figura 6. Estruturas da Trebuchet.

No modelo de Von Neumann, um programa termina quando o fluxo de controle associado ao mesmo chega ao final. No modelo *DataFlow* nao é tão simples identificar o fim da execução do programa por não se dispor de uma informação localizada, como o conteúdo de um contador de programa. Como cada unidade de processamento só tem visão do seu estado, ela não pode inferir que o programa chegou ao fim somente por se encontrar ociosa, pois uma nova carga de trabalho pode chegar por uma de suas arestas de entrada. Para solucionar este problema foi implementado um algoritmo de detecção de terminação global para a *Trebuchet*, baseado no algoritmo de *snapshots* distribuídos [4]. A terminação será detectada quando todos os EPs estiverem ociosos e não existirem mensagens com operandos em trânsito.

3. Experimentos e Resultados

Para avaliar o modelo de execução de programas *DataFlow* proposto e sua implementação na forma da máquina virtual *Trebuchet*, foram selecionados três *benchmarks* reduzidos (dadas as atuais limitações de compilação): *(i) ray tracing* de esferas; *(ii)* cálculo recursivo do determinante de matrizes; *(iii)* multiplicação de matrizes. A descrição de cada aplicação, do método de paralelização e dos experimentos realizados são exibidos nesta Seção.

A máquina hospedeira usada para a execução da *Trebuchet* nestes experimentos possui: *(i)* um processador Intel® Core™i7 920 (2,66 GHz), com quatro núcleos de processamento físicos, cada um com dois contadores de programa (*Hyper Threading*); *(ii)* 6GB de memória DDR 3 1600 *Triple Channel* (3x2GB); *(iii)* sistema operacional Linux Ubuntu 8.10 64bits (kernel 2.6.27-7).

Para todos os *benchmarks* foram elaboradas versões *DataFlow* e *OpenMP*, executadas entre 10 e 200000 vezes (dependendo do tamanho do *benchmark*) para minimizar impactos causados por aplicações do sistema que possam iniciar sua execução durante os experimentos. O *OpenMP* foi escolhido para comparação com a *Trebuchet* por sua popularidade como ferramenta para paralelizar programas para máquinas *multicore*. Uma das razões desta popularidade se deve à facilidade para paralelização de programas sequenciais através do uso de *Pragmas* marcando as regiões de código a serem paralelizadas. A execução é feita com 1, 2, 4 e 8 *threads*. Gráficos mostram a aceleração obtida pelas versões em *DataFlow* e *OpenMP* (DF e OMP) com relação à versão sequencial, além da aceleração obtida pela versão *DataFlow* em relação à *OpenMP*.

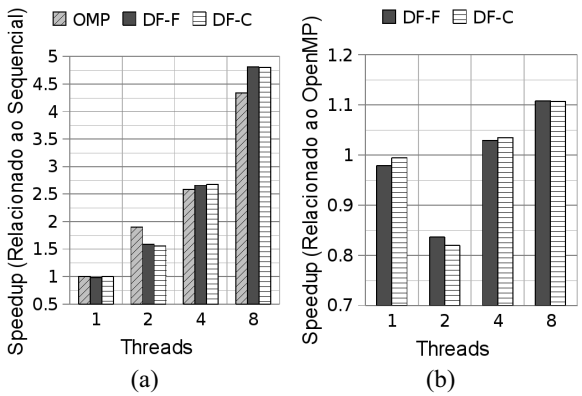

Figura 7. Resultados: Ray-Tracing.

A Figura 7 mostra os resultados para o ray tracing de esferas. Esta aplicação faz a renderização de uma cena com diversas esferas cujas informações de posição e cor são definidas em um arquivo. O resultado é uma imagem com resolução de 800x600 pixels salva em arquivo. A renderização é feita pixel a pixel, varrendo linhas e colunas em dois laços aninhados. Superinstruções são criadas para a computação da carga do arquivo e gravação da saída. A paralelização da porção de renderização é feita de duas maneiras: *(i) DataFlow* de granularidade fina (DF-F); *(ii) DataFlow* de granularidade grossa (DF-C). Na versão DF-F é feita a descrição em código de montagem *DataFlow* do laço externo e criação de uma superinstrução para o laço interno, tendo como objetivo de verificar a versatilidade da *Trebuchet* e o impacto no desempenho causado pela interpretação de instruções simples. Na versão DF-C a superinstrução engloba os dois laços aninhados. Embora custos com interpretação sejam amenizados, laços aninhados aumentam a quantidade de desvios de controle a serem executados na máquina hospedeira e o efeito das predições de desvio incorretas podem prejudicar o desempenho da aplicação.

Em ambas as soluções, instâncias distintas da instrução de renderização farão seu trabalho sobre grupos de colunas diferentes na imagem. Os resultados mostram acelerações de até 4,81 em relação à versão sequencial. Na comparação com o *OpenMP* uma aceleração de até 1,11 foi obtida. A comparação entre as versões *DataFlow* de granularidade fina e grossa mostra que os custos com interpretação não foram tão significativos, visto que pode ter ocorrido uma diminuição de custos com erros na predição de desvios.

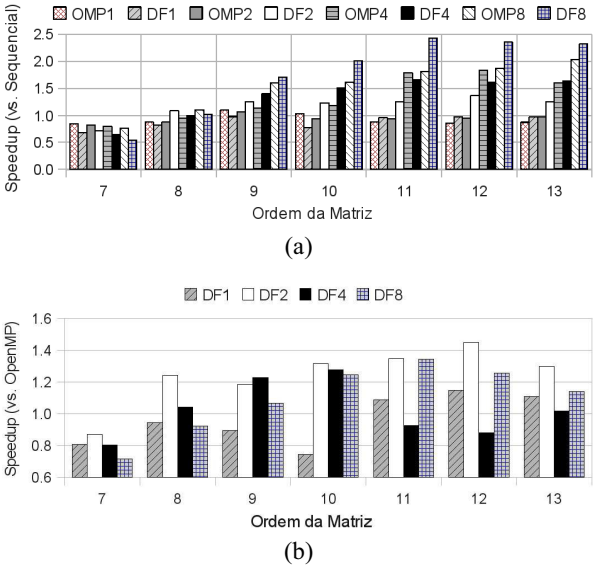

Figura 8. Resultados: Determinante.

A Figura 8 mostra os resultados para a aplicação de cálculo recursivo de determinante de matrizes. Os cenários de execução *DataFlow* e *OpenMP* são identificados pro DFx e OMPx, onde x representa o número de *threads*. Uma superinstrução é criada para a leitura da matriz de um arquivo e outra para a impressão do resultado, enquanto que o laço que executa o primeiro nível de recursão é transformado em outra superinstrução cujas instâncias dividirão a tarefa dos cálculos dos determinantes das submatrizes do nível dois em diante. Os experimentos foram realizados para diferentes tamanhos de matrizes (ordens 8, 9, 10, 11, 12 e 13) para permitir a avaliação da quantidade de computação necessária para compensar os custos de comunicação, tanto para a versão *DataFlow* quanto para a versão *OpenMP*. Os resultados mostram que a versão *DataFlow* foi superior à *OpenMP* para os cenários com maior quantidade de computação (matrizes de maior ordem). Com a diminuição da quantidade de computação, a versão *DataFlow* se mostra mais sensível e passa a apresentar uma desaceleração antes da versão *OpenMP*. A versão *Data-*

Flow apresenta acelerações de até 2,4, quando comparada à versão sequencial, e de até 1,3, em relação ao *OpenMP*.

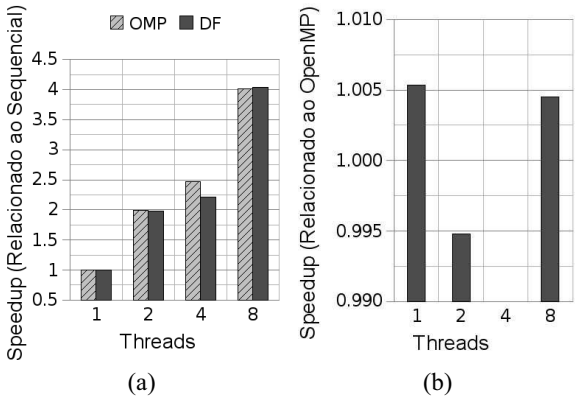

Figura 9. Resultados: Multiplicação Matricial.

A Figura 9 exibe os gráficos para o programa de multiplicação de matrizes. Neste caso o código de montagem *DataFlow* inclui a descrição de uma função na qual são executadas superinstruções que paralelizam o laço externo que varre as matrizes fazendo a multiplicação. Cada instância da superinstrução fica responsável pela produção de algumas linhas na matriz resultado. Os resultados mostram um desempenho equivalente ao *OpenMP*, com acelerações de até 4,03, em relação à versão sequencial.

4. Trabalhos Relacionados

Máquinas *Dataflow* nunca se tornaram um padrão, principalmente porque não suportavam a semântica de memória requerida pelas linguagens imperativas, exigindo uma nova arquitetura e linguagem de programação especialmente orientada para este modelo. A arquitetura *WaveScalar* [13] proposta por Swanson é a primeira arquitetura *Dataflow* a respeitar esta semântica, podendo, portanto, executar programas escritos em linguagens imperativas, como C e C++. A idéia chave é a separação entre o modelo de execução e a interface de memória que atende às requisições de leitura e escrita segundo a ordem do programa. O compilador numera as instruções de acesso à memória para que a interface possa garantir a semântica do programa.

A máquina *Trebuchet* pode ser considerada uma máquina *dataflow* híbrida, pois permite a compilação de blocos de instruções onde a execução dentro de cada bloco segue o modelo de Von Neumann e a execução entre blocos é guiada pelo fluxo de dados. A máquina *SDF* [8] adota o mesmo princípio. Já a arquitetura *TRIPS* [3] é uma máquina dataflow híbrida ortogonal à *Trebuchet*. A execução entre blocos segue o modelo de Von Neumann, sendo guiada pelo fluxo de dados dentro de cada bloco. A execução de programas no modelo *TRIPS* requer um *hardware DataFlow*.

O *Program Demultiplexing* [2] é um paradigma de execução onde métodos ou funções são demultiplexados para que possam executar concorrentemente com o restante do programa, segundo o fluxo de dados. O código fonte dos programas é modificado para permitir a execução de funções em outros núcleos de processamento antes do seu ponto de chamada. Assim os resultados são produzidos e recebidos pelo fluxo principal do programa para serem usados no momento da chamada. Os conjuntos de leitura também podem ser especulados para permitir o início da execução dos métodos com maior antecedência. A implementação requer mudanças no protocolo de coerência de cache, estruturas adicionais para armazenar resultados de execuções especulativas, bem como ferramentas para identificar e separar os métodos a serem demultiplexados.

A *Transactional WaveCache* [9] é um mecanismo que permite a execução especulativa e fora-de-ordem de operações de memória em máquinas *DataFlow* dinâmicas. O mecanismo é inspirado em conceitos de Memória Transacional, pois instruções pertencentes a uma mesma iteração de um laço são tratadas como uma única transação, aninhada com transações associadas a iterações anteriores. As transações armazenam em um *Undo Log* uma cópia de todas as posições de memória alteradas para que seja feito o *rollback*, mediante a detecção de um conflito com alguma transação anterior. O *commit* é feito segundo a ordem do programa e permite o descarte do *Undo Log*. A *Transactional WaveCache* foi implementada no *WaveScalar* e serve como base para o entendimento e criação de possíveis soluções para o problema de acesso à memória para blocos de instruções independentes, como discutido na Seção 5, a seguir.

5. Conclusões e Trabalhos Futuros

Neste trabalho foi apresentado um modelo de execução de programas, baseado em arquiteturas *Dataflow*, que transforma *ILP* em *TLP*. Esse modelo é demonstrado através da implementação de uma máquina virtual multi-threaded, a Trebuchet. Os experimentos mostram a versatilidade do modelo na criação de superinstruções, sua capacidade de acelerar aplicações e comparação com o *OpenMP*. Para as três aplicações compiladas e executadas por 8 *threads*, acelerações de 4,81, 2,4 e 4,03 foram atingidas em relação à versão sequencial, enquanto que acelerações de 1,11, 1,3 e 1,0 foram obtidas em relação ao *OpenMP*. Vale lembrar que para programas com múltiplas regiões paralelizaveis e independentes o uso do modelo proposto pode trazer mais benefícios, pois é possível executar blocos independentes sem a terminação dos anteriores, segundo o fluxo de controle.

A alocação de instruções aos EPs é um fator de extrema importância no desempenho. Instruções alocadas a EPs associados a processadores hospedeiros fisicamente próximos minimizam os custos de comunicação na troca de operandos. Por outro lado, alocar instruções de forma esparsa aumenta o paralelismo. A automatização deste procedimento é importante para tornar a *Trebuchet* mais transparente para o usuário, sendo objeto de estudos futuros. Embora já existam soluções para este problema [10], a flexibilidade da *Trebuchet*, por ser uma máquina virtual, permite o estudo de soluções diferentes e mais agressivas. Afinal, a rede de interconexão e a disposição dos EPs pode ser alterada dinamicamente. Além disso, é interessante que a disposição dos EPs na rede de interconexão virtual seja semelhante à organização dos núcleos na máquina hospedeira, usando *schedule affinity* para associar cada *thread* (que representa um EP) a um núcleo físico.

Máquinas *DataFlow* geralmente apresentam problemas de explosão de paralelismo, especialmente em laços. Porções do laço que executam mais rapidamente podem produzir uma série de operandos que não serão consumidos por um longo período de tempo, causando o uso excessivo das estruturas de memória (de armazenamento de operandos nos EPs) e uma possível degradação no desempenho. Embora seja possível limitar o paralelismo manualmente, com a inclusão de arestas no grafo *DataFlow*, uma solução automatizada, via compilação ou com mecanismos implementados na máquina virtual, se faz necessária e faz parte de pesquisa em andamento.

Os *benchmarks* utilizados neste trabalho permitiram a definição de superinstruções que não apresentavam perigos no acesso à memória. É o tipo de aplicação que se pode facilmente paralelizar com uma biblioteca para programação paralela, como o *OpenMP*. A execução de *benchmarks* clássicos se faz necessária para uma melhor avaliação do modelo. No entanto, em *benchmarks* mais complexos, podem ocorrer perigos de memória para superinstruções distintas. Neste caso, elas deveriam ser executadas sequencialmente (uma aresta no grafo *DataFlow* deve existir entre elas). Em alguns casos não é possível determinar a existência de perigos em tempo de compilação, o que obrigaria a criação de arestas entre pares de superinstruções onde não é possivel determinar se perigos existem ou não. Para evitar a adição destas arestas e a consequente perda de paralelismo, um mecanismo de execução especulativa pode ser útil. Além disso, é trabalho em andamento melhorar os custos com emulação da *Trebuchet*.

6. Agradecimentos

Agradecemos aos colegas Ana Luísa Duboc, Alexandre Sardinha, Lawrence Bandeira e Lúcio Paiva pelo auxílio na revisão do presente texto. Agradecemos também à CAPES e ao consórcio Euro-Brazilian Windows (EBW), pertentece ao Programa Erasmus Mundus External Window Cooperation pelo suporte finaceiro aos autores deste trabalho.

Referências

[1] Mpi: A message passing interface. In *Proc. Supercomputing '93*, pages 878–883, Nov. 15–19, 1993.

[2] S. Balakrishnan and G. S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 302–313, Washington, DC, USA, 2006. IEEE Computer Society.

[3] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and t. T. Team. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, 2004.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[5] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[6] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[7] L. Jian and J. F. Martinez. Power-performance implications of thread-level parallelism on chip multiprocessors. In *ISPASS '05: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, pages 124–134, Washington, DC, USA, 2005. IEEE Computer Society.

[8] K. Kavi, R. Giorgi, and J. Arul. Scheduled dataflow: Execution paradigm, architecture, and performance evaluation. *IEEE Transactions on Computers*, 50(8):834–846, 2001.

[9] L. A. Marzulo, F. M. Franca, and V. S. Costa. Transactional wavecache: Towards speculative and out-of-order dataflow execution of memory operations. *Computer Architecture and High Performance Computing, Symposium on*, 0:183–190, 2008.

[10] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers. Modeling instruction placement on a spatial architecture. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 158–169, New York, NY, USA, 2006. ACM.

[11] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[12] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *IEEE Computer*, pages 2–11, 1996.

[13] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 291–302, 2003.

[14] R. M. Tomasulo. An efficient algorithm for exploring multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, Jan 1967.