TRATAMENTO DE CÓDIGO SEQÜENCIAL NO MODELO DE FLUXO DE DADOS

Marcos C. Visoli[1]
Departamento de Ciência da Computação
UNICAMP
Caixa Postal 6065
13081-970 — Campinas — SP
(0192) 39-8442
mcv@dcc.unicamp.br

Arthur J. Catto[2]
Departamento de Ciência da Computação
UNICAMP
Caixa Postal 6065
13081-970 — Campinas — SP
(0192) 39-8442
catto@dcc.unicamp.br

RESUMO

Uma das questões ainda não resolvidas eficientemente em multiprocessamento é a determinação do tamanho ou granularidade ideal das tarefas ou processos, com o objetivo de obter desempenho aceitável.

A dificuldade decorre do fato de que a granularidade conflita com o volume de sincronizações a serem realizadas. Máquinas multiprocessadoras baseadas no modelo von Neumann exploram paralelismo ao nível de subprograma e comandos e, conseqüentemente, suportam granularidade grossa. Isso implica em um número menor de sincronizações, mas também numa inadequada exploração do paralelismo das aplicações.

Máquinas de fluxo de dados, por sua vez, exploram paralelismo ao nível de instrução, suportando, portanto, granularidade fina. Apesar de tratarem o paralelismo mais adequadamente do que as máquinas de von Neumann, as máquinas de fluxo de dados pecam pela necessidade de um grande número de sincronizações.

Este artigo trata da redução do número de sincronizações no modelo de fluxo de dados a partir do agrupamento de código seqüencial.

ABSTRACT

One of unanswered questions in multiprocessing is that of determining the ideal size or "granularity" of tasks or processes, aiming at an acceptable machine performance.

The difficulties here stem from the fact that granularity conflicts with synchronization requirements. Multiprocessors based on the von Neumann model exploit parallelism at procedure and command level, thus supporting coarse granularity. This requires fewer synchronizations, but does not fully exploit application parallelism.

Dataflow machines, on the other hand, exploit instruction-level parallelism, thus supporting fine granularity. Despite there better approach to parallelism, dataflow machines are penalized by the large number of required synchronizations.

This paper addresses the issue of reducing the number of required synchronizations in the dataflow model by grouping sequential pieces of code.

---

[1] Mestrando em Ciência da Computação.

[2] Professor Assistente Doutor. Ph.D. in Computer Science, University of Manchester, 1981.

1 INTRODUÇÃO

Arquiteturas paralelas surgiram basicamente do desejo de aumentar o desempenho dos computadores, de modo a permitir o tratamento de problemas cada vez maiores e um melhor aproveitamento dos grandes avanços da tecnologia de microeletrônica.

Ainda hoje, a grande maioria das arquiteturas baseia-se no modelo de computação seqüencial dirigido por controle [Tre82], proposto por von Neumann há mais de 40 anos. As arquiteturas paralelas convencionais, que utilizam múltiplos elementos de processamento baseados nesse modelo, apresentam deficiências em questões tais como determinação, escalonamento e sincronização de tarefas paralelas e tempo de latência dos acessos à memória.

Um outro modelo possível é o dirigido pelo fluxo de dados [Tre82]. Baseado na execução de instruções provocada pela disponibilidade dos dados, este modelo resolve com elegância os problemas encontrados no modelo anterior. Com elegância mas ainda não com eficiência: as primeiras máquinas construídas não demonstraram os resultados esperados pelos pesquisadores, desencadeando uma série de novos trabalhos [Sri86] [Gurd83] [Gur85].

A tendência dominante nos últimos anos tem sido buscar no modelo dirigido pelo controle soluções eficientes para os problemas encontrados. As arquiteturas híbridas, originadas desses trabalhos, apresentam características dos dois modelos e mostram melhoras em relação aos resultados iniciais.

Entre os temas pesquisados está o da determinação do tamanho das tarefas potencialmente paralelas. O modelo de fluxo de dados puro explora paralelismo de granularidade fina, demandando sincronizações para todas as instruções executadas. Isso ocorre mesmo em situações onde as sincronizações seriam desnecessárias, como, por exemplo, em trechos de código estritamente seqüencial.

Este artigo aborda esse problema, propondo alguns métodos para reconhecimento e agrupamento de código seqüencial em programas destinados à execução em máquinas de fluxo de dados. A seção 2 descreve o modelo de fluxo de dados. A seção 3 mostra questões relacionadas ao processamento paralelo. A seção 4 relata alguns métodos para identificação de código seqüencial. A seção 5 apresenta resultados preliminares obtidos da análise do código objeto gerado por programas escritos em SISAL [McG83].

2 MODELO DE FLUXO DE DADOS

O modelo de fluxo de dados surgiu como uma nova e revolucionária abordagem para a realização de processamento paralelo. Tomando-se como termo de comparação o modelo de von Neumann, as principais características do modelo de fluxo de dados são [Tre82] [Vee86]:

- não apresenta o conceito de memória de dados compartilhada, associado ao uso de variáveis.
- o seqüenciamento do programa é restringido apenas pelas dependências de dados entre as instruções. Não existe controle explícito de execução através de um contador de programa[1].
- resultados da execução de instruções são passadas como fichas para as próximas instruções.

Neste modelo, programas executáveis são geralmente representados por um grafo orientado, chamado **grafo de fluxo de dados (GFD)** [Dav82] [Tre82] [Vee86]. Num GFD, cada vértice representa uma instrução e cada aresta orientada, uma dependência de dados entre duas instruções. A figura 1 apresenta um trecho de programa e sua representação como um GFD.

O ritmo de execução de um programa é determinado pelo fluxo de dados através do GFD. Os dados, representados por fichas, percorrem as arestas chegando às instruções. Quando todos o dados de que uma instrução necessita tiverem chegado, ela torna-se habilitada e pronta para ser executada. Com essa execução, novos dados serão produzidos e habilitarão outras instruções, até a obtenção do resultado final.

[1] *program counter*

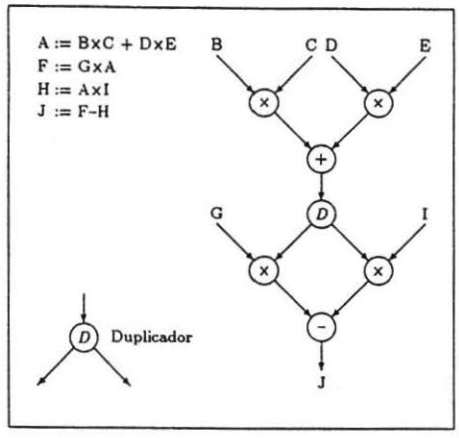

Figura 1: Um trecho de programa e seu Grafo de Fluxo de Dados.

Programas para o modelo de fluxo de dados são normalmente expressos em linguagens específicas [Ack82]. Essas linguagens têm características de linguagens funcionais, como por exemplo, ausência de efeitos colaterais[2]. Outras características, como localidade de efeito, atribuição única[3] e notação própria para iterações, também são apresentadas por essas linguagens.

3 QUESTÕES EM PROCESSAMENTO PARALELO

Máquinas multiprocessadoras baseadas no modelo von Neumann têm dificuldades em resolver algumas questões, tais como:

- *a exploração de paralelismo em programas.* As linguagens utilizadas nestas máquinas pertencem à classe imperativa [Bac78] [Ghe87]. As linguagens imperativas obtêm um bom desempenho em arquiteturas monoprocessadoras, pois refletem diretamente o modelo von Neumann, mas apresentam características como, por exemplo, efeitos colaterais e atribuições múltiplas, que prejudicam a exploração de paralelismo [Ack82] [Veg84]. Nessas linguagens, a determinação das tarefas a serem executadas concorrentemente é deixada a cargo do programador — o que aumenta chances de inserção de erros nos programas – ou realizada automaticamente pelo compilador — dificultando a obtenção de bons resultados em trechos não vetorizáveis.

- *a granularidade das tarefas ou o tamanho de cada unidade de trabalho a ser executada em um processador.* Dado o objetivo de minimizar o tempo de execução de um programa, o ideal é maximizar a ocupação das unidades de processamento, o que pode ser conseguido pela diminuição do tamanho das unidades de trabalho, ou seja, tornando-se a granularidade do paralelismo explorado tão fina quanto possível. No entanto, essa intenção é prejudicada pela excessiva sobrecarga[4] associada ao escalonamento e sincronização destas tarefas.

- *escalonamento de processos.* É a determinação de que processo deve ser executado em que processador em cada instante. O escalonamento de processos pode ser estático ou

[2] *side effects*
[3] *single assignment*
[4] *overhead*

dinâmico[Gaj 85]. No escalonamento estático as tarefas são alocadas a processadores durante o projeto do algoritmo, pelo programador, ou em tempo de compilação, pelo compilador. O escalonamento dinâmico, realizado durante a execução do programa, permite melhor utilização de recursos, mas ao preço de um tempo maior de escalonamento.

- *sincronização das tarefas.* Tarefas paralelas necessitam de mecanismos de sincronização, para atuarem de modo cooperativo (visando atingir um objetivo comum) ou concorrente (disputando um recurso comum) [Alm89]. Os mecanismos de sincronização são basicamente agrupados em dois grupos. Os do primeiro grupo, entre eles *test/set*, semáforos, monitores e barreiras[5], são utilizados em ambientes com compartilhamento de memória. Já os do segundo grupo são utilizados em ambientes com transmissão de mensagens, sendo representados principalmente por mecanismos baseados nas primitivas *send/receive*. A utilização desses mecanismos implica geralmente num atraso à espera da validação de uma determinada condição.

- *balanceamento de carga dos processadores.* Como as tarefas geralmente possuem tamanhos muito díspares, alguns processadores podem estar sobrecarregados enquanto outros estão desocupados, diminuindo o desempenho total.

- *contenção de memória.* O número de acessos à memória geralmente é grande e em arquiteturas de memória compartilhada, o tempo de latência é considerável. Mecanismos como, por exemplo, distribuição de memória, utilização de memória de acesso rápido[6] e busca antecipada de instruções amenizaram este problema [Arv87].

Arquiteturas de fluxo de dados apresentam soluções elegantes, embora nem sempre eficientes, para os problemas citados acima:

- o paralelismo está implícito no modelo, retirando a necessidade de o programador especificar as tarefas concorrentes e facilitando a identificação automática de código paralelo pelo compilador mesmo em trechos não vetorizáveis.

- a granularidade dos processos é bastante fina — geralmente ao nível de instruções de máquina — permitindo a execução simultânea de um grande número de instruções.

- o escalonamento das instruções depende somente da chegada dos dados de que elas necessitam, sendo, portanto, realizado em tempo de execução. A tarefa mais complexa é a identificação das instruções que estão prontas para executar, por envolver sincronização dos dados.

- a sincronização está embutida no modelo. É a sincronização dos dados que habilita a execução das instruções de um programa.

- como todas as tarefas têm aproximadamente o mesmo tamanho, o problema de balanceamento de carga fica muito atenuado.

- a inexistência de memória para armazenamento de dados no modelo de fluxo de dados puro exclui o problema de latência de memória. A introdução de memória para estruturas de dados em implementações recentes não altera este fato, pois os acessos são divididos em duas fases. Um processador emite uma requisição de acesso à memória de estruturas e fica liberado para executar outras instruções. Quando o dado desejado estiver disponível, a memória de estruturas o envia novamente a algum processador, que então completa a operação.

A maioria das máquinas que seguem o modelo de fluxo de dados foi implementada com base numa estrutura de dois componentes interligados em anel, como mostra a figura 2:

- uma unidade de emparelhamento, responsável pela sincronização dos dados.

- uma unidade de processamento, responsável pela execução das instruções.

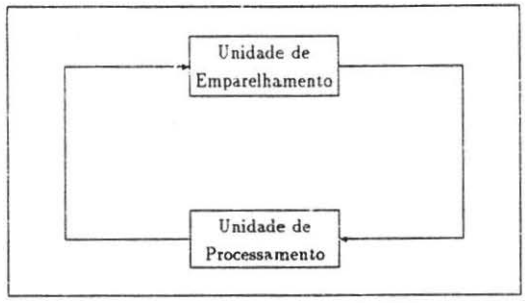

Figura 2: Estrutura básica de máquina de fluxo de dados.

O anel estabelece o caminho percorrido pelos dados, sincronizados pela primeira unidade e transformados pela segunda. Exemplos de arquiteturas com esta estrutura são as Máquina de Fluxo de Dados de Manchester [Wat82] [Gur85] e a de Irvine [Vee86].

As máquinas de fluxo de dados não apresentaram o desempenho esperado. Manipulação de estruturas de dados, mecanismo de emparelhamento e custos de comunicação entre instruções foram alguns dos principais motivos [Gaj82]. Vários trabalhos estão sendo realizados na tentativa de viabilizá-las, principalmente buscando no modelo von Neumann soluções adequadas para os problemas [Pap90] [Ian88] [Sak89] [Bue 87] [Gra 89] [Gao90].

Uma das linhas de pesquisa seguidas nesses trabalhos é a alteração da granularidade explorada pelo modelo. A próxima seção discute este tema, apresentando alguns métodos para agrupamento de instruções.

4 AGRUPAMENTO DE CÓDIGO SEQÜENCIAL

4.1 PORQUE AGRUPAR

A sincronização de instruções e a comunicação entre elas são alguns dos fatores críticos na computação de alto desempenho em ambientes multiprocessadores.

Em máquinas de fluxo de dados, sincronizações envolvem armazenamento de dados, além de mecanismos de emparelhamento de dados e ativação de instruções. Já a comunicação entre instruções envolve o tráfego de dados por todo um anel de interconexão das unidades de uma máquina.

Códigos seqüenciais apresentam uma característica que pode ser utilizada para a redução da interferência negativa destes dois fatores sobre o desempenho: uma instrução depende somente do resultado da instrução anterior para a sua execução.

Num trecho de código seqüencial, dada uma instrução, é sempre possível determinar de antemão sua sucessora. Assim, o emparelhamento de dados torna-se desnecessário e a sincronização pode ser eliminada.

Decorrente ainda da característica de código seqüencial, a comunicação entre instruções pode ser reduzida. Ao invés de corresponder a um ciclo de anel, pode-se transformá-la em escritas e leituras de dados em registradores de um único processador, desde que a ele seja atribuída a execução das instruções fonte e destino.

Estas duas últimas afirmações baseiam-se na existência de tecnologia e modelo de execução apropriados. Mas executar código seqüencial eficientemente nada mais é do que as máquinas monoprocessadores realizam suportando o modelo von Neumann.

[5] *barriers*
[6] *cache memory*

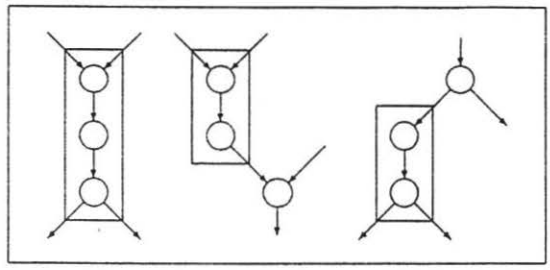

Figura 3: Exemplos de blocos identificados pelo método de agrupamento sem emissão de resultados.

Então, reconhecer e agrupar código seqüencial pode propiciar uma geração de código de melhor qualidade, possibilitando uma redução da interferência de fatores como a sincronização e comunicação entre instruções no desempenho de máquinas de fluxo de dados.

4.2 COMO AGRUPAR

Esta seção descreve alguns métodos para agrupamento de código seqüencial em blocos. Os métodos, baseados em algoritmos sobre GFD, são executados sobre o código objeto gerado pelo compilador da linguagem SISAL [McG83]. O código gerado é compatível com a Máquina de Fluxo de Dados de Manchester.

4.2.1 BLOCOS SEM EMISSÃO DE RESULTADOS INTERMEDIÁRIOS

Este é o método mais simples para agrupamento de código seqüencial. Baseia-se na identificação de estruturas de código seqüencial com as seguintes restrições:

a com exceção da primeira, todas as instruções possuem uma única dependência de entrada;

b com exceção da última, todas as instruções produzem apenas um resultado.

A restrição **a** garante que o bloco pode ser executado da primeira à última instrução sem interrupções.

A restrição **b** determina que não é mais possível armazenar no processador os resultados intermediários. Assim, o processador responsável pelo processamento dessa seqüência de instruções, preocupa-se com a emissão de resultados somente ao final da execução do bloco.

A figura 3 mostra alguns exemplos de blocos de código seqüencial agrupados pos este método.

4.2.2 BLOCOS COM EMISSÃO DE RESULTADOS INTERMEDIÁRIOS

Os métodos apresentados a seguir desconsideram a restrição **b** da seção anterior, permitindo que resultados intermediários possam ser produzidos por um bloco seqüencial. Para tanto, o processador deve suportar a emissão de resultados durante a execução de um bloco.

Os algoritmos que implementam os métodos basicamente percorrem o GFD em profundidade, procurando os maiores trechos seqüenciais de código. Para uma instrução que emite dois resultados é escolhido o bloco seqüencial de maior tamanho entre o da esquerda, e o da direita para incorporação. No caso de os blocos serem do mesmo tamanho é escolhido arbitrariamente um deles.

No estágio atual estão sendo examinados três métodos:

- PURO

  Este método implementa a descrição dada acima, relaxando apenas a restrição **b**. A figura 4 apresenta exemplos de blocos agrupados por este método.

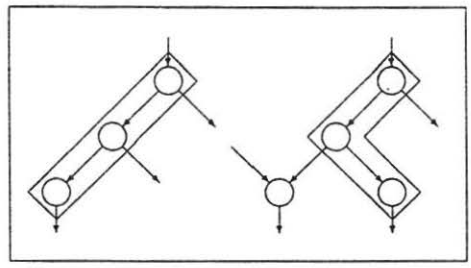

Figura 4: Exemplos de blocos identificados pelo método de agrupamento com emissão de resultados puro.

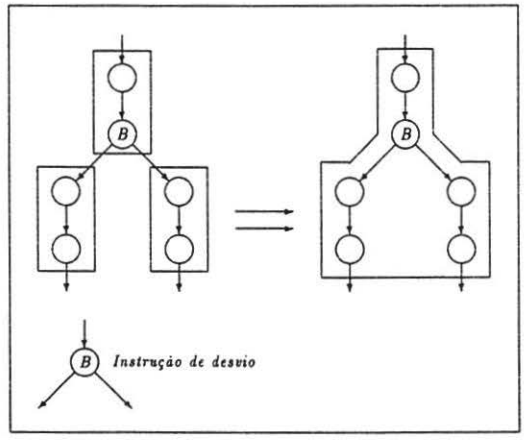

Figura 5: Exemplo de agrupamento pelo método de com emissão de resultados realizando tratamento de instruções de desvio.

- COM TRATAMENTO DE INSTRUÇÕES DE DESVIO

  Este método estende o anterior identificando instruções de desvio condicional e agrupando os dois possíveis caminhos em um único bloco.

  No método anterior a ocorrência de instruções de desvio define o fim de um bloco. Aqui, com a retirada desta restrição, é possível reduzir mais uma sincronização de instruções. Porém, é importante notar que há um preço para este ganho: o bloco conterá instruções que não serão executadas e dependendo do tamanho dos possíveis caminhos e do custo da carga de instruções na unidade de processamento, não seja vantajoso aplicá-lo.

  A figura 5 mostra um exemplo de agrupamento realizado por este método.

- COM AGRUPAMENTO DE INSTRUÇÕES DUPLICADORAS

  Para arquiteturas baseadas no modelo de fluxo de dados, que utilizam a passagem de dados como mecanismo de transferência de controle, são geradas instruções duplicadoras quando é necessário que um mesmo dado seja consumido por mais que uma instrução. Essas instruções duplicadoras são organizadas em árvores balanceadas, cujas folhas emitem os dados para as

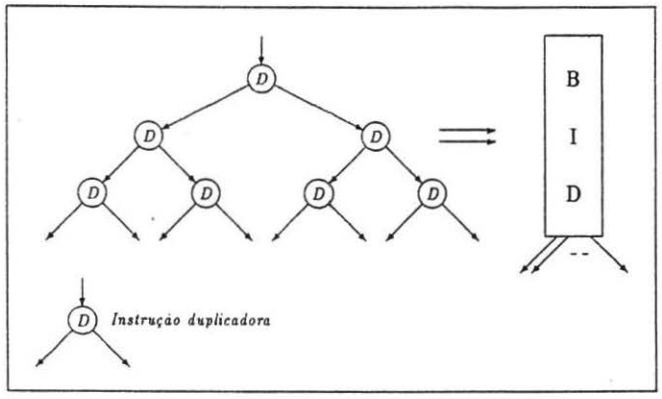

Figura 6: Exemplo de agrupamento pelo método de com emissão de resultados realizando tratamento de instruções duplicadoras.

instruções consumidoras.

O método aqui descrito, estendendo o método PURO, realiza o agrupamento desta árvore em apenas um bloco que emite o número de resultados equivalente ao número de resultados emitidos pelas folhas. Assim, os custos de sincronização e de comunicação respectivo as arestas internas da árvore podem ser reduzidos.

A figura 6 mostra um exemplo de aplicação deste método, produzindo um bloco de instruções duplicadoras (BID), e merece uma análise mais cuidadosa. O fato de seqüencializar a emissão de resultados duplicados faz com que, tomando como exemplo a figura 6, uma instrução que recebia um dado depois de dois ciclos de anel e da execução de três instruções passa a recebê-lo somente depois do i-ésimo envio de resultados. O i-ésimo envio é determinado pela posição da instrução entre dos destinos da árvore de instruções duplicadoras. O método, então, deve levar em consideração os tempos da máquina alvo para a realização do agrupamento.

O método também trata de instruções do tipo *TUPLICATE*, que resultam de uma otimização de árvores de instruções duplicadoras realizadas pelo compilador utilizado.

Uma variação deste método permite incorporar a cada bloco encontrado o maior dos blocos seqüenciais que o sucedem.

4.3 CONSIDERAÇÕES

A realização do agrupamento de instruções seqüenciais em blocos tem algumas conseqüências nas arquiteturas de fluxo de dados:

- a granularidade passa de fina a variável, uma vez que passam a ser definidos blocos de tamanhos distintos;
- a unidade de processamento deve permitir o armazenamento de resultados intermediários, assim como a execução de trechos de código utilizando o modelo von Neumann;
- o fato de a granularidade ser fina permitia que o algoritmo de escalonamento fosse simples. Com a alteração da granularidade, devem-se utilizar meios mais complexos para tal tarefa [Lor].

5 RESULTADOS PRELIMINARES

O métodos foram aplicados sobre o código objeto de seis programas. O código objeto desses programas foi gerado por um compilador SISAL, utilizando o conjunto de instruções da Máquina de Fluxo de Dados de Manchester. São eles:

- BININT - integração binária;
- LAPLACE - transformada de LAPLACE;
- MMS - multiplicação de matrizes;
- QUICK - ordenação pelo método *quicksort*;
- INTERP - interpolação;
- GAUSS - eliminação de Gauss.

O número de instruções e o número de dependências de dados ou arestas de cada um destes programas são mostrados na tabela 1.

	BININT	LAPLACE	MMS	QUICK	INTERP	GAUSS
Instruções	149	306	327	577	720	1145
Arestas	242	464	483	854	1136	1807

Tabela 1: Informações sobre os programas analisados

A tabela 2 apresenta os resultados provenientes da utilização do método de agrupamento sem emissão de resultados. Os valores médios obtidos são baixos: em média, cerca de apenas 33% de instruções pertencem a algum grupo e somente aproximandamente 14% das arestas podem ser reduzidas.

	BININT	LAPLACE	MMS	QUICK	INTERP	GAUSS	média
% inst. agrupadas	31.5	24.3	30.2	41.2	37.0	36.7	33.5
% arestas retiradas	11.5	14.0	17.5	15.1	13.2	13.2	14.1

Tabela 2: Dados obtidos com o método sem emissão de resultados

Com os valores obtidos utilizando-se o método com emissão de resultados intermediários PURO, mostrados na tabela 3, pode-se observar uma melhora significativa na qualidade do agrupamento. Em média, tanto o número de arestas reduzidas quanto o número de instruções agrupadas tiveram seus valores dobrados. Isso vem confirmar o fato de que a restrição b inibe a possibilidade de uma boa qualidade de agrupamento de código.

	BININT	LAPLACE	MMS	QUICK	INTERP	GAUSS	média
% inst. agrupadas	59.7	58.8	59.9	68.4	59.3	58.8	60.8
% arestas retiradas	22.7	26.7	29.3	32.3	25.6	26.0	27.1

Tabela 3: Dados obtidos com o método com emissão de resultados puro

Os resultados obtidos com a utilização do método de agrupamento de instruções com emissão de resultados tratando as instruções de desvio são apresentados na tabela 4. A diferença, em relação ao método anterior, não é significativa e leva-nos a questionar a validade desta política de agrupamento.

	BININT	LAPLACE	MMS	QUICK	INTERP	GAUSS	média
% inst. agrupadas	59.7	58.8	59.9	68.9	59.3	59.3	60.9
% arestas retiradas	22.7	26.7	29.3	32.3	25.6	26.0	27.2

Tabela 4: Dados obtidos com o método com emissão de resultados com tratamento de instruções de desvio

A tabela 5 mostra resultados utilizando-se o tratamento de instruções duplicadoras. Observa-se que a qualidade de agrupamento diminuiu. Embora, em média, o número de instruções agrupadas tenha aumentado, o número de arestas reduzidas diminuiu, provocando um "esfarelamento" dos blocos, ou seja, uma maior quantidade de blocos de tamanho menor.

	BININT	LAPLACE	MMS	QUICK	INTERP	GAUSS	média
% inst. agrupadas	69.7	62.0	61.7	68.4	60.1	60.6	63.7
% arestas retiradas	28.0	25.4	26.2	27.2	23.3	24.1	25.7

Tabela 5: Dados obtidos com o método com emissão de resultados com tratamento de instruções duplicadoras

Permitindo o acoplamento a um bloco de instruções duplicadoras de um dos seus blocos sucessores, obteve-se uma melhora na qualidade de agrupamento, em relação ao método de somente tratar instruções duplicadoras. Essa variação permitiu que o número de arestas reduzidas aumentasse, eliminando o problema de "esfarelamento". Seus resultados são apresentados na tabela 6.

	BININT	LAPLACE	MMS	QUICK	INTERP	GAUSS	média
% inst. agrupadas	70.7	66.0	65.7	74.1	64.4	65.1	67.5
% arestas retiradas	29.7	31.6	34.3	35.9	29.5	30.2	31.9

Tabela 6: Dados obtidos com o método com emissão de resultados tratando instruções duplicadoras usando variação

6 CONCLUSÃO

Atacando problemas apresentados no modelo de fluxo de dados, foram elaborados alguns métodos para agrupamento de código seqüencial. Também foram apresentados os resultados obtidos com a utilização de tais métodos a partir do código objeto de programas escritos em SISAL. Embora sejam preliminares, mostram indícios da possibilidade de melhorar a qualidade do código gerado.

Mais trabalho ainda deve ser realizado. A utilização de um número mais representativo e significativo de programas e uma análise do comportamento dinâmico destes são de suma importância para conclusões mais consistentes sobre a real utilidade dos métodos de agrupamento.

REFERÊNCIAS

[Ack82] Ackerman, W. B. Data Flow Languages. *IEEE Computer*, 15(2):15–25, fev 1982.

[Alm89] Almasi, G. S. & Gottlieb, A. *Highly Parallel Computing.* Benjamin/Cummings, 1989.

[Arv87] Arvind & Iannucci, R. A. Two Fundamentals Issues in Multiprocessing. In *Thakkar, S. S., ed., Selected Reprints on Dataflow and Reduction Architectures*, pp. 140–164, 1987.

[Bac78] Backus, J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, ago 1978.

[Bue 87] Buehrer, R. & Ekanadham, K., Incorporating Data Flow Ideas into von Neumann Processors for Parallel Execution. *IEEE Trans. on Computers*, 36(12):1515–1522, dez 1987.

[Dav82] Davis, A. L. & Keller, R. M. Data Flow Program Graphs. *IEEE Computer*, 15(2):26–41, fev 1982.

[Gaj82] Gajski, D. D., Padua, D. A., Kuck, D. J. & Kuhn, R. H. A Second Opinion on Data Flow Machines and Languages. *IEEE Computer*, 15(2):58–69, fev 1982.

[Gaj 85] Gajski, D.D. & Pier, J., Essential Issues in Multiprocessor Systems, *IEEE Computer, pp. 9–27*, jun 1985.

[Gao90] Gao, G. R., Hum, H. H. J. & Wong, Y. Towards Efficient Fine-Grain Software Pipelining. In *International Conference on Supercomputing*, pp. 369–379, 1990.

[Ghe87] Ghezzy, C. & Jazayeri, M. *Programming Languages Concepts 2/E* Cap. 7, Wiley, 1987.

[Gra 89] Grafe, V.G. et al, The epsilon Dataflow-Processor. In *Proceedings of the* 16^{th} *Annual International Symposium on Computer Architecture*, pp. 36–45, mai 1989.

[Gurd83] Gurd, J. & Watson, I. Preliminary Evaluation of a Prototype Dataflow Computer. *IFIP*, pp. 545–551, 1983.

[Gur85] Gurd, J. R., Kirkham, C. C. & Watson, I. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, 28(1):34–52, jan 1985.

[Ian88] Iannucci, R. A. Towards a Dataflow/von Neumann Hybrid Architectures. In *Proceedings of the* 15^{th} *Annual International Symposium on Computer Architectures*, pp. 131–140, jun 1988.

[Lor] Lorenzo, P. A. R. Escalonamento de Processos em uma Arquitetura de Fluxo de Dados. Dissertação de Mestrado em elaboração. Departamento de Ciência da Computação, UNICAMP.

[McG83] McGraw, J. et. al. SISAL: Stream and Iteration in a Single-Assignment Language. Language Reference Manual, version 1.1, Departament of Computer Science, University of Manchester, jul 1983.

[Pap90] Papadopoulos, G. M. & Culler, D. E. Monsoon: An Explicit Token-Store Architecture. In *Proceedings of the* 17^{th} *Annual International Symposium on Computer Architectures*, pp. 82–91, 1990.

[Sak89] Sakai, S. et al. An Architecture of a Dataflow Single Chip Processor. In *Proceedings of the* 16^{th} *Annual International Symposium on Computer Architectures*, pp. 46–53, mai 1989.

[Sri86] Srini, V. P. An Architectural Comparison of Dataflow Systems. *IEEE Computer*, 19(3):68–88, mar 1986.

[Tre82] Treleaven, P. C., Brownbridge, D. R. & Hopkins, R. P. Data-Driven and Demand-Driven Computer Architectures. *ACM Computing Surveys*, 14(1):93–143, mar 1982.

[Vee86] Veen, A. H. Dataflow Machine Architecture. *ACM Computing Surveys*, 18(4):365–396, dez 1986.

[Veg84] Vegdahl, S. R. A Survey of Proposed Architectures for the Execution of Functional Languages. *IEEE Transactions on Computers*, C-23(12):1050–1071, dez 1984.

[Vis] Visoli, M. C. Tramento de Código Seqüencial no Modelo de Fluxo de Dados. Dissertação de Mestrado em elaboração. Departamento de Ciência da Computação, UNICAMP.

[Wat82] Watson, I. & Gurd, J. R A Practical Dataflow Computer. *IEEE Computer*, 15(2):51-57, fev 1982.