Armazenamento de Estruturas de Dados em Computadores a Fluxo de Dados

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Manoel Pedro Sá e aprovada pela comissão julgadora.

Campinas, 15 de janeiro de 1991.

Prof. Dr. Arthur João Catto

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para a obtenção do Título de Mestre em Ciência da Computação.

Armazenamento de Estruturas de Dados em Computadores a Fluxo de Dados

Dissertação de Mestrado
Departamento de Ciência da Computação

IMECC - Unicamp

Orientador: Prof. Dr. Arthur João Catto[1]
Autor: Manoel Pedro Sá[2]

19 de dezembro de 1990

[1]Ph.D. Universidade de Manchester, Professor MS-4 IMECC Unicamp, Diretor do Centro Tecnológico para Informática (CTI)
[2]Eng. Eletricista PUC/RJ, Mestrando IMECC Unicamp

À Suzete M. Cerutti (Nenê),
em nome de um amor único.

Agradecimentos

Gostaria, em primeiro lugar, de agradecer ao governo brasileiro, que através do CNPq, financiou esta pesquisa.

Agradeço ao Departamento de Ciência da Computação, pela oportunidade desse trabalho.

Dentro do grupo que estuda fluxo de dados obtive apoio e surgiram diversas idéias em discussões com o Prof. Dr. Carlos Ruggiero da USP e com Cecília Calsavara e Antônio Fernando C. Branco.

Os agradecimentos especiais são dirigidos ao meu orientador Prof. Dr. Arthur J. Catto por ter me introduzido no assunto de arquiteturas dirigidas pelos dados e pelo seu incentivo e colaboração durante a pesquisa.

Sumário

Esta dissertação faz uma discussão dos principais conceitos relacionados ao armazenamento de estruturas de dados em computadores a fluxo de dados dinâmicos e a relação destes conceitos com a arquitetura. Como exemplos de computadores que têm armazenamento de estruturas são apresentados os computadores a fluxo de dados da Universidade de Manchester e do MIT. Introduzimos a seguir uma nova organização para suportar operações locais na unidade responsável pelo armazenamento de estruturas com o objetivo de aumentar o desempenho dos computadores a fluxo de dados. A avaliação parcial que realizamos sobre esta proposta baseia-se em resultados de simulação.

Abstract

This dissertation makes a survey of the most important concepts related to stored data structures in tagged dataflow computers and studies their relation to the architecture. As examples of computers that have stored data structures it discusses the dataflow computers of Manchester and MIT. On the dissertation we propose local operations in the unit responsible for the storage of data structures as a way to increase the performance of dataflow computers. This proposal is partly evaluated on the basis of results obtained from simulation.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Atualmente existem diversas linhas de pesquisa em computadores paralelos. Dentre elas, o modelo Fluxo de Dados baseia-se na exploração implícita do paralelismo de operações e no processamento dirigido pela disponibilidade de dados. Este modelo tem despertado atenção porque apresenta um paralelismo elevado e resolve com simplicidade certos problemas difíceis como a sincronização de eventos e a divisão de tarefas entre processadores [5,6,24].

No modelo Fluxo de Dados programas são expressos como grafos orientados: os vértices representam instruções e as arestas que os unem, os caminhos percorridos por fichas representando os dados. Existe uma divisão do modelo conforme o modo de execução do grafo do programa. No modelo *estático* não existe código reentrante enquanto no modelo *dinâmico* esta limitação é superada introduzindo-se rótulos nas fichas a fim de separar logicamente ativações diferentes do mesmo código.

A concepção destes sistemas tem sido feita de forma a isolar o programador dos detalhes de organização e implementação. Desta forma, como aqueles detalhes não influem sobre o ambiente do programador, a tarefa de programar fica facilitada. Linguagens funcionais como SISAL [33] e Id [34], intimamente voltadas à arquitetura de fluxo de dados, foram desenvolvidas. Estas linguagens adotam uma semântica funcional com suporte para estruturas de dados.

Esta dissertação diz respeito ao armazenamento de estruturas de dados em computadores a fluxo de dados dinâmicos.

1.1 Organização e Objetivos da Dissertação

A dissertação tem dois objetivos: abordar as formas de processamento envolvendo estruturas em fluxo de dados e as dificuldades associadas, além de propor uma nova abordagem de processamento a fluxo de dados que supere algumas das limitações das propostas existentes.

O trabalho apóia-se numa resenha (*survey*) dos conceitos mais importantes que surgiram relacionados ao armazenamento de estruturas em fluxo de dados [9,19,28,38,36]. A resenha contém duas partes: uma envolvendo os conceitos propriamente ditos, outra descrevendo implementações existentes de unidades concebidas para efetuar aquele armazenamento. Considera-se que esses conceitos são indispensáveis para uma avaliação correta dos problemas e dos compromissos envolvidos nessa linha de pesquisa. Como o nosso interesse está voltado para a arquitetura a nível de sistema, daremos maior ênfase aos conceitos de baixo nível, próximo ao *harware* do computador.

O trabalho é apresentado em sete capítulos, dos quais este é o primeiro.

O Capítulo 2 é uma introdução sucinta aos conceitos básicos do modelo Fluxo de Dados, especialmente dirigida aos leitores que não estão familiarizados com o assunto.

A alteração do modelo para um tratamento adequado de estruturas de dados assim como as repercussões das estruturas de dados sobre a arquitetura do computador são apresentadas no Capítulo 3. As primeiras idéias nesse assunto originaram-se de um modelo que tratava exclusivamente os dados escalares, mas a extensão uniforme desse modelo para estruturas de dados foi inviável por questões de eficiência e desempenho. Discutiremos, desta forma, os motivos para o armazenamento de estruturas de dados, a representação de estruturas de dados armazenadas e as diferenças entre estruturas ditas *estritas* e *não-estritas*.

O Capítulo 4 descreve duas implementações semelhantes de unidades para o armazenamento de estruturas de dados (*Memórias de Estruturas*): a Memória de Estruturas do computador de Manchester, base deste trabalho, e a Memória de Estruturas do computador do MIT. Discutem-se os aspectos positivos e algumas das limitações destas implementações.

O Capítulo 5 descreve a contribuição deste trabalho à linha de pesquisa relacionada ao armazenamento de estruturas de dados em computadores a fluxo de dados. O capítulo mostra a conveniência da realização de operações sobre vetores na própria Memória de Estruturas. Não encontramos referência na literatura a uma idéia semelhante.

Diante da complexidade do assunto, o Capítulo 6 dedica-se a uma avaliação, por simulação, de parte desta proposta. A avaliação é feita comparando-se os resultados que seriam produzidos numa Memória de Estruturas como a de Manchester com os de uma Memória de Estruturas que possua as características necessárias ao suporte de operações vetoriais. As operações vetoriais propriamente ditas não são simuladas. No trabalho, usamos o simulador [13] do computador a fluxo de dados de Manchester.

O Capítulo 7 apresenta as conclusões da dissertação e sugere tópicos de pesquisa suplementar. Os resultados da simulação confirmam que a reunião das ações de armazenamento e operações sobre vetores na Memória de Estruturas é conceitualmente vantajosa e pode resultar num aumento da eficiência de computadores a fluxo de dados.

Capítulo 2

Conceitos Básicos

Fluxo de Dados é uma dentre as muitas linhas de modelos de processamento que dão origem a *ambientes funcionais* [45], sistemas de computação que procuram superar as deficiências intrínsicas do modelo convencional de von Neumann. As deficiências do modelo de von Neumann estão sobretudo no controle centralizado e explícito do processamento [5,12].

Discutimos alguns dos conceitos preliminares de fluxo de dados, inicialmente com um exame entre a semântica imperativa do modelo de von Neumann e a semântica funcional do modelo Fluxo de Dados.

2.1 Semântica Imperativa e Semântica Funcional

Um tema importante em sistemas de processamento paralelo diz respeito à semântica adotada na computação, pois esta repercute diretamente na forma de armazenamento e tratamento das estruturas de dados. Neste trabalho, entende-se por semântica a maneira lógica como o sistema se comporta.

A semântica do modelo de von Neumann é imperativa, ou seja, existe um controle *explícito* do programa sobre a ordem de execução das instruções. Além disto, este modelo caracteriza-se pela presença de *estados*, aparentes ao programador, em decorrência da associação existente entre nomes (variáveis) no programa e posições de armazenamento. Como conseqüência da combinação destas duas propriedades, podem ocorrer *efeitos colaterais* (*side-effects*) na execução de instruções.

Nos modelos dirigidos pelos dados, como o Fluxo de Dados, a semântica das operações sobre os dados é funcional. Não existem efeitos colaterais pois a única dependência que se estabelece está entre dado e operação. Uma operação comporta-se como uma função: toma argumentos e produz um resultado. Os argumentos permanecem inalterados. O controle sobre a execução das operações é *implícito*, sendo estabelecido a partir das dependências existentes entre as próprias operações.

2.1.1 Relação entre Semântica e Paralelismo

O modelo de von Neumann monoprocessado é perfeitamente adequado a uma forma de processamento seqüencial. Apesar da dependência entre dado e local de armazenamento, as instruções não interferem entre si porque são executadas seqüencialmente.

Quando se tenta empregar o modelo de von Neumann ao processamento paralelo, as dependências entre dado e local de armazenamento e entre dado e operação restringem o paralelismo. Para aumentar o paralelismo é necessário empregar técnicas sofisticadas de compilação como *renaming* e expansão de escalares [37]. Kumar [31] argumenta, a partir de estudos de simulação, que o grau de paralelismo atingível num programa FORTRAN típico é menor que dez instruções por ciclo de processamento quando as variáveis do programa são alocadas estaticamente (isto é, antes da execução). Quando se empregam *variáveis dinâmicas* (variáveis alocadas durante a execução e por isto com mais de um endereço associado ao seu nome) o paralelismo pode chegar a centenas de instruções por ciclo. O aumento mais significativo do grau de paralelismo entre operações é obtido empregando-se a técnica de variáveis dinâmicas para dados escalares. No caso de variáveis dinâmicas para vetores, o acréscimo de paralelismo não parece ser tão significativo porque se incorre num uso proporcionalmente maior de memória.

Quando comparadas às técnicas estáticas (*renaming* e expansão de escalares), as variáveis dinâmicas mostram-se mais flexíveis porque são capazes de superar dependências não observáveis em tempo de compilação. Variáveis dinâmicas, contudo, incorrem no custo adicional de uma maior necessidade de espaço e na dificuldade trazida pela gerência dinâmica deste espaço.

O modelo Fluxo de Dados, por ser funcional, não introduz dependências desnecessárias entre as instruções. O paralelismo que pode surgir na execução

concorrente das instruções é maior do que no modelo de von Neumann, e a alocação de instruções a processadores fica simplificada. O que se observa é que as técnicas de compilação adotadas no modelo de von Neumann para processamento paralelo originam um desdobramento "artificial" de variáveis semelhante ao que acontece naturalmente no modelo Fluxo de Dados.

2.1.2 Relação entre Semântica e Memória

No que concerne ao uso de memória para o armazenamento de dados, um modelo com semântica imperativa é mais eficiente do que um modelo com semântica funcional. Existe uma economia de espaço originada na computação baseada no *estado* de "variáveis" (num programa em linguagem imperativa as variávies são de fato *variáveis de estado*) e uma economia de processamento decorrente da associação estática dessas variáveis a endereços na memória, no caso das variáveis globais. Estas economias podem ser observadas principalmente no processamento de estruturas de dados. No modelo com semântica imperativa, uma estrutura de dados que é argumento de uma operação pode ser *atualizada* escrevendo-se o resultado sobre as mesmas posições lidas do argumento. Num modelo com semântica funcional, uma operação sobre uma estrutura de dados produz como resultado uma estrutura nova que nem sempre pode ocupar as mesmas posições da estrutura argumento, por isso exigindo espaço adicional.

O consumo de memória que ocorre na execução de um programa funcional geralmente é imprevisível devido à maneira recursiva de realizar-se a computação. A forma como a memória é empregada depende do modelo, mas a característica comum a todos eles é a associação de nomes a endereços somente durante a execução. O espaço de memória associado a um nome pode ser reaproveitado no momento em que não for mais útil, isto é, no momento em que for desnecessária uma outra consulta ao valor armazenado naquela posição. A recuperação de posições de memória fora de uso é feita implicitamente por um processo dedicado de gerência dinâmica de espaço.

Para obterem maior flexibilidade, as linguagens imperativas mais modernas também apresentam *alocação dinâmica* de memória (ou seja a criação dinâmica de variáveis), instruída por meio de comandos presentes na linguagem. Contudo, ainda assim, a gerência de memória para estas linguagens em geral é feita explicitamente no programa, embora isto seja uma tarefa difícil e induza o programador a erros. A vantagem da gerência explícita

sobre a implícita é a maior eficiência obtida no processamento.

2.2 Grafos de Fluxo de Dados

A semântica da computação baseia-se essencialmente na semântica da linguagem de programação, uma vez que o programador tem contato apenas com os recursos que a linguagem lhe oferece. Os recursos e o comportamento da máquina são aparentes ao programador apenas através da sua interação com a linguagem de programação. Se a arquitetura não tiver um modelo de computação adequado à semântica da linguagem, o mapeamento entre o "processador" que executa a linguagem de alto nível e o processador físico que de fato realiza a computação será inadequado, contribuindo para uma queda na eficiência e desempenho do sistema como um todo. No modelo Fluxo de Dados o mapeamento entre linguagem e processadores ocorre de modo natural, com base na definição de uma linguagem intermediária. A perspectiva de bom desempenho é favorável, principalmente em virtude do aproveitamento correto do paralelismo oferecido pelo uso de uma linguagem funcional.

Uma linguagem funcional pode ser traduzida em uma outra linguagem, de nível mais baixo, que pode ser representada sob a forma de um grafo orientado [2]. Nesse grafo, os vértices representam operações e as arestas, os caminhos percorridos pelos dados. Os dados por sua vez são representados como *fichas*. De acordo com a perspectiva, uma ficha pode ser um resultado ou um argumento. Um grafo deste tipo é ilustrado na Figura 2.1. Estes grafos, chamados grafos de fluxo de dados, são interpretados diretamente pelo *hardware* dos computadores a fluxo de dados, com base na equivalência entre as operações (vértices) do grafo e as instruções ou operações da máquina. Assim sendo, no nível de abstração mais baixo, o grafo de um programa será constituido da linguagem de máquina.

Nesta forma gráfica, o paralelismo é visualmente aparente quando as operações dependentes são colocadas umas sobre as outras, enquanto as operações independentes, lado a lado. As operações que estiverem no mesmo nível do grafo podem ser executadas simultaneamente.

São duas as abordagens com respeito ao compartilhamento do código de grafos de fluxo de dados [43]: *estática* e *dinâmica*. A diferença entre elas

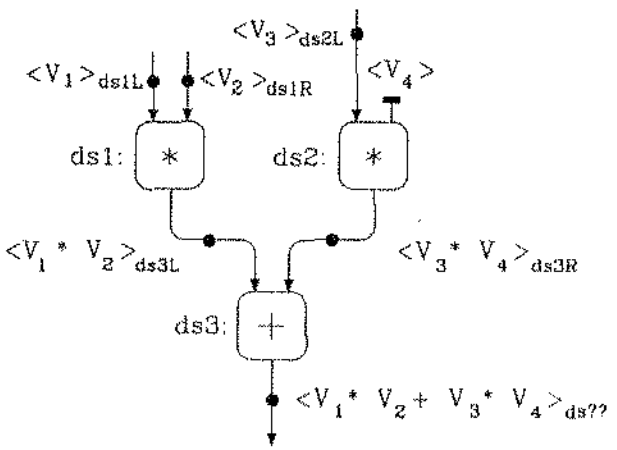


Figura 2.1: Exemplo de um grafo de fluxo de dados. O vértice em $ds2$ tem V_4 como um literal.

reside em que, no modelo de fluxo de dados *dinâmico*, o código pode ser usado de forma reentrante, isto é, o mesmo código pode servir para mais de uma aplicação. O modelo *estático* não admite esta propriedade. O modelo dinâmico, por sua vez, apresenta uma divisão subseqüente no que concerne à forma do compartilhamento do código entre aplicações concorrentes:

- O modelo *rotulado* (*tagged*) acrescenta a cada ficha um rótulo (ou contexto) que inclui um *nome de ativação*, identificando a ativação do procedimento do qual ela faz parte. Para estender o compartilhamento a modalidades de controle iterativas, costuma-se associar também um campo de *índice* a cada ficha. O rótulo é composto por estas duas descrições, nome de ativação e índice.
- O modelo *por cópia* realiza o compartilhamento mediante a duplicação do código em múltiplas cópias. O destino das fichas é alterado dinamicamente segundo a ativação.

Este trabalho dirige-se exclusivamente à forma rotulado do modelo de fluxo de dados dinâmico.

2.2.1 Adequação de Grafos à Arquitetura

As instruções básicas executadas pelo *hardware* de um computador a fluxo de dados são da mesma natureza das instruções básicas executadas num computador convencional: existem instruções lógicas, aritméticas e de controle. Contudo, de modo a seguir o modelo Fluxo de Dados uma instrução não preserva seus argumentos, e o resultado que ela produz tem um destino estabelecido explicitamente, quer pela instrução, quer pelo argumento, ou quer por uma combinação desses dois. Não existe uma associação pré-definida entre um nome e uma posição de memória, exceto excepcionalmente.[1] Em decorrência destas propriedades, um programa em linguagem de máquina de um computador a fluxo de dados pode ser representado sob a forma de um grafo orientado como explicamos anteriormente.

A execução de uma instrução ocorre a partir da disponibilidade dos seus argumentos, sendo apenas necessário determinar o momento em que todos os argumentos estão definidos. Desta forma, a formação do conjunto de argumentos tem de ser sincronizada pelo *hardware*. Uma das limitações geralmente impostas sobre os grafos pela arquitetura atual dos computadores a fluxo de dados é de possuírem no máximo dois argumentos por instrução, porque, sob o aspecto da engenharia destes computadores, é caro e complexo fazer com que o *hardware* sincronize um número maior de argumentos. Quanto ao número de destinos dos resultados das instruções, existe maior flexibilidade nas opções de engenharia disponíveis e há propostas que sugerem um número arbitrário de destinos [15].

2.3 Tipos de Paralelismo

Na seção anterior mencionamos que os programas funcionais podem apresentar um alto grau de paralelismo na execução de instruções. A seguir, realizamos uma discussão de algumas formas de aproveitamento desse paralelismo.

2.3.1 Paralelismo Temporal (*Pipelining*)

A realização de algumas *atividades repetitivas* (operações idênticas em vetores, por exemplo) pode ser adaptada para extrair esta forma de paralelismo.

[1] A exceção à regra é o caso de estruturas de dados explicado no próximo capítulo.

A técnica consiste em dividir uma atividade em sub-atividades independentes porém logicamente seriadas. As sub-atividades são realizadas consecutivamente e cada uma tem uma duração que é uma fração do ciclo de repetição.

Fisicamente isto se implementa numa organização de unidades físicas (estágios) dispostas ao longo de uma linha (*pipeline*), onde cada estágio se ocupa de uma sub-atividade. Quando em regime, o sistema pode executar concorrentemente um número de atividades igual ao número de estágios, onde cada estágio se ocupa da sub-atividade associada a uma repetição (ciclo) diferente. O número de ciclos necessário para atingir-se o estado de regime do *pipeline* é conhecido por tempo de latência.

O efeito da técnica de *pipelining* é a redução do tempo de execução da atividade, quando o sistema atinge o regime, na proporção do número de estágios em que ela foi dividida.

2.3.2 Paralelismo Espacial

O paralelismo espacial caracteriza-se pela presença de diversas atividades *do mesmo tipo, em locais distintos* do computador. O seu aproveitamento decorre da replicação das unidades básicas do sistema em diferentes lugares. A principal limitação do emprego deste paralelismo provém da comunicação que é necessária entre as unidades para troca de informação. As vias de comunicação estabelecem, a partir da topologia adotada, um compromisso na capacidade de transferência de informação entre a quantidade de informação trocada e o tempo de transferência.

2.3.3 Paralelismo na Execução de Grafos de Fluxo de Dados

Nos computadores paralelos o tamanho das tarefas executadas de modo indivisível pelos processadores é conhecido como *granularidade*. No modelo Fluxo de Dados a granularidade é pequena ou fina porque o tamanho básico da tarefa é uma instrução. Como foi mencionado anteriormente, cada uma das instruções está associada a um vértice do grafo do programa.

A execução das tarefas pode ser logicamente agrupada em processos e estes associados a procedimentos no programa de alto nível. De modo inteiramente análogo aos vértices do grafo de programa, esses processos comportam-se como produtores e consumidores de informação. Processos, contudo,

podem exibir comportamento produtor e consumidor alternada ou simultaneamente.

O paralelismo de um grafo é uma medida do paralelismo implícito no *software* que pode tanto simular algo parecido ao paralelismo temporal (por meio de diversas ativações de um grafo correspondendo a uma construção repetitiva da linguagem, ou por processos produtor-consumidor) como algo parecido ao paralelismo espacial (por meio da execução simultânea de instruções que se encontram no mesmo nível do grafo).

2.4 Organização de um Computador a Fluxo de Dados

A organização dos computadores a fluxo de dados varia muito [44]. Entretanto, apesar das diferenças, algumas características de arquitetura são comuns a todos esses computadores. Uma das principais características é a natureza distribuída e assíncrona do processamento: as unidades que compõem o computador a fluxo de dados são funcionalmente independentes e por isso podem estar *fracamente acopladas* (*loosely coupled*).

As principais funções necessárias para o funcionamento de um computador a fluxo de dados são [43]:

1. o armazenamento de fichas;
2. o agrupamento (sincronização) das fichas-argumento destinadas a uma instrução;
3. o armazenamento das instruções do programa;
4. a busca do código de operação de uma instrução;
5. a execução de uma instrução;
6. o armazenamento das estruturas de dados.

As unidades que compõem um computador a fluxo de dados são discutidas a seguir bem como a implementação das funções. A topologia usualmente empregada dispõem as unidades ao longo de um anel. A comunicação entre elas realiza-se por meio de mensagens que fluem no anel, sempre no mesmo sentido. Este tipo de organização é adequado ao aproveitamento do paralelismo que pode existir na execução das atividades de cada unidade.

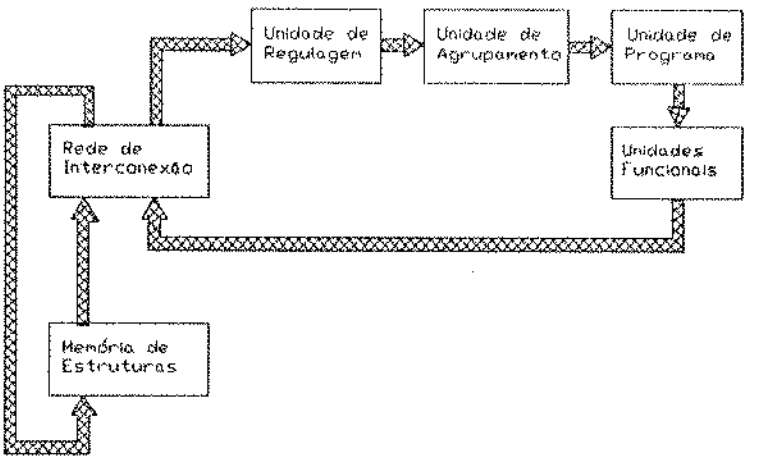


Figura 2.2: Organização do computador a fluxo de dados de Manchester

A Figura 2.2 expõe esquematicamente a organização de um computador a fluxo de dados.

2.4.1 Unidade de Regulagem (*Token Queue*)

Uma Unidade de Regulagem administra uma fila de mensagens entre quaisquer duas outras unidades do computador onde a taxa de comunicação de dados possa variar muito. Ela desempenha o papel de controlador do fluxo de mensagens entre duas unidades. Na maioria das arquiteturas existentes trata-se de uma unidade de arquitetura simples em relação às demais.

2.4.2 Unidade de Agrupamento (*Matching Store*)

A Unidade de Agrupamento é uma das que caracteriza melhor a arquitetura de um computador a fluxo de dados dinâmico. Ela realiza o agrupamento e armazenamento de fichas, ou seja, a sincronização dos eventos necessários à execução de uma instrução. Operações no grafo do programa com um único argumento naturalmente não requerem agrupamento ou armazenamento. Por isso uma ficha destinada a uma operação de um argumento não necessita passar pela Unidade de Agrupamento.

Fichas destinadas a uma operação de dois argumentos requerem sincro-

nização. Ao receber uma ficha deste tipo, a Unidade de Agrupamento certifica-se se o outro argumento está presente. Em caso afirmativo, a ficha é removida e o par colocado no anel; do contrário, a ficha é armazenada.

Nos computadores a fluxo de dados rotulados esta memória costuma ser endereçada pelo conteúdo, geralmente de modo pseudo-associativo (por exemplo, através do *hashing* do rótulo e do destino da ficha). Por isso, a Unidade de Agrupamento torna-se uma memória cara e limitada em espaço.

2.4.3 Unidade de Programa (*Instruction Store*)

Antes do início da execução, as instruções do programa são carregadas para a Unidade de Programa. Durante a execução, a unidade realiza a busca (*fetch*) de instruções a partir da chegada dos seus argumentos. Cada instrução leva consigo explicitamente os endereços para onde se destinam os resultados.

Em alguns computadores a fluxo de dados, como o computador de Manchester, a Unidade de Programa situa-se depois da Unidade de Agrupamento no anel. Entretanto, como a identificação da operação está contida nas ficha-argumento, a Unidade de Programa também pode situar-se em paralelo com a Unidade de Agrupamento [42].

2.4.4 Unidade Funcional (*Functional Unit*)

A Unidade Funcional realiza operações lógicas, aritméticas e de controle sobre as fichas. Num computador a fluxo de dados dinâmico rotulado, as operações lógicas e aritméticas preservam o rótulo dos argumentos e o destino dos resultados. As operações de controle que modificam a seqüência de avaliação, são executadas alterando o rótulo ou determinando dinamicamente o destino das fichas de resultado.

2.4.5 Memória de Estruturas (*Structure Store*)

A Memória de Estruturas é a unidade onde são armazenadas as estruturas de dados. Ela distingue-se da memória de fichas presente na Unidade de Agrupamento pela sua forma de organização. Como será visto no próximo capítulo, esta memória se faz necessária para um processamento eficiente envolvendo estruturas de dados.

2.4.6 Rede de Interconexão

Num computador a fluxo de dados o paralelismo espacial pode ser explorado de dois modos: aumentando o número de Unidades Funcionais no anel; ou aumentando o número de anéis do sistema.

É comum adotar-se para a ligação das unidades em paralelo no anel o uso de uma via de comunicação em barramento; e para a ligação dos anéis entre si, uma rede de interconexão. Outros tipos de vias de comunicação entre anéis não se apresentam tão favoráveis. A ligação por meio de barramento, apesar de simples e barata, é quase impraticável para um número elevado de anéis devido à contenção que ocorre no direito de uso do barramento. A ligação por meio de um *crossbar*, embora flexível, é de custo e complexidade proibitivos para um número elevado de anéis.

Algumas organizações ligam as Memórias de Estruturas aos anéis por meio da rede de interconexão [7,26], enquanto outras a colocam em paralelo ao anel [9,42].

Capítulo 3

Conceitos Relacionados a Estruturas de Dados

Este capítulo apresenta diversos conceitos relacionados à implementação de estruturas de dados em computadores a fluxo de dados. O ramo da teoria de fluxo de dados envolvendo estruturas encontra-se em fase de consolidação. Geralmente os conceitos são introduzidos na literatura de modo informal e com o tempo são corrigidos e formalizados.

3.1 Definições

Iniciamos com algumas definições elementares que apesar de intuitivas podem ajudar o leitor.

Definições 1 *Um* escalar *é uma unidade básica de informação. Um escalar tem um* tipo *que determina a sua representação. Uma* estrutura de dados *é uma coleção de escalares. Uma estrutura de dados tem um* tipo *que determina a forma como os escalares nela se encontram ordenados, isto é, logicamente organizados. Chama-se* representação *a forma da organização física da estrutura de dados.*

Uma característica das estruturas de dados é que existem dois modos distintos de serem encaradas:

1. como um conjunto de elementos, onde cada elemento pode ser uma outra estrutura ou um escalar, dependendo da representação escolhida;
2. como uma função de escalares para escalares, ou de escalares para outras funções.

Definições 2 *Uma estrutura é* homogênea *quando tem uma representação onde todos os elementos têm o mesmo tipo. Um* array *é uma estrutura homogênea. Um* vetor *é um array onde todos os elementos são escalares. Um* registro*(record) é uma estrutura não homogênea.*

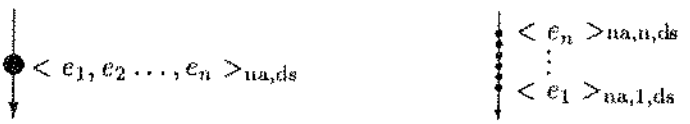
Uma estrutura de dados com um determinado tipo pode ter uma organização lógica diferente da organização física. Neste caso a estrutura e o conjunto operações sobre ela constituem um objeto chamado *tipo abstrato de dados.* A construção de tipos abstratos costuma ser feita explicitamente na linguagem, embora possam também surgir na tradução do alto nível da linguagem para o nível mais baixo em conseqüência de existir uma única representação para estruturas.

A representação de uma estrutura de dados é muito importante pois repercute sobre a forma e o tempo de acesso a seus elementos. Existem compromissos entre representação, espaço de armazenamento e tempo de acesso a elementos de uma estrutura. No restante do capítulo dirigiremos a discussão para as formas de representação de estruturas tendo em vista esses compromissos.

3.2 Estruturas Não-Armazenadas

No modelo de fluxo de dados as fichas são o veículo de transporte para os dados. Para dados escalares, uma ficha é suficiente para transportar o dado. Para dados estruturados, para que uma ficha fosse bastante seu tamanho teria de ser variável (o que repercutiria na organização da Unidade de Agrupamento). Além disso, a formação de uma ficha de dados estruturados seria difícil de realizar-se de uma *única vez* devido à dificuldade de sincronização dos elementos constituintes.

O modo natural de representar uma estrutura de dados num computador a fluxo de dados é como um conjunto de escalares, onde cada escalar possui uma identificação que permite localizá-lo dentro da estrutura. Uma forma conveniente de fazer isto é empregando-se o campo de índice no rótulo da ficha (veja a seção 2.2), conforme mostra a Figura 3.1. Na Figura 3.1 o campo Nome de Ativação (*na*) identifica a ativação do procedimento da qual a estrutura é parte e o campo Destino (*ds*) a operação a que ela se destina; o Índice (*ix*) varia de 1 a n elementos.



ficha ::= <dado><rótulo><destino>
rótulo ::= <nome de ativação><índice>
estrutura ={ < e_{ix} >na,ix,ds | ix = 1..n }

Figura 3.1: Representação de uma estrutura por um conjunto de escalares empregando uma única ficha e um conjunto de fichas.

No contexto dos computadores a fluxo de dados, esta forma de representação é chamada *stream*. Diz-se que ela é *não-armazenada* porque não emprega a Memória de Estruturas.

Ocorrem diversas restrições associadas à representação de estruturas não-armazenadas:

- É necessário um outro campo de índice na ficha para permitir que estruturas não-armazenadas estejam presentes dentro de iterações. Se a ficha tiver um único campo de índice, estruturas de dados não podem estar presentes nos trechos de código que correspondem à tradução de construções iterativas da linguagem (*do*, *for*, *while*, etc.), uma vez que o único campo então existente é usado para identificar o nível de iteração em que a ficha se encontra.
- A representação não-armazenada pode ser adequada para estruturas unidimensionais, mas estruturas multidimensionais requerem o mesmo número de campos de índice que a dimensão da estrutura tratada [46]. A implementação é complexa para um número variável de campos e muito cara para um número fixo de campos. Admitindo-se que o rótulo pudesse ter seu formato adaptado ao número e comprimento destes campos, ainda assim seria necessário mais um campo no rótulo para especificar o formato.
- A representação não-armazenada é mais adequada a estruturas de da-

dos homogêneas onde as arestas do grafo têm tipos determinados pelas operações que residem nos seus vértices. Para tratar estruturas não-homogêneas seria necessária a criação de operações para separar tipos.

- O rótulo de uma ficha tem um formato de comprimento limitado impondo uma limitação no tamanho das estruturas representáveis.
- Estruturas não-armazenadas incorrem no custo do armazenamento e transporte do rótulo da ficha para cada elemento da estrutura, sendo que apenas o campo de índice é diferente entre os rótulos dos elementos.

A maior virtude da representação de estruturas não-armazenadas está no paralelismo que pode ser explorado durante o processamento. Nessas estruturas, *os elementos são sincronizados individualmente entre processos produtor e consumidor*, ou seja, todos os elementos podem ser processados quase simultaneamente, pois são tratados como escalares.

Apesar do paralelismo e além das restrições já mencionadas, existe um problema mais grave ocasionado pela cópia de estruturas. Pela semântica do modelo Fluxo de Dados, operações não preservam os seus argumentos. Usando-se estruturas não-armazenadas, *uma estrutura terá de ser copiada tantas vezes quantas forem as operações sobre ela.* A conseqüência é que a ocupação da Unidade de Agrupamento aumenta substancialmente, acentuando-se também o tráfego de fichas entre as unidades do computador. Quando as operações realizadas são apenas de consulta parcial sobre a estrutura de dados, observa-se um consumo excessivo de recursos do computador.

3.3 Estruturas Armazenadas

Para contornar as deficiências das estruturas não-armazenadas desenvolveu-se o conceito de armazenamento para estruturas de dados. Assim, a *estrutura* passa a possuir um caráter de objeto passível de armazenamento numa unidade específica, a Memória de Estruturas.

Num grafo de fluxo de dados que suporta estruturas armazenadas, a estrutura propriamente dita é que é representada como uma ficha. Na ficha, consta o nome da estrutura, em geral um apontador para o local da memória onde ela se encontra armazenada. Para acesso ao elemento de uma estrutura

realiza-se uma operação que tem como argumentos o nome da estrutura e um seletor. Para o armazenamento de um dado numa estrutura, por sua vez, são necessários três argumentos: o nome da estrutura, o seletor e o próprio elemento. O armazenamento pode ser implementado através de duas operações de dois argumentos, ou através de uma única operação empregando-se um artifício que explicaremos posteriormente no Capítulo 6.

As estruturas armazenadas não têm a natureza dinâmica das estruturas não-armazenadas. Uma estrutura armazenada transforma-se num objeto estático. Com isto *incorre-se no custo de pelo menos duas operações extras para cada elemento*: uma para o armazenamento e outra para uma consulta.

Como conseqüência do armazenamento de uma estrutura de dados tem-se basicamente:

1. O espaço empregado para o armazenamento da estrutura é usado com eficiência se os processos que a consomem realizam apenas consultas parciais sobre ela. Nesse caso as vias de comunicação são empregadas apenas conforme o necessário.
2. O armazenamento de estruturas de dados não-homogêneas e de estruturas aninhadas (como listas e árvores, por exemplo) é de execução mais simples a partir da associação existente entre a estrutura e o seu apontador, ou seja, uma estrutura pode conter outras estruturas.
3. Torna-se viável o tratamento de problemas com estruturas de muitos elementos, uma vez que fica superado o limite imposto pelo comprimento fixo do rótulo.

Na maioria dos casos é impraticável a determinação *a priori* do consumo de memória de um programa funcional. Para tornar eficiente o uso da Memória de Estruturas e evitar o consumo excessivo de espaço, inclui-se no sistema, como um processo autônomo, um gerente responsável pelo controle da memória. Faz-se necessária a gerência dinâmica do espaço de memória para reaproveitamento de posições fora de uso durante a execução de um programa, ainda que isto aumente o custo do processamento e a complexidade do sistema. O processo de gerência pode ser conduzido por *software*, *hardware* ou uma combinação de ambos.

3.3.1 Armazenamento Empregando Árvores

A primeira proposta de um armazenamento específico para estruturas de dados em computadores a fluxo de dados foi feita por Dennis [19]. Num ambiente de estruturas armazenadas, como o proposto por ele, a organização interna é sob a forma de árvore, onde os elementos da estrutura situam-se no lugar das folhas. De modo a suportar adequadamente o armazenamento, a Memória de Estruturas é constituída por células de tamanho fixo. A idéia possui analogia com a forma de armazenamento de dados usada em LISP.

Como mencionamos anteriormente, a ausência de efeitos colaterais em ambientes funcionais proíbe alterações em um objeto: toda a transformação de um objeto origina um objeto novo. No entanto, pela representação em árvore, objetos logicamente distintos podem ser fisicamente compartilhados evitando com isto o esforço da cópia dos elementos comuns. A estrutura formada por um conjunto de objetos compartilhando elementos resulta num grafo dirigido e acíclico.

Para determinar quando uma parte da estrutura pode ser descartada, por encontrar-se fora de uso, cada vértice da árvore possui um contador de referências indicando o número de estruturas que o compartilham. Se o contador de referências for maior do que 1 o vértice está sendo compartilhado por um número de estruturas igual a esse valor. Se o contador for 1 o vértice não está sendo compartilhado.

A implementação de estruturas no modo proposto por Dennis é difícil devido à possível ordem (generalidade) das árvores. Por isso Ackerman [1] propôs restringir as árvores ao caso de árvores binárias como em LISP. A argumentação a este favor baseia-se em que, restringindo a generalidade das árvores, a implementação se simplifica. Na proposta de Ackerman, uma estrutura é percorrida usando um seletor formado por uma cadeia de bits. Um bit '1' no seletor determina a escolha da sub-árvore da esquerda, enquanto que um bit '0', a sub-árvore da direita. Um objeto é inserido (*appended*) ou lido (*selected*) a partir da posição estabelecida pelo seletor conforme ilustrado a seguir:

operação	funcionalidade
append (e_1,s,e_2)	árvore_binária, cadeia_bits, árvore_binária → árvore_binária
select (e,s)	árvore_binária, cadeia_bits → árvore_binária

A formação de estruturas com base na operação de *append* é um processo moroso. Além disto, o processo requer uma serialização uma vez que ele depende da existência dos vértices da árvore para que *appends* a esses vértices sejam efetuados. Esta serialização na transformação de estruturas é responsável, em parte, pelas críticas atribuídas a este tipo de representação [23]. Por outro lado, *appends* concorrentes podem dar origem a uma estrutura de dados diferente da desejada, se houver uma alteração imprópria na sua seqüência lógica. Num ambiente concorrente como o de fluxo de dados, esta é uma limitação severa. Outra conseqüência de *appends* concorrentes são os diversos acessos a uma estrutura que podem ocorrer simultaneamente. No caso de estruturas com um número grande de elementos, a raiz da árvore torna-se, sem dúvida, uma posição de memória com alta probabilidade de conflitos de acesso (*hot spot*), uma vez que todos os acessos às folhas da árvore passam necessariamente por ela.

Gaudiot [25] sugeriu a criação de macro-atores, ou seja, macro-instruções responsáveis pela manipulação das estruturas de dados. Estes macro-atores, introduzidos no grafo do programa pelo compilador, realizam a produção e a transformação das estruturas de dados diretamente, evitando a formação na memória de versões intermediárias de uma estrutura. Embora os macro-atores pareçam necessários, eles introduzem nos programas operações de granularidade variada e de alocação explícita de processadores, justamente num modelo onde granularidade fina e alocação dinâmica de processadores são atributos básicos.

Uma crítica mais séria, mencionada por Gajski [23] está na contagem de referências em cada vértice de uma árvore. Ao se criar uma estrutura compartilhada, todos os vértices comuns têm de ser atualizados, o que aumenta muito o custo do controle da memória. Quando a contagem de referências de um vértice chega a zero, por exemplo, é necessário iniciar um processo recursivo de exame dos vértices que são referidos pelo vértice descartado. Estes, por sua vez, têm a sua contagem de referências decrementada podendo ir a zero. Nos casos onde isto ocorrer, sucede um reexame nos vértices dependentes.

Finalmente, as estruturas em árvores compartilhadas não permitem estabelecer com certeza se o espaço economizado no compartilhamento compensa o espaço extra usado por vértices intermediários (isto é, vértices compostos

por estruturas) e por contadores de referências, já que em inúmeros casos o compartilhamento é desnecessário. A representação de escalares de tipos distintos, literais e numéricos, por exemplo, também é uma causa de desperdício na ocupação de memórias com células de tamanho fixo, pois o tamanho da célula é estabelecido a partir do escalar mais demandante de espaço.

3.3.2 Armazenamento Empregando Blocos

Uma representação interna empregando árvores parece adequada apenas quando o tipo de estrutura tratado no programa é também uma árvore. O uso de uma representação de árvore para um *array* é muito ineficiente devido aos inúmeros problemas mencionados anteriormente. Para armazenar *arrays*, a organização mais conveniente é um espaço contíguo de memória (bloco). Neste caso, o apontador é um endereço-base que determina onde a estrutura começa e o índice é empregado no cálculo do deslocamento até a posição do elemento desejado. O maior benefício que se obtém com este tipo de representação é um tempo de acesso constante aos elementos, neste caso determinado unicamente pelo *hardware* da máquina, independendo do tamanho da estrutura.[1]

Uma vez armazenada, uma estrutura pode ser compartilhada nos acessos para leituras, copiando o seu apontador. Entretanto, o compartilhamento de versões diferentes de uma estrutura é muito limitado em razão da sua complexidade. Quando não é feito compartilhamento de estruturas, a formação de uma nova estrutura leva um tempo proporcional ao tamanho da estrutura original porque a maioria dos elementos têm de ser copiados de uma estrutura para outra. Quanto à contagem de referências, esta é feita para toda a estrutura, ao invés de para cada um de seus elementos, como na representação por árvores, simplificando e diminuindo o custo deste processo.

O problema da representação por blocos está na gerência da memória. Como os blocos têm tamanho variável, surgem dificuldades nos processos de alocação e recuperação de memória, destacando a fragmentação de memória, a gerência de espaço livre e o tempo de alocação de espaço.

[1] Consideramos que a estrutura seja *estrita* (um conceito que será apresentado na seção 3.4.1).

Os computadores a fluxo de dados têm sido dirigidos principalmente ao processamento numérico onde as estruturas básicas empregadas são do tipo *array*. As linhas de pesquisa em fluxo de dados vêm adotando o armazenamento de estruturas empregando blocos. A exceção é a linha estática seguida por Dennis [19].

Compartilhamento de Estruturas Empregando Blocos

Uma das deficiências do armazenamento de estruturas empregando blocos está na dificuldade de compartilhar elementos comuns entre as estruturas, o que pode elevar o consumo de memória em certos problemas. Considere-se o exemplo em SISAL:

```
type intvec = array [integer]

function swap (a: intvec; i,j: integer  % a [i: v] e uma expressao
               returns intvec)          % que cria uma estrutura
                                        % com os mesmos elementos de
   a [i: a[j]; j: a[i]]                 % a, exceto na posicao a [i]
                                        % que tem o valor v
end function

function selectionSort (a: intvec; k: integer
                        returns intvec)
   if k = 1
      then a
      else selectionSort (swap (a, imax (a, k), k), k - 1)
   end if
            % imax determina a posicao do maior elemento
end function % em 'a' entre os indices 1 e k
```

Neste exemplo, o número de estruturas criadas é igual ao número de elementos do vetor, independendo da ordenação dos elementos da estrutura a inicial. Como o algorítimo é intrinsecamente seqüencial (basicamente devido à recursão de cauda), observando que as versões intermediárias de a serão descartadas, poder-se-ia aproveitar a versão antiga de a e *reatribuir* valores a ela na função swap. A própria função swap pode ser melhorada para realizar a alteração em a apenas quando necessário:

```
if i = j
   then a
   else a [i:  a[j]; j:  a[i]]
end if
```

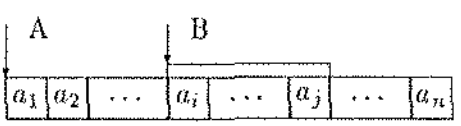
O exemplo ilustra de modo muito evidente que, para alguns problemas, uma das melhores formas de evitar o consumo excessivo de memória é reaproveitar o espaço de uma estrutura-argumento reatribuindo elementos nas posições alteradas. Isto pode ser feito se não existirem outras referências (isto é, apontadores) para a estrutura. Um compilador pode introduzir dependências, de modo que a cada momento durante a execução do programa esta condição venha a ser satisfeita. As dependências introduzidas pela compilação são sincronizações que serializam as operações sobre a estrutura para permitir a reatribuição. Apesar de uma solução deste tipo ser simples e barata, ela não pode ser aplicada a qualquer caso por restringir o paralelismo.

O compartilhamento de estruturas empregando blocos pode ser mais difícil do que aquele empregando árvores por serem maiores as restrições provenientes das operações implementadas com o propósito do compartilhamento. Neste caso, o número de versões compartilhadas em blocos deve, em geral, ser menor do que em árvores.

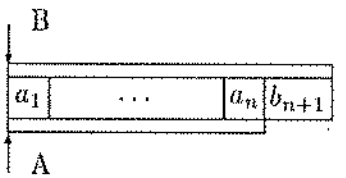
A seguir são descritas três das propostas existentes para compartilhamento empregando blocos.

Compartilhamento de Segmentos A forma realizada no computador a fluxo de dados de Manchester implementa compartilhamento de um segmento de um bloco. As operações *array_adjust*, *array_addh* e *array_addl* presentes em SISAL [33], dependendo do contexto, podem permitir o compartilhamento. Nos esquemas a seguir A e B são estruturas:

- *array_adjust* cria uma estrutura B que é parte de uma outra A.

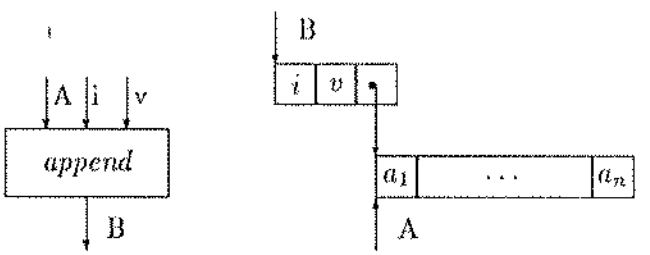


- *array_addh* cria uma estrutura B a partir de outra A, acrescentando-lhe um elemento a mais na última posição. *array_addl* é a operação recíproca a *array_addh*.



As operações que produzem compartilhamento na forma de segmentos incrementam a contagem de referências sobre o espaço compartilhado. O efeito é quase equivalente à cópia do descritor da estrutura original.

Compartilhamento Empregando Nós Esta forma de compartilhamento é proposta em EXMAN [38]. Uma operação de *append* (do tipo a [i: v]) cria B a partir de A:



No acesso aos elementos de B, o seletor é inicialmente testado para determinar se este está presente no nó. Se estiver ausente, a estrutura A é consultada. *appends* sucessivos podem originar uma lista, que neste caso é percorrida em busca do primeiro seletor igual ao da consulta. De modo a manter correto o número de referências feitos à estrutura, a estrutura-argumento de um *append* tem a contagem de referências incrementada.

Compartilhamento Empregando Estruturas Híbridas O objetivo das estruturas híbridas [28] está em alcançar um equilíbrio entre a necessidade de múltiplas cópias e o compartilhamento de uma cópia única de uma estrutura. A idéia principal consiste em criar um descritor (Figura 3.2) que estabelece onde se encontra o elemento que deve ser lido. Este descritor também é armazenado, sendo constituído por cinco campos:

- o status, que indica se o descritor refere-se a uma estrutura modificada;
- a contagem de referências, que determina o número de apontadores para o descritor;
- a localização do elemento, campo constituído por um conjunto de bits, um bit associado a cada elemento; e
- os apontadores esquerdo e direito, que são empregados para acesso aos elementos.

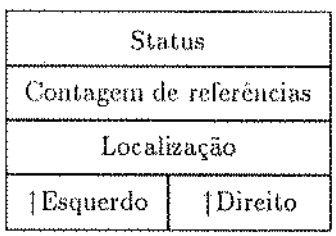


Figura 3.2: Descritor de uma estrutura híbrida

Na criação de uma estrutura, aloca-se espaço para o descritor e para os elementos. O descritor usa o apontador esquerdo para determinar onde a estrutura inicia. O apontador endereça um bloco ocupando o espaço correspondente ao tamanho da estrutura. O valor '0' é atribuído a todos os bits do campo de localização como indicação de que se deve usar o apontador esquerdo para acessos. Ao apontador direito atribui-se o valor nulo, à contagem de referências o valor '1' e ao status a indicação '*normal*'. Estes procedimentos criam a estrutura.

Uma operação de *append* numa estrutura híbrida cria um novo descritor e aloca o espaço correspondente a uma nova estrutura. O novo descritor tem o status de '*alterado*' como indicação de que a estrutura antiga é agora compartilhada. O apontador esquerdo do novo descritor aponta para o descritor da estrutura antiga. O apontador direito aponta para o novo espaço alocado e o elemento, argumento do *append*, é inserido na posição adequada. Ao bit de localização correspondente à posição do novo elemento é atribuído o valor '1' indicando que para acesso a este elemento deve ser empregado o apontador direito do descritor. Os outros elementos são consultados através do apontador esquerdo e do descritor antigo. A contagem de referências do

descritor antigo é incrementada devido à referência pelo apontador esquerdo do novo descritor.

No caso em que a contagem de referências da estrutura é 1 pode-se realizar diretamente uma alteração, pois sabe-se que a estrutura-argumento será descartada.

Para evitar o desperdício de espaço em estruturas híbridas, ao invés de alocar-se área para um bloco do tamanho da estrutura, alocam-se sub-blocos de tamanho fixo a partir da necessidade. Isto otimiza o uso de espaço, mas requer uma tabela de acesso para endereçar os sub-blocos. Como os sub-blocos podem ser compartilhados, é necessária uma contagem de referências associada a cada um.

Apesar de os dados de simulação indicados na literatura [28] parecerem favoráveis, a representação híbrida é bastante complexa e requer um suporte de *hardware* sofisticado para sua implementação. Em qualquer situação, o descritor de uma estrutura grande será um *hot spot*, uma séria desvantagem. Ao mesmo tempo, tudo indica que a representação híbrida comporta-se melhor para estruturas grandes, pois o tempo de formação das estruturas derivadas é menor do que o tempo de uma cópia integral. Isto pode ser uma vantagem.

A presença de descritores intermediários para estruturas que sofreram diversas modificações introduz uma latência elevada no acesso aos elementos na estrutura original. Naturalmente esta latência pode ser amenizada por meio de *pipelining* e, possivelmente, pela formação mais rápida de uma estrutura derivada.

3.4 Paralelismo e Estruturas de Dados

Quando se imagina paralelismo envolvendo estruturas de dados, freqüentemente pensa-se em aplicações envolvendo operações vetoriais e equações de recorrência. No entanto, esses problemas constituem apenas um subconjunto das possíveis aplicações, apresentando um paralelismo familiar e bem comportado, explorado por processadores vetoriais. Os computadores a fluxo de dados não se limitam exclusivamente a problemas deste tipo.

O aspecto mais importante do tema paralelismo com estruturas de dados está em como se realiza a sincronização entre os processos produtor e consumidor. Num ambiente paralelo, produtor e consumidor podem ser processos independentes constituídos de vários sub-processos. Quanto menor o número de sincronizações entre o produtor e o consumidor, mais eficiente o processamento. Contudo é difícil obter-se uma solução abrangente e eficiente para todos os tipos de problemas.

No modelo Fluxo de Dados dinâmico, todas as estruturas não-armazenadas são representadas sob a forma de *streams*. Sob o aspecto de paralelismo, *streams* são convenientes por serem estruturas *dinâmicas*. Entretanto, como foi visto anteriormente, uma abordagem que adota exclusivamente *streams* é pouco praticável. Geralmente uma estrutura pode ser representada como um *stream*, apresentando vantagens, quando atendidas as seguintes condições:

- os elementos da estrutura são escalares;
- o padrão de acesso para a formação e o consumo dos elementos é conhecido;
- existem poucos consumidores ou, se existirem muitos, eles usam a maioria dos elementos da estrutura.

Quando as condições anteriores não podem ser confirmadas convém armazenar a estrutura. Neste caso, a representação por meio de *arrays* é a mais empregada. Ao armazenar uma estrutura, perde-se em dinamismo e aumenta-se a *estaticidade*, característica decorrente da falta de mobilidade dos elementos da estrutura. A seguir, abordaremos uma forma de compensar a *estaticidade* por meio de *arrays* não-estritos.

3.4.1 Estruturas Estritas e Não-Estritas

Define-se uma função como sendo *não-estrita* (*non-strict*) sobre um argumento quando existem situações onde o resultado da função independe desse argumento. Em fluxo de dados a definição de função não-estrita leva em conta primeiramente o aspecto operacional da avaliação. Uma função em fluxo de dados é não-estrita quando a sua avaliação se inicia a partir da disponibilidade de qualquer argumento. O conceito de estruturas não-estritas é semelhante, no que diz respeito à presença de valores, ao conceito de funções não-estritas. Isto significa que uma *estrutura não-estrita* está disponível ao processo consumidor antes da definição de todos os seus elementos. Ao

contrário das estruturas não-estritas, as *estruturas estritas* requerem a presença de todos os elementos para que seu uso tenha início.

Weng [47] foi o primeiro a empregar o conceito de estruturas não-estritas para *streams*. Empregando-se *streams* não-estritos, é possível que o produtor do *stream* "passe à frente" não aguardando a formação dos elementos mais demorados e que o consumidor o acompanhe neste processo. O consumidor não fica bloqueado aguardando a formação dos elementos indefinidos. Arvind [10,11] criou as estruturas-I (estruturas incrementais) como *arrays* não-estritos, a fim de obter paralelismo entre o produtor e o consumidor de um *array* de modo análogo ao de Weng para *streams*.

Duas questões a abordar sobre estruturas não-estritas são as relacionadas à sincronização entre processos produtor e consumidor e à recuperação de memória.

Características de Sincronização

Para implementar estruturas não-estritas armazenadas é necessária uma sincronização de *hardware* que impeça a leitura de elementos ausentes (indefinidos). Isto é feito por um *bit* associado a cada elemento da estrutura indicando a sua presença. Quando o espaço de uma estrutura é alocado aquelas posições de memória são marcadas com o estado ausente.

O acesso a estruturas não-estritas é caracterizado como *split-phase* [6]. Distinguem-se duas fases no acesso aos elementos da estrutura por parte do consumidor: a fase de requisição e a de resposta. O tempo de consulta geralmente varia porque, uma vez feita uma requisição, a resposta se obtém somente no momento em que o elemento endereçado estiver presente. Uma consulta é considerada antecipada, devendo ser suspensa se ocorrer antes de o respectivo elemento estar definido. Existem dois modos de tratar as consultas antecipadas:

- *busy-waiting*: neste caso as consultas serão recusadas e tentadas em algum momento futuro;
- *queueing*: as consultas a um elemento indefinido são colocadas em uma lista associada àquela posição de memória. Se houver consultas antecipadas, elas serão respondidas prontamente no momento da definição do elemento.

Para o caso de *busy-waiting*, não é necessário um *hardware* complexo. Entretanto, se o número de consultas antecipadas for elevado, o desempenho do computador pode degradar em decorrência do congestionamento das vias de comunicação pelas mensagens de consulta que competem com todas as demais. No caso de *queueing*, por sua vez, o suporte de *hardware* é consideravelmente mais complexo. É necessário um *bit* adicional associado a cada posição de memória para indicar a presença de consultas antecipadas e também uma gerência dinâmica do espaço associado às listas de consultas antecipadas.

Em estruturas não-estritas, a sincronização entre produtor e consumidor ocorre *elemento a elemento* de modo que não existem *hot-spots* decorrentes da sincronização. Em estruturas estritas, contudo, as duas fases do processamento – a fase da produção da estrutura e a fase do consumo – não se sobrepõem. Para determinar quando a fase de produção foi concluída, o número de elementos da estrutura deve ser conhecido e estes elementos devem ser contados ao longo da produção. Isto corresponde a uma *sincronização global* para a estrutura, conhecida por *collect*. Naturalmente a formação dos elementos da estrutura pode ocorrer em paralelo, mas a presença de um único contador será um *hot-spot*, quando estruturas grandes forem sincronizadas.

Recuperação de Memória

A recuperação do espaço de memória alocado a uma estrutura estrita é mais imediata do que a recuperação do mesmo espaço alocado para uma estrutura não-estrita. Uma vez tendo sido a estrutura estrita descartada, todos os seus elementos são descartados ao mesmo tempo. A recuperação de espaço para uma estrutura não-estrita, entretanto, é mais complexa, pois ela pode iniciar no momento em que todos os elementos requeridos pelo consumidor tenham sido lidos. A possível presença de elementos indefinidos exige a recuperação do espaço individual de cada um dos elementos. Se um elemento estiver ausente, é preciso aguardar sua definição para, em seguida, recuperar o espaço que ele ocupa. Do contrário, pode ocorrer um comportamento não determinístico, ocasionado pela reatribuição a um elemento, quando este espaço tiver sido alocado prematuramente a uma outra estrutura. Além da recuperação individual de suas posições, uma estrutura não-estrita exige também uma *sincronização global* das remoções de forma a determinar quando o processo de limpeza foi concluído. Esta sincronização

pode ser realizada junto com o processo de recuperação dos elementos individuais, de modo serial ou paralelo. A sincronização serial pode levar muito tempo para estruturas grandes, enquanto que a totalmente paralela seria indiscutivelmente muito cara pelo número de sincronizações requerido. A forma mais adequada parece ser uma combinação de ambas aquelas formas.

Comentários

Pela discussão anterior, torna-se claro que é necessária uma sincronização global qualquer que seja a estrutura. No caso das estruturas estritas durante a formação, com o propósito de indicar quando a estrutura foi concluída. No caso das estruturas não-estritas durante a recuperação de espaço, com o propósito de determinar que o espaço associado à estrutura encontra-se disponível e sobre ele não existe uma requisição de acesso pendente.

A não ser pelas listas de consultas antecipadas das estruturas não-estritas, os custos de sincronização de estruturas estritas e não-estritas parecem iguais. A vantagem das estruturas não-estritas sobre as estruturas estritas está em que, naquelas, o produtor e o consumidor trabalham simultaneamente e, conforme o problema envolvido, o tempo de processamento se reduz substancialmente. Todavia, existem circunstâncias em que esta característica transforma-se em uma desvantagem: se o produtor é lento em decorrência de um processamento longo e se o consumidor é ansioso, as listas de consultas antecipadas crescem proporcionalmente ao número de consultas. O número de listas cresce então em proporção à diferença entre as velocidades do produtor e do consumidor.

Os custos de *hardware* de estruturas estritas são menores do que os de estruturas não-estritas. Por isto, parece haver um compromisso na engenharia das Memórias de Estruturas entre a capacidade de armazenamento e a capacidade do paralelismo dos acessos.

3.4.2 Estruturas Preguiçosas

A idéia de uma estrutura com condições de conter "infinitos" elementos deu origem às estruturas ditas *preguiçosas* (*lazy data structures*). Num contexto onde se admitem estruturas preguiçosas, os elementos de uma estrutura são definidos a partir de um conjunto de funções. As estruturas preguiçosas têm este nome porque o produtor só realiza a avaliação de um elemento a

partir da sua *demanda* pelo consumidor. Naturalmente podem haver dependências entre os elementos (por exemplo, para uma estrutura definida recursivamente) e a demanda é propagada indiretamente a outros elementos.

As estruturas preguiçosas surgem naturalmente nas linguagens com semântica operacional preguiçosa (*lazy evaluation*), como conseqüência de passagem de argumentos de modo *call-by-need*. Nas linguagens com semântica operacional ansiosa (*eager evaluation*), como conseqüência da passagem de argumentos por valor (*call-by-value*), estruturas preguiçosas são formadas por meio de anotações no programa, isto é, são declaradas explicitamente. Na literatura, o conceito de estruturas preguiçosas aparece pela primeira vez no trabalho de Friedman [21,22].

Existem algumas vantagens importantes no emprego dessas estruturas:

- maior capacidade de expressão, do ponto de vista da linguagem;
- somente os elementos necessários são avaliados evitando processamento inútil;
- as funções que têm uma avaliação muito cara em termos de processamento podem empregar estruturas preguiçosas para "memorizar" avaliações anteriores. Antes de avaliar a função, verifica-se se o resultado para um certo argumento está disponível.

Em fluxo de dados, o desenvolvimento da pesquisa de estruturas preguiçosas foi feito no trabalho de Heller [27], que elaborou uma extensão do ambiente convencional de fluxo de dados – um ambiente ansioso – para suportar estruturas preguiçosas. A base do trabalho de Heller é um mecanismo de sincronização de baixo nível para estruturas-I. Na tradução do alto nível para o baixo nível associa-se a cada posição da estrutura um fechamento (*closure*) da função, chamado *thunk*. O *thunk* consite de duas partes: uma referência para o código da função e outra para o contexto onde se encontram as definições das suas variáveis livres, ou seja, o contexto de invocação para essa função.

O comportamento operacional de estruturas preguiçosas é muito semelhante ao de estruturas não-estritas armazenadas. A diferença básica entre eles está em que a chegada de uma requisição para a leitura de um elemento da estrutura inicia um processo que corresponde à avaliação do *thunk* para aquela

posição caso o elemento esteja ausente. Todas as requisições feitas durante a avaliação são armazenadas. Quando o elemento fica pronto ele é armazenado na posição adequada e as requisições são atendidas.

Não é difícil estender um ambiente com os mecanismos de sincronização de estruturas não-estritas para suportar estruturas preguiçosas. Talvez o maior problema sejam as conseqüências da extensão da linguagem de alto nível para a representação de estruturas preguiçosas. A semântica operacional presente em fluxo de dados é naturalmente ansiosa e, por este motivo, os elementos avaliados de forma preguiçosa têm de ser declarados explicitamente. Assim, perde-se algo da uniformidade e transparência que havia anteriormente.

3.5 Estruturas de Dados em Linguagens de Alto Nível

Nesta seção vamos apresentar duas linguagens, SISAL (*Streams and Iteration on a Single Assignment Language*) [33] e Id (Irvine dataflow) [34], para demonstrar como as estruturas de dados podem ser tratadas no alto nível de uma linguagem e vamos apresentar algumas das características da sua tradução para o nível mais baixo do computador. As linguagens Id e SISAL são funcionais na maioria das construções sintáticas e, por isso, comportam-se bem no processamento em fluxo de dados. A semântica operacional destas linguagens baseia-se num mecanismo de passagem de parâmetros por valor e resulta num tipo de avaliação ansiosa.

No que diz respeito a estruturas de dados, a abordagem difere em SISAL e Id. As estruturas de dados em SISAL são tratadas como valores. Id trata estruturas como valores, mas admite a possibilidade de considerar uma estrutura como um conjunto de posições (*slots*). SISAL fornece ao programador um conjunto de funções primitivas para a manipulação de estruturas, enquanto em Id o programador é responsável pela definição das funções básicas que desejar, implementando-as como abstrações. A diferença principal está em como estruturas são construídas, aspecto sob o qual Id permite uma flexibilidade maior.

O computador a fluxo de dados do MIT [9] adota Id como linguagem de programação. A compilação de programas em Id é feita de modo a imple-

mentar as estruturas de dados como estruturas não-estritas. Id# [27], uma extensão de Id que admite a definição de estruturas preguiçosas, também pode ser empregada como linguagem para este sistema. O computador de Manchester [26], entretanto, adota SISAL como linguagem de programação e a compilação de SISAL implementa estruturas de dados como estruturas estritas.

3.5.1 Estruturas de Dados em SISAL

SISAL é uma linguagem funcional da classe de *atribuição única*. A primeira versão de SISAL, concluída em 1984 [33], foi um esforço conjunto do Laboratório Nacional Lawrence Livermore (LLNL), da Universidade de Manchester, da Universidade de Colorado e da Digital Equipment Corporation (DEC). SISAL foi desenvolvida para uso em computadores paralelos [35] de modo a estabelecer uma plataforma para comparação entre estas máquinas [32]. Um programa em SISAL é primeiramente traduzido para um formato intermediário (denominado padrão IF1) sendo as compilações subseqüentes realizadas conforme o ambiente em questão.

Não se considera SISAL totalmente funcional. Suas funções são de primeira ordem, isto é, não têm o mesmo status dos valores e por isso não podem ser argumento ou resultado de outra função. Além disso, a construção *for* de SISAL não é realmente declarativa, uma vez que existe controle explícito sobre a sua forma de avaliação: existe uma forma iterativa (*for initial*) e outra paralela (*forall*).

SISAL admite dois tipos de dados estruturados: *arrays* e *streams*. O suporte a esses tipos é obtido mediante operadores específicos presentes na linguagem. Em geral *streams* são traduzidos para uma representação não-armazenada como vimos na seção 3.2. Contudo, quando um *stream* é composto de elementos de tipos estruturados, a conveniência de realizar o armazenamento dessa estrutura cabe ao compilador.

A construção de outros tipos estruturados, como *listas* e *árvores*, fica a cargo do programador. Logicamente, em termos de operações, uma lista é equivalente a um *stream*. Todavia, as listas em SISAL dão origem a uma representação armazenada. Na construção dos tipos lista e árvore, no exemplo da Figura 3.3, são empregados os tipos básicos *record* e *union*. A declaração *record* cria um tipo de estrutura constituído por um agregado de tipos. A

declaração *union* origina um tipo de estrutura que é uma união de tipos e que pode comportar-se como qualquer membro desta união. O acesso aos tipos integrantes de *records* e *unions* é realizado por meio de seletores (*tags*). Cabe ao programador definir as operações sobre as estruturas assim construídas, a partir de funções que tomarão como argumentos essas estruturas e os seus respectivos tipos. SISAL adota um estreito acoplamento

```
type eTipo = ...
type eLista = record [elemento: eTipo;
                      lista: union [vazia: null;   % lista vazia
                                    nlista: eLista ]]
type arvBin = union [vazia: null;
                     elemento: eTipo;
                     vertice: record [esquerdo, direito: arvBin]]
```

Figura 3.3: Construção dos tipos de estrutura lista e árvore em SISAL

```
type intvec = array [integer]
type intmat = array [intvec]
type intarray_3d = array [intmat]
```

Figura 3.4: Exemplo de um tipo para um array tridimensional

entre estruturas de dados e estruturas de controle: para cada tipo básico estruturado existe uma estrutura de controle associada, sendo o caso mais importante o relativo a *arrays*. O caso de *streams* é semelhante. A idéia de mapear o nível de iteração da construção *for* para o índice da estrutura tratada é empregada extensamente, como ilustrado na Figura 3.5. Este tipo de mapeamento é natural, tendo sido adotado pela primeira vez em FORTRAN.

SISAL permite a criação de *arrays* multidimensionais a partir da construção de *arrays* com um número menor de dimensões. Deste modo, *arrays* multidimensionais têm por base *arrays* unidimensionais como ilustrado na Figura 3.4. Para tratar uma estrutura multidimensional é necessário empregar a construção *for* na forma iterativa, ou usar a forma produto repetida e aninhadamente sobre cada dimensão da estrutura.

Quando o mapeamento entre iteração e índice é de um para um e não existem dependências de dados entre iterações, a tradução do *for* pode-se realizar

```
     % c0, c1 e c2 sao arrays formados a partir
     % da soma de dois arrays a e b, de n elementos.

c0:= for initial i:= 1        % forma iterativa
         until   i = n
         repeat
            s:= a [old i] + b [old i];
            i:= old i + 1
         returns array of s
     end for;

c1:= for i in 1,n             % forma produto (forall)
         s:= a [i] + b [i]
         returns array of s
     end for;

c2:= for e1 in a dot e2 in b % forma produto
         s:= e1 + e2          % insensivel ao indice
         returns array of s
     end for;
```

Figura 3.5: Exemplos da construção *for* em SISAL

como uma operação de espalhamento (*scatter*) onde o *array* é transformado num *stream* e cada elemento é atribuído a um nível de iteração. A construção *for* torna-se *insensível ao índice* uma vez que o índice é estabelecido implicitamente. Isto é mostrado na Figura 3.6.

A criação dinâmica de *arrays* em SISAL é obtida através da operação de agrupamento (*gather*). A operação *gather* é recíproca à operação *scatter*: enquanto *scatter* introduz um elemento da estrutura para cada iteração, *gather* retira um elemento de cada de iteração a fim de criar uma nova estrutura (Figura 3.7). Enquanto a operação *scatter* pode ser empregada apenas na forma produto da construção *for*, *gather* pode ser empregada tanto na forma produto como na forma iterativa.

O problema que SISAL ocasiona na programação de ambientes a fluxo de dados dinâmicos está no aninhamento das construções *for*. Cada *for* comporta-se como se fosse uma função e portanto requer um nome de ativação distinto,

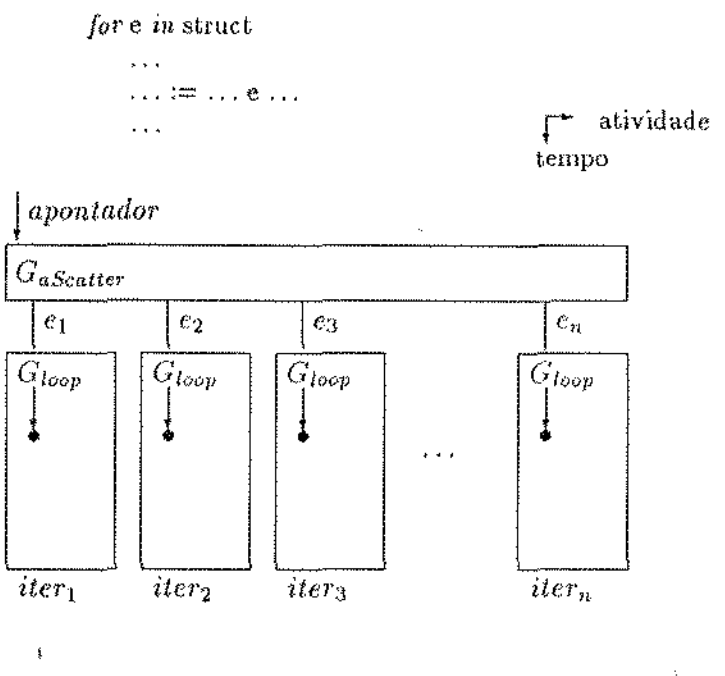


Figura 3.6: Construção *for* e a operação de *scatter*

ou seja, um novo contexto. Os argumentos de entrada e os resultados são passados implicitamente através de uma vinculação dos nomes ao contexto externo (isto é, empregando-se escopo léxico). Assim sendo, quando se passam valores ou *streams* para *for's* internos, os rótulos têm de ser alterados para o novo contexto. O rotulamento na entrada e na saída tem um custo adicional de processamento proporcional ao número de elementos comunicados entre os contextos. Este custo pode ser suprimido se for possível tratar várias dimensões no mesmo contexto.

O problema de aninhamento de *for's* geralmente pode ser atenuado quando é possível empregar um *for* insensível ao índice, pois, para este caso particular, não se faz necessário o uso de um novo nome da ativação. Neste caso, a estrutura argumento é transformada num *stream* e os novos contextos para cada uma das iterações são identificados a partir do índice. A importância deste tipo de *for* é que a passagem de valores para os contextos internos identificados pelo índice tem um custo menor, uma vez que não requer a

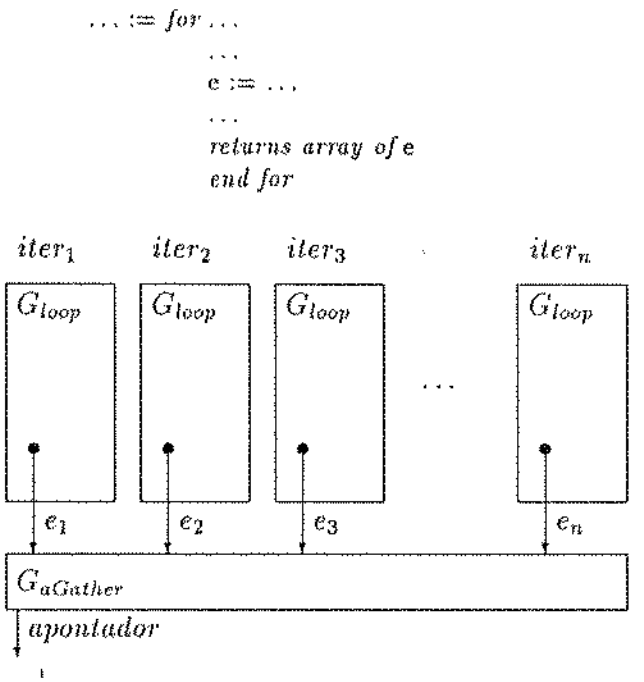


Figura 3.7: Construção *for* e a operação de *gather*

alteração do nome de ativação nos rótulos das fichas. Para a formação de estruturas multidimensionais não há como escapar do uso de vários nomes de ativação, pois é necessário distinguir cada dimensão da estrutura e isto tem de ser feito a partir deste identificador.

3.5.2 Estruturas de Dados em Id

Geradores

Id emprega *geradores* em construções de alto nível denominadas compreensões (*comprehensions*) [8].[2] Geradores são construções que produzem seqüências de valores. Apesar de serem equivalentes a construções de controle iterativas e condicionais muito seria perdido caso tivessem de ser expressos indiretamente na linguagem por meio dessas construções: assim se-

[2]SISAL também emprega geradores na forma produto da construção *for* [14].

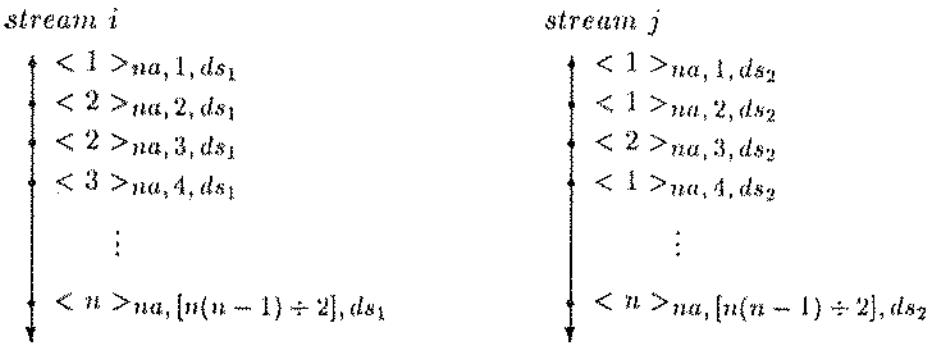


Figura 3.8: Exemplo do gerador `i <- 1 to n & j <- 1 to i`, que produz pares de índices para endereçar os elementos de uma matriz, sobre e abaixo da diagonal principal.

riam menos eficientes uma vez que não teriam uma tradução direta. Geradores podem ser empregados na indexação ao longo da formação de *arrays* e listas, e posteriormente no acesso a essas estruturas. Um exemplo do resultado de um gerador é apresentado na Figura 3.8. As Figuras 3.9 e 3.10 mostram o exemplo de compreensões em Id.

Num ambiente de processamento paralelo a presença de geradores é muito interessante porque eles se comportam como processos independentes que possuem *pipelining* interno entre o sub-processo (*loop*) que gera a seqüência de valores e o filtro que elimina os elementos indesejados desta seqüência. Uma vez inseridos num contexto, comportam-se como mais um estágio no *pipeline* deste contexto. O processamento de *arrays* torna-se mais rápido quando se emprega geradores, pois o tempo de acesso aos elementos da estrutura pode se reduzir como conseqüência do paralelismo entre o cálculo de índices, o cálculo de endereços e acesso aos elementos propriamente dito. Isto está ilustrado na Figura 3.11.

Id

Id possui como tipos básicos de estruturas *arrays* e listas. Na sua versão mais recente [34] (Id 88), todas as estruturas são armazenadas e por isto não existem *streams* na linguagem. Como já foi mencionado, sob o aspecto lógico *streams* e listas são equivalentes por suportarem o mesmo tipo de

```
fib n = {array (1,n)
         | [1] = 1
         | [2] = 1
         | [i] = fib [i-1] + fib [i-2] || i <- 3 to n}
```

Figura 3.9: Seqüência de Fibonnacci em Id

```
mat1_id n = {matrix ((1,n),(1,n))
             | [i,j] = 0 || i <- 1 to n & j <- 1 to i - 1
             | [i,i] = 1 || i <- 1 to n
             | [i,j] = 0 || i <- 1 to n & j <- i + 1 to n}

mat2_id n = {matrix ((1,n),(1,n))
             | [i,i] = 1 || i <- 1 to n
             | [i,j] = 0 || i <- 1 to n & j <- 1 to n
                                       & i <> j}
```

Figura 3.10: Matrizes Identidade em Id

operações, de modo que a falta de *streams* não prejudica o programador.

Id, como SISAL, não tem muitas operações primitivas responsáveis pela manipulação de estruturas. A maior preocupação de Id é facilitar a formação incremental de estruturas, algo que evita a formação de estruturas intermediárias tornando o programa mais claro e sucinto. Arvind [4,10] observa que a necessidade de formação incremental de estruturas é mais visível em problemas físicos que possuem condições de contorno.

Inicialmente o conceito de estruturas-I associava uma construção incremental de alto nível ao mecanismo de sincronização de baixo nível entre produtor e consumidor [11]. No entanto os conceitos de estruturas-I e de estruturas não-estritas são ortogonais. É possível ter uma linguagem funcional com formação de estruturas de modo não-incremental com uma sincronização elemento-a-elemento entre produtor e consumidor.

A primeira versão de Id possuía estruturas-I criadas explicitamente através de uma operação primitiva (`I_array`) que alocava um conjunto de posições

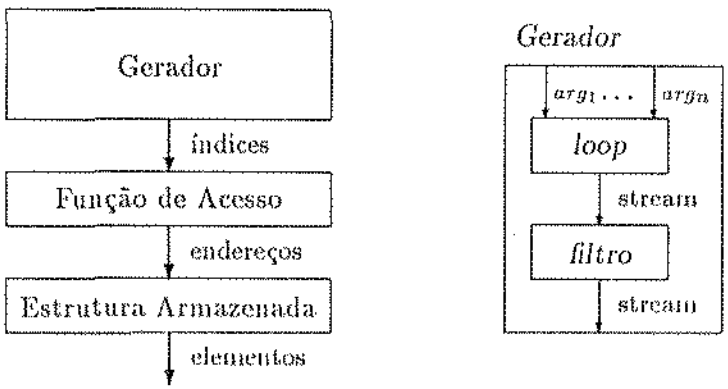


Figura 3.11: Acesso a uma estrutura empregando um gerador

livres (*slots*) às estruturas. Num *slot* podia ser colocado qualquer objeto. Entretanto, como na alocação deste conjunto não era necessária a vinculação de um nome, a transparência referencial podia ser violada nos trechos responsáveis pelo seu preenchimento. No exemplo seguinte, apresentado por Arvind [10], o primeiro trecho de programa aloca n posições, enquanto que o segundo $2n$:

```
{ a = I_array (1,n)
  in
    a,a }

I_array (1,n), I_array (1,n)
```

A perda da referência transparencial provocada pelas estruturas-I torna não-funcionais alguns trechos de um programa em Id. No entanto, na época em que foram concebidas as estruturas–I, desconhecia-se outro modo para realizar com eficiência a formação incremental e, apesar da perda da referência transparencial, o determinismo era preservado pela condição de atribuição única aos elementos de uma estrutura–I.

Em Id 88 foram introduzidas compreensões como mecanismo básico de formação de estruturas. As compreensões são suficientemente gerais para

permitirem a formação incremental de estruturas de modo que a presença explícita de estruturas–I, como descrito anteriormente torna-se desnecessária, permitindo que a linguagem possa voltar a ser funcional. As estruturas–I não foram eliminadas permanecendo como opção para algumas aplicações. As compreensões poderiam ser consideradas apenas como uma alternativa sintática já que, de fato, são equivalentes a estruturas–I. Não obstante, elas mantêm a transparência referencial e introduzem tamanha clareza na descrição de estruturas, que a elegância dos programas que as empregam é indiscutível.

Capítulo 4

Unidades de Armazenamento

O capítulo anterior foi dedicado à exposição dos principais conceitos associados ao tema do processamento de estruturas de dados em fluxo de dados. Neste capítulo voltamo-nos para a discussão das unidades de armazenamento de estruturas – Memórias de Estruturas – que são peças fundamentais de um computador a fluxo de dados. Dois exemplos de como essas unidades funcionam são tirados dos computadores de Manchester e do MIT. As principais implementações de computadores a fluxo de dados como SIGMA-1 [42] e o próprio computador de Manchester [29] baseiam-se no projeto da Memória de Estruturas do MIT, de cuja concepção surgiram as principais idéias implantadas no armazenamento de estruturas.

4.1 Memória de Estruturas de Manchester

A Memória de Estruturas foi desenvolvida para melhorar o desempenho de programas envolvendo estruturas de dados. No protótipo inicial do computador de Manchester, todas as estruturas eram armazenadas na Unidade de Agrupamento usando *sticky-tokens* (fichas "grudantes", ou seja, fichas que quando agrupadas, são copiadas, mas não são extraídas) . Entretanto, esta abordagem revelou-se ineficiente por três razões principais [41]:

- O número elevado de instruções intermediárias necessárias para o acesso a elementos de uma estrutura reduz o desempenho e aumenta a latência de leitura.

- O espaço disponível para armazenamento fica limitado ao tamanho da Unidade de Agrupamento que também é usada por escalares.
- O custo de armazenamento é alto.

As metas básicas exploradas no projeto da Memória de Estruturas de Manchester foram modularidade, extensibilidade e paralelismo interno à unidade. A Memória de Estruturas de Manchester foi dirigida para o armazenamento de estruturas-I e isto repercutiu consideravelmente na sua organização e complexidade [29,41].

Uma Memória de Estruturas é formada por quatro partes:

- a Sub-unidade de Alocação;
- a Sub-unidade de Armazenamento;
- a Sub-unidade de Limpeza; e
- a Sub-unidade (de armazenamento) de Leituras Antecipadas.

Estas sub-unidades colaboram entre si para a execução das tarefas da Memória de Estruturas como um todo. As tarefas consistem basicamente na gerência de memória e no controle das consultas, através de uma troca assíncrona de mensagens. As sub-unidades realizam tarefas independentes e podem operar concorrentemente. Para melhorar a capacidade de armazenamento ou o desempenho de uma Memória de Estruturas, o número de módulos de cada sub-unidade pode ser aumentado até a configuração desejada.

4.1.1 Sub-unidade de Alocação

A Sub-unidade de Alocação efetua a reserva de espaço para o armazenamento de estruturas na Sub-unidade de Armazenamento. Atualmente o sistema *buddy* [30] de gerência de memória é empregado onde o número de posições alocadas é par. A estratégia de alocação pode variar. Quando a Sub-unidade de Alocação emprega listas de espaço livre por tamanho, a melhor estratégia é *first-fit*. Existe em geral um compromisso entre *first-fit* e *best-fit* envolvendo complexidade e desempenho. O que deve ser destacado é que a gerência de memória é um processo realizado totalmente em *hardware* (*firmware*).

4.1.2 Sub-unidade de Armazenamento

Na Sub-unidade de Armazenamento são mantidos os elementos de uma estrutura. O acesso a esta memória é realizado diretamente, sem qualquer tradução de endereço e sem a participação intermediária de qualquer outra sub-unidade. Por esta razão, o tempo de acesso aos elementos já armazenados de uma estrutura, é baixo. Para suportar estruturas-I por meio de *queueing*, existem em cada endereço dois bits que determinam se o elemento está presente e se ocorreram leituras antecipadas naquele endereço, como discutimos na seção 3.4.1. As leituras antecipadas são dirigidas à Sub-unidade de Leituras Antecipadas.

4.1.3 Sub-unidade de Limpeza

O papel da Sub-unidade de Limpeza é remover os elementos de uma estrutura fora de uso. Os bits de status de cada endereço são marcados com os estados *ausente* e *nenhuma leitura pendente*, depois de cada remoção. Na realidade, por intermédio da Sub-unidade de Limpeza, realiza-se uma sincronização semelhante ao *collect*, neste caso dirigida à recuperação de memória. A partir do momento em que uma estrutura foi removida, o seu espaço é devolvido à lista de espaço livre na Sub-unidade de Alocação.

4.1.4 Sub-unidade de Leituras Antecipadas

Na Sub-unidade de Leituras Antecipadas são armazenadas as requisições de leitura de elementos que ainda não se encontram presentes na Sub-unidade de Armazenamento. Estas requisições são armazenadas em listas sendo que cada lista está associada a um único endereço na Sub-unidade de Armazenamento.

4.1.5 Protocolo entre as Sub-unidades

A tabela seguinte apresenta, em resumo, o protocolo entre as sub-unidades anteriores:

requisição	resposta
Anel → Sub-unidade de Alocação	
solicitação de espaço	endereço-base da área livre
Anel → Sub-unidade de Armazenamento	
pedido de leitura	valor do elemento endereçado
pedido de escrita	(sem confirmação)
Sub-unidade de Armazenamento → Sub-unidade de Leituras Antecipadas	
leitura antecipada	(sem confirmação)
liberar leituras antecipadas	pedido de leitura
Sub-unidade de Armazenamento → Sub-unidade de Limpeza	
início de limpeza	pedido de limpeza (primeiro elemento)
limpeza confirmada	pedido de limpeza (elemento seguinte)
Sub-unidade de Alocação → Sub-unidade de Armazenamento	
reserva de espaço	liberação de espaço
unir áreas livres	liberação de espaço
Sub-unidade de Alocação → Sub-unidade de Armazenamento	
apontador de descritor	(sem confirmação)

A Sub-unidade de Alocação mantém três estruturas de dados: uma lista de *descritores* para áreas livres na Sub-unidade de Armazenamento, uma lista de *descritores livres* e uma pilha com *apontadores* para descritores livres. Na iniciação do sistema, uma parte dos apontadores para descritores livres é enviada para a Sub-unidade de Armazenamento, onde também são armazenados numa pilha.

Os espaços (áreas) na Sub-unidade de Armazenamento são divididos em blocos e caracterizados pelo seu endereço inicial e tamanho. Se um pedido de *reserva de espaço* para uma estrutura for maior do que o necessário e o bloco puder ser dividido, a Sub-unidade de Armazenamento pode liberar o espaço excedente através de uma mensagem. A mensagem de *liberação de espaço* contém um apontador para um descritor livre e informações sobre a área propriamente dita. Ao receber esta mensagem, a Sub-unidade de Alocação insere o descritor na lista de descritores para áreas livres e envia um novo apontador para um descritor livre, a fim de que a altura média da pilha de descritores livres na Sub-unidade de Armazenamento seja mantida.

requisição	resposta
Sub-unidade de Leituras Antecipadas → Sub-unidade de Armazenamento	
apontador para lista de leitura antecipada	(sem confirmação)

Existe um protocolo semelhante ao anterior entre Sub-unidade de Leituras

Antecipadas e a Sub-unidade de Armazenamento para realizar a gerência de leituras antecipadas. Uma pilha na Sub-unidade de Leituras Antecipadas contém apontadores para listas de leituras antecipadas que estão vazias.

A recuperação de memória fora de uso é um processo envolvendo as Sub-unidades de Armazenamento, Limpeza e Alocação. No momento em que a contagem de referências de uma estrutura atinge o valor zero, a Sub-unidade de Armazenamento envia para a Sub-unidade de Limpeza uma mensagem de *início de limpeza*, que contém o endereço do início da área e o número de elementos da estrutura. Assim que a Sub-unidade de Limpeza recebe esta mensagem, ela inicia a remoção da estrutura. Cada elemento da estrutura é removido pela Sub-unidade de Limpeza por um *pedido de limpeza* para a Sub-unidade de Armazenamento. Se o elemento a ser removido é um apontador para uma outra estrutura, antes de removê-lo, a Sub-unidade de Armazenamento envia um *pedido de escrita* para a Sub-unidade de Armazenamento onde a estrutura está armazenada (que conforme o caso pode ser ela mesma), para decrementar aquele contador de referências. Depois que o elemento foi removido, a Sub-unidade de Armazenamento envia para a Sub-unidade de Limpeza uma mensagem de *limpeza confirmada*. Quando o último elemento da estrutura foi removido, a Sub-unidade de Armazenamento envia para a Sub-unidade de Alocação uma mensagem de *liberação de espaço*.

Se uma área liberada pela da Sub-unidade de Armazenamento tem uma área adjacente (*buddy*) livre, a Sub-unidade de Alocação envia de volta uma mensagem para *unir áreas livres*, que será respondida com uma nova mensagem de *liberar área livre*.

4.1.6 Interface de Software

A interface de *software* para a Memória de Estruturas de Manchester é bastante simples. As poucas funções necessárias são as seguintes:

- solicitação de espaço;
- escrita de um elemento;
- leitura de um elemento;
- valor inicial da contagem de referências de uma estrutura;
- incremento ou decremento de uma contagem de referências.

No nível da linguagem de máquina existe uma instrução associada a cada uma destas operações. Essas instruções criam uma mensagem com as informações necessárias e a dirigem para a Memória de Estruturas. Como já foi descrito na seção anterior, o protocolo é basicamente o seguinte:

operação	requisição	resposta
alocação	<tamanho, contextoDestino>	<endereçoBase>$_{contextoDestino}$
escrita	<valor, endereço>	sem confirmação
leitura	<endereço, contextoDestino>	<valor>$_{contextoDestino}$
inic. CR	<endereçoBase, valor, contextoDestino>	sem confirmação
inc/dec CR	<inc/dec, endereçoBase>	<ack>$_{contextoDestino}$

A sincronização dos elementos entre produtor e consumidor pode ser global (*collect*), realizada para toda a estrutura, ou individual, sobre cada elemento. A sincronização global requer uma contagem dos elementos da estrutura ou uma árvore de sincronização. A opção adotada em Manchester para a sincronização global foi pela contagem de elementos.

Para a sincronização individual dos elementos, todos os requisitos estão presentes no *hardware* da Sub-unidade de Armazenamento. Para a sincronização global existem dois modos de operação:

- Na Unidade de Agrupamento: cria-se um contador numa posição desta unidade usando-se uma *sticky-token*. O valor inicial da ficha corresponde ao tamanho da estrutura. Para esta posição são enviadas mensagens em quantidade equivalente ao tamanho da estrutura, que decrementam o valor da *sticky-token*.
- Na Memória de Estruturas: cria-se uma estrutura "fantasma". Esta estrutura não tem qualquer elemento, mas a sua contagem de referências tem o valor do número de elementos da estrutura a ser sincronizada. No momento que um elemento fica pronto, é enviada uma mensagem para decrementar a contagem de referências da estrutura fantasma. Quando a contagem de referências chega a zero, a Sub-unidade de Armazenamento envia um sinal para o anel e inicia, junto com a Sub-unidade de Limpeza, um processo para a remoção desta estrutura.

A implementação de SISAL para a máquna Manchester emprega apenas estruturas estritas, pois assim pode-se fazer uso de uma contagem de referências otimizada, elaborada por Sargeant [40]. O processo de recuperação

de memória baseia-se na contagem de Sargeant e no mecanismo automático de recuperação de memória presente na Memória de Estruturas.

Comentários

À primeira vista parece um contra-senso o emprego de estruturas estritas num computador onde estão presentes todos os mecanismos necessários para suportar estruturas não-estritas. A menos que o sistema de contagem de referências para estruturas não-estritas leve a uma perda expressiva de desempenho com relação a um mesmo programa que use estruturas estritas com o sistema otimizado de contagem de referências, não parece haver sentido em se limitar o computador ao uso de estruturas estritas. Insistir nestas estruturas provoca um duplo desperdício: o *hardware* voltado para estruturas não-estritas não é usado e é preciso recorrer a mecanismos artificiais para sincronizar as estruturas estritas.

Em qualquer dos casos, a sincronização global realizada como descrito anteriormente provoca um tráfego elevado de fichas. Admita-se que seja n o tamanho da estrutura sendo sincronizada. Para a sincronização na Unidade de Agrupamento são enviadas n fichas para aquela unidade. Para a sincronização na Memória de Estruturas a situação é ainda pior, pois são necessárias n sincronizações na Unidade de Agrupamento para formar as n mensagens para a Memória de Estruturas. Neste último caso, se n for grande e todos os elementos ficarem prontos ao mesmo tempo, o endereço do contador de referências da estrutura fantasma torna-se um *hot spot*.

4.2 Memória de Estruturas do MIT

Apesar da Memória de Estruturas de Manchester ter sido a primeira a implementar os mecanismos de sincronização para estruturas-I, esta idéia é devida a Arvind, do MIT [11]. A Memória de Estruturas do MIT, ao contrário da de Manchester, tem apenas os recursos indispensáveis à implementação de estruturas-I [9]. A alocação e recuperação de memória são tarefas realizadas por *software*, através de processos chamados *Managers* (gerentes). Um *Manager* é responsável pela iniciação dos *bits* de status de cada célula de memória através de um conjunto de mensagens para a Memória de Estruturas.

O espaço de endereçamento é global. Numa máquina que tem diversas unidades de Memória de Estruturas cada uma tem uma parte do espaço de endereçamento disponível. Ao receber uma requisição, o gerente reserva uma área e responde com um apontador para o seu início. Na literatura não parece existir menção à técnica de gerência normalmente empregada. Como a alocação se faz por *software*, existe muita flexibilidade na escolha da técnica mais conveniente e das alterações nela necessárias.

Com relação à recuperação das áreas de memória, também não se tem ao certo na leitura da pesquisa contemporânea como esta tem sido feita. Tudo indica que a contagem de referências é desnecessária. A recuperação de memória pode ser realizada por região. Cada anel é destinado a executar um bloco de código (*code block*) ao final de cuja execução todos os recursos usados são descartados. As estruturas são passadas explicitamente entre regiões que, possivelmente, estão organizadas de forma herárquica.

Cada Memória de Estruturas é composta de um controlador e da memória propriamente dita, que está dividida em duas áreas:

- a área de dados onde são armazenados os elementos de uma estrutura. Dois *bits* de status associados a cada célula de memória dão o suporte às estruturas–I;
- a área de leituras antecipadas onde são armazenadas essas leituras. Os pedidos de leituras antecipadas estão organizados como uma lista encadeada que tem como origem o local do dado ausente.

4.2.1 Protocolo com a Memória de Estruturas

O seguinte esquema ilustra o protocolo de interação entre o anel e a Memória de Estruturas do MIT:

atividade	requisição	reposta
leitura	<leitura, endereço, contexoDestino >	<valor>$_{contextoDestino}$
escrita	<escrita, valor, endereço>	<sinal>$_{contextoDestino}$

Ao contrário do que ocorre no computador de Manchester, as escritas de elementos para a Memória de Estruturas do MIT são confirmadas. A confirmação de uma escrita não é feita pela Memória de Estruturas, mas pelo anel, indicando que a instrução de escrita foi executada, isto é, que a mensagem de escrita para a Memória de Estruturas está a caminho. Isto é

necessário para que o processo de sincronização possa reconhecer quando toda a atividade de um bloco de código terminou.

4.2.2 Comentários

Alguns artigos na literatura mencionam a presença da Memória de Estruturas junto a cada anel. Outros as colocam em separado, sendo o acesso realizado pela rede de interconexão.

O *hardware* da Memória de Estruturas do MIT é aparentemente mais simples de ser implementado do que o da memória correspondente de Manchester. Contudo, até o momento, não existem resultados concretos do desempenho da Memória de Estruturas do MIT. Um protótipo deste computador começou a ser implementado em meados de 1988 e os resultados preliminares ainda estão para ser divulgados [17].

Observa-se que a simplicidade prometida por esse *hardware* pode dar origem a uma Memória de Estruturas menor e modular. Haverá um destaque do *software*, que será responsável por todo o processo de gerência de memória. Isto está de acordo com a abordagem do grupo do MIT, que dimensionou cada anel de uma forma apenas modesta.

Capítulo 5

Localidade na Memória de Estruturas

Este capítulo apresenta uma abordagem para o tratamento de estruturas em fluxo de dados, o que se constitui a principal contribuição deste trabalho para o desenvolvimento do tema. A abordagem consiste basicamente da incorporação de localidade em operações sobre estruturas de dados armazenadas. Os principais requisitos do *hardware* da Memória de Estruturas são o suporte para a sincronização de estruturas estritas e o suporte para operações vetoriais. Operações locais na Memória de Estruturas não modificam a granularidade de operações executadas no anel, mas apenas transferem para aquela unidade as operações que atuam sobre um *conjunto* de dados. Como conseqüencia da localidade das operações, obtém-se uma eficiência maior no uso das vias de comunicação e nas operações sobre vetores.

5.1 Processos Relacionados a Estruturas

É possível dividir a execução de um programa em *processos* distintos. Os processos relativos a operações sobre estruturas são importantes devido a sua generalidade e freqüência. Destacam-se em operações sobre estruturas os seguintes processos:

1. Geradores, que como vimos na seção 4.2, são construções que produzem seqüências de valores. O resultado de um gerador é um *stream*.
2. Construções *forall* (*for* na forma produto), que aplicam uma função repetidamente sobre elementos de um *stream*. Na forma mais genérica

os valores podem ser tuplas (isto é, existe mais de um stream argumento). Se *forall* for aplicado a uma estrutura armazenada, ela será inicialmente convertida para um *stream*. O resultado de um *forall* também é um *stream*.

3. Acumuladores [36], que criam uma estrutura de dados armazenada. Seja E a estrutura de dados, $E[i]$ a sua i-ésima posição e E_k a sua k-ésima versão. Seja f_a uma função (associativa) de acumulação. l representa leituras e w escritas, sendo que $w\ [\ E_k[i]\ ]$ representa o valor escrito na i-ésima posição da estrutura E na sua k-ésima versão. Seja n_i o número total escritas necessárias para que $E[i]$ seja considerado pronto. $\perp$ representa um valor indefinido e e e c representam valores. As propriedades dos acumuladores são:

   (a) $E_0[i] = e_i$ (valor inicial)

   (b) $E_k[i] = f_a(E_{k-1}[i],\ w\ [\ E_{k-1}[i]])$, $0 < k \leq n_i$

   (c) $l\ [\ E_k[i]\ ] = \perp$, $0 < k < n_i$

   (d) $l\ [\ E_{n_i}[i]\ ] = e_i$ (valor final)

   (e) $w\ [\ E_{n_i}[i]\ ] = \perp$

   Acumuladores são interessantes para implementar contadores e histogramas.

Em geral as primeiras estruturas encontradas num programa estão prontas na forma de argumentos, ou seja, são estruturas estáticas, já armazenadas. A característica principal dessas estruturas é que seu tamanho e o valor de seus elementos foram definidos antes do início da execução do programa. Durante a execução, estabelece-se uma dinâmica na transformação destas estruturas. Em fluxo de dados, processa-se uma transformação armazenando a estrutura e realizando acessos conforme o necessário: os elementos são trazidos ao anel, processados e novamente armazenados, como mostra a Figura 5.1.

Algumas transformações podem ser realizadas de maneira mais eficiente do que outras, porque apresentam um padrão de acesso mais simples e uniforme sobre os elementos da estrutura, uma observação que pode ser feita da análise do *software* e do comportamento do *hardware* nestas transformações. Para operações de acesso simples, como por exemplo, a soma de dois vetores, a busca (*fetch*) dos elementos pode ser otimizada enviando-se uma única mensagem à Memória de Estruturas. Embora estas operações apresentem

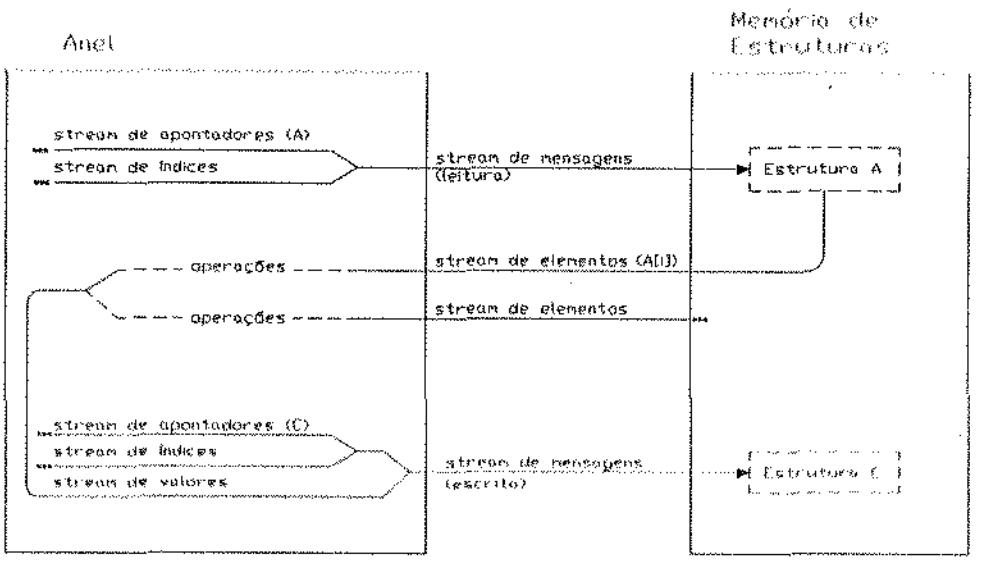


Figura 5.1: Processamento de estruturas do modo usual

dinâmica melhor, isto ainda não é o ideal, porque ao trazer os elementos para o anel não se faz uso da sua proximidade e do padrão de acesso conhecido de antemão. Em algumas situações, tudo indica ser desnecessário fixar o processamento no anel diante dos benefícios que podem ser obtidos se a Memória de Estruturas possuir *hardware* apropriado para dar prosseguimento ao processamento lógico/aritmético nos vetores.

Aparentemente as transformações que melhor se enquadram entre as que podem ser realizadas de modo local são:

- *append*.
  append pode ser realizado como uma cópia integral da estrutura seguido de uma atribuição. Cabe ao compilador identificar um conjunto de *appends* de modo a evitar versões intermediárias desnecessárias. Elementos iguais devem ser copiados de modo local na Memória de Estruturas.

- Criação de estruturas.
  Algumas estruturas são formadas por um único valor ou por uma seqüência de valores com uma regra de formação muito simples.

- Operações sobre vetores.
  Certas operações vetoriais equivalem a um *forall*; outras a um acumulador. De modo geral, operações sobre vetores envolvem processamento. Para suportar localmente operações lógico/aritméticas vetoriais, a organização da Memória de Estruturas tem de ser adaptada. Para aproveitar o paralelismo temporal em *hardware*, os vetores envolvidos nessas operações têm de ser do tipo *estrito*. Estruturas não-estritas poderiam violar o paralelismo temporal no caso em que a operação vetorial precisasse ser suspensa devido à ausência de elementos nas estruturas-argumento.

Na seqüência desta exposição damos enfoque a operações vetoriais localizadas na Memória de Estruturas, pelo ganho potencial de desempenho que podem trazer a um computador a fluxo de dados.

5.2 Operações Vetoriais

Discutimos inicialmente conceitos gerais de operações vetoriais, em seguida as operações vetoriais em fluxo de dados e, finalmente, apresentamos uma organização para uma Memória de Estruturas que pode servir o propósito de processar vetores localmente.

5.2.1 Conceitos de Operações Vetoriais

Ao contrário de um processador escalar (que realiza operações apenas sobre escalares) um processador vetorial tem como principais características:

- operações sobre vetores;
- organização em estágios[1] (3 a 5 estágios tipicamente);
- ciclo do processador geralmente inferior ao ciclo de memória;
- alta taxa de comunicação de dados com a memória.

O bom desempenho de um processador vetorial decorre do *hardware* dirigido para explorar a natureza repetitiva que existe nas operações sobre vetores. *Pipelining*, como vimos na seção 2.3.1, é a principal técnica empregada. Outra diferença importante no processador vetorial é que ele decodifica a instrução associada a uma operação vetorial uma única vez ao invés de um número de vezes proporcional ao número de elementos do vetor, como faria

[1] *pipeline*

um processador escalar.

Se um processador vetorial estiver equipado com mais de uma unidade funcional, ele pode explorar ainda mais o paralelismo, atribuindo operações independentes a unidades distintas (*overlapping*) ou, se as operações forem encadeadas, associando-as em série (*chaining*).

O tempo de execução numa unidade funcional é modelado em $n\tau + \sigma$ unidades de tempo onde n é o tamanho do vetor, τ é o tempo de ciclo do *pipeline* e σ o tempo total para configurar os estágios e para que entrem em regime. O tempo *médio* de uma operação vetorial aproxima-se do tempo de ciclo τ para vetores longos [3].

Como geralmente o ciclo de uma unidade funcional costuma ser inferior a um ciclo de memória – algo que pode provocar estados-de-espera (*wait-states*) no processador para sincronizar as transferências de dados – usam-se duas técnicas em processadores vetoriais na tentativa de compensar este efeito:

- prover um ou mais conjuntos de registradores vetoriais de modo a reduzir a comunicação com a memória;
- dividir a memória em bancos distintos entrelaçados (*interleaved*). Neste caso usa-se normalmente o entrelaçamento de baixa-ordem (*low-order interleaving*), onde elementos consecutivos de um vetor são atribuídos a bancos de memória adjacentes. Desta forma o acesso aos bancos, um por ciclo do processador, pode ocorrer em paralelo, obtendo-se um aumento na taxa de comunicação entre o processador vetorial e a memória como um todo.

5.2.2 Operações Vetoriais em Fluxo de Dados

Com o objetivo de melhorar o desempenho de operações sobre vetores em computadores a fluxo de dados, pode-se aproveitar o fato de as estruturas estarem armazenadas na Memória de Estruturas. Algumas vantagens desta abordagem são:

- Redução da comunicação de dados entre anel e Memória de Estruturas em operações vetoriais, evitando que elementos sejam levados para processamento externo à Memória de Estruturas e depois trazidos de volta para serem armazenados. Assim:

  - elimina-se o cálculo de endereços para operações onde se conhece a posição dos argumentos, tornando-o implícito;
  - eliminam-se as sincronizações no acesso a elementos e seu armazenamento;
  - a sincronização de *collect* é desnecessária porque operações sobre estruturas estritas produzem estruturas estritas.

- Diminuição do tamanho do código, eliminando-se instruções redundantes.
- Redução do tempo de operações sobre vetores por meio de *pipelining*.

De modo geral, a idéia consiste em aliar as vantagens do processamento MIMD e SIMD.[2] No caso de fluxo de dados, associar os processamentos SIMD e MIMD é fácil porque se estabelece um protocolo que provê uma comunicação entre ambos. A parte MIMD fica no anel e dirige o processamento; a parte SIMD fica na Memória de Estruturas e atende às requisições do anel.

Algumas operações primitivas sobre vetores, interessantes de serem observadas nesta linha de concepção, estão na Tabela 5.1. As três primeiras operações envolvem processamento, as duas seguintes movimentação e as duas últimas transformação de dados. A última operação da Tabela 5.1 está disponível somente ao compilador e procura reduzir a movimentação desnecessária de dados. Note-se que *append* preserva a estrutura-argumento, caso isto seja necessário, mas *update* constrói o resultado sobre o argumento.

Naturalmente uma implementação direta, em vão da complexidade de algumas das operações da Tabela 5.1, pode ter um custo muito elevado para as aplicações em que se deseja empregar a arquitetura. Não se pode pretender implementar em *hardware* operações de custo elevado utilizadas apenas esporadicamente. O propósito deve ser o de empregar uma tecnologia amadurecida e eficiente para melhorar uma parte do processamento envolvendo estruturas.

[2] *Multiple Instruction Stream Multiple Data Stream* e *Single Intruction Stream Multiple Data Stream* respectivamente, na classificação de Flynn [20].

operação	operadores (op)	sintaxe	funcionalidade
pto. a pto.	$+, -, *, /, <, >, =, \vee, \wedge$	$a\ op\ b$	*array, array* $\rightarrow$ *array*
unária	$-, \neg$	$op\ a$	*array* $\rightarrow$ *array*
redução	Σ, Π, max, min	$op\ a$	*array* $\rightarrow$ *escalar*
concat	$\|\|$	a op b	*array, array* $\rightarrow$ *array*
extract	$\downarrow$	$op\ a_{i..j}$	*array, int, int* $\rightarrow$ *array*
append	$\leftarrow$	$a_i\ op\ v$	*array, int, escalar* $\rightarrow$ *array*
update	$:=$	$a_i\ op\ v$	*array, int, escalar* $\rightarrow$ *array*

Tabela 5.1: Algumas *primitivas* para operações vetoriais

Soma de dois vetores	$a + b$	1 operação
Soma dos elementos de um vetor	Σa	1 operação
Comparação de dois vetores	$\Sigma(a = b)$	2 operações
Produto interno de dois vetores	$\Sigma(a * b)$	2 operações

Tabela 5.2: Exemplos de operações vetoriais empregando as primitivas da Tabela 5.1

5.3 Requisitos de uma Memória de Estruturas Vetorial

Fundamentalmente os requisitos que devem ser atendidos para que uma Memória de Estruturas possua operações vetoriais são os seguintes:

1. Operações realizadas de modo local devem levar menos tempo do que se forem executadas fora da Memória de Estruturas, uma imposição do próprio objetivo desta arquitetura.
2. Consultas externas não devem ser oneradas relativamente à arquitetura convencional, ou seja, ler/escrever um escalar na Memória de Estruturas deve independer de outras operações locais em curso.
3. A sincronização de estruturas (*collect*) deve ser realizada localmente, de modo transparente para o anel.

O objetivo é acrescentar eficiência ao computador a fluxo de dados sem a perda da vantagem do armazenamento de estruturas. Uma característica importante desses computadores, que se deve procurar manter, é a expansibilidade dos seus módulos básicos de *hardware*. Isto implica em dizer que o dimensionamento da Memória de Estruturas deve ser realizado com base no anel se ela estiver ligada diretamente a ele. Na concepção em questão, isto é uma necessidade, pois, para que o desempenho possa manter-se elevado, a

conversão de *streams* em vetores armazenados e vice-versa deve ser rápida. Desta forma, o agregado anel-Memória de Estruturas constitui um módulo para a expansão do sistema. Para que a Memória de Estruturas e o anel possam operar concorrente e assincronamente, a Memória de Estruturas deve possuir filas de entrada e saída.

5.4 Granularidade das Operações Vetoriais

A granularidade das operações está associada ao tamanho da tarefa atribuída a cada processador. Se uma estrutura grande, contendo, digamos 100 mil escalares, for alocada e operada por apenas uma Memória de Estruturas, o tempo de formação do resultado pode ser elevado apesar do paralelismo temporal existente. A conveniência de partir estruturas grandes em diversos grãos na forma de blocos ou páginas, alocando-os a Memórias de Estruturas distintas, está no aumento do paralelismo e no tempo possivelmente menor para se obter o resultado.

Se o computador apresenta apenas uma Memória de Estruturas, a granularidade será variável em todas as operações vetoriais atribuídas a esta unidade. Num computador com diversas Memórias de Estruturas a granularidade deve ser determinada a partir do desempenho do processador vetorial situado em cada Memória de Estruturas e da capacidade de armazenamento destas unidades. Trata-se essencialmente de uma questão de engenharia que pode ser realizada simuladamente, mas que foge ao objetivo deste trabalho. Limitamo-nos aos aspectos qualitativos do assunto.

Algumas das questões relacionadas à granularidade dos blocos são as seguintes:

- Que tamanho deve ter um bloco?
- Como formar os blocos de uma estrutura a partir dos seus elementos?
- Como alocar estes blocos na presença de mais de uma Memória de Estruturas?

A primeira questão deve ser resolvida levando em conta que a formação dos elementos de um bloco tem de ser sincronizada porque o bloco é parte de uma estrutura estrita. Existem três aspectos relacionados ao tamanho do bloco:

- O tempo de latência na formação do bloco é proporcional ao seu tamanho.
- Quanto maior o bloco, mais difícil a sua sincronização num único ponto devido à contenção naquela posição de memória (como vimos no final da seção 3.4.1).
- Cada escrita deve ser sincronizada. O tempo envolvido na sincronização pode variar segundo a implementação. Se a sincronização é realizada por *firmware* (micro-código), usando um contador ela toma mais tempo do que a escrita propriamente dita, pois requer um ciclo tipo *read-modify-write*. Nesse caso a taxa de formação dos elementos de uma estrutura pode ser maior que a sua taxa de sincronização, sendo necessário que a fila de mensagens na entrada da Memória de Estruturas possa absorver essa diferença. Naturalmente deseja-se a fila menor possível por razões de custo e espaço.

Ainda no que diz respeito ao tamanho do bloco, há um compromisso entre o acréscimo de desempenho obtido por blocos pequenos e o custo de replicação de Memórias de Estruturas para suportar o processamento simultâneo de muitos blocos.

A segunda e terceira questões que formulamos acima não podem ser resolvidas facilmente. São questões intimamente associadas às operações que, num determinado momento, estão em curso entre estruturas, dependentes por isso do programa e da linguagem de programação. Num ambiente funcional, como o considerado neste trabalho, as estruturas são criadas dinamicamente. Operações vetoriais localizadas na Memória de Estruturas vão requerer modificações importantes no ambiente de programação. Será preciso que o programador, por exemplo, possa oferecer elementos ao compilador sobre o tamanho e a dimensão das estruturas, de forma a permitir a análise do contexto, a formação dos blocos e a alocação dos mesmos nas diversas Memórias de Estruturas.

5.5 Interface de Software

Vamos admitir a divisão do espaço na Memória de Estruturas em blocos ou páginas de tamanho fixo, onde o tamanho da página possa ser passado ao compilador. Como uma estrutura pode estar distribuída ao longo de diversas páginas, é necessário introduzir uma função de acesso a seus elementos.

Esta função é introduzida pelo compilador na tradução do programa-fonte. Esquematicamente, a Figura 5.2 mostra uma estrutura 'x' distribuída em páginas $P_1 \ldots P_k$. A função de acesso é representada pelo retângulo que envolve as páginas. O seletor é um conjunto de índices que determina o elemento a que se deseja ter acesso. A função determina a página e a posição onde o elemento está armazenado.

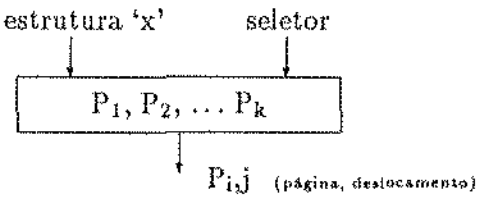


Figura 5.2: Função de acesso para uma estrutura distribuída em páginas

Quando o tamanho da estrutura é menor do que a página a função de acesso pode ser dispensável dependendo do mapeamento que houver entre a disposição dos elementos na estrutura e a sua disposição na página. Por exemplo, se a estrutura for um vetor com tamanho menor do que uma página, a função de acesso é trivial e portanto desnecessária. Porém, se for uma matriz, mesmo que todos os elementos caibam numa página a função de acesso é uma necessidade.

Para estruturas maiores do que uma página, a função de acesso é um dos pontos importantes a se examinar na abordagem do processamento localizado. A função de acesso introduz uma latência adicional, originada no seu processamento, para cada acesso à estrutura feito a partir do anel. Durante a formação e o consumo da estrutura a função de acesso é necessária. Entretanto, nas operações realizadas localmente na Memória de Estruturas ela é suprimida. As simulações que realizamos e que são apresentadas no Capítulo 6 não contemplaram especificamente as compensações deste ônus com os ganhos de eficiência devido ao processamento vetorial. Dependendo da implementação na arquitetura pode haver um limite de granularidade a partir do qual é mais conveniente realizar as operações vetoriais no anel, como nos computadores de Manchester e do MIT. Para vetores menores do que uma página, entretanto, não existe o ônus da função de acesso e, sendo assim, operações nesses vetores poderão ser realizadas com vantagens numa

Memória de Estruturas vetorial.

Para que as operações vetoriais possam ser exploradas pelo *hardware* é preciso inseri-las no programa durante o processo de geração de código. Algumas alternativas para fazer isto são:

- através de uma biblioteca de funções disponíveis ao programador;
- dotando o compilador com capacidade para identificar os trechos de programa vetorizáveis;
- introduzindo na linguagem, como em APL, operações sobre vetores e matrizes.

A primeira alternativa é a mais simples, mas limita o transporte dos programas a sistemas com as mesmas bibliotecas e pode tornar os programas dependentes das funções presentes nessas bibliotecas. A segunda alternativa vem sendo estudada há tempo no contexto dos processadores vetoriais programados em FORTRAN e apresenta resultados positivos [37]. Esta alternativa pode ser estendida sem dificuldade para as linguagens funcionais, uma vez que a sua análise é mais simples do que a das linguagens imperativas pela ausência de *aliasing* e efeitos colaterais. A terceira alternativa é mais simples do que a segunda, porém menos flexível. A disseminação de uma nova linguagem pode levar muito tempo.

A recuperação do espaço de estruturas em desuso pode empregar o mesmo mecanismo disponível para estruturas estritas no computador de Manchester [40,41]. Um contador é associado a cada estrutura estabelecendo a todo momento o número de referências por parte de outras estruturas ou operações. Durante a compilação são introduzidas instruções para incrementar e decrementar este contador. Isto é feito com base nas operações sobre a estrutura, se esta é argumento ou resultado de uma função. Quando o número de referências for zero, as páginas ocupadas pela estrutura podem ser reaproveitadas.

5.6 Gerente de Alocação de Espaço e Processamento

Em analogia com os computadores de Manchester e do MIT, é necessário um processo responsável pela gerência da memória livre e pelas reservas de espaço associadas aos pedidos de alocação. A gerência de memória livre é

relativamente simples, em decorrência da divisão da memória em páginas de tamanho fixo. Contudo a estratégia de alocação dessas páginas a estruturas pode ser complexa, se houver mais de uma Memória de Estruturas no computador.

Em razão das operações vetoriais, a estratégia de alocação pode levar em conta se a estrutura é um argumento para uma dessas operações. Nas operações vetoriais com duas estruturas-argumento, as reservas de espaço para uma estrutura-argumento podem ser feitas nas mesmas Memórias de Estruturas onde foram reservadas as páginas para a outra estrutura-argumento. Com isso, a operação vetorial pode ser tratada como um conjunto de sub-operações vetoriais em páginas, realizadas simultâneamente em todas as Memórias de Estruturas onde encontram-se as páginas das estruturas-argumento.

Para que esta associação possa ocorrer entre todas as páginas das estruturas-argumento, é necessário indicar ao gerente a operação e a identificação das estruturas, antes dos pedidos de alocação propriamente ditos. Se ocorrer que as páginas de uma operação vetorial não estão nas mesmas Memórias de Estruturas, uma das páginas pode ser dirigida para onde está a outra ou a operação pode realizar-se no anel como um conjunto de operações escalares. A conveniência de um modo de procedimento sobre o outro será determinada basicamente através do custo do transporte de uma página de uma Memória de Estruturas para outra. A fim de tornar mais rápida a comunicação entre estas Memórias de Estruturas, elas podem ser interligadas por barramentos organizados numa hierarquia.

Existe bastante flexibilidade sobre como e onde implementar o gerente. Para um computador com muitos anéis, vários gerentes devem estar ativos, cada um associado a um grupo de Memórias de Estruturas para evitar um *gargalo* de alocações de espaço devido ao número de processos-clientes que podem existir. A alternativa de implementar o gerente por *software* como um processo em execução num anel, ao invés de por *hardware* ou *firmware* usando processadores dedicados à tarefa, parece ser interessante pelo custo mais baixo de implementação e pelo paralelismo que pode ser explorado na sua execução. Esta implementação pode basear-se no estudo feito por Catto [16] para a programação da gerência de recursos num ambiente de fluxo de dados.

5.7 Protocolos

5.7.1 Protocolo entre o Anel e o Gerente

O protocolo entre o anel e o gerente de alocação de espaço e processamento disciplina as trocas de mensagens entre os processos em execução no anel e no gerente. Se o gerente situar-se numa unidade dedicada, as mensagens dirigidas a ele não são rotuladas. Contudo se for mais um processo do anel, as mensagens devem ser rotuladas e destinadas a pontos de entrada pré-definidos. Nesta exposição, por simplicidade, assume-se o primeiro caso.

No esquema seguinte apresenta-se um protocolo entre o anel e o gerente. No esquema, 'id' é o nome ou identificação de uma estrutura; '#páginas' o número de páginas que ela ocupa; P_i o endereço de uma página; 'contextoDestino' a associação entre rótulo e destino da ficha; 'ack' é uma ficha para indicar a conclusão de uma tarefa ou o recebimento de uma mensagem. Assim o protocolo é:

atividade	requisição	resposta
alocação	<id, #páginas, contextoDestino>	$<P_1>_{contextoDestino1} \cdots <P_q>_{contextoDestinoq}$
liberação	$<P_i>$	sem confirmação
collect	$<P_i>$	sem confirmação
definição	<id, $P_1 \ldots P_k$, contextoDestino >	$<ack>_{contextoDestino}$
consulta	<id, contextoDestino>	$<P_1>_{contextoDestino1} \cdots <P_p>_{contextoDestinoq}$

Cabe ao gerente de uma Memória de Estruturas, servindo a um grupo destas unidades, a tarefa de atender à requisição de espaço de um pedido de alocação. A resposta do gerente é um *stream* de páginas enviado ao anel. A distinção entre os contextos das fichas de resposta é estabelecida pelo índice do rótulo, isto é, as páginas são todas dirigidas para a mesma ativação de código e mesma instrução destino, mas o índice entre elas difere: P_i tem índice 'i'. Uma página é devolvida ao gerente quando ficar em desuso. Isto deverá ocorrer no momento em que o número de referências à estrutura a que pertence chegar a zero.

De modo a permitir flexibilidade e um certo grau de compartilhamento, um processo pode assumir a responsabilidade de associar um nome a um conjunto de páginas. Em caso de compartilhamento, a contagem de referências

tem de ser feita sobre cada página, para determinar as páginas compartilhadas e evitar que sejam removidas antes da hora.

Páginas que contêm uma estrutura podem ser determinadas através de uma consulta para o gerente com o nome da estrutura. A indicação para o gerente de que uma página contém todos os seus elementos é enviada por uma mensagem de *collect*. Esta mensagem é importante para dar início às operações sobre estruturas.

No esquema a seguir, contemplam-se as operações entre estruturas que podem ser realizadas com o auxílio do gerente. Neste esquema, 'op' identifica a operação e 'id' a estrutura. As operações consideradas são basicamente aquelas presentes nas três primeiras linhas da Tabela 5.2. O 'modo' estabelece se as estruturas-argumento podem ser descartadas, isto é, se a estrutura-resultado pode ou não aproveitar páginas das estruturas-argumento. É conveniente ressaltar que, na presença de mais de uma Memória de Estruturas, uma operação envolvendo duas estruturas deve ser solicitada ao gerente antes dos respectivos pedidos de alocação. Caso contrário, as páginas podem encontrar-se em Memórias de Estruturas distintas, sendo necessário o transporte de uma delas para onde a outra se encontra.

requisição	resposta
<op, modo, id, contextoDestino>	<id >contextoDestino ou <valor>contextoDestino
<op, modo, id_1, id_2, contextoDestino>	<id >contextoDestino ou <valor>contextoDestino

5.7.2 Protocolo entre o Gerente e a Memória de Estruturas

Como uma operação sobre uma estrutura é mapeada em operações sobre páginas o fato de uma requisição poder ocorrer antes de a estrutura estar pronta exige que o gerente só inicie uma operação envolvendo duas páginas no momento em que ambas estiverem prontas e as páginas para os resultados, reservadas. Com base nas operações propostas sobre as estruturas, as operações podem envolver uma ou duas páginas e apresentar como resultado uma página ou um valor.

O esquema a seguir ilustra este protocolo. Nele P_1, P_2 e P_3 são páginas; 'op' é a operação; id_{op} é a identificação da chamada; ack_id_{op} é uma confirmação indicando que a operação foi concluída; 'valor' é o resultado de uma operação que produz um escalar.

requisição	resposta
<op, P_1, P_2, P_3, id_{op} >	<ack_id_{op} >
<op, P_1, P_2, id_{op} >	<ack_id_{op} >
<op, P_1, id_{op} >	<ack_id_{op} >ou <ack_id_{op}, valor >

Se, ao receber uma mesagem solicitando uma operação vetorial, o processador da Memória de Estruturas estiver ocupado, a mensagem é colocada numa fila. Por isso, junto com a operação, o gerente remete uma identificação. Essa identificação será devolvida assim que a operação for concluída.

5.7.3 Protocolo entre o Anel e a Memória de Estruturas

O protocolo que propomos para a comunicação entre o anel e a Memória de Estruturas é apresentado abaixo:

atividade	requisição	resposta
sincronização	<P_i, #elementos, contexoDestino>	<ack>$_{contextoDestino}$
leitura	<P_j, deslocamento, contextoDestino>	<valor>$_{contextoDestino}$
escrita	<valor, P_j, deslocamento>	sem confirmação

Antes de o processo-produtor começar a preencher uma página envia-se para Memória de Estruturas uma mensagem dizendo quantos elementos (no esquema, '#elementos') a formarão e para onde o sinal indicando a conclusão da página deve ser enviado. Este sinal é usado para acordar o processo-consumidor para que este possa iniciar leituras ou para indicar ao gerente que a página da estrutura ficou pronta.

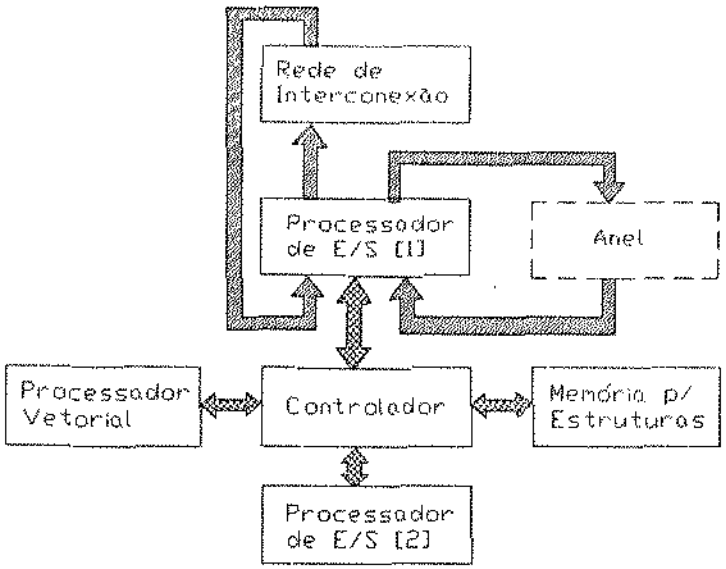


Figura 5.3: Organização para uma Memória de Estruturas com operações vetoriais

5.8 Organização de uma Memória de Estruturas

A Figura 5.3 apresenta a organização de uma Memória de Estruturas para suportar operações vetoriais. A Memória de Estruturas é convencionalmente constituída dos seguintes componentes:

- a memória para as estruturas;
- o controlador da memória para as estruturas;
- a interface de comunicação com o anel (assinalada com [1]).

A isso acrescentamos:

- o processador vetorial;
- a interface de comunicação com outras Memórias de Estruturas (assinalada com [2]).

Os elementos das estruturas são armazenados numa memória para estruturas contidas na unidade. Ela é formada por bancos de memória convencional, numa organização que deve empregar entrelaçamento para compensar as diferenças entre o ciclo de memória e o ciclo do processador vetorial, como

mencionamos ser prática normal na seção 5.2.1. Nesta memória são desnecessários *bits* de status associados a cada endereço porque contemplamos apenas o uso de estruturas do tipo estrito.

A interface de comunicação com o anel é responsável pela troca de mensagens entre o anel e a unidade. Este sub-sistema é constituído por um processador dedicado e uma memória local usada para implementar filas de entrada e saída. O processador da interface tem de distinguir entre os acessos de leitura, escrita e as instruções para o processador vetorial. Como a sincronização de estruturas estritas (*collect*) é realizada localmente, cada escrita participa deste processo. Uma forma simples e direta para realizar a sincronização é associar um contador de elementos ausentes para cada página de uma estrutura. Deste modo haverá um contador associado a cada página da Memória de Estruturas.

Nos computadores com mais de uma Memória de Estruturas pode ser interessante ligar estas unidades por uma outra via de comunicação além da rede de interconexão. Desta forma, a troca de páginas entre estas unidades fica favorecida. Junto a esta via é necessária uma outra interface dirigida às características envolvidas neste tipo de comunicação, que são basicamente a velocidade alta de transferência e o volume elevado de informação transferido.

O controlador da memória de estruturas realiza a arbitragem sobre quem tem o direito de acesso à memória em cada ciclo. As prioridades, em ordem decrescente, devem ser: interface com o anel, processador vetorial e interface com outras Memórias de Estruturas, se houver a via alternativa de comunicação que mencionamos anteriormente. Este esquema de prioridades pode introduzir mais *estados-de-espera* (*wait-states*) no processador vetorial do que se este tiver a maior prioridade. Contudo, convém lembrar que é o anel quem dirige o processamento e a opção vetorial presente na Memória de Estruturas é considerada um suporte adicional.

O processador vetorial, que sustentamos, dedica-se às operações lógicas e aritméticas sobre vetores situados na memória. Devido à granularidade imposta, estes vetores terão como tamanho máximo uma página. O processador é *pipelined* e deve ser organizado em arranjos que permitam encadeamento (*chaining*) e superposição (*overlapping*) de operações.

Capítulo 6

Resultados de Simulação

O capítulo anterior foi dedicado à descrição de operações locais sobre vetores numa Memória de Estruturas vetorial. Agora, pretende-se comprovar que o uso da localidade pode ser conveniente importando tanto numa melhora de desempenho do computador como num aumento de simplicidade na organização de suas partes. O foco da nossa análise está dirigido para a movimentação de dados e para a sincronização de estruturas estritas.

A base empírica desta pesquisa vem do uso de um simulador do computador a fluxo de dados de Manchester [13], uma ferramenta adequada para o estudo de computadores a fluxo de dados dinâmicos, empregada por algumas pesquisas recentes na área [15,39,41]. Como mencionamos na seção 3.5, a linguagem de programação adotada para o computador de Manchester é SISAL. O simulador e a linguagem SISAL constituem o nosso ambiente de programação.

A simulação integral da proposta da Memória de Estruturas do Capítulo 5 foge ao alcance deste trabalho, mas pretendemos avaliar o mecanismo básico para o suporte a estruturas (formação e consumo) que consiste do protocolo entre o anel e a Memória de Estruturas, apresentado na seção 5.7.3. As operações vetoriais propriamente ditas não são simuladas, mas o seu efeito é inferido. Adotamos esta sistemática porque será difícil com os instrumentos disponíveis comparar em todos os efeitos uma Memória de Estruturas vetorial com uma não vetorial e, do mesmo modo, difícil comparar o desempenho de um conjunto anel–Memória de Estruturas vetorial com outro conjunto anel–Memória de Estruturas não-vetorial. Faltam-nos informações para que o custo dos dois conjuntos possa ser equiparado de forma empírica

definitiva.

Para efetuarmos o trabalho, alteramos o simulador de Manchester para incorporar uma Memória de Estruturas que se diferencia da Memória de Estruturas de Manchester no modo de endereçamento e no suporte que fornece para a sincronização de estruturas estritas. O endereçamento é inteiramente baseado em páginas, ao contrário do endereçamento da Memória de Estruturas de Manchester, que é linear.

As diferenças entre a Memória de Estruturas proposta e a do computador de Manchester são as seguintes:

1. A sincronização das estruturas é realizada de modo local à Memória de Estruturas e transparente para o anel. Na Memória de Estruturas de Manchester esta sincronização ocorre no anel (veja a seção 4.1.6).
2. Na implementação de Manchester, a operação de armazenamento requer apenas uma instrução para a mensagem de escrita. O endereço de armazenamento é estabelecido pelo apontador e pelo índice no rótulo do argumento. Deste modo torna-se necessário apenas sincronizar o apontador da estrutura com o valor a ser escrito. Na implementação que efetuamos, o índice no rótulo é desvinculado do armazenamento e, por isso, para um acesso de escrita são necessárias três sincronizações para cada elemento de uma estrutura, a saber: estrutura, índice e valor. Deste modo, a formação da mensagem para a unidade de estruturas precisa de duas instruções ao invés de uma, pois a Unidade de Agrupamento pode sincronizar apenas duas fichas-argumento de cada vez.
3. Na Memória de Estruturas proposta, o armazenamento de estruturas multidimensionais é feito por meio de uma representação unidimensional. A memória é dividida em páginas e as estruturas são mapeadas nestas páginas. Na Memória de Estruturas de Manchester, estruturas multidimensionais são representadas a partir de diversas estruturas unidimensionais. Uma matriz, por exemplo, é um vetor cujos elementos são apontadores para outros vetores. Assim o número de acessos intermediários para ler ou escrever um elemento é proporcional à dimensão da estrutura. A primeira forma de representação é chamada *flat* (*achatada*) enquanto a segunda *non-flat* (*não-achatada*).
4. Particularidades de algumas operações implementadas na Memórias

de Estruturas de Manchester e na Memória de Estruturas proposta:

- na Memória de Manchester existe uma instrução ('fss', veja o apêndice A) que lê todos os elementos de um vetor, mas esta instrução não foi implementada na Memória de Estruturas proposta;
- uma operação para copiar um elemento de uma página em outra ('mse', veja o apêndice B), que não existe em Manchester, foi incorporada à nossa proposta.

Como não dispunhamos do fonte do gerador (IF1) de código SISAL, não foi possível alterar o compilador de forma a gerar código que reproduzisse diretamente as características da Memória de Estruturas proposta. Todos os programas nos testes de simulação a respeito desta unidade foram elaborados em linguagem de baixo nível. Em decorrência disto, os resultados dos testes são sensíveis aos seguintes aspectos:

- O código programado diretamente em linguagem primitiva é mais eficiente que o código traduzido.
- O tamanho dos problemas testados impede a avaliação do custo extra da gerência de páginas. As estruturas em todos os exemplos são menores do que uma página, devido à complexidade adicional do código que teríamos de escrever para incorporar estruturas de múltiplas páginas.
- Nos programas para a Memória de Estruturas proposta não foi introduzido código para efetuar a contagem de referências de uma estrutura. Isto se deve ao fato de que os exercícios simulados, pela opção do item anterior e pelo tamanho do programas, não necessitam deste requisito.

Os exercícios simulados representam funções simples para que o comportamento que se deseja observar fique aparente. Cada exercício possui uma característica interessante e todos dizem respeito a funções que ocorrem naturalmente na prática. Os exemplos são:

1. criação de um vetor;
2. transformação de um vetor;
3. produto de dois vetores;
4. criação de uma matriz;
5. transposição de uma matriz;
6. produto de duas matrizes.

De certa forma os exercícios são cumulativos, ou seja, todos são analisados individualmente mas têm como base um exercício anterior. O produto de dois vetores por exemplo, usa o exercício para criação de vetores, o mesmo ocorrendo no produto de duas matrizes que usa os exercícios de transposição de uma matriz e criação de matrizes. Deste modo, são apresentados os resultados da simulação integral sendo a análise particular de cada uma das funções discutida no texto.

Por questão de clareza, apenas a função de interesse é apresentada em SISAL. O restante do programa é omitido. Os computadores simulados apresentam apenas um conjunto anel-Memória de Estruturas. Na fase atual dos estudos de aperfeiçoamento dos mecanismos básicos de suporte a estruturas de dados, os exemplos não justificam a complexidade de um sistema com múltipos conjuntos.

6.1 Parâmetros de Avaliação

Adotamos na comparação dos exercícios, parâmetros de avaliação normalmente usados para caraterizar o comportamento de um programa num sistema de processamento paralelo. Estes parâmetros dizem respeito a uma simulação idealizada, onde todas as instruções têm tempo de execução exatamente igual a um ciclo (*time-step*) do simulador. A partir dessa correspondência, existe uma associação entre número de ciclos e número de instruções executadas. Os parâmetros são:

S_1: Tempo máximo de execução do programa ou número total de instruções executadas na avaliação do programa.

S_{inf}: Tempo mínimo de execução do programa. Também se pode considerar este valor como o número de instruções diretamente dependentes que têm de ser executadas para produzir o resultado. Esta seqüência de instruções é conhecida como o *caminho crítico* do grafo equivalente à execução do programa, o *grafo de execução* [15]. Visualmente S_{inf} representa a altura do grafo de execução quando este é desenhado em diferentes níveis e as instruções em dependência (instruções ligadas por arestas do grafo) são colocadas uma sobre as outra, enquanto as independentes adjacentemente. O cálculo de S_{inf} pressupõe a existência de infinitos processadores, para que todas as instruções no mesmo nível sejam executadas simultaneamente, e desta forma não ocorra que instruções independentes no mesmo nível sejam executadas em instantes

diferentes, o que acarretaria num outro grafo de execução. Generalizando, S_p é a altura do grafo de execução para p processadores.

π: Paralelismo médio do programa definido como a razão entre S_1 e S_{inf}. Quanto maior π maior o paralelismo do programa.

Tráfego: Número total de fichas que circulam pelo anel durante a execução do programa.

#acc: Número total de acessos à Memória de Estruturas durante a execução do programa.

#acc(RFC): Número total de acessos à Memória de Estruturas associados às contagens de referências sobre estruturas efetuados durante a execução do programa.

#alloc: Número total de alocações de espaço realizadas durante a execução do programa.

O resultado da simulação apresenta valores para esses parâmetros como consta nas Tabelas 6.1 a 6.13. Nessas tabelas n é o tamanho da estrutura. Para vetores, n é o número de elementos; para matrizes, n é a ordem da matriz. Quando os resultados apresentam regularidade, na forma de uma função algébrica, ela é exposta em $f(n)$.

6.2 Comparação entre Implementações

Nesta seção, vamos mostrar por meio dos resultados das simulações a importância da localidade em operações sobre estruturas numa Memória de Estruturas. Como nossa ênfase está em estruturas estritas, a importância de um mecanismo de sincronização adequado fica evidente. Infere-se, a partir dos resultados, que a granularidade fina não favorece os casos onde há regularidade nas operações pela repetição de procedimentos decorrentes de se tratar uma operação sobre um objeto como um conjunto de operações individuais sobre os elementos desse objeto.

6.2.1 Criação de um Vetor

O programa na Figura 6.1 é um exemplo da criação de um vetor de quatro elementos.

As Tabelas 6.1 e 6.2 apresentam os resultados da simulação obtidos para a Memória de Estruturas de Manchester e Memória de Estruturas proposta,

```
type intvec = array [integer]

function mkvec (returns intvec)

   array [1: 1,2,3,4]   % esta expressao cria um vetor de quatro
                        % elementos: 1,2,3 e 4. O indice do primeiro
end function            % elemento e 1.
```

Figura 6.1: Criação de um vetor

n	1	2	4	8	$f(n)$
S_1	35	42	53	77	$6n + 29$
S_{inf}	18	18	18	18	18
π	1.9	2.3	2.9	4.3	$0.33n + 1.61$
Tráfego	41	50	65	97	$8n + 33$
#acc	2	3	5	9	$n + 1$
#alloc	1	1	1	1	1

Tabela 6.1: Criação de um vetor no computador de Manchester

respectivamente. Todas as medidas favorecem a última. Notar que S_1 é proporcional a $6n$ no computador de Manchester e a $3n$ na aquitetura proposta como decorrência da sincronização local. O tráfego de fichas segue o comportamento de S_1 e assim também cai pela metade de um a outro caso.

Nos programas para as duas Memórias de Estruturas foram empregadas as seguintes instruções para a criação estática de um vetor (isto é, quando o tamanho e os elementos do vetor são conhecidos antes da compilação do programa):

- Uma instrução foi usada para a definição do elemento, considerado um literal.
- Uma instrução foi usada para a definição do índice do elemento.
- Uma instrução foi usada para o armazenamento.

Assim se o tamanho do vetor for n, são necessárias $3n$ instruções para a sua criação porque estas instruções agrupam os elementos do vetor num *stream*,

convertem-nos em mensagens e enviam-nos para a Memória de Estruturas. Böhm [14] sugere que a criação estática de um vetor pode tomar apenas n instruções se a alocação da estrutura for realizada estaticamente porque, neste caso, os endereços para armazenar os elementos são calculados durante a compilação. Acreditamos que o mesmo resultado possa ser obtido na Memória de Estruturas proposta. De fato, a alocação estática pode ser dispensável se for criada uma instrução que permita dois literais, definindo o valor do elemento e o seu deslocamento na página. Deste modo apenas a página é o argumento para as instruções.

No computador de Manchester (em contraste com a criação estática) a criação dinâmica de uma estrutura requer apenas $3n$ instruções. Os elementos provenientes de um *stream* são armazenados diretamente sendo necessária apenas a sincronização global. Por isso, realizando-se a sincronização localmente, S_1 diminui em $3n$ instruções relativamente ao modelo de Manchester. Para efetuar a sincronização, empregamos uma instrução especial ('spg', veja o Apêndice B) que estabelece quantos elementos são escritos na estrutura. A sincronização é estabelecida com base na página. Se a estrutura couber toda em uma página, apenas uma sincronização é necessária; do contrário deverão ser sincronizadas todas as páginas.

n	1	2	4	8	$f(n)$
S_1	20	23	29	41	$3n + 17$
S_{inf}	11	11	11	11	11
π	1.8	2.1	2.6	3.7	$0.27n + 1.54$
Tráfego	26	30	38	54	$4n + 22$
#acc	4	5	7	11	$n + 3$
#alloc	1	1	1	1	1

Tabela 6.2: Criação de um vetor usando sincronização local

6.2.2 Transformação de um Vetor

A função para a transformação de um vetor está na Figura 6.2. Transformar significa, neste caso, obter uma estrutura a partir de outra. A estrutura-argumento permanece inalterada.

```
function tvec (v: intvec returns intvec)
                    % esta expr retorna vetor
   v [1: 1]         % com os mesmos elementos
                    % de v, mas com o valor 1
end function        % na posicao [1]
```

Figura 6.2: Transformação de um vetor

n	1	2	4	8	$f(n)$
S_1	79	89	109	149	$10n + 69$
S_{inf}	37	37	37	37	37
π	2.1	2.4	2.9	4.0	$0.27n + 1.86$
Tráfego	99	113	141	197	$14n + 85$
#acc	7	10	16	28	$3n + 4$
#alloc	2	2	2	2	2

Tabela 6.3: Transformação de um vetor no computador de Manchester.

Os resultados da simulação aparecem nas Tabelas 6.3 e 6.4 e são favoráveis ao modelo com sincronização local. Os resultados da simulação para o processo de transformação de um vetor se obtém subtraindo os respectivos valores das Tabelas 6.1 e 6.3, e das Tabelas 6.2 e 6.4 tendo em vista que os programas de transformação empregam as funções para a criação de vetores. Pode-se notar que a transformação de um vetor é um processo que tem S_1 proporcional a $4n$ para o computador de Manchester e para um computador com a arquitetura que propusemos. Entretanto o tráfego e o número de acessos são menores neste último em virtude da realização de operações locais na Memória de Estruturas: o vetor é copiado localmente na Memória de Estruturas por meio de um conjunto de mensagens, onde cada uma copia o valor de uma posição para a outra (usando a instrução 'mse', veja o Apêndice B).

A transformação de uma estrutura é implementada no computador de Manchester como uma cópia integral seguida de uma atualização. A estrutura é toda extraída, levada ao anel e depois de volta à Memória de Estruturas, numa outra área de memória. A estrutura resultado é alterada na posição e valor determinados pela função de transformação.

Ao invés de copiar um elemento de cada vez de uma posição para a outra, a Memória de Estruturas proposta poderia realizar a cópia integral de uma página de cada vez. Deste modo os ganhos medidos seriam comparativamente maiores pois este processo poderia ser assistido por *hardware* DMA (acesso-direto-a-memória).

Ganhos expressivos podem advir de uma otimização do código para efetuar uma transformação. Observe-se que na função deste exemplo, o espaço e os elementos da estrutura-argumento podem ser aproveitados diretamente para a formação da estrutura resultado e a reatribuição efetuada apenas para o valor requerido, porque a função de transformação é o único consumidor da estrutura-argumento. Por isso ela pode ser alterada diretamente nas posições necessárias dando origem a um processo de transformação significativamente mais rápido.

n	1	2	4	8	$f(n)$
S_1	52	59	73	101	$7n + 45$
S_{inf}	32	32	32	32	32
π	1.6	1.8	2.3	3.2	$0.21n + 1.4$
Tráfego	71	82	104	148	$11n + 60$
#acc	10	12	16	24	$2n + 8$
#alloc	2	2	2	2	2

Tabela 6.4: Transformação de um vetor de acordo com a nova proposta.

6.2.3 Produto de Dois Vetores

O programa do produto de dois vetores realiza a criação de dois vetores de mesmo tamanho e logo a seguir cria um terceiro vetor que é o produto ponto-a-ponto dos primeiros. A função que efetua o produto ponto-a-ponto encontra-se na Figura 6.3. Os resultados da simulação são apresentados nas Tabelas 6.5 e 6.6.

A explicação do valor de S_1, para o computador de Manchester, está em que $12n$ instruções são necessárias para criar e sincronizar as estruturas iniciais; n instruções para realizar o produto ponto-a-ponto; n instruções para efetuar

```
function vp (v1, v2: intvec returns intvec)

   for e1 in v1 dot e2 in v2
       ep:= e1 * e2
       returns array of ep
   end for

end function
```

Figura 6.3: Produto de dois vetores

n	1	2	4	8	$f(n)$
S_1	109	125	157	221	$16n + 93$
S_{inf}	32	32	32	32	32
π	2.1	3.9	4.9	6.9	$0.5n + 2.9$
Tráfego	133	156	202	294	$23n + 109$
#acc	8	13	23	43	$5n + 3$
#alloc	3	3	3	3	3

Tabela 6.5: Produto de dois vetores no computador de Manchester

o armazenamento e $2n$ instruções para efetuar a sincronização da estrutura final. Esta relativa eficiência observada no valor global de S_1 se deve a que o acesso aos n elementos de uma estrutura, ao invés de requerer n instruções, requer apenas uma instrução ('fss', veja o Apêndice A) produzindo n mensagens. No conjunto anel-Memória de Estrutura que implementamos, uma operação otimizadora deste tipo não está presente e efetivamente requer n instruções para efetuar o acesso aos n elementos do vetor.

Empregando o modelo de sincronização local, o valor de S_1 torna-se proporcional a $11n$. Desse valor, $6n$ devem-se à criação dos dois vetores, enquanto a função para o produto ponto-a-ponto propriamente dita requer apenas $5n$ instruções, ou seja, $2n$ instruções para a leitura dos $2n$ elementos dos dois vetores, n instruções para o produto ponto-a-ponto e $2n$ instruções para o armazenamento. Notar que o armazenamento toma duas instruções como explicamos anteriormente. Se tivéssemos adotado a otimização presente no computador de Manchester (que efetua o acesso aos elementos da estrutura

executando apenas uma instrução), S_1 reduzir-se-ia em $2n$ instruções no modelo de sincronização local e o tráfego de flechas no anel em $6n$.

Havendo uma instrução específica destinada ao produto de dois vetores localmente à Memória de Estruturas, então, independentemente da observação anterior, pode-se estabelecer que S_1 será proporcional apenas a $6n$, ao invés de $11n$ como mostra a simulação, porque $6n$ instruções são necessárias para a criação dos dois vetores. Neste caso o ônus de S_1 se transfere do anel para a Memória de Estruturas. Esta operação realizada localmente é mais eficiente porque as sincronizações envolvidas no cálculo de endereços são estabelecidas implicitamente e a distância percorrida por um dado entre origem e destino se reduz.

n	1	2	4	8	$f(n)$
S_1	56	67	89	133	$11n + 45$
S_{inf}	20	20	20	20	20
π	2.8	3.3	4.4	6.7	$0.55n + 2.25$
Tráfego	77	95	131	203	$18n + 59$
#acc	14	19	29	49	$5n + 9$
#alloc	3	3	3	3	3

Tabela 6.6: Produto de dois vetores empregando sincronização local.

O número de acessos à Memória de Estruturas obtido para os dois exemplos, apresentado nas Tabelas 6.5 e 6.6, é proporcional a $5n$. Isto se deve a que em ambos os modelos são efetuados n acessos para armazenar e n acessos para ler os elementos da primeira estrutura; $2n$ acessos são efetuados para a segunda estrutura em analogia à primeira; e n acessos são efetuados para armazenar a estrutura final.

6.2.4 Criação de uma Matriz

O programa para a criação de uma matriz de ordem 3 está ilustrado na Figura 6.4.

Os resultados da simulação estão expostos na Tabela 6.7 para o computador de Manchester e na Tabela 6.2 para o computador com a Memória de

```
type intmat = array [intvec]

function mkmat (returns intmat)

   array [1: array [1: 1,0,0],        % esta expressao cria
              array [1: 0,1,0],       % uma matriz identidade
              array [1: 0,0,1]]       % 3 x 3

end function
```

Figura 6.4: Criação de uma matriz

n	1 x 1	2 x 2	3 x 3	4 x 4	$f(n)$
S_1	77	135	205	287	$6n^2 + 40n + 4$
S_{inf}	31	31	31	31	31
π	2.5	4.4	6.6	9.3	$0.19n^2 + 1.29n + 0.12$
Tráfego	94	171	260	367	
#alloc	2	3	4	5	$n + 1$
#acc	6	13	22	33	$n^2 + 4n + 1$
#acc(RFC)	1	2	3	4	n

Tabela 6.7: Criação de uma matriz no computador de Manchester

Estruturas proposta. No último, a criação estática de uma matriz não se distingue daquela de um vetor.

Uma característica da tradução de SISAL no computador a fluxo de dados de Manchester é o emprego de *arrays* de apontadores para representar *arrays* multidimensionais, isto é, Manchester não emprega um espaço contíguo de memória para o armazenamento de estruturas multidimensionais. Como dissemos, as estruturas são não-achatadas. Para representar uma estrutura $n \times n$, por exemplo, são necessárias $n+1$ áreas de memória e, conseqüentemente, $n+1$ alocações. Sendo assim, a criação de uma matriz de ordem n é equivalente à criação de $n+1$ vetores. Uma outra conseqüência da forma de representação não-achatada é que o acesso a um escalar numa estrutura multidimensional requer d acessos, onde d é o número de dimensões da estrutura.

Em virtude disso, no computador de Manchester este programa tem um comportamento quadrático em n para S_1. Observe-se que este comportamento é linear no computador proposto.

No modelo que adotamos, a representação achatada faz com que as estruturas multidimensionais desdobrem-se num único vetor. Assim o acesso a um escalar pode ser feito diretamente, após o cálculo do seu endereço.

Apesar das limitações, a representação não-achatada encontra compensação pois favorece o compartilhamento de estruturas multidimensionais. A construção *for* na forma insensível ao índice (*forall*) também aproveita esta forma de representação, baseando-se numa tradução que explora o uso de geradores para acesso aos elementos da estrutura.

6.2.5 Transposição de uma Matriz

Transposição no Computador de Manchester

Existem dois modos para se realizar a transposição de uma matriz em SISAL: o modo sensível ao índice e o insensível ao índice. As funções para estes dois modos são apresentadas na Figura 6.5. Os resultados da simulação para o computador de Manchester encontram-se na Tabela 6.8.

A forma insensível ao índice é mais eficiente porque o cálculo do índice pode ser suprimido, uma vez que é implícito. Geralmente pode-se transformar uma das dimensões da estrutura num *stream* e realizar uma operação de *scatter*, isto é, inserir cada elemento numa ativação distinta do código da função.

Subtraindo-se das tabelas indicadas as instruções necessárias para a criação da matriz-argumento, os valores de S_1 e S_{inf} em cada um dos casos são respectivamente:

- transposição sensível: $S_1 = 7n^2 + 59n + 82$; $S_{inf} = 44$
- transposição insensível: $S_1 = 7n^2 + 52n + 81$; $S_{inf} = 41$

De maneira análoga a S_1, o tráfego e o número de acessos são medidas que dependem do quadrado da ordem da matriz.

```
function transpose_S (m: intmat; n: integer
                      returns intmat)

   for i in 1,n cross j in 1,n
       e:= m [j,i]
       returns array of e
   end for

end function %transpose_S


function transpose_I (mat: intmat; n: integer
                      returns intmat)
   for i in 1,n
       newrow:= for row in mat              % acc as linhas (forall)
                    col_e:= row [i]         % acc i-esimo elem linha
                    returns array of col_e  % formacao da coluna
                end for                     % (col -> lin nova mat)
     returns array of newrow                % formacao do vetor cols
   end for

end function %transpose_I
```

Figura 6.5: Transposição de matriz nas formas sensível e insensível ao índice

Antes de discutirmos a transposição de uma matriz para a implementação proposta é conveniente uma nova discussão sobre processos *geradores*.

Estudo sobre os geradores

Os geradores foram apresentados na seção 3.5.2 como processos que produzem *streams*, e que podem ser usados para determinar a seqüência de índices no acesso à uma estrutura.

Devido à representação de estruturas de forma não-achatada, em Manchester, os geradores implementados em SISAL produzem seqüências de valores individuais, ou seja, não produzem pares, triplas, etc. Desse modo, para acesso aos elementos de uma matriz de ordem n, por exemplo, são necessários $n+1$ geradores: um para a primeira dimensão e n geradores para a dimensão seguinte. Esta implementação é natural e muito eficiente quando o acesso aos elementos da estrutura se faz de acordo com a forma de armazenamento,

n	1 x 1	2 x 2	3 x 3	4 x 4	$f(n)$
S_1	198	336	501	692	$13n^2 + 99n + 86$
S_{inf}	75	75	75	75	75
π	2.6	4.4	6.6	9.2	$0.17n^2 + 1.32n + 1.14$
Tráfego	260	454	691	970	$21n^2 + 131n + 108$
#alloc	4	6	8	10	$2(n + 1)$
#acc	24	52	92	144	$6n^2 + 10n + 8$
#acc(RFC)	6	10	14	18	$4n + 2$
n	1 x 1	2 x 2	3 x 3	4 x 4	$f(n)$
S_1	190	321	479	663	$13n^2 + 92n + 85$
S_{inf}	72	72	72	72	72
π	2.6	4.4	6.6	9.2	$0.18n^2 + 1.27n + 1.18$
Tráfego	246	424	641	986	
#alloc	4	6	8	10	$2(n + 1)$
#acc	24	52	92	144	$6n^2 + 10n + 8$
#acc(RFC)	6	10	14	18	$4n + 2$

Tabela 6.8: Transposição em Manchester empregando as funções sensível e insensível ao índice respectivamente

ou seja, por exemplo, para uma matriz armazenada pela ordem das linhas (o vetor da primeira dimensão é um vetor de linhas) a forma de acesso codificada no programa também se faz pela ordem das linhas.

Para uma representação achatada de armazenamento, o acesso a estruturas multidimensionais requer um único gerador que produz tuplas de índices. A abordagem descrita no parágrafo anterior não é satisfatória.

A codificação de um gerador em alto nível é insatisfatória em razão da sua pouca eficiência. A Figura 6.6 ilustra o exemplo do que seria um gerador para acesso aos elementos de uma matriz pela ordem das linhas. Os resultados da simulação desta função estão na Tabela 6.9. Para obter o valor real de S_1 devem ser subtraídas $2n^2$ instruções correspondentes à saída do programa. Os valores elevados, tanto de S_1 como de S_{inf}, se devem à criação dos contextos para cada uma das iterações da construção *for*.

```
type intstm = stream [integer]

function gen_ij (n: integer
                 returns intstm, intstm)

   for initial i,j:= 1,1
       until   (i = n & j = n)
       repeat  i,j:= if old j < n
                        then old i,     old j + 1
                        else old i + 1, 1
                     end if
       returns stream of i stream of j
   end for

end function %gen_ij
```

Figura 6.6: Gerador dos pares de índices para acesso a uma matriz

n	1 x 1	10 x 10	20 x 20	30 x 30	$f(n)$
S_1	29	2612	10422	41642	$26n^2 + n + 2$
S_{inf}	14	1103	4403	9903	$11n^2 + 13$
π	2.1	2.4	2.4	2.4	
Tráfego	36	3708	14818	33328	$37n^2 + n - 2$

Tabela 6.9: Gerador gen_ij (n)

Para testar a solução proposta, foi implementado um gerador eficiente em linguagem primitiva. Os resultados da simulação com esse gerador estão na Tabela 6.10. O gerador apresenta S_1 linear em n (devem ser subtraídas as $2n^2$ instruções relativas à saída do programa) ao invés de ser quadrático como se viu ser o gerador de alto nível. Isto se deve, em parte, à criação de duas novas instruções que foram concebidas para implementar sub-geradores e de uma nova instrução, que emula um contador. Com isto, é possível implementar um gerador que produz tuplas num único contexto.

n	1 x 1	10 x 10	20 x 20	30 x 30	$f(n)$
S_1	28	370	1130	2290	$2n^2 + 16n + 10$
S_{inf}	15	69	129	189	$6n + 9$
π	1.9	5.4	8.8	12.1	
Tráfego	36	441	1271	2501	$2n^2 + 23n + 14$

Tabela 6.10: Gerador em baixo nível equivalente a gen_ij (n)

n	1 x 1	2 x 2	3 x 3	4 x 4	5 x 5	6 x 6	$f(n)$
S_1	74	134	222	338	482	654	$14n^2 + 18n + 42$
S_{inf}	23	29	35	41	47	53	$6n + 17$
π	3.2	4.6	6.3	8.2	10.3	12.3	
Tráfego	99	185	311	477	683	929	$20n^2 + 26n + 53$
#alloc	2	2	2	2	2	2	2
#acc	7	16	31	52	79	112	$3n^2 + 4$

Tabela 6.11: Transposição de uma matriz empregando geradores de índice e sincronização local

Transposição na Memória de Estruturas Proposta

Subtraindo os valores obtidos para a criação da matriz dos valores constantes da Tabela 6.11, o valor de S_1 para transposição na forma achatada é $11n^2 + 18n + 75$, onde se tem que $2n^2$ instruções são provenientes da saída do gerador; $6n^2$ são provenientes função de acesso, metade para a leitura, metade para a escrita; n^2 são provenientes da leitura da matriz argumento; e $2n^2$ da criação da matriz resultado. O valor de S_{inf} é basicamente o do gerador uma vez que este predomina para n grande.

Observa-se que o valor de total de S_1 é quase o mesmo entre as duas abordagens, pela ineficiência do processo de criação no computador de Manchester. A transposição em si é mais eficiente em Manchester, pois S_1 é menor. Entretanto, deve-se levar em consideração que, em Manchester, o número de alocações é maior, bem como o número de acessos. Além disto, o fato de π ser linear no tamanho do problema, ao invés de quadrático, pode ser considerado uma vantagem, porque o paralelismo o acompanha de modo proporcional.

No exemplo da transposição de matriz o ônus da função de acesso é evidente. Apesar de mais elaborada, a inclusão de uma operação deste tipo de modo local à Memória de Estruturas garantiria a redução do ônus. A ressalva é que a estrutura teria de estar, toda ela, numa página.

6.2.6 Produto de Duas Matrizes

Produto de Matrizes no Computador de Manchester

```
function mp_S (m1, m2: intmat; n: integer
               returns intmat)

   for i in 1,n cross j in 1,n
       e:= for k in 1,n
               ep:= m1 [i,k] * m2 [k,j]
               returns value of sum ep
           end for
       returns array of e
   end for

end function %mp_S


function mp_I (m1, m2: intmat
               returns intmat)
   let
      m2t: intmat:= transpose2 (m2, array_size (m2))
   in
      for r1 in m1 cross r2 in m2t
          e:= for e1 in r1 dot e2 in r2
                  ep:= e1 * e1
                  returns value sum of ep
              end for
          returns array of e
      end for
   end let

end function %mp_I
```

Figura 6.7: Produto de matrizes nas formas sensível e insensível ao índice

Para o produto de duas matrizes em SISAL, como no exemplo da transposição, existem as formas sensível e insensível ao índice. As funções do produto encontram-se na Figura 6.7. O algoritmo para o produto de duas matrizes quadradas de ordem n tem como característica principal $2n^3$ acessos efetuados para a leitura dos elementos e $2n^3$ operações aritméticas provenientes de n^2 produtos internos entre as linhas da primeira matriz com as colunas da segunda. Para se empregar a forma insensível ao índice, é necessário transpor a segunda matriz para colocá-la no modo adequado a efetuar os produtos internos mencionados anteriormente. O produto interno efetua-se como se fosse um produto ponto-a-ponto.

Os valores obtidos da simulação destes exemplos estão na Tabela 6.12.

n	1 x 1	2 x 2	3 x 3	4 x 4	5 x 5	$f(n)$
S_1	328	710	1328	2248	3512	
S_{inf}	91	93	95	97	97	
π	3.6	7.6	13.9	23.1	36.2	
Tráfego	443	1021	2013	3551	5729	
#alloc	6	9	12	15	18	$3(n+1)$
#acc	35	102	247	506	915	$6n^3+3n^2+16n+8$
#acc(RFC)	10	17	24	31	38	$7n+3$
n	1 x 1	2 x 2	3 x 3	4 x 4	5 x 5	$f(n)$
S_1	303	614	1061	1662	2410	
S_{inf}	80	82	84	86	86	
π	3.7	7.4	12.6	19.3	28.0	
Tráfego	396	824	1458	2334	3449	
#alloc	6	9	12	15	18	$3(n+1)$
#acc	35	82	163	290	475	$2n^3+5n^2+18n+10$
#acc(RFC)	10	17	24	31	38	$7n+3$

Tabela 6.12: Produto de matrizes nas formas sensível insensível ao índice

Os valores de S_1 e S_{inf} obtidos a partir da simulação não deram origem a um polinômio exato possivelmente em razão das otimizações de código que ocorreram durante o processo de tradução. Por causa disto, apresentamos os resultados para cada uma das funções publicados em [14]:

n	1 x 1	2 x 2	3 x 3	4 x 4	5 x 5	$f(n)$
S_1	149	312	823	1574	2697	$12n^3 + 42n^2 + 13n + 82$
S_{inf}	33	42	51	60	69	$9n + 24$
π	4.5	7.4	16.1	26.2	39.0	
Tráfego	216	602	1408	2778	4856	$24n^3 + 66n^2 + 20n + 106$
#alloc	3	3	3	3	3	3
#acc	11	34	87	182	331	$2n^3 + 3n^2 + 6$

Tabela 6.13: Produto de matrizes na Memória de Estruturas proposta

- forma sensível: $S_1 = 7n^3 + 47n^2 + 71n + 236$; $S_{inf} = 6n + 95$
- forma insensível: $S_1 = 3n^3 + 36n^2 + 93n + 226$; $S_{inf} = 6n + 101$

É interessante observar na Tabela 6.12 como a forma insensível ao índice também é eficiente com relação ao número de acessos. Isto se deve a algo como um acesso em etapas, proveniente do emprego de geradores. Ao invés de realizar dois acessos para cada elemento (linha e depois coluna), lêem-se todas as linhas e depois todos os elementos de cada linha. Numa organização por linha, cada um dos elementos do vetor de linhas de cada matriz é copiado n vezes e depois são realizados n acessos a cada uma das suas linhas como um todo, ou seja, todos os elementos da linha são lidos conjuntamente. Para tornar a operação de cópia eficiente, existe uma instrução ('prl' [14]) no computador de Manchester dedicada a este propósito que copia um valor repetidas vezes colocando-o sob a forma de um *stream*.

Convém ressaltar que foi necessário um estudo cuidadoso sobre este exercício por parte do grupo de Manchester para obter as otimizações que conduziram aos resultados da forma insensível [14,41].

Produto de Matrizes na Memória de Estruturas Proposta

Os resultados da simulação para o produto de matrizes num computador com a arquitetura proposta encontram-se na Tabela 6.13. O esquema do código para este exemplo é apresentado na Figura 6.8. Os resultados obtidos, na maioria dos parâmetros, são piores do que no computador de Manchester.

O valor de S_1 para a arquitetura proposta é mais elevado basicamente devido à função de acesso que requer 3 instruções por acesso, totalizando $3n^3$

instruções para cálculo de endereços por matriz-argumento. Assim, para a leitura dos elementos, são necessárias $8n^3$ instruções. Este valor é maior que o valor de S_1 obtido em qualquer um dos casos de Manchester. Como mencionado na introdução desta seção, $2n^3$ instruções são aritméticas de produto e soma; e $2n^3$ instruções são de controle para acúmulo de valor. As n^3 instruções restantes são para a criação da matriz resultado.

Apesar dos geradores implementados em baixo nível para a produção das seqüências de índices serem eficientes, o ônus da função de acesso parece ser uma limitação da abordagem que propomos.

Mais uma vez, saliente-se, toda a operação pode ser realizada localmente na Memória de Estruturas. A representação achatada, neste caso, favorece esta abordagem. Mas haverá de novo uma limitação vinculada ao tamanho da matriz. No caso de matrizes maiores do que uma página, uma opção de tratamento é a extração das linhas e colunas das matrizes-argumento de modo a se realizarem os produtos linha-coluna como produtos internos.

6.3 Comentários

Ao longo deste capítulo, pôde-se observar o comportamento empírico das operações sobre estruturas realizadas tanto na Memória de Estruturas de Manchester quanto na Memória de Estruturas proposta. As abordagens diferem muito no que diz respeito à forma de representação (achatada e não-achatada) e à forma de sincronização de uma estrutura estrita (*collect*). A principal característica observada foi o valor elevado da medida S_1. Algumas das conclusões que podem ser estabelecidas são:

1. É conveniente que a sincronização de estruturas estritas seja realizada de modo local à Memória de Estruturas, porque isto diminui o número total de instruções executadas (S_1) bem como o tráfego de fichas no anel.

2. A representação não-achatada comporta-se melhor do que a achatada quando podem ser usadas estruturas de controle insensíveis ao índice. Neste caso o acesso à estrutura ao longo de uma dimensão (isto é, variando um de seus índices e mantendo os demais fixos) pode ser realizado com eficiência, se a forma de acesso no programa obedecer à forma de disposição dos elementos na memória. A otimização feita no computador de Manchester, neste caso, é o transporte para o anel de todos os

elementos do vetor por meio de uma única instrução. Ao se usar esta instrução, os cálculos dos índices são suprimidos, pois os elementos do vetor encontram-se em endereços consecutivos de memória.

3. As formas achatada e não-achatada comportam-se de modo semelhante nas estruturas de controle sensíveis ao índice. Na forma achatada, o custo principal do processamento recai sobre a função de acesso, enquanto na não-achatada sobre os acessos intermediários. O benefício da primeira forma, em todos os exemplos, é um número menor de alocações – todas de tamanho fixo – e um número menor de acessos à Memória de Estruturas.

4. O modelo de fluxo de dados explora bem qualquer tipo de paralelismo presente num algoritmo. A medida do paralelismo pode ser caracterizada pela relação S_1/S_{inf} (π). Nos exemplos apresentados, π é sempre crescente, pois S_{inf} aumenta numa proporção menor do que S_1. Em fluxo de dados, o fato de π aumentar com o tamanho do problema nem sempre é uma vantagem, se o aumento não for linear. Isto ocorre porque podem ser exigidos mecanismos de controle para adequar o paralelismo do problema aos recursos presentes no *hardware* (veja-se [18,39]). Em virtude dos valores medidos de π, pode-se dizer que o custo do paralelismo no anel será mais caro que o do mesmo paralelismo na Memória de Estruturas. Para esta conclusão também contribui a forma como são realizadas as sincronizações dos argumentos para uma operação.

5. Operações vetoriais locais na Memória de Estruturas devem proporcionar uma melhora da eficiência do processamento:

   - algumas operações repetitivas sobre vetores são transferidas do anel para uma unidade mais adequada a este tipo de operação;
   - o percurso da informação é reduzido porque os dados permanecem na própria Memória de Estruturas; e
   - o tempo de processamento pode ser desprezível nos casos em que estiver sobreposto ao tempo de acesso à memória, em decorrência do tempo de ciclo mais baixo do processador vetorial.

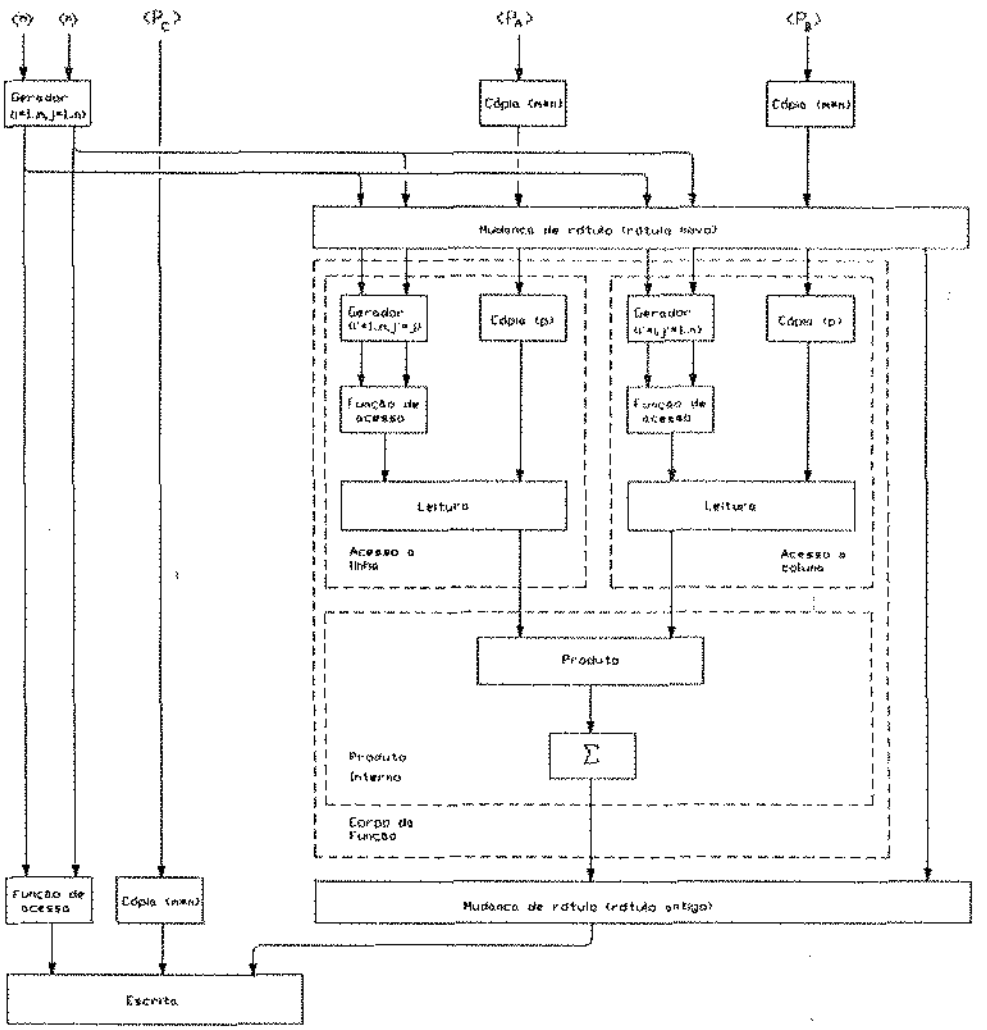


Figura 6.8: Esquema do produto de matrizes codificado em baixo nível. Neste esquema A é uma matriz m x p; B uma matriz p x n; C é a matriz-resultado.

Capítulo 7

Conclusão

Este trabalho abordou alguns dos assuntos mais importantes relacionados às estruturas de dados em computadores a fluxo de dados. Apresentamos no início do trabalho diversos conceitos relacionados à teoria para, em seguida, discutir dois exemplos de implementação de unidades dedicadas ao armazenamento de estruturas, as Memórias de Estruturas dos computadores de Manchester e do MIT.

No Capítulo 5 propusemos a produção de operações locais em uma unidade de armazenamento de estruturas. O Capítulo 6 apresentou os resultados de uma simulação para exercícios que verificaram o comportamento desta proposta nos seus mecanismos básicos. Nossa conclusão é que a arquitetura com operações locais comporta resultados promissores que convidam a uma intensificação da pesquisa nessa área, uma vez que o tema é muito abrangente e apresenta muitas alternativas de tratamento.

7.1 Temas de Pesquisa

Vemos da seguinte forma o prosseguimento da pesquisa relacionada ao tratamento de estruturas de dados:

1. *Operações Vetoriais na Memória de Estruturas*

    - Dando continuidade à proposta do Capítulo 5, pode-se estudar a implementação do gerente de uma Memória de Estruturas. Se o gerente for um processo no anel, a simulação pode avaliar o seu paralelismo e custo de processamento.

- Devem ser estudadas formas adequadas para expor as operações vetoriais em linguagem de alto nível. O modo mais imediato será através de uma biblioteca de funções. Em seguida poderá ser observada a variação do volume de processamento no anel para vários problemas.

2. *Criação Estática de Estruturas*
   A criação estática de estruturas pode ser realizada como uma transferência direta de um dispositivo externo para a Memória de Estruturas de modo análogo à carga do programa executável para a Unidade de Programa, ao invés de ser realizada por meio de instruções de linguagem.

3. *Atualização de Estruturas*
   A pesquisa de processos de atualização de estruturas é dirigida ao aproveitamento de operações seqüenciais, tratando-se basicamente de um estudo de geração de código. Durante a compilação de um programa são identificados trechos de natureza intrinsecamente seqüencial onde incidem operações sucessivas sobre uma estrutura. Ao invés de se criar a nova estrutura sobre outra área de memória, pode-se aproveitar a área da estrutura antiga, alterando-se apenas as posições necessárias. Os ganhos associados a este procedimento são:

   - Evita-se a alocação de uma área de memória para a estrutura-resultado e a recuperação da área de memória da estrutura-argumento.
   - Evitam-se as cópias de elementos comuns entre as duas versões. Com isto reduz-se o tráfego de dados.
   - O tempo para a formação da nova estrutura é menor.
   - Reduz-se o número de instruções no código final.

   O estudo pode ser estendido a trechos de programa com baixo paralelismo. A decisão de atualizar uma estrutura passaria a ser função do nível de paralelismo do programa, do tamanho da estrutura e do tempo médio de alocação.

4. *Armazenamento de Estruturas Grandes*
   A organização da Memória de Estruturas em páginas favorece o uso de memórias secundárias. Neste caso, é preciso determinar se os programas paralelos apresentam, como os seqüenciais, uma localidade no

uso das páginas, suficiente para permitir uma memória secundária em cada Memória de Estruturas. Aparentemente, no caso de trechos com operações repetitivas e com acessos de comportamento regular, existem indicações favoráveis.

5. *Estruturas Estritas, Não-Estritas e Preguiçosas*
   Na realidade estes conceitos são ortogonais, de modo que conceitualmente podem coexistir na mesma máquina. É possível trabalhar em fluxo de dados apenas com estruturas estritas. Mas quais os ganhos reais de se dar suporte também para estruturas não-estritas? No presente trabalho defendemos com ênfase o uso de estruturas estritas por terem um custo de implementação menor e permitirem operações locais na Memória de Estruturas. Isto contudo não exclui a possibilidade de se implementar, numa máquina de maior porte unidades para esses tipos distintos de estruturas. Neste caso, a pesquisa pode ser dirigida para o dimensionamento da capacidade de armazenamento de um tipo de Memória de Estruturas em relação às demais.

7.2 Limitações

Foi mencionado que as *compreensões* podem ser uma ferramenta interessante para a descrição de estruturas e que a linguagem poderia prover operações lógicas e aritméticas sobre estruturas do tipo *vetor*. Entretanto, não explicamos a forma adequada de fazer isto. É preciso para isso elaborar uma forma conveniente de:

- caracterizar estruturas que podem ser alocadas estaticamente;
- realizar a geração de código de compreensões para estruturas estritas;
- modificar a linguagem de alto nível para que as operações diponíveis no baixo nível possam ser aproveitadas no alto nível de uma forma limpa.

Com respeito a nossos exercícios de simulação, é clara a limitação de se aterem a problemas de pequeno porte. Não testamos as operações vetoriais como se de fato estivessem localizadas na Memória de Estruturas do modo proposto, porque seria difícil quantificar o tempo associado a estas operações no estágio atual desses estudos. Deste modo preferimos comparar duas Memórias de Estruturas que tivessem formas diferentes de armazenamento e sincronização, mas com aproximadamente os mesmos custos de

implementação, podendo apenas inferir que, se operações vetoriais forem executadas próximo ao local de armazenamento elas dar-se-ão com grande rendimento. Desejávamos que fossem executados *benchmarks* mais complexos na presença de um número maior de Memórias de Estruturas mas isto se tornou impraticável no limitado ambiente da nossa simulação. Para que possamos progredir é necessário antes trabalhar na geração automática de código, neste caso indispensável, para a execução de *benchmarks* realistas.

7.3 Conclusões

Este trabalho procurou apresentar os principais conceitos associados ao armazenamento de estruturas de dados em computadores a fluxo de dados dinâmicos. Foram apresentados os tipos de estruturas dados e suas representações. Como exemplos de implementações de unidades de armazenamento, foram descritas as Memórias de Estruturas do computador de Manchester e do MIT. Como contribuição principal, foi proposto um novo tipo de organização para a Memória de Estruturas, dirigida para o armazenamento de estruturas estritas e com operações locais.

As críticas geralmente formuladas ao modelo Fluxo de Dados são que a granularidade das suas tarefas é muito fina (pequena) e o modelo é inadequado para tratar problemas que têm estruturas de dados. No primeiro caso, o problema está realmente no controle do assincronismo, controle esse que pode ser administrado por meio da divisão da execução do programa em processos adequadamente geridos [39]. No segundo caso, o problema está no armazenamento de estruturas de dados como objetos. O armazenamento de estruturas não é natural ao modelo porque os dados armazenados perdem dinamismo, uma característica intrínseca do modelo. Tornam-se necessárias sincronizações para acesso a elementos armazenados e o tempo de acesso aos dados costuma ser maior porque eles residem em outra unidade física do computador que pode ser externa ao anel. Entretanto, por questões de eficiência, o armazenamento de estruturas é muito importante .

O ponto em comum nos dois problemas é a localidade. A solução está no uso correto desta localidade. A coordenação dos processos evita a perda de tempo e de recursos em tarefas precoces. A realização de operações sobre estruturas na Memória de Estruturas aproveita a proximidade entre os dados decorrente do armazenamento, evita a comunicação desnecessária de dados

e reduz de forma geral o tempo destas operações.

Elaboramos nesta dissertação uma organização e os requisitos para o aproveitamento da localidade no tratamento de estruturas em computadores a fluxo de dados dinâmicos. Ainda há muito a ser feito nesta linha, principalmente sob o ponto de vista da linguagem de programação que deve facilitar a identificação dos trechos de programa onde a localidade pode ser explorada. Quanto à organização em si, esta pode ser refinada e alterada, baseada em futuros testes de simulação.

Ainda que reconheçamos que muito há para ser feito na confirmação de nossa linha de pesquisa, os resultados que colhemos apontam para a conclusão de que o uso da localidade em Memórias de Estruturas em computadores a fluxo de dados dinâmicos pode ser muito promissor.

Bibliografia

[1] Ackerman, W.B. *"A Structure Memory for Dataflow Computers."* Masters Thesis. TR-186. Laboratory for Computer Science, MIT, ago 77.

[2] Ackerman, W.B. *"Data Flow Languages."* IEEE Computer, 15(2): 15-25, fev 82.

[3] Almasi, G.S. & Gottlieb, A. *Highly Parallel Computing* cap. 9, pp. 303-310, The Benjamin/Cummings Publishing Company, Inc., 1989.

[4] Arvind & Ekanadham, K.K. *"Future Scientific Programming on Parallel Machines."* Journal of Parallel and Distributed Computing (5): 460-493, 1988.

[5] Arvind & Ianucci, R.A.. *"A Critique of Multiprocessing von Neumann Style,"* Proceedings, 10th International Symposium on Computer Architecture, pp. 426-436, Stockholm, 1983.

[6] Arvind & Ianucci, R.A.. *"Two Fundamental Issues in Multiprocessing,"* Internal Report Memo 269, Laboratory for Computer Science, MIT, jul 86.

[7] Arvind; Kathail, V. & Pingali, K. *A Dataflow Architecture with Tagged Tokens.* Technical Report TM-174, Laboratory for Computer Science, MIT, set 80.

[8] Arvind; Heller, S. & Nikhil, R.S. *"Programming Generality and Parallel Computers."* Internal Report Memo 287, Laboratory for Computer Science, MIT, mai 88.

[9] Arvind & Nikhil, R.S. *"Executing a Program on the MIT Tagged-Token Dataflow Architecture."* Internal Report Memo 271, Laboratory for Computer Science, MIT, jun 88.

[10] Arvind; Nikhil, R.S. & Pingali, K.K. *"I-Structures: Data Structures for Parallel Computing."* Internal Report Memo 269, Laboratory for Computer Science, MIT, fev 87.

[11] Arvind & Thomas, R.E. *"I-Structures: An Efficient Data Type for Functional Languages."* Technical Report TM-178, Laboratory for Computer Science, MIT, jun 80.

[12] Backus, J. *"Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs."* CACM, **21**(8): 613-641, ago 78.

[13] Böhm, A.P.W. & Teo, Y.M. *"The Heterogenous Multiring Dataflow Simulator."* Technical Report UCMS 89-3-1, 1989.

[14] Böhm, A.P.W. & Sargeant, J. *"Code Optimization for Tagged-Token Dataflow Machines."* IEEE Transactions on Computers, **38**(1): 4-14, jan 89.

[15] Calsavara, C.M.F.R. *"Estudo de Geração de Código para uma Máquina de Fluxo de Dados."* Dissertação de Mestrado, Departamento de Ciência da Computação, IMECC, Unicamp, nov 89.

[16] Catto, A.J. *"Non-Deterministic Programming in a Dataflow Environment."* Ph.D. Thesis, Department of Computer Science, University of Manchester, jun 81.

[17] Computation Structures Group *Annual Report 1987-1988*. Laboratory for Computer Science, MIT jun 88.

[18] Culler, D.E. & Arvind. *"Resource Requirements in Dataflow Programs."* Proceedings, 15th Annual Symposium on Computer Architecture. Computer Architecture News 1988.

[19] Dennis, J.B. *"First Version of a Data Flow Procedure Language."* in Programming Symposium, LNCS, **19**: 362-376, Spring-Verlag, abr 74.

[20] Flynn, M.J. *"Some Computer Organizations and their Effectiveness."* IEEE Transactions on Computers, **C-21**: 948-960, 72.

[21] Friedman, D.P. & Wise, D. *"CONS should not evaluate its arguments."* in Automata, Languages and Programming: Third International Colloqium. Michaelson, S. & Milner, R. (Editors). Edingburg University Press 1976, pp. 257-284.

[22] Friedman, D.P. & Wise, D. *"Aspects of Applicative Programming for Parallel Computers."* IEEE Transactions on Computers, **C-27**(4): 289-296, abr 78.

[23] 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): 58-70, fev 82.

[24] Gajski, D.D. & Pier, K.K. *"Essential Issues on Multiprocessor Systems."* IEEE Computer, jun 1985.

[25] Gaudiot, J.L. & Ercegovac, M.D. *"A Scheme for Handling Arrays in Dataflow Systems."* Proceedings, 9th Annual International Symposium on Computer Architecture, **14**(2): 724-729, 1982.

[26] Gurd, J.R.; Barahona, P.M.C.C.; Böhm, A.P.W.; Kirkham, C.C.; Parker, A.J.; Sargeant, J. & Watson, I. *"Fine-Grain Parallel Computing: The Dataflow Approach."* in Future Parallel Computers, LNCS, **272**: 82-152, Spring-Verlag, jun 86.

[27] Heller, S.K. *Efficient Lazy Data Structures on a Dataflow Machine.* Ph.D. Dissertation. TR-438. Laboratory for Computer Science, MIT, fev89.

[28] Hurson, A.R.; Lee, B. & Shirazi, B. *"Hibrid Structure: A Scheme for Handling Data Structures in a Dataflow Environment."* in PARLE '89 vol.I, LNCS, **365**: 323-340.

[29] Kawakami, K. & Gurd, J.R. *"A Scalable Data Flow Structure Store."* Proceedings, 13th Annual International Symposium on Computer Architecture, **14**(2): 243-250, jun 86.

[30] Knuth, D.E. *The Art of Computer Programming vol. I (second edition).* pp. 442-445. Addison-Wesley. 1973.

[31] Kumar, M. *"Effect of Storage Allocation/Reclamation Methods on Paralellism and Storage Requirements"* Proceedings, 14th Annual Symposium on Computer Architecture. Computer Architecture News pp. 197-205, 1987.

[32] Lee, C.C.; Skedzielewski, S. & Feo, J. *"On the Implementation of Applicative Languages on Shared-Memory, MIMD Multiprocessors."* Sigplan Notices, **23**(9): 188-197, set 88.

[33] McGraw, J. et. al. *SISAL - Streams and Iteration in a Single Assignment Language.* Language Reference Manual, ver. 1.2, M-146, La-

wrence Livermore National Lab., ago 84.

[34] Nikhil, R.S. *ID.* ver. 88.1, Reference Manual Memo 284, Laboratory for Computer Science, MIT, ago 88.

[35] Oldehoeft, R.R. & Cann, D.C. *"Applicative Parallelism on a Shared-Memory Multiprocessor."* IEEE Software, **5**(1): 62-70, jan 88.

[36] Pingali, K. & Ekanadham, K. *"Accumulators: A New Array Abstraction for Functional Languages."* in Foundations of Computer Science and Theoretical Computer Science, LNCS, **338**: 375-339, dez 88.

[37] Padua, D.A. & Michael J. Wolfe, *"Advanced Compiler Optimizations for Supercomputers,"* CACM **29**(12): 1184-1201, dez 86.

[38] Patnaik, L.M.; Govindarajan, R. & Ramadoss, N.S. *"EXMAN: An Extended Manchester Dataflow Architecture."* IEEE Transactions on Computers **35**(3): 229-243, mar 1986.

[39] Ruggiero, C.A. *Throttle Mechanisms for Manchester Dataflow Machine.* Ph.D. Dissertation , Department of Computer Science, University of Manchester, jul 87.

[40] Sargeant, J. *Efficient Stored Data Structures for Dataflow Computing.* Ph.D. Dissertation , Department of Computer Science, University of Manchester, abr 85.

[41] Sargeant, J. & Kirkham, C.C. *"Stored Data Structures on the Manchester Data Flow Machine."* Proceedings, 13th Annual International Symposium on Computer Architecture, **14**(2): 235-242, jun 86.

[42] Shimada, T.; Hiraki, K.; Nishida, K. & Sekiguchi, S. *"Evaluation of a Prototype Dataflow Processor of the Sigma-1 for Scientific Computation."* Proceedings, 13th Annual International Symposium on Computer Architecture, **14**(2): 226-234, jun 86.

[43] Treleaven, P.C.; Brownbridge, D.R. & Hopkins, R.P. *"Data-Driven and Demand-Driven Computer Architecture."* Computing Surveys, **14**(1): 93-143, mar 82.

[44] Veen, A.H. *"Dataflow Machine Architecture,"* ACM Computing Surveys **18**(4): 365-369, dez 86.

[45] Vegdahl, S.R. *"A Survey of Proposed Architectures for The Execution of Functional Languages."* S.S. Thakkar, Dataflow and Reduction Architectures, pp. 55-76, 1987.

[46] Wei, Y.H. & Gaudiot J.L. *"Compiling Programs to Direct Acess Data-flow Graphs."* IBM Research Report RC 15419 (# 68611) 1 / 25 / 90.

[47] Weng, K.S. *"Stream Oriented Computation in Recursive Dataflow Schemas."* TM-68, Laboratory for Computer Science, MIT, out 75.

Índice

Apêndice A

Conjunto de Instruções para a Memória de Estruturas do Computador de Manchester

A.1 Formato do Apontador para uma Estrutura

O tipo *pont* no computador de Manchester é uma ficha com os seguintes campos:

endereço	tamanho da estrutura

A.2 Instruções Implementadas

tipos: pont - apontador
int - inteiro
qq - qualquer tipo

Código	Argumentos	Resultados	Comentários
srq	int	pont	"Space ReQuest" solicita uma área de 'n' posições.
msp	pont, int	pont	"Modify Size in Pointer" modifica o campo de tamanho do apontador.
rss	pont, int	qq	"Read Structure Store" lê o elemento de uma estrutura.
wss	pont, qq		"Write to Structure Store" escreve um valor numa estrutura.
irc	pont		"Increment Reference Count" incrementa o contador de referências de uma estrutura.
drc	pont		"Decrement Reference Count" decrementa o contador de referências de uma estrutura.
fss	pont	qq	"Fetch from Structure Store" lê todos os elementos de uma estrutura colocando-os sob a forma de um *stream*. Na realidade são enviadas a Memória de Estruturas mensagens individuais para a leitura dos seus elementos.
ess	pont, int		"Extract Structure Store" lê e remove um elemento de uma estrutura. O 'bit' de presença deste endereço é atribuído o valor ausente.

A.3 Associação entre Instruções e Mensagens

contextoDestino ::= <contexto> <destino>
contexto ::= <nome de ativação> <índice>

Código	Requisição	Resposta
srq	<int, contextoDestino>	<pont>$_{contextoDestino}$
rss/ess	<SSaddr,contextoDestino>	<qq>$_{contextoDestino}$
fss	<SSaddr1,contextoDestino1>... <SSaddr*n*,contextoDestino*n*>	<qq>$_{contextoDestino1}$... <qq>$_{contextoDestino*n*}$
wss	<SSaddr, qq>	sem confirmação
irc/drc	<SSaddr>	<int>$_{contextoDestino}$

Observações:

- *irc* recebe uma ficha de resposta para cada mensagem, mas *drc* apenas quando a contagem de referências chegar a zero;
- SSaddr é um endereço absoluto na Memória de Estruturas.

Apêndice B

Conjunto de Instruções para a Memória de Estruturas Proposta

B.1 Formato do Apontador para uma Página

O tipo *pont* para esta Memória de Estruturas é uma ficha que tem dois campos:

número da página	deslocamento

B.2 Instruções Implementadas

Código	Argumentos	Resultados	Comentários
prq	int	pont	"Page ReQuest" solicita n páginas a Memória de Estruturas. A resposta é um *stream* de páginas separadas pelo índice.
pgr	pont	pont	"PaGe Release" retorna a página para a Memória de Estruturas.
spg	pont, int	bool, bool	"Syncronize PaGe" determina o número de elementos numa página. A resposta inicial é uma confirmação da Memória de Estruturas indicando o recebimento da mensagem. Quando a página fica pronta é enviada uma outra confirmação.
ppp	pont, int	pont	"Prepare Page Pointer" prepara o apontador para endereçar uma posição numa página, modificando o campo de deslocamento. Esta instrução não envia mensagem para a Memória de Estruturas.
wsm	qq, pont		"Write to Structure Memory" escreve um elemento numa página. (A sincronização é estabelecida implicitamente.)
rsm	pont, int	qq	"Read Structure Memory" lê um elemento de uma página.
mse	pont, pont		"Move Structure Element" copia um elemento de uma página para outra.

B.3 Outras Instruções Interessantes (não implementadas)

Código	Argumentos	Resultados	Comentários
cpg	pont, pont	bool	"Copy PaGe" copia todos os elementos de uma página para outra. Quando a operação foi concluída é enviada uma confirmação.
fpg	pont	qq, bool	Instrução semelhante a 'fss' do computador de Manchester. "Fetch PaGe" lê todos os elementos de uma página (i.e. constrói um stream com os elementos da página). Quando todos os elementos foram lidos é emitida uma confirmação.
fsp	pont,int	qq, bool	"Fetch Sub-Page" lê um conjunto consecutivo de posições de uma página. No fim envia da leitura a Memória de Estruturas envia uma confirmação.

B.4 Associação entre Instruções e Mensagens

Ins	Requisição	Resposta
prq	<int, contextoDestino>	$<P_1>_{contextoDestino1} \ldots <P_n>_{contextoDestinon}$
pgr	<P>	sem confirmação
spg	<pont, int, contextoDestino>	$<true>_{rótulo,ds+1}$, $<true>_{rótulo,\ ds}$
wsm	<pont, qq>	sem confirmação
rsm	<pont, contextoDestino>	$<qq>_{contextoDestino}$
mse	<pont, pont, contextoDestino>	$<bool>_{contextoDestino}$

Observações:

- As instruções *prq* e *pgr* devem ser enviadas ao gerente. Na implementação para a simulação este gerente foi colocado na Memória

de Estruturas.

- Fichas em *streams* são separadas pelo campo de índice no contexto da ficha.