ANÁLISES ESTÁTICAS PARA A PARALELIZAÇÃO DE LINGUAGENS DE ATRIBUIÇÃO ÚNICA PARA SISTEMAS DE MEMÓRIA DISTRIBUÍDA

RAUL JUNJI NAKASHIMA

Tese apresentada ao Instituto de Física de São Carlos, Universidade de São Paulo, para obtenção do título de Doutor em Ciências "Física Aplicada-Opção Física Computacional"

Orientador: Prof. Dr. Gonzalo Travieso

USP/IFSC/SBI

8-2-001415

São Carlos – São Paulo
2001

IFSC-USP SERVIÇO DE BIBLIOTECA E INFORMAÇÃO

Nakashima, Raul Junji

"Análises Estáticas para a Paralelização de Linguagens de Atribuição única para Sistemas de Memória Distribuída"/ Raul Junji Nakashima – São Carlos, 2001

Tese (Doutorado) – Instituto de Física de São Carlos da Universidade de São Paulo, 2001 – páginas: 124
Área: Física Aplicada-Opção Física Computacional
Prof. Dr. Gonzalo Travieso

1. Compilador Paralelo, 2. Memória Distribuída, 3. Linguagem Funcional

I. Título

UNIVERSIDADE DE SÃO PAULO
Instituto de Física de São Carlos

Caixa Postal 369
13560-970 São Carlos, SP
Av. Trabalhador São-carlense, 400,
13566-590 São Carlos, SP

Fone/Fax 16 273 9777
www.if.sc.usp.br
wladerez@if.sc.usp.br

MEMBROS DA COMISSÃO JULGADORA DA TESE DE DOUTORADO DE **RAUL JUNJI NAKASHIMA**, APRESENTADA AO INSTITUTO DE FÍSICA DE SÃO CARLOS, UNIVERSIDADE DE SÃO PAULO, EM 24-09-2001.

COMISSÃO JULGADORA:

Prof. Dr. GONZALO TRAVIESO (Orientador e Presidente) – IFSC /USP

Prof. Dr. JAN FRANS WILLEN SLAETS – IFSC /USP

Prof. Dr. HÉLIO CRESTANA GUARDIA – UFSCar

Prof. Dr. JAIRO PANETTA – INPE

Prof. Dr. EDUARDO WHITAKER BERGAMINI – INPE

Aos meus país Shimji Nakashima e Satoy Nakashima pela dedicação e por sempre terem me incentivado nos estudos.

AGRADECIMENTOS

- Ao professor Gonzalo Travieso pela orientação imprescindível para a realização deste trabalho.
- Ao professor Jan Franz Willem Slaets pela orientação e colaboração nos mais diversos assuntos.
- Aos professores Claudio Shyinti Kiminami e Ruth Herta Goldschmidt Aliaga Kiminami pela amizade e pelas oportunidades que me ofereceram desde os meus tempos de graduação.
- Ao amigo André Muezerie pela amizade, pela ajuda nos diversos trabalhos do grupo.
- Ao amigo Márcio Alexandre Marques pelos valorosos conselhos.
- Ao amigo Cláudio Kakuda sempre solícito e atencioso.
- Ao amigo Bruno Otto Teodoro Rosa, conselheiro e ouvinte nos mais diversos assuntos.

CONTEÚDO

1. INTRODUÇÃO **1**

2. CONTEXTO **4**

2.1. ARQUITETURA DE COMPUTADORES PARALELOS 6

2.1.1 Interesses Comerciais *6*

2.1.2 Tendência em Arquitetura Paralela *7*

2.2. PROGRAMAÇÃO E LINGUAGENS 9

2.2.1 Modelo de Programação Paralela *9*

2.2.2 Linguagens Imperativas *10*

2.2.3 Linguagens declarativas *11*

2.3 TRANSFORMAÇÕES EM COMPILADORES 13

2.3.1 Compiladores *13*

2.3.2 Compiladores Paralelizantes *15*

2.4. *SISAL* 19

2.4.1. A Linguagem SISAL *19*

2.4.2. Forall *21*

2.5. A LINGUAGEM IF1 – INTERMEDIATE FORMAT 1 24

2.6. O COMPILADOR OSC 26

2.7 CONCLUSÃO 30

3. DISTRIBUIÇÃO AUTOMÁTICA **31**

3.1. TRANSFORMAÇÕES DOS DADOS 32

3.1.1. Balanceamento de carga estático *33*

3.1.2. Alinhamento *34*

3.1.3. Divisão e Mapeamento *37*

3.2. TRABALHOS RELACIONADOS 40

3.3. CONCLUSÃO 51

4. DISTRIBUIÇÃO DE DADOS E COMPUTAÇÃO **52**

4.1. ALINHAMENTO 52

4.2. DIVISÃO 56

4.3. MAPEAMENTO 60
4.4 ESTUDO DOS MÉTODOS COM *SISAL* 65
4.4.1. Alinhamento *65*
4.4.2 Divisão *69*
4.4.3. Mapeamento *70*
4.5. CONCLUSÃO 72

5. IMPLEMENTAÇÃO 73

5.1. LINGUAGEM INTERMEDIÁRIA 73
5.1.1. Nós compostos [SKE85a] *75*
5.1.2. Nó Composto FORALL [SKE85a] *76*
5.2. FERRAMENTAS FLEX E BISON 78
5.2.1 FLEX *78*
5.2.2 BISON *80*
5.3. INTEGRAÇÃO COM O OSC 82
5.4. ESTRUTURA UTILIZADA 83
5.5 ALGORITMO GERAL 84
5.6. ALINHAMENTO 85
5.6.1 EXTRAÇÃO DE FUNÇÕES AFINS 85
5.6.2 Soluções do sistema de Equações *87*
5.7. DIVISÃO 87
5.8. MAPEAMENTO 90
5.9. GERAÇÃO DE CÓDIGO IF1 COM MPI 91
5.10. CONCLUSÕES 92

6. RESULTADOS 94

6.1. CÓDIGO GERADO 94
6.2. ESTUDO DE CASOS 98
6.2.1. Caso 1 ..... *99*
6.2.2. Caso 2 ..... *102*
6.2.3. Caso 3 ..... *105*
6.2.4. Diferenças Finitas *108*
6.2.5. Multiplicação de Matrizes *113*
6.3 Conclusões *115*

7. CONCLUSÕES **116**

7.1. ALINHAMENTO 116

7.2. DIVISÃO 117

7.3. MAPEAMENTO 118

7.4. IMPLEMENTAÇÃO 119

7.5. RESULTADOS 121

7.6. TRABALHOS FUTUROS 121

7.7. CONCLUSÕES GERAIS 122

APÊNDICE A **125**

APÊNDICE B **126**

APÊNDICE C **130**

APÊNDICE D **132**

APÊNDICE E **138**

APÊNDICE F **140**

LISTA DE FIGURAS

FIGURA 1 ORGANIZAÇÃO GENÉRICA DE UM MULTIPROCESSADOR ESCALÁVEL. FONTE [CUL99] 8

FIGURA 2 ESTRUTURA PARA UM COMPILADOR COM A OTIMIZAÇÃO REALIZADA NO CÓDIGO INTERMEDIÁRIO DE BAIXO NÍVEL [MUC97] 14

FIGURA 3 ESTRUTURA PARA UM COMPILADOR COM A OTIMIZAÇÃO DIVIDIDA EM DUAS FASES, UMA CÓDIGO INTERMEDIÁRIO DE NÍVEL MÉDIO E OUTRA NO CÓDIGO INTERMEDIÁRIO DE BAIXO NÍVEL [MUC97] 14

FIGURA 4 ORDEM DAS OTIMIZAÇÕES [MUC97] 17

FIGURA 5 SINTAXE DA CONSTRUÇÃO FOR NA FORMA PRODUTO [MGR85] 22

FIGURA 6 REGRA DE PRODUÇÃO DA GERAÇÃO DE ÍNDICES DA CONSTRUÇÃO FOR NA FORMA PRODUTO [MGR85] 22

FIGURA 7 REGRA DE PRODUÇÃO DA PARTE DE RETORNO DA CONSTRUÇÃO FOR NA FORMA PRODUTO [MGR85] 23

FIGURA 8 GRAFO DE (A+B)/2 25

FIGURA 9 REPRESENTAÇÃO DO GRAFO DA FIGURA 8 EM IF1 26

FIGURA 10 NÓ COMPOSTO *FORALL* 26

FIGURA 11 PROCESSAMENTO DA LINGUAGEM *SISAL* 28

FIGURA 12. IDÉIA BÁSICA DO ALINHAMENTO DE DADOS 35

FIGURA 13. REPRESENTAÇÃO DO ACESSO AOS *ARRAYS* A E B NO *LOOP* 36

FIGURA 14. ALINHAMENTO DAS MATRIZES DA FIGURA 13. 37

FIGURA 15. (A) DISTRIBUIÇÃO POR LINHAS (B) DISTRIBUIÇÃO POR COLUNAS (C) DISTRIBUIÇÃO POR BLOCOS (D) DISTRIBUIÇÃO COLUNA-CICLICA (E) DISTRIBUIÇÃO BLOCO-CÍCLICA 38

FIGURA 16. DIVISÃO POR LINHAS DE UMA MATRIZ EM P PROCESSADORES 39

FIGURA 17. MAPEAMENTO POR BLOCOS DE UMA MATRIZ NUMA GRADE DE $\sqrt{p} \times \sqrt{p}$ PROCESSADORES 40

FIGURA 18. EXEMPLO SIMPLES DE UM CÓDIGO *SISAL* ONDE É POSSÍVEL ALINHAR OS DADOS 65

FIGURA 19. EXEMPLO SIMPLES DE UM CÓDIGO *SISAL* ONDE NÃO É POSSÍVEL ALINHAR OS DADOS 66

FIGURA 20. EXEMPLO DE UM CÓDIGO *SISAL* ONDE É POSSÍVEL ALINHAR OS DADOS 67

FIGURA 21. EXEMPLO DE UM CÓDIGO *SISAL* ONDE É POSSÍVEL ALINHAR OS DADOS 68

FIGURA 22. ILUSTRAÇÃO DA DISTRIBUIÇÃO DA MATRIZ B DO EXEMPLO DA FIGURA 21 69

FIGURA 23. REPRESENTAÇÃO GERAL DE UM NÓ COMPOSTO IF1 75

FIGURA 24. REPRESENTAÇÃO GERAL DO NÓ COMPOSTO *FORALL* DE IF1 79

FIGURA 25. TRECHO DO ARQUIVO DE ENTRADA DO FLEX 80

FIGURA 26. TRECHO DO ARQUIVO DE ENTRADA BISON 82

FIGURA 27. ESTRUTURA PARA ARMAZENAR O GRAFO IF1 83

FIGURA 28. ESTRUTURA PARA ARMAZENAR O NÓ IF1 (SIMPLES E COMPOSTO) 83

FIGURA 29. ESTRUTURA PARA ARMAZENAR O ARCO IF1 84

FIGURA 30. ESTRUTURA PARA ARMAZENAR TIPOS DE IF1 (SIMPLES E ESTRUTURADOS) 84

FIGURA 31. ESTRUTURA UTILIZADA PARA ARMAZENAR ANINHAMENTOS DE *LOOPS* 86

FIGURA 32. ESTRUTURA UTILIZADA PARA ARMAZENAR AS FUNÇÕES AFINS 86

FIGURA 33. REPRESENTAÇÃO DE COMO FICA ARMAZENADO UM ANINHAMENTO DE *LOOPS* 88

FIGURA 34. REPRESENTAÇÃO DE COMO FICA ARMAZENADO O ACESSO A *ARRAY* (EM IF1) 88

FIGURA 35. EXEMPLO DE UMA SOLUÇÃO DE MAPEAMENTO DE DADOS E COMPUTAÇÕES. OS QUADRADOS PINTADOS REPRESENTAM AS POSSÍVEIS SOLUÇÕES PARA O SISTEMA 90

FIGURA 36. ILUSTRAÇÃO DA SOLUÇÃO DE DIVISÃO DE DADOS E COMPUTAÇÕES DA FIGURA 35 91
FIGURA 37. EXEMPLO DE UM CÓDIGO SISAL E AS FUNÇÕES DE ACESSO AOS ARRAYS 95
FIGURA 38. ALINHAMENTO PARA OS *ARRAYS* DO PROGRAMA DA FIGURA 37 95
FIGURA 39. ALINHAMENTO DOS ACESSOS PARA O PROGRAMA DA FIGURA 37 95
FIGURA 40. RESULTADO OBTIDO DA ANÁLISE PARA DIVISÃO DO PROGRAMA DA FIGURA 39 96
FIGURA 41. RESULTADO OBTIDO DA ANÁLISE PARA DIVISÃO DO PROGRAMA DA FIGURA 39 98
FIGURA 42. EXEMPLO DE UM CÓDIGO SISAL E AS FUNÇÕES DE ACESSO AOS ARRAYS 99
FIGURA 43. ALINHAMENTO PARA OS *ARRAYS* DO PROGRAMA DA FIGURA 42 99
FIGURA 44.. ALINHAMENTO DOS ACESSOS PARA O PROGRAMA DA FIGURA 43 99
FIGURA 45. RESULTADO OBTIDO DA ANÁLISE PARA DIVISÃO DO PROGRAMA DA FIGURA 39 100
FIGURA 46. RESULTADO OBTIDO DA ANÁLISE PARA DIVISÃO DO PROGRAMA DA FIGURA 42 101
FIGURA 47. EXEMPLO DE UM CÓDIGO SISAL E AS FUNÇÕES DE ACESSO AOS *ARRAYS* 102
FIGURA 48. ALINHAMENTO PARA OS *ARRAYS* DO PROGRAMA DA FIGURA 47 102
FIGURA 49. ALINHAMENTO DOS ACESSOS PARA O PROGRAMA DA FIGURA 47 103
FIGURA 50. RESULTADO OBTIDO DA ANÁLISE PARA DIVISÃO DO PROGRAMA DA FIGURA 47 103
FIGURA 51. RESULTADO OBTIDO DA ANÁLISE PARA DIVISÃO DO PROGRAMA DA FIGURA 47 105
FIGURA 52. EXEMPLO DE UM CÓDIGO SISAL E AS FUNÇÕES DE ACESSO AOS ARRAYS 106
FIGURA 53. RESULTADO OBTIDO DA ANÁLISE PARA DIVISÃO DO PROGRAMA DA FIGURA 51 106
FIGURA 54. RESULTADO OBTIDO DA ANÁLISE PARA DIVISÃO DO PROGRAMA DA FIGURA 52 108
FIGURA 55. IMPLEMENTAÇÃO DO ALGORITMO DE *JACOBI-RICHARDSON* PARA SOLUÇÃO PARA DIFERENÇAS FINITAS .. 109
FIGURA 56. ALINHAMENTO PARA OS *ARRAYS* DO PROGRAMA DA FIGURA 54 109
FIGURA 57. ALINHAMENTO DOS ACESSOS PARA O PROGRAMA DA FIGURA 54 110
FIGURA 58. RESULTADO OBTIDO DA ANÁLISE PARA DIVISÃO DO PROGRAMA DA FIGURA 54 110
FIGURA 59. RESULTADO OBTIDO DA ANÁLISE PARA DIVISÃO DO PROGRAMA DA FIGURA 39 113
FIGURA 60. PROGRAMA PARA O ALGORITMO TRADICIONAL DE MULTIPLICAÇÃO DE MATRIZES 114
FIGURA 61. SISTEMA PARA ENCONTRAR O ALINHAMENTO ENTRE A E C E B E C 114
FIGURA 62. SISTEMA PARA ENCONTRAR A DIVISÃO ENTRE A E C E B E C 114

LISTA DE TABELAS

TABELA 1. CLASSIFICAÇÃO DE LINGUAGENS FUNCIONAIS [HUT00] 12
TABELA 2. EXEMPLOS DO COMANDO *FORALL* DE *SISAL* 21
TABELA 1. DESCRIÇÃO DAS CLASSES DE PORTAS DO NÓ COMPOSTO FORALL 77
TABELA 2. ASSOCIAÇÃO DAS PORTAS NÓ COMPOSTO FORALL 78

RESUMO

Este trabalho descreve técnicas de análise estática de compilação baseadas na álgebra e programação linear que buscam otimizar a distribuição de *loops forall* e *array* em programas escritos na linguagem *SISAL* visando a execução em máquinas paralelas de memória distribuídas. Na fase de alinhamento, nós trabalhamos com o alinhamento de hiperplanos onde objetivo é tentar encontrar as porções dos diferentes *arrays* que necessitam ser distribuídas juntas. Na fase de divisão, que tenta quebrar em partes independente dados e computações, nós usamos duas funções afins, a função de decomposição de dados e a função de decomposição de computação. A última fase, o mapeamento, distribui os elementos de computação nos elementos de processamento usando um conjunto de inequações. As técnicas foram implementadas num compilador SISAL, mas pode ser usada sem mudanças em outras linguagens de associação simples e com a adição de análise de dependências pode ser usada em linguagens imperativas.

ABSTRACT

This work describes static compiler analysis techniques based on linear algebra and linear programming for optimizing the distribution of *forall* loops and of array elements in programs written in the SISAL programming language for distributed memory parallel machines.

In the alignment phase, attempt is made in the identification of portions of different arrays that need to be distributed jointly by means of alignment of hyperplanes. In the partitioning phase, effort is made in breaking as even possible the computation and pertinent data in independent parts, by means of using related functions: the data decomposition function and the computation decomposition function. The last phase is dedicated to the mapping, which comprises the distribution of the elements of computation into the existing processing elements by means of a set of inequations. These techniques are being implemented in a SISAL compiler, but can be also used without changes by means of other single assignment languages or, with the addition of dependecy analysis when using other set of languages, as well.

1. INTRODUÇÃO

A tecnologia de otimização [BAC93, MUC97, WOL96] desempenha uma função importante no desempenho de um computador moderno. Como as arquiteturas de computadores estão se tornando cada vez mais complexas, isto torna muito difícil, senão impossível, que os programadores projetem sistemas eficientes, aplicando técnicas de otimizações dependentes ou independentes da plataforma destino. Assim, os compiladores otimizadores estão se tornando um componente essencial dos modernos sistemas de computação de alta performance que, além de traduzir o programa fonte em linguagem de máquina, analisam e aplicam várias transformações para reduzir, no programa destino, o tempo de execução ou tamanho ou ambos.

Com os compiladores otimizadores tornando-se cada vez mais efetivos, os programadores precisam preocupar-se menos com os detalhes das arquiteturas das máquinas utilizadas e podem empregar construções de programação e organizações de programas de mais alto nível mais sucintos e de forma mais intuitiva. Simultaneamente, os projetistas de hardware estão aptos a empregar projetos que resultem em grande melhora no desempenho porque eles necessitam apenas preocupar-se com a adequação do projeto com um compilador destino e não com uma interface direta com o programador [BAC93, MUC96].

Entre as otimizações mais importantes estão aquelas que operam sobre *loops* [BAC93, KUL93, WOL92, WOL96] (tal como mover computações invariantes para fora de um *loop*). Entretanto, existem muitos tipos de otimização que podem ser relevantes para um programa particular e estas variam segundo a estrutura e detalhes do programa.

Nas últimas três décadas um grande número de transformações de compiladores para a otimização de programas foi desenvolvido [BAC93]. A maioria das otimizações para monoprocessadores reduz o número de instruções executadas pelo programa e analisa as propriedades de variáveis usando técnicas de análise de fluxo. Em contraste, as otimizações para processadores de alta performance, vetoriais e paralelos, maximizam o paralelismo e a localidade de memória, principalmente rastreando as propriedades de *arrays* através de análises de dependência de *loops*.

Em linguagens imperativas[1], normalmente são realizadas transformações que visam expor o paralelismo de um programa, muitas delas aplicadas em comandos de repetição tipo *for*. Em linguagens funcionais[2] [HUT00, REA93] como a SISAL [MGR85, FEO90, CAN92a] o paralelismo já é inerente à linguagem, o que dispensa este tipo de tratamento e assim os esforços do compilador ficam concentrados na exploração do paralelismo.

Nos sistemas de memória distribuída, a localidade de dados [WOL96] nos processadores é de importância crítica para a eficiência do programa paralelo; visto que a comunicação entre os processadores é mais custosa que a computação, é essencial que o processador tenha a possibilidade de executar o máximo possível usando dados locais. Uma comunicação excessiva entre processadores pode facilmente inviabilizar qualquer ganho que poderia ser obtido pelo uso de paralelismo.

Outra consideração importante para uma distribuição de dados ideal é que ela deve permitir que a carga de trabalho entre os processadores seja o mais uniforme possível de modo que se possa explorar o paralelismo inerente na computação em sua totalidade [GUP92, PIE93]. Um mapeamento eficiente, de dados e de computações, depende de muitos fatores complexos tais como a estrutura do programa, as capacidades dos processadores, as características da máquina destino e o volume a estrutura dos dados do programa [AYG98, GAR96].

Assim, neste trabalho propomos a paralelização de programas SISAL em sistemas de memória distribuída, através de análises que, num primeiro momento, realizem o alinhamento de *arrays*, para melhorar a localidade no acesso aos dados e, em seguida, realizem a distribuição de dados e computações num sistema virtual de processadores e, então, mapeiem este sistema nos processadores físicos disponíveis. Essas transformações estão descritas mais detalhadamente no Capítulo 3, nas seções 3.1.2 e 3.1.3.

Os modelos, de álgebra linear, apresentados em Anderson [AND97] e O'Boyle [OBO92] formam a base para o desenvolvimento deste trabalho de doutorado. Como os modelos apresentados por Anderson e Lam são voltados à análise de programas imperativos, algumas modificações foram tiveram que ser introduzidas para que eles pudessem ser utilizados na análise de programas escritos na linguagem funcional *SISAL*.

---

[1] A essência da programação em linguagens imperativas é a computação, passo a passo e repetida, de valores de baixo nível e atribuição desses valores a posição de memória [GUE85]

[2] Uma linguagem funcional faz uso das propriedades matemáticas das funções. A essência da programação functional é combinar funções para produzir outras mais poderosas [GUE85].

As modificações realizadas no modelo de O'Boyle foram necessárias para que suas técnicas pudessem ser utilizadas com o modelo apresentado por Anderson e Lam.

2. CONTEXTO

A grande disponibilidade atual de computadores paralelos torna clara a falta de software que existe neste segmento da computação e deixa ainda mais evidente que a conversão dos programas escritos nas linguagens mais utilizadas atualmente não é uma tarefa trivial, visto que além da adaptação eles devem ser otimizados para executar eficientemente no sistema com múltiplos processadores [PIE93], ou seja, o programa paralelo deve ser significativamente mais rápido que o programa seqüencial. Além disso, a programação paralela é muito mais complexa que a programação seqüencial, seja na fase de projeto, depuração ou verificação de correção e eficiência.

Uma das formas de reduzir esta complexidade é o desenvolvimento de compiladores paralelizantes, isto é, compiladores que realizem a paralelização automática de programas escritos em alguma linguagem-fonte, gerando um código-destino adequado à execução em computadores paralelos. Atualmente os compiladores paralelizantes mais populares são aqueles que analisam as linguagens C, C++ e FORTRAN com extensões para a programação paralela [COH94, CUL99]. Esses compiladores paralelizantes analisam os programas fontes na tentativa de expor o máximo de paralelismo do código, para então em seguida fazer o melhor uso deste paralelismo.

Além dos compiladores para as linguagens tradicionais também existem os compiladores para as linguagens declarativas. Como o paralelismo existente nestas linguagens é implícito [SKE91, WOL96], ele deve, portanto ser extraído pelo compilador, ou seja, por meio de diversas otimizações, tendo como objetivo obviamente a geração de um código paralelo eficiente, tanto em desempenho como no acesso à memória. Segundo Anderson [AND97], grande parte do sucesso nas otimizações dos compiladores é dependente da forma como os dados serão distribuídos no sistema de memória do computador onde a aplicação será executada.

Podemos observar que os compiladores paralelizantes tanto para as linguagens imperativas como para as declarativas realizam uma série de transformações sobre um programa de forma a torná-lo adequado à execução paralela. Ainda com relação às otimizações utilizadas nos compiladores paralelizantes podemos observar que existe um certo consenso de quais às funcionalidades necessárias para uma ferramenta deste tipo, mas não de uma técnica específica a se usar [COH94].

De uma forma geral, quando tratamos da otimização de programas na construção de compiladores, temos duas divisões básicas, mais precisamente uma categoria de otimizações independentes de hardware e a outra dependente de características específicas de hardware [MUC97]. Os compiladores modernos são construídos de forma que a otimização seja realizada em várias etapas [MUC97] em parte em função da existência de diversos tipos de otimizações possíveis e em parte devido ao fato que muitas vezes a aplicação de uma otimização gera alterações no código que exigem que uma otimização que já tenha sido aplicada anteriormente seja reaplicada. Assim, é mais adequado que cada tipo de otimização seja implementada como um elemento separado e reutilizada cada vez que for necessária. Nos compiladores paralelizantes a mesma estratégia é adotada.

Aliado ao conhecimento adquirido nestas décadas de pesquisas em compiladores paralelizantes, temos que atualmente, com a convergência de arquiteturas paralelas no sentido de uma máquina paralela genérica [CUL99], as técnicas de otimização desenvolvidas podem ser no geral melhor reaproveitadas, o que vai de encontro ao citado por Ian Foster [FOS95] de que o avanço da computação paralela depende muito da existência de uma arquitetura genérica, como existe na computação seqüencial com a máquina de von Neumann. Podemos observar atualmente que existe convergência na organização dos sistemas paralelos [CUL99, STE99], mas não uma definição de como deveria ser o controle de execução dos processos nesses sistemas. Assim esse seria o próximo passo para a definição de uma arquitetura paralela genérica, o que representaria com certeza um avanço para a construção não só de compiladores paralelos, mas também a construção de ferramentas em geral e aplicações para esse tipo de sistema.

Nas seções seguintes apresentaremos uma revisão de alguns dos elementos que compõe um sistema paralelo, a saber, arquitetura, modelos de programação, linguagens e compiladores. Também apresentaremos a linguagem *SISAL*, que é a linguagem aceita pelo compilador OSC, que recebeu as otimizações do trabalho desenvolvido. Veremos na seção 2.1 a tendência atual de hardware paralelo, abordando os interesses comerciais e organizações atuais, em seguida apresentaremos na seção 2.2 algumas das características de linguagens e modelos de programação atualmente utilizados nos sistemas paralelos e na seção 2.3 características de compiladores seqüenciais e paralelos, enfocando a otimização de código. A seção 2.4 apresenta uma breve descrição a linguagem *SISAL*, com foco na construção paralela `forall` e em seguida descreve a organização do compilador OSC, mais precisamente o *front-end* (que converte um programa *SISAL* linguagem Intermediária IF1) e *back-end* (demais otimizações do compilador).

2.1. ARQUITETURA DE COMPUTADORES PARALELOS

2.1.1 Interesses Comerciais

Em contraste há décadas anteriores, na década de 90 grandes empresas fabricantes de computadores entrarem no segmento de computação científica, tendência esta que pode ser observada na lista [DON99,MEU00] dos 500 computadores mais rápidos do mundo ou pelo levantamento apresentado em [CUL99] que apresenta um gráfico com os tipos de sistemas paralelos utilizados entre 1993 e 1996. A grande maioria dos computadores apresentados em [DON99, MEU00, STE99] pertencem a novos integrantes deste seleto clube de fabricantes de computadores paralelos, por exemplo, a Compaq, Intel, Silicon Graphics, Sun e a IBM.

A conseqüência da entrada dessas empresas neste setor é que os projetos de supercomputadores que estão sendo desenvolvidos são predominantemente baseados em componentes utilizados nos computadores pessoais e estações de trabalho. Assim, a construção de computadores com milhares de processadores pode ser abastecida pela larga escala de produção destes componentes, resultando numa diminuição de custo, e da necessidade de componentes feitos sob medida, diminuindo assim o tempo necessário para a construção dos supercomputadores.

O interesse na construção de máquinas paralelas pode ser pelo fato que nesta década, o aumento do desempenho dos computadores mais velozes foi em torno de 500 vezes, ao passo que o aumento dos processadores individuais ficou em torno de 15 vezes, o que mostra que mesmo o mais rápido processador atual ainda não pode suprir a demanda por poder computacional necessária para certas computações científicas [STE94].

Um questionamento que pode surgir é que se em alguns anos o poder computacional dos atuais computadores paralelos estará embutido em um único *chip* então qual a necessidade de pesquisa nesta área? Além da já tão discutida limitação tecnológica que um dia imporá uma barreira para o aumento do desempenho de um chip [SIA97], uma outra resposta a este questionamento seria que a espera de alguns anos para algumas áreas que necessitam vorazmente de cada vez mais poder computacional não é razoável e mais ainda, provavelmente a união destes *chips* mais poderosos poderiam fornecer um poder computacional ainda maior.

Podemos observar também que o poder computacional disponível com o uso de sistemas paralelos é imensamente maior que aquele que pode ser obtido com um processador único [CUL99]. E devido a esta disponibilidade muitas áreas fazem uso deste poder de computação podendo executar suas aplicações com a melhor resolução possível, avançando assim em suas pesquisas, por exemplo, a tradicional área de meteorologia.

2.1.2 Tendência em Arquitetura Paralela

Como citado na seção anterior, 2.1.1, os computadores paralelos atuais seguem a tendência de utilização dos mesmos microprocessadores que geralmente são utilizados em computadores pessoais e estações de trabalho, tornando-se assim parte do estudo das arquiteturas paralelas. Podemos observar que as arquiteturas dos microprocessadores apresentaram nos últimos anos avanços evolucionários [CUL99], visto que do início da década de 80 até os dias atuais eles vêm incorporando a maior parte das características avançadas dos *mainframes* e supercomputadores dos anos 60 e 70, bem como novas técnicas [DAL99]. Foram adicionados ao processador durante esse período caches, unidades de gerenciamento de memória, unidades de ponto flutuante e multiplicadores em *pipeline* e funcionalidade para múltipla emissão de instrução e ainda emissão de instruções fora de ordem. Além da adição desses recursos, o tamanho da palavra foi quadruplicado, de 16 para 64 bits [DAL99].

Entretanto a arquitetura individual dos processadores é apenas uma parte da explicação de como o poder computacional dos sistemas paralelos tem crescido. No levantamento de sistemas de computação paralela e vetorial apresentado no relatório técnico *Overview of Recent Supercomputers* de 1999 [STE99], podemos observar que a maior parte dos sistemas disponíveis utiliza a arquitetura MIMD com memória distribuída ou compartilhada.

Vemos que o mercado vem seguindo uma tendência de organização parecida com a organização genérica de um multiprocessador escalável, apresentada por Culler [CUL99]. A **Figura 1** ilustra tal sistema que é composto de uma coleção de computadores essencialmente completos (incluindo um ou mais processadores e memória), comunicando-se através de uma interconexão escalável de alta performance de propósito geral.

Apesar da convergência no sentido de uma organização comum, ainda não existe um modelo geral devido à complexidade do fluxo de execução e controle, o que ainda não

permitiu a elaboração de um modelo como a máquina de von Neumann. Segundo Ian Foster [FOS95], o avanço obtido com os computadores seqüenciais deve-se muito a este modelo geral e, portanto espera-se o mesmo avanço quando for estabelecido este modelo geral para máquinas paralelas e distribuídas.

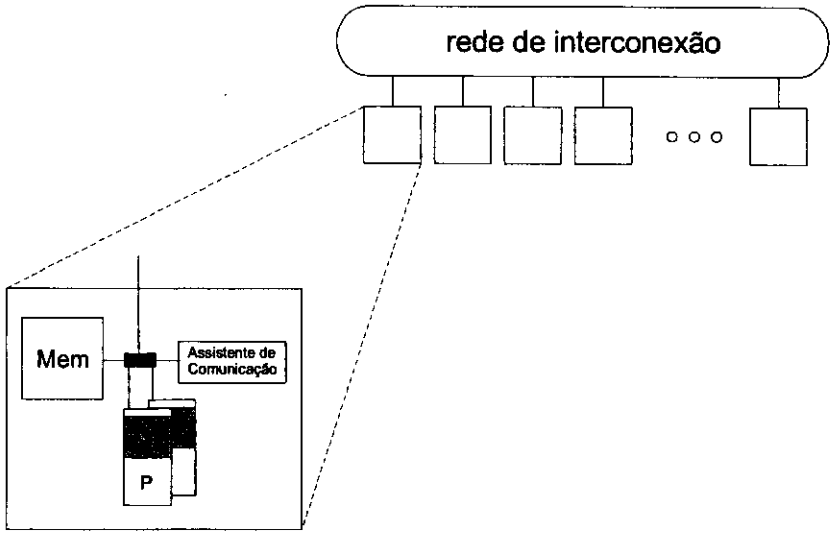


Figura 1 Organização genérica de um multiprocessador escalável. Fonte [CUL99]

Podemos observar os benefícios desta tendência notando que as aplicações começam a ficar mais portáteis, sendo possível a sua execução em vários tipos de sistemas paralelos diferentes [CUL99]. Podemos observar também a existência de um amadurecimento no paradigma de programação pelo fato de estarem surgindo padronizações para sistemas paralelos e distribuídos, por exemplo, nos sistemas de passagem de mensagem com MPI [MPI93, MPI94] e a tecnologia *Middleware* [BAK00] baseada em objetos distribuídos com CORBA [VIN97]. Vemos, entretanto que devido à falta de um modelo de fluxos de execução e controle muitas vezes alguns ajustes são necessários nas aplicações, principalmente àquelas voltadas à programação paralela de modo que tenha execução ótima em um dado sistema, ou seja, divisão e balanceamento de carga ótimos.

Como a alteração de programas nem sempre é uma tarefa fácil, quanto mais adequados forem linguagem e compilador, menor será o trabalho para o projetista de uma aplicação. Assim uma das bases para a viabilização de arquiteturas paralelas é a utilização de compiladores paralelizantes para uma dada linguagem de forma que o programador preocupe-se apenas com a aplicação que está desenvolvendo e não com a paralelização.

Para entendermos melhor esta colocação abordaremos nas próximas duas seções o assunto linguagens funcionais e compiladores paralelizantes.

2.2. PROGRAMAÇÃO E LINGUAGENS

2.2.1 Modelo de Programação Paralela

Um programa paralelo consiste de um ou mais fluxos de controle de execução operando sobre os dados. Neste modelo de programação devemos ter especificado quais dados podem ser nomeados nos fluxos de controle de execução, que operações podem ser realizadas sobre os dados nomeados e qual a ordenação existente entre essas operações [CUL99].

A nomeação e o conjunto de operações são o que tipicamente caracterizam os modelos de programação; entretanto no modelo paralelo à ordenação tem uma importância chave. Um programa paralelo deve coordenar as atividades de seus fluxos de execução para assegurar que as dependências dentro de um programa sejam forçadas; isto requer que sejam utilizadas operações de sincronização explícita quando a ordenação implícita das operações não é suficiente.

Do ponto de vista dos construtores de compiladores é necessário o entendimento das propriedades de ordenação para que seja possível escolher as estratégias de ordenação que melhorariam o desempenho do programa paralelo. Os modelos de programação mais utilizados atualmente são aqueles baseados em memória compartilhada e passagem de mensagem.

O modelo de programação com endereçamento compartilhado assume a existência do compartilhamento do mesmo espaço de endereço e o modelo de passagem de mensagem assume que os processos operam em espaços de endereços privados, com as operações utilizadas nos uniprocessadores e operações adicionais de envio e recepção de mensagem, que podem operar tanto localmente como globalmente.

Podemos pensar então que basta seguir estes modelos de programação paralela para o desenvolvimento de aplicações paralelas. Entretanto a tarefa não é tão fácil como parece. Segundo [PIE93] potencialmente temos três fontes para a perda do paralelismo: a primeira é a imperícia do programador em expressar o paralelismo em seu problema, a segunda é a

inabilidade do compilador em descobrir paralelismo no programa do usuário e a terceira a inabilidade do compilador em expressar este paralelismo na máquina destino.

Somado a isto temos ainda o problema da inadequação das atuais linguagens para a utilização em programação paralela [PIE93, CUL99, REA93]. Para um melhor entendimento desta afirmação abordaremos na seção seguinte características de linguagens imperativas e linguagens declarativas que as tornam atrativas ou não para a programação paralela.

2.2.2 Linguagens Imperativas

De uma forma simplista podemos definir um programa como uma seqüência definida de instruções para uma determinada máquina. Em uma visão mais completa consideramos os programas como descrições de valores, propriedades, métodos, problemas e soluções [REA93]. A função da máquina seria acelerar a manipulação e avaliação dessas descrições para fornecer soluções para problemas particulares. E a linguagem de programação seria a convenção para a escrita das descrições que podem ser avaliadas.

As linguagens de programação convencionais têm como base o comando de atribuição, em torno do qual são construídas as abstrações de controle e seqüência, desvios, repetições, etc. As linguagens desse tipo são conhecidas como imperativas (ou procedurais), devido ao fato das instruções serem executadas seguindo as ordens no programa, que são na realidade abstrações do modelo de von Neumann para máquinas seqüenciais cujos mecanismos de controle são abstrações do contador de programa para localizar a próxima instrução ou realizar saltos.

A existência de variáveis referindo-se diretamente a células de armazenamento exige que o programador se preocupe com a utilização e reutilização para a manutenção dos diferentes dados. Com isso o programador fica encarregado durante a programação de realizar a descrição do que deve ser computado, organizar a seqüência correta da computação e organizar o gerenciamento de memória durante a computação.

O ideal seria o programador concentrar-se apenas na primeira das três tarefas [REA93], descrição do que deve ser computado, sem a preocupação com as outras duas que são tarefas mais de administração. A separação destes conceitos também traz como vantagem a maior facilidade na prova de correção de programas.

Assim, a automatização dos aspectos administrativos significa que o implementador do compilador para uma linguagem deste tipo deveria tratá-los, possibilitando assim o uso de diversos mecanismos de computação para diferentes arquiteturas. Em particular, o programador pode querer usar todo o paralelismo disponível e ter o balanceamento correto no uso dos recursos. Muitas vezes um compilador bem escrito pode realizar uma otimização mais eficiente que um programador, especialmente quando não está amarrado por detalhes de programa desnecessários ou inapropriados.

2.2.3 Linguagens declarativas

O projeto de uma linguagem de mais alto nível deveria levar em conta a expressividade na descrição dos problemas, de modo que situações, métodos e solução não sejam difíceis de escrever; a possibilidade de extensão, permitindo que a partir de uma base simples obtenham-se construções mais complexas, existência de meios para a prevenção de erros que possam ser cometidos pelos usuários, a possibilidade de ser eficiente nas máquinas correntemente disponíveis e ser matematicamente elegante a fim de permitir o suporte à maioria das atividades de programação.

As linguagens declarativas (também chamadas aplicativas ou descritivas), que são consideradas de mais alto nível, são fundamentadas principalmente em expressões, visando reduzir a influência dos detalhes de máquinas particulares e enfocando as descrições matemáticas.

Uma das principais ausências nas linguagens declarativas é o comando de atribuição, substituído por formas alternativas para a descrição de valores e cálculos. As linguagens declarativas são usualmente planejadas a partir de princípios matemáticos e como tais centram-se na descrição e raciocínio de objetos complexos e, por conseguinte provêm um modelo de programação claro e fácil de usar, o que facilita o desenvolvimento de algoritmos e simplifica a compilação [FEO90].

Existem duas classes de linguagens declarativas atualmente, as linguagens funcionais [HUT00], tais como *SISAL* e *Haskell* [REA93], e as linguagens lógicas, tal como *PROLOG*, [HUT00]. A primeira baseia-se em expressões descrevendo dados (construídas de aplicações e definições de funções) e a segunda nas asserções lógicas descrevendo os relacionamentos entre os valores de dados e descrições automáticas de

respostas às questões das asserções. Nos dois casos os programas podem ser vistos como descrições dos programas ao invés de instruções para a computação de valores ou efeitos.

Uma linguagem declarativa evita o paralelismo explícito em favor do paralelismo implícito que deve ser extraído pelo compilador [WOL96]. Os compiladores para essas linguagens devem realizar várias otimizações com o intuito de melhorar a qualidade do código gerado (velocidade de acesso e eficiência no uso da memória), como por exemplo, as otimizações relativas à regra de associação única que normalmente insere cópias de dados, já que um valor não pode ser redefinido. Portanto mesmo numa expressão em que apenas uma posição de um *array* *é alterada*, existe a criação de um novo nome para o "novo *array*", gerando assim uma cópia praticamente do *array* inteiro. Para eliminar ou na pior das hipóteses minimizar o efeito do uso excessivo de memória temos análises já bem estudadas e testadas que podem ser aplicadas, conhecidas como *build-in-place* e *update-in-place* [CAN92a, FEO90, WOL96]. Também, a tecnologia para a otimização de linguagens declarativas é a mesma das linguagens imperativas, apenas o processo de encontrar operações paralelas difere, mas os mesmos métodos de análise e as mesmas estratégias de otimização e mapeamento de código podem ser utilizadas em ambas [WOL96].

Tabela 1. Classificação de linguagens funcionais [HUT00]

	Seqüencial		Concorrente	
Estrita	ASpecT	Mercury	Erlang	Pizza
	Caml	ML	NESL	*SISAL*
	FP	OPAL	Oz	
	J	Scheme		
Não Estrita	Gofer	Hugs		
	Haskell	Miranda	Clean	Id
	Hope			

Na tabela temos como uma das linguagens funcionais, *SISAL*, linguagem está que foi a escolhida para o trabalho realizado e apresentados em mais detalhes na seção 2.4.

Apresentaremos a seguir, na **Tabela 1**, uma lista com várias linguagens funcionais [HUT00] classificando-as como estrita/não-estrita[3] [LOI98] e seqüencial/concorrente.

[3] Uma linguagem funcional estrita é aquela onde as expressões são avaliadas antes de serem passadas como argumentos às funções a não estrita é aquela onde as expressões são avaliadas apenas quando necessário [REA93].

Podemos observar que com relação à programação paralela existe uma tendência em direção a uma abordagem mais declarativa para programação o que tende a reduzir a distância programação e especificação. Assim qualquer especificação será um programa executável com variações do grau de eficiência, que depende das transformações aplicadas. Também ferramentas de transformações de propósito geral podem ser construídas e fornecidas com o compilador, como uma forma de auxílio ao programador.

2.3 TRANSFORMAÇÕES EM COMPILADORES

2.3.1 Compiladores

As pesquisas para construção de compiladores vêm sendo realizadas desde a metade dos anos 50 [MUC97] e já com as bibliotecas otimizadas de FORTRAN podemos observar qual a importância da geração de um código eficiente, que em grande parte foi um dos motivos de seu sucesso. Atualmente, o sucesso dos compiladores modernos também está relacionado à qualidade do código que este consegue gerar, qualidade esta obtida através de diversas transformações visando à otimização do código gerado [YEW97, MUC97, BAC93]. Assim, um compilador projetado para produzir um código objeto veloz deve inclui componentes otimizadores.

Seguindo essa tendência, nas últimas três décadas diversas técnicas para transformações e otimizações de código foram implementadas e com sucesso, grande parte nos compiladores para sistemas monoprocessados [BAC93]. Estas otimizações visam sempre a maior eficiência do código executado, seja pela redução do número de instruções, pelo acesso eficiente ao sistema de memória ou aproveitamento de unidades funcionais de forma mais eficiente.

Para atingir esses objetivos, os compiladores modernos utilizam algoritmos de otimização que exigem intensivo processamento e grande espaço de memória. Nas **Figuras 2** e **3** abaixo [MUC97], podemos ver duas das principais estruturas utilizadas em um compilador otimizador.

Normalmente um compilador que realiza as transformações utilizando um código intermediário [COH94, MUC99] de nível baixo, próximo da linguagem de máquina, é mais dependente de características de hardware do que um compilador que utiliza um

modelo misto com código intermediário de nível médio e nível baixo, pois no código de nível médio são realizadas as otimizações independentes de arquitetura. Assim, a representação intermediária é um elemento extremamente importante num compilador, que deve ser projetado cuidadosamente para não ser o fator limitador das análises que serão realizadas pelo compilador [COH94].

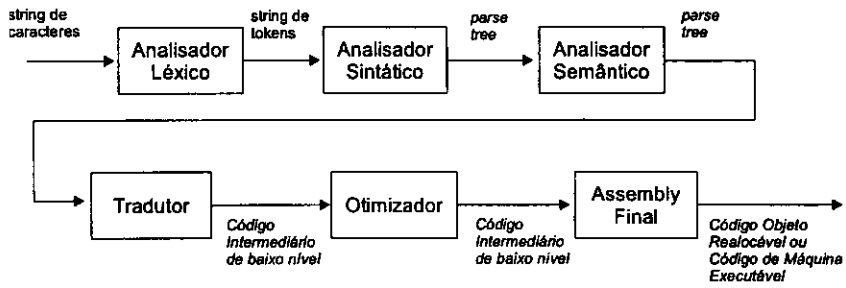


Figura 2 **Estrutura para um compilador com a otimização realizada no código intermediário de baixo nível [MUC97]**

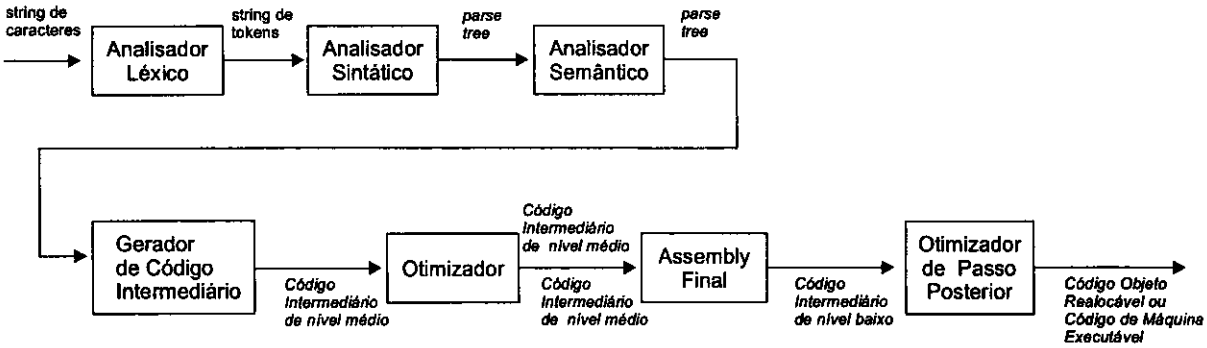


Figura 3 **Estrutura para um compilador com a otimização dividida em duas fases, uma código intermediário de nível médio e outra no código intermediário de baixo nível [MUC97]**

Um compilador que utiliza um otimizador de modelo misto é mais facilmente adaptado a novos e diferentes tipos de arquiteturas, enquanto um compilador que utiliza um otimizador de baixo nível é muito pouco portátil entre arquiteturas a menos que elas sejam praticamente iguais. Em ambos os modelos a fase de otimização analisa o código e transforma o código na tentativa de melhorá-lo de forma que este realize as tarefas a que foi projetado da forma mais eficiente possível. Mais detalhadamente temos na **Figura 4**

uma relação das otimizações normalmente realizadas nos compiladores modernos com uma otimização agressiva.

Podemos citar como exemplos de compiladores que utilizam o modelo misto os compiladores das linguagens C, C++, Fortran 77 e Pascal para SPARC da Sun Microsystems, os compiladores das linguagens C, C++, Fortran 77, Fortran 90, Ada, Cobol, PL/I e BLISS para *Alpha* da *Digital Equipament Coorporation* (projeto GEM) e compiladores C, C++, Fortran 77 e Fortran 90 da Intel para x86. O modelo de baixo nível é utilizado nos compiladores XL para POWER e PowerPC da *IBM* e nos compiladores para PA-RISC da *Hewlett-Packard* [MUC97].

As técnicas de otimização para compiladores paralelos e vetoriais são também um assunto bastante estudado e extenso e será abordado na próxima seção.

2.3.2 Compiladores Paralelizantes

Os compiladores paralelizantes são aqueles que analisam um código fonte, realizando transformações e análises complexas e otimizações agressivas, para a geração de um código que seja eficiente para a execução em paralelo [AND97]. As otimizações para processadores vetoriais e paralelos de alta performance buscam maximizar o paralelismo e a localidade de memória e para isso fazem uso de técnicas de análise de dependência e distribuição de dados geralmente internos aos *loops* [BAC93]. Essas otimizações são cruciais para a obtenção de performance em sistemas distribuídos devido ao alto custo da comunicação de dados pela rede de comunicação, o que poderia inviabilizar o processamento em paralelo.

Nos últimos 30 anos, técnicas para a construção de compiladores para computadores paralelos foram desenvolvidas tanto no meio acadêmico como no comercial [WOL96]. Os diversos trabalhos disponíveis na literatura sobre construção de compiladores paralelizantes enfocam questões como o que pode ser paralelizado, quais otimizações podem ser realizadas e quais técnicas são requeridas para a otimização. Entretanto mais recentemente estão sendo tratadas outras questões igualmente importantes tais como: quão efetivamente os compiladores paralelizantes expõem o paralelismo, qual é a contribuição de cada técnica de otimização e análise, como as técnicas de otimização e análise interagem e que técnicas de otimização e análise seriam benéficas [PIE93].

Por um longo tempo as pesquisas sobre compiladores paralelizantes estavam focadas na exploração do paralelismo de granularidade baixa, devido ao tipo de arquitetura disponível até então. Nesta década observamos que arquiteturas multicomputadores estão se tornando cada vez mais disponíveis [DON99, MEU00], entre estas aquelas com memória distribuídas. Neste tipo de arquitetura o tipo de paralelismo explorado é aquele com granularidade grossa. Nesse sentido vemos que vários compiladores paralelizantes seguem nessa direção. Entre os mais conhecidos estão SUIF da Universidade de Stanford [AMA93], Polaris [FAI94], Parafrase-2 [POL89] e PARADIGM [BAN95, GUP92a, GUP92b] da Universidade de Illinois.

Uma característica comum a todos estes compiladores paralelizantes é que todos eles são projetados para analisar códigos de linguagens seqüenciais tradicionais como Fortran e C na sua maioria, visando o aproveitamento de todo o código já escrito nessas linguagens. Apenas recordando que os compiladores tradicionais para essas linguagens são projetados para gerar códigos que exploram o paralelismo dentro do processador, ou seja, de granularidade fina e a execução em sistemas de memória distribuída a granularidade grossa é a exigida.

A utilização dessas linguagens tradicionais como fonte exige análises e transformações para que seja exposto o paralelismo dos programas [PIE97, AMA97]. Normalmente essas análises e transformações concentram-se sobre *loops*, pois são elas as prováveis fontes de paralelismo desses programas, principalmente aqueles desenvolvidos para aplicações científicas, que são na realidade o domínio mais comum dos compiladores paralelizantes [AND97].

As transformações em *loops* têm sido reconhecidas como importantes componentes da tecnologia para construção de compiladores paralelizantes. A maioria dos compiladores paralelizantes aplica um conjunto de transformações de *loops* básicas, uma por vez [KNI95]. A cada passo deve ser decidido se uma transformação é legal, ou seja, respeita as dependências de dados, e se é benéfica sua aplicação. As diversas pesquisas de análise e transformação mostra que a melhor estratégia para se obter a eficiência desejada é uma combinação de análises e transformações, ou seja, apenas uma técnica em geral não é suficiente para a obtenção de um código eficiente.

Um conjunto de transformações referidas como transformações unimodulares [KNI95, PIE97, WOL96] englobam uma larga faixa de transformações básicas sobre *loops* que pode ser representada como matrizes unimodulares e não singulares. Na realidade qualquer transformação linear modelada com uma matriz não singular pode ser vista como

uma composição ou produto das quatro transformações básicas: permuta de *loop*, reversão de *loop*, *loop skewing* e mudança de escala de *loop* [KUL92, BAC93]. Somada a estas transformações unimodulares, uma outra conhecida como *tiling* [JIM97, WOL96] pode também ser utilizada. A transformação *tiling* visa melhorar a localidade e também pode ser empregada para expor o paralelismo de grão grosso. As técnicas convencionais de *tiling* são implementadas através do uso das transformações *strip-mine*, *intercalação* e *unroll* [JIM97, WOL96].

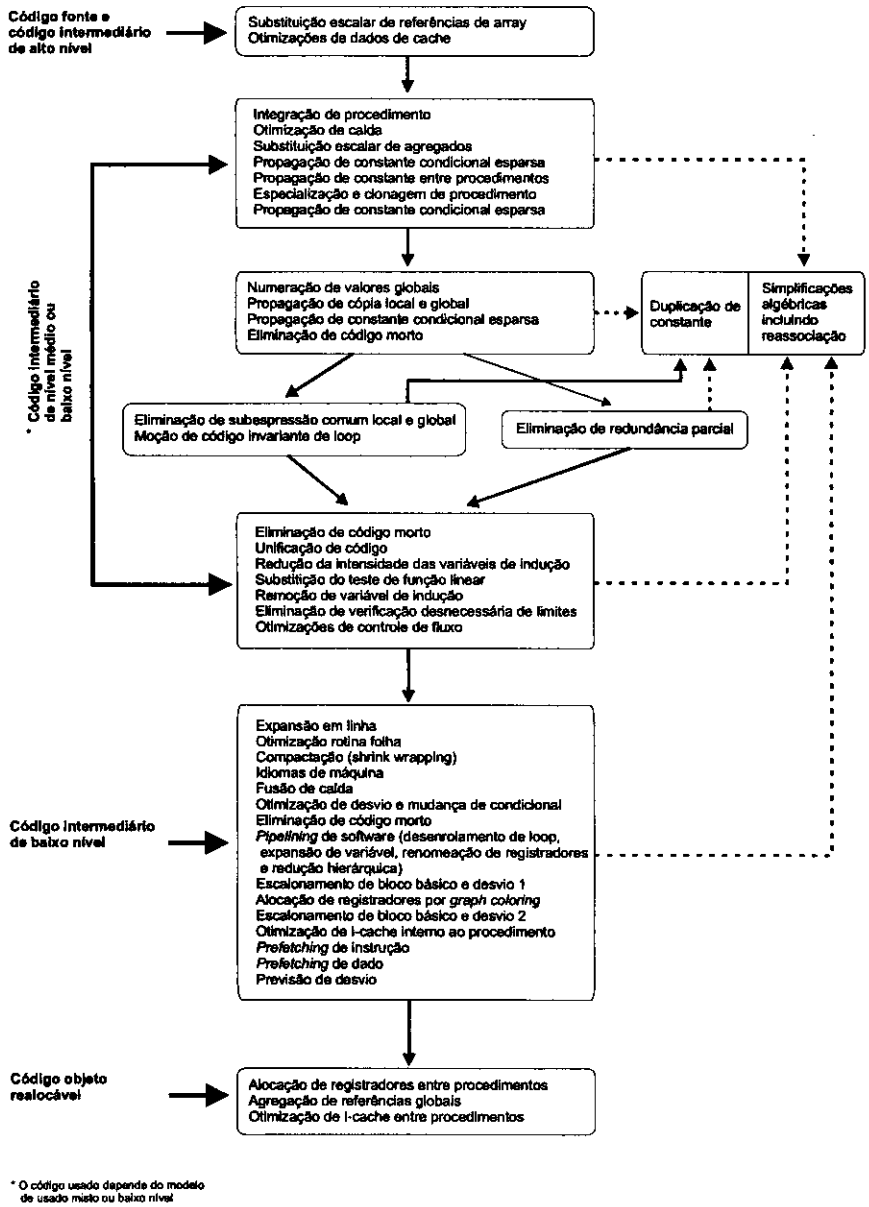


Figura 4 Ordem das otimizações [MUC97]

Além das otimizações para a exposição do paralelismo e melhoria da localidade de dados, também são necessárias otimizações visando a exploração dos benefícios obtidos com as transformações prévias. Para tanto devem ser realizadas análises de dados e computação visando o mapeamento de dados e computação ao sistema disponível de forma que este seja explorado da melhor maneira possível. A maioria dos compiladores paralelizantes necessita da ajuda do usuário e extensões da linguagem para que possam realizar este mapeamento. Temos como exemplo o padrão HPF (*High Performance Fortran)* [ROT97, WOL96]. Esse tipo de abordagem exige então que o programador realize as análises de decomposição e a forneça ao compilador usando as extensões da linguagem o que na maioria das vezes não é uma tarefa trivial [AND97]. Assim seria desejável que o próprio compilador realize esta tarefa de análises complexas que possam conduzir à geração de um código eficiente.

Existem diversas pesquisas tratando o problema da distribuição automática de dados diferindo apenas na estrutura selecionada para representar o problema, o modelo de estimativa de performance e as técnicas adotadas para encontrar uma solução [GAR97]. A maioria delas divide o problema estático em dois passos principais independentes, o alinhamento e a distribuição. O alinhamento tenta achar uma relação entre as dimensões dos diferentes *arrays*, segundo algum padrão de acesso e então o passo da distribuição determina quais das dimensões alinhadas serão distribuídas e o número de processadores que serão a elas associados.

Entre os diversos trabalhos podemos citar o modelo desenvolvido por Li e Chen [ULR95 apud LC90a, LC90b, LC91a, LC91b] para o compilador para a linguagem Crystal, que utiliza um grafo de afinidade de componente (*Component Affinity Graph* - CAG) para resolver o problema de alinhamento, discutido na seção 3.1.2, e para resolver o problema de distribuição, discutido na seção 3.1.3, um algoritmo heurístico. Cada rotina tem o custo dependente da arquitetura parametrizado em termos do número de processadores envolvidos na moção e quantia de dados movidos. Wholey utiliza no compilador ALEXI [WHO91] do *Carnagie Mellon University*, o modelo de preferência de grafo desenvolvido por Knobe, Lukas e Steele [KNO90] para arquiteturas SIMD, solucionando através de um algoritmo também baseado em técnicas heurísticas mais especificamente ele utiliza o método de busca *hill climbing* para solucionar o problema da distribuição de dados. Gupta e Banerjee [GUP92a, GUP92b], da *University of Illinois Urbana-Champaign*, utilizam no compilador PARADIGM o modelo CAG desenvolvido por Li e Chen, realizando alterações na fase de distribuição de dados. Ambos os modelos

anteriores não tratam da decomposição dinâmica. Um levantamento de modelos de alinhamento e distribuição de dados pode ser encontrado em [GAR97].

O modelo utilizado neste trabalho está baseado no trabalho de Anderson e Lam [AND97] que implementaram seu algoritmo no compilador SUIF da Universidade de Stanford. Elas desenvolveram uma estrutura matemática para a expressão e cálculo de decomposições de dados e computação e apresentam um algoritmo que tenta encontrar uma decomposição que maximize o paralelismo e não requeira uma comunicação custosa. Elas descrevem um grafo de comunicação para modelar o problema da decomposição dinâmica e um algoritmo heurístico para encontrar as decomposições dinâmicas finais. No Capítulo 3 discutiremos com mais detalhe este modelo.

2.4. *SISAL*

2.4.1. A Linguagem *SISAL*

SISAL ou *Streams and Iteraction in a Single Assignment Language* é uma linguagem aplicativa de propósito geral que pode ser usada tanto em sistemas convencionais como em sistemas distribuídos [FEO90, MGR85, SKE91]. A primeira versão de *SISAL* surgiu em 1983 do trabalho conjunto de três instituições de pesquisa, *Lawrence Livermore National Laboratories*, *Colorado State University* e *University of Manchester* mais a empresa *Digital Equipment Corporation* e foi revisada em 1985 e desde esta versão continua a mesma. Uma outra versão de *SISAL*, nomeada *SISAL* 90 [KIM96] foi definida, mas não temos notícias da implementação de um compilador.

Com o projeto da linguagem *SISAL* foi definido uma linguagem aplicativa de propósito geral e uma linguagem para uma forma intermediária independente; foram desenvolvidas técnicas de otimização para computação aplicativa paralela de alto desempenho; e um ambiente de microtarefas que suporta *dataflow* em sistemas de computador convencionais [FEO90, MGR85] e atingiu-se um desempenho de execução comparável àquela de FORTRAN [FEO90, KIM96] em aplicações científicas de larga escala, validando assim o estilo de programação aplicativo.

A sintaxe da linguagem *SISAL* foi projetada para ser parecida com Pascal visando facilitar o aprendizado e melhorar a legibilidade. Quanto à semântica de tipos esta

IFSC-USP SERVIÇO DE BIBLIOTECA E INFORMAÇÃO

linguagem é considerada fortemente tipada, as funções não apresentando efeito colateral algum e são fundamentadas na concisão matemática. As referências dos programas são transparentes, visto que os nomes são ligados a valores e não a posições de memória, não existindo então *aliasing*. A linguagem é de associação simples, ou um nome é associado a um valor apenas uma vez. Essas propriedades tornam trivial a transformação do código fonte para um grafo *dataflow*.

Um programa *SISAL* pode ser composto de unidades de compilação separadas, cada uma incluindo uma lista das funções exportadas que serão visíveis fora do módulo, a definição destas funções e definições de funções locais adicionais. Uma função pode não ter argumentos de entrada, mas deve retornar um ou mais valores. Os argumentos de entrada e saída são declarados no cabeçalho da definição do programa. Uma função tem acesso somente ao seus argumentos, não existe efeito colateral. Além disso, uma função não retém informação de estado de uma invocação para outra, cada invocação de função é estritamente independente. Portanto os valores retornados por uma função dependem somente dos valores de argumentos apresentados a ela, ou seja, uma função de *SISAL* implementa uma verdadeira função no sentido matemático.

Os tipos escalares suportados em *SISAL* são *boolean*, *character*, *integer*, *real* e *double precision* e os tipos agregados são *array*, *record*, *union* e *stream*. Os *records* têm um formato fixo, cada campo tem um tipo especificado. Um *array* possui um conjunto de índices de inteiros e seu componentes são de tipo arbitrário, mas uniforme. Uma *stream* é uma seqüência ordenada de valores de tipo arbitrário, mas também uniforme. Os componentes de uma *stream* são acessíveis somente através de um conjunto de funções pré-definidas (por exemplo, *stream_first*, *stream_rest* [Can92a]). Os tipos union podem ser formados de *tags* que permitem um dentre um conjunto de tipos constituintes.

Cada tipo de dados tem um conjunto de operações e predicados associados. Os tipos *array*, *stream*, *union* e *record* são tratados como conjuntos matemáticos de valores, da mesma forma que os tipos escalares básicos. As operações em *arrays*, *streams*, *unions* e *records* são selecionadas para suportar a identificação de concorrência para a execução num processador altamente paralelo [MGR85].

Visto que *SISAL* é uma linguagem livre de efeitos colaterais, as subexpressões podem ser avaliadas em qualquer ordem sem afetar os resultados computados. Assim as estruturas de controle de *SISAL* usam a avaliação da forma sintática de uma expressão [MGR85] que produz uma *tupla* de valores. Existem construções da linguagem para expressões condicionais (*if / then/ else*), e expressões para iterações, *loops* seqüenciais (*for*

initial) e *loops* paralelos (*forall*) [MGR85]. Os *loops* seqüenciais assemelham-se ao *loops* de linguagens convencionais, mas com semântica de associação única; são compostos de uma parte de inicialização, o corpo, teste para término e cláusula de retorno e esse tipo de estrutura normalmente apresenta uma granularidade de grão fino. Os *loops forall* são compostos de elementos independentes, assim os valores definidos dentro de iteração não podem ser acessados fora dela; são compostos de uma parte geradora de série, corpo e cláusula de retorno. O gerador de série é um produto escalar ou cartesiano de um conjunto de seqüências ou *scatters*. Uma instância do corpo é executada para índice, valor ou *n-tupla*. A cláusula de retorno define os resultados da cardinalidade da expressão.

2.4.2. *Forall*

Existem duas forma de construção *for* em *SISAL*, Tabela 2, uma das quais permite que sejam especificados produtos (cartesianos) índices de *array* e *streams* internos ou externos (forma produto) e a outra que não (forma "não-produto") [MGR85]. Nos trabalhos desenvolvidos foi utilizada a primeira forma também conhecida como construção *forall*, escolhida em razão da possibilidade de extração de paralelismo de grão grosso.

Tabela 2. Exemplos do comando *forall* de *SISAL*

Operações	Comentários
`for x in A`	Um scatter
`for i in 1, n`	Uma seqüência
`for x in A dot y in B`	Um produto escalar de dois *scatters*
`for i in 1, n cross j in 1,m`	Um produto cartesiano de duas seqüências

A forma produto da construção for é uma versão especial da forma não-produto que fornece um meio mais conciso para especificar índices e conjuntos de elementos de *array* e *stream*. Os valores da *tupla* resultante depende somente dos valores definidos por `in-exp-list` e `decldef-part`, da sintaxe para construção *forall*, **Figura 5**.

A construção **`for`** da forma produto pode inserir zero ou mais nomes para valores de índices do tipo inteiro, zero ou mais nomes para valores dos elementos de um *array* ou *stream* e zero ou mais nomes para valores temporários. Pelo menos um nome para valor de índice ou elemento deve ser introduzido, isto é, deve existir pelo menos uma `in-exp-`

`list`, **Figura 6**, seguindo a palavra `for`. Os nomes para valores de índices, elementos e temporários devem ser todos diferentes para o escopo da construção inteira, exceto em qualquer bloco interno.

```
iteration-exp ::= for in-exp-list
                  [ decldef-part ]
                  returns return-exp-list
                  end for
```

Figura 5 Sintaxe da construção `for` na forma produto [MGR85]

```
in-exp-list ::= in-exp [dot in-exp] ...
                in-exp [cross in-exp]...
```

Figura 6 Regra de produção da geração de índices da construção `for` na forma produto [MGR85]

Uma seqüência de expressões de índice pode ser fornecida em `in-exp-list`, separada pela palavra reservada `dot`. Esta definição de série produz produtos internos de expressões de índice. Se todas as séries não contiverem o mesmo número de valores de índice, o número de *tuplas* gerado será aquele da série mais extensa.

A outra forma para a seqüência de expressões de índices é fornecida com a lista de expressões, `in-exp-list`, separada pela palavra reservada `cross`. A série de índices formada é o produto cartesiano das expressões de índices.

O valor resultante do `for` na forma produto é a *tupla* de valores definido pela `return-exp-list`, **Figura 7**. Visto que todos os elementos da série são independentes uns dos outros, a ordenação natural não é tão clara. Entretanto para assegurar uma ação determinada, *SISAL* impõe uma ordenação dos resultados produzidos pela forma produto.

O caso básico é uma expressão de série simples. Neste caso, cada cláusula resultado é avaliada uma vez para cada elemento da série. Os resultados dessas avaliações são colocadas na seqüência, usando as expressões de ordenamento da série (do menor para o maior) como ordenação da seqüência. As cláusulas de mascaramento [MG85] podem ser usadas para eliminar elementos específicos desta seqüência. Portanto a seqüência resultante pode ser menor que o tamanho original da série. As expressões das séries geradas com `dot` produzem resultados de uma maneira análoga.

As expressões da série **cross** diferem, pois elas produzem seqüências de seqüências em seus padrões para geração de resultados. Se não existe cláusula de mascaramento, a estrutura dos resultados equivale exatamente à estrutura da série **cross**. A seqüência mais externa corresponde à primeira expressão de série no cabeçalho. As cláusulas de mascaramento podem ser aplicadas como nos casos anteriores, mas o resultado vai ser um tanto surpreendente. Se um elemento resultado é mascarado, ele causa uma compressão dos elementos válidos na última dimensão das séries descritas no cabeçalho. As compressões somente podem ocorrer nesta dimensão.

```
return-exp-list ::= return-clause ...

return-clause   ::= [ old ] return-exp-list [ masking-clause ]

masking-clause  ::= unless expression
                  | when   expression

return-exp      ::= value of [[direction] reduction-op] expression
                  | array of expression
                  | stream of expression

direction       ::= left
                  | right
                  | tree

reduction-op    ::= sum
                  | product
                  | least
                  | greatest
                  | catenate
```

Figura 7 Regra de produção da parte de retorno da construção `for` na forma produto [MGR85]

Se **value of** não for seguida por um dos operadores de redução, o resultado produzido por esta cláusula é o último elemento da seqüência como definido acima. No caso das séries de produtos cartesianos, é o último valor da última série de valores. Se ele for seguido por um dos operadores de redução, a operação é realizada sobre a coleção de valores na seqüência. A ordem na qual a operação de redução é realizada está sujeita as obrigações **left**, **right** e **tree**, se especificado. No caso das séries de produtos cartesianos, a operação de redução é realizada primeiro sobre todos os elementos dentro das seqüências dimensionadas no nível mais baixo e então sucessivamente aplicada até as seqüências dimensionadas no nível mais alto.

O `array of` produz um *array* tendo tamanho igual ao da seqüência de resultados como descrito acima e contendo exatamente aqueles valores na ordem definida pela seqüência. O limite inferior do *array* é igual ao valor do índice da cláusula série que produziu o primeiro elemento da seqüência. Se o *for* define uma série de produto cartesiano, então o *array* resultante é multidimensional, com a forma correspondendo à estrutura da seqüência de seqüências definida acima.

O `stream of` produz uma *stream* que tem tamanho igual àquele da seqüência de resultados como descrito acima e contém valores na ordem exata definida pela seqüência. Se o *for* definir uma série de produto cartesiano, então a *stream* resultante é multidimensional, com a forma correspondendo à estrutura da seqüência de seqüências definida acima.

2.5. A LINGUAGEM IF1 – INTERMEDIATE FORMAT 1

IF1 [SKE85a, SKE85b, SUW97] é uma linguagem intermediária baseada em grafos acíclicos que descrevem as dependências de dados. Ao contrário de outras linguagens intermediárias, o código IF1 não indica uma seqüência de instruções a serem executadas, ao contrário ela define um grafo de dependência de dados que visualmente descreve o fluxo de dados do programa.

Embora seja possível escrever um programa na linguagem IF1 ela foi originalmente projetada como uma forma de representar um programa de uma linguagem de mais alto nível num formato intermediário de mais baixo nível.

IF1 além de servir para separar o *front-end* das análises dependentes de máquina, é um formato pronto para ser utilizado por programas de otimização de código. Existem quatro componentes básicos para um grafo: nós que denotam as operações (como *add* e *divide* [SKE85a]), os arcos que representam os valores que são passados de um nó para outro, os tipos que podem estar junto a um dado ou função e fronteiras dos grafos cercando os nós e arcos.

Um nó pode ser executado assim que todas as suas entradas estejam disponíveis. Atualmente existem mais de cinqüenta nós definidos no IF1 [Ske85a]. Em geral o número de entradas e saídas dos nós varia de acordo com o tipo de operação que eles representam. Por exemplo, o nó *add* recebe dois arcos de entrada, realiza a soma e retorna o resultado

um arco de saída, como podemos ver na **Figura 8** abaixo um grafo que representa "`(a+b)/2`".

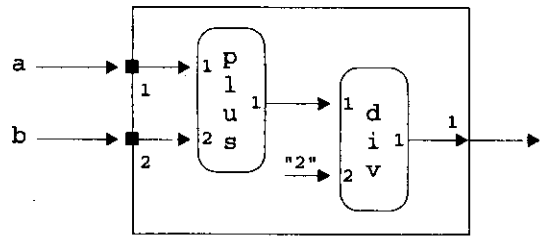


Figura 8 Grafo de `(a+b)/2`

Os números dentro do grafo indicam os números das portas, que são usados para distinguir entradas e saídas múltiplas. As setas indicam os arcos de entrada e saída, com um arco especial para representar o valor constante 2 (apenas destino, sem origem). A cada um destes arcos de dados está associado um tipo que não aparece representado na **Figura 8**. Os tipos podem ser especificados pelo usuário estendendo assim os tipos pré-definidos.

Um programa IF1 é um arquivo com diversas linhas com caracteres ASCII, apenas os que podem ser impressos, separadas pelo caractere de nova linha. O primeiro caractere não branco na linha indica o que ela representará, declaração de tipo, arco ou arco literal, função local ou global, grafo, nó simples ou composto e comentário. A **Figura 9** representa o código IF1 do grafo `(a+b)/2` mostrado na **Figura 8**, escrito segundo as regras de produção e semântica definidas no manual de especificação de IF1 [SKE85a].

Um grafo pode representar uma função ou as partes de um nó composto. No exemplo da **Figura 9** vemos um grafo representando a função `main`. Quando utilizado para representar nós compostos o grafo é classificado como um subgrafo, pois representa comandos que estão dentro da função, ou seja, o grafo.

Como um exemplo de grafos em um nó composto, tomemos o *forall*, onde temos os subgrafos gerador, corpo e retorno, **Figura 10**. O subgrafo gerador do *forall*, contém somente os nós *Ascatter* e *RangeGenerator*. Para os produtos escalares de *SISAL* um *Ascatter/ RangeGenerator* é colocado na lista de escalares, fora isso um nó gerará os valores do *loop*. Os produtos cartesianos são representados em IF1 como *loops forall* aninhados. O retorno dos subgrafos contém nós que tomam tipos múltiplo como entrada. As expressões na cláusula *returns* de *SISAL* podem fazer com que o código apareça no corpo do subgrafo ou fora do nó *forall*, ao invés de aparecer no subgrafo de retorno (visto que essas expressões podem necessitar de operadores cujas entradas não são do tipo

múltiplo). No corpo podemos ter tanto nós simples como compostos representando as mais diversas operações.

C	Media						Comentário
T	1	1					Declaração do tipo inteiro
x	0	"main"					Início da função principal
N	1	141					Nó plus
E	0	1	1	1	1		Arco para a entrada a
E	0	2	1	1	2		Arco para a entrada b
N	2	122					Nó div
E	1	1	1	1	1		Arco do nó div com o resultado de a + b
L			1	2	1	"2"	Arco do nó div com a constante 2
E	2	1	1	0	1		Arco com o resultado de (a+b)/2

Figura 9 Representação do grafo da Figura 8 em IF1

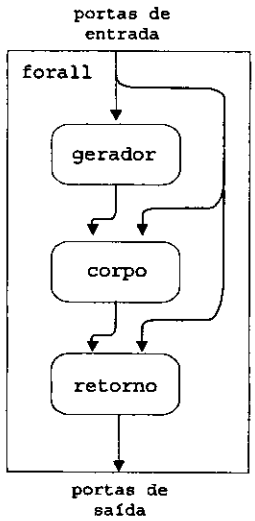


Figura 10 Nó composto *forall*

2.6. O COMPILADOR OSC

Na construção de compiladores para a linguagem *SISAL*, devemos observar duas importantes características que são a associação única e semântica orientada a valores. Isto na realidade significa que *SISAL* não suporta operações de atualização de memória, ou seja, sua semântica define que cada "variável" tem no máximo um valor ligado a ela no momento da execução. Por essa regra, uma variável é na realidade um nome para um valor e não um local de armazenagem, criado sempre que um novo valor é definido. Linguagens

de associação única não possuem antidependência de dados ou dependência de saída de dados e podem produzir mais paralelismo que as linguagens com operações de atualização à memória e outros efeitos colaterais.

No entanto, essas linguagens de associação simples geram uma cópia de dados para cada valor que for manipulado, o que pode ser um problema se eles forem tipos agregados, como por exemplo, os *arrays*. Em aplicações científicas, onde o uso de arrays já é grande, essas cópias podem se tornar custosas demais seja em tempo ou espaço. Em *SISAL*, as cópias resultam de duas classes de operações [CAN95]: aquelas que incrementalmente atualizam ou modificam agregados existente e aquelas que incrementalmente constroem um novo agregado. Na maioria dos programas *SISAL* [CAN95], felizmente as operações de cópia são inerentes somente da semântica da linguagem e não aos algoritmos nos programas.

Portanto, para que um compilador seja efetivo para uma linguagem de associação simples ele deveria conseguir minimizar esse problema de cópia de dados. O compilador *SISAL* do Lawrence Livermore National Laboratories resolve esses problemas através de diversas análises. Na realidade, junto com o projeto da linguagem *SISAL* foi também desenvolvido um compilador chamado OSC (*Optimizing SISAL Compiler*) [CAN92b, GAU97], cujo esquema está na **Figura 11**. Este compilador, como podemos ver pela figura, é composto de vários estágios, a saber: o *front-end* (PARSER) e o *back-end* (IF1LD até IF2PART).

O *front-end* (*parser*) do OSC é responsável por traduzir cada um dos módulos de um programa *SISAL* para um módulo em código intermediário IF1, visto anteriormente.

A segunda fase da compilação, IF1LD, toma cada um dos módulos IF1 gerados pelo parser e forma um único programa IF1 monolítico. Este código é então lido por um otimizador independente de máquina, IF1OPT que aplica otimizações convencionais [Can92b, MUC97] tais como expansão de função, remoção de código invariante, eliminação de subexpressão comum, duplicação de constante, fusão de *loop* e remoção de código morto.

As otimizações IF2MEM e IF2UP são as duas das mais importante no OSC. O objetivo dessas duas otimizações é analisar os grafos IF1 e aplicar otimizações de gerenciamento de memória de modo que as cópias dinâmicas de *array* sejam reduzidas.

O analisador *build-in-place* (IF2MEM), pré-aloca *arrays* onde é possível calcular os seus tamanhos finais, seja no momento da compilação ou na execução, tentando assim reduzir o problema da construção incremental de agregados. O resultado desta análise é um

programa semanticamente equivalente agora em IF2 [WEL96], uma linguagem de grafo que é uma extensão de IF1.

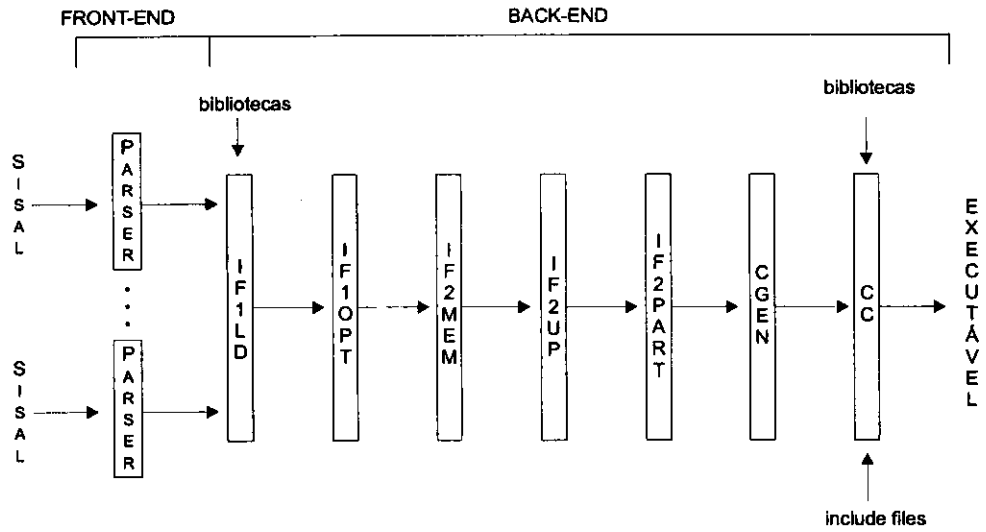


Figura 11 Processamento da Linguagem *SISAL*

O IF2 é um superconjunto de IF1, mas não é mais aplicativo porque suporta operações que diretamente referenciam e manipulam memórias "abstratas" (chamados nós AT). Ele inclui também arcos de dependência artificiais para a definição de restrições de sincronização no acesso a memória, *pragmas* contagem de referências para gerenciamento de agregados e *pragmas* marca para especificação dos direitos de acesso a agregados (mutável, imutável, possivelmente imutável) [FEO90, CAN95].

O passo seguinte da compilação, *updade-in-place* (IF2UP) [FEO90, CAN95] trata o problema da atualização incremental, que gera muitas cópia de agregados desnecessárias. Nesta fase os grafos são reestruturados para ajudar a identificar as operações de atualização que pode executar *in-place* e melhorar as chances para operações *in-place* no momento da execução, quando as análises estáticas falharem. A análise é realizada em três fases, uma delas prepara cada grafo IF2 para análises posteriores, inserindo nós especiais de duplicação de agregados visando desacoplar a cópia lógica dos modificadores de agregados no programa.

A meta das fases restantes é eliminar quaisquer duplicadores desnecessários. A fase dois reordena os nós em cada grafo inserindo arcos de dependência artificiais, que buscam atrasar a execução dos modificadores de agregados até o término das operações de leitura relacionadas. Esta fase também remove todas as operações de contagem de referência

[GAU97] que são desnecessárias em razão da reordenação de nós. Por último, a fase três elimina as operações de duplicação desnecessárias introduzidas na primeira fase e anota todos os nós transmitindo agregados com os direitos de acessos apropriados.

A análise que segue, IF2PART, é a fase da compilação que busca a paralelização de loops para execução concorrente ou vetorial. As análises realizadas em IF2PART definem a granularidade desejada de paralelismo e são baseada em estimativas de tempos de execuções e outros parâmetros [FEO90]. O usuário pode mudar esses parâmetros em tempo de compilação e tendo assim algum controle sobre a paralelização. Somente expressões *for* com custos estimados maiores que um certo *threshold* e aninhados com profundidade maior que algum nível de aninhamento paralelo (todos os níveis por *default*) são selecionados para o divisão.

Na última fase, o CGEN traduz os grafos IF2 otimizados para o código C, que são então compilados usando o compilador C local para produzir um programa executável. Diretivas de pré-processador fornecem as definições de operações e valores dependentes da arquitetura-destino. O software da biblioteca ligado durante esta fase de compilação fornece o suporte necessário para a execução em paralelo, gerenciamento de armazenamento e interação com o usuário.

Em geral o compilador OSC realiza as seguintes tarefas [GAU97]: traduz os programas fontes em imagens de memória executáveis, incluindo os componentes do sistema de tempo de execução requeridos para gerenciar automaticamente a memória, tarefas e entrada/saída. A quantia de paralelismo a ser explorado pelo programa pode ser controlada por opções do usuário, e após ter sido compilado, um programa pode ser executado por qualquer número de processos trabalhadores [GAU97], por meio de um simples parâmetro de tempo de execução.

Devido à sua natureza funcional, *SISAL* oferece paralelismo em vários níveis e pode ser destinado a uma variedade de modelos de execução concorrente. O sistema de tempo de execução do OSC suporta o modelo *fork-join* de execução paralela através do fornecimento de um *kernel* de micro-tarefas para a exploração do paralelismo no nível de *loop*. O modelo pode ser facilmente transportado para todos os multiprocessadores de memória compartilhada existentes comercialmente.

2.7 CONCLUSÃO

Apresentamos neste capítulo uma revisão geral da tendência atual de arquitetura paralela, a importância de linguagens e compiladores paralelizantes na exploração do hardware disponível.

Vimos que em geral os computadores paralelos atuais estão utilizando os mesmos microprocessadores que são utilizados nos computadores pessoais e estações de trabalho e a construção de computadores com milhares de processadores pode ser abastecida pela larga escala de produção destes componentes, resultando numa diminuição de custo, e da necessidade de componentes feitos sob medida, diminuindo assim o tempo necessário para a construção dos supercomputadores.

Quanto ao modelo de programação utilizado atualmente em computação paralela vimos que atualmente dois se destacam: programação com endereçamento compartilhado e passagem de mensagem. Também apresentamos os paradigmas de linguagem de programação utilizados em programação paralela, imperativa e funcional e discutimos nesse contexto de programação as dificuldades existentes para se desenvolver um programa paralelo eficiente e que a existência de um compilador que paralelizasse o programa automaticamente seria de muita importância.

3. DISTRIBUIÇÃO AUTOMÁTICA

A otimização da distribuição de dados nos sistemas de memória distribuída visa a redução da comunicação de dados que não estejam na memória local dos processadores, ou seja, os dados que devem ser transmitidos pelo sistema de interconexão entre os processadores. Esta situação ocorre se o código executado num processador precisa acessar um dado que não está na sua memória local. Ele então deve fazer uma chamada explícita a uma rotina de passagem de mensagens que passa o dado pelo subsistema de interconexão até ele chegar ao processador destino desejado.

Apesar de aparentemente simples, a operação de passagem de mensagens nos sistemas de memória distribuída apresenta o problema de ser demasiado lenta se comparada com o acesso a um dado na memória local [AND97, BOO97, CUL99, SIA97], ou seja, a latência para o acesso a um dado é muitas vezes proibitivamente elevada. Isto se deve em parte ao fato do acesso à memória remota via sistema de interconexão entre processadores ser muito mais lento que um acesso à memória local realizado via barramento. Também o fato da passagem de mensagens ser realizada via software acarreta uma latência adicional se comparado com o acesso direto via hardware da memória local (acesso via barramento).

Assim, nos sistemas de memória distribuída, o sucesso de uma aplicação vai além da existência de paralelismo e é fortemente dependente da localidade de dados e computações do programa. Essa localidade pode ser obtida basicamente de duas formas [GAR95]: através de indicações do usuário, ou automaticamente via análises de compiladores ou outras ferramentas. Ambas as formas fornecem informações que permitem a distribuição de dados e computações entre os processadores de forma que o acesso aos dados não inviabilize a execução em paralelo.

A decisão automatizada da decomposição de dados de um programa pode ser dividida em três partes principais: 1) alinhamento, 2) divisão e 3) mapeamento. A etapa 1 do alinhamento, que é independente de arquitetura, tenta relacionar as dimensões dos diferentes *arrays* de acordo com o melhor padrão de acesso entre eles, buscando a redução da sobrecarga total envolvida em possíveis movimentações de dados entre os processadores (acessos a dados não locais). Realizado o alinhamento, na etapa 2 seguinte, a divisão, são tomadas as decisões de quais dimensões alinhadas serão distribuídas no espaço

de processadores virtual[4], ou seja, como serão associados os processadores virtuais a cada dimensão distribuída e a forma de distribuição desses *arrays*. Uma distribuição ótima visa a maximização do paralelismo potencial, balanceia a carga de computação e oferece a possibilidade adicional da redução do movimento de dados. Na etapa 3, o mapeamento, são realizadas as associações entre processadores virtuais e processadores reais segundo relações entre os custos de comunicação e custos de computação. Essa fase é dependente do sistema utilizado na execução do código paralelo e do compilador utilizado.

Nas seções seguintes descrevemos o alinhamento de dados e a distribuição de dados. Antes, porém descreveremos a otimização conhecida como balanceamento de carga estático que visa a distribuição eficiente da computação aos processadores evitando assim sobrecargas que venham a inviabilizar a execução em paralelo de uma aplicação.

3.1. TRANSFORMAÇÕES DOS DADOS

Como dito anteriormente, para alcançar o objetivo de um aproveitamento ótimo de uma arquitetura paralela, além da existência do paralelismo em uma aplicação, temos que saber como explorar este paralelismo. Normalmente, em aplicações científicas, as maiores fontes de paralelismo são os *loops* cujo desempenho dependerá tanto da organização do código como da organização dos dados.

Assim, tanto em linguagens imperativas como funcionais, na primeira etapa para a paralelização automática realizamos a exposição ou extração de paralelismo no programa analisado. Com linguagens imperativas, geralmente são aplicadas diversas técnicas [BAC93, KUL93, WOL92, WOL96] que permitem a análise e transformação do código para a exposição do paralelismo existente. Em linguagens funcionais, onde temos um paralelismo implícito, este pode ser extraído por um compilador sem a necessidade de técnicas elaboradas [LOI98].

Exposto o paralelismo, o passo seguinte trata da utilização eficiente deste paralelismo, ou seja, como serão mapeados dados e computações no sistema destino desejado. Este procedimento, conhecido como alinhamento [AND97, KUL93] determina que porções de dois *arrays* serão colocadas no mesmo processador para uma divisão de

[4] O espaço de processadores definido é virtual e assim é comum que se assuma uma dimensão que seja adequada para as análises realizadas. [AND97]

dados particular. Se eles estão alinhados de modo que ambas as porções estejam envolvidas na mesma computação, então teremos como resultado um número de acessos não locais reduzido no caso de uma distribuição dos dados entre diferentes processadores, ou seja, o alinhamento está relacionado com a alocação relativa de *arrays* de modo a maximizar os acessos locais.

3.1.1. Balanceamento de carga estático

O balanceamento de carga estático [OBO92], como citado anteriormente, é uma transformação que visa o mapeamento eficiente da computação aos processadores e será descrito devido à sua importância, tanto quanto às demais técnicas para a paralelização de programas utilizadas neste trabalho.

Basicamente, o balanceamento de carga busca a melhor opção de distribuição da carga de computação entre os processadores disponíveis para um dado programa paralelo. Como o balanceamento perfeito em geral não é possível, o que se busca é minimizar a diferença entre o elemento de maior carga de computação e a média das carga distribuídas entre os processadores, ou seja, o balanceamento busca encontrar uma função

$$f : q \mapsto x$$

onde q representa os blocos de computação e x representa o conjunto de processadores. A função que encontra o balanceamento perfeito pode ser representada por

$$|q|_x = |q|_y \qquad \forall x, y \in 1,\ldots,p$$

onde $|q|_x$ é o número de elementos no bloco q do processador x e p é o número de processadores.

Entretanto, como dito anteriormente o balanceamento perfeito não é possível em muitos casos. A função que obtenha o melhor balanceamento possível pode ser representada por

$$\textit{mínimo}\left(\underset{x=1,\ldots,p}{\textit{máximo}}\left(|q|_x\right)-\frac{\sum_{x=1}^{p}|q|_x}{p}\right)$$

Essa função de balanceamento procurada consiste basicamente de transformações no espaço de iteração dos comandos de repetição. Essas transformações podem ser a reestruturação ou a reordenação dos iteradores e a intercalação dos valores do iterador [OBO92, WOL92]. Outra transformação que busca o balanceamento da distribuição da carga de computação é conhecida como *tiling* [WOL92]. Não detalharemos além deste ponto o balanceamento de carga devido à sua abrangência, o que poderia ser desenvolvido em trabalhos futuros.

3.1.2. Alinhamento

A idéia básica do alinhamento de dados é a procura por padrões de referências no corpo do *loop*, ou seja, o alinhamento está relacionado com a orientação relativa dos *array* segundo um espaço de índices comuns, visando melhorar a localidade dos dados. Assim, se dois *arrays* acessados num mesmo comando estiverem alinhados, eles sempre serão colocados no mesmo processador, independente de como o dado for dividido.

O exemplo da **Figura 12** ilustra como seria o alinhamento dos dados de um *array* `b` a um *array* `a`, no corpo de um comando `for` imperativo. Se considerarmos que cada elemento dos *arrays* `a` e `b` seja colocado num processador diferente, o alinhamento determinará que elementos de `a` e `b` deverão ficar no mesmo processador, para que os acessos remotos sejam minimizados.

Também podemos observar, na **Figura 12** abaixo, que nas iterações do *loop* para cada `i`-ésimo elemento de `a` é realizada uma referência para o `(i+1)`-ésimo elemento de `b`, ou `a(i) = b(i+1)`. Assim, se o par `a(i)` e `b(i+1)` for armazenado no mesmo processador, então não seria necessário nenhum acesso não-local. Neste caso se `b` for alocado com deslocamento de uma posição à esquerda em relação a `a` então não existiriam problemas na divisão do iterador `i`, pois `a` e `b` estariam sempre no mesmo processador.

```
for i := 1 to 4
   for j := 1 to 5
      a(i,j) = b(i+1,j);
```

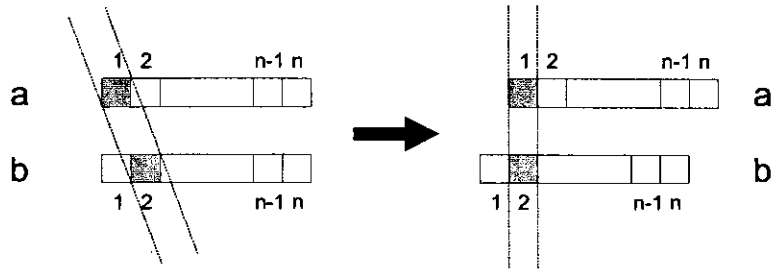


Figura 12. Idéia básica do alinhamento de dados

Portanto, analisando as ocorrências dos iteradores nas referências aos *arrays*, podemos determinar o exato alinhamento requerido [OBO92]. Para tanto, devemos encontrar uma função de alinhamento, que poderia ser chamada de *f*, por exemplo. Assim, para o exemplo acima, no alinhamento de `b` com `a`, seria necessário que encontrássemos uma função *f* que deslocasse `b` o quanto necessário

$$f: \texttt{b(i+1)} \mapsto \texttt{b'(i)}$$

ou

$$f: \texttt{b(i)} \mapsto \texttt{b'(i-1)}$$

Apresentaremos a seguir uma notação matemática, da álgebra linear [BOL80], para expressar a função acima citada, ou seja, apresentaremos uma representação de referência formal a um *array*. Para um *loop* de dimensão *m* e um *array* A de dimensão N_A, ela é dada por

$$F_A(\vec{a}) = R.\vec{a} + \mathrm{r}$$

onde $R_{N_A \times m}$ é uma matriz de inteiros, chamada matriz ocorrência, $r_{N_A \times 1}$ é um vetor de inteiros, chamado vetor de ocorrência e $(\vec{a})_{m \times 1}$ é o espaço iteração do *loop*.

Na **Figura 13**, a seguir apresentamos uma ilustração de um exemplo utilizando a notação apresentada anteriormente.

Agora, suponhamos dois *arrays*, A com dimensão N_A e B com dimensão N_B, com referências $F_A(\vec{a}) = R.\vec{a} + c$ e $F_B(\vec{a}) = U.\vec{a} + u$ respectivamente e dentro de um espaço de iteração de *loop*, V, de dimensão m. Para encontrarmos os alinhamentos entre A e B, utilizando a notação acima necessitamos encontrar o hiperplano S, onde as referências a A e B estão alinhadas. Um hiperplano é dado por $S = \{\vec{a} \in V \mid F_A(\vec{a}) = F_B(\vec{a})\}$ ou seja um alinhamento entre os elementos do *array* A com os elementos do *array* B ocorre em $\vec{a}$ se $R\vec{a} + r$ for igual a $U\vec{a} + u$ e N_A igual a N_B.

```
for i = 1, n
   for j = 2, n
      a(i,j) = b(j-1, i+2).
```

$$F_A(\vec{a}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad F_B(\vec{a}) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

Figura 13. Representação do acesso aos *arrays* a e b no *loop*

A otimização que busca o alinhamento dos dados pode então ser definida como a transformação $T: U\vec{a} + u \mapsto \alpha U\vec{a} + \alpha u + t$ para a qual $S' = \{\vec{a} \in V \mid F_A(\vec{a}) = T F_B(\vec{a})\}$ é máximo.

Assim para o exemplo da **Figura 13** cujas referências de A e B são

$$F_A(\vec{a}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad F_B(\vec{a}) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

respectivamente a transformação para o alinhamento dada pela função $T: U\vec{a} + u \mapsto \alpha U\vec{a} + \alpha u + t$ ficaria

$$\alpha = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad e \qquad t = \begin{bmatrix} -2 \\ 1 \end{bmatrix}$$

A **Figura 14** exibe uma ilustração de como seria o alinhamento da matriz A e B, supondo ambas com dimensão 10x10, para o exemplo da **Figura 13.**

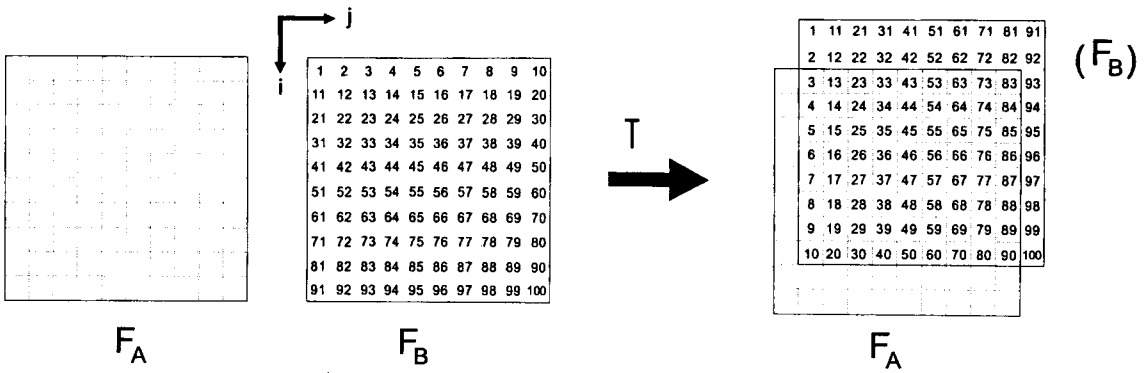


Figura 14. Alinhamento das matrizes da Figura 13.

3.1.3. Divisão e Mapeamento

Após a determinação do alinhamento dos *arrays*, é realizada então a divisão do espaço de dados no espaço de processadores virtuais. A divisão pode ser definida como a fase onde um sistema abstrato de processadores e todos os *array* a ele alinhados são mapeados no espaço real de processadores por algum modelo de distribuição. Os modelos mais comumente usados são a distribuição por linhas ou colunas, distribuição cíclica, distribuição em bloco e distribuição bloco-cíclica, linha-cíclica ou coluna-cíclica [AND97].

Lembrando apenas que, se dois *arrays* estão alinhados a índices particulares, então não existirão problemas de como os elementos individuais do *array* serão divididos, pois quaisquer referências usando esses índices serão locais. Como os índices não alinhados não possuem essa propriedade, eles deveriam ser evitados. Se um *array* tem N dimensões e digamos *k* delas estão alinhadas, então essas *k* dimensões do *array* poderão ser divididas e *N-k* deverão ser serializadas. A **Figura 15** exibe alguns exemplos de possíveis modelos de distribuições de dados para uma matriz qualquer de duas dimensões.

Uma função de divisão de dados pode ser definida como $f_p : \mathrm{R} \mapsto \overline{\mathrm{R}}$ onde $\mathrm{R}_{\mathrm{N_A} \times \mathrm{m}}$ é matriz de ocorrência e $\overline{\mathrm{R}}_{\mathrm{N_A} \times \mathrm{m}}$ é a matriz de ocorrência dividida. De modo oposto, a função de serialização, que descreve os índices que não serão divididos, pode ser definida como uma função $f_s : \mathrm{R} \mapsto \underline{\mathrm{R}}$ onde $\mathrm{R}_{\mathrm{N_A} \times \mathrm{m}}$ é matriz de ocorrência e $\underline{\mathrm{R}}_{\mathrm{N_A} \times \mathrm{m}}$ é a matriz de ocorrência serializada e $f_p(\mathrm{R}) + f_s(\mathrm{R}) = \mathrm{R}$. Na realidade, f_p pode ser definida como uma matriz $P_{N_A \times \mathrm{m}}$ cujas linhas vêm da matriz identidade ou da matriz nula e da mesma forma,

f_s como uma matriz $S_{N_A \times m}$ cujas linhas também vêm da matriz identidade ou da matriz nula; então, $P \times \mathrm{R} = \overline{\mathrm{R}}$ e $S \times \mathrm{R} = \underline{\mathrm{R}}$.

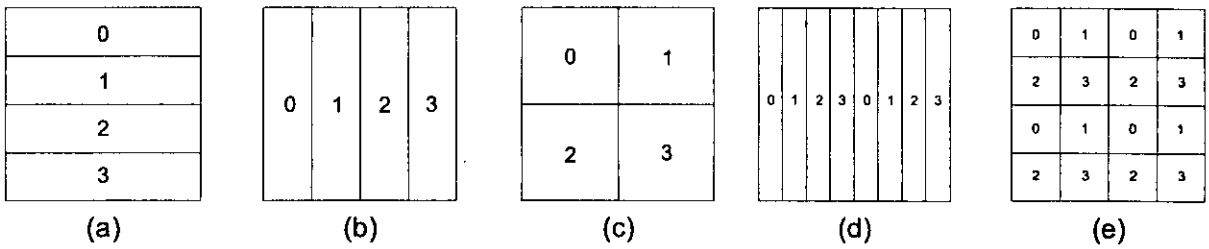


Figura 15. **(a) Distribuição por linhas (b) Distribuição por colunas (c) Distribuição por blocos (d) Distribuição coluna-ciclica (e) Distribuição bloco-cíclica**

Portanto, para selecionar quais as dimensões da matriz ocorrência que serão distribuídas basta colocar na matriz de divisão a linha da matriz identidade equivalente da matriz ocorrência. E, para as dimensões que não serão divididas, coloca-se a linha equivalente da matriz nula. É importante também estender a divisão para o caso onde é desejado que seja mantido o alinhamento dos dados a serem distribuídos. Assim, suponha que dois *arrays* tenham as seguinte matrizes de ocorrência, $R_{N_A \times m}$ e $\mathrm{U}_{\mathrm{N_A \times m}}$. A matriz de divisão $\mathrm{P}_{\mathrm{N_A \times m}}$ será construída da seguinte forma: uma dada linha i de P será a i-ésima linha da matriz identidade se e somente se a i-ésima linha da matriz R e a i-ésima linha da matriz U forem iguais; caso contrário, ela será uma linha da matriz nula.

Assim suponha a função de ocorrência para a matriz A da **Figura 13**. Para distribuir a matriz A, da **Figura 13**, por linhas ou por colunas construiríamos as seguintes matrizes $\mathrm{P}_{\mathrm{N_A \times m}}$ de divisão

$$\mathrm{P}_{\mathrm{N_A \times m}} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \qquad \text{para a distribuição das linhas}$$

e

$$P_{N_A \times m} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \qquad \text{para distribuição das colunas}$$

Escolhida a forma de divisão, o passo seguinte é a realização do cálculo dos novos índices das matrizes distribuídos pelo espaço de processadores virtual. Para ilustrar este passo apresentamos a seguir, na **Figura 16**, a divisão de uma matriz bidimensional num conjunto de *p* processadores, com a distribuição por linhas.

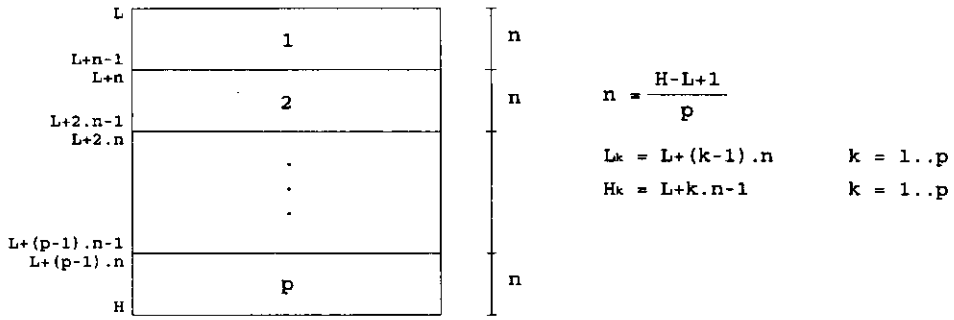


Figura 16. Divisão por linhas de uma matriz em p processadores

Temos na **Figura 16** acima que L é o limite inferior da matriz, H o limite superior e n o número de linhas da matriz a serem distribuídas em cada processador. Na fórmula para cálculo de n estamos considerando o caso onde a divisão por p processadores é exata e assim todos eles receberão o mesmo número de elementos, n. Para o cálculo dos mapeamentos de índices L_k e H_k, da matriz temos envolvidos L e H, p e o número do processador que receberá os pares de índices.

Para a distribuição por colunas podemos utilizar as mesmas fórmulas para o cálculo dos índices, entretanto para uma distribuição por blocos como na **Figura 17** abaixo, elas devem ser reescritas levando-se sem consideração a disposição dos processadores que podem estar organizados na forma de uma grade quadrada.

Além da divisão dos dados aos processadores, também é necessário realizar a análise para o mapeamento das iterações do *loop* nos processadores, ou seja, devemos encontrar o espaço de iteração local para cada processador onde todas as escritas são locais. O espaço de iteração é dividido de uma maneira similar ao espaço de índices do *array*, ou seja $f:\vec{a} \mapsto \vec{a}_1 \times \vec{a}_2 \times \cdots \times \vec{a}_p$. Uma abordagem mais abrangente sobre este assunto e os anteriores será realizada no capítulo seguinte, onde discutiremos os métodos de alinhamento, divisão e mapeamento utilizados no trabalho.

Após a escolha das dimensões que serão distribuídas, a etapa seguinte é a realização do mapeamento de dados e computação nos processadores físicos existentes, ou

seja, encontrar uma função que mapeie os índices dos *arrays* e *loops* nos processadores disponíveis.

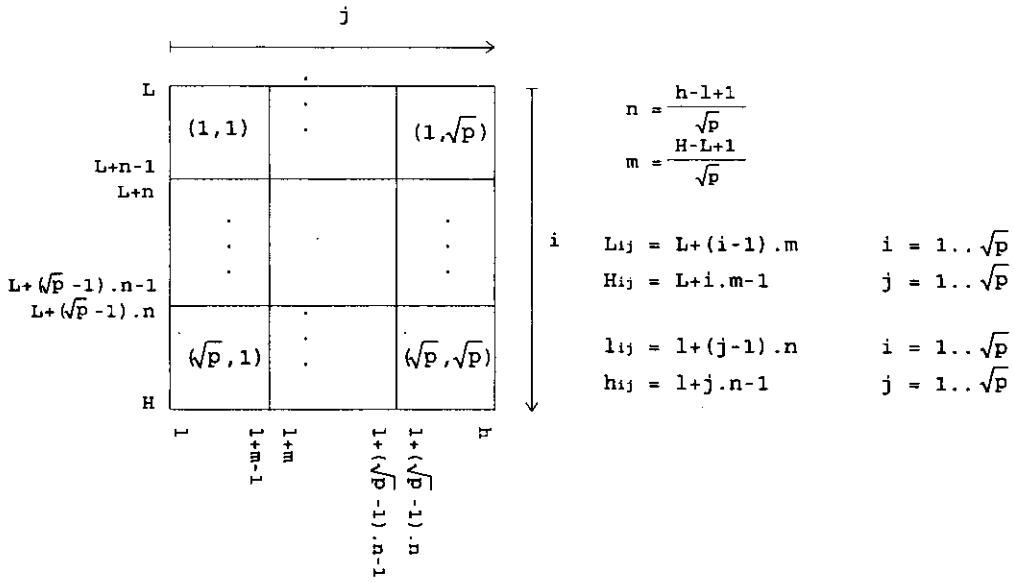


Figura 17. Mapeamento por blocos de uma matriz numa grade de $\sqrt{p} \times \sqrt{p}$ processadores

O mapeamento trata então da forma como os dados e iterações divididas no espaço de processadores virtual serão associados aos processadores disponíveis. A escolha do melhor mapeamento deve considerar o sistema que esta sendo utilizado, ou seja, o hardware utilizado, sistema operacional e compilador, o que deixa claro que esta fase é dependente do sistema escolhido.

Como esta fase depende do sistema utilizado, ela necessita então de funções que calculem o custo de diversas combinações de mapeamento para que aquele que seja ótimo seja escolhido. Como o número de possibilidades de mapeamento é extenso [GAR97, GUP92a, KRE95] as soluções utilizadas em geral são baseadas em algoritmos heurísticos sobre representações de grafos ou programação linear inteira 0-1. Na seção seguinte apresentaremos algumas implementações de alinhamento, divisão e mapeamento encontrados na literatura.

3.2. TRABALHOS RELACIONADOS

Na literatura existe um grande número de trabalhos que tratam do problema da distribuição automática de dados no contexto das aplicações regulares, diferindo no tipo de estrutura escolhido para representar o problema, o modelo de estimativa de performance e

as técnicas aplicadas para encontrar uma solução. No caso da divisão estática, a maioria das abordagens divide o problema em duas partes: alinhamento e distribuição. As principais características de alguns sistemas serão descritos a seguir.

I. Salazar e Böhm

O trabalho apresentado por Salazar e Böhm [SAL97] é o resultado da proposta e implementação de análises para a paralelização de programas em sistemas de memória distribuída, num protótipo do compilador OSC nomeado D-OSC que resultam em programas com chamadas para rotinas MPI otimizadas por análises visando reduzir a comunicação.

O compilador D-OSC foi implementado em quatro fases, a saber: Base onde é realizada a distribuição por igual de *arrays* e *loops* aos processadores; *Retangular Arrays* que substitui quando possível os *arrays* de dimensão mais alta de *SISAL* por *arrays* bidimensionais, considerados mais adequados à distribuição por linhas, blocos ou colunas; *Block Messages* que tenta otimizar a leitura/escrita de elementos dentro de *loops* através da colocação destas mensagens dentro de um mesmo processador; e *Multiple Alignment* que visa reduzir a comunicação e assim melhorar a performance do programas.

Parte do trabalho, mais especificamente o alinhamento, é uma extensão do trabalho de Ramanujam e Sadayappan [RAM91] para encontrar uma divisão para a distribuição dos *arrays* que não exija a comunicação entre processadores mesmo em sistemas onde seja necessário analisar alinhamentos múltiplos. Os trabalhos diferem no sentido que Salazar trata o problema de alinhamentos conflitantes e Ramanujam e Sadayappan consideram apenas os alinhamentos não conflitantes. Não entraremos em maiores detalhes, pois mais adiante apresentamos uma descrição do trabalho de Ramanujam e Sadayappan.

A distribuição dos dados é realizada através da técnica de divisão conhecida como *snake* [SAL95], uma derivação da técnica de grafo de reuso de dados [SAL95] para a escolha da forma dos dados a distribuir. A ação principal de um grafo de reuso de dados será caracterizar a ação de reuso para qualquer elemento de um *array* sendo criado e a técnica *snake* selecionará qual a melhor divisão dos dados.

Com relação à implementação no compilador OSC, as análises são realizadas sobre código IF2, após o módulo IF2PART (que realiza a paralelização de *loops*), mais precisamente por um módulo nomeado IF2DIST que é o encarregado de analisar e transformar os códigos para as necessidades de sistema de memória distribuída.

II. Anderson e Lam

O trabalho apresentado por Anderson e Lam [AND97] que faz parte do projeto de compilador paralelizante, SUIF, trata do mapeamento automático de dados e computações de códigos de aplicações científicas contendo *loops* com matrizes densas. A representação dos limites dos *loops* e subscritos dos *arrays* é feita através de funções afins dos índices dos *loops*. Os *loops* analisados estão representados na forma canônica de aninhamentos de *loops* completamente permutáveis [WOL96], assim transformados pelas técnicas apresentadas por Wolfe e Lam [WOL96] para a melhoria de localidade e paralelismo em *loops* alinhados.

O foco do trabalho, descobrir um mapeamento de dados e computações eficiente sobre um espaço de processadores virtual, foi formulado utilizando uma estrutura baseada em álgebra linear, mais especificamente um sistema de equações das condições que devem ser satisfeitas nas decomposições de computações e dados. É também apresentada uma forma de resolver sistematicamente essas equações de decomposições.

A formulação matemática apresentada modela o problema da decomposição de dados e computações livres de comunicação tanto para um aninhamento de *loop* único ou grupos de aninhamentos de *loops*. As restrições das decomposições de dados e computação são expressas através de requerimentos para o núcleo das funções lineares que representam os mapeamentos de dados e computações. A computação do núcleo, baseada num método iterativo que compara as restrições mútuas dos mapeamentos de dados e computação, resulta no conjunto de todas as decomposições livres de comunicação.

O cálculo de decomposições afins é realizado sobre um espaço de processadores virtual e é realizado em duas fases, uma para solução das decomposições lineares e outra para a solução das decomposições de *offset*. Para o cálculo das decomposições lineares primeiro é fixado um sistema de equações que representa as condições necessárias que uma decomposição linear básica válida deve satisfazer e então esse sistema é solucionado primeiro pelo cálculo de núcleos das matrizes que representam as decomposições lineares e então pelo cálculo das matrizes específicas em passo separado. Através do núcleo que representa os dados e as computações que devem ser associadas ao mesmo processador, também é determinado o grau de paralelismo na computação.

Formas restritas de comunicação (replicação e *multicast*) e sincronizações regulares dentro dos aninhamentos de *loops* permutáveis completamente também são tratadas.

O método de Anderson e Lam também trata da distribuição dinâmica de dados, utilizando para isso um grafo não direcional com valores de custo (peso) para vértices e arcos. No grafo, os vértices correspondem a aninhamentos de *loops* e os arcos representam possíveis remapeamentos de dados entre os aninhamentos de *loops*. O custo de um vértice é dado pelo máximo de uma lista de *k* pesos, representando o tempo de execução do aninhamento de *loop* quando cada um dos *k loops* é distribuído. Os pesos dos arcos são a combinação do pior caso de custo de comunicação com a probabilidade de ocorrência dos remapeamentos. Além de formular uma representação do problema de layout dinâmico de dados e divisão, também provaram que o problema da decomposição dinâmica é NP-completo [AND97].

O algoritmo para a decomposição dinâmica de dados utiliza heurística para encontrar as soluções. A abordagem utilizada, *greedy*, tenta primeiro eliminar a maior quantidade possível de potenciais comunicações das partes do programa mais freqüentemente executadas. Os arcos são visitados pela ordem decrescente de pesos, num modelo *bottom-up*, iniciando com o *loop* mais interno. Dois vértices de *loops* candidatos no nível corrente são fundidos num único componente se o desempenho dos nós unidos for maior que o desempenho dos vértices individualmente, incluindo os custos de remapeamentos. Uma vez que dois nós tenham sido unidos num certo nível, eles são considerados um único componente para todos os níveis subseqüentes.

Após o algoritmo *greedy* terminar, o grafo inteiro é dividido em componentes, e mapeamentos de dados e computações para cada componente terão sido determinadas. Visto que os mapeamentos de dados e computações somente especificam que elemento do *array* e iteração são locais a um único processador, um único passo é necessário para determinar o processador real no qual a computação e o dado serão mapeados. Este último passo é referido como computação de orientação e deslocamento e um algoritmo *greedy* é utilizado para combinar tão fielmente quanto possível a orientação entre os componentes.

III. O'Boyle e Gurd

A proposta de divisão automática de dados de O'Boyle e Gurd [OBO92] trata dos fatores balanceamento de carga, alinhamento, divisão e *pre-fetching* de *arrays*, na compilação de programas SISAL, com sintaxe reduzida [OBO92], para sistemas de memória distribuída. Para as análises e transformações, baseadas na álgebra linear, eles desenvolveram uma seqüência de transformações que expressam os quatro fatores, acima citados. A prova dos métodos propostos consistiu do estudo de vários problemas bem

conhecidos, por exemplo, multiplicação de matrizes simétricas, implementados como programas SISAL.

O algoritmo para balanceamento de carga proposto trata da questão de mapeamento eficiente de computações de *arrays* nos processadores, pois considera que a maior sobrecarga associada com o mapeamento de computações é o desbalanceamento de carga e assim tenta encontrar mapeamentos para os quais esta sobrecarga seja minimizada. O método de divisão dos iteradores apresentado trata o espaço de iterações como um espaço poliedral, representando os limites do *loop* e condicionais. Eles formulam a transformação para o balanceamento de carga como a divisão do iterador que não faz referência a qualquer outro iterador nos limites de seu *loop* ou não é referenciado por qualquer outro e expressam-no matematicamente como uma condição de invariância do espaço de iteração. Também são apresentadas transformações auxiliares para reordenação e intercalação dos iteradores.

O alinhamento relativo de *arrays* é descrito como a uma região poliedral formada da interseção de hiperplanos de um sistema de equações com uma região factível de computação, ou seja, o alinhamento é descrito em termos das ocorrências dos *arrays* (matrizes ocorrência expressando os subscritos dos *arrays*) e regiões factíveis das computações. A função usada para determinar o alinhamento dos *arrays* é chamada de função êxito, que essencialmente compara cada subscrito do *array* e retorna um valor que é o número de linhas que são iguais. Após a identificação de alinhamentos, são buscadas funções de alinhamentos válidas para as quais a função êxito seja máxima e ocorre a propagação do alinhamento pelo programa inteiro. Uma vez que o alinhamento relativo tenha sido determinado, os *arrays* são então mapeados ao espaço de processadores, onde somente partições ortogonais são consideradas. A divisão de um *array* em k dimensões consiste na criação de uma matriz de k linhas, onde cada uma das k linhas corresponde às dimensões do *array* que serão distribuídas e são obtidas da matriz ocorrência.

O método desenvolvido para o mapeamento de dados e computação a processadores distribui igualmente os dados entre os processadores, mantendo o alinhamento dos *arrays* de tamanhos diferentes. A distribuição dos dados é elaborada em função do número de processadores, do domínio dos índices dos *arrays*, do número de ciclos na distribuição do *array* e da quantidade de elementos de dados contínuos, permitindo distribuições em colunas, linhas, blocos e cíclicas. O passo seguinte determina o espaço de iteração local de cada processador, buscando determinar todas as escritas locais. O método apresentado para a partição do espaço de iteração é similar àquele do

espaço de índices. Ele é formulado em função do número de processadores, espaço poliedral das iterações e as matrizes de ocorrência do *array*. E finalmente, após a divisão ter sido realizada, são aplicadas então transformações de *prefetching* (baseadas em *loop interchange* [WOL92]) visando a redução de acessos não-locais. A escolha para a realização ou não do pre-fetching é dada por condições de invariância.

As técnicas apresentadas no trabalho foram tratadas separadamente e entre elas existem conflitos das funcionalidades, e visando saná-los, O'Boyle e Gurd, desenvolveram uma solução para a utilização de seu métodos baseada em heurística. Nesta heurística o balanceamento de carga é considerado mais importante que os acessos não-locais. O alinhamento de dados é requerido antes da divisão de dados e é considerado um processo complementar. A divisão de dados é restrita a fim de permitir uma distribuição de dados homogênea e o *prefetching* é realizado seguindo condições para que não viole restrições de escalabilidade.

IV. Couvertier e Ramanujam

O foco da pesquisa de Couvertier e Ramanujam [COU96] é a otimização da comunicação entre processadores nos computadores de memória distribuída obtida através da automatização do alinhamento e distribuição de dados e computação e aplicação de transformações que permitam o uso de vetorização sempre que possível.

Eles tratam o problema da divisão de dados dividindo-o em duas fases, alinhamento e distribuição, para as quais são apresentadas diversas abordagens baseadas em programação linear inteira 0-1 e heurística. O foco da análise são os *loops* com os elementos *array* à esquerda do comando de atribuição. Em oposição à maioria das pesquisas, a regra *owner-computes* foi aplicada de uma forma menos rigorosa.

Para a fase de alinhamento, eles apresentam três métodos baseados em restrições. O primeiro deles trata o problema como um problema geral de programação linear e o soluciona como tal. Neste caso, o problema é expresso como um conjunto de inequações baseadas em restrições sobre a primeira norma da distância entre os processadores virtuais com os quais os modelos são associados. O segundo método usa o método do Multiplicador de Lagrange [COU96] e o software Matemática. No terceiro método utilizam uma heurística para decidir quais das restrições serão deixadas sem solução, decisões essas tomadas baseadas nas informações armazenadas numa tabela que chamam de *Resource Information Table* (RIT).

Para a fase da distribuição, são apresentados dois métodos para integrar a colocação de computação com mapeamento de dados, no escopo da distribuição unidimensional do *array*. O primeiro método, que utiliza transformações unimodulares, tenta encontrar a melhor combinação de mapeamento de dados e computação que resultam numa sobrecarga de comunicação baixa através da vetorização de mensagens. Também faz uso da transformação *tiling* para distribuir as iterações em pedaços para depois vetorizar as mensagens. A segunda abordagem para a distribuição, baseada em grafo de preferência (variação do grafo de afinidade de componente), visa integrar as transformações de reestruturação de *loop* e mapeamentos de dados.

V. Garcia e Ayguadé

Garcia e Ayguadé [GAR97] apresentam uma abordagem automática para o mapeamento de dados em sistemas maciçamente paralelos, considerando as aplicações com computações regulares que envolvem *arrays*. A ferramenta por eles implementada, PDDT [GAR97], analisa programas FORTRAN 77 e decide uma estratégia de mapeamento e paralelização para esses programas utilizando para isso diretivas de mapeamento e paralelização de HPF. As estratégias de mapeamento podem ser estáticas ou dinâmicas em uma ou duas dimensões e com um modelo de distribuição em blocos ou cíclicas. A solução sugerida leva em conta os efeitos do fluxo de controle dos comandos entre as fases e inclui o alinhamento e a distribuição de todos os *arrays* de cada fase, um conjunto de ações de remapeamentos entre as fases quando vantajoso e uma estratégia de paralelização de *loop* induzida pelo mapeamento de dados.

Eles definem um modelo baseado em restrições sobre uma estrutura chamada CPG (*Communication-Parallelism Graph*), que mantém as informações de todos os movimentos de dados e de paralelismo inerente em cada fase do programa e informações adicionais entre as fases denotando o custo do movimento de dados ocorrido se a distribuição de um mesmo *array* for diferente entre as fases. Os nós das CPG, organizados em colunas, representam a dimensão distribuída de um *array*, e os arcos representam as restrições de comunicação e paralelização. As restrições são formuladas e solucionadas usando técnicas de programação inteira 0-1 linear, buscando uma solução ótima sem a utilização de algoritmos heurísticos.

A solução derivada inclui o alinhamento e a distribuição de cada *array* em cada fase, ações de remapeamentos entre as fases se elas forem vantajosas e o conjunto de *loops* que devem ser paralelizados segundo a estratégia de mapeamento de dados.

VI. Gupta e Banerjee

As técnicas de divisão automática de dados desenvolvidas por Banerjee e Gupta [GUP92a] foram implementadas no compilador PARADIGM [GUP92a, BAN95], que analisa código fonte FORTRAN 77 e gera programas SPMD com comunicações explícitas.

As análises realizadas partem do pressuposto que qualquer estratégia de divisão necessita de um mecanismo para a realização de análises comparando diferentes esquemas de divisão de dados para um mesmo programa, e para tanto necessitam estimar o desempenho do programa paralelo destino. O critério utilizado para a seleção são as chamadas medidas de qualidade [GUP92a]; através delas, o compilador tenta combinar restrições para cada *array* de uma maneira consistente de forma a minimizar o tempo de execução total.

Os focos das análises são as referências a *arrays* em comandos dentro de *loops*, tendo como escopo o programa inteiro, procurando assim o que consideram balanceamento de carga e localidade dos dados ótimos. As análises de alinhamento e distribuição de dados são feitas através de uma abordagem baseada em restrições, às quais estão associadas medidas de qualidade. A medida de qualidade é uma função representando o custo para o caso em que os *arrays* não estão alinhados. As restrições que refletem a distribuição dos *arrays* alinhados têm parametrizadas as funções de custo do tempo de execução que representam as medidas de qualidade. Os parâmetros incluem o tamanho do problema, o número de processadores e os esquemas de distribuição usados em cada dimensão. As técnicas automáticas manipulam distribuições cíclicas, bloco e bloco-cíclica, sendo considerada também a replicação de *arrays* considerados pequenos e escalares.

A partir de informações do tamanho do problema e da máquina, o sistema seleciona então, em tempo de compilação, um esquema de decomposição que distribua o *array* através de um número ótimo de processadores da grade bidimensional.

Para a escolha de um esquema de alinhamento utilizam a estrutura CAG (*Component Affinity Graph*) [GUP92a apud 48] e métodos heurísticos de busca. O custo de comunicação para cada comando com referência a *array* é expresso como uma função do tamanho da máquina, número de processadores em cada dimensão e o método de divisão

utilizado. As funções tentam refletir os efeitos das transformações dos *loop* e otimizações de comunicação, tais como a vetorização e a agregação, nos custos de comunicação de comandos simples.

Os métodos apresentados por Gupta e Banerjee são voltados à análise de problemas de aplicações científicas, com computação regular, não sendo adequados para análise de computações irregulares, tais como matrizes esparsas e grades com estruturas irregulares. Também não realizam a análise inter-procedural; um único esquema de decomposição estática é derivado para o programa inteiro, sem suporte para realinhamento ou redistribuição dinâmicos.

VII. Kremer e Kennedy

O trabalho desenvolvido por Kremer e Kennedy [KRE95] discute o projeto e a implementação de uma ferramenta de seleção de disposições de dados que gera automaticamente especificações no estilo de Fortran D e HPF para problemas de computações regulares. O sistema otimiza a disposição dos dados baseando-se no tamanho de problema e no número de processadores disponíveis. Além de análises estáticas da disposição dos dados, também tratam do problema de remapeamentos dinâmico dos dados.

A estrutura proposta divide o problema de disposição automática de dados em quatro subproblemas bem definidos: divisão do programa, construção de espaços de busca de disposições candidatas, estimação de desempenho e seleção da disposição de dados, cujas soluções são obtidas com técnicas gerais de programação linear e programação inteira.

O passo de divisão divide o programa em segmentos de código, nomeados fases. Uma fase é o *loop* mais externo de um aninhamento de *loops* tal que o *loop* define uma variável de indução que ocorre em uma expressão subscrita de uma referência a *array* no corpo do *loop*. A estrutura utilizada para a representação das fases é uma extensão dos grafos de fluxo de controle apresentados por Aho, Sethi e Ullman [AHO86].

O segundo passo, dividido em dois estágios, alinhamento e distribuição constrói espaços de buscas de disposições de dados candidatas para cada fase, gerados com base em expectativas de performance. A representação central para o problema de alinhamento interdimensional é o grafo de afinidade de componente (CAG), não direcionado e com peso, introduzido por Li e Chen [apud KRE95]. Realizadas em dois estágios, primeiro as análises consideram somente preferências de alinhamentos interdimensionais entre *array* e

num segundo estágio cada *array* é mapeado num único *template* de programa de modo que as preferências de alinhamento são respeitadas. Ele é tratado como um problema de programação linear inteira 0-1, cuja solução maximiza uma função afim construída em função dos pesos dos arcos do grafo.

A análise de distribuição, realizada após a análise de alinhamento ou mapeia cada dimensão do *template* por bloco ou ciclos na arquitetura paralela ou replica ou torna local cada dimensão do *template*. A solução para o problema da distribuição é obtida de diferentes algoritmos heurísticos.

O terceiro passo avalia o tempo de execução esperado de cada disposição de dados candidata e verifica possíveis remapeamentos. Estas estimativas são obtidas comparando as distribuições candidatas. A avaliação de performance proposta é baseada em coleções de rotinas de *kernel* que medem o custo de vários padrões de comunicação e computação, do compilador que é usado para gerar o código executável para uma determinada arquitetura destino e das características da máquina destino.

O último passo, a seleção de disposição de dados é modelada como um problema de otimização sobre um grafo de disposição de dados (DLG) direcionado e com valores de peso. A seleção é tratada como um problema de programação inteira 0-1 com restrições lineares (disposição e remapeamentos) e resolvida como tal.

VIII. Outros trabalhos

- **Li e Chen**

No projeto do compilador para a linguagem funcional Crystal, Li e Chen [apud WOL94] tratam do problema da distribuição de dados para máquinas paralelas de memória distribuída, alinhamento interdimensional e mapeamento de dados estáticos. O problema de alinhamento interdimensional é modelado como um problema de grafo, CAG (*Component Affinity Graph*), a partir do qual são analisados os padrões de referência aos *arrays* de um programa, dados valores de peso. Eles provam que o problema de alinhamento é NP-completo [LC90 apud GAR97] e por isso utilizam-se de heurística para solucioná-lo. Após as análise de alinhamento o código funcional é convertido para um programa imperativo e nele são inseridas rotinas de comunicação para a distribuição das dimensões com índice comum a um padrão de distribuição pré-determinado.

- **Dierstein, Hayer e Rauber**

O trabalho de Dierstein, Hayer e Rauber [DIE96], parte do projeto para compilador paralelizante do sistema iPSC/860 da Intel, utiliza o método de *branch-and-bound* para solucionar o problema da distribuição e redistribuição automática de *arrays* na paralelização de programas na linguagem Pascal. Na distribuição de dados são realizadas análises de dependências utilizando grafo de referência de dados [DIE96] e utilizados valores de custo de comunicação. A escolha de qual a melhor distribuição envolve a construção de uma árvore de decisão onde cada nó representa uma distribuição do *array* no programa analisado. A construção dos níveis da árvore envolve a análise dos custos de distribuição, escolhidos aqueles que mais pesam no custo de comunicação, reduzindo assim o número de ramos na árvore. O estudo dos custos é realizado através de um grafo bipartido. Os valores de custo de comunicação são obtidos de uma ferramenta de análise de custo que usa vários parâmetros do sistema paralelo suporte como o tempo de *startup* para a transferência da mensagem e o tempo de transferência para um *byte* [DIE96].

- **Kulkarni e Stumm**

A técnica apresentada por Kulkarni e Stumm [KUL05], CDA (*Computation Decomposition and Alignment*) trata do problema de alinhamento de dados em dois passos, decomposição de computação e alinhamento de computação, aplicadas sobre *loops* com computações regulares. O passo da decomposição de computação primeiro decompõe corpos de *loops* em comandos individuais e então decompõe os comandos individuais em comandos de granularidade mais fina. Um comando é decomposto pela sua reescrita como uma seqüência de comandos menores que acumulam os resultados intermediários e produzem o mesmo resultado final. No alinhamento de computação, transformações lineares são aplicadas em todos os novos espaços de computação, formando o novo espaço de iteração.

- **Ramanujam e Sadayappan**

Ramanujam e Sadayappan [RAM91] apresentam uma técnica para o problema de alinhamento de dados aplicado a aninhamento de *loops* inteiramente paralelos individualmente. As partições de *arrays*, definidas por uma família de hiperplanos, paralelos são consideradas na busca por divisões livres de comunicação. Também tratam do caso da impossibilidade de divisão livre de comunicação através de minimização da

comunicação durante o balanceamento de carga. A técnica apresentada é independente de máquina e assume a regra *owner-computes*.

- **Sarkar e Vazquez**

A proposta apresentada por Sarkar e Vasquez [SAR94] discute a compilação para sistemas de memória distribuída utilizando técnicas de compilação herdadas de sistemas de memória compartilhada. O problema da distribuição dinâmica dos dados, em comandos *forall*, é reduzido à seleção de mapeamentos de forma a minimizar o custo da comunicação entre os processadores, cujas soluções são baseadas na estrutura do grafo de dependência de dados no nível dos *loops*. Para um único grafo de dependência, o custo mínimo é obtido pela simples soma dos termos independentes obtidos de custos mínimos nos nós filhos e com conjuntos de grafos de dependência de dados a solução é obtida por um algoritmo heurístico.

3.3. CONCLUSÃO

Apresentamos neste capítulo alguns dos passos envolvidos na compilação automática de programas para arquiteturas paralelas, mais especificamente a transformação de dados. Discutimos a distribuição dos dados em função de balanceamento de carga, 1) alinhamento, 2) divisão e 3) mapeamento de dados. A discussão apresentada tentou ser o mais abrangente possível, não se prendendo a algum trabalho em específico.

Em seguida discutimos brevemente alguns trabalhos sobre a transformação automática de dados disponíveis na literatura. Em geral os trabalhos envolviam apenas um dos passos envolvidos na distribuição automática de dados, sendo assim bastante ricos em detalhes em cada um dos assuntos.

4. DISTRIBUIÇÃO DE DADOS E COMPUTAÇÃO

Neste capítulo descreveremos os métodos utilizados nas análises estáticas de códigos *SISAL* para a otimização da distribuição de dados e computações. Esses métodos são semelhantes àqueles apresentados nos trabalhos de Anderson e Lam [AND97] e O'Boyle [OBO92] que utilizam Álgebra Linear e Programação Linear para a elaboração de técnicas para alinhamento e decomposição de dados e computações.

Como no trabalho de Anderson e Lam partimos do pressuposto que o código analisado encontra-se na forma canônica [AND97, WOL92] com *loops* inteiramente permutáveis e também como as análises realizadas nesse trabalho estão baseadas nas construções *forall*.

A seguir, apresentaremos algumas formulações e notações utilizadas no trabalho, sendo que em um primeiro momento veremos o alinhamento de *arrays* no corpo de um *loop forall* e em seguida análises para a distribuição conjunta de dados e computações do *loop forall*. Na seção 4.1 apresentaremos brevemente a técnica de alinhamento e divisão de dados utilizada, na seção 4.2 e 4.3 apresentaremos as análises de distribuição de dados e computações a processadores e na seção 4.4 discutiremos a aplicação das técnicas discutidas nas seções anteriores à linguagem *SISAL*.

4.1. ALINHAMENTO

Como visto anteriormente, em *SISAL* temos como uma das construções o *forall*, e esta foi à estrutura escolhida para aplicação de transformações e análises que possibilitem a distribuição automática de dados.

O alinhamento dos *arrays* determinará quais as porções de *arrays* que serão colocadas no mesmo processador, reduzindo a quantia de acessos não locais dos dados divididos entre os processadores. Utilizando a notação de funções afins da Álgebra Linear podemos formular o problema do alinhamento como a busca de hiperplanos de alinhamento [BOY93].

Assim, para nosso trabalho temos que o *forall* cercando um segmento de comandos qualquer será representado como o vetor de iteradores $\vec{i} = (i_1, i_2, \cdots, i_n)$ onde n é o número de iteradores ou *loops* aninhados definindo um espaço de iteração $\mathcal{I}$. Um *array* de m

dimensões define um espaço de *array* A e cada elemento é acessado por um vetor de índices $\vec{a} = (a_1, a_2, \cdots, a_m)$.

Iniciamos a discussão do alinhamento de dados utilizando uma função afim $f : \mathcal{I} \mapsto A$ para representar o acesso aos *arrays*, dada por $f(\vec{i}) = F\vec{i} + \vec{\zeta}$, onde F é uma matriz e $\vec{\zeta}$ é um vetor de constantes. Temos então que dado um conjunto de *arrays* o alinhamento no acesso a eles ocorre quando as funções afins dos *arrays* são iguais, ou seja, as partes matrizes e as partes vetores de constantes são iguais. No entanto, essa condição nem sempre ocorre naturalmente dentro de um código, sendo necessário à realização de uma transformação no acesso aos *arrays*. Para tanto vamos definir as funções

$$l_{xj}^{k}(\vec{i}) = L_{xj}^{k}\ \vec{i} + \vec{\zeta}_{xj}^{k} \tag{1}$$

que representará a *k-ésima* função de acesso ao *array* x (com m dimensões) num aninhamento de *loops* j e

$$r_{yj}^{q}(\vec{i}) = R_{yj}^{q}\ \vec{i} + \vec{\zeta}_{yj}^{q} \tag{2}$$

que representará a *q-ésima* função de acesso a um *array* y (com m dimensões) no mesmo aninhamento de *loops* j.

Podemos então formular o alinhamento de dados como o seguinte sistema onde buscamos uma transformação linear que iguale as duas funções de acesso a *array*, l e r:

$$L_{xj}^{k}\ \vec{i} + \vec{\zeta}_{xj}^{k} = T_{yj}^{q}\left[R_{yj}^{q}\ \vec{i} + \vec{\zeta}_{yj}^{q}\right] + \vec{t}_{yj}^{q}$$

ou seja, procuramos o par $(T_{yj}^{q}, \vec{t}_{yj}^{q})$. Para facilitar a observação podemos reescrever o sistema acima como

$$T_{yj}^{q} R_{yj}^{q} = L_{xj}^{k} \tag{3}$$

e

$$T_{yj}^{q} \vec{\zeta}_{yj}^{q} + \vec{t}_{yj}^{q} = \vec{\zeta}_{xj}^{k}$$

Onde podemos ver que a solução para o sistema $T^q_{yj} R^q_{yj} = L^k_{xj}$ pode ser expressa como $T^q_{yj} = L^k_{xj}(R^q_{yj})^{-1}$; ou seja, ela dependerá da existência da matriz $(R^q_{yj})^{-1}$ que é a inversa de R^q_{yj}. Se a matriz R^q_{yj} puder ser reduzida à matriz identidade por uma seqüência de operações elementares com linhas, então R^q_{yj} é inversível e a matriz inversa de R^q_{yj} é obtida a partir da matriz identidade, I, aplicando-se a mesma seqüência de operações [BOL86]. Na prática operamos simultaneamente as matrizes R^q_{yj} e I através de operações elementares até chegarmos à matriz I na posição correspondente à matriz R^q_{yj}. A matriz obtida no lugar correspondente à matriz I será a inversa de R^q_{yj}.

Essa resolução do problema trata apenas do caso onde existe a solução trivial; entretanto, existem casos onde não há solução ou ainda existem infinitas soluções. Para tratar esse problema podemos reformular a equação acima de forma a facilitar o tratamento de problemas com soluções além das triviais.

Escrevendo a equação $T^q_{yj} R^q_{yj} = L^k_{xj}$ na forma matricial, supondo m índices no *array* sendo acessado, temos:

$$\begin{pmatrix} t_{1,1} & t_{1,2} & \cdots & t_{1,m} \\ t_{2,1} & t_{2,2} & \cdots & t_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ t_{m,1} & t_{m,2} & \cdots & t_{m,m} \end{pmatrix} \begin{pmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,m} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ r_{m,1} & r_{m,2} & \cdots & r_{m,m} \end{pmatrix} = \begin{pmatrix} l_{1,1} & l_{1,2} & \cdots & l_{1,m} \\ l_{2,1} & l_{2,2} & \cdots & l_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ l_{m,1} & l_{m,2} & \cdots & l_{m,m} \end{pmatrix}$$

Desenvolvendo o sistema acima teremos

$$\begin{pmatrix} r_{1,1} & r_{2,1} & \cdots & r_{m,1} \\ r_{1,2} & r_{2,2} & \cdots & r_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ r_{1,m} & r_{2,m} & \cdots & r_{m,m} \end{pmatrix} \begin{pmatrix} t_{1,1} \\ t_{1,2} \\ \vdots \\ t_{1,m} \end{pmatrix} = \begin{pmatrix} l_{1,1} \\ l_{1,2} \\ \vdots \\ l_{1,m} \end{pmatrix}$$

$$\begin{pmatrix} r_{1,1} & r_{2,1} & \cdots & r_{m,1} \\ r_{1,2} & r_{2,2} & \cdots & r_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ r_{1,m} & r_{2,m} & \cdots & r_{m,m} \end{pmatrix} \begin{pmatrix} t_{2,1} \\ t_{2,2} \\ \vdots \\ t_{2,m} \end{pmatrix} = \begin{pmatrix} l_{2,1} \\ l_{2,2} \\ \vdots \\ l_{2,m} \end{pmatrix}$$

$$\vdots$$

$$\begin{pmatrix} r_{1,1} & r_{2,1} & \cdots & r_{m,1} \\ r_{1,2} & r_{2,2} & \cdots & r_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ r_{1,m} & r_{2,m} & \cdots & r_{m,m} \end{pmatrix} \begin{pmatrix} t_{m,1} \\ t_{m,2} \\ \vdots \\ t_{m,m} \end{pmatrix} = \begin{pmatrix} l_{m,1} \\ l_{m,2} \\ \vdots \\ l_{m,m} \end{pmatrix}$$

que por sua vez pode ser escrito como

$$\left[R^q_{yj}\right]^T \left[\left(T^q_{yj}\right)_i\right]^T = \left[(L^k_{xj})_i\right]^T$$

onde $\left(T^q_{yj}\right)_i$ é a linha i da matriz de transformação T^q_{yj} e $(L^k_{xj})_i$ é a linha i da matriz de acesso L^k_{xj}.

Vemos então que o problema foi reduzido à solução de sistemas lineares da forma $Ax=b$. Assim podemos reduzir o sistema à forma escada reduzida para encontrar a solução de cada um dos i sistemas de equações. Entretanto, o mais eficaz seria aplicar esse método a todos os sistemas ao mesmo tempo [BOL86], ou seja, reduzir a matriz de coeficientes a uma matriz na forma escada reduzida, aplicando as mesmas operações a cada vetor coluna b, que ao final conterá o resultado do sistema.

$$\left[\begin{array}{cccc|cccc} r_{1,1} & r_{2,1} & \cdots & r_{m,1} & l_{1,1} & l_{2,1} & \cdots & l_{m,1} \\ r_{1,2} & r_{2,2} & \cdots & r_{m,2} & l_{1,2} & l_{2,2} & \cdots & l_{m,2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ r_{1,m} & r_{2,m} & \cdots & r_{m,m} & l_{1,m} & l_{1,m} & \cdots & l_{m,m} \end{array}\right] \overset{\substack{\textit{forma} \\ \textit{escada} \\ \textit{reduzida}}}{\Rightarrow} \left[\begin{array}{c|cccc} \textit{Matriz} & l'_{1,1} & l'_{2,1} & \cdots & l'_{m,1} \\ \textit{Escada} & l'_{1,2} & l'_{2,2} & \cdots & l'_{m,2} \\ \textit{Reduzida} & \vdots & \vdots & \ddots & \vdots \\ & l'_{1,m} & l'_{1,m} & \cdots & l'_{m,m} \end{array}\right]$$

Mas apenas a redução à forma de matriz escada reduzida não significa que o sistema possui alguma solução. Devemos analisar o sistema resultante e verificar se ele é consistente, isto é possui pelo menos uma solução. Para um sistema da forma $Ax=b$, a consistência pode ser verificada segundo um dos critérios equivalentes abaixo [MEY00]:

- Durante a redução de $Ax=b$, uma linha da forma abaixo nunca aparece:

$$\left(0 \quad 0 \quad \cdots \quad 0 \mid \alpha\right) \qquad onde \quad \alpha \neq 0$$

- b é uma coluna não básica [4] em $[A \mid b]$
- $rank([A \mid b]) = rank(A)$ [5]
- b é uma combinação das colunas básicas em A

Portanto, teremos a transformação linear caso a consistência seja verificada, através de um dentre os critérios acima listados.

A solução de $T^q_{yj}\vec{\zeta}^q_{yj} + \vec{t}^q_{yj} = \vec{\zeta}^k_{xj}$ é facilmente obtida por operações de soma de vetores, ou seja o sistema pode ser rearranjado ficando como

$$\vec{t}^q_{yj} = \vec{\zeta}^k_{xj} - T^q_{yj}\vec{\zeta}^q_{yj} \qquad (4)$$

Realizado o alinhamento dos dados então o passo seguinte seria a divisão de computações e dados nos processadores virtuais.

4.2. DIVISÃO

A divisão também será representada por funções afins, mas renomeadas para decomposições afins [AND97]. São definidas duas decomposições afins, a saber: decomposição de dado afim e a decomposição de computação afim. A primeira busca uma função afim $d : A \mapsto P$ que divida um *array* de m dimensões num espaço de processadores de p dimensões, ou seja uma função do tipo $d(\vec{a}) = D\vec{a} + \vec{\delta}$ onde D é uma matriz de transformação linear e $\vec{\delta}$ é um vetor constante. A outra função afim, $c : I \mapsto P$, busca a divisão de um aninhamento de *loops* de l dimensões no espaço de processadores, $c(\vec{i}) = C\vec{i} + \vec{\gamma}$ onde C é uma matriz de transformação linear e $\vec{\gamma}$ é um vetor constante.

Para relacionar dado e computação, uma nova relação é definida tendo como agente de ligação a função de acesso a *array*, f^k_{xj} que representa a *k-ésima* função de acesso ao *array* x num aninhamento de *loops* j, dada por:

---

[4] Uma definição de coluna básica é apresentada no Apêndice A.

[5] Uma definição de rank é apresentada no Apêndice A.

$$f_{xj}^{k}(\vec{i}) = F_{xj}^{k}\ \vec{i} + \vec{\zeta}_{xj}^{k}$$

A definição da equação que relaciona dado e computação envolve $c_j(\vec{i})$ que é a decomposição de computação afim para um aninhamento de *loops* j e $d_x(\vec{a})$ que é a decomposição de computação afim para um *array* x. A ausência ou não da comunicação pode ser encontrada pela seguinte equação:

$$D_x\ \vec{a} + \vec{\delta}_x = C_j\ \vec{i} + \vec{\gamma}_j$$

que indica quando dado e computação serão associados ao mesmo processador virtual. Substituindo $\vec{a}$ pela função de acesso ao *array*:

$$D_x\,(f_{xj}^{k}(\vec{i})) + \vec{\delta}_x = C_j(\vec{i}) + \vec{\gamma}_j$$

$$D_x\,F_{xj}^{k}(\vec{i}) + D_x\vec{\zeta}_{xj}^{k} + \vec{\delta}_x = C_j(\vec{i}) + \vec{\gamma}_j$$

Como o objetivo da relação é maximizar o paralelismo, a decomposição de computação e dado afim buscada é tal que para todos os aninhamentos de *loops* j, o $rank(C_j)$ deve ser o maior possível ou seja o núcleo[6], $N(C_j)$, deve ser o menor possível. A comunicação devido ao deslocamento pode ser deixada de lado num primeiro momento resultando em:

$$D_x\,F_{xj}^{k}(\vec{i}) = C_j(\vec{i}) \qquad (5)$$

que é chamada de equação de comunicação.

Como resultado das equações de comunicação temos que as decomposições lineares não terão comunicação devido à reorganização de dados, mas poderão ter comunicação com o vizinho mais próximo devido aos deslocamentos, o que pode requerer sincronização dentro e/ou entre os aninhamentos de *loops*.

---

[6] Uma definição de núcleo é apresentada no Apêndice A.

A solução para o problema da sincronização, proposta por Anderson e Lam [AND97], parte do pressuposto de que o paralelismo não é tão efetivo em aninhamentos com requerimentos de sincronização quanto é naqueles sem, e conclui que é preferível encontrar decomposições lineares onde a sincronização é necessária apenas fora dos aninhamentos de *loops* e para tanto escolhem para a distribuição os *loops* paralelos mais externos [WOL92]. O resultado dessa escolha é que a não sincronização dentro de um alinhamento de *loops* significa que as iterações dos *loops* que não sejam os paralelos mais externos devem ser associadas ao mesmo processador.

As equações de sincronização que descrevem os *loops* que serão associados ao mesmo processador consideram o caso onde não existe sincronização, ou seja, quaisquer *loops* que não sejam os *loops* paralelos mais externos devem ser associados ao mesmo processador. Assim, para um alinhamento de *loops* j de profundidade l, sejam os *loops* de $1 \ldots s$ os *loops* paralelos mais externos e os *loops* $q = (s+1) \ldots l$ aqueles não paralelos. Temos então que as iterações $\vec{i}$ e $\vec{i} + \vec{e}_q$ devem ser associados ao mesmo processador, onde $\vec{e}_q$ é o $q-ésimo$ vetor elementar de dimensão l [AND97]:

$$C_j(\vec{i} + \vec{e}_q) \quad = C_j(\vec{i})$$

Ou seja, para cada alinhamento de *loops* j, e para cada *loop* $q = (s+1) \ldots l$ é gerado

$$C_j(\vec{e}_q) = 0 \qquad (6)$$

As equações de sincronização (6) e comunicação (5) representam as condições necessárias para um decomposição linear que permite somente a comunicação com o próximo vizinho e nenhuma sincronização dentro do alinhamento de *loops*

No caso do forall de *SISAL* essa situação não será um problema visto que ele é um "*loop*" paralelo por natureza. Usando essa propriedade temos a liberdade para a escolha do nível de paralelismo e podemos assim adotar uma política diferente para a distribuição das "iterações". A política para a distribuição das iterações pode selecionar quais iterações dos *loops* deixar juntas e quais deixar separado, segundo a carga de processamento existente, ou seja estamos livres para escolher o valor de s em $q = (s+1) \ldots l$.

Como a decomposição afim completa consiste de uma decomposição linear e uma decomposição de deslocamento, o passo seguinte seria então o cálculo da decomposição de deslocamento, desconsiderada num primeiro momento.

Assim, dada uma decomposição de dados afim completa, $D_x \vec{a} + \vec{\delta}_x$, para um *array* x referenciado num aninhamento de *loops* j, $F^k_{xj}(\vec{i}) + \vec{\zeta}^k_{xj}$, o deslocamento da função de divisão da computação $\vec{c}_j$ é dado por:

$$\vec{\gamma}_j = D_x \vec{\zeta}^k_{xj} + \vec{\delta}_x$$

O deslocamento da decomposição de dados, $\vec{\delta}_y$, para outro *array* y acessado no mesmo *loop* é:

$$\vec{\delta}_y = \vec{\gamma}_j - D_x \vec{\zeta}^k_{yj} \qquad (7)$$

Apesar da comunicação devido ao deslocamento não ser custosa, sempre que possível ela deve ser eliminada, mas não a ponto de colocar todo o *loop* num único processador.

O método utilizado realiza o alinhamento de dados e computação e também permite a exposição da opção de divisão de dados que possua o paralelismo no máximo grau possível ou seja o domínio da decomposição de dados, $rank(C_j)$, é o máximo possível. Isto é equivalente a dizer que o núcleo da decomposição de computação é o mínimo possível, pois a dimensão do domínio pode ser obtida por $n - \dim(Nucleo(C_j))$.

A solução para o sistema acima pode ser obtida solucionando-se o sistema que relaciona dados e computação (5) e (6) através da verificação dos valores possíveis que C_j e D_x podem assumir como solução do sistema, adotado por Anderson e Lam [AND97]. A solução adotada neste trabalho, difere daquele adotado por Anderson e Lam, visto que as condições encontradas primeiro devido à construção *forall* serem paralelas e segundo devido à aplicação do alinhamento, favorecem a escolha de C_j e D_x.

Portanto temos englobado neste método o alinhamento de dados e a computação é deixada disponível para a divisão quais as dimensões de dados e computações devem ser distribuídas entre os processadores. O passo seguinte é o cálculo dos índices de dados e computações que serão distribuídos nos processadores físicos.

4.3. MAPEAMENTO

Uma vez que o alinhamento de computação e dados tenha sido realizado e a seleção de quais dimensões de *arrays* e aninhamentos de *loops* serão distribuídos entre os processadores virtuais, assim estabelecendo a relação dados/computação, o passo seguinte é mapear as distribuições escolhidas nos processadores físicos. No texto que segue, nos referimos apenas à iteração de *loops*, visto que os relacionamentos com os dados já foram estabelecidos, a referência a *arrays* a mapear fica implícita.

Até este ponto, consideramos um espaço de processadores virtual com dimensão suficiente mesmo para um paralelismo de grau máximo, mas caso o número de processadores virtuais seja maior que o número de processadores físicos disponíveis, é necessária então uma função que realize esse ajuste de processadores virtuais nos processadores disponíveis, mais especificamente escolher uma distribuição por blocos, em ciclos ou uma mescla das duas anteriores, bloco/cíclico.

Antes de apresentarmos as funções para as distribuições citadas, vejamos algumas definições. Consideremos um domínio de índices $\vec{a}$ para o *array* v de m dimensões e domínio de iteradores $\vec{i}$ para um aninhamento de *loops* de profundidade n, como definido anteriormente

$$\vec{a} = \begin{bmatrix} a_1 & a_2 & \cdots & a_m \end{bmatrix}^T$$

e

$$\vec{i} = \begin{bmatrix} i_1 & i_2 & \cdots & i_n \end{bmatrix}^T$$

onde cada a_x é o *x-ésimo* índice do *array* v e $\vec{a}$ é um vetor contendo todos os índices deste *array* e cada i_x é o *x-ésimo* iterador do aninhamento de *loops* de profundidade n.

Como vimos anteriormente, no domínio de índices cada índice corresponde a uma linha não zero da matriz F que representa a ocorrência dos índices de v, cada linha não zero terá uma referência a um iterador cujos limites determinarão o intervalo destes índices, que pode ser definida como as inequações:

$$\vec{l} \leq \vec{a} \leq \vec{h}$$

onde $\vec{l}$ e $\vec{h}$ são vetores de dimensão $m \times 1$ que representam os limites inferior e superior do *array*. Rearranjando a inequação acima temos

$$\left[\frac{-I}{I}\right]\vec{a} \leq \left[\frac{-\vec{l}}{\vec{h}}\right] \tag{7}$$

Da mesma forma, podemos obter intervalo para os iteradores a serem mapeados.

$$\vec{\lambda} \leq \vec{i} \leq \vec{\eta}$$

onde $\vec{\lambda}$ e $\vec{\eta}$ são vetores de dimensão $n \times 1$ que representam respectivamente os limites inferior e superior dos *loops*. Rearranjado a inequação acima temos

$$\left[\frac{-I}{I}\right]\vec{i} \leq \left[\frac{-\vec{\lambda}}{\vec{\eta}}\right] \tag{8}$$

Apesar das inequações (7) e (8) terem sido apresentadas separadamente, na representação de dados e computações de forma disjunta, sabemos pela equação (5) como eles estão relacionadas e, portanto realizaremos a divisão levando em conta esta informação. Seguindo neste sentido, temos que os número de aninhamentos de loops ou índices dos arrays a serem divididos podem ser relacionados pelo número de *loops* a distribuir, q, sendo o valor máximo dado por $q_{\max} = (1, \min(m,n)) = Q$.

Podemos então usar este valor de q, na divisão das dimensões do *array* (mais de uma dimensão) e dos aninhamentos de *loops* (mais de um aninhamento), para rearranjar os p processadores do espaço de processadores, ρ, e teríamos o seguinte sub-espaço:

$$\rho \mapsto \underline{\rho_1} \times \underline{\rho_2} \times \ldots \times \underline{\rho_q}$$

onde ρ_x representa os processadores da *x-ésima* dimensão, $1 \leq x \leq q$ com cada um dos sub-espaços do espaço de processadores associado a uma dimensão do *array* e um aninhamento de *loops*.

Como não trataremos detalhadamente o balanceamento de carga, o que formulamos a seguir é apenas uma tentativa de estabelecer um número de processadores que será associado a cada iteração no aninhamento de *loops*, de forma que a distribuição dos dados seja a mais uniforme possível.

A carga média será calculada por

$$b = \left\lceil \frac{\sum_{x=1}^{q} (\eta_x - \lambda_x + 1)}{p} \right\rceil \qquad (9)$$

onde b é a carga média e λ_x e η_x são os limites superior e inferior de cada uma das iterações do aninhamento de *loops*. Calculado o valor de carga média, b, podemos calcular o número de processadores virtuais em cada uma das iterações do *loop* por:

$$p_x = \left\lceil \frac{\eta_x - \lambda_x + 1}{b} \right\rceil \qquad (10)$$

Estabelecida à forma como serão distribuídos os p processadores entre os q sub-espaços, o passo seguinte apresenta como será realizada a divisão tanto de *arrays* como *loops*.

As iterações dos aninhamentos de *loops* a serem divididas são dadas por

$$i_1, i_2, \ldots, i_q$$

e as dimensões do *array* a serem divididas são dadas por

$$a_1, a_2, \ldots, a_q$$

A função de divisão de cada uma das q dimensões é dada por

$$f_i : i_i \mapsto \underline{i_{i1}} \times \underline{i_{i2}} \times \cdots \times \underline{i_{iz_x}}$$

para o aninhamento de *loops* e

$$f_x : a_x \mapsto \underline{a_{x1}} \times \underline{a_{x2}} \times \cdots \times \underline{a_{xz_x}}$$

para os índices do *array*.

Para algum processador, $z_x \in 1 \ldots p_x$, os limites dos índices $\underline{i_{iz_x}}$ descrevem os elementos de aninhamento de *loops*, locais ao processador z da dimensão x e os limites dos índices $\underline{a_{xz_x}}$ descrevem os elementos de um *array* v, locais ao processador z da dimensão x.

Os limites dos *loops* locais ao processador z_x consistem de seqüências de valores distintos dados por

$$\underline{i_{iz_x}} = \underline{i^1_{iz_x}} \times \underline{i^2_{iz_x}} \times \ldots \times \underline{i^{d_x}_{iz_x}}$$

sendo que d_x é o número de vezes que o *loop* é empacotado nos processadores, geralmente uma vez. Os limites dos *arrays* locais ao processador z_x consistem de seqüências de valores distintos dados por

$$\underline{a_{xz_x}} = \underline{a^1_{xz_x}} \times \underline{a^2_{xz_x}} \times \ldots \times \underline{a^{d_x}_{xz_x}}$$

Reformulando a inequação (8) anterior para definir o valor dos índices locais em algum processador z, temos então como novo conjunto de inequações:

$$\left[\frac{-e_x}{e_x}\right]\begin{bmatrix} i^{k_x}_{1z_x} \\ i^{k_x}_{2z_x} \\ \vdots \\ i^{k_x}_{Qz_x} \end{bmatrix} \leq \left[\frac{-\left[(z_x-1)\times b+(k_x-1)\times r_x+\lambda_x\right]}{z_x \times b+(k_x-1)\times r_x+\lambda_x-1}\right] \qquad (11)$$

onde e_x é a x-ésima linha da matriz identidade.

Para o aninhamento de loops de (7) temos

$$\left[\frac{-e_x}{e_x}\right]\begin{bmatrix} a^{k_x}_{1z_x} \\ a^{k_x}_{2z_x} \\ \vdots \\ a^{k_x}_{Qz_x} \end{bmatrix} \leq \left[\frac{-[(z_x-1)\times b_x+(k_x-1)\times r_x+l_x+(\lambda_x-1)+\delta_x]}{z_x\times b_x+(k_x-1)\times r_x+l_x+(\lambda_x-1)+\delta_x-1}\right] \tag{12}$$

para o índices dos *arrays* onde δ_x é o deslocamento encontrado no acesso aos dados.

As formas de *d*, *b*, *z* e *k* para a divisão em mais que um índice são apresentadas a seguir.

(I) d é o número de seqüências ou empacotamentos, representada por:

$$d=(d_1,d_2,\cdots,d_q) \tag{13}$$

onde d_x número de seqüências ou empacotamentos associados a iteração x, dada por

$$d_x=\left\lceil\frac{\eta_x-\lambda_x+1}{r_x}\right\rceil \tag{14}$$

onde

$$r_x=b\times p_x \tag{15}$$

onde b é a quantidade de dados associada ao processador p_x.

(II) z_x é o processador onde estão associados os índices correspondentes à dimensão x.

$$1\leq z_x\leq p_x \tag{16}$$

(III) *k* é o índice do empacotamento na seqüência de índices de uma iteração de *loops* associado ao processador *x*, dado por:

$$1\leq k_x\leq d_x \tag{17}$$

onde k_x é a k-ésima seqüência de índices associados à dimensão x.

4.4 ESTUDO DOS MÉTODOS COM *SISAL*

4.4.1. Alinhamento

Apresentaremos nessa seção a aplicação da transformação, vista na seção 3.2, que busca verificar se é possível o alinhamento de dados de exemplos de trechos de código *SISAL*. Iniciaremos com um exemplo simples, no código da **Figura 18** abaixo, onde temos a criação de um *array* `a` partir de um *array* `b`, e também vemos que o *array* `a` é criado com os índices `(i,j,k)` e o *array* `b` é acessado pelos índices `(j,i,k)`, o que mostra que os dois acessos não estão com um alinhamento perfeito.

```
% dimensão de b                b[1..100][1..100][1..100]
a := for i in 1, 100 cross j in 1, 100 cross k in 1, 100
     returns array of b[j,i,k]
      end for
```

Figura 18. Exemplo simples de um código *SISAL* onde é possível alinhar os dados

Os acessos aos *array* a e b são dados por:

$$l^1_{a1}(\vec{a}) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{e} \qquad r^1_{b1}(\vec{a}) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Como vimos, pela equações (3) e (4), o alinhamento pode ser formulado como

$$T^1_{b1} R^1_{b1} = L^1_{a1} \qquad \vec{t}^1_{b1} = \vec{\zeta}^1_{a1} - T^1_{b1} \vec{\zeta}^1_{b1}$$

Assim, resolvendo o sistema como proposto anteriormente na seção 4.1 temos

$$\left[\begin{array}{ccc|ccc} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right] \overset{\substack{\textit{trocando} \\ \textit{a 1° e 2°} \\ \textit{linhas}}}{\Rightarrow} \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

Vemos pelo sistema acima que ele possui soluções consistentes pois uma linha do tipo $\left(0 \quad 0 \quad \cdots \quad 0 \;\middle|\; \alpha\right)$ onde $\alpha \neq 0$ não aparece no sistema. Portanto temos como solução do sistema

$$T_{b1}^{1} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{e} \quad \vec{t}_{b1}^{1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Analisemos um outro exemplo um pouco diferente apresentado abaixo na **Figura 19**.

```
a := for i in 1, 100 cross j in 1, 100 cross k in 1, 100
     returns array of b[j,j,k]
     end for
```

Figura 19. Exemplo simples de um código *SISAL* onde não é possível alinhar os dados

Os acessos aos *array* a e b são dados por:

$$l_{a1}^{1}(\vec{a}) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{e} \quad r_{b1}^{1}(\vec{a}) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Assim, resolvendo o sistema resultante, temos a seguinte matriz escada

$$\left[\begin{array}{ccc|ccc} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right] \xRightarrow{\substack{\textit{multiplicando} \\ \textit{por } -1 \textit{ a } 1^{\circ} \textit{linha} \\ \textit{e somando à } 2^{\circ}}} \left[\begin{array}{ccc|ccc} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right] \xRightarrow{\substack{\textit{trocando a } 2^{\circ} \\ \textit{linha com a } 3^{\circ}}}$$

$$\left[\begin{array}{ccc|ccc} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 & 0 \end{array}\right]$$

Vemos pelo sistema acima que ele não possui soluções consistentes pois a 3ª linha é do tipo $\left(0 \quad 0 \quad \cdots \quad 0 \;\middle|\; \alpha\right)$ onde $\alpha \neq 0$, logo não será possível realizar o alinhamento.

Para finalizar vejamos um exemplo um pouco mais interessante. Seja o código da **Figura 20**, abaixo:

```
a := for i in 1, 100 cross j in 1, 100 cross k in 1, 100
     returns array of b[i+j,k+j,k]+c[k+j+2,j,i+j+1]
     end for
```

Figura 20. Exemplo de um código *SISAL* onde é possível alinhar os dados

Na **Figura 20**, temos como funções de acesso aos *arrays* a, b e c

$$l_{a1}^{1}(\vec{a}) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad r_{b1}^{1}(\vec{a}) = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

e

$$r_{c1}^{1}(\vec{a}) = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}$$

E teremos então dois sistemas distintos

$$T_{b1}^{1} R_{b1}^{1} = L_{a1}^{1} \qquad \vec{t}_{b1}^{1} = \vec{\zeta}_{a1}^{1} - T_{b1}^{1} \vec{\zeta}_{b1}^{1}$$

e

$$T_{c1}^{1} R_{c1}^{1} = L_{a1}^{1} \qquad \vec{t}_{c1}^{1} = \vec{\zeta}_{a1}^{1} - T_{c1}^{1} \vec{\zeta}_{c1}^{1}.$$

À parte que calcula a transformação linear escrita na forma de matriz pode ser visualizada abaixo.

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right] \quad \text{e} \quad \left[\begin{array}{ccc|ccc} 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{array}\right]$$

Após a redução para a matriz escada temos então:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 1 & 0 \\ 0 & 0 & 1 & 1 & -1 & 1 \end{array}\right] \quad \text{e} \quad \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & -1 & 1 & -1 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{array}\right]$$

Pela matrizes escada podemos constatar que temos soluções consistentes para os dois alinhamentos pois linhas do tipo $\begin{pmatrix} 0 & 0 & \cdots & 0 & | & \alpha \end{pmatrix}$ *onde* $\alpha \neq 0$ não aparecem nos sistemas.

Resolvendo para a parte do deslocamento

$$\vec{t}_{b1}^{1} = \vec{\zeta}_{a1}^{1} - T_{b1}^{1}\vec{\zeta}_{b1}^{1} \qquad \text{e} \qquad \vec{t}_{c1}^{1} = \vec{\zeta}_{a1}^{1} - T_{c1}^{1}\vec{\zeta}_{c1}^{1}$$

temos

$$\vec{t}_{b1}^{1} = \begin{bmatrix} 1 & -1 & 1 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{e} \qquad \vec{t}_{c1}^{1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 & -1 & 1 \\ 0 & 1 & 0 \\ 1 & -1 & 0 \end{bmatrix}\begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ -2 \end{bmatrix}$$

Portanto temos como solução

$$T_{b1}^{1} = \begin{pmatrix} 1 & -1 & 1 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \end{pmatrix}, \ \vec{t}_{b1}^{1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{e} \qquad T_{c1}^{1} = \begin{pmatrix} 0 & -1 & 1 \\ 0 & 1 & 0 \\ 1 & -1 & 0 \end{pmatrix}, \ \vec{t}_{c1}^{1} = \begin{bmatrix} -1 \\ 0 \\ -2 \end{bmatrix}$$

O que vimos acima nesses exemplos, no entanto é o padrão segundo o qual os dados deveriam estar alinhados. Se aplicarmos a transformação necessária ao código *SISAL* da **Figura 21 (a)**, de forma a ter os dados alinhados no momento que realizarmos a divisão do *forall*, o código obtido seguindo a transformação linear ficaria como apresentado na **Figura 21 (b)**.

```
a := for i in 1, 4 cross j in 1, 4
     returns array of b[i+j,j]
     end for
```

(a)

```
b1 := for i in 1,4 cross j in 1,4
      returns array of b[i+j,j]
      end for

a:= for i in 1,4 cross j in 1,4
    returns array of b1[i,j]
    end for;
```

(b)

Figura 21. Exemplo de um código *SISAL* onde é possível alinhar os dados

ou seja, a função de acesso à matriz b será agora realizada por

$$\left(r_{b1}^{1}\right)'(\vec{b}) = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix}$$

Ao invés desta transformação no código podemos ter um acesso na matriz `b` pela transformação encontrada, ou seja, pode ser realizada durante a distribuição dos dados na divisão do *forall*. Para ilustrar mais claramente o que foi descrito suponha que vamos dividir o *forall* apenas pelo índice `i`. O padrão de acesso à matriz está ilustrado na **Figura 22**.

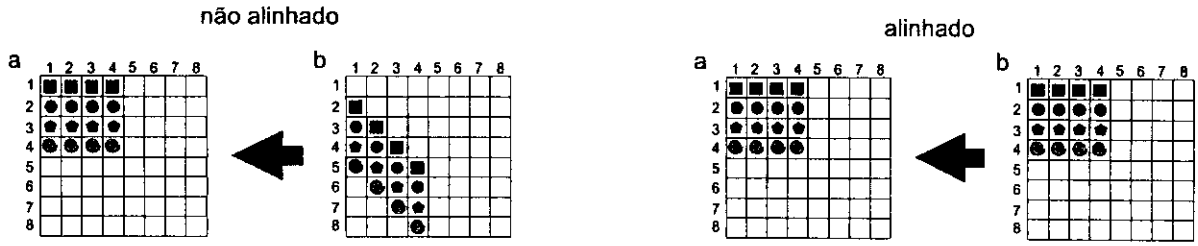


Figura 22. Ilustração da distribuição da matriz b do exemplo da Figura 21

A **Figura 21** significa que todo o dado a ser distribuído deverá ser feito através da aplicação da matriz de transformação encontrada. Assim o dado estará ajustado segundo o novo padrão de acesso devido às mudanças realizadas no código do corpo do *forall*, **Figura 21**.

4.4.2 Divisão

O passo seguinte após o alinhamento de dados segundo a proposta das seções anteriores é a verificação de quais dados e computações deveriam ser colocadas no mesmo processador, ou seja, através das equações (5) e (6) que relacionam dados e computação. Para o exemplo da **Figura 21** após o alinhamento, supondo que desejamos distribuir apenas o *loop* mais externo do *forall*, encontraríamos:

$$C_1(\vec{e}_2) = 0$$

para os *foralls* que serão colocados no mesmo processador e

$$D_1 F_{a1}^1 = C_1 \qquad\qquad D_1 F_{b1}^1 = C_1$$

para dados e computações que serão colocados juntos. Expandindo esses sistemas temos

$$C_1\begin{bmatrix}0\\1\end{bmatrix}=0 \qquad\qquad D_1\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}=C_1 \qquad\qquad D_1\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}=C_1$$

ou seja temos como uma possível solução $C_1 = [1 \quad 0]$ e $D_1 = [1 \quad 0]$. Isto indica que uma opção seria distribuir dados e computação pelo primeiro índice.

Entretanto vemos que essa não é a única solução possível. Outra solução seria $C_1 = [0 \quad 0]$ e $D_1 = [0 \quad 0]$ entretanto esta solução significa que não distribuiremos dados e computações entre os processadores, mas colocaríamos todos os elementos no mesmo processador. Com essa solução não teríamos distribuição de computação e dados, o que vai contra exatamente o que procuramos.

4.4.3. Mapeamento

Decidido o que vai ser distribuído, o passo final seria então calcular os índices dos foralls e dos *arrays* que serão distribuídos entre os processadores. Assim, pelo resultado encontrado anteriormente, a opção é a distribuição do primeiro *forall* e da primeira dimensão do *array*.

Usando sistema de inequações (7) e (8) para o exemplo da **Figura 18**, supondo que já tenham sido realizados o mapeamento e a divisão, temos então o seguintes intervalos para *arrays* e iterações:

$$\begin{bmatrix}-1 & 0 & 0\\0 & -1 & 0\\0 & 0 & -1\\\hline 1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}a_1\\a_2\\a_3\end{bmatrix}\le\begin{bmatrix}-1\\-1\\-1\\\hline 100\\100\\100\end{bmatrix} \quad \text{e} \quad \begin{bmatrix}-1 & 0 & 0\\0 & -1 & 0\\0 & 0 & -1\\\hline 1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix}\le\begin{bmatrix}-1\\-1\\-1\\\hline 100\\100\\100\end{bmatrix}$$

Pelo resultado obtido anteriormente, na seção 4.3, $C_1 = [1 \quad 0 \quad 0]$ e $D_1 = [1 \quad 0 \quad 0]$, uma possível distribuição seria $q = 1$, ou seja, pelo primeiro índice do aninhamento de *loops* e primeira dimensão do *array*.

Supondo então que tenhamos 4 processadores físicos e realizando a distribuição seguindo (9) e (10) teríamos

$$b = \left\lceil \frac{\sum_{x=1}^{1}(\eta_x - \lambda_x + 1)}{4} \right\rceil = \left\lceil \frac{(100-1+1)}{4} \right\rceil = 25$$

e

$$p_1 = \left\lceil \frac{\eta_1 - \lambda_1 + 1}{b} \right\rceil = \left\lceil \frac{100-1+1}{b} \right\rceil = 4$$

ou seja, o número de elementos por processador será 25 e o número de processadores associado à dimensão que será dividida são 4.

Antes de desenvolver as inequações dadas por (11) e (12), calculemos (13) a (17) que serão utilizados no cálculo do índices. Teríamos então

$$x = 1 \qquad b = 25 \qquad r_1 = b \times p_1 = 25 \times 4 = 100$$

$$d = (d_1) \qquad 1 \le z_1 \le 4 \qquad 1 \le k_1 \le 1 \qquad d_1 = \left\lceil \frac{\eta_1 - \lambda_1 + 1}{r_1} \right\rceil = \left\lceil \frac{100-1+1}{100} \right\rceil = 1$$

Calculados esses valores, podemos desenvolver (11) e (12) e teremos então:

$$\left[\frac{-e_1}{e_1}\right] \begin{bmatrix} \vec{i}_{1z_1}^{k_1} \\ \vec{i}_{2z_1}^{k_1} \\ \vec{i}_{3z_1}^{k_1} \end{bmatrix} \le \left[\frac{-[(z_1-1)\times b + (k_1-1)\times r_1 + \lambda_1]}{[z_1 \times b + (k_1-1)\times r_1 + \lambda_1 - 1]}\right] =$$

$$\left[\frac{-1 \quad 0 \quad 0}{1 \quad 0 \quad 0}\right] \begin{bmatrix} \vec{i}_{1z_1}^{1} \\ \vec{i}_{2z_1}^{1} \\ \vec{i}_{3z_1}^{1} \end{bmatrix} \le \left[\frac{-[(z_1-1)\times 25 + (1-1)\times 100 + 1]}{[z_1 \times 25 + (1-1)\times 100 + 1 - 1]}\right] =$$

$$\left[\frac{-1 \quad 0 \quad 0}{1 \quad 0 \quad 0}\right] \begin{bmatrix} \vec{i}_{1z_1}^{1} \\ \vec{i}_{2z_1}^{1} \\ \vec{i}_{3z_1}^{1} \end{bmatrix} \le \left[\frac{-[(z_1-1)\times 25 + 1]}{[z_1 \times 25]}\right]$$

$z_1 = 1$	$z_1 = 2$	$z_1 = 3$	$z_1 = 4$
$1 \le i_{11}^1 \le 25$	$26 \le i_{12}^1 \le 50$	$51 \le i_{13}^1 \le 75$	$76 \le i_{14}^1 \le 100$

e

$$\left[\frac{-e_1}{e_1}\right]\begin{bmatrix}\vec{a}^{k_1}_{1z_1}\\ \vec{a}^{k_1}_{1z_1}\\ \vec{a}^{k_1}_{1z_1}\end{bmatrix} \leq \left[\frac{-[(z_1-1)\times b+(k_1-1)\times r_1+l_1+(\lambda_1-1)+\delta_1]}{[z_1\times b+(k_1-1)\times r_1+l_1+(\lambda_1-1)+\delta_1-1]}\right]=$$

$$\left[\frac{-1\quad 0\quad 0}{1\quad 0\quad 0}\right]\begin{bmatrix}\vec{a}^{1}_{1z_1}\\ \vec{a}^{1}_{1z_1}\\ \vec{a}^{1}_{1z_1}\end{bmatrix} \leq \left[\frac{-[(z_1-1)\times 25+(1-1)\times 100+1+(1-1)+0]}{[z_1\times 25+(1-1)\times 100+1+(1-1)+0-1]}\right]$$

$$\left[\frac{-1\quad 0\quad 0}{1\quad 0\quad 0}\right]\begin{bmatrix}\vec{a}^{1}_{1z_1}\\ \vec{a}^{1}_{1z_1}\\ \vec{a}^{1}_{1z_1}\end{bmatrix} \leq \left[\frac{-[(z_1-1)\times 25+1]}{[z_1\times 25]}\right]$$

$z_1=1$	$z_1=2$	$z_1=3$	$z_1=4$
$1\leq a^1_{11}\leq 25$	$26\leq a^1_{12}\leq 50$	$51\leq a^1_{13}\leq 75$	$76\leq a^1_{14}\leq 100$

4.5. CONCLUSÃO

Neste capítulo descrevemos os métodos utilizados nas análises estáticas dos códigos *SISAL* visando a otimização da distribuição de dados e de computações. O métodos apresentados utilizam Álgebra Linear e Programação Linear para a elaboração de técnicas para alinhamento e decomposição de dados e computações.

Num primeiro momento vimos como realizar o alinhamento de *arrays* no corpo de um *loop forall*. Essencialmente, o alinhamento consistia da busca de uma transformação linear que satisfizesse o acesso padrão escolhido.

Em seguida são realizadas as análises para a distribuição conjunta de dados e computações do *loop forall*, baseadas em funções de afins, para computação e para dados, relacionadas pela função de acesso ao *array*. Num primeiro momento são definidos quais os *loops* que serão distribuídos e de posse dessa informação, quais as dimensões do *arrays* serão distribuídas.

E, para finalizar, com a informação obtida sobre quais dimensões do *array* e *loops* do aninhamento distribuir, realiza-se a divisão de dados e computações.

5. IMPLEMENTAÇÃO

Apresentaremos a seguir uma descrição de quais estruturas e ferramentas foram utilizadas para a implementação dos métodos para a análise de dados e códigos propostos no Capítulo 4. No entanto, antes dos detalhes de implementação, será apresentada a forma intermediária, IF1, utilizada pelo compilador OSC para representar os programas *SISAL*. Especificamente, a construção que será apresentada em detalhes é o nó composto *forall* que representa o comando *forall* de *SISAL*.

Na seção 5.3 apresentaremos detalhes da integração do módulo de otimização desenvolvido e o compilador OSC. Na seção 5.4, descreveremos a estrutura utilizada para armazenar os programas IF1, visto que no planejamento das estruturas tivemos o cuidado de representar fielmente o código IF1, disponibilizado pelo *front-end* do compilador OSC. Uma grande vantagem desse código IF1, que representa grafos, é o fato dela ser definida como uma linguagem intermediária, com regras léxicas, sintáticas e semânticas, ou seja, podemos dizer que as análises da estrutura que representa esse código são na realidade análises sobre uma linguagem de grafos.

As estruturas acima mencionadas são montadas em tempo de compilação e os programas que realizam esta tarefa foram gerados usando-se as ferramentas *Flex* [PAX95] (análise léxica) e *Bison* [DON95] (análise sintática) disponíveis no *site* da *Free Software Foundation, Inc*, o que facilitou bastante o nosso trabalho.

Nas seções seguintes, 5.5, 5.6, 5.7 e 5.8 apresentaremos uma descrição de como foram implementadas as propostas apresentadas no Capítulo 4, ou seja, como foi realizada a extração das funções afim, a base para as análises de alinhamento e divisão, os métodos utilizados para a solução dos sistemas de equações do alinhamento e da divisão e qual a política adotada para o mapeamento de código e dado a um sistema de processadores. Na seção 5.9 apresentaremos como foi realizada a geração de código *IF1* acrescida de chamadas a MPI.

5.1. LINGUAGEM INTERMEDIÁRIA

Na seção 2.5 do Capítulo 2, apresentamos brevemente a linguagem intermediária IF1, utilizada pelo compilador OSC nas análises para a otimização de programas SISAL.

Como já citado, IF1 é uma forma intermediária desenvolvida para uso com linguagens aplicativas, fortemente baseada na característica de associação simples de SISAL e VAL [GHA88].

Um programa em IF1 é uma hierarquia de grafos de fluxo de dados acíclicos, que representam as dependências de dados inerentes e descrevem o fluxo de dados (arcos) e fluxo de controle de um programa (nó composto e nó de chamada de função). A estrutura hierárquica dos programas IF1 deve-se ao fato que um grafo pode conter nós compostos, que por sua vez podem conter grafos.

São quatro componentes básicos para um grafo de IF1, a saber, os nós (ou vértices) que denotam as operações, os arcos que representam os valores que são passados de um nó para outro, os tipos que podem estar junto a um dado ou função e fronteiras do grafo cercando os nós e arcos. Em geral, o número de entradas e saídas dos nós varia de acordo com o tipo de operação que eles representam.

Também já vimos que um programa IF1 é um arquivo com diversas linhas com caracteres ASCII, separadas pelo caractere de nova linha. O primeiro caractere não branco na linha indica o que ela representará, podendo ser declaração de tipo, arco ou arco literal, função local ou global, grafo, nó simples ou composto e comentário.

Detalharemos a seguir o nó composto de IF1 que representa o comando *forall* de *SISAL*, visto que esta foi a estrutura central das análises propostas neste trabalho. Uma representação deste nó, composta de três subgrafos (gerador, corpo e retorno) pode ser vista na Figura 10, da Seção 2.5, do Capítulo 2.

A definição de nó composto de IF1 é expressa tanto por regras de sintaxe, quanto regras semânticas; no caso do nó composto *forall* isto não é diferente. Do manual de definição da linguagem IF1 [SKE85a] temos como regras de produção definindo a sintaxe para o nó composto:

```
Line  ::= {  "Compound"  Label     Node
      |   G  TypeReference
      |   N  label     node
      |   E  Source    Destination  TypeReference
      |   L            Destination  TypeReference  "literal"
      |   L            Destination  TypeReference  ErrorValue
      |   }  Label     Node         Count          Assoc
```

Entretanto, devemos ressaltar que apenas as regras de produção de IF1 não são suficientes para expressar um comando *forall* de SISAL, ou seja, o nó composto *forall* de IF1. Entretanto, na definição de IF1 além das regras de produção para a sintaxe da linguagem, temos regras semânticas definindo as estruturas de nós simples, nós compostos, etc.

Apresentaremos a seguir as regras sintáticas e semânticas gerais de um nó composto de IF1 e num segundo momento as definições específicas para um nó composto *forall*. A listagem completa das regras de IF1 pode ser vista no Apêndice B.

5.1.1. Nós compostos [SKE85a]

Os nós compostos de IF1 podem conter um número fixo de subgrafos, no caso dos nós *Forall*, *LoopA* e *LoopB*, ou um número variável de subgrafos no caso dos nós *Select* e *TagCase*. As regras semânticas de cada nó composto ditam como estarão relacionadas suas entradas e saídas com as entradas e saídas dos subgrafos.

A representação de um nó composto em IF1 é:

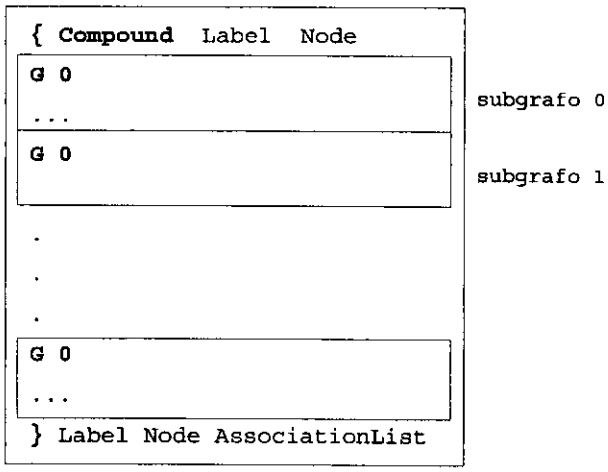


Figura 23. Representação geral de um nó composto IF1

A linha, { `Compound` `Label` `Node` marca o início de um nó composto e a linha, } `label` `Node` `AssociationList`, o final dele. O início do grafo tem como informações o rótulo do nó composto e o tipo de nó composto que está sendo representado. No final temos a mesma informação sobre o rótulo do nó composto, o tipo do nó e adicionalmente temos `AssociationList` que representa uma lista de *k* inteiros,

IFSC-USP SERVIÇO DE BIBLIOTECA E INFORMAÇÃO

$k \ a_1 a_2 \ldots a_k$, nomeada lista de associação. Essa lista representa a semântica de ligação entre os subgrafos de cada nó composto. Como os subgrafos podem aparecer em qualquer ordem, no arquivo texto, a lista de associação define um mapeamento entre a ordem textual e a ordem real de uso no nó composto. Visto que cada nó usa os subgrafos de forma diferente, a lista de associação tem significado diferente em cada tipo de nó composto.

Entre o início e o final do nó composto, vemos linhas com `G 0`, que indica o início de um subgrafo, que termina com outra linha do mesmo tipo ou com a linha que indica o final do nó composto.

Além da semântica que dita a estrutura de um nó composto, podemos ter três tipos de dependências implícitas dentro de um nó composto, representando dependência de dados ou controle. São elas:

I. Dependência de dados entre o nó composto e seus subgrafos, que ocorre quando um nó composto passa alguns de seus valores de entrada para seus subgrafos, ou quando um subgrafo retorna valores que se tornam os resultados do nó composto.

II. Dependência de dados entre os subgrafos que usualmente aparece em *loops*, onde os valores calculados na inicialização e corpo do *loop* são passados para a parte de resultado e de volta ao corpo do *loop*.

III. Dependência de controle que ocorre em seleções e *loops* iterativos, onde um predicado determina que expressão avaliar ou quando um *loop* iterativo deve terminar.

E, finalmente, valores que são passados a subgrafos de nós compostos serão definidos como classes de valores, tais como valores importados ou valores de *loops*. Os membros de cada classe de valores serão associados a números de portas contíguos nas fronteiras dos subgrafos.

5.1.2. Nó Composto *FORALL* [SKE85a]

Como já apresentado anteriormente, o nó composto *forall* é usado para denotar a execução independente de múltiplas instâncias de uma expressão, sendo composto de três subgrafos: gerador, corpo e retorno.

O corpo contém a expressão a ser avaliada, o gerador produz os valores de cada instância do corpo, ou seja, produz pelo menos um valor múltiplo nas portas de classe M,

Tabela 3. Cada um dos elementos do valor múltiplo é enviado para cada uma das instâncias distintas do corpo. Como os valores e instâncias do corpo estão ordenados, o subgrafo de retorno coletará o resultado nessa mesma ordem.

Quando mais que um valor múltiplo é produzido pelo gerador, o primeiro valor em cada classe M de arcos é enviado à primeira instância do corpo e assim por diante.

Os produtos cartesianos são representados por *foralls* aninhados e os produtos *dot* são representados por mais que um nó *Ascatter* ou *RangeGenerate* no subgrafo gerador.

Como em todo nó composto, no *forall* temos também a definição de dependências implícitas, descritas a seguir.

I. Todas as entradas do nó *forall* são acessíveis a cada subgrafo, através da classe de portas K.
II. O subgrafo corpo recebe um valor de cada classe de portas M.
III. Os resultados do corpo (classe T) estão conectados às entradas dos subgrafos retorno.
IV. Os resultados do subgrafo retorno (classe R) estão conectados às portas de saída do nó forall.

As tabelas a seguir descrevem o uso das portas no nó composto *forall*. A primeira tabela associa números de portas a cada classe e a segunda as portas que podem ser usadas como importação e como resultado.

Tabela 3. Descrição das classes de portas do nó composto forall

Classe	Uso	Intervalo
K	Valores importados para o nó *forall*	$1..n_K$
M	Valores múltiplos	$n_K+1..n_K+n_M$
T	Resultados de cada corpo	$n_K+n_M+1..n_K+n_M+n_T+1$
T	Resultados do nó *forall*	$1..n_R$

Na **Figura 24** apresentaremos um exemplo de um nó *forall* representando o seguinte trecho de código *SISAL*.

```
for i in 1, 10 cross j in 1, 10 cross k in I, 10
returns array of a[k,j,i]
end for
```

Tabela 4. Associação das portas nó composto forall

	Importação	Resultado
forall	K	R
gerador	K	M
corpo	K, M	T
retorno	K, M, T	R

5.2. FERRAMENTAS FLEX E BISON

Descreveremos a seguir as ferramentas *flex* e *bison*, utilizadas na construção do *front-end* para a tradução da linguagem Intermediária IF1 para a estrutura de dados em C que será utilizada na implementação das análises propostas no Capítulo 4. Também apresentaremos os arquivos de entrada de ambas as ferramentas, que expressam as regras léxicas e sintáticas da linguagem intermediária IF1.

5.2.1 FLEX

O FLEX é um gerador de analisador léxico, distribuído gratuitamente, que produz um programa C a partir de uma especificação léxica (gramática regular). Essa geração automática é possível visto que a construção de um *Autômato Finito Determinista* [AHO86] e sua respectiva *Tabela de Transições* [AHO86] a partir de uma gramática é uma tarefa mecânica que já foi bastante estudada.

O arquivo de entrada fornecido ao FLEX consiste de regras que são pares de expressão regulares e código C e como resultado ele gera o arquivo `lex.yy.c`, que contém o analisador léxico, nomeado `yylex`, várias tabelas usadas para classificação dos *tokens* [AHO86], e várias rotinas e macros auxiliares. O formato geral consiste de três partes que são apresentadas a seguir.

```
Definições
%%
Regras
%%
Código do usuário
```

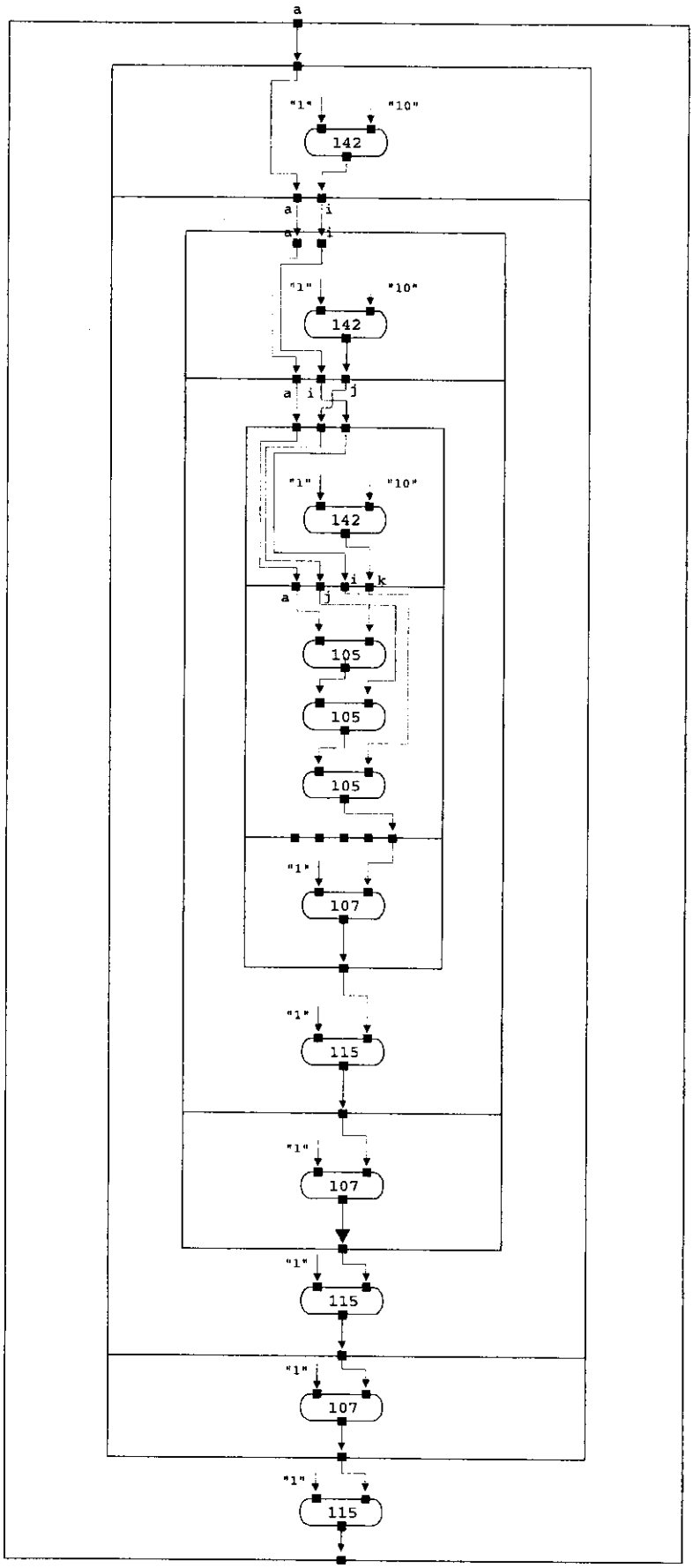


Figura 24. Representação geral do nó composto *forall* de IF1

Abaixo temos trechos que exemplificam as regras que compõe o arquivo de entrada para o FLEX.

```
[0-9][0-9]*   {  yylval.value = atoi(yytext);
                 return(INTEGER);
              }
```

Figura 25. Trecho do arquivo de entrada do FLEX

No trecho acima temos a expressão regular, `[0-9][0-9]*`, que define números inteiros positivos em IF1 e entre as chaves o código C que será executado caso a cadeia de entrada do autômato seja reconhecido por essa expressão. A variável `yytext`, do tipo *string*, definida pelo FLEX, contém o texto do *token* atual.

Com relação ao analisador gerado, ele pode ser usado isoladamente para a análise léxica de algum arquivo IF1, ou como no nosso caso ele será utilizado em conjunto com o analisador sintático, descrito na seção 5.2.2, abaixo.

Detalhes de como escrever expressões regulares com regras do FLEX podem ser obtidos no manual *Flex: A Fast Scanner Generator* [PAX95] disponível no site da *Free Software Foundation Inc.* Também é necessário o conhecimento da notação formal de expressões regulares para entender o que é definido numa linguagem. Um conhecimento básico deste assunto pode ser obtido no livro de compiladores de Aho e Sethi [AHO86], *Compilers: Principles, Techniques, and Tools.*

No Apêndice C, são apresentadas todas as regras do arquivo de entrada do FLEX que definem as normas léxicas de IF1 utilizadas pela nossa aplicação.

5.2.2 BISON

O BISON é um gerador de analisador sintático de propósito geral, disponível gratuitamente, que converte uma descrição de uma gramática livre de contexto LALR(1) [AHO86] para um programa C que analise programas escritos seguindo essa gramática. Como uma gramática formal é uma construção matemática, para definir a linguagem para o BISON, devemos em geral reescrevê-la segundo a sintaxe definida no BISON.

O processo para a partir da especificação de uma linguagem e usando o BISON, construir um compilador ou parte dele, consiste das seguintes etapas: especificação das regras gramaticais segundo as regras estabelecidas no BISON; escrita de um analisador

léxico para processar a entrada e passar *tokens* para o analisador sintático; escrita de uma função de controle que chame o analisador sintático produzido pelo BISON e escrita de rotinas que relatem possíveis erros.

O arquivo de entrada para o utilitário BISON é um arquivo texto que contém códigos segundo a gramática BISON estabelecida em [DON95], cuja forma geral é exibida a seguir.

```
%{
Declarações do C
%}
Declarações do BISON
%%
Regras Gramaticais
%%
Código C Adicional
```

Como exemplo da especificação das regras gramaticais em BISON, vejamos o trecho abaixo na **Figura 26**, para a regra de produção que termina a definição de um nó composto de IF1, `Line ::= } Label Node Count Assoc`. Poderemos ver que para cada regra gramatical pode estar associada uma ação (conjunto de expressões em C) a ser tomada quando esta regra for reconhecida.

No trecho de código bison da **Figura 26** temos uma regra, entre as chaves a ação em código C e o ponto e vírgula indica o final das ações associadas. Esse trecho de código constrói uma parte da estrutura de dados que armazena o grafo IF1 na memória. Para as demais regras, o mesmo procedimento é aplicado, visando a montagem do grafo na memória.

Apesar da montagem de um autômato que reconhecesse a linguagem definida pela gramática de IF1 não ser extremamente difícil, a utilização desta ferramenta diminuiu o tempo para o desenvolvimento de nosso módulo para as análises visando a otimização para sistemas de memória distribuída.

```
BRACEC label nodecmpd count assoclist {
   lntype = $1;
   popn(&nodei, &nodef);
   nodef->gi = init;
   nodef->ge = end;
```

```
        popg(&init, &end);

        insnode(&nodei, &nodef, $1, $2, $3, assoc, "");

        end->nodei = nodei;
        end->nodef = nodef;
        edge = 'N';
        edgsi = NULL;
        edgsf = NULL;
        free(assoc);
    }
```

Figura 26. Trecho do arquivo de entrada BISON

A listagem completa das regras gramaticais da linguagem IF1 no formato exigido pelo BISON pode ser vista no Apêndice D e as regras estabelecidas pelo BISON pode ser encontrada no manual disponibilizado pela *Free Software Foundation Inc.*, Bison: *The YACC-compatible Parser Generator* [DON95].

5.3. INTEGRAÇÃO COM O OSC

Como citado acima, no início deste Capítulo, o *front-end* do OSC retorna arquivos de programas IF1, representando programas SISAL. O *back-end* do OSC toma esses arquivos como entrada e realiza as análises a que foi projetado, gerando no final o código objeto, a menos que alguma diretiva de compilação lhe seja fornecida.

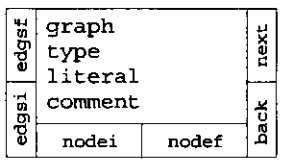
A nossa opção de estruturação das otimizações, tentando aproveitar as análises para computadores de memória compartilhada e local existente no OSC, nos levou a inserir nossas análises logo após o *front-end* ter gerado os arquivos IF1, ou seja, no início do *back-end*.

Para isso alteramos dois pontos do compilador OSC, um nos arquivos `osc.c` e `options.h` que contém declarações de variáveis e constantes e onde são chamadas as rotinas tanto do *front-end* como *back-end* e toda preparação relacionada a essas chamadas e `if2preamble.c` onde foram inseridas alterações relacionadas com a biblioteca *MPI*. Nestes arquivos basicamente inserimos uma fase para a chamada da nossa otimização, nomeada *if1mpi*. Assim como nas outras opções já existentes, também disponibilizamos a possibilidade de passar uma diretiva que diz ao compilador para parar a otimização após as análises *if1mpi*. Todos os novos códigos inseridos estão seguidos de comentários da data de sua inserção.

A outra alteração é a criação de uma nova etapa no *back-end*, que inclui as técnicas de otimizações para memória distribuídas discutidas no Capítulo 4. Essas otimizações são ativadas pelas alterações realizadas no `osc.c`. Mais abaixo, nas outras seções discutiremos como foram implementadas cada uma dessas otimizações. No apêndice F apresentamos o novo esquema do compilador OSC após a inserção das alterações acima citadas.

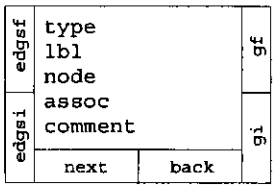
5.4. ESTRUTURA UTILIZADA

A escolha da estrutura utilizada para o armazenamento do grafo de um arquivo IF1 foi sem dúvida um dos pontos mais delicados da implementação das otimizações propostas neste trabalho, visto que todas as análises a serem realizadas fazem uso da estrutura de armazenamento gerada e, por esta razão, a elaboração exigiu uma boa parte do tempo de análise do projeto. Após diversos estudos para a estruturação, chegamos a uma versão que fielmente representa a sintaxe dos grafos IF1 e facilita a realização das análises. A representação resultante para grafos, nós e arcos de IF1 é exibida abaixo nas **Figuras 27, 28, 29** e **30.**




```
typedef struct graph{
   char graph, *literal, *comment;
   int type;
   pedges edgsi, edgsf;
   pnodes nodei, nodef;
   struct graph *next, *back;
} *pgraph;
```

Figura 27. Estrutura para armazenar o grafo IF1

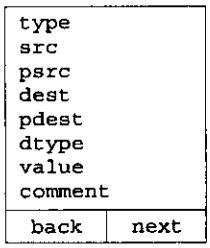



```
typedef struct nodes {
  char type, *comment;
  int lbl, node, *assoc;
  pedges edgsi, edgsf;
  struct graph **gi, *ge;
  struct nodes *next, *back;
} *pnodes;
```

Figura 28. Estrutura para armazenar o nó IF1 (simples e composto)

Para o armazenamento efetivo do arquivo IF1 nas estruturas acima listadas, escrevemos uma série de rotinas para a manipulação de listas não lineares duplamente encadeadas, entre elas temos rotinas de inserção, remoção, ordenação, busca, entre outras.

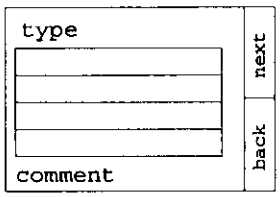
No armazenamento da tabela de tipos do IF1 utilizamos rotinas que manipulam listas lineares duplamente encadeadas.




```
typedef struct edges {
  char type, *value, *comment;
  int src, psrc, dest, pdest, dtype;
  struct edges *next, *back;
} *pedges;
```

Figura 29. Estrutura para armazenar o arco IF1

E para armazenar os tipos utilizamos a seguinte estrutura.




```
typedef struct typetbl {
   int type[4];
   char *comment;
   struct typetbl *next, *back;
} *ptypetbl;
```

Figura 30. Estrutura para armazenar tipos de IF1 (simples e estruturados)

No Apêndice E apresentamos uma ilustração do armazenamento de um código IF1 simples.

5.5 ALGORITMO GERAL

Nesta seção descrevemos o algoritmo principal para o módulo que foi desenvolvido neste trabalho, que pode ser resumido basicamente como percorrer os grafos de uma aplicação e a cada função encontrada, procurar nós *forall* e, se possível, aplicar as propostas de otimização apresentadas no Capítulo 4. Assim, o que apresentamos abaixo é apenas o fluxo geral de como são realizadas as análises, sem, entretanto apresentar detalhes delas.

```
algoritmo
   enquanto não percorrer o grafo todo faça
```

```
      para cada subgrafo representando uma função
         - Alterar o código para o formato SPMD mestre/escravo
         enquanto não percorrer a função todo faça
             para cada nó forall encontrado
                  - Alinhar dados/computação do nó forall
                  - Selecionar os dados e computações a distribuir
                  - Mapear dados/computações nos processadores
                    físicos
                  - Inserir as rotinas para a distribuição de
                    dados e computações
         fim enquanto
   fim enquanto
fim algoritmo
```

Apesar de aparentemente simples, o algoritmo acima na realidade torna-se bastante complexo e extenso; assim, os detalhes de implementação para as otimizações propostas serão apresentados nas seções seguintes.

5.6. ALINHAMENTO

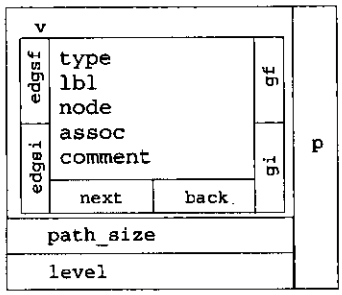
5.6.1 EXTRAÇÃO DE FUNÇÕES AFINS

A extração das funções afins dos *array* e *aninhamentos* de *forall* nos código IF1 é uma etapa fundamental antes da aplicação das otimizações para o alinhamento de dados e divisão de computações e dados.

Ao percorrermos os *loops forall* em busca de operações onde o retorno do tipo *array* envolvia operações com dados também do tipo *array*, utilizamos duas estruturas principais para armazenar as informações necessárias nos algoritmos. Uma delas é basicamente uma estrutura que permite armazenar todos os aninhamentos de *loops* existentes, informação esta que será utilizada para a extração das funções afins relacionadas a cada retorno. Abaixo apresentamos as estruturas.

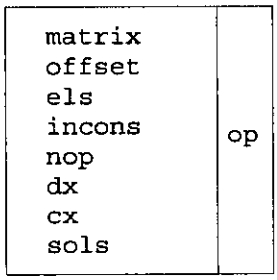
O algoritmo geral da rotina elaborada para realizar a tarefa de extração das funções é apresentado a seguir. Basicamente, percorremos as saídas do grafo de retorno de um nó

forall, em busca de entradas que sejam *arrays* e que foram geradas por operações sobre *arrays*.




```
typedef struct path {
   pnodes v;
   int path_size;
   int level;
   struct path **p;
} *ppath;
```

Figura 31. Estrutura utilizada para armazenar aninhamentos de *loops*




```
typedef struct affine {
   int **matrix,
       *offset,
       els,
       incons,
       nop,
       **dx,
       **cx,
       sols;
   struct affine *op;
} afine;
```

Figura 32. Estrutura utilizada para armazenar as funções afins

```
algoritmo para alinhamento
   enquanto não percorrer o aninhamento de foralls todo faça
      para cada loop forall encontrado faça
         - inserir o forall na árvore para aninhamentos
      enquanto não percorrer toda a árvore de aninhamentos faça
          para cada arco de saída do forall que seja do tipo array
             - inserir a função de acesso do resultado no nó pai
               da árvore para funções afim
             - para cada resultado do forall inserido na árvore
               inserir os operandos associados à saída
     fim enquanto

     enquanto não percorrer toda a árvore de funções afins faça
          para cada arco de saída do forall que seja do tipo array
             - se possível encontrar a matriz de transformação
```

```
                linear para cada array resultante do forall.
      fim enquanto
    fim enquanto
fim algoritmo
```

A extração desta informação se mostrou bastante trabalhosa, em função da estrutura hierárquica dos grafos IF1, o que exigiu a elaboração de um algoritmo que mantivesse os estados da execução nas mudanças da hierarquia mais externa para a mais interna dentro do aninhamento, visto que estes seriam necessários no retorno para a hierarquia mais externa.

Para ilustrar o algoritmo acima exibimos na **Figura 33** como ficaria o armazenado um *aninhamento* de *foralls* na estrutura proposta. Na **Figura 34**, seguinte, apresentamos a ilustração de como ficam armazenados os acessos aos *arrays* do código SISAL apresentado. Pela ilustração podemos observar que devemos percorrer a árvore para obter a matriz de acesso a esses *arrays*.

5.6.2 Soluções do sistema de Equações

Após a extração das funções afins, o passo que segue é a procura de uma transformação linear que alinhe os acessos aos *arrays*. Para isto, utilizamos o método de Gauss-Jordan [MEY00] para transformar a matriz afim para o formato escada e em seguida verificamos a consistência do sistema resultante.

5.7. DIVISÃO

A fase da divisão, que vem em seguida ao alinhamento, é implementada como a busca por todas as soluções possíveis aos sistemas apresentados na seção 4.2, cujos sistemas indicam quais os dados e computações devem ser colocados juntos. Assim, primeiro escolhemos qual *loop* do aninhamento de *loops* queremos distribuir e quais manteremos juntos. Neste trabalho, optamos por dividir o *loop* mais externo do aninhamento de loops e a primeira dimensão do array. Caso não seja possível, o segundo *loop* mais externo do aninhamento é o escolhido, bem como a dimensão seguinte do *array*. Esse procedimento segue até que tenhamos uma escolha válida.

```
①... for i in 1, 100
②     ... for j in 1, 100
③         ... for k in 1, 100
              ...
              end for
④         ... for k in 1, 100
              ...
              end for
          end for
⑤     ... for j in 1, 100
⑥         ... for k in 1, 100
              ...
              end for
          end for
      end for
```

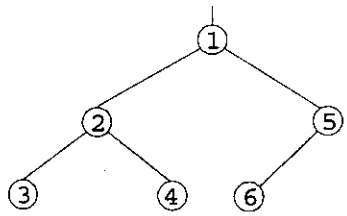


Figura 33. Representação de como fica armazenado um aninhamento de *loops*

```
a := for i in 1, 100 cross j in 1, 100 cross k in 1, 100
     returns array of b[i,k,j]+c[j+1,i+2,2*k]
     end for
```

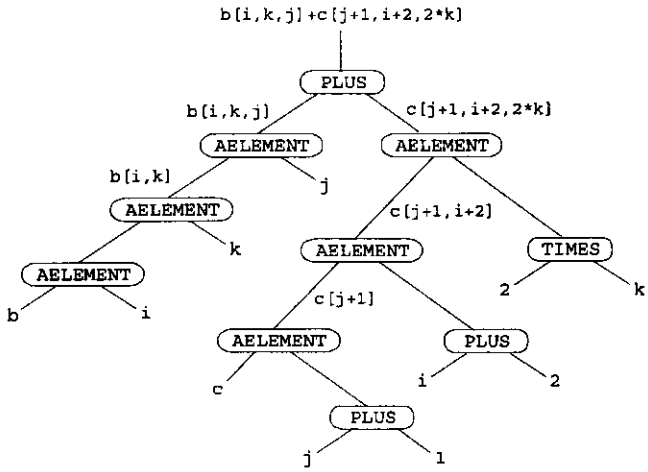


Figura 34. Representação de como fica armazenado o acesso a *array* (em IF1)

Para cada *loop* escolhido, verificamos todas as soluções possíveis para o sistema $D_x F^k_{xj} = C_j$, que permite verificar quais dados deverão ser distribuídos e quais mantidos locais. A implementação consiste em montar uma árvore com as possíveis soluções para C_j, no nó pai e D_x nas folhas.

O algoritmo que busca a transformação linear é apresentado a seguir e consiste em gerar e testar as possíveis soluções para o sistema. Caso a solução seja encontrada, ela é armazenada na árvore de funções afins.

```
algoritmo
   para cada nó pai da árvore de funções afins faça
      para cada loop do aninhamento de loops faça
         - selecionar qual aninhamento deixar local
         - para cada loop selecionado faça
               - gerar as possíveis soluções C_j para o sistema
                 C_j(ē_q) = 0
               - se a solução C_j for válida
                     então armazenar C_j numa matriz de resultados
                 fim se
           fim para

         - para cada um dos C_j armazenados faça
               - para todo nó da árvore de funções afins faça
               - gerar as possíveis soluções D_x para o sistema
                 D_x F^k_xj = C_j
               - se a solução D_x for válida
                     então armazenar D_x na matriz de resultados do
                           nó pai da função afim correspondente
                           (lado esquerdo)
                 fim se
           fim para
      fim para
   fim para
fim algoritmo
```

Abaixo apresentamos como exemplo do algoritmo descrito acima a busca por soluções para um aninhamento de *loops*. Ilustramos na **Figura 35** somente a solução para a opção de deixar o terceiro aninhamento local ao processador; entretanto, no algoritmo são buscadas soluções mantendo-se local cada um dos três aninhamentos de *loops*.

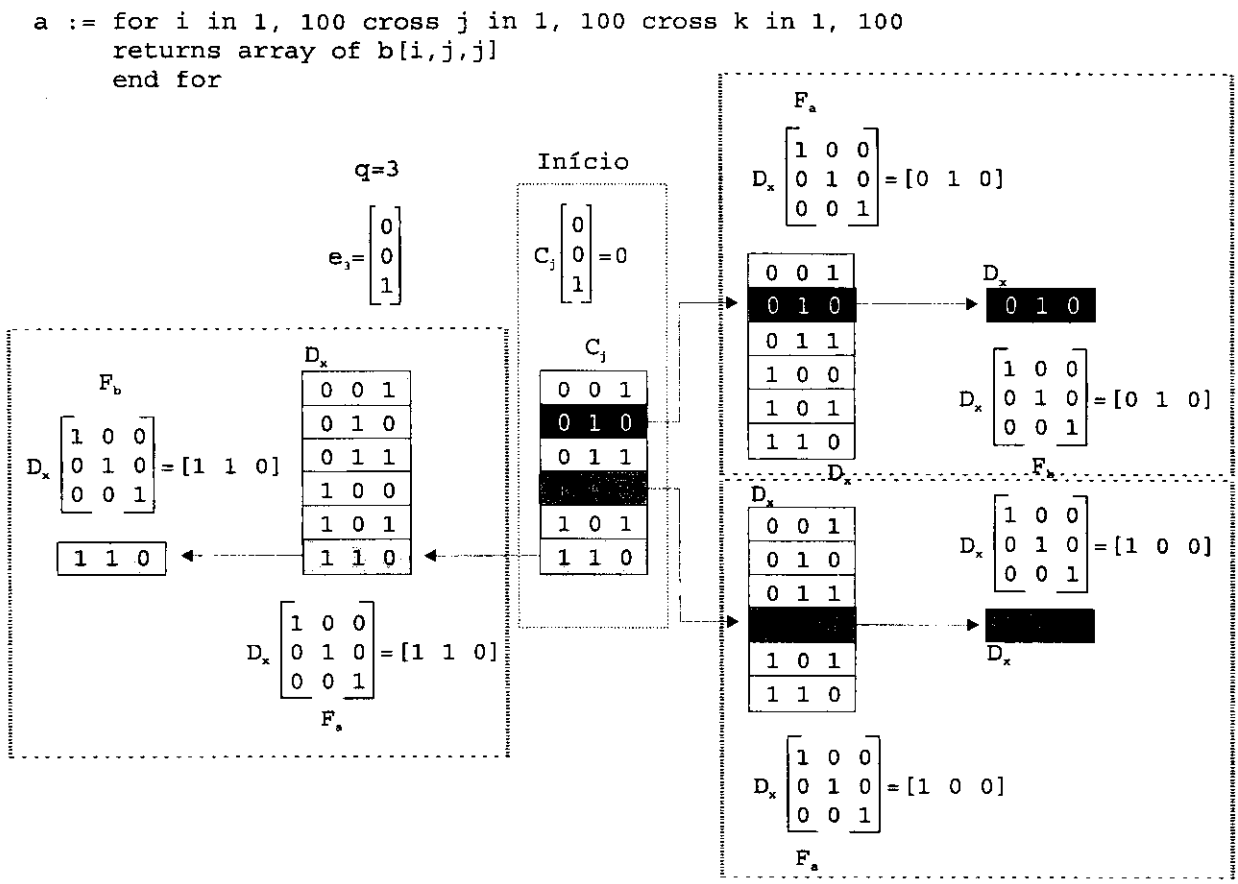


Figura 35. Exemplo de uma solução de mapeamento de dados e computações. Os quadrados pintados representam as possíveis soluções para o sistema

5.8. MAPEAMENTO

Após a aplicação dos métodos de alinhamento e divisão, temos disponíveis informações que nos indicam se o mapeamento pode ser realizado ou não, isto é se existe ou não a possibilidade de minimizar os efeitos da latência com comunicação de dados de forma que seja interessante explorar o paralelismo.

Assim, da(s) solução(ões) obtida(s) podemos selecionar qual o melhor mapeamento a se realizar. Para tanto realizamos dois tipos de alterações no código IF1 original. Um deles será a alteração dos intervalos de iteração dos *loops forall*, ou seja, a alteração dos limites inferior e superior do(s) *loop(s)* escolhido(s) para ser mapeado entre os processadores. As alterações aos índices são realizadas percorrendo-se a árvore que aponta para os aninhamentos de *loops*, até encontrar o aninhamento escolhido para o mapeamento. A outra alteração, a comunicação de dados, é realizada inserindo-se chamadas a funções

MPI, conforme definido em [NAK96]. Essa transmissão leva em conta a seleção realizada na fase de divisão.

Para ilustrar a fase de mapeamento suponha o exemplo da **Figura 35**, onde obtivemos pela análise de divisão, três possíveis soluções: $C_j = [1 \quad 0 \quad 0]$ e $D = [1 \quad 0 \quad 0]$; $C_j = [0 \quad 1 \quad 0]$ e $D = [0 \quad 1 \quad 0]$; e $C_j = [1 \quad 1 \quad 0]$ e $D = [1 \quad 1 \quad 0]$. Caso escolhamos a primeira solução para a divisão, temos então

$$C_j \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 0 \qquad C_j = [1 \quad 0 \quad 0] \qquad D = [1 \quad 0 \quad 0]$$

```
for i in 1, 50 cross j in 1, 100 cross k in 1, 100
returns array of b[i,j,j]
end for

for i in 51, 100 cross j in 1, 100 cross k in 1, 100
returns array of b[i,j,j]
end for
```

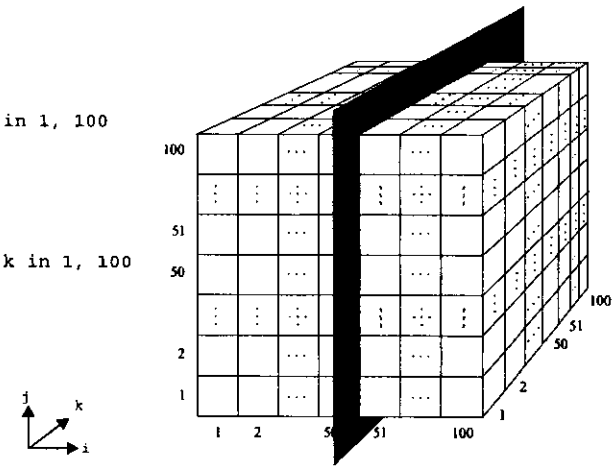


Figura 36. Ilustração da solução de divisão de dados e computações da Figura 35

Vemos por este exemplo que devemos então alterar os índices do *forall* mais externo e distribuir os dados no sentido da primeira dimensão da matriz, pela **Figura 36**, no sentido de `i`. Esta ilustração não apresenta, entretanto o código para a distribuição dos dados, que envolve chamadas a rotinas MPI. A descrição de tais chamadas será realizada a seguir, na seção 5.9.

5.9. GERAÇÃO DE CÓDIGO IF1 COM MPI

Após realizar o alinhamento, a divisão e o mapeamento, a etapa seguinte deve realizar as alterações no código *IF1* que reflitam as análises de otimização realizadas, ou

seja, alterações dos limites dos *loops forall* nos processos mestre e escravos, a distribuição dos dados necessários a cada processo, segundo as análises realizadas.

A forma como realizamos a alteração dos limites dos *loops forall* e a distribuição dos dados exigiram a utilização de rotinas de passagem de mensagem, e para este trabalho utilizamos o padrão MPI.

Diversas questões devem ser tratadas para que seja possível a utilização das rotinas MPI no código de um programa *SISAL*. No entanto, todas elas já foram discutidas em [NAK96] e, portanto apresentaremos somente a utilização das rotinas MPI segundo o que foi necessário para realizar a geração do código com as otimizações.

Temos somente o seguinte comentário com relação à notação que foi utilizada no trabalho anterior apresentado em [NAK96] utilizamos o prefixo sisal_mpi_ para representar uma conversão de C para SISAL. Agora abreviamos a notação para _sm_, apenas para abreviar os nomes utilizados.

Posto isto, passamos agora a detalhar como foi realizada a distribuição dos *arrays* e dos dados necessários para o cálculo dos novos índices dos *loops forall* usando rotinas MPI para comunicação ponto a ponto e coletiva.

Pelo exemplo da **Figura 36** o *loop forall* original foi dividido em duas partes, com iterações em `i` de `1` a `50` e `51` a `100` e `j` e `k` de `1` a `100`. Os dados necessários a cada aninhamento de *loops*, com a distribuição pelo primeiro índice do array será `b[1..50,1..100,1..100]` e `b[51..100,1..100,1..100]`.

5.10. CONCLUSÕES

Neste Capítulo descrevemos os algoritmos e as estruturas utilizadas para a implementação das análises estáticas dos códigos SISAL visando a otimização da distribuição de dados e computações. Conforme apresentado no Capítulo 4 anterior, os métodos para alinhamento e decomposição de dados e computações estão fundamentados na Álgebra Linear e na Programação Linear, e conforme previsto pudemos nos abstrair das diversas análises que teriam que ser realizadas e nos concentramos na elaboração de algoritmos que extraíssem e manipulassem as informações necessárias, por exemplo, as funções afins.

Apesar da abstração com relação a análise para distribuição, a fase para a extração das informações dos *loops forall* foi bastante trabalhosa e exigiu uma atenção especial,

pois todo o restante do desenvolvimento dependeria dela. Realizada a fase de extração das funções afins, a etapa seguinte foi implementar dos métodos matemáticos apresentados.

Com base nas propostas matemáticas realizamos a elaboração e implementação dos algoritmos para alinhamento, divisão e mapeamento. O que pudemos observar claramente durante o desenvolvimento é que como parte do problema já havia sido analisada, a elaboração do algoritmo deveria tentar garantir a implementação da proposta, reduzindo assim a quantia de problemas a tratar.

6. RESULTADOS

A seguir apresentaremos os resultados obtidos dos métodos propostos no Capítulo 4 e a implementação descrita no Capítulo 5. Utilizaremos programas em SISAL ao invés de IF1 visto que o entendimento para aqueles que não conhecem a linguagem é praticamente inviável. Outro fator é a extensão do código gerado que também seria um problema. Posto isto, apresentaremos a seguir programas que ilustrem casos em que as análises serão aplicadas. Os programas apresentados estão em conformidade com os resultados apresentados no trabalho de mestrado sob o título "Paralelização de programas SISAL para sistemas MPI", [NAK96]. Nesse trabalho foi testada proposta e validada uma forma para a utilização de chamadas a rotinas MPI em código de programas SISAL. Assim, o código SISAL apresentado a seguir utiliza essa mesma proposta para a utilização de chamadas a rotinas MPI.

Devemos lembrar que implementamos no compilador as fases de análises propostas, alinhamento, divisão e mapeamento e a fase de geração de código ficou para um segundo momento do trabalho, já em fase de andamento.

6.1. CÓDIGO GERADO

Para apresentarmos o código que deveria ser gerado ao final da análise vejamos o programa da **Figura 37** abaixo onde temos um *loop forall* com acessos a um *array* `b` pelos índices `[i,k,j]` e `[j,i,k]` e a criação de um *array* `a` pelo índice `[i,j,k]`.

Iniciaremos a discussão da implementação pelas funções de acesso dos *array* extraídas atualmente pelo módulo inserido no compilador OSC

```
define main

function main(b: array[array[array[integer]]]
              returns array[array[array[integer]]])

    for i in 1, 400 cross j in 1, 400 cross k in 1, 400
    returns array of b[i,k,j] * b[j,i,k]
    end for

end function
```

$$l_{a1}^{1}(\vec{a}) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad r_{b1}^{1}(\vec{a}) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad r_{b1}^{2}(\vec{a}) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Figura 37. Exemplo de um código SISAL e as funções de acesso aos arrays

Podemos observar pelas funções afins do *array* b que os acessos são distintos daqueles do *array* a que está sendo criado e, portanto, o alinhamento é necessário neste caso. Com a implementação realizada para o alinhamento obteremos as seguintes transformações lineares apresentadas na **Figura 38**.

$$T_{b1}^{1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \ \vec{t}_{b1}^{1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{e} \qquad T_{b1}^{2} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \ \vec{t}_{b1}^{2} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Figura 38. Alinhamento para os *arrays* do programa da Figura 37

Este resultado indica que podemos realizar a transformação para o código analisado ou mais especificamente gerar o código apresentado na **Figura 39**.

```
define main

function main(b: array[array[array[integer]]]
              returns array[array[array[integer]]])
let

   sm_b1, sm_b2 :=
   for i in 1, 400 cross j in 1, 400 cross k in 1, 400
   returns array of b[i,k,j]
           array of b[j,i,k]
   end for

in

   for i in 1, 400 cross j in 1, 400 cross k in 1, 400
   returns array of sm_b1[i,j,k] * sm_b2[i,j,k]
   end for

end let
end function
```

Figura 39. Alinhamento dos acessos para o programa da Figura 37

Podemos observar que pelo código gerado estamos criando novos *arrays* com o padrão de acesso desejado, `[i,j,k]`, a transformação linear não foi aplicada ao código. Na

realidade, o que a transformação linear indica é que podemos realizar s mudança de base nos acessos ao *array* b, ou seja, podemos mudar o acesso de b para que siga o mesmo padrão de a. Esta mudança sob o ponto de vista computacional consiste basicamente em copiar ou mover os dados que estavam sob um padrão e passá-los ao padrão desejado. No nosso caso optamos por criar um novo *array*.

Realizado o alinhamento, o passo seguinte verifica quais dados e computações serão distribuídas. Visto que conseguimos realizar o alinhamento dos dados, na divisão dispomos de uma ampla possibilidade de granularidade. Vejamos o resultado obtido pelo módulo de análise para divisão.

$$C_1(\vec{e}_2) = 0 \qquad C_1(\vec{e}_3) = 0 \qquad \vec{e}_2 = [0 \;\; 1 \;\; 0]' \qquad \vec{e}_3 = [0 \;\; 0 \;\; 1]'$$

$$D_1 F^1_{a1} = C_1 \qquad D_1 F^1_{(sm_b1)1} = C_1 \qquad D_1 F^1_{(sm_b2)1} = C_1$$

$$D_1 = C_1 = [1 \;\; 0 \;\; 0]$$

Figura 40. Resultado obtido da análise para divisão do programa da Figura 39

O resultado obtido da análise de divisão, $D_1 = C_1 = [1 \;\; 0 \;\; 0]$, indica que podemos dividir os arrays pelas primeiras dimensões e o *loop* mais externo do aninhamento. Este resultado é então passado ao módulo de mapeamento que deveria gerar o seguinte código *spmd*.

```
%$c=sm_rank
%$c=sm_size
%$c=sm_sendAI
%$c=sm_recvAI
%$c=sm_scatterI
%$c=sm_bcastAI

define main

global sm_rank(insm_dep: integer returns integer, integer)
global sm_size(insm_dep: integer returns integer, integer)
global sm_sendAI(buf:array[integer]; cnt, dest, tg, in_dep: integer;
                 vd: array[integer] returns integer, array[integer])
global sm_recvAI(cnt, dest, tg, in_dep: integer; vd: array[integer]
                 returns integer, array[integer])
global sm_scatterI(buf:array[integer]; in_dep: integer
                   returns integer, integer)
global sm_bcastAI(buf:array[integer]; cnt, in_dep: integer
                  returns integer, array[integer])

function
main(b:array[array[array[integer]]] returns array[array[array[integer]]])
let
   sm_1, rank := sm_rank(0);
   sm_2, sz := sm_size(sm_1);

   sm_a :=
   if (rank = 0)  then
   let
      sm_b1 , sm_b2 :=
```

```
    for i in 1, 400 cross j in 1, 400 cross k in 1, 400
    returns array of b[i,k,j]
            array of b[j,i,k]
    end for;

    sm_L := array[0:   1, 101, 201, 301];
    sm_H := array[0: 100, 200, 300, 400];

    sm_3, sm_lx := sm_scatterI(sm_L, sm_2);
    sm_4, sm_hx := sm_scatterI(sm_H, sm_3);

    vd := array[0: 0, 1, 1, 0, 399, 1, 400, 1];
    sm_5, vd1 := sm_bcastAI(vd, 8, vd[0]);

    sm_b3 := for x in 1, sz-1  cross i in sm_L[x], sm_H[x] cross j in 1, 400
             sm_6, sm_b1 := sm_sendAI(sm_b1[i,j], 400, x, j, sm_5, vd)
             returns array of sm_b1
             end for;

    sm_b4 := for x in 1, sz-1  cross i in sm_L[x], sm_H[x] cross j in 1, 400
             sm_6, sm_b2 := sm_sendAI(sm_b2[i,j], 400, x, j, sm_5, vd)
             returns array of sm_b2
             end for;

    a :=
    for i in sm_L[0], sm_H[0] cross j in 1, 400 cross k in 1, 400
    returns array of sm_b1[i,j,k]*sm_b2[i,j,k]
    end for;

    sm_a :=
    for x in 1, sz-1
       sm_a :=
       for i in sm_L[x], sm_H[x] cross j in 1, 400
          sm_6, sm_a := sm_recvAI(400, x, j, sm_5, vd)
       returns array of sm_a
       end for
    returns value of a || sm_a
    end for;
in
  sm_a
end let
else
let
    sm_L := array[0:0];
    sm_H := array[0:0];

    sm_3, sm_lx := sm_scatterI(sm_L, 0);
    sm_4, sm_hx := sm_scatterI(sm_H, sm_3);

    vd := array[0: 0];
    sm_5, vd1 := sm_bcastAI(vd, 8, vd[0]);

    sm_b1 :=
    for i in sm_lx, sm_hx cross j in 1, 400
       sm_6, sm_b1 := sm_recvAI(400, 0, j, sm_5, vd1)
    returns array of sm_b1
    end for;

    sm_b2 :=
    for i in sm_lx, sm_hx cross j in 1, 400
       sm_6, sm_b2 := sm_recvAI(400, 0, j, sm_5, vd1)
    returns array of sm_b2
    end for;

    a :=
    for i in sm_lx, sm_hx cross j in 1, 400 cross k in 1, 400
    returns array of sm_b1[i,j,k]*sm_b2[i,j,k]
    end for;
```

```
      sm_a :=
      for i in sm_lx, sm_hx cross j in 1, 400
         sm_6, sm_a := sm_sendAI(a[i,j], 0, 400, j, sm_5, vd)
      returns array of sm_a
      end for;
   in
      sm_a
   end let
   end if
in
   sm_a
end let
end function
```

Figura 41. Resultado obtido da análise para divisão do programa da Figura 39

Vemos no código apresentado acima o uso de várias funções `sm_sendAI`, `sm_recvAI`, `sm_scatterI`, `sm_bcastAI`, `sm_rank` e `sm_size` que são na realidade a interface para chamadas a funções MPI. O interfaceamento entre C e SISAL já foi discutido em [NAK96] por esta razão não nos prolongaremos neste assunto. Os índices para os mapeamentos, gerados com base na formulação apresentada no Capítulo 4, equação (11), estão armazenados em `sm_L` e `sm_H`. Os mesmos índices poderão ser usados para mapear os *loops* visto que tivemos sucesso no alinhamento dos dados.

Abstraindo os detalhes do código gerado, vemos que no processo mestre, *rank* zero, temos o envio dos dados necessários aos processos remotos, a realização da computação e a coleta de dados remotos resultantes dos processamentos dos escravos. Nos processos escravos vemos o recebimento de dados enviados pelo processo mestre, a realização de computações e o envio ao mestre dos resultados obtidos.

6.2. ESTUDO DE CASOS

A seguir apresentamos alguns exemplos de programas *SISAL* e os códigos que deveriam ser gerados após as análises propostas e as partes até agora implementadas.

Para não estendermos além do necessário este capítulo, para os casos abaixo apresentaremos se possível a transformação linear, as matrizes de decomposição e o código resultante.

6.2.1. Caso 1

Exemplo de um programa que contém um aninhamento de *loops* no qual são realizados acessos aos *arrays* com a indexação com deslocamento.

```
define main

function main(b, c: array[array[array[integer]]]
              returns array[array[array[integer]]])

   for i in 1, 400 cross j in 1, 400 cross k in 1, 400
   returns array of b[i+j,j+k,k] + c[i,i+j,i+j+k]
   end for

end function
```

$$l^1_{a1}(\vec{a}) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad r^1_{b1}(\vec{a}) = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad r^1_{c1}(\vec{a}) = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Figura 42. Exemplo de um código SISAL e as funções de acesso aos arrays

$$T^1_{b1} = \begin{pmatrix} 1 & -1 & 1 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix}, \ \vec{t}^1_{b1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{e} \qquad T^1_{c1} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}, \ \vec{t}^1_{c1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Figura 43. Alinhamento para os *arrays* do programa da Figura 42

```
define main

function main(b: array[array[array[integer]]]
              returns array[array[array[integer]]])
let

   sm_b1, sm_c1 :=
   for i in 1, 400 cross j in 1, 400 cross k in 1, 400
   returns array of b[i+j,j+k,j]
           array of b[i,i+j,i+j+k]
   end for

in
   for i in 1, 400 cross j in 1, 400 cross k in 1, 400
   returns array of sm_b1[i,j,k] * sm_c1[i,j,k]
   end for
end let
end function
```

Figura 44.. Alinhamento dos acessos para o programa da Figura 43

$$C_1(\vec{e}_2)=0 \qquad C_1(\vec{e}_3)=0 \qquad \vec{e}_2=[0\ \ 1\ \ 0]' \qquad \vec{e}_3=[0\ \ 0\ \ 1]'$$

$$D_1 F^1_{a1}=C_1 \qquad D_1 F^1_{(sm_b1)1}=C_1 \qquad D_1 F^1_{(sm_c1)1}=C_1$$

$$D_1=C_1=[1\ \ 0\ \ 0]$$

Figura 45. Resultado obtido da análise para divisão do programa da Figura 39

```
%$c=sm_rank
%$c=sm_size
%$c=sm_sendAI
%$c=sm_recvAI
%$c=sm_scatterI
%$c=sm_bcastAI

define main

global sm_rank(insm_dep: integer returns integer, integer)
global sm_size(insm_dep: integer returns integer, integer)
global sm_sendAI(buf:array[integer]; cnt, dest, tg, in_dep: integer;
                 vd: array[integer] returns integer, array[integer])
global sm_recvAI(cnt, dest, tg, in_dep: integer; vd: array[integer]
                 returns integer, array[integer])
global sm_scatterI(buf:array[integer]; in_dep: integer
                   returns integer, integer)
global sm_bcastAI(buf:array[integer]; cnt, in_dep: integer
                  returns integer, array[integer])


function main(b, c: array[array[array[integer]]]
              returns array[array[array[integer]]])
let
   sm_1, rank := sm_rank(0);
   sm_2, sz   := sm_size(sm_1);

   sm_a :=
   if (rank = 0)  then
   let
      sm_b1, sm_c1 :=
      for i in 1, 400 cross j in 1, 400 cross k in 1, 400
      returns array of b[i+j,j+k,k]
              array of c[i,i+j,i+j+k]
      end for;

      sm_L := array[0:   1, 101, 201, 301];
      sm_H := array[0: 100, 200, 300, 400];

      sm_3, sm_lx := sm_scatterI(sm_L, sm_2);
      sm_4, sm_hx := sm_scatterI(sm_H, sm_3);

      vd := array[0: 0, 1, 1, 0, 399, 1, 400, 1];
      sm_5, vd1 := sm_bcastAI(vd, 8, vd[0]);

      sm_b2, sm_c2 :=
      for x in 1, sz-1 cross i in sm_L[x], sm_H[x] cross j in 1, 400
         sm_6, sm_b := sm_sendAI(sm_b1[i,j], 400, x, j, sm_5, vd);
         sm_7, sm_c := sm_sendAI(sm_c1[i,j], 400, x, j, sm_6, vd);
      returns array of sm_b
              array of sm_c
```

```
      end for;

      a :=
      for i in sm_lx, sm_hx cross j in 1, 400 cross k in 1, 400
      returns array of sm_b1[i,j,k] + sm_c1[i,j,k]
      end for;

      sm_a :=
      for x in 1, sz-1
         sm_a :=
         for i in sm_L[x], sm_H[x] cross j in 1, 400
            sm_1, sm_a := sm_recvAI(400, x, j, sm_3, vd)
         returns array of sm_a
         end for
      returns value of a || sm_a
      end for;

   in
     sm_a
   end let
   else
   let
      sm_L := array[0:0];
      sm_H := array[0:0];

      sm_3, sm_lx := sm_scatterI(sm_L, 0);
      sm_4, sm_hx := sm_scatterI(sm_H, sm_1);

      vd := array[0: 0];
      sm_5, vd1 := sm_bcastAI(vd, 8, vd[0]);

      sm_b1, sm_c1 :=
      for i in sm_lx, sm_hx cross j in 1, 400
         sm_6, sm_b := sm_recvAI(400, 0, j, sm_5, vd1);
         sm_7, sm_c := sm_recvAI(400, 0, j, sm_6, vd1);
      returns array of sm_b
              array of sm_c
      end for;

      a :=
      for i in sm_lx, sm_hx cross j in 1, 400 cross k in 1, 400
      returns array of sm_b[i,j,k] + sm_c[i,j,k]
      end for;

      sm_a :=
      for i in sm_lx, sm_hx cross j in 1, 400
         sm_4, sm_a := sm_sendAI(a[i,j], 0, 400, j, sm_3, vd)
      returns array of sm_a
      end for;
   in
      sm_a
   end let
   end if
in
   sm_a
end let
end function
```

Figura 46. Resultado obtido da análise para divisão do programa da Figura 42

6.2.2. Caso 2

O programa apresentado a seguir é um caso onde existem múltiplos acessos a *arrays* tanto do lado direito como do lado esquerdo das expressões. Além da quantidade de *arrays* acessados vemos também uma variedade de acessos sem deslocamento.

```
define main

function main(d, c: array[array[array[integer]]] returns
              array[array[array[integer]]], array[array[array[integer]]])

   for i in 1, 400 cross j in 1, 400 cross k in 1, 400
     a := c[i,i+j,i+j+k] + d[i+k,j,j+k];
     b := c[i+j,j,k] + d[j,i,k];
   returns array of a
           array of b
   end for

end function
```

$$l^1_{a1}(\bar{a}) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad r^1_{c1}(\bar{a}) = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad r^2_{c1}(\bar{a}) = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$r^1_{d1}(\bar{a}) = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad r^2_{d1}(\bar{a}) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Figura 47. Exemplo de um código SISAL e as funções de acesso aos *arrays*

$$T^1_{c1} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}, \quad \bar{t}^1_{c1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad T^2_{c1} = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \bar{t}^2_{c1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$T^1_{d1} = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \bar{t}^1_{d1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad T^2_{d1} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \bar{t}^2_{d1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Figura 48. Alinhamento para os *arrays* do programa da Figura 47

```
define main

function main(d, c: array[array[array[integer]]] returns
              array[array[array[integer]]], array[array[array[integer]]])

let
```

```
   sm_c1, sm_d1, sm_c2, sm_d2 :=
   for i in 1, 400 cross j in 1, 400 cross k in 1, 400
   returns array of c[i,i+j,i+j+k]
           array of d[i+k,j,j+k]
           array of c[i+j,j,k]
           array of d[j,i,k]
   end for

in

   for i in 1, 400 cross j in 1, 400 cross k in 1, 400
     a := sm_c1[i,j,k] + sm_d1[i,j,k];
     b := sm_c2[i,j,k] + sm_d2[i,j,k];
   returns array of a
           array of b
   end for

end let
end function
```

Figura 49. Alinhamento dos acessos para o programa da Figura 47

$$C_1(\vec{e}_2) = 0 \qquad C_1(\vec{e}_3) = 0 \qquad \vec{e}_2 = [0 \;\; 1 \;\; 0]^t \qquad \vec{e}_3 = [0 \;\; 0 \;\; 1]^t$$

$$D_1 F^1_{a1} = C_1 \qquad D_1 F^1_{(sm_c1)1} = C_1 \qquad D_1 F^1_{(sm_d1)1} = C_1 \qquad D_1 F^1_{(sm_c2)1} = C_1 \qquad D_1 F^1_{(sm_d2)1} = C_1$$

$$D_1 = C_1 = [1 \;\; 0 \;\; 0]$$

Figura 50. Resultado obtido da análise para divisão do programa da Figura 47

```
%$c=sm_rank
%$c=sm_size
%$c=sm_sendAI
%$c=sm_recvAI
%$c=sm_scatterI
%$c=sm_bcastAI

define main

global sm_rank(insm_dep: integer returns integer, integer)
global sm_size(insm_dep: integer returns integer, integer)
global sm_sendAI(buf:array[integer]; cnt, dest, tg, in_dep: integer;
                 vd: array[integer] returns integer, array[integer])
global sm_recvAI(cnt, dest, tg, in_dep: integer; vd: array[integer]
                 returns integer, array[integer])
global sm_scatterI(buf:array[integer]; in_dep: integer
                   returns integer, integer)
global sm_bcastAI(buf:array[integer]; cnt, in_dep: integer
                  returns integer, array[integer])


function main(c, d: array[array[array[integer]]]  returns
              array[array[array[integer]]], array[array[array[integer]]])
let

   sm_1, rank := sm_rank(0);
```

```
sm_2, sz   := sm_size(sm_1);

sm_a, sm_b :=
if (rank = 0)  then
let

   sm_c1, sm_d1, sm_c2, sm_d2 :=
   for i in 1, 400 cross j in 1, 400 cross k in 1, 400
   returns array of c[i,i+j,i+j+k]
           array of d[i+k,j,k]
           array of c[i+j,j,k]
           array of d[j,i,k]
   end for;

   sm_L := array[0:   1, 101, 201, 301];
   sm_H := array[0: 100, 200, 300, 400];

   sm_3, sm_lx := sm_scatterI(sm_L, sm_2);
   sm_4, sm_hx := sm_scatterI(sm_H, sm_3);

   vd := array[0: 0, 1, 1, 0, 399, 1, 400, 1];
   sm_5, vd1 := sm_bcastAI(vd, 8, vd[0]);

   sm_c3, sm_d3, sm_c4, sm_d4 :=
   for x in 1, sz-1 cross i in sm_L[x], sm_H[x] cross j in 1, 400
      sm_6, sm_c3 := sm_sendAI(sm_c1[i,j], 400, x, j, sm_5, vd);
      sm_7, sm_d3 := sm_sendAI(sm_d1[i,j], 400, x, j, sm_6, vd);
      sm_8, sm_c4 := sm_sendAI(sm_c2[i,j], 400, x, j, sm_7, vd);
      sm_9, sm_d4 := sm_sendAI(sm_d2[i,j], 400, x, j, sm_8, vd);
   returns array of sm_c3
           array of sm_d3
           array of sm_c4
           array of sm_d4
   end for;

   a, b :=
   for i in sm_lx, sm_hx cross j in 1, 400 cross k in 1, 400
       a := sm_c1[i,j,k] + sm_d1[i,j,k];
       b := sm_c2[i,j,k] + sm_d2[i,j,k];
   returns array of a
           array of b
   end for;

   sm_a, sm_b :=
   for x in 1, sz-1
      sm_a, sm_b :=
      for i in sm_L[x], sm_H[x] cross j in 1, 400
         sm_6, sm_a := sm_recvAI(400, x, j, sm_5, vd);
         sm_7, sm_b := sm_recvAI(400, x, j, sm_6, vd);
      returns array of sm_a
              array of sm_b
      end for
   returns value of a || sm_a
           value of b || sm_b
   end for;

 in
   sm_a, sm_b
 end let
 else
 let
```

```
      sm_L := array[0:0];
      sm_H := array[0:0];

      sm_1, sm_lx := sm_scatterI(sm_L, 0);
      sm_2, sm_hx := sm_scatterI(sm_H, sm_1);

      vd := array[0: 0];
      sm_3, vd1 := sm_bcastAI(vd, 8, vd[0]);

      sm_c1, sm_d1, sm_c2, sm_d2 :=
      for i in sm_lx, sm_hx cross j in 1, 400
         sm_4, sm_c1 := sm_recvAI(400, 0, j, sm_3, vd1);
         sm_5, sm_d1 := sm_recvAI(400, 0, j, sm_4, vd1);
         sm_6, sm_c2 := sm_recvAI(400, 0, j, sm_5, vd1);
         sm_7, sm_d2 := sm_recvAI(400, 0, j, sm_6, vd1);
      returns array of sm_c1
              array of sm_d1
              array of sm_c2
              array of sm_d2
      end for;

      a,b :=
      for i in sm_lx, sm_hx cross j in 1, 400 cross k in 1, 400
        a := sm_c1[i,j,k] + sm_d1[i,j,k];
        b := sm_c2[i,j,k] + sm_d2[i,j,k];
      returns array of a
              array of b
      end for;

      sm_a, sm_b :=
      for i in sm_lx, sm_hx cross j in 1, 400
         sm_4, sm_a := sm_sendAI(a[i,j], 0, 400, j, sm_3, vd);
         sm_5, sm_b := sm_sendAI(a[i,j], 0, 400, j, sm_4, vd);
      returns array of sm_a
              array of sm_b
      end for;

   in
      sm_a, sm_b
   end let
   end if
in
   sm_a, sm_b
end let
end function
```

Figura 51. Resultado obtido da análise para divisão do programa da Figura 47

6.2.3. Caso 3

Programa para exemplificar um caso onde não existe uma transformação linear possível, mas a distribuição dos dados pode ser definida com as equações de comunicação.

```
define main

function main(d, c: array[array[array[integer]]] returns
              array[array[array[integer]]], array[array[array[integer]]])

   for i in 1, 400 cross j in 1, 400 cross k in 1, 400
   returns array of b[i,k,k]*c[i,j,j]
   end for

end function
```

$$l^1_{a1}(\vec{a}) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad r^1_{b1}(\vec{a}) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad r^1_{c1}(\vec{a}) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Figura 52. Exemplo de um código SISAL e as funções de acesso aos arrays

Pelo sistema proposto, o alinhamento dos arrays do programa da Figura 51 não é possível visto que os critérios de existência apresentados na seção 4.1 não são satisfeitos, portanto não temos uma transformação linear válida. Entretanto, pela análise de divisão podemos definir quais dados e computações deverão ser mantidas locais.

$$C_1(\vec{e}_2) = 0 \qquad C_1(\vec{e}_3) = 0 \qquad \vec{e}_2 = [0 \;\; 1 \;\; 0]^t \qquad \vec{e}_3 = [0 \;\; 0 \;\; 1]^t$$

$$D_1 F^1_{a1} = C_1 \qquad D_1 F^1_{b1} = C_1 \qquad D_1 F^1_{c1} = C_1$$

$$D_1 = C_1 = [1 \;\; 0 \;\; 0]$$

Figura 53. Resultado obtido da análise para divisão do programa da Figura 51

```
%$c=sm_rank
%$c=sm_size
%$c=sm_sendAI
%$c=sm_recvAI
%$c=sm_scatterI
%$c=sm_bcastAI

define main

global sm_rank(insm_dep: integer returns integer, integer)
global sm_size(insm_dep: integer returns integer, integer)
global sm_sendAI(buf:array[integer]; cnt, dest, tg, in_dep: integer;
                 vd: array[integer] returns integer, array[integer])
global sm_recvAI(cnt, dest, tg, in_dep: integer; vd: array[integer]
                returns integer, array[integer])
global sm_scatterI(buf:array[integer]; in_dep: integer
                  returns integer, integer)
global sm_bcastAI(buf:array[integer]; cnt, in_dep: integer
                 returns integer, array[integer])
```

```
function main(c, d: array[array[array[integer]]]  returns
              array[array[array[integer]]], array[array[array[integer]]])
let
   sm_1, rank := sm_rank(0);
   sm_2, sz   := sm_size(sm_1);

   sm_a, sm_b :=
   if (rank = 0)  then
   let

      sm_L := array[0:   1, 101, 201, 301];
      sm_H := array[0: 100, 200, 300, 400];

      sm_1, sm_lx := sm_scatterI(sm_L, sm_2);
      sm_2, sm_hx := sm_scatterI(sm_H, sm_3);

      vd := array[0: 0, 1, 1, 0, 399, 1, 400, 1];
      sm_3, vd1 := sm_bcastAI(vd, 8, vd[0]);

      sm_b1, sm_c1 :=
      for x in 1, sz-1 cross i in sm_L[x], sm_H[x] cross j in 1, 400
         sm_4, sm_b1 := sm_sendAI(a[i,j], 400, x, j, sm_3, vd);
         sm_5, sm_c1 := sm_sendAI(b[i,j], 400, x, j, sm_4, vd);
      returns array of sm_b1
              array of sm_c1
      end for;
      a :=
      for i in sm_lx, sm_hx cross j in 1, 400 cross k in 1, 400
      returns array of b[i,k,k]*c[i,j,j]
      end for;

      sm_a, sm_b :=
      for x in 1, sz-1
         sm_a, sm_b :=
         for i in sm_L[x], sm_H[x] cross j in 1, 400
            sm_4, sm_a := sm_recvAI(400, x, j, sm_3, vd);
         returns array of sm_a
         end for
      returns value of a || sm_a
      end for;

   in
     sm_a
   end let
   else
   let
      sm_L := array[0:0];
      sm_H := array[0:0];

      sm_1, sm_lx := sm_scatterI(sm_L, 0);
      sm_2, sm_hx := sm_scatterI(sm_H, sm_1);

      vd := array[0: 0];
      sm_3, vd1 := sm_bcastAI(vd, 8, vd[0]);

      sm_b1, sm_c1 :=
      for i in sm_lx, sm_hx cross j in 1, 400
         sm_4, sm_b1 := sm_recvAI(400, 0, j, sm_3, vd1);
         sm_5, sm_c1 := sm_recvAI(400, 0, j, sm_4, vd1);
      returns array of sm_b1
              array of sm_c1
```

```
      end for;

      a :=
      for i in sm_lx, sm_hx cross j in 1, 400 cross k in 1, 400
      returns array of sm_b1[i,k,k] + sm_c1[i,j,j]
      end for;

      sm_a, sm_b :=
      for i in sm_lx, sm_hx cross j in 1, 400
         sm_4, sm_a := sm_sendAI(a[i,j], 0, 400, j, sm_3, vd);
         sm_5, sm_b := sm_sendAI(a[i,j], 0, 400, j, sm_4, vd);
      returns array of sm_a
              array of sm_b
      end for;

   in
      sm_a, sm_b
   end let
   end if
in
   sm_a, sm_b
end let
end function
```

Figura 54. Resultado obtido da análise para divisão do programa da Figura 52

6.2.4. Diferenças Finitas

O programa a seguir é uma implementação simplificada do algoritmo de *Jacobi-Richardson* para encontrar uma solução para um sistema de diferenças finitas, comumente utilizado para tratar numericamente equações diferenciais parciais. No programa da **Figura 54** podemos observar pelas funções de acesso aos *arrays* u e x que para alinhar os dados, apenas o deslocamento será necessário, visto que

$$R^1_{u1}(\vec{a}) = R^1_{x1}(\vec{a}) = R^2_{x1}(\vec{a}) = R^3_{x1}(\vec{a}) = R^4_{x1}(\vec{a}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

ou seja pelas funções de acesso vemos que apenas os deslocamentos são diferentes.

```
define main

function main(a: array[array[real]] returns array[array[real]])
   let
      u := for initial
              k := 1;
              u := a;
              x := array[0:array[0:0.0]];
              dx2, dy2 := 0.25, 0.25;
```

```
            dx_dy := ((dx2*dy2)/(2.0*dx2+2.0*dy2));
            erro, epslon := 1.0, 0.001;
            dif, gr := 0.0, 0.0;

          repeat
            x := old u;
            u :=
            for i in 1, 400 cross j in 1, 400
            returns array of ((x[i-1,j]+x[i+1,j])/dx2+
                              (x[i,j-1]+x[i,j+1])/dy2)/(dx_dy)
            end for;

            gr := for i in 1, 400 cross j in 1, 400
                  returns value of greatest abs(u[i,j])
                  end for;
            dif := for i in 1, 400 cross j in 1, 400
                  returns value of greatest abs(u[i,j]-x[i,j])
                  end for;
            erro := dif/gr;

          while (erro > epslon)
          returns array of u
          end for;
   in
     u
   end let
end function
```

$$r^1_{x1}(\bar{a}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad r^1_{x1}(\bar{a}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix} \qquad r^2_{x1}(\bar{a}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$r^3_{x1}(\bar{a}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} \qquad r^4_{x1}(\bar{a}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Figura 55. Implementação do algoritmo de *Jacobi-Richardson* para solução para diferenças finitas

$$T^1_{x1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ \bar{t}^1_{x1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}; \quad T^2_{x1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ \bar{t}^2_{x1} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}; \quad T^3_{x1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ \bar{t}^3_{x1} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad T^4_{x1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ \bar{t}^4_{x1} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

Figura 56. Alinhamento para os *arrays* do programa da Figura 54

```
define main

function main(a: array[array[real]] returns array[array[real]])

   let

      u := for initial
             k := 1;
             u := a;
             x := array[0:array[0:0.0]];
```


IFSC-USP SERVIÇO DE BIBLIOTECA E INFORMAÇÃO

```
            dx2, dy2 := 0.25, 0.25;
            dx_dy := ((dx2*dy2)/(2.0*dx2+2.0*dy2));
            erro, epslon := 1.0, 0.001;
            dif, gr := 0.0, 0.0;

          repeat

            x := old u;

            sm_x1, sm_x2, sm_x3, sm_x4 :=
            for i in 1, 400 cross j in 1, 400
            returns array of x[i-1,j]
                    array of x[i+1,j]
                    array of x[i,j-1]
                    array of x[i,j+1]
            end function;

            u :=
            for i in 1, 400 cross j in 1, 400
            returns array of ((sm_x1[i,j]+sm_x2[i,j])/dx2+
                              (sm_x3[i,j]+sm_x4[i,j])/dy2)/(dx_dy)
            end for;

            gr := for i in 1, 400 cross j in 1, 400
                  returns value of greatest abs(u[i,j])
                  end for;

            dif := for i in 1, 400 cross j in 1, 400
                  returns value of greatest abs(u[i,j]-x[i,j])
                  end for;
            erro := dif/gr;

          while (erro > epslon)

          returns array of u
          end for;
   in

     u

   end let

end function
```

Figura 57. Alinhamento dos acessos para o programa da Figura 54

$$C_1(\vec{e}_2) = 0 \qquad \vec{e}_2 = [0 \quad 1]^t$$

$$D_1 F^1_{u1} = C_1 \qquad D_1 F^1_{x1} = C_1 \qquad D_1 F^2_{x1} = C_1 \qquad D_1 F^3_{x1} = C_1 \qquad D_1 F^4_{x1} = C_1$$

$$D_1 = C_1 = [1 \quad 0]$$

Figura 58. Resultado obtido da análise para divisão do programa da Figura 54

```
%$c=sm_rank
%$c=sm_size
%$c=sm_sendAR
```

```
%$c=sm_recvAR
%$c=sm_reduceR
%$c=sm_scatterI
%$c=sm_bcastAI
%$c=sm_bcastR

define main

global sm_rank(dep: integer returns integer, integer)
global sm_size(dep: integer returns integer, integer)
global sm_sendAR(buf:array[real]; cnt, dest, tg, in_dep: integer;
                 vd: array[integer] returns integer, array[real])
global sm_recvAR(cnt, dest, tg, in_dep: integer; vd: array[integer]
                 returns integer, array[real])
global sm_reduceR(buf: real; oper, in_dep: integer;
                  returns integer, real)
global sm_scatterI(buf:array[integer]; in_dep: integer
                   returns integer, integer)
global sm_bcastAI(buf:array[integer]; cnt, in_dep: integer
                  returns integer, array[integer])
global sm_bcastR(buf: real; in_dep: integer returns integer, real])


function main(a: array[array[real]] returns array[integer])
let
   b :=
   for initial
      k := 1;
      u := a;
      x := a;
      erro := 1.0;
      dif := 0.0;
      gr := 0.0;
      MPI_MAX := 1;

      sm_1, rank := sm_rank(0);
      sm_2, size := sm_size(0);

   repeat
      x := old u;

      u :=
      if (rank = 0) then
      let
         sm_x1, sm_x2, sm_x3, sm_x4 :=
         for i in 1, 400 cross j in 1, 400
         returns array of x[i-1,j]
                 array of x[i+1,j]
                 array of x[i,j-1]
                 array of x[i,j+1]
         end for;

         sm_L := array[0:   1, 101, 201, 301];
         sm_H := array[0: 100, 200, 300, 400];

         sm_3, sm_lx := sm_scatterI(sm_L, sm_2);
         sm_4, sm_hx := sm_scatterI(sm_H, sm_3);

         vd := array[0: 0, 1, 1, 0, 399, 1, 400, 1];
         sm_5, vd1 := sm_bcastAI(vd, 8, vd[0]);
```

```
      sm_x5, sm_x6, sm_x7, sm_x8 :=
      for x in 1, size-1  cross i in sm_L[x], sm_H[x]
         sm_6, sm_x5 := sm_sendAR(sm_x1[i], 400, x, i, sm_5, vd);
         sm_7, sm_x6 := sm_sendAR(sm_x2[i], 400, x, i, sm_6, vd);
         sm_8, sm_x7 := sm_sendAR(sm_x3[i], 400, x, i, sm_7, vd);
         sm_9, sm_x8 := sm_sendAR(sm_x4[i], 400, x, i, sm_8, vd);
      returns array of sm_x5
              array of sm_x6
              array of sm_x7
              array of sm_x8
      end for;

      u :=
      for i in sm_lx, sm_hx cross j in 1, 400
      returns array of ((sm_x1[i,j]+sm_x2[i,j])/0.25+
                        (sm_x3[i,j]+sm_x4[i,j+1])/0.25)/(0.0625)
      end for;

      gr := for i in sm_lx, sm_hx cross j in 1, 400
            returns value of greatest abs(u[i,j])
            end for;
      dif := for i in sm_lx, sm_hx cross j in 1, 400
             returns value of greatest abs(u[i,j]-x[i,j])
             end for;

      sm_6, sm_gr1 := sm_reduce(gr, MPI_MAX, sm_5);
      sm_7, sm_dif1 := sm_reduce(dif, MPI_MAX, sm_6);

      sm_u :=
      for x in 1, size-1
         sm_u :=
         for i in sm_L[x], sm_H[x]
            sm_8, sm_u := sm_recvAR(400, x, i, sm_7, vd)
         returns array of sm_u
         end for
      returns value of u || sm_u
      end for;

      erro := sm_dif1/sm_gr1;
      sm_8, erro := sm_bcastR(erro, sm_7);

   in
     sm_u
   end let
   else
   let
      sm_L := array[0:0];
      sm_H := array[0:0];

      sm_3, sm_lx := sm_scatterI(sm_L, 0);
      sm_4, sm_hx := sm_scatterI(sm_H, sm_3);

      vd := array[0: 0];
      sm_5, vd1 := sm_bcastAI(vd, 8, vd[0]);

      sm_x1, sm_x2, sm_x3, sm_x4 :=
      for i in sm_lx, sm_hx
         sm_6, sm_x5 := sm_recvAR(400, 0, i, sm_5, vd1);
         sm_7, sm_x6 := sm_recvAR(400, 0, i, sm_6, vd1);
         sm_8, sm_x7 := sm_recvAR(400, 0, i, sm_7, vd1);
         sm_9, sm_x8 := sm_recvAR(400, 0, i, sm_8, vd1);
```

```
        returns array of sm_x5
                array of sm_x6
                array of sm_x7
                array of sm_x8
        end for;

        u :=
        for i in sm_lx, sm_hx cross j in 1, 400
        returns array of ((sm_x1[i,j]+sm_x2[i,j])/0.25+
                          (sm_x3[i,j]+sm_x4[i,j])/0.25)/(0.0625)
        end for;

        gr := for i in sm_lx, sm_hx cross j in 1, 400
              returns value of greatest abs(u[i,j])
              end for;
        dif :=  for i in sm_lx, sm_hx cross j in 1, 400
                returns value of greatest abs(u[i,j]-x[i,j])
               end for;

        sm_6, sm_gr1 := sm_reduce(gr, MPI_MAX, int(gr));
        sm_7, sm_dif1 := sm_reduce(dif, MPI_MAX, int(grt));

        sm_u :=
        for i in sm_lx, sm_hx
           sm_8, sm_u := sm_sendAR(u[i], 0, 400, i, sm_7, vd)
        returns array of sm_u
        end for;

        sm_8, erro := sm_bcastR(erro, sm_7);

      in
         sm_u
      end let
      end if;
   while (erro > 0.001)
   returns array of k
   end for;
in
   b
end let
end function
```

Figura 59. Resultado obtido da análise para divisão do programa da Figura 39

6.2.5. Multiplicação de Matrizes

Como estudo final, apresentaremos a multiplicação de matrizes bidimensionais, caso onde as análises para a distribuição de dados não terão sucesso, tanto no alinhamento como na divisão e, portanto não será então realizada a distribuição. Como podemos observar na multiplicação de matrizes temos três *loops* envolvendo matrizes bidimensionais o que não possibilita a solução do sistema. A transformação linear não será encontrada visto que o critério de existência da seção 4.1 não será satisfeito e não temos

matrizes de decomposição válidas para o programa. Assim o programa não será transformado pelas nossas análises.

```
define main

function main(a, b: array[array[integer]] returns array[array[integer]])

   for i in 1, 400 cross j in 1, 400
      c :=
      for k in 1, 400
      returns value of sum a[i,k]*b[k,j]
      end for
   returns array of c
   end for
end function
```

$$l^1_{c1}(\vec{a}) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad r^1_{a1}(\vec{a}) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad r^1_{b1}(\vec{a}) = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Figura 60. Programa para o algoritmo tradicional de multiplicação de matrizes

$$[T^1_{a1}] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \qquad [T^1_{b1}] \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Figura 61. Sistema para encontrar o alinhamento entre a e c e b e c

Para as funções afins apresentadas acima, não teremos uma transformação linear válida para o sistema da **Figura 60**. O mesmo ocorre com a divisão, como podemos ver no sistema abaixo, na **Figura 61**. Temos três possibilidades para sincronização e comunicação, mas qualquer que seja o valor de C_1 ele não satisfaz simultaneamente os sistemas.

$$(1) \quad C_1 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = 0 \qquad D_1 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = C_1 \qquad D_1 \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = C_1$$

$$(2) \quad C_1 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = 0 \qquad D_1 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = C_1 \qquad D_1 \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = C_1$$

$$(3) \quad C_1 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 0 \qquad D_1 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = C_1 \qquad D_1 \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = C_1$$

Figura 62. Sistema para encontrar a divisão entre a e c e b e c

Assim como não temos possibilidade de alinhamento ou divisão pelas análises propostas, o módulo inserido no compilador OSC deverá manter o código sem transformações e passa-lo a fase seguinte.

6.3 Conclusões

Apresentamos neste capítulo exemplos de programas SISAL aos quais aplicamos as análises para a distribuição de dados e computações e o resultado final que deveria ser obtido da geração de código.

Podemos observar pelos programas que o paradigma utilizado é o SPMD com organização mestre/escravo. Também temos a chamada de rotinas MPI que possibilitam a comunicação entre os processos que forem criados. Para realizar esta interface foi utilizado o suporte para interação com a linguagem C oferecida pelo compilador OSC.

Devemos ressaltar, no entanto, que esta não é a única possibilidade de código que poderia ser obtida das análises apresentadas. Um exemplo seria optar por realizar o alinhamento durante a transmissão dos dados para os processos escravos ao invés de realizar a criação de novos *arrays*. Também poderíamos utilizar somente a análise de divisão caso desejemos a granularidade mais grossa possível, visto que, com o alinhamento, o que obtemos principalmente é a maior possibilidade de divisão de carga.

Um último comentário diz respeito à transição entre a proposta matemática e a implementação em algum sistema, que muitas vezes apresenta dificuldades devido à restrição imposta pelos computadores, uma delas a memória, e não estamos nos referindo à quantidade de memória, mas ao conceito de manipulação de memória. Por exemplo, no alinhamento, quando realizamos uma transformação linear sobre uma base, a quantidade de pontos envolvidos não preocupa, pois estamos tratando com espaço matemático. Quando realizamos esta análise sobre um *array* alocado por um programa de computador, uma das informações que temos são os limites deste *array*, o que nos impõe restrições quanto à mudança de base e, portanto a mudança de base de um *array* deve ter o cuidado de verificar se os limites estão sendo respeitados.

7. CONCLUSÕES

Iniciaremos a seguir a apresentação das principais conclusões obtidas deste trabalho relembrando que o escopo foi estudar técnicas de otimização estática de dados e código a serem utilizadas na compilação automática de programas e que propusemos três métodos de otimização: 1) alinhamento de dados, 2) divisão e 3) mapeamento de dados e computação, todos fundamentados na álgebra linear, e realizamos a implementação das análises no compilador OSC.

Posto isto, a seguir apresentaremos na forma de seções as principais conclusões deste trabalho.

7.1. ALINHAMENTO

Na seção 4.1 apresentamos o método de alinhamento de dados usado, a busca de uma matriz de transformação linear que descreve a dependência entre variáveis. Caso obtenhamos sucesso nessa busca, temos como resultado a possibilidade de ter o acesso aos dados seguindo o mesmo padrão. Uma das principais vantagens em se utilizar métodos matemáticos para a análise está na validação do método. No nosso caso sabemos que transformações lineares são estudadas há bastante tempo e, por conseguinte já bem conhecidas.

Vejamos então como tiramos proveito disto no alinhamento de dados. Primeiros recordem, que o método proposto baseia-se em funções afins (translações de funções lineares). A busca pelo alinhamento envolve funções afins e uma transformação linear que permite o mesmo padrão de acesso aos *arrays* do lado esquerdo e do lado direito de uma expressão. Após algumas manipulações obtemos um sistema de equações lineares que podem ser resolvidas simultaneamente. Na nossa implementação usamos o método de inversão de matrizes de Gauss-Jordan visando resolver o sistema.

A matriz resultante da aplicação do método de Gauss-Jordan, não pode, entretanto, ser considerada a solução para o conjunto de sistema até que algum critério de consistência seja utilizado. No nosso caso utilizamos o critério de consistência apresentado por Meyer em [MEY00]. Em contraste com o método proposto por O'Boyle apresentamos o critério

para a verificação da existência ou não de uma transformação linear válida. Assim, existirá uma transformação linear válida se e somente se os critérios forem satisfeitos.

O uso deste método de alinhamento por um compilador envolve outras considerações, entre elas a classe de linguagens a que será aplicada. A utilização deste método em linguagens imperativas exige análises para verificação de dependência de dados, na tentativa de detecção de quais códigos são passíveis de aplicação do método de alinhamento. No caso de linguagens funcionais, a aplicação do método não implica em análises de dependência, visto que esta classe de linguagens tem sua base na semântica matemática onde não temos a manipulação de posições de memória. Na seção 7.4 discutiremos como foi realizada a implementação do alinhamento para a linguagem funcional SISAL.

Como conclusão final desta seção, recordemos que existem critérios que definem ou não a existência do alinhamento. Caso eles sejam satisfeitos, a divisão de dados e computação pode ser realizada livremente em qualquer uma das dimensões do *array* ou aninhamentos de *loops*. Caso contrário, o método de divisão proposto tentará realizar a análise de quais dados e computações deveriam ser mantidas juntas.

7.2. DIVISÃO

A análise para a divisão de dados e computações também se baseia na análise de funções afins; no entanto, ao contrário do alinhamento onde apenas o acesso a dados é levado em conta, ele tenta relacionar dados e computações, mais precisamente qual divisão de dados e computações não geraria comunicação numa eventual distribuição. Novamente temos como aliada à álgebra linear, permitindo assim que a formulação beneficie-se de todo estudo já realizado previamente.

Esse relacionamento é obtido por um sistema denominado equação de comunicação que é composto por uma função afim denominada decomposição de dados afim e uma outra denominada decomposição de computação afim, ambas ligadas por uma função de acesso a *array* e pela a equação de sincronização. Assim, o processo para encontrar uma divisão de dados ideal pode ser definido como a busca por decomposições de dados e computações que satisfaçam o sistema elaborado.

A solução para o sistema elaborado consiste em combinar todas as possíveis soluções para cada uma das equações do sistema e então selecionar aquelas que são

comuns às duas. A nossa implementação primeiro seleciona quais aninhamentos de *loops* serão distribuídos e para a solução obtida quais dados podem ser então distribuídos, ou seja, o resultado obtido indica a possibilidade ou não da divisão de dados. Se apenas a solução trivial existir, dados e computações deverão ser mantidos locais. Do contrário caso exista mais de uma possível solução escolheremos aquela que possibilite a distribuição de maior granularidade possível, ao contrário do que foi proposto por Anderson [AND97]. A escolha deve-se ao fato de buscarmos distribuição para sistemas de memória distribuída.

A princípio, a busca de todas as combinações possíveis pode indicar uma quantidade de computação muito elevada com *arrays* de muitas dimensões. No entanto, problemas que envolvem *arrays* desse tipo não são comuns. Assim, uma opção diferente para solucionar o sistema pode ser implementada caso esse tipo de problema torne-se mais freqüente.

Portanto se encontramos alguma solução diferente da trivial, podemos então realizar a análise seguinte que visa mapear dados e computações nos processadores físicos.

7.3. MAPEAMENTO

O método de mapeamento consiste basicamente em calcular o limite inferior e o superior de *loops* e *arrays* que devem ser distribuídos entre os processadores físicos disponíveis no sistema. A escolha de qual dimensão do *array* e aninhamento de *loops* terá os índices recalculados foi obtida da análise de divisão, que também estabeleceu a relação entre eles.

A base da formulação apresentada é um sistema de inequações que representa os limites de um aninhamento de *loops* e outro que representa os limites de uma dimensão do *array*. Os novos limites consistem de seqüências de valores calculados a partir da quantidade de processadores físicos disponíveis, quantidade de empacotamentos desejados, a quantidade de dados associados por processador e os limites originais.

Nestes cálculos apresentamos um cálculo simplificado de balanceamento de carga, considerando que o sistema que será utilizado será homogêneo. Entretanto, para casos onde não temos esta característica, o sistema deveria ser alterado e o tamanho dos blocos associados a cada um dos processadores deveria ser diferente.

Contrastando com as duas análises anteriores o cálculo dos índices é aquele que exige o conhecimento do sistema destino a ser utilizado, não ficando restrito apenas à

análise do código fonte. Assim, as alterações realizadas no sistema deverão a partir deste ponto realizar chamadas que permitam obter informações a respeito da configuração do sistema. Na nossa implementação, ao invés de informações a respeito do número de processadores disponíveis, levaremos em conta o número de processos utilizados, o que nos leva a manter um processo por processador.

A seguir discutiremos as principais conclusões obtidas da implementação da nossa proposta.

7.4. IMPLEMENTAÇÃO

O principal objetivo com a implementação do método de distribuição apresentado foi verificar a viabilidade da utilização do método em linguagem funcionais tipo SISAL, o qual foi atingido com sucesso.

Os benefícios de se utilizar um modelo matemático também ficaram evidentes, visto que a preocupação durante o desenvolvimento não foi com a elaboração dos métodos de análise, mas sim com a implementação de um programa que fielmente representasse a nossa proposta. Essa redução da "carga-de-trabalho" foi importante, pois apesar de não nos preocuparmos com o método de análise para distribuição, muitas outras questões tiveram que ser abordadas.

Apesar de atingirmos nossos objetivos podemos constatar que entre uma proposta e a sua efetiva implementação a distância é grande. Como citado anteriormente, a transição entre a proposta matemática e a sua implementação em algum sistema, muitas vezes apresenta dificuldades devido às restrições impostas pelos computadores, entre elas o conceito de manipulação de posições de memória que não existe na matemática.
Foram diversos os aspectos que tiveram que ser tratados, entre eles qual seria a linguagem a analisar e como seria realizada a extração dos dados para a aplicação das análises propostas.

A escolha pela linguagem SISAL deveu-se ao fato dela ser uma linguagem funcional (associação simples e sem efeitos colaterais) com sintaxe atraente para computação numérica e a experiência já adquirida em relação ao compilador OSC. Além disto, a forma intermediária associada a ele tem entre outros atrativos o fato de ser definida como uma linguagem de grafos, chamada IF1, o que indicava a facilidade de manipulação.

Verificamos neste trabalho que a escolha foi acertada, independente dos pequenos problemas ocorridos. Um dos principais pontos que vamos destacar é a forma modular como foi implementado o compilador OSC. A independência entre os módulos que implementam as diversas análises existentes no compilador permitiu-nos a inserção de um módulo com a implementação de nossas análises. Apesar de não termos disponíveis documentos a respeito da implementação do compilador, o estudo do código fonte foi o suficiente para conseguirmos realizar a inserção de nosso módulo.

Após este estudo, passamos à implementação da extração dos dados, que foi uma das mais trabalhosas visto que a implementação de IF1 no compilador OSC está um pouco modificada em relação à especificação apresentada no manual da linguagem IF1. Essas alterações eram justamente na construção *forall*, escolhida para as nossas análises.

Abreviadamente podemos dizer que são em relação a como são passados os valores ao corpo do *forall*. Na semântica apresentada temos que os índices gerados por loops aninhados e que são passados aos loops mais internos não sofrem alteração. Toda operação a ser realizada, por exemplo, o deslocamento de um índice por algum fator, ocorre dentro do corpo do *loop* mais interno. Na implementação realizada no compilador OSC, as operações sobre um índice ocorrem no corpo do *forall* onde foi gerado o índice.

Como citado anteriormente, o fato da forma intermediária ser definida como uma linguagem permitiu que a detecção desta alteração fosse visualizada com certa facilidade, após constatarmos que a implementação que estava de acordo com o manual não funcionava corretamente. Realizadas as alterações passamos então a conseguir extrair os dados a serem analisados.

Em relação às estruturas escolhidas para armazenar as informações extraídas do código IF1, elas satisfizeram plenamente as nossas necessidades. Basicamente, consistiram de árvores e grafos, o que está em conformidade com o que é proposto em geral na literatura.

Também como esperado, o método de Gauss-Jordan e os critérios de consistência apresentados por Meyer foram plenamente satisfatórios na busca pelo alinhamento. A busca por todas soluções possíveis para o sistema de comunicação da fase de divisão foi tranqüilamente obtida, no que diz respeito ao tempo computacional envolvido.

Por fim notamos que uma alteração na gramática da linguagem intermediária IF1 seria interessante, não apenas para corrigir a diferença anteriormente citada, mas também para retirar definições que são apresentadas como semântica e pode muito bem ser

colocadas como regras sintáticas. Assim, na seção de sugestões de trabalhos futuros apresentaremos a sugestão de alteração na gramática.

7.5. RESULTADOS

O que fica claro com os resultados apresentados é que a análise para a distribuição de dados e de computações pode ser aplicada em uma variedade de programas SISAL e ao final gerado o código necessário. O padrão apresentado em todos os programa indica que podemos realizar esta geração automaticamente, via compilador. Entretanto, vemos também que a quantidade de código gerado torna-se muito maior que o código inicial o que indica que a divisão valerá a pena somente se a carga de computação for elevada. Assim a implementação das análises deve tentar ao máximo reduzir a quantidade de código gerado. Como citado anteriormente, o uso das análises de alinhamento e de divisão feitas pode ajudar nessa redução do código gerado.

7.6. TRABALHOS FUTUROS

Um importante trabalho a ser realizado é o balanceamento de carga, tanto estático como dinâmico; ele será importante caso desejemos utilizar sistemas que sejam heterogêneos ou que não sejam herméticos, montados apenas para realizar processamento. Caso as análises sejam visando o processamento paralelo em sistemas dedicados, as análises desenvolvidas poderiam ser acrescidas de módulos para realizar o balanceamento de carga estático. Entretanto, se optarmos por realizar o processamento em sistemas não dedicados, o balanceamento de carga dinâmico seria o mais interessante. Além disso, as análises realizadas poderiam ser estendidas, incluindo também a análise para a distribuição de procedimentos e funções. Junto com as análises, também seria necessário disponibilizar um gerenciador de carga do sistema, que seria o responsável em reescalonar os processos.

Como citado anteriormente, uma alteração na gramática da linguagem IF1 visando passar para a sintaxe da linguagem definições que atualmente são atribuídas à semântica traria um benefício na escrita de um analisador sintático. Justificamos nossa proposta levando em consideração que atualmente a gramática de IF1 não estabelece a ordem em que devem ser aplicadas as diversas regras gramaticais; ou seja, quase não existe dependência entre as regras de produção da linguagem. Por exemplo, as regras

apresentadas para definir nós e os arcos desses nós não possuem ligação sintática, é a semântica que dita a relação entre eles. Entretanto, com alterações na gramática podemos passar esta ligação para as regras gramaticais, tornando assim mais claro o que está intrinsecamente relacionado como a sintaxe e a semântica.

Outra importante questão é quanto à implementação em linguagens que sejam mais utilizadas em computação científica, por exemplo, C e FORTRAN. Um caminho para isso seria estudar o compilador SUIF [AMA93], do *Computer Systems Laboratories* da *Stanford University*, onde são implementadas diversas otimizações para sistemas de memória compartilhada. Para realizar estas otimizações também são realizadas análises para verificar a dependência de dados em códigos C e FORTRAN, transformando-os se possível em códigos livres de dependência. Partindo destes códigos livres de dependências, poderíamos então realizar nossas análises visando a distribuição.

Também como trabalho futuro seria interessante estudar uma outra área da Matemática relacionada com a otimização conhecida como Programação Linear. Esse estudo permitiria que fossem propostos ou utilizados novos modelos de análise de programas visando a otimização de códigos de computador. Atualmente, uma técnica conhecida como *tiling* vem sendo bastante estudada, visto que ela pode conseguir reduzir a quantidade de dados mantidos na memória, tanto em sistemas de memória local como em sistemas de memória distribuída, o que é ideal para sistemas que utilizam memória hierárquica.

7.7. CONCLUSÕES GERAIS

Como citado inicialmente, o trabalho apresentado foi baseado nos trabalhos de O'Boyle [OBO92] e Anderson [AND97] que apresentaram as propostas de alinhamento/ mapeamento e divisão de dados respectivamente.

Na proposta apresentada por O'Boyle para o alinhamento de dados não é realizada a verificação de existência ou não das transformações lineares procuradas o que pode levar a resultado incorretos. Neste trabalho introduzimos os critérios para verificar a consistência dos resultados obtidos.

Além da proposta de alinhamento, O'Boyle também apresenta uma proposta de mapeamento de computações e dados e para isso apresenta um sistema para relacionar dados e computações baseado nos limites de ambos. Diferente de O'Boyle, nós obtivemos

o relacionamento entre dados e computações através da análise de divisão apresentada por Anderson. Após o estabelecimento da relação, o cálculo de índices a serem mapeados é realizado com um modelo derivado da proposta apresentada por O'Boyle.

Os estudos realizados por O'Boyle basearam-se na linguagem SISAL e ficaram no âmbito de análises teóricas. Ao contrário dele realizamos a implementação das análises de alinhamento, divisão e mapeamento no compilador OSC o que nos permitiu verificar a viabilidade ou não da proposta.

Com relação ao trabalho de Anderson, uma grande diferença está no paradigma de linguagem escolhido para realizar as análises; o escolhido por Anderson é o das linguagens imperativas e o nosso, é o das linguagens funcionais. Esta diferença, no entanto, não inviabilizou a utilização da sua proposta para divisão de dados na linguagem SISAL. A diferença de paradigmas implicou principalmente no ponto de partida das análises, ou seja, exigiu de Anderson a utilização de alguma análise para exposição de paralelismo, no caso, àquela proposta por Wolfe [WOL92]. No nosso caso, o ponto de partida foi à escolha de qual estrutura seria paralelizada, visto que o paralelismo já é inerente nas linguagens funcionais, e em especial em SISAL que foi projetada para computações numéricas.

Outra diferença entre os trabalhos está na escolha da granularidade para o mapeamento dos programas nos processadores. Anderson faz a opção por uma granularidade fina visto que as análises foram implementadas no compilador SUIF. Neste trabalho optamos pela granularidade grossa, pois visamos a compilação para sistemas de memória distribuída usando passagem de mensagens. Entretanto, em função da análise de alinhamento realizada em nossa proposta, é maior a possibilidade da escolha por granularidade que na proposta apresentada por Anderson, ou seja, em geral deverá ser de média a grossa.

Com relação ao trabalho apresentado por Salazar e Böhm [SAL97], a semelhança dos trabalhos está na escolha da linguagem *SISAL*, sistemas de memória distribuída e passagem de mensagem usando MPI. No restante, as análises para distribuição e opções para implementação são bastante distintas.

Comparando primeiro as análises realizadas, Salazar realiza análises para alinhamento dos dados baseadas em hiperplanos paralelos e nós utilizamos as transformações lineares, ambas da Álgebra Linear. Para a divisão dos dados, ele utiliza a técnica *snake*, derivada da técnica de grafos de reuso de dados e nós utilizamos sistemas de equações de comunicação e sincronização, novamente, com base na Álgebra Linear. No nosso caso, as técnicas utilizadas permitem que dados coletados numa fase de análise sejam facilmente

passados a outra fase visto que elas estão baseadas em funções afins. No caso de Salazar e Böhm existe a necessidade do tratamento dos dados visto que uma das análises usa informações como funções afins e outra como estruturas de grafos.

A opção que mais difere, no entanto está na escolha da fase onde serão realizadas as análises de distribuição. No D-OSC as análises são realizadas sobre código IF2 e no nosso caso sobre código IF1. A opção de Salazar e Böhm é justificada visto que as análises de alinhamento e divisão de dados que realizam estão desde o início intimamente ligadas ao sistema destino escolhido. A nossa opção foi a de tentar realizar uma análise de mais alto nível, independente ao máximo da arquitetura escolhida. Junto a essas análises, estão as análises de mapeamento que realizam então a ligação com o sistema destino.

O motivo principal da escolha foi o de desejarmos que nossas análises interferissem o mínimo possível nas otimizações já implementadas no OSC e assim optamos por inserir o módulo para analise de distribuição logo após o *front-end* que gera o código IF1.

Um outro motivo não menos importante foi à linguagem IF1, que além das características já citadas é independente de máquina e, portanto teria o potencial de ser utilizada como forma intermediária de outras linguagens, permitindo assim aproveitar as otimizações já realizadas.

APÊNDICE A

A seguir os conceitos coluna básica, *rank,* e núcleo apresentados por Meyer [MEY00].

Rank e coluna básica de uma matriz

Suponha que $A_{m\times n}$ foi reduzida a forma escada E através de operações sobre suas linhas. O rank de A é definido como sendo

rank (A) = o número de *pivôs* (primeiro elemento não zero de uma linha da matriz escada)
= numero de linhas não zero de **E**
= número de colunas básicas em **A**

onde as colunas básicas de A são definidas como as colunas de A que contém as posições dos *pivôs*.

Núcleo

Para uma matriz A, o conjunto $N(A) = \{x_{n\times 1} \mid Ax = 0\} \subseteq \Re^n$ é chamado núcleo de A. Em outras palavras, $N(A)$ é simplesmente o conjunto de todas as soluções do sistema homogêneo $Ax = 0$.

APÊNDICE B

Listagem das regras de produção de IF1.

```
File  ::= [Line Comment Newline]...

Line  ::= C
        | E    Source  Destination  TypeReference
        | G    Type    Reference    "Literal"
        | G    TypeReference
        | I    Type Reference "Literal"
        | L            Destination  TypeReference "Literal"
        | L            Destination  TypeReference ErrorValue
        | N    Label   Node
        | T    Label   TypeTableEntry
        | {    "Compound"     Label          Node
        | }    Label   Node         Count    Assoc
        | X    Type    Reference     "Literal"

Label              ::= PosInteger
Source             ::= SourceNode          SourcePort
Destination        ::= DestinationNode     DestinationPort
SourceNode         ::= Integer
SourcePort         ::= PosInteger
DestinationNode    ::= Integer
DestinationPort    ::= PosInteger
TypeReference      ::= Integer
Count              ::= Integer
AssocList          ::= Integer ...

Node  ::=      0                          (Forall)
        |      1                          (Select)
        |      2                          (TagCase)
```

3	(LoopA)
4	(LoopB)
100	(AAddH)
101	(AAddL)
102	(AExtract)
103	(ABuild)
104	(Acatenate)
105	(AElement)
106	(AFill)
107	(AGather)
108	(AIsEmpty)
109	(ALimH)
110	(ALimL)
111	(ARemH)
112	(ARemL)
113	(AReplace)
114	(AScatter)
115	(ASetL)
116	(ASize)
117	(Abs)
118	(BindArguments)
119	(Bool)
120	(Call)
121	(Char)
122	(Div)
123	(Double)
124	(Equal)
125	(Exp)
126	(FirstValue)
127	(FinalValue)
128	(Floor)
129	(Int)
130	(IsError)
131	(Less)

```
        |    132                          (LessEqual)
        |    133                          (Max)
        |    134                          (Min)
        |    135                          (Minus)
        |    136                          (Mod)
        |    137                          (Neg)
        |    138                          (NoOp)
        |    139                          (Not)
        |    140                          (NotEqual)
        |    141                          (Plus)
        |    142                          (RangeGenerate)
        |    143                          (RBuild)
        |    144                          (RElements)
        |    145                          (RReplace)
        |    146                          (RedLeft)
        |    147                          (RedRigth)
        |    148                          (RedTree)
        |    149                          (Reduce)
        |    150                          (RestValues)
        |    151                          (Single)
        |    152                          (Times)
        |    153                          (Trunc)

TypeTableEntry ::= Array     TypeReference
                 | Stream    TypeReference
                 | Multiple  TypeReference
                 | FunctionT TypeReference  TypeReference
                 | Tuple     TypeReference  TypeReference
                 | Record    TypeReference
                 | Field     TypeReference  TypeReference
                 | Union     TypeReference
                 | Tag       TypeReference  TypeReference
                 | Basic     BasicType
                 | Wild
```

```
Array      ::= 0
Basic      ::= 1
Field      ::= 2
FunctionT  ::= 3
Multiple   ::= 4
Record     ::= 5
Stream     ::= 6
Tag        ::= 7
Tuple      ::= 8
Union      ::= 9
Wild       ::= 10

BasicType ::= 0                    (Boolean)
            | 1                    (Character)
            | 2                    (Double)
            | 3                    (Integer)
            | 4                    (Null)
            | 5                    (Real)
            | 6                    (WildBasic)

PosInteger      ::= NotZeroDigit [Digit]...

Digit           ::= 0 | NotZeroDigit

NotZeroDigit    ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

APÊNDICE C

Normas léxicas de IF1 no formato de entrada do FLEX .

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "if1.tab.h"
%}

%option noyywrap
%%

\"[^\"]*\" {/*  printf("%s\n", yytext);  */
              yylval.string = (char *) malloc(strlen(yytext));
              strcpy(yylval.string, yytext);
              return(LITERAL);
           }
[0-9][0-9]*  {/*  printf("%s\n", yytext); */
               yylval.value = atoi(yytext);
               return(INTEGER);
             }
C {/*  printf("C\n"); */
     yylval.car = yytext[0];
     return(C);
  }
E {/*  printf("E\n"); */
     yylval.car = yytext[0];
     return(E);
  }
G {/*  printf("G\n"); */
     yylval.car = yytext[0];
     return(G);
  }
I {/*  printf("I\n"); */
     yylval.car = yytext[0];
     return(I);
  }
L {/*  printf("L\n"); */
     yylval.car = yytext[0];
     return(L);
  }
N {/*  printf("N\n");*/
     yylval.car = yytext[0];
     return(N);
  }
T {/*  printf("T\n");*/
     yylval.car = yytext[0];
     return(T);
  }
X {/*  printf("X\n");*/
     yylval.car = yytext[0];
     return(X);
  }
\{ {/*  printf("{\n");*/
      yylval.car = yytext[0];
      return(BRACEO);
```

```
    }
\} {/*  printf("}\n");*/
      yylval.car = yytext[0];
      return(BRACEC);
    }
$[^\n]*  {/*  printf("%s\n", yytext);*/
            yylval.string = (char *) malloc(strlen(yytext)+1);
            strcpy(yylval.string, yytext);
            return(COMMENT);
          }
%[^\n]*  {  yylval.string = (char *) malloc(strlen(yytext)+1);
           /*  printf("%s\n", yytext);*/
            strcpy(yylval.string, yytext);
            return(COMMENT);
          }
Compound {/*  printf("Compound\n");*/
            yylval.string = (char *) malloc(strlen(yytext)+1);
            strcpy(yylval.string, yytext);
            return(COMPOUND);
          }
error(\[[.]*\])  {  yylval = atoi(yytext);
                    yylval.string = (char *) malloc(strlen(yytext)+1);
                    strcpy(yylval.string, yytext);
                    return(ERROR_VALUE);
                 }
error(\{[.]*\})  {  yylval = atoi(yytext);
                    yylval.string = (char *) malloc(strlen(yytext)+1);
                    strcpy(yylval.string, yytext);
                    return(ERROR_VALUE);
                 }
error(\<[.]*\>)  {  yylval = atoi(yytext);
                    yylval.string = (char *) malloc(strlen(yytext)+1);
                    strcpy(yylval.string, yytext);
                    return(ERROR_VALUE);
                 }
[\n]+ {  yylval.car = '\n';
         /*  printf("NEWLINES\n");*/
         return(NEWLINES);
      }
[' '\t]*  /* eat up white spaces */

.  /* unrecognized character */

%%
```

APÊNDICE D

A listagem das regras gramaticais da linguagem IF1 no formato exigido pelo BISON.

```
/*********************************************************************/
/*                          file name : if1.y
*/
/*                                IF1 grammar
*/
/*********************************************************************/

%{
#include <stdlib.h>
#include <string.h>
#include "transformation.h"
#include "bstack.h"

char lntype, *comment, *string, edge = ' ';
pgraph init, end, master = NULL, slave = NULL, g;
pedges edgsi = NULL, edgsf = NULL;
pnodes nodei = NULL, nodef = NULL;
int src, psrc, dest, pdest, i, data[4], *assoc;
extern int basic[7];
ptypetbl tpinit = NULL, tpend = NULL, t;

%}

%union {
  char car;
  char *string;
  int value;
}

%token <string> LITERAL       /* String literal          */
%token <string> COMPOUND      /* The string "Compound"   */
%token <car> T                /* The character T         */
%token <car> C                /* The character C         */
%token <car> E                /* The character E         */
%token <car> G                /* The character G         */
%token <car> I                /* The character I         */
%token <car> L                /* The character L         */
%token <car> N                /* The character N         */
%token <car> X                /* The character X         */
%token <car> BRACEO           /* The character {         */
%token <car> BRACEC           /* The character }         */
%token <value> INTEGER        /* Integer                 */
%token <string> ERROR_VALUE   /* Error value             */
%token <car> NEWLINES         /* character for new line */
%token <string> COMMENT       /* Comment                 */

%type <value> typereference
%type <value> label
%type <value> node
%type <value> nodecmpd
%type <value> sourcenode
```

```
%type <value> sourceport
%type <value> destinationnode
%type <value> destinationport
%type <value> count

%start file

%%
/* Grammar follows */

file:  /*    empty string     */
|  file line COMMENT NEWLINES {
    switch (lntype) {
    case 'T':
       tpend->comment = (char *) realloc(tpend->comment, strlen($3)+1);
       strcpy(tpend->comment, $3);
       break;
    case 'E': case 'L':
       if (dest == 0)
       {  end->edgsf->comment = (char *) realloc(end->edgsf->comment,
                                                 strlen($3)+1);
          strcpy(end->edgsf->comment, $3);
       }
       else
       {  edgsf->comment = (char *) realloc(edgsf->comment,
                                            strlen($3)+1);
          strcpy(edgsf->comment, $3);
       }
       break;
    case 'C': case 'X': case 'G': case 'I':
       end->comment = (char *) realloc(end->comment, strlen($3)+1);
       strcpy(end->comment, $3);
       break;
    case '{': case '}': case 'N':
       nodef->comment = (char *) realloc(nodef->comment, strlen($3)+1);
       strcpy(nodef->comment, $3);
       break;
   };
 }
 |  file line NEWLINES
 ;
 line: T label typetableentry {
    lntype = $1;
    data[0] = $2;
    if (data[1] == 1)
       basic[$2-1] = $2;
    instypetbl(&tpinit, &tpend, data, "");
 }
 | C {
    lntype = $1;
    insgraph(&init, &end, $1, -1, "", "");
    edge = ' ';
    edgsi = NULL;
    edgsf = NULL;
 }
 | I typereference LITERAL  {
    lntype = $1;
    insgraph(&init, &end, $1, $2, $3, "");
    edge = ' ';
    edgsi = NULL;
    edgsf = NULL;
```

```
    }
    | G typereference LITERAL {
       lntype = $1;
       insgraph(&init, &end, $1, $2, $3, "");
       edge = 'G';
       edgsi = NULL;
       edgsf = NULL;
    }
    | X typereference LITERAL  {
       lntype = $1;
       insgraph(&init, &end, $1, $2, $3, "");
       edge = 'G';
       edgsi = NULL;
       edgsf = NULL;
    }
    | E source destination typereference {
       lntype = $1;
       if (dest == 0)
          insedge(&end->edgsi, &end->edgsf, $1, src, psrc, dest, pdest,
                  $4, "", "");
       else
       {  insedge(&edgsi, &edgsf, $1, src, psrc, dest, pdest, $4, "", "");

          if (edge == 'G')
          {  end->edgsi = edgsi;
             end->edgsf = edgsf;
          }
          else if (edge == 'N')
          {  nodef->edgsi = edgsi;
             nodef->edgsf = edgsf;
          }
       }
    }
    | L destination typereference LITERAL {
       lntype = $1;
       if (dest == 0)
          insedge(&end->edgsi, &end->edgsf, $1, -1, -1, dest, pdest, $3, $4,
                  "");
       else
       {  insedge(&edgsi, &edgsf, $1, -1, -1, dest, pdest, $3, $4, "");

          if (edge == 'G')
          {  end->edgsi = edgsi;
             end->edgsf = edgsf;
          }
          else if (edge == 'N')
          {  nodef->edgsi = edgsi;
             nodef->edgsf = edgsf;
          }
       }
    }
    | L destination typereference ERROR_VALUE {
       lntype = $1;
       if (dest == 0)
          insedge(&edgsi, &edgsf, $1, -1, -1, dest, pdest, $3, $4, "");
       else
       {  insedge(&edgsi, &edgsf, $1, -1, -1, dest, pdest, $3, $4, "");

          if (edge == 'G')
          {  end->edgsi = edgsi;
             end->edgsf = edgsf;
```

```
        }
        else if (edge == 'N')
        {  nodef->edgsi = edgsi;
           nodef->edgsf = edgsf;
        }
    }
}
| G typereference {
   lntype = $1;
   insgraph(&init, &end, $1, $2, "", "");
   edge = 'G';
   nodei = NULL;
   nodef = NULL;
   edgsi = NULL;
   edgsf = NULL;
}
| N label node    {
   lntype = $1;
   insnode(&nodei, &nodef, $1, $2, $3, NULL, "");
   end->nodei = nodei;
   end->nodef = nodef;
   edge = 'N';
   edgsi = NULL;
   edgsf = NULL;
}
| BRACEO COMPOUND label INTEGER {
   lntype = $1;
   insnode(&nodei, &nodef, $1, $3, $4, NULL, "");
   end->nodei = nodei;
   end->nodef = nodef;
   edge = 'N';
   pushg(init, end);
   pushn(nodei, nodef);
   init = NULL;
   end = NULL;
   edgsi = NULL;
   edgsf = NULL;
}
| BRACEC label nodecmpd count assoclist {
   lntype = $1;
   popn(&nodei, &nodef);
   nodef->gi = init;
   nodef->ge = end;
   popg(&init, &end);
   insnode(&nodei, &nodef, $1, $2, $3, assoc, "");
   end->nodei = nodei;
   end->nodef = nodef;
   edge = 'N';
   edgsi = NULL;
   edgsf = NULL;
   free(assoc);
}
;
label: INTEGER {
}
;
source: sourcenode sourceport {
   src = $1; psrc = $2;
}
;
destination: destinationnode destinationport {
```

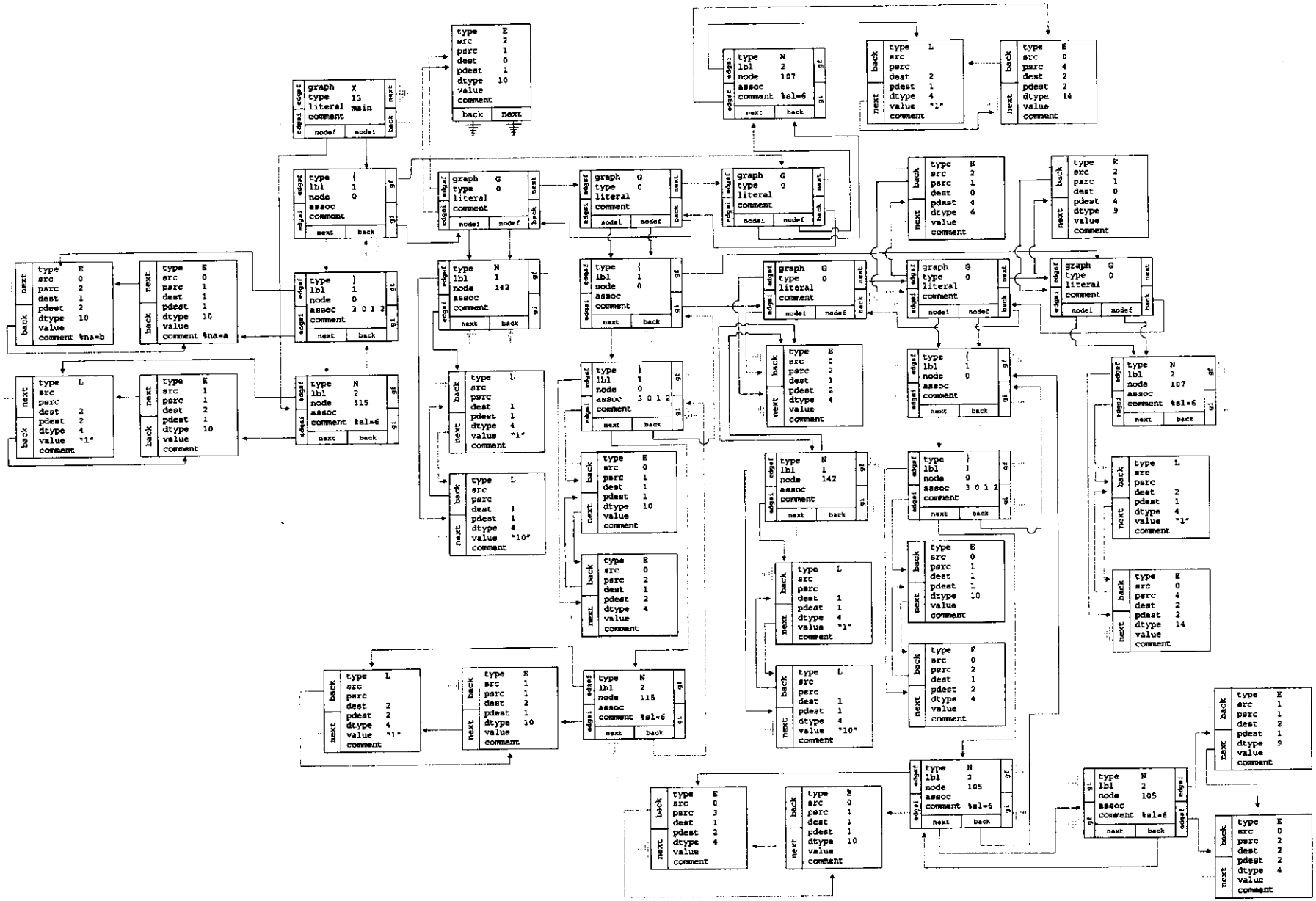
```
    dest = $1; pdest = $2;
}
;
sourcenode: INTEGER
;
sourceport: INTEGER
;
destinationnode: INTEGER
;
destinationport: INTEGER
;
typereference: INTEGER {
}
;
count: INTEGER {
    assoc = (int *) malloc(sizeof(int)*($1+1));
    assoc[0] = $1;
    i = 1;
}
;
assoclist: INTEGER {
    assoc[i] = $1;
    i++;
}
| assoclist INTEGER {
    assoc[i] = $2;
    i++;
}
;
nodecmpd: INTEGER
;
node: INTEGER
;
typetableentry: INTEGER {
    data[1] = $1;
    data[2] = -1;
    data[3] = -1;
}
| INTEGER INTEGER {
    data[1] = $1;
    data[2] = $2;
    data[3] = -1;
}
| INTEGER INTEGER INTEGER {
    data[1] = $1;
    data[2] = $2;
    data[3] = $3;
}
;
%%

int yyerror(s)
char *s;
{
  return 0;
}

extern FILE *yyin;

void main(int argc, char **argv)
{
```

```
    ++argv, --argc;
    if (argc > 0)
    {
       yyin = fopen(argv[0], "r");
       ++argv, --argc;
    }
    else
       yyin = stdin;
    initgraph(&init, &end);
    yyparse();

    fclose(yyin);

    Transformation(&init, &end, &tpinit, &tpend);
    if (argc > 0)
    {  printtypetbl(tpinit, argv[0], "w");
       printgraph(init, argv[0], "a");
    }
    t = tpinit;
    while (t != NULL)
    {  deltypetbl(&tpinit, &tpend, t);
       t = tpinit->next;
    }
    g = init;
    while (g != NULL)
    {  delgraph(&init, &end, g);
       g = init;
    }
}
```

APÊNDICE E

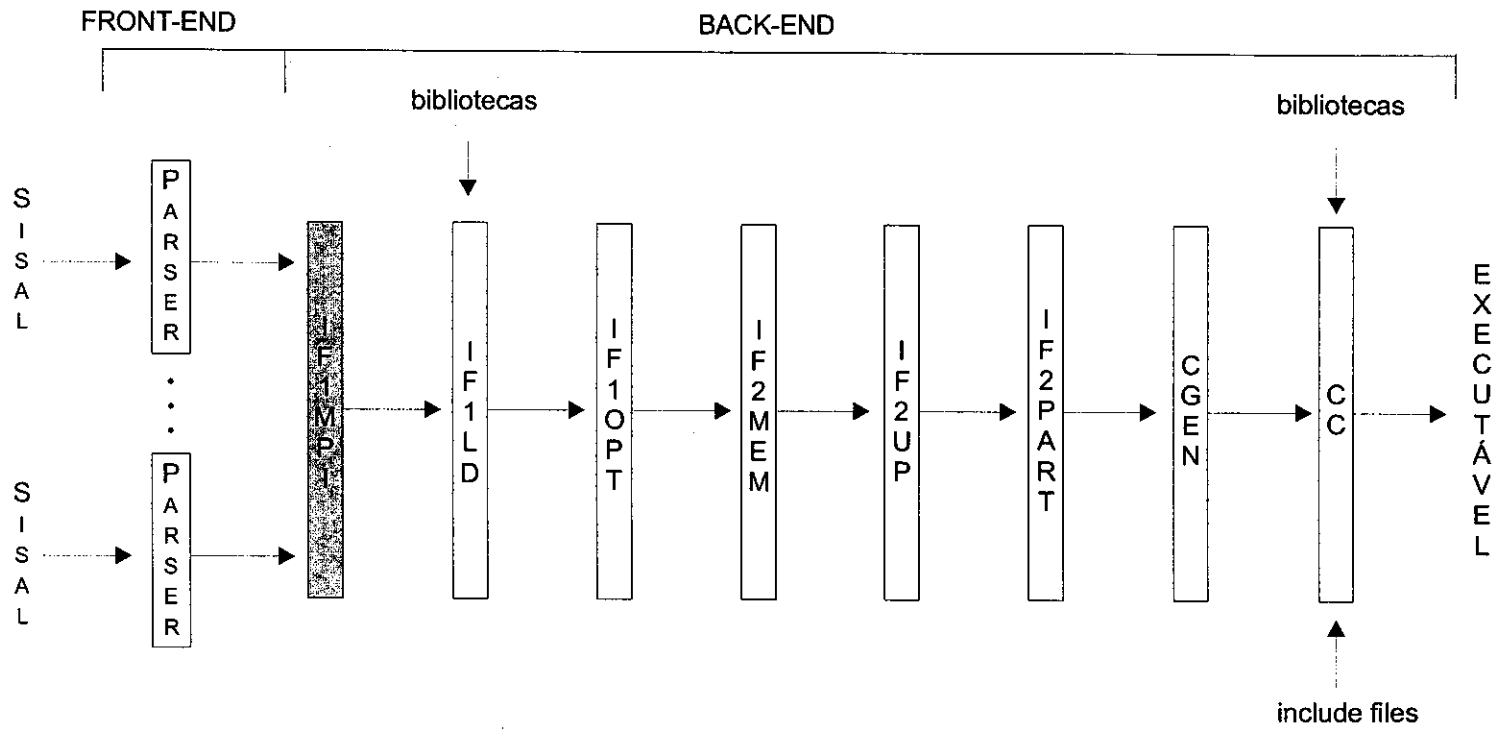
Ilustração do armazenamento do código IF1 para o código SISAL abaixo

```
define main

function main(a: array[array[array[integer]]]
              returns array[array[array[integer]]])

for i in 1, 10 cross j in 1, 10 cross k in i, 10
returns array of a[k,j,I]
end for

end function
```

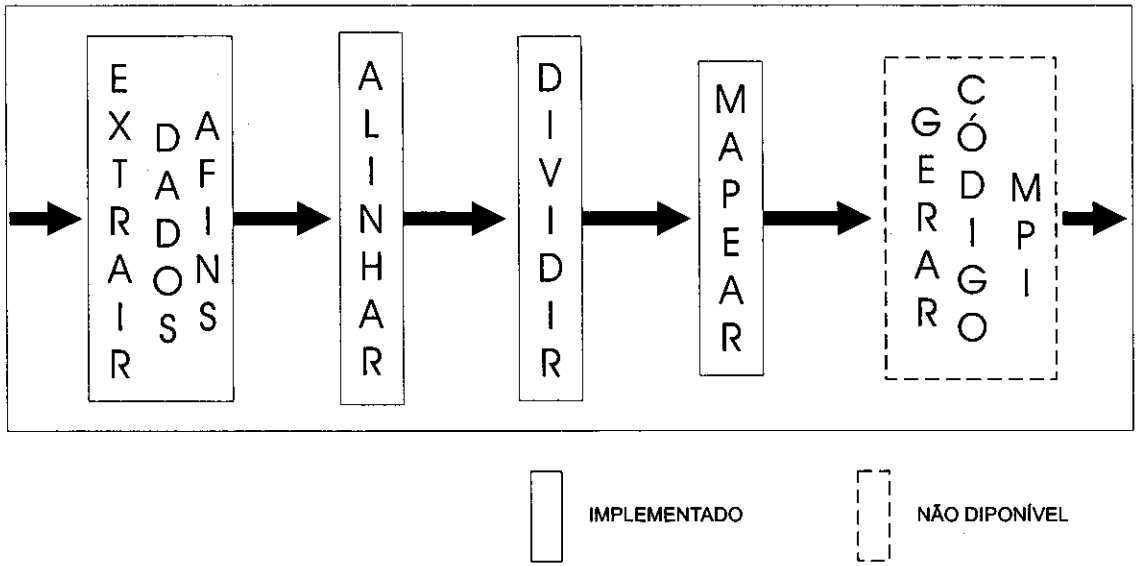


APÊNDICE F

Novo esquema do compilador OSC após a inserção do módulo para a otimização para a distribuição de dados e computações.



IF1MPI


EXTRAIR DADOS AFINS
ALINHAR
DIVIDIR
MAPEAR
GERAR CÓDIGO MPI
IMPLEMENTADO
NÃO DIPONÍVEL