UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE FÍSICA DE SÃO CARLOS
DEPARTAMENTO DE FÍSICA E INFORMÁTICA

"Paralelização de Programas SISAL para Sistemas MPI".

RAUL JUNJI NAKASHIMA

Dissertação apresentada ao Instituto de Física de São Carlos, Universidade de São Paulo, para obtenção do título de Mestre em Ciências "Física Aplicada-opção Física Computacional".

Orientador: Prof.Dr. Gonzalo Travieso

USP / IFQSC / SBI
8-2-001116

SÃO CARLOS SÃO PAULO
1996

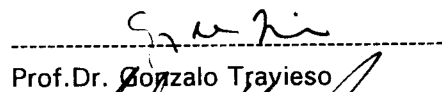
UNIVERSIDADE DE SÃO PAULO
Instituto de Física de São Carlos

Av. Dr Carlos Botelho. 1465
CEP 13560-250 · São Carlos · SP
Brasil

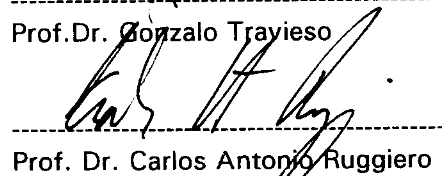
Fone (016) 272-6222
Fax (016) 272-2218

MEMBROS DA COMISSÃO JULGADORA DA DISSERTAÇÃO DE MESTRADO DE RAUL JUNJI NAKASHIMA APRESENTADA AO INSTITUTO DE FÍSICA DE SÃO CARLOS, DA UNIVERSIDADE DE SÃO PAULO, EM 15 DE MARÇO DE 1996.

COMISSÃO JULGADORA:



Prof.Dr. Gonzalo Travieso



Prof. Dr. Carlos Antonio Ruggiero

Profa.Dra.Liria Matsumoto Sato

Aos meus pais, Shimji e Satoy, pela dedicação e esforços e por sempre terem me incentivado nos estudos.

Agradecimentos

- Ao Prof. Gonzalo Travieso, meu orientador, sempre incansável, não medindo esforços para a realização deste trabalho; pela amizade. apoio e confiança depositados em mim e pelo integridade de caráter imprescindível para a realização de um trabalho conjunto.

- Ao CNPq pelo fornecimento de uma bolsa de estudo

- Aos amigos Patrícia Magna, Rafael Humberto Scapin e Regina Fumie Eto pelos manuais e materiais, imprescindíveis para o nosso trabalho

Sumário

LISTA DE FIGURAS I

LISTA DE TABELAS II

RESUMO III

ABSTRACT IV

1. INTRODUÇÃO 1

2. *SISAL* 4

2.1. Descrição de um programa em *SISAL* 5
2.2. Associação 7
2.3. Tipos 8
2.3.1. Tipos escalares e operadores padrão *8*
2.3.2. Arrays *8*
2.4. Funções e Expressões 12
2.4.1. Expressões *12*
2.4.2. Funções *13*
2.5. Expressões de seleção 14
2.6 Expressão *FORALL* 15
2.6.1. Arrays resultantes de expressões FORALL *16*
2.6.2. Produtos Dot e Cross *17*
2.7. Conclusão 18

3. *IF1* - FORMA INTERMEDIÁRIA PARA LINGUAGENS APLICATIVAS 19

3.1. Descrição geral de *IF1* 19
3.2. Formato de um arquivo *IF1* 20
3.2.1. Comentário *21*
3.2.2. Pragmas *22*
3.3. Descritores de tipo 22
3.3.1. Tipo básico *23*
3.3.2 Tipos array *24*
3.3.3 Tipo function *24*
3.4. Literais 25
3.5. Arcos direcionais 26
3.6. Fronteira do Grafo 27
3.7. Função 28
3.8. Nó 29
3.8.1. Nó simples *29*
3.8.2. Nó composto *30*
3.9. Conclusão 37

4. PADRÃO *MPI* (*MESSAGE PASSING INTERFACE*) 38

4.1. Visão Geral 38
4.2. Especificações do *MPI* 39
4.3. O Padrão *MPI* 40
4.3.1. Mensagens e processos 41
4.3.2. Tipos de dados do MPI 44
4.4. Comunicação 45
4.4.1. Comunicação ponto a ponto 45
4.4.2. Comunicação coletiva 47
4.5. Conclusão 53
5. PARALELIZAÇÃO DOS PROGRAMAS 54
5.1. Fonte de paralelismo e modelo de particionamento 54
5.2. Código fonte e fase do particionamento 55
5.3. *SISAL/IF1* mesclado com *C* 58
5.4. Interface entre *SISAL/IF1* e *MPI* 61
5.5. Transformação do programa *SISAL/IF1* 64
5.6. Algoritmo 67
5.7. Implementação e execução 68
5.8. CONCLUSÕES 71
6. CONCLUSÕES 72
6.1) SISAL/IF1 72
6-2) *MPI* 73
6-3) Propostas de alteração 73
6-3) Implementação 73
6-4) Trabalhos Futuros 74
BIBLIOGRAFIA 75
APÊNDICE A 79
APÊNDICE B 89

Lista de Figuras

Figura 1 - Grafo de (a+b)/5 **20**

Figura 2 - Programa *IF1* do *Exemplo 29* contendo um nó composto *Select* **33**

Figura 3 -Representação do programa *IF1* do *Exemplo* 30 para um nó composto *forall* **36**

Figura 4- Ilustração do *Exemplo 26* **48**

Figura 5 - Ilustração do *Exemplo 27* **49**

Figura 6 - Ilustração do *Exemplo 28* **51**

Figura 7 - Ilustração do *Exemplo 29* **52**

Figura 8 - *Slice* de um *forall* com 9 instâncias sendo dividido em 3 *forall* de 3 instâncias. **54**

Figura 9 - Ilustração das fases do compilador *OSC* **55**

Figura 10 - Particionamento *slice* criando processos *mestre* e *escravo* no modelo *SPMD* **64**

Lista de Tabelas

Tabela I - Código das entradas do descritor de tipo 23

Tabela II - Códigos para os tipos básicos definidos em *IF1* 23

Tabela III - Exemplos de literais 26

Tabela IV - Associação de portas para *Select* 32

Tabela V - Utilização de portas para *Select* 32

Tabela VI - Associação de portas para *Forall* 34

Tabela VII - Utilização de portas para *Forall* 34

Tabela VIII - Alguns tipos *MPI* e os equivalentes em *C* 43

Resumo

Este trabalho teve como finalidade a implementação de um método para a paralelização parcial de programas, escritos na linguagem funcional, *SISAL* utilizando as bibliotecas do padrão *MPI* (*Message Passing Interface*). Para tal, propusemos a transformação dos programas *SISAL* através do particionamento do *loop* paralelo *forall*, através do método de particionamento *slice* e a utilização do modelo de implementação do paralelismo *SPMD* (*Single Program Multiple Data*) no estilo de programas *mestre/escravo*. A validação de nossa proposta foi obtida através da realização de testes onde foram comparados os resultados obtidos com os programas originais e os programas com as alterações propostas.

Abstract

This work describe a method for the partial paralelization of *SISAL* programs into a programs with calls to *MPI* rotines. We focused on the paralelization of the *forall* loop (through slicing of the index range). The generated code is a *master/slave SPMD* program. The work was validated through the compilation of some simple *SISAL* programs and comparison of the results with an unmodified version.

Capítulo 1

1. Introdução

Os computadores seqüenciais modernos estão aproximando-se do máximo de desempenho possível, limitados pela tecnologia de interconexões metálicas cuja taxa máxima de *clock* sustentável é em torno de 200 a 250 MHz para uma geometria de condutor típica, Stone (1993). No entanto, vários problemas científicos correntes requerem o poder computacional de milhares destes computadores seqüenciais e a demanda está em crescimento. Esta situação tem conduzido os arquitetos de computadores a projetar sistemas de computadores multiprocessados na esperança do uso do poder computacional coletivo de múltiplos processadores para resolver esses problemas, Haines (1993).

Os computadores de memória distribuída representam hoje a mais poderosa classe de computadores e a melhor esperança para obtenção de um paralelismo redimensionável para a resolução de problemas *Grand Challenge* que enfrenta a comunidade científica, Haines (1993).

O uso de multiprocessadores de memória distribuída torna comum que usuários exijam ferramentas portáveis e eficientes para melhor explorar este ambiente e proteger seu investimento de *software*, Foisy; Chailloux (1995). Mas é difícil a criação de programas paralelos eficientes e corretos, isto porque cada arquitetura requer um programa diferente. Seria fácil se os programadores pudessem escrever programas que pudessem ser portáveis em uma variedade de processadores seqüenciais, vetoriais e paralelos, Freeh; Andrews (1995).

As linguagens funcionais, Barendregt; Leeuwen (1986), têm o potencial para prover uma solução simples, portável e eficiente para programação paralela. O uso de uma linguagem funcional freqüentemente reduz o esforço do programador pela simplicidade dos programas (nenhum gerenciamento explícito de concorrência ou memória), facilidade de manutenção (programas pequenos), facilidade para testar (resultados determinísticos) e facilidade de compreensão. Somado a isto, programas funcionais tem uma abundância de concorrência explícita, devido às iterações dos

loops e expressões independentes que podem ser avaliadas em paralelo, Freeh; Andrews (1995).

SISAL, Feo; Cann (1990), Skedzielewski (1991), Cann (1992b), é uma linguagem funcional que tem provado ser útil para a programação de aplicações numericamente intensivas, especialmente aplicações paralelas Freeh; Andrews (1995). O *OSC* (*Optimizing SISAL Compiler*), Cann (1992a), cria código para diversos processadores de memória compartilhada e vetoriais (por exemplo *Cray X-MP*, *Warp*, *Connection Machine*, *Mac II* e uma variedade de máquinas *dataflow*), Feo; Cann (1990). No entanto, ele não cria código para máquinas de memória distribuída, Freeh; Andrews (1995). Trabalhos estão sendo realizados na tentativa de suprir esta lacuna. Podemos citar Grit (1990), que descreveu como implementar *SISAL* em um ambiente de passagem de mensagem; Haines (1993), que desenvolveu um suporte para gerenciamento de dados e tarefas distribuídos de programas *SISAL*; Haines; Böhn (1993), que descreveram o projeto e performance inicial de um sistema de tempo de execução para *SISAL* em processadores de memória distribuída; Pande et al (1994), que propuseram um limiar para estratégia de escalonamento para *SISAL* em máquinas de memória distribuída e Freeh; Andrews (1995), que desenvolveram um protótipo de um compilador *SISAL* que suporta máquinas de memórias distribuídas e compartilhadas.

Esta dissertação apresenta um método para paralelização parcial de programas *SISAL* usando *MPI*, um padrão de interface para passagem de mensagem (paradigma largamente usado em certas classes de máquinas paralelas, especialmente aquelas com memória distribuída), Message Passing Forum (1994a).

Um padrão para passagem de mensagem é um componente chave na construção de um ambiente de computação concorrente no qual aplicações, bibliotecas de *software* e ferramentas possam ser usadas transparentemente entre diferentes máquinas, Message Passing Forum (1994a).

O modelo de programação *SPMD* (*Single Program Multiple Data*), Foisy; Chailloux (1995), Almasi; Gottlieb (1994), com processos executando no estilo *mestre/escravo* Haines; Böhn (1993), foi escolhido para receber as rotinas de comunicação coletiva de *MPI*. Para a transformação de um programa *SISAL* para o

modelo *SPMD* realizou-se o particionamento *slice*, Sarkar; Cann (1990), da estrutura *forall,* Skedzielewski (1991), de modo a inserir cada fatia obtida em um processo diferente. O código fonte manipulado foi o do formato *IF1*, Skedzielewski; Glauert (1985), Skedzielewski; Welcome (1985), representação em forma de grafos acíclicos da linguagem *SISAL*.

Para uma implementação do modelo proposto utilizamos o compilador *SISAL* do *Lawrence Livermore National Laboratories, OSC*, a implementação de *MPI*, do *Ohio Supercomputer Centre, LAM*, Burns et al (1994), Pacheco (1995), Dongarra (1995), e o compilador *GNU C* da *Free Software Foundation, Inc.*

O restante da dissertação está organizado da seguinte maneira: os três capítulos seguintes descrevem *SISAL, IF1* e *MPI.* O capítulo 5, descreve detalhes do modelo proposto e questões de implementação e o capítulo 6 apresenta as conclusões e sugestões para a extensão do trabalho de mestrado. No Apêndice B apresentamos a biblioteca de interface entre *SISAL* e *MPI* e no Apêndice B alguns exemplos de programas com as alterações propostas.

Capítulo 2

2. *SISAL*

A linguagem funcional *SISAL*, foi escolhida para o nosso trabalho em virtude principalmente do paralelismo inerente, presente em todos os níveis de comandos, englobando desde as mais simples operações ou as mais elaboradas como as estruturas condicionais ou *loops* até as chamadas de funções. Vários estudos de programas *SISAL* expuseram esta massiva quantidade de paralelismo potencial Sarkar; Cann (1990).

SISAL, algumas vezes chamada de linguagem de associação única, um caso especial de linguagens aplicativas, Skedzielewski (1991), Oldehoeft; Cann (1988), apresenta como outras características:

- As expressões e funções são mapeamentos de valor único de um intervalo para um domínio. Elas não são tão flexíveis como funções e subrotinas em linguagens convencionais que são baseadas em procedimentos ou comandos, Skedzielewski (1991);
- Independente do número de vezes que uma função será usada, nós sempre obteremos as mesmas respostas para os mesmos argumentos, Skedzielewski (1991);
- Nenhum efeito colateral indeterminado. Todas as entradas e resultados de uma função são locais, tornando fácil o controle e entendimento do fluxo dos programas. O único efeito colateral de uma função é o retorno de seu resultado no ponto de invocação, Skedzielewski (1991);
- A ausência de variáveis ou estados globais substituídos pela construção *valor-nome* que não se modifica dentro de seu escopo no decorrer da execução, Skedzielewski (1991).

O conceito do par *valor-nome* em *SISAL* é muito importante, pois programas *SISAL* não manipulam valores diretamente. A tarefa de associação de valores a

memória é deixado a cargo do compilador *SISAL*. Portanto, a performance de um programa *SISAL* dependerá pesadamente do sucesso do compilador na associação à memória . Os valores são manipulados através dos nomes a eles associados. Um nome pode receber apenas um valor em cada escopo. Após a associação, o nome torna-se um valor *read-only*, que pode ser compartilhado por qualquer número de consumidores. Esta propriedade elimina uma das maiores causas de indeterminismo: a condição de disputa que produz um comportamento aleatório na execução paralela, Skedzielewski (1991).

2.1. Descrição de um programa em *SISAL*

Um programa em *SISAL* é constituído por uma ou mais unidades que podem ser compiladas separadamente, Feo; Cann (1990).

A linguagem *SISAL* é *free-form* (sem restrições de posição ou formato) e *case insensitive* (não existe diferenciação entre letras maiúsculas de minúsculas), Cann (1992b); a regra da definição antes do uso é estritamente obrigatória e a tipagem de dados é rigorosa; comentários iniciam-se com o sinal de porcentagem (%) e terminam com o caracter de nova linha.

Um nome é visível em todo escopo de sua definição e qualquer escopo englobado excetuando-se qualquer redefinição. Não são permitidas definições múltiplas dentro de um mesmo escopo, pois isto violaria a regra de associação única de *SISAL*.

Podemos utilizar três tipos de unidades de compilação Cann (1992b) para uma aplicação em *SISAL*. São elas:

- *Programa* - pode conter um conjunto completo de definições de tipos e funções, sendo que uma das funções desta unidade podem iniciar uma computação.
- *Interface* - declara subconjuntos de tipos e funções públicos para o módulo a que pertence.
- *Módulo* - Inclui uma definição completa de tipos e funções exportadas e internas ao módulo .

Exemplo 1) Abaixo, apresentamos o exemplo de um programa *SISAL*. Os números que aparecem no início de cada linha não são partes integrantes do programa, apenas foram inseridos para possibilitar referências às linhas do programa, além dos comandos na linha.

```
1   %
2   % arquivo: exporta.sis
3   %
4   % definicao das funcoes exportadas Inc e Cubo
5     define Cubo
6
7   % declaracao da funcao local Sqr
8     function Sqr(x : real returns real)
9        % exp e' intrinsico a SISAL, aqui definindo x**2.0
10       exp(x, 2.0)
11    end function
12
13  % declaracao da funcao exportada Cubo
14    function Cubo(x: real returns real)
15       Sqr(x)*x
16    end function
17
1   %
2   % arquivo: principa.sis
3   %
4   % funcao que inicia a computacao
5     define main
6     % importacao da funcao Cubo declarada em um modulo
externo
7     global Cubo(x: real returns real)
8
9
10  % declaracao da funcao principal
11    function main(returns real)
12       Cubo(3.0)   %chamada da funcao Cubo
13    end function
```

Nesse exemplo, o primeiro bloco apresenta o arquivo fonte *SISAL* onde é declarada uma função exportada e uma função local. A palavra reservada **`define`** na linha 5, especifica a função que será exportada, no caso `Cubo`. Observe que na declaração de exportação não é realizada a definição dos parâmetros da função.

A função `Sqr`, cujo nome não se encontra na linha 5, é uma função local ao arquivo `exporta.sis`.

O arquivo `principa.sis`, realiza a importação da função exportada por `exporta.sis`. A palavra reservada `global`, na linha 7, define a função que esta sendo importada (`Cubo`), neste caso com a declaração dos parâmetros da função.

Na linha 5 temos a declaração da função principal **define main** (função *default*) que iniciará a execução do programa.

Apresentada a estrutura de um programa, passaremos agora às partes componentes dos módulos *SISAL* que diretamente estiveram envolvidas com o trabalho realizado. Uma referência mais detalhada de *SISAL* pode ser obtida em Feo; Cann (1990), Skedzielewski (1991), Cann (1992a) e Cann (1992b).

2.2. Associação

Em linguagens funcionais não existem operações de atualização à memória. Todos os programas devem satisfazer a *regra da associação única*, Skedzielewski (1991), que especifica que cada *nome* tem no máximo um *valor associado* à ele em tempo de execução. Sempre que um novo *valor* é computado, ele deve ser associado a um novo *nome* ou uma nova *instância do nome*. Como conseqüência, as linguagens funcionais produzem mais paralelismo que as linguagens com atualização de memória e outros efeitos colaterais Barendregt; Leeuwen (1986).

Uma operação de associação em *SISAL* tem o seguinte formato:

```
nome1,[nome2] ... := expressão1 [,expressão2] ...
```

Uma construção sintática conveniente para a associação de nomes de valores com subexpressões é a expressão **let** em cujo escopo de definições nomes podem ser usados livremente como uma representação abreviada, substitutos para a subexpressão que denotam.

Exemplo 2) Expressão `let` criando dois nomes de valor e retornando duas expressões

```
let
      X := 3.0;
      A := X*X;
in
      A*X, X
end let
```

Na primeira parte do **let** associamos o valor 3.0, real, ao nome X e o valor X*X, real, ao nome A; na segunda parte realizamos o retorno de dois valores 3.0*3.0*3.0 e 3.0. Ao associarmos o valor real 3.0 a X criamos um nome, do tipo real, que poderá ser referenciado em todo o escopo da construção **let**.

2.3. Tipos

2.3.1. Tipos escalares e operadores padrão

Os tipos escalares em *SISAL* são os tipos básicos familiares, encontrados nas outras linguagens de alto nível: **boolean**, **character**, **integer**, **real** e **double_real**.

Cada um dos tipos de dados escalares tem o seu próprio domínio de valores e são caracterizados por um conjunto de operações que criam e transformam os valores neste domínio. São suportados todos os operadores relacionais padrão (=, ~=, >, <, <=, >=) podendo ser aplicados a qualquer tipo escalar e retornando um valor *boolean* (**true** ou **false**), os operadores lógicos padrão (&, |, ~) e também, os operadores aritméticos padrão (+, -, *, /, **mod**, **exp**, **abs**, **max**, **min**). Os tipos dos resultados são os mesmo dos operadores e os tipos dos argumentos devem ser os mesmos, em operações com mais de um operador.

Em razão de não suportar conversão implícita de tipos, Cann (1992b), uma variedade de operações de conversão explícita são suportadas.

2.3.2. *Arrays*

Arrays em *SISAL* podem conter zero ou mais elementos que são indexados por inteiros, possuindo as seguintes características:

- Acesso direto: O tempo gasto para acessar qualquer elemento é constante, Skedzielewski (1991);
- Condensação densa: O *array* contém um conjunto de elementos que são indexados sobre uma extensão contígua de um limite inferior até um limite superior, Skedzielewski (1991);

- Tipo de elemento uniforme: Todos os elementos do *array* têm o mesmo tipo, Skedzielewski (1991);
- Tamanho dinâmico: O tamanho de um objeto *array* não é necessariamente conhecido até o momento da execução, Skedzielewski (1991).

A intenção de *arrays* em *SISAL* é fornecer acesso eficiente a *arrays* potencialmente grandes. Dado este conjunto de caracterizações, nós podemos usar aritmética de endereço e ponteiros para acessar um elemento diretamente, ao invés da utilização do método de acesso linear, como numa lista enlaçada. O fato dos limites do *array* variarem em tempo de execução, representa uma pequena sobrecarga, mas provê flexibilidade e permite *arrays* não retangulares. O compilador pode tomar nota de casos especiais para gerar o código mais eficiente possível, Cann; Evripidou (1995).

2.3.2.1. Construtores de *Array*

Os construtores de *arrays* criam *valores de arrays* de tamanho fixo.

A sintaxe é:

```
array [limite_inferior : valor, valor, ..]
```

e

```
array : tipo [limite_inferior : valor, valor, ...]
```

Exemplo 3) Alguns exemplos de construtores de *array*

- *Array* de inteiros de `1` a `9` e com limite inferior `1`
  ```
  array [1:1,2,3,4,5,6,7,8,9]
  ```
- *Array* de caracteres com os elementos `'S','t','r','i','n','g'` e `' '` e com limite inferior `1`
  ```
  array [1:'S','t','r','i','n','g',' ']
  ```
- *Array* de inteiros com os elementos `25,16,9,4,1,0,1,4,9,16` e `25` e com limite inferior `-5`
  ```
  array[-5:25,16,9,4,1,0,1,4,9,16,25]
  ```

Observamos que o limite inferior de um construtor de *array* deve ser uma expressão inteira e todos os elementos devem ser do mesmo tipo.

Uma maneira conveniente para criar *arrays*, cujo tamanho é conhecido em tempo de execução, inicializados com zero ou outro valor é através da função pré-definida

```
array_fill(Low, Length, Value)
```

onde `Low` é o limite inferior do *array*

`Length` é o número de elementos do *array* e

`Value` é o valor com a qual se preeencherá o *array*.

Às vezes podemos modificar um *array* pela troca de um ou mais elementos. Visto que não podemos modificar qualquer valor em *SISAL*, uma expressão de substituição de *array* retorna um outro *array* que difere de sua entrada.

Exemplo 4) Arrays de inteiros tendo seus "valores alterados"

```
let
      v1 := array[2:1.0,2.0];
      v2 := array[3:3.0,4.0,5.0];
in
      v2[4:0.0],          %produz array[3:3.0,0.0,5.0]
      v1[2:6.0,7.0],      %produz array[2:6.0,7.0]
      v1,                 %produz array[2:1.0,2.0]
      v2                  %produz array[3:3.0,4.0,5.0]
end let
```

Pela definição de *array*, podemos imaginar que no *Exemplo 4* que tenhamos copiado o *array* e mudado um valor, mas na prática os compiladores *SISAL* detectam circunstâncias onde a cópia de *array* não é necessária e emitem um código que simplesmente modifica o *array* original.

A concatenação de *array* é uma operação natural em *SISAL*, possuindo a seguinte sintaxe:

```
ArrayName1 || ArrayName2 || ...
```

onde o tipo base dos *arrays* `ArrayName1,ArrayName2,...` devem ser os mesmos. O limite inferior do *array* resultante é o mesmo de *ArrayName1*. Os elementos do *array* serão todos os elementos de ambos *arrays* com `ArrayName1`

seguido de ArrayName2 e assim por diante. Seguindo a mesma regra das outras operações com *arrays* o resultado é logicamente distinto de suas entradas.

Exemplo 5) Considerando o Exemplo 4 acima

```
v1 || v2
```

retornará o *array*

```
array[2:1.0,2.0,3.0,4.0,5.0]
```

2.3.2.2. Acesso ao *Array*

SISAL utiliza colchetes [] para acessar um elemento de um *array*.

Exemplo 6) Considerando o *Exemplo 4*

```
v1[2], v1[4]
```

retornam os valores 1.0 e 4.0 respectivamente.

2.3.2.3. *Arrays* multidimensionais

Array multidimensionais são simplesmente *arrays* de *arrays* com considerável liberdade quanto à forma do *array*. Os *arrays* não necessitam ser retangulares como é freqüente no caso de outras linguagens. Cada linha de um *array* multidimensional pode determinar o seu próprio comprimento e limite inferior; e se os limites não são uniformes o *array* é chamado dentado.

Exemplo 7) Construtores de arrays

```
let
        A := array[3:2.0,4.0,6.0];
        B := array[0:0.0,3.0];
in
        array[1:A,B,A]
end let
```

onde A e B são do tipo array[integer], enquanto o resultado da expressão **let** é um *array* bidimensional de nove elementos no total, mas cada linha da matriz possui uma dimensão diferente. Os elementos tem os subscritos [1,3], [1,4], [1,5], [2,0], [2,1], [3,3], [3,4] e [3,5].

2.4. Funções e Expressões

Funções e expressões são os blocos básicos de construção dos programas *SISAL*. Eles podem mapear qualquer número de entradas a qualquer número de saídas, garantindo para cada mapeamento um valor único. Podemos definir o principio da programação *SISAL* como:

"Dado um conjunto fixo de entradas, uma função sempre retornará o mesmo resultado", Skedzielewski (1991).

Uma razão para que tal garantia seja verdadeira é o fato de uma função não poder modificar o ambiente onde está inserida. Ela simplesmente utiliza sua entradas para computar um resultado. Todas as suas entradas são dadas explicitamente como parâmetros e o efeito da função é limitado ao retorno de um conjunto de valores.

2.4.1. Expressões

A expressão é a unidade sintática básica de *SISAL* e denota uma lista de valores de algum tipo. O tamanho de uma expressão é denominado *arity* (tamanho de sua lista de valores). Duas expressões são correspondentes se elas tem a mesma *arity* e valores do mesmo tipo na mesma ordem. Os tipos mais simples de expressões de *arity* um são constantes, nomes de valores ou resultados de operações ou expressões de *arity* um. A expressão de *arity* alta mais simples é uma série de expressões de *arity* um, separadas por vírgula.

As expressões *SISAL* utilizam uma combinação de operadores infixos e prefixos.

Exemplo 8) Abaixo alguns exemplos de expressões:

```
x := 1+2;
k,z,w := 1,2,3;
```

onde `x`, `1` e `2` são expressões *SISAL* de *arity* um e `k,z,w` e `1,2,3` é são expressões *SISAL* de *arity* três.

2.4.2. Funções

Uma definição de função consiste de um cabeçalho, seguido por uma lista de expressões que definem o corpo.

O cabeçalho da função declara o nome e o tipo de cada parâmetro formal para a função. As declarações de parâmetros são separadas por vírgula, mas cada declaração pode conter vários nomes, separados também por vírgula.

O escopo de um parâmetro formal é o corpo da função menos qualquer construtor interno que reintroduza o mesmo nome de valor. O tipo de cada valor retornado por uma função deve ser definido pela lista de especificação de tipo seguindo a palavra reservada `returns`. Esta lista de tipo deve corresponder à lista de expressões componentes do corpo da função.

O número de parâmetros reais passados para uma função deve ser igual ao número de parâmetros especificados no cabeçalho da função e seus respectivos tipos devem corresponder.

Uma função tem acesso apenas aos dados apresentados a ela através de seus parâmetros e os resultados das chamadas de funções podem ser passados diretamente para outras funções.

Uma forma simplificada de sintaxe para a declaração da função seria:

```
function nome_da_função (entradas returns saidas)
      <corpo da função>
end function
```

Exemplo 9) Dois exemplos de declaração de função

```
function Teste (x,y integer returns boolean)
      x < y
end function

function main(returns boolean)
   Teste(3,4)
end function
```

2.5. Expressões de seleção

A operação de seleção em *SISAL* é suportada pela construção **if**. Como em **let**, se a expressão **if** define um valor aritmético simples, ele poderá aparecer dentro de uma expressão aritmética. A forma geral para a construção **if** é:

```
if Expressão tipo Boolean
   then lista de expressões
   [elseif Expressão tipo Boolean
           then lista de expressões...]
   else lista de expressões
end if
```

As expressões que se seguem às palavras reservadas **if** e **elseif** são expressões de teste do tipo *boolean*. As listas de expressões seguindo as palavras reservadas **then** e **else** dão os braços da construção. Todos os braços devem ter a mesma *arity*, tipo e ordem.

Os valores produzidos pela construção **if** são os resultados do primeiro braço se a expressão de teste do tipo *boolean* for verdadeira ou pelo braço final se todas as expressões de teste forem falsas.

A avaliação da construção inicia-se com **if** e continua se necessário através dos **elseif** na ordem de apresentação.

A seguir alguns exemplos de construções **if**:

Exemplo 10) Expressões de seleção simples e compostas

- Construção **if** simples

```
n := if (x < 0.0D0)
        then 0.0D0
        else qmult
     end if
```

- Construção **if** composta

```
ib, ic := if (x >= xright)
             then 1.0D0, 3.0D0
             elseif (x < xleft)
                  then 2.0D0, 4.0D0
                  else 0.0D0, 2.0D0
          end if
```

2.6 Expressão *FORALL*

A expressão *forall*, não é propriamente uma iteração, pois inicia várias instâncias de corpos de *loop* independentes. De qualquer modo, a forma desta expressão em *SISAL* é muito similar a forma das expressões de iteração. Como cada instância é independente das outras, ela pode ser executada em paralelo sobre uma arquitetura que suporte paralelismo a nível de *loop*. Esta forma de paralelismo é também de fácil implementação em arquiteturas vetoriais, Skedzielewski (1991).

A única interação entre os corpos do *loop* ocorre na parte de retorno através da cláusula **return**.

Em *SISAL*, uma expressão *forall* é escrita como uma expressão **for** que possui um gerador de valores que pode ser usado dentro do corpo do *loop*. Abaixo o exemplo simples de um *loop* para o cálculo de uma soma:

Exemplo 11) Expressão *forall*

```
for I in 1, N
      T := I*I*I
returns value of sum T
end for
```

As expressões *forall* em *SISAL* usam três cláusulas para expressar as partes da iteração: gerador, corpo e retorno.

O gerador fornece os valores inicial e condição de teste para expressão, declara um nome de iteração e fornece a faixa de valores nos quais ele será limitado gerando corpos de *loop* independentes uns dos outros. Sua sintaxe é:

```
for nome in mínimo, máximo
```

A seção entre o gerador e a parte de retorno pode conter construções de nomes de valor. A construção de um nome deve seguir a regra de definição antes do uso.

Cada um dos nomes definidos no corpo da expressão *forall* pode participar na parte de retorno. Como nas expressões de iteração, a expressão *forall* pode retornar o valor de uma expressão. O retorno de valores é feito por:

```
return array of expressão
return value of expressão/
                expressão de redução
```

onde:

array of : empacota um **array**;
value of : retorna o último valor;
e
value of sum : retorna a soma do elementos
value of product: retorna o produto dos elementos
value of least: encontra o menor elemento
value of greatest: encontra o maior elemento
value of catenate: concatenação de **arrays** e **streams**

Exemplo 12) Expressão *forall* com exemplos de expressões de retorno

```
for i in 1,3
returns array of i                    % produz
      array[1:1,2,3]
         value of i                   % produz 3
         value of sum i               % produz 1+2+3 ou 6
         value of product i           % produz 1*2*3 ou 6
         value of catenate array[1:i] % produz array[1:1,2,3]
end for
```

2.6.1. *Arrays* resultantes de expressões *FORALL*

Os *arrays* podem ser retornados como um resultado de uma expressão *forall* através da cláusula **array of** *expressão* na parte de retorno.

Arrays produzidos por expressões *forall* terão cada um dos elementos contribuidos por cada uma das instâncias do corpo do *forall*. Os índices dos elementos do *array* variarão sobre os mesmos valores dos índices de cada *loop*. Por exemplo, na expressão *forall* **for** I **in** 1,N, o limite inferior de qualquer *array* produzido por esta expressão será 1 e existirão no máximo N elementos no *array*.

O número de elementos no *array* será o mesmo que o número de instâncias do corpo do *forall*.

Exemplo 13) Expressão *forall* retornando um *array* de *real*

```
for I in L,H
      A := sin(2.0*Pi/360.0 + real(I-L)/real(H-L))
returns array of A*A
end for
```

Para produzir *arrays multidimensionais*, basta aninharmos expressões *forall*. Os *loops* aninhados que retornam *arrays*, juntam uma linha de cada *loop* interno para dentro de um *array* no nível mais externo.

2.6.2. Produtos *Dot* e *Cross*

Outras duas construções para a manipulação com *arrays* são os produtos **dot** e **cross**.

Com a operação **dot** podemos introduzir uma ou mais seqüências de índices em uma expressão *forall*. A sintaxe é:

```
for Nome in Intervalo1 [dot Intervalo2]...
```

onde todos os intervalos devem ter o mesmo número de valores.

Exemplo 14) Um *forall* gerando índices utilizando a construção dot

```
for I in 1,4 dot j in 5,8
returns array of i+j
end for
```

produz **array**[1:6,8,10,12] (**ou array**[1:1+5,2+6,3+7,4+8])

A operação **cross** fornece-nos um produto cartesiano de seqüências que define um aninhamento implícito de expressões *forall*. A forma

```
for Nome in Intervalo1 [cross Intervalo2]
```

criará pares (ou trios, quadras, etc) de construções de nome-valor.

Exemplo 15) Um *forall* gerando seus índices utilizando a construção cross

```
for i in 1,2 cross j in 1,2
returns array of 1.0/real(i+j-1)
end for
```

define a matriz de *Hilbert*, Skedzielewski (1991), sendo equivalente à

```
for i in 1,2
```

```
        Row := for j in 1,2
                   returns array of 1.0/real(i+j-1)
                   end for
    returns array of Row
    end for
```

produzindo

```
array[1:array[1:1.0/1.0,1.0/2.0],array[1:1.0/2.0,1.0/3.0]]
```

2.7. Conclusão

Neste capítulo apresentamos uma breve introdução à *SISAL* exibindo as principais características da linguagem e construções utilizadas em nosso trabalho. Dentre todas as construções, uma maior atenção foi voltada para a expressão *forall* devido a esta estrutura ser uma das peças chaves para o modelo de paralelização utilizado.

Capítulo 3

3. *IF1* - Forma Intermediária para Linguagens Aplicativas

Como forma alternativa para a manipulação de programas *SISAL* podemos utilizar a linguagem de grafo acíclico *IF1*. Esta é uma forma intermediária para linguagens funcionais fortemente baseada nos aspectos de linguagens de associação única. Todas as implementações de *SISAL* produzem o código *IF1*, Oldehoeft; Cann (1988).

O código *IF1* não é útil apenas para separar o *front-end* do compilador do gerador de código, mas esta é uma forma lida por muitos programas de análises. Otimizações independentes de máquina tais como eliminação de subexpressão comum e remoção de invariante de *loop* operam sobre os grafos expressos em *IF1*. Análises dependentes de máquina tais como encontrar operações com vetor ou particionamento do grafo para multiprocessadores também serão realizados usando um superconjunto de *IF1*, Skedzielewski; Glauert (1985).

IF1 representa *SISAL* da seguinte forma: grafos representam as funções, nós simples representam as operações; nós compostos representam estruturas condicionais, acesso a um campo de uma *union* e estruturas de repetição; arcos representam os nomes; literais representam as constantes e tipos correspondentes para os tipos de *SISAL*.

Um benefício resultante da utilização do grafo *IF1* é que os algoritmos, utilizando as técnicas convencionais de otimização baseadas em análises dos grafos, são pequenos, fáceis de entender e bastante rápidos, Skedzielewski; Welcome (1985).

3.1. Descrição geral de *IF1*

Um programa *IF1* é uma hierarquia de grafos de fluxo de dados acíclicos onde os nós denotam as operações e os arcos direcionais transportam os dados, Skedzielewski; Welcome (1985).

O grafo é uma coleção de nós simples ou compostos conectados através de suas portas por arcos direcionais identificados pelo número do nó e porta de seu produtor e consumidor, podendo conter opcionalmente um tipo, caso das linguagens fortemente tipadas como *SISAL*. Um tipo especial de arco, denominado literal, é usado para valores constantes sendo identificado pelo número do nó destino, o número da porta do nó destino e o valor a ser passado representado por uma *string*.

Os nós simples denotam operações elementares onde as saídas são funções diretas das entradas. *IF1* possui em torno de 50 nós simples, Skedzielewski; Glauert (1985), que representam por exemplo operações aritméticas ou booleanas.

Os nós compostos contém subgrafos e as saídas são dependentes das interações entre esses subgrafos. Os cinco tipos de nós compostos disponíveis em *IF1* são *Select*, *Tagcase*, *Forall*, *Loop while* e *Loop until*, Skedzielewski; Glauert (1985).

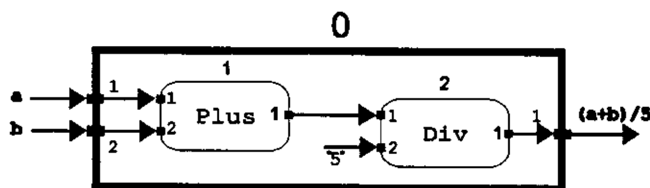


Figura 1 - Grafo de `(a+b)/5`

Podemos observar pela Figura 1 a fronteira do grafo representada pela linha mais externa, englobando os nós, com um conjunto de portas de entrada (`a` e `b` ou `1` e `2`) e saída (`(a+b)/5` ou `1`) servem como canais de comunicação entre os nós do grafo e o exterior do grafo.

Os tipos básicos em *IF1* incluem *boolean*, *character*, *integer*, *real* e *double*, *arrays* (dinamicamente extensíveis), *streams*, *records* e *unions* são usados para construir tipos mais complexos, Skedzielewski; Glauert (1985).

3.2. Formato de um arquivo *IF1*

Os arquivos *IF1* são compostos por caracteres *ASCII* imprimíveis dispostos em linhas; o primeiro caractere não branco de uma linha indica a entidade que está

sendo representada, nós, arcos, etc. Abaixo exibimos e descrevemos os caracteres que podem iniciar uma linha em um arquivo *IF1*.

`C`	comentário	`{, }`	nó composto (início e fim)
`E`	arcos direcionais	`T`	definição de tipo
`L`	arco direcional de literal	`G, X, I`	grafo
`N`	nó simples		

O resto de cada linha pode compreender campos que são delimitados por espaço em branco ou tabulações. O número de campos em cada linha dependerá do tipo da linha. Por exemplo, os arcos serão sempre compostos por cinco campos *nó origem, número da porta no nó origem, nó destino, número da porta no nó destino* e *tipo do dado.*

Nós e tipos possuem campos *label* representados por números inteiros. Os *labels* para nós e tipos não necessitam ser diferentes, pois pelo contexto é possível a distinção de um nó e um tipo.

Os *labels* dos tipos existem em um escopo global, devendo portanto ser únicos dentro de um arquivo e *labels* de nós necessitam ser únicos apenas dentro de um grafo.

3.2.1. Comentário

Iniciando com o caractere `C`, os comentários podem aparecer no final de qualquer linha em um arquivo *IF1* ou no início da linha.. Excetuando o caracter de nova linha, os demais caracteres *ASCII* imprimíveis podem ser usados em um comentário.

Outra forma de comentário são as *stamps*, linhas iniciadas com `C$`. Elas são usadas para marcar um arquivo *IF1* registrando algum processamento que já tenha sido realizado. Um único caractere segue `C$` para distinguir o tipo da *stamp*.

3.2.2. *Pragmas*

Pragmas podem aparecer no final das linhas nos campos de comentários. Eles são usados para registrar a linha no arquivo fonte que gerou o arco, dar o nome do arco, etc. A forma é:

`%aa=<algo>`

onde **aa** é um código de dois caracteres e `<algo>` pode ser terminada por *branco, tab, formfeed* ou *newline.* Caso mais que um `<algo>` seja necessário, vírgulas devem ser usadas como separador.

Os *pragmas* mais comumente gerados são:

%na - para o nome de valor a que corresponde o arco

%sl - número da linha no arquivo fonte de onde foi gerada a linha *IF1*

%mk - marca um arco com "*por referência*" (**%mk=**R) ou "*por valor*" (**%mk=**V)

Podemos ter por exemplo **%na=**`start` é um *pragma* para o nome `start`, **%mk=**V representa a manipulação de `start` por valor e **%sl=**9 é o número da linha no arquivo fonte de onde foi gerada essa linha de *IF1*.

3.3. Descritores de tipo

Os descritores de tipo são compostos por entradas para os tipos básicos e entradas para tipos estruturados (*arrays, stream, records* e *unions*). Eles são organizados como coleções de listas encadeadas que associam construtores a tipos e funções aos argumentos e resultados. O formato da declaração de um tipo pode ser composto por três ou quatro campos além de comentário e *pragma.* As declarações com três campos são as seguintes: *array*, tipo básico, valor múltiplo, *record, stream* e *union.*

A seguir a tabela com os códigos para entradas dos descritores de tipo.

Tabela I - Código das entradas do descritor de tipo

Entrada	Código	Entrada	Código
Array	0	*Record*	5
Basic	1	*Stream*	6
Field	2	*Tag*	7
Function	3	*Tuple*	8
Multiple	4	*Union*	9

3.3.1. Tipo básico

A declaração de um tipo básico em *IF1* possui a seguinte sintaxe:

```
T <label> <descriptor entry> <basic type> <comment>
```

onde

- `T` indica que a linha representa a declaração de tipo;
- `<label>` é o *label* que está sendo associado ao tipo;
- `<descriptor entry>` entrada do descritor de tipo, `1` (um) para a declaração de um tipo básico (Tabela I, Entrada *Basic*);
- `<basic type>` pode conter qualquer um dos valores da tabela abaixo com os códigos para os tipos básicos;
- `<comment>` comentário.

Tabela II - Códigos para os tipos básicos definidos em *IF1*

Tipo	*boolean*	*character*	*double*	*integer*	*null*	*real*
Código	0	1	2	3	4	5

Exemplo 16) Declaração dos tipos básicos de *IF1*

```
C       descr basic
C label entry type   comment
T   1     1    0     %na=Boolean
T   2     1    1     %na=Character
T   3     1    2     %na=Double
T   4     1    3     %na=Integer
T   5     1    4     %na=Null
T   6     1    5     %na=Real
```

Podemos observar no exemplo que a terceira coluna mantém valores constantes para todas as linhas de declaração de tipo. Isto era de se esperar pois o exemplo está apresentando a declaração de tipo básico, a segunda coluna varia porque está representando `<label>`, que não devem ser repetidos, a quarta coluna indica os tipos básicos que estão sendo declarados (Tabela II) e a última coluna apresenta *pragmas* com os nomes dos tipos declarados.

3.3.2 Tipos *array*

A sintaxe para a declaração de um *array* é dada abaixo:

```
T <label> <descriptor entry> <base type> <comment>
```

onde

- `T` indica que a linha representa uma declaração de tipo;
- `<label>` é o *label* que está sendo associado ao tipo;
- `<descriptor entry>` entrada do descritor de tipo, 0 (zero) para *array*;
- `<base type>` pode possuir qualquer tipo já declarado;
- `<comment>` comentário.

Exemplo 17) Utilizando os tipos do *Exemplo 16* a declaração de dois *arrays*

```
C       descr  base
C label entry  type comment
T  76   0       6   %na=array[reals]
T  77   0      76   %na=array[array[real]]
```

3.3.3 Tipo *function*

Funções compreendem uma lista de argumentos e uma lista de resultados que são compostas de *tuplas*. A sintaxe para o tipo função é:

```
T <label> <descriptor entry> <arguments> <results> <comment>
```

onde

- `T` indica que a linha representa uma declaração de tipo;
- `<label>` é o *label* que está sendo associado ao tipo;

- `<descriptor entry>` entrada do descritor de tipo, 3 para *function* (Tabela I);
- `<arguments>` aponta para a lista de argumentos;
- `<results>` aponta para a lista de resultados;
- `<comment>` comentário .

e para o tipo *tupla*

```
T <label> <descriptor entry> <tuple type> <next tuple> <comment>
```

onde

- `T` indica que a linha representa uma declaração de tipo;
- `<label>` é o *label* que está sendo associado ao tipo;
- `<descriptor entry>` entrada do descritor de tipo, 8 para *tuple* (Tabela I);
- `<tuple type>` é o tipo da *tuple* (tipos básicos + tipos do usuário)
- `<next tuple>` aponta para o próxima *tuple* da *função*, 0 (zero) representa a inexistência de próxima *tuple*.
- `<comment>` comentário .

Exemplo 18) Declaração de um tipo função utilizando os tipos do *Exemplo 22*

```
C function f(x, y :integer returns integer)
C       descr tuple next
C label entry type  tuple
T  10     8     4     0
T  11     8     4     10
C       descr
C label entry args  results
T  12     3    11     10
```

3.4. Literais

A notação de literais segue o formato utilizado na linguagem *SISAL* onde os nomes das funções são *strings* de caracteres alfanuméricos, não existindo a diferenciação de maiúsculas e minúsculas, booleanos são `T` ou `F`, inteiros são *strings* de dígitos decimais, ponto flutuante de precisão simples são *strings* de dígitos decimais que contêm o ponto decimal ou estão na notação exponencial com `E` ou `e`, ponto flutuante em precisão dupla usam a notação exponencial com um `D` ou `d`,

caracteres são caracteres imprimíveis entre aspas ou são representações imprimíveis de caracteres não imprimíveis usados em *SISAL* e valores nulos são denotados por *nil*.

Tabela III - Exemplos de literais

Tipo	*function*	*integer*	*real*	*double*	*character*
Exemplo	"sqrt"	"137"	"2.7182"	".0221D2"	"x"

3.5. Arcos direcionais

Os arcos representam a dependência de dados explícita dentro do grafo. Eles podem dar informações sobre o tipo do valor que eles representam e também carregar informação extra no seu campo de comentário. A informação extra pode ser o nome associado com o arco ou a linha no arquivo fonte que gerou o arco.

Nem todas as dependências são explicitamente dadas pela lista de arcos, algumas estão implícitas na semântica, como para um nó composto e seus subgrafos. Essas conexões implícitas estão descritas na semântica de cada nó composto.

Um arco é representado por uma *6-tupla* composta por:

```
E <source node> <port> <destination node> <port> <type> <comment>
```

onde:

- <source node> número do nó fonte de onde origina-se o arco;
- <port> número da porta do nó fonte de onde origina-se o arco;
- <destination node> número do nó destino para arco;
- <port> número da porta do arco no nó destino;
- <type> tipo do valor transportado pelo arco;
- <comment> comentário.

e um arco para um literal é dado por uma 5-tupla:

```
L <destination node> <port> <type> <string>
```

onde

- `<destination node>` número do nó destino para o arco;
- `<port>` número da porta do arco no nó destino;
- `<type>` tipo do valor transportado pelo arco;
- `<string>` denota o valor para o literal.

Exemplo 19) Na Figura 1, página 20, observarmos três nós de grafo. O grafo com *label* 0 engloba dois outros *Plus* e *Div* de *label* 1 e 2 respectivamente. Tomando este exemplo e a declaração de tipo do *Exemplo 16*, temos o seguinte código *IF1* para denotar os 5 arcos direcionais entre os nós:

```
C  source    destin          comment/
C node port node port type string
E  0    1    1    1    4    %na=a
E  0    2    1    2    4    %na=b
C  source    destin          comment/
C node port node port type string
E  1    1    2    1    4
L            2    2    4     "5"
E  2    1    0    1    4
```

3.6. Fronteira do Grafo

Uma fronteira de um grafo é representada por uma linha iniciada com **G** e uma referência a um tipo. O escopo do grafo estende-se até a ocorrência de uma nova fronteira de grafo **G, X, I** ou **}**. A sintaxe é:

```
G <type reference>
```

onde:

- **G** indica o início da fronteira de um grafo
- `<type reference>` é o tipo do grafo

Subgrafos dentro de um nó composto podem ser sem tipo, denotado por zero no campo de tipo. O `<type reference>` atribui um tipo às fronteiras de grafos que representam funções. As fronteiras de grafos que denotam funções tem as seguintes sintaxes:

`G <type reference> "name"` para definição de função local

```
X <type reference> "name"
```
para definição da função principal

```
I <type reference> "name"
```
para função importada

Uma fronteira de grafo que representa uma função importada meramente associa um descritor de tipo a um nome de função.

A fronteira de um grafo serve como um coletor de valores importados e exportados para e dos arcos internos ao grafo. A fronteira é sempre implicitamente rotulada com `0` (*label*). Quando um valor externo à fronteira do grafo é necessário, ele é obtido referenciando-se um arco fonte para o nó com o *label* `0`. Similarmente, quando um valor deve ser passado para fora da fronteira do grafo referenciamos um arco destino para o nó `0`. Isto significa que dentro do corpo do grafo, a numeração de nós e subgrafos deve iniciar a partir de `1` (um).

3.7. Função

Uma função é representada por um grafo que contém o corpo da função. Em uma linguagem que não importa valores globais, as importações do grafo correspondem aos argumentos da função e os resultados do grafo correspondem aos resultados da função. A passagem de valores para o grafo ocorre na chamada da função e o retorno dos resultados no retorno da função.

Como visto acima, temos três formas de representar uma função. Destas apenas as duas primeiras representam uma função completa pois a terceira forma está associada ao cabeçalho das funções importadas de outros módulos. A forma,

```
I <type reference> "name"
```

é oferecida para que seja possível a verificação de tipos entre a unidade de compilação, ou seja módulo principal e módulos com funções importadas.

Exemplo 20) Os três tipos de representações de funções

```
C global subtrai(a, b: integer returns integer)
C function soma(a, b: integer returns integer)
C function main(a, b :integer returns integer, integer)
C ...
C       descr  tuple next
```

```
C label entry  type  tuple
T  9     8      4     0
T  10    8      4     9

C        descr
C label entry  args results
T  11    3      10    9
T  12    3      10    10
C   type    name
I    11     "subtrai"
G    11     "soma"
C corpo do grafo
...
X    12     "main"
C corpo do grafo
```

3.8. Nó

3.8.1. Nó simples

A semântica dos nós simples descreve a relação entre suas entradas e saídas. Os nós mais simples tem um número fixo de portas de entrada e saída e os mais complexos um número variado de entradas manipulando tipos de dados estruturados.

Os nós simples não são necessariamente simples, apesar de não terem subgrafos. O tipo da operação pode ser determinado examinando-se os tipos dos arcos de entrada e saída. A sintaxe é:

```
N <label> <node number>
```

onde:

- **N** indica que a linha representa um nó;
- `<label>` é o *label* que está sendo associado ao nó;
- `<node number>` número do nó desejado, pré-definido em *IF1*, Skedzielewski; Glauert (1985).

Podemos citar os nós simples aritméticos, booleanos, manipuladores de *array*, manipuladores de valores múltiplos, manipuladores de funções e mais alguns outros.

A totalidade das definições dos nós está disponíveis em, Skedzielewski; Glauert (1985).

Exemplo 21) O nó *Plus*, cujo *label* é `141`, recebe dois valores de entrada (*integer, real, double* ou *boolean*) e retorna um valor (*integer, real, double* ou *boolean*).

```
C  ... := x+y
C label number
N   1    141
C  source     destin         comment
C node port node port  type  string
E   0    1    1    1    4    %na=x        %mk=V  %sl=6
E   0    2    1    2    4    %na=y        %mk=V  %sl=6
E   1    1    0    1    4                 %mk=V
```

3.8.2. Nó composto

Os nós compostos contém, além de subgrafos, uma semântica que define o modo como os subgrafos interagem entre si. Os nós compostos coletam quaisquer valores necessários para seus subgrafos, devendo ter então um número arbitrário de portas de entrada e saída.

A semântica de cada nó composto relaciona suas entradas e saídas às entradas e saídas dos subgrafos. A representação de um nó composto é:

```
{
G 0
.
.
.
G 0
...
} <label> <OperationCode> <k a1 a2 ..ak>
<arcos de entrada>
<arcos de saída>
```

A primeira linha meramente marca o início do nó composto. Cada subgrafo inicia com **G** seguido de `0` zero (sem tipo) e termina com um outro **G** ou um **}** (final de algum nó composto). A linha com **}** "fecha" o nó composto dando o *label* do nó, `<OperationCode>` é o código para o nó composto e `<k a1 a2 ..ak>` é chamado *lista de associação.*

A *lista de associação* encadeia os subgrafos que estão descrevendo a semântica do nó composto. Os subgrafos de um nó composto podem aparecer em qualquer ordem no texto. A *lista de associação* define o mapeamento entre a ordem textual dos subgrafos e a ordem como será usado pelo nó composto. Como cada nó

composto usa os subgrafos diferentemente, a *lista de associação* tem um significado diferente para cada classe de nós compostos.

Cada um dos arcos utiliza o nó `0` zero para identificar as entradas `<arcos de entrada>` e saídas `<arcos de saída>` do subgrafo (*imports* e *results*).

Dos nós compostos definidos em *IF1* exibiremos apenas os dois tipos utilizados em nosso trabalho. Os demais nós compostos podem ser estudados em, Skedzielewski; Glauert (1985).

Quando discutirmos valores que são passados para os subgrafos dos nós compostos, referenciaremos estes como *classes de valores* tais como *valores importados*. Os membros de cada classe de valores estarão associados a números de portas contíguos nas fronteiras dos subgrafos. Assim, podemos definir uma *classe de portas* na fronteira de um subgrafo. Duas tabelas de portas serão dadas para cada nó composto. A primeira associará números de portas a cada classe e a segunda define que portas podem ser usadas como importadora e resultado.

3.8.2.1 *Select*

O nó *Select* é usado para implementar uma seleção de múltiplas vias. Apenas uma seleção de duas vias existe em *SISAL* (**`if-then-else`**). Em um nó *Select* podem existem `N+1` subgrafos: um *seletor* (ou *predicado*) e `N` *alternativas*. O seletor retorna exatamente um valor entre `0` e `N-1`, que determina qual subgrafo de resultado deve ser usado. Os seletores booleanos serão mapeados para inteiros pelo nó *Int*, *falso* para `0` e *verdadeiro* para `1`.

A *lista de associação* na linha de fechamento do nó composto identifica o *selector* e as *alternativas*. O contador de subgrafos deve ser pelo menos 3, visto que o nó *Select* necessita de um *seletor* e pelo menos duas *alternativas*. O primeiro elemento da *lista de associação* identifica o subgrafo *seletor*.

<u>Dependências implícitas</u>

- Todas as portas de entrada para o nó *Select* estão conectadas às portas de entrada correspondentes de todos os seus subgrafos (*classe K*).
- As portas de resultado de cada subgrafo para as *alternativas* estão similarmente conectadas às correspondentes portas do nó *Select* (*classe R*)

Tabela IV - Associação de portas para *Select*

Classe	Utilização	Results
K	valores importados pelo nó composto	1..n_K
R	valores dos resultados	1..n_R
S	valor do *selector*	1..1

Tabela V - Utilização de portas para *Select*

	Select	seletor	*alternativa*
Importações	*K*	*K*	*K*
Resultados	*R*	*S*	*R*

Exemplo 22) Abaixo código de um programa *SISAL* contendo uma expressão de seleção seguido do código *IF1* equivalente e figura do código *IF1*.

- **Código *SISAL***

```
define main
function main(A,B,C,D :integer returns integer, integer)
   if A+B < C+D then
      A,B
   else
      C,D
   end if
end function
```

Código *IF1*

```
T 1 1 0              %na=Boolean
T 2 1 1              %na=Character
T 3 1 2              %na=Double
T 4 1 3              %na=Integer
T 5 1 4              %na=Null
T 6 1 5              %na=Real
T 7 1 6              %na=WildBasic
T 8 10
T 9 8 4     0
T 10 8      4     9
T 11 8      4     10
T 12 8      4     11
T 13 3      12    10
X     13    "main"                                       %sl=3
N 1   141                                                %sl=4
E     0 1   1 1   4     %na=a          %mk=V             %sl=4
E     0 2   1 2   4     %na=b          %mk=V             %sl=4
N 2   141                                                %sl=4
E     0 3   2 1   4     %na=c          %mk=V             %sl=4
E     0 4   2 2   4     %na=d          %mk=V             %sl=4
N 3   131                                                %sl=4
E     1 1   3 1   4                    %mk=V
E     2 1   3 2   4                    %mk=V
```

```
N 4    129
E      3 1   4 1   1                           %mk=V
{ Compound    5    1
G      0
E      0 1   0 1   4                           %mk=V
G      0                                                   %sl=6
E      0 4   0 1   4          %na=c            %mk=V       %sl=7
E      0 5   0 2   4          %na=d            %mk=V       %sl=7
G      0                                                   %sl=5
E      0 2   0 1   4          %na=a            %mk=V       %sl=5
E      0 3   0 2   4          %na=b            %mk=V       %sl=5
} 5 1 3 0 1 2                                              %sl=5
E      4 1   5 1   4                           %mk=V
E      0 1   5 2   4          %na=a            %mk=V       %sl=5
E      0 2   5 3   4          %na=b            %mk=V       %sl=5
E      0 3   5 4   4          %na=c            %mk=V       %sl=7
E      0 4   5 5   4          %na=d            %mk=V       %sl=7
E      5 1   0 1   4                           %mk=V
E      5 2   0 2   4                           %mk=V
```

- **Representação do grafo *IF1***

Abaixo temos uma figura representando o nó composto *Select* onde os retângulos nas fronteiras dos grafos, subgrafos ou nós, sem arcos representam as dependências de dados implícitas e aqueles com arcos representam as dependências de dados explícitas. A correspondência dos dados implícitos é representada pelo tipo de preenchimento dos retângulos.

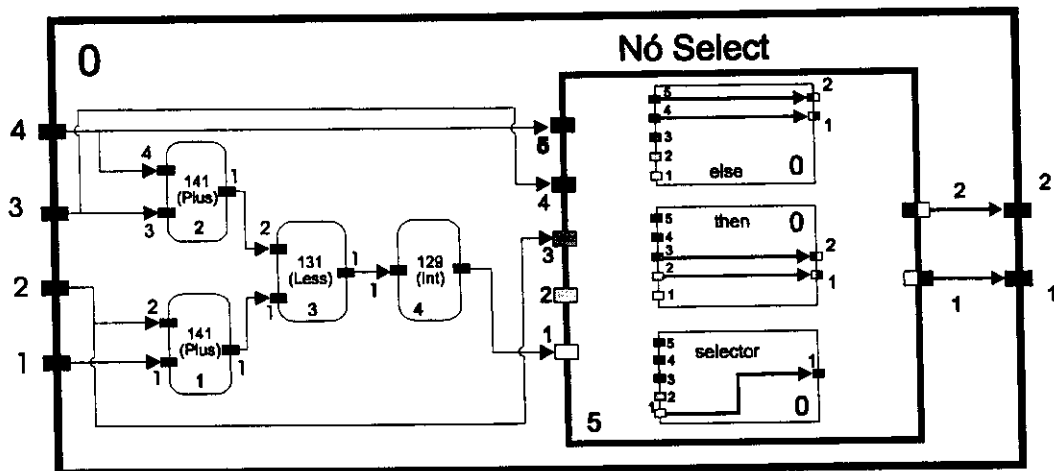


Figura 2 - Programa *IF1* do *Exemplo 29* contendo um nó composto *Select*

3.8.2.2. *Forall*

O nó *Forall* (equivalente ao *forall* de *SISAL*) é usado para denotar a execução independente de múltipla instâncias de uma expressão através de três subgrafos: *Gerador*, *Corpo* e *Resultado*.

O *Gerador* produz valores para cada instância do *Corpo*, que contém a expressão a ser avaliada. Pelo menos um valor múltiplo nas portas da *classe M* é produzido pelo *Gerador*. Cada elemento de um valor múltiplo é enviado para uma instância distinta do *Corpo*. Os valores e instâncias do *Corpo* estão ordenados e o subgrafo de *Resultado* coletará os resultados utilizando a mesma ordem. Quando mais que um valor múltiplo é produzido pelo *Gerador*, o primeiro valor em cada *classe M* é enviado para a primeira instância do corpo e assim por diante.

Dependência implícita

- Todas entradas para o nó *Forall* estão disponíveis para cada subgrafo nas portas da *classe K*.
- O subgrafo *Corpo* recebe um valor de cada porta da *classe M*.
- Os resultados do *Corpo*, *classe T*, estão conectados às entradas do subgrafo de *Resultado*.
- Os resultados do subgrafo de *Resultado*, *classe R*, estão conectados à portas de saída do nó *Forall*.

Tabela VI - Associação de portas para *Forall*

Classe	Utilização	Results
K	valores importados pelo nó composto	$1..n_K$
M	valores múltiplos	$n_K+1..n_K+n_M$
T	resultados de cada *Corpo*	$n_K+n_M+1..\ n_K+n_M+n_T$
R	resultados do nó *Forall*	$1..n_R$

Tabela VII - Utilização de portas para *Forall*

	Forall	*Gerador*	*Corpo*	*Resultado*
Importações	K	K	K, M	K, M, T
Resultados	R	M	T	R

Exemplo 23) Abaixo código de um programa *SISAL* contendo uma expressão *forall* seguido do código *IF1* equivalente e figura do código *IF1*.

- **Código *SISAL***

```
define main
function main(Low, High :integer
              returns array[integer])
      for K in Low,High
```

```
        C := K*K
      returns array of C
      end for
end function
```

- **Código *IF1***

```
T 1 1 0           %na=Boolean
T 2 1 1           %na=Character
T 3 1 2           %na=Double
T 4 1 3           %na=Integer
T 5 1 4           %na=Null
T 6 1 5           %na=Real
T 7 1 6           %na=WildBasic
T 8 10
T 9 0 4
T 10 8      4    0
T 11 8      4    10
T 12 8      9    0
T 13 3      11   12
T 14 4      4
C$  F Livermore Frontend   Version1.8
X      13     "main"                                         %sl=3
{ Compound    1    0
G      0                                                     %sl=5
N 1    142                                                   %sl=5
E      0 1    1 1    4     %na=low       %mk=V               %sl=4
E      0 2    1 2    4     %na=high      %mk=V               %sl=4
E      1 1    0 3    14    %na=k         %mk=V
G      0                                                     %sl=5
N 1    152                                                   %sl=5
E      0 3    1 1    4     %na=k         %mk=V               %sl=5
E      0 3    1 2    4     %na=k         %mk=V               %sl=5
E      1 1    0 4    4     %na=c         %mk=V               %sl=5
G      0                                                     %sl=5
N 1    107                                                   %sl=7
L             1 1    4 "1"               %mk=V
E      0 4    1 2    14    %na=c         %mk=V               %sl=6
E      1 1    0 1    9                   %mk=V
} 1 0 3 0 1 2                                                %sl=5
E      0 1    1 1    4     %na=low       %mk=V               %sl=5
E      0 2    1 2    4     %na=high      %mk=V               %sl=5
N 2    115                                                   %sl=7
E      1 1    2 1    9                   %mk=V
E      0 1    2 2    4     %na=low       %mk=V               %sl=4
E      2 1    0 1    9                   %mk=V
```

- **Representação do grafo *IF1***

A figura abaixo ilustra o nó composto *forall* onde os valores disponíveis, nas fronteiras, para os subgrafos são representados por arcos (linha de arco mais fina) ou como quadrados preenchidos de preto mais o respectivo número da entrada a que está associado. A parte do subgrafo correspondente ao gerador apresenta as

portas de entrada 1 e 2 (externas); o corpo apresenta as portas de entrada 1 e 2 (externas) e 3 (saída do gerador); e o retorno apresenta as portas de entrada 1 e 2 (externas), 3 (saída do gerador), 4 (corpo) e 5 (valor de retorno do nó composto).

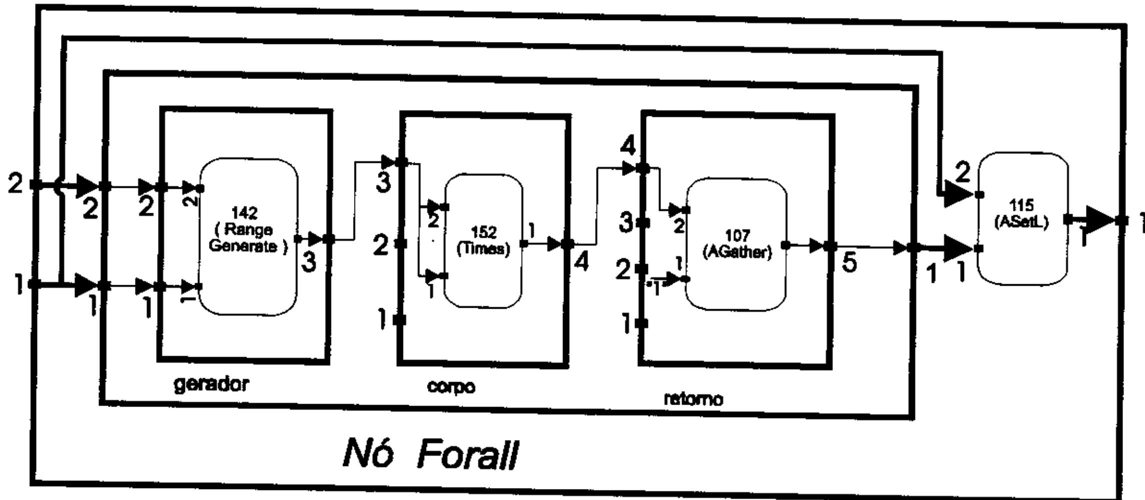


Figura 3 -Representação do programa *IF1* do *Exemplo* 30 para um nó composto *forall*

3.9. Conclusão

Apresentamos neste capítulo a linguagem baseada em grafos acíclicos *IF1*, primeira tradução realizada por um compilador *SISAL* e como no capítulo anterior expusemos apenas as estruturas utilizadas em nosso trabalho.

IF1 foi escolhido como o código a ser manipulado em razão das propriedades decorrentes da representação em forma de grafos. Entre elas podemos citar o número reduzido de estruturas existentes para a representação de um programa *IF1*, resultando em uma semântica bem definida com um número de regras de produção reduzido.

Além disso, *IF1* é uma representação direta de *SISAL*, então manipular o código *IF1* é o mesmo que estar manipulando o código *SISAL*.

Capítulo 4

4. Padrão *MPI* (*Message Passing Interface*)

A passagem de mensagem é um paradigma de programação largamente usado em computadores paralelos especialmente em Computadores Massivamente Paralelos com memória distribuída e em Redes de *Workstations*, Snir et al (1995).

O problema com os sistemas de programação paralela disponíveis alguns anos atrás era a falta de compatibilidade entre eles, já que cada fabricante desenvolvia o seu próprio sistema. Mas, pesquisas mostraram que sistemas de passagem de mensagem poderiam ser implementados eficientemente e de uma forma portável, Snir et al (1995).

Surgiu então a proposta da *Interface de Passagem de Mensagem* ou *MPI*, uma biblioteca de macros e funções que podem ser usadas em programas (*C*, *FORTRAN* e *C++*) para explorar a existência de múltiplos processadores através de passagem de mensagem, Pacheco (1995).

O *MPI* incorporou aspectos de variados sistemas ao invés de adotar como padrão um sistema específico. A definição do *MPI* foi fortemente influenciada pelos trabalhos do *Centro de Pesquisa T. J. Watson da IBM*, *NX/2 da Intel*, *Express*, *Vertex*, *p4* e *PARMACS*. Outras importantes contribuições vieram dos sistemas *ZipCode*, *Chimp*, *PVM*, *Chameleon* e *PICL*, Snir et al (1995).

4.1. Visão Geral

O *MPI* é usado para especificar a comunicação entre um conjunto de processos formando um programa concorrente. O paradigma de passagem de mensagem é atrativo devido à fácil portabilidade de programas usando esse paradigma. Ele é facilmente compatível tanto com multicomputadores de memória distribuída como com multiprocessadores de memória compartilhada sendo que a passagem de mensagem não se torna obsoleta com o aumento na rede ou em

arquiteturas que contenham componentes de memória compartilhada e distribuída, Message Passing Forum (1994a).

Mais do que padronizar as práticas comuns dos sistemas de passagem de mensagem, o *MPI* define características avançadas como tipos de dados definidos pelo usuário, portas de comunicação persistentes, operações de comunicação coletivas poderosas e mecanismos de definição de escopo de comunicação. Nenhum outro sistema anterior havia incorporado todos esses aspectos, Snir et al (1995).

4.2. Especificações do *MPI*

O padrão *MPI*, Snir et al (1995), Message Passing Forum (1994a), Message Passing Forum (1994b), Dongarra et al (1995), Skejellum et al (1993), inclui:

- Comunicação ponto a ponto: mensagens entre pares de processos;
- Comunicação coletiva: operações de comunicação ou sincronização que envolvem grupos inteiros de processos;
- Grupos de processos: especificam conjuntos de processos definidos pelo usuário;
- Comunicadores: um mecanismo para fornecer escopos de comunicação separados para módulos e bibliotecas. Cada comunicador especifica um espaço de nome distinto para os processos, um contexto de comunicação distinto para as mensagens e pode transportar informação adicional específica do escopo;
- Topologia de processos: funções que permitem a manipulação conveniente dos rótulos dos processos, quando estes estão formando uma topologia em particular, tal como uma *grade cartesiana*;
- *Bindings* para *FORTRAN 77* e *C* padrão: *MPI* foi projetado para que versões dele em *C* e *FORTRAN* tenham sintaxe direta. De fato, as formas detalhadas da interface nessas duas linguagens são especificadas e são parte do padrão;
- Interface de *profiling*: a interface é projetada de modo que ferramentas de *profiling* em tempo de execução e monitoramento de performance possam ser acopladas ao sistema. Não é necessário estar de posse do código fonte do *MPI* para realizar o acoplamento, e portanto sistemas de *profiling* portáveis podem ser facilmente construídos;

- Funções de gerenciamento e inquisição de ambiente: essas funções fornecem um temporizador portável, alguma capacidade de *system-querying* e a habilidade para influenciar ações de erros e funções para manipulação de erros.

Existem importantes aspectos de programação paralela que não são cobertos pelo padrão. Abaixo uma lista deles Snir et al (1995), Message Passing Forum (1994b), Dongarra et al (1995):

- Operações com memória compartilhada;
- Operações que requerem mais suporte do sistema do que o existente no padrão corrente, por exemplo recepções dirigidas por interrupção, execução remota e mensagens ativas;
- Ferramentas para a construção de programas;
- Suporte para depuração;
- Suporte explícito para *threads*;
- Gerenciamento de processos ou tarefas;
- Funções de entrada e saída.

A principal razão pela qual estas questões não foram tratadas foi o tempo auto-imposto pelo comitê e a percepção de que muitas delas são dependentes do sistema.

4.3. O Padrão *MPI*

É importante definirmos os conceitos básicos dos termos usados no *MPI*. Isto deve-se ao fato de que muitos dos termos utilizados em se tratando de passagem de mensagem apresentam uma utilização diferente no *MPI*.

Os conceitos elementares do padrão *MPI* são processos e mensagens. A comunicação ponto a ponto e a comunicação coletiva são os conceitos centrais do *MPI* enquanto que grupos de processos, contexto de comunicação, tipos de dados e topologias virtuais são outros conceitos importantes embutidos dentro do padrão *MPI*, Malard (1995)

Ao tratarmos de rotinas de comunicação ponto a ponto e coletivas é estritamente necessário o conhecimento dos conceitos de grupos de processos e de contexto de comunicação. Por esta razão, definiremos primeiro estes conceitos para só então apresentarmos as rotinas de comunicação do padrão *MPI*.

4.3.1. Mensagens e processos

Um processo *MPI*, Malard (1995), Message Passing Forum (1994a), Message Passing Forum (1994b), Snir at al (1995), é uma entidade indefinida que toma parte na execução de alguma tarefa computacional. No modelo de programação *MPI*, cada processo está associado a uma única memória, chamada memória local, a qual apenas ele pode acessar e atualizar.

A mensagem consiste de um pedaço de informação, chamado corpo, junto com alguns dados adicionais chamados envelope.

O corpo da mensagem consiste da informação a ser transmitida, tamanho da informação e o tipo da informação.

O envelope da mensagem, Message Passing Forum (1994b), contém informações adicionais que podem ser usadas pelo processo destinatário para decidir se e quando abrir a mensagem assim como o modo de acessar os dados no corpo da mensagem, uma vez que ela tenha sido recebida. Os campos são:

fonte *destino* *tag* *comunicador*

A *fonte* da mensagem é implicitamente determinada pela identificação do emissor da mensagem, o *destino* da mensagem é especificado pela identificação do processo destino e o *tag* é um valor inteiro utilizado para distinguir os diferentes tipos de mensagens. O campo *comunicador* será explicado mais abaixo.

A troca de mensagens entre processos é referida como comunicação e é o único modo pelo qual os processos *MPI* podem acessar dados das memórias dos outros processos.

4.3.1.1. Contexto de comunicação

O contexto de comunicação, Message Passing Forum (1994a), Message Passing Forum (1994b), Snir et al (1995), é uma propriedade que permite que o espaço de comunicação possa ser compartilhado por vários fluxos de mensagens, pois uma mensagem enviada em um contexto não pode ser recebida em outro.

Com o suporte de contexto de mensagem, a interface de passagem de mensagem possibilita que mensagens de usuário e bibliotecas paralelas possam trafegar sem conflitos através do mesmo espaço de comunicação.

Ao usuário não é permitida a execução de operações explícitas sobre o contexto, isto é, não existe um tipo de dados contexto que seja visível a nível de usuário. Portanto, o contexto não pode ser explicitamente usado em qualquer função *MPI*. Por outro lado, eles são implicitamente associados aos grupos quando os comunicadores são criados.

4.3.1.2. Grupos de processos

Um grupo de processos, Message Passing Forum (1994a), Message Passing Forum (1994b), Snir et al (1995), é uma coleção ordenada de processos onde cada processo é identificado unicamente por um número, cuja numeração inicia-se por 0 (zero). O número associado com o processo em um grupo é definido como o *rank* deste no grupo. Os grupos são formados baseados nas tarefas a serem realizadas, isto é, um grupo de processos *MPI* tipicamente reune processos que estejam trabalhando em uma tarefa comum.

Os grupos de processos do *MPI* são ditos estáticos porque todos eles derivam de um grupo inicial e nenhum novo processo pode ser criado durante a execução do programa. À parte esta limitação, o *MPI* fornece uma extensa quantidade de facilidades para manipulação de processos. Podemos utilizar grupos de processos para especificar quais processos estão envolvidos em uma operação coletiva de comunicação, tal como um *broadcast* ou eles podem ser usados para introduzir paralelismo a nível de tarefas em uma aplicação, de modo que diferentes grupos executem diferentes tarefas.

4.3.1.3. Topologias Virtuais

No *MPI*, uma *topologia*, Message Passing Forum (1994a), Message Passing Forum (1994b), Snir et al (1995), é um mecanismo para associação de diferentes esquemas de endereçamento com processos pertencendo a um grupo. Observe que as topologias do *MPI* são *topologias virtuais* isto é, pode não existir nenhuma relação entre a estrutura dos processos definidos pela *topologia virtual* e a estrutura física real da máquina paralela.

O *MPI* suporta essencialmente dois tipos de estruturas de topologias: a *topologia de grafo* e a *topologia cartesiana* ou *grade.*

A especificação de topologia virtual por meio de grafo cobre qualquer tipo de aplicação, mas muitas vezes não é a melhor forma de expressar uma aplicação e em razão disto existe o suporte para a outra forma de topologia, Message Passing Forum (1994b),.

4.3.1.4. Comunicadores

Os *comunicadores*, Message Passing Forum (1994b), Snir et al (1995), são objetos abstratos utilizados para definir o escopo de uma operação de comunicação. Este objeto abstrato está sempre associado a um grupo de processos, a um contexto de comunicação e a uma topologia virtual. Todos os processos que estiverem envolvidos em uma comunicação devem fornecer um mesmo *comunicador* como um argumento nas chamadas de procedimento correspondente.

O *MPI* fornece a função `MPI_Comm_rank`, a qual retorna o *rank* de um processo em seu segundo argumento. Sua sintaxe é:

```
int MPI_Comm_rank(MPI_Comm comm,/* comunicador              */
                  int *rank     /* rank do processo em comm */
                 )
```

O primeiro argumento é um *comunicador*. Essencialmente um *comunicador* é uma coleção de processos que podem enviar mensagens um para o outro. Para programas básicos o único *comunicador* necessário é `MPI_COMM_WORLD`. Ele está definido no *MPI* e consiste de todos os processos executando quando o programa inicia a execução.

Os programas também podem depender do número de processos executando. Assim o *MPI* fornece a função MPI_Comm_size para a determinação deste valor. O primeiro argumento é um *comunicador*. Ela retorna o número de processos de um *comunicador* em seu segundo argumento. Sua sintaxe é:

```
int MPI_Comm_size(MPI_Comm comm, /* comunicador              */
                  int *size /* numero de processos em comm */
                 )
```

Um *comunicador* utilizado em uma comunicação dentro de um grupo é referido como um *intra-comunicador* estando este ligado a um contexto e a um grupo e um *comunicador* usado em uma comunicação entre os grupos é referido como um *inter-comunicador* estando ligado a um contexto e dois grupos.

4.3.2. Tipos de dados do *MPI*

O sistema de tipos das mensagens, Message Passing Forum (1994b), Snir et al (1995), do *MPI* foi definido sobre o sistema de tipos das linguagens *C* e *FORTRAN*. Os tipos de dados mais simples são basicamente a união dos tipos de dados atômicos de *C* e *FORTRAN* precedidos do prefixo MPI_.

O *MPI* fornece também procedimentos para definir tipos de dados derivados, cujo formato consiste de uma seqüência de valores onde é conhecido o número de elementos na seqüência, o tipo de cada elemento e a distância em *bytes* entre os elementos da seqüência. Estes tipos de dados derivados são criados em tempo de execução.

Tabela VIII - Alguns tipos *MPI* e os equivalentes em *C*

MPI datatype	*C datatype*
MPI_CHAR	*signed char*
MPI_SHORT	*signed short int*
MPI_INT	*signed int*
MPI_FLOAT	*float*
MPI_DOUBLE	*double*

4.4. Comunicação

4.4.1. Comunicação ponto a ponto

O padrão *MPI* fornece um conjunto de rotinas *send* e *receive* para comunicação ponto a ponto de dados associados a tipos, Message Passing Forum (1994a), Message Passing Forum (1994b), Snir et al (1995), Malard (1995). A atribuição de tipo à mensagem enviada é necessária para que conversões corretas das mensagens possam ser realizadas quando o dado é enviado de uma arquitetura para outra.

O *MPI* suporta ainda seletividade explicita das mensagens, Message Passing Forum (1994b), através das informações disponíveis no envelope da mensagem. Uma mensagem pode ser recebida por uma operação de *receive* se os campos *fonte*, *tag* e *comunicador* do envelope da mensagem corresponderem aos valores especificados na operação de recepção.

A comunicação ponto a ponto envolve apenas dois processos: um processo enviando uma mensagem e um processo recebendo esta mensagem. Todos os procedimentos de comunicação ponto a ponto do *MPI* têm uma versão bloqueante e uma versão não bloqueante. O modo de uma operação de comunicação ponto a ponto determina o momento em que uma operação *send* é iniciada ou quando ela é completada. Os modos de operação são: *padrão*, *ready*, *síncrono* e *buffered*.

Um *send* no modo *padrão* pode ser iniciado mesmo que um *receive* equivalente não tenha sido iniciado. Neste modo o *MPI* é que decide se a mensagem de saída será colocada em um *buffer*. Se o *MPI* colocar em um *buffer* a mensagem de saída, a chamada do *send* pode completar antes que um *receive* equivalente seja invocado, caso contrário a chamada do *send* não completará até que um *receive* equivalente seja postado e o dado tenha sido movido para o processo receptor.

Um *send* no modo *buffered* é similar a um *send* no modo *padrão*, mas a sua conclusão é sempre independente do *receive* correspondente, sendo necessário colocar em *buffers* a mensagem, para assegurar esta independência.

O modo *síncrono* e o modo *padrão* de um *send* são em sua maior parte iguais, exceto que no modo *síncrono* o *send* não completará até que a recepção da

mensagem seja garantida. Isto significa que a conclusão de um *send síncrono* não indica apenas que o *buffer* utilizado pelo *send* pode ser reutilizado mas também que o receptor já alcançou o ponto de execução do *receive* correspondente.

O *send* no modo *ready* pode ser iniciado apenas se um *receive* equivalente tiver sido iniciado. Uma operação *send* no modo *ready* possui a mesma semântica que as operações *send* no modo *padrão* e no modo *síncrono*, apenas que no modo *ready*, o emissor fornece informação adicional para o sistema (um *receive* equivalente já postado) permitindo diminuir alguma sobrecarga. Portanto, um *send* no modo *ready*, dentro de um programa correto, poderia ser trocado por um *send* no modo *padrão* com nenhuma alteração no resultado do programa a não ser pelo desempenho.

As rotinas *send* na sua versão bloqueante não retornam até que as locações dos dados especificados no *send* possam ser seguramente reutilizadas, sem corromper a mensagem. Já um *send* não bloqueante não espera que nenhum evento particular ocorra para retornar. Por esta razão retorna um *handle* para um objeto de comunicação que pode ser usado subseqüentemente por rotinas que verificam o termino da operação *send*.

Para a recepção das mensagem existem dois tipos de *receive*: um bloqueante e um não bloqueante. Qualquer um dos dois tipos de *receive* pode ser usado para corresponder aos vários tipos de *send*.

Um *receive* bloqueante não retornará até que a mensagem tenha sido armazenada nas posições indicadas pelo *receive*. Um *receive* não bloqueante retorna um *handle* para um objeto de comunicação e não espera que qualquer evento particular ocorra. O *handle* pode ser usado para verificar o *status* da operação *receive* ou para bloquear até que ela complete.

Exemplo 24) A sintaxe de um procedimento *send* padrão do *MPI* é exibida abaixo.

```
int MPI_Send(void *buf,       /* endereço do buffer do send     */
             int count,       /* número de entradas do send     */
             MPI_Datatype datatype,    /* tipo da entrada       */
             int destin,               /* rank do destino       */
             int tag,                  /* tag da mensagem       */
             MPI_Comm comm             /* comunicador           */
            )
```

O corpo da mensagem consiste de uma seqüência de valores, cuja disposição na memória principal é representada na lista de argumentos dos procedimentos *MPI* por um trio consistindo do endereço do primeiro elemento na seqüência, o comprimento da seqüência e o tipo de dado *MPI* de todos os elementos na seqüência.

O trio corresponde aos parâmetros: `buf`, `count` e `datatype`, nesta ordem. Os outros argumentos pertencem ao envelope da mensagem e especificam o *rank* do processo destino e o *comunicador*. O *tag* pode ser fixado pelo processo emissor para indicar o conteúdo da mensagem. Ambos processos *send* e *receive* devem pertencer ao grupo associado ao *comunicador* `comm`.

Exemplo 25) A sintaxe do procedimento *receive* padrão do *MPI* é apresentada abaixo.

```
int MPI_Recv(void *buf, /* endereço inicial do buffer de recepção */
             int count, /* número máximo de entradas para receive */
             MPI_Datatype datatype,/* tipo de cada entrada         */
             int source,          /* rank do emissor da mensagem */
             int tag,             /* tag da mensagem             */
             MPI_Comm comm,       /* comunicador                 */
             MPI_Status *status   /* status de retorno           */
            )
```

A lista de parâmetros é similar àquela anterior para *send*. O endereço inicial do corpo da mensagem e como ela vai ser colocada na memória é especificada no trio `buf`, `count` e `datatype`. O parâmetro `buf`, é o endereço inicial da mensagem recebida; `count` é o comprimento máximo da mensagem; `datatype` o tipo dos dados da mensagem; os parâmetros `source` e `tag` são chamados de selecionadores da mensagem; `comm` o comunicador; e `status` retorna informações sobre o `source` e o `tag` da mensagem recebida.

4.4.2. Comunicação coletiva

As rotinas de comunicação coletiva fornecem meios para a realização de comunicação coordenada entre processos no grupo especificado por um intra-comunicador. O padrão *MPI* define os seguintes tipos de comunicação coletiva:

- Sincronização *barrier* entre todos os membros do grupo;

- Funções de comunicação global: *broadcast, scatter* e *gather* e
- Operações de redução global tais como *sum, max, min* ou funções definidas pelo usuário.

A sintaxe e a semântica das funções coletivas do *MPI* são consistentes com as funções de comunicação ponto a ponto, apenas que foram impostas restrições quanto a sua variedade. As restrições são:

- Quantidade de dado a ser enviada deve ser exatamente igual à quantidade de dados a ser recebida especificada no *receive*;
- Funções coletivas não utilizam *tag*, assim dentro de cada domínio de comunicação num grupo, as chamadas coletivas devem ter a ordem de chamada rigorosamente corretas e
- Funções coletivas apresentam-se apenas de um modo que pode ser considerado como equivalente ao modo padrão da comunicação ponto a ponto.

Uma função coletiva pode retornar tão logo sua participação na comunicação global tenha se completado, o que significa que o processo que invocou a função, agora está livre para acessar e modificar as posições no *buffer* de comunicação.

Em uma operação de comunicação coletiva os processos envolvidos devem chamar a função de comunicação coletiva desejada com os mesmos argumentos.

As chamadas para comunicação coletiva e as chamadas para comunicação ponto a ponto podem usar o mesmo comunicador pois o *MPI* garante a separação das mensagens.

4.4.2.1. *Broadcast*

Um *broadcast* é uma operação de comunicação coletiva no qual um único processo, chamado *root*, envia o mesmo dado para todos os processos associados a um *comunicador*. Todos os processos envolvidos no *broadcast* devem ter argumentos idênticos àqueles do processo root. No *MPI* a função para realizar o *broadcast* de dados é `MPI_Bcast`.

```
int MPI_Bcast(void *buffer,           /* endereço inicial do buffer   */
              int count,              /* número de entradas no buffer */
              MPI_Datatype datatype,  /* tipo do dado do buffer       */
              int root,               /* rank do root do broadcast    */
              MPI_Comm comm           /* comunicador                  */
             )
```

Uma operação *broadcast* envia uma cópia do dado no parâmetro `buffer` do processo `root` para todos os processos associados ao comunicador `comm`. Os parâmetros `count` e `datatype` têm a função de especificar o tamanho da mensagem.

Exemplo 26) Exemplo em *C* de um *broadcast* de 100 inteiros do processo 0 para todos os processo no grupo.

```
MPI_Comm comm;
int array[100];
int root = 0;
...
MPI_Bcast(array, 100, MPI_INT, root, comm)
```

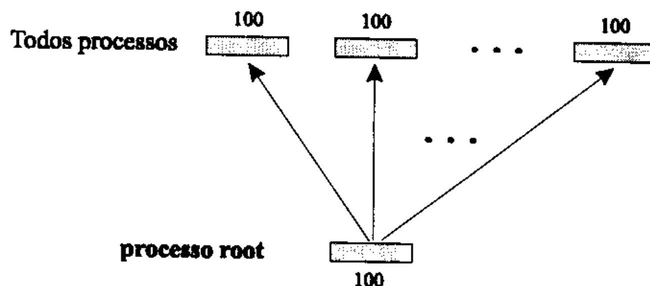


Figura 4- Ilustração do *Exemplo 26*

4.4.2.2. *Gather*

Em uma operação *gather*, todos os processos envolvidos na comunicação enviam o conteúdo de seu *buffer* de envio para o processo *root*, inclusive o próprio *root*. O processo *root*, que recebe os dados, armazena-os por ordem de *rank*. O *buffer* de recepção é significativo apenas para o *root*, sendo ignorado por todos os outros processos. A especificação é:

```
int MPI_Gather(void *sendbuf,     /* endereço do buffer de envio    */
               int sendcount,     /*número de elementos no sendbuf  */
               MPI_Datatype sendtype,/* tipo do dado no sendbuf     */
               void *recvbuf,     /* endereço inicial do recvbuf    */
               int recvcount,     /* número de elementos no recvbuf */
               MPI_Datatype recvtype, /* tipo do dado no recvbuf    */
               int root,              /* rank do root do gather     */
               MPI_Comm comm          /* comunicador                */
               )
```

Uma operação *gather* pode ser semanticamente interpretada como *n* chamadas da função `MPI_Send`, uma para cada um dos n processos, enviando dados para o processo *root* que executa n processos `MPI_Recv`, concatenando as mensagens enviadas por ordem de *rank*.

Lembrando ainda que todos os argumentos são significativos para o *root* e para os processos restantes os parâmetros relacionados com o *buffer* de recepção `recvbuf` são ignorados. O argumento `root` deve ter o mesmo valor em todos os processos e `comm` deve representar o mesmo comunicador intra-grupo.

Exemplo 27) Uma operação *gather* de 100 elementos inteiros de todos os processos no grupo para o root

```
#define ROOT = 0

MPI_Comm comm;
int sendarray[100], gsize, myrank, *rbuf;
...
MPI_Comm_rank(comm, &myrank);
if (myrank == ROOT)
{   MPI_Comm_size(comm, &gsize);
    rbuf = (int *) malloc(gsize*100*sizeof(int));
}
...
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, ROOT,
           comm);
```

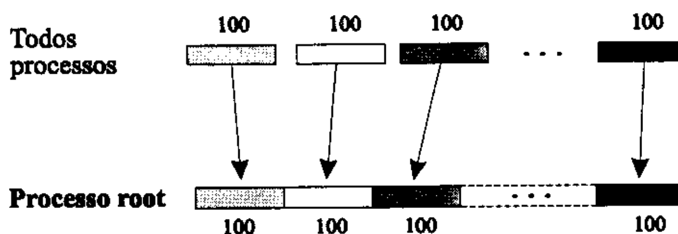


Figura 5 - Ilustração do *Exemplo 27*

4.4.2.3. *Scatter*

A operação *scatter* é o inverso da operação *gather*. A sintaxe é:

```
int MPI_Scatter(void *sendbuf,/* endereço do buffer de envio      */
                int sendcount, /* número de elementos no sendbuf  */
                MPI_Datatype sendtype, /* tipo do dado no sendbuf */
                void *recvbuf, /* endereço inicial do recvbuf     */
                int recvcount, /* número de elementos no recvbuf  */
                MPI_Datatype recvtype, /* tipo do dado no recvbuf */
                int root,      /* rank do root do gather          */
                MPI_Comm comm  /* comunicador                     */
               )
```

Podemos considerar que uma operação *scatter* funciona como se o processo *root* estivesse realizando *n* chamadas `MPI_Send`, uma para cada um dos `n` processos do comunicador, enviando `sendcount` elementos sucessivos de `sendbuf` a todos os *n* processos. Estes realizam a operação `MPI_Recv` para receber `recvcount` elementos em `recvbuf`.

Todos os argumentos são significativos no processo *root*, enquanto que para os outros processos são significativos `recvbuf`, `recvcount`, `recvtype`, `root` e `comm`. O argumento `root` deve ter o mesmo valor para todos os processos e `comm` deve representar o mesmo comunicador intra-grupo.

Exemplo 28) Operação *scatter* de um conjunto de 100 unidades de inteiros de root para todos os processos no grupo

```
#define ROOT = 0

MPI_Comm comm;
int gsize, *sendbuf, myrank, recvbuf[100];

MPI_Comm_rank(comm, &myrank);

if (myrank == 0)
{   MPI_Comm_size(comm, &gsize);
    sendbuf = (int *) malloc(gsize*100*sizeof(int));
}
MPI_Scatter(sendbuf, 100, MPI_INT, recvbuf, 100, MPI_INT,
            ROOT, comm);
```

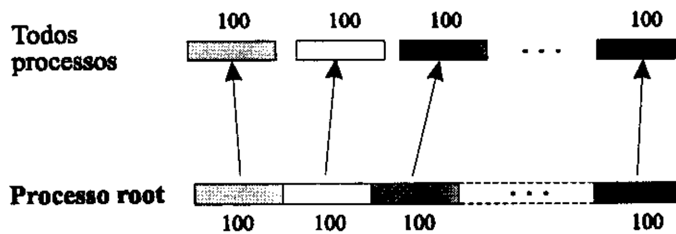


Figura 6 - Ilustração do *Exemplo 28*

4.4.2.4. *Reduce*

Em uma operação de redução global, todos os processos de um dado comunicador contribuem com dados que são combinados usando alguma operação binária. A sintaxe é:

```
int MPI_Reduce(void *sendbuf, /* endereço do buffer de envio      */
               void *recvbuf, /* endereço do buffer de recepcao   */
               int count,     /* número de elementos no sendbuf */
               MPI_Datatype datatype, /* tipo dos elementos do    */
                                      /* buffer de envio          */
               MPI_Op op,             /* operação de redução     */
               int root,              /* rank do processo root */
               MPI_Comm comm          /* comunicador             */
              )
```

A operação `MPI_Reduce` combina os operandos armazenados em `*sendbuf` usando a operação `op` e armazena o resultado no processo `root`. Os argumentos `count`, `op` e `root` devem ter valores idênticos em todos os processos. O argumento `datatype` deve representar o mesmo tipo e `comm` deve representar o mesmo *comunicador*.

Exemplo 29) Rotina que realiza o produto escalar de dois vetores que são distribuídos através de um grupo de processos e retorna a resposta no nó zero.

```
float a[10], /* fatia local do array a */
      b[10], /* fatia local do array b */
      c,     /* possui o resultado no nó zero */
      s = 0.0;
int   i;

/* soma local */
for (i = 0; i < 10; i++)
   s = s + a[i]*b[i];
/* soma global */
MPI_Reduce(s, &c, 1, MPI_REAL, MPI_SUM, 0, MPI_COMM_WORLD);
```

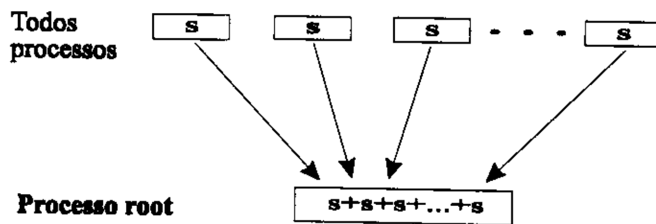


Figura 7 - Ilustração do *Exemplo 29*

4.5. Conclusão

Neste capítulo apresentamos, resumidamente, o padrão *MPI* descrevendo os conceitos elementares: processos e mensagens; os conceitos centrais: comunicação ponto a ponto e a comunicação coletiva e outros importantes conceitos embutidos do padrão: grupos de processos, contexto de comunicação, tipos de dados e topologias virtuais.

Para o nosso trabalho utilizamos principalmente as rotinas de comunicação coletiva (*broadcast, gather, scatter* e *reduce*). Além destas utilizamos as rotinas para inicialização e finalização do ambiente *MPI* é claro e as rotinas para a manipulação do contexto de comunicação. Estas rotinas supriram satisfatoriamente as nossas necessidades na paralelização dos programas *SISAL*.

Capítulo 5

5. Paralelização dos programas

Para obter a paralelização de *SISAL/IF1* no sistema *MPI* as seguintes questões foram abordadas: escolha da estrutura a ser paralelizada, o método de particionamento a utilizar, o modelo de paralelismo das tarefas e em qual fase da compilação realizar as alterações. Isto resolvido, iniciou-se a abordagem de questões de implementação tais como a ligação entre *SISAL/IF1* e *MPI* e a forma para compilar e executar os programas com as alterações.

A maior parte dos exemplos apresentados nas próximas seções não apresentarão o código *IF1* mas o código *SISAL* equivalente por razões de simplicidade e facilidade no entendimento, mas lembrando que as alterações foram feitas de fato no *IF1*.

5.1. Fonte de paralelismo e modelo de particionamento

Segundo Sarkar; Hennessy (1986), três são os problemas fundamentais a serem resolvidos quando compilamos um programa para execução paralela em um multiprocessador:

- Identificação do paralelismo potencial;
- Particionamento do programa em tarefas seqüenciais e
- Escalonamento das tarefas na execução concorrente.

e segundo Grit (1990), são críticos para a performance de sistemas de passagem de mensagem:

- Proporção entre a velocidade de comunicação e a capacidade de computação;
- Custo da sobrecarga na criação de tarefas.

Em *SISAL* temos como fontes de paralelismo os *loops forall,* os *loops* iterativos, as funções, as expressões múltiplas e as *streams*, Skedzielewski (1991).

Escolhemos a expressão *forall* como nossa fonte de paralelismo. O que motivou a escolha foi o fato de *forall* apresentar a forma mais explícita de paralelismo, onde cada uma das instâncias do corpo podem ser executadas em paralelo e sendo o número de iterações conhecido no momento da entrada na estrutura.

O método de particionamento escolhido foi o *slice*, Sarkar; Cann (1990), Cann; Evripidou (1995), dos *loops forall* apenas no seu nível mais externo. O método *slice* consiste na divisão de todas as instâncias do *forall* em grupos de instâncias consecutivas, de modo a formar *foralls* de menor número de iterações.

Lembrando que, cada um dos novos *forall* pode ser executado em um processo diferente, em paralelo, onde cada um dos *forall* calcula parte do resultado total do *forall* original, sendo necessário então, após cada *forall* realizar o seu trabalho, agrupar os resultados de modo a formar um único resultado.

Após ser escolhido o método, foram iniciados os levantamentos dos aspectos necessários para realizar o particionamento de um *loop forall*, sem a preocupação ainda com *MPI*.

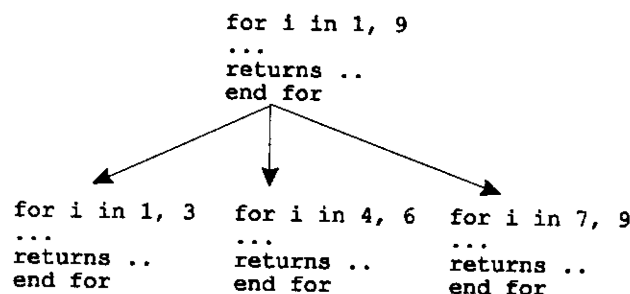


Figura 8 - *Slice* de um *forall* com 9 instâncias sendo dividido em 3 *forall* de 3 instâncias.

5.2. Código fonte e fase do particionamento

Esta foi uma fase importante no desenvolvimento do projeto pois além da escolha do passo da compilação onde seriam realizada as alterações necessárias decidiu-se pela manipulação do código *IF1* ao invés de *SISAL* em virtude de todas as propriedades de *IF1*, citadas no Capítulo 3. Podemos verificar entre outras coisas a simplicidade do código *IF1*, em função do reduzido número de regras de produções

definindo sua sintaxe, o que o torna de fácil manipulação. Outro aspecto importante para a escolha de *IF1* foi a relativa independência com uma linguagem específica.

Para gerar arquivos *IF1* a partir de fontes *SISAL*, utilizamos o compilador *OSC*, Cann (1992a), que apresenta o seguinte processo de compilação, Cann; Evripidou (1995), Cann (1992a):

1. Tradução de *front-end* de *SISAL* para *IF1*, Cann (1992a);
2. Otimizações convencionais independentes de máquina, Skedzielewski; Welcome (1985), Cann; Evripidou (1995), Gaudiot; Lee (1987);
3. Construção de um programa *IF1* monolítico (OSC92a, PAM94]
4. Análises *build-in-place* e tradução para *IF2*, Cann (1992a), Welcome et al (1986), Feo; Cann (1990), Cann ; Evripidou (1995), Gaudiot; Lee (1987);
5. Análises *update-in-place*, Cann (1992a), Welcome et al (1986), Feo; Cann (1990), Cann; Evripidou (1995);
6. Paralelização e vetorização, Cann (1992a), Cann; Evripidou (1995);
7. Geração de código *C*, Cann (1992a);
8. Compilação do código *C*, Cann (1992a).

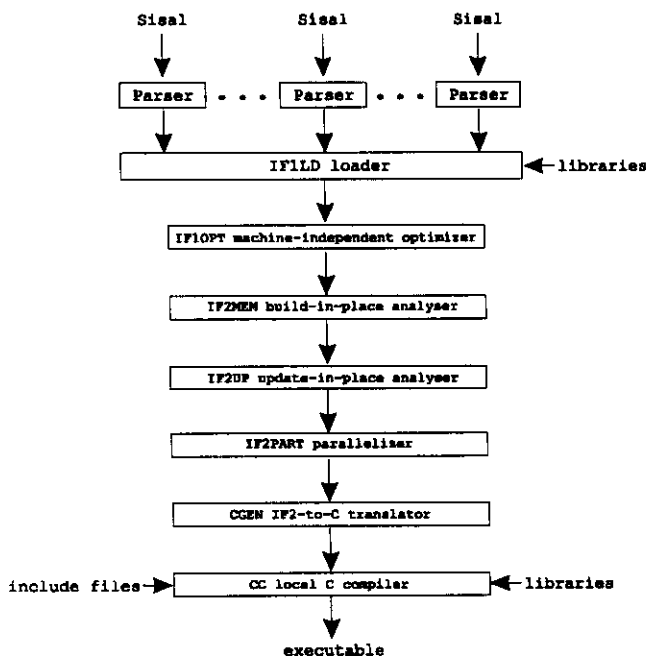


Figura 9 - Ilustração das fases do compilador *OSC*

O código utilizado para a realização do particionamento seria então obtido após o *front-end* do OSC, na Figura 9 correspondente ao *parser*. O programa particionado nos *loops forall* em *IF1* iria ainda se beneficiar de todas otimizações que o compilador realizasse, já que *IF1* é o primeiro produto da compilação de um programa *SISAL*.

Nesta fase escolheu-se o modelo de implementação do paralelismo, *SPMD* (*Single Program Multiple Data*), Almasi; Gottlieb (1994), Pacheco (1995), um caso especial de *MIMD*, no qual todos os processadores executam o mesmo programa, se bem que em qualquer momento a instrução, e é claro os dados, podem ser diferentes para cada processador em virtude de desvios dependentes dos dados. Este estilo de programação traz uma série de benefícios, Foisy; Chailloux (1995):

- ele oferece um modo de gerenciar a complexidade de *software* paralelo pois existe apenas um código a ser depurado, existe um número limitado de modos que uma dada cópia pode interagir com alguma outra e um programa paralelo geralmente é obtido de um programa seqüencial;
- a maioria dos programas adaptam-se a diferentes números de processadores;
- para computadores massivamente paralelos, ele parece o único meio razoável para obter programas eficientes;
- ele permite escrever mais facilmente programas com resultados determinísticos.

Em *SPMD*, o número de processos ou tarefas é decidido antes do início da execução, assim evitando a sobrecarga do sistema operacional associado com o estilo *fork-join* de criação de processos durante a execução. O *SPMD* pode ser usado tanto em sistemas de memória compartilhada como em sistemas de memória distribuída (passagem de mensagem)..

Exemplo 30) Trecho de um programa *SPMD* em *C*

```
if (myrank == 0) /* root */
{    x = sin(x)*lambda * fatorial(30);   ...
}
else /* escravo */
{ x = cos(x)*lambda   *   fatorial(30);    ...
}
```

No programa acima podemos ver a expressão `if` que para o caso `myrank=0` executaria o processo *root* caso contrário um outro processo com qualquer outra identificação.

5.3. *SISAL/IF1* mesclado com *C*

Em virtude da heterogeneidade entre *SISAL* e *MPI*, surgiu o problema de como realizar a ligação entre programas *SISAL/IF1* e a biblioteca *MPI* escrita em *C*. Devemos no entanto citar um fator positivo: o suporte da interface para *linguagens estrangeiras*, oferecida por *SISAL* e presente no compilador do *Lawrence Livermore National Laboratories*, *OSC*, o que facilitaria a realização da ligação.

O suporte fornecido pelo compilador OSC provê interfaces através das quais programas *SISAL* podem realizar chamadas para rotinas em *C* ou *FORTRAN* ou programas *FORTRAN* e *C* podem realizar chamadas para rotinas em *SISAL*. Deve-se no entanto observar que existem restrições quanto aos tipos dos parâmetros de entrada e retorno das funções na interface, não sendo inclusos os tipos *stream*, *record*, *union*, *null* e *boolean*

Em função desta restrição, definimos que neste trabalho apenas os tipos permitidos no interfaceamento de *OSC* com *C* seriam usados: *integer*, *real*, *double_real*, *character* e *arrays* unidimensionais destes tipos.

A primeira providência a ser tomada quando desejamos utilizar uma função *C*, em *SISAL/IF1*, é inserir um *pragma* de interfaceamento, Cann (1992a), com o nome da função desejada e a declaração de importação da função com os parâmetros de entrada e retorno desejados, mantendo a estrita correspondência de tipo e ordem.

Exemplo 31) Abaixo um programa importando uma função *C* que recebe dois argumentos inteiros como entrada e retorna um inteiro e em seguida a função *C*.

```
%Programa SISAL
%$c=soma

define main

global soma(a, b:integer returns integer)

function main(returns integer)
```

```
    soma(10, 3)
end function
%fim do código SISAL

/*  A partir daqui codigo C com a função importada */
int soma(x, y)
int x, y;
{
    returns (x + y);
}
```

No código acima identificamos `%$c=` como o *pragma* declarando que a linguagem de importação é *C* e `soma` é o nome da função importada. A declaração efetiva da função importada é dada pela linha `global soma(a, b:integer returns integer)`. Observe que a linha de declaração da importação de uma função é idêntica a uma linha de importação de uma função *SISAL*, permitindo então que a manipulação da função importada seja idêntica à de uma função que tivesse sido escrita em código *SISAL*.

Uma função retornando objetos simples possui uma semântica de ligação bastante simples e direta, como pudemos observar. No entanto esta simplicidade não está presente quando o retorno de uma função envolve um tipo *array*. Nesta situação, a interface exige que para cada *array* retornado seja associado um descritor, o qual fornecerá ao compilador as informações necessárias para o retorno correto do *array*. Este descritor é necessário em função de *SISAL* permitir que um *array* assuma qualquer índice inicial e, se multidimensional, as dimensão poderem diferir no comprimento.

Um descritor de *array*, Cann (1992a), possui as seguintes informações:

- O primeiro componente define como é a disposição do *array*, 0 em coluna e 1 por linha;
- O segundo componente define como os dados se movimentam através da interface, 0 transposição e 1 inalterado;
- O terceiro componente especifica a mutabilidade dos dados transmitidos, 0 para imutável e 1 para mutável. Se o descritor está associado a um resultado este componente é ignorado;
- Os componentes restantes aparecem em grupos de cinco, representando cada dimensão existente. O primeiro e o segundo elemento, de cada grupo de cinco,

definem o limite inferior real e o limite superior real, o terceiro e quarto elementos definem os limites inferior e superior lógicos desejados e o último elemento define o índice do primeiro valor a ser retornado para os descritores associados a resultados.

Exemplo 32) Rotina em *SISAL* chamando uma rotina *C* que toma um *array* e um escalar como entrada e retorna um *array* como saída

```
%inicio do codigo SISAL
%$c=biggest
define main

type OneI = array[integer];

global biggest(vector:OneI; size:integer; vectdescr:oneI
               returns OneI)

function main(vector :OneI; size :integer returns OneI)
   let
       vectdescr := array[0:1,1,0,0,size-1,0,size-1,0]
   in
       biggest(vector, size, vectdescr)
   end let
end function
%fim do código SISAL

/* inicio de código C com a função biggest */
void big(vector, size, vectdescr, retvector)
int *vector,
    size,
    *vectdescr,
    *retvector;
{
 ...
}
```

No exemplo acima, a função *C* possui como retorno um vetor de inteiros, `vector`, e em razão disto a necessidade da declaração do descritor de vetor `vectdescr`. O primeiro elemento do descritor contém o valor 1, pois em *C* *arrays* são manipulados por linha, o segundo com o valor 1 indica que não deve ocorrer transposição do *array*, o terceiro elemento pode assumir qualquer valor inteiro pois o descritor representa um retorno, logo este campo será ignorado. Os elementos seguintes definem os limitantes reais e lógicos do *array* e o último o índice para primeiro elemento do *array* de dimensão única.

Lembrando novamente que, com a intenção de simplificar o entendimento, não exibimos o exemplo acima no código *IF1* mas no código *SISAL*.

5.4. Interface entre *SISAL/IF1* e *MPI*

Utilizando o conhecimento de interfaceamento entre *SISAL/IF1* e funções *C* foi possível a elaboração de uma biblioteca em *C*, utilizada para chamadas de rotinas da biblioteca *MPI*, por programas *SISAL/IF1*.

A primeira decisão tomada, para a elaboração da biblioteca, foi a forma como seria identificada uma função de interfaceamento entre *SISAL/IF1* e *MPI*. Definiu-se então que esta função possuiria o prefixo `sisal_mpi_` seguido do nome da função, em minúsculo, devendo ser igual ou pelo menos o mais próximo do nome original da função *MPI* que ela estiver representando.

Definiu-se a seguir que o único *comunicador* utilizado nas funções seria `MPI_COMM_WORLD`, pois ele engloba todos os processos existentes no momento da execução.

Durante a elaboração da biblioteca de interface foi resolvida a questão da reordenação de nós do grafo *IF1*, Skedzielewski; Glauert (1985), Pande et al (1994), realizadas pelo compilador *SISAL*. A chamada de uma função está associada a um nó *IF1*, correspondente a chamada de função. A preocupação com reordenação dos nós de chamadas de funções *MPI* está no fato de que ela poderia gerar resultados inconsistentes ou mesmo *deadlock*, pois a ordem das funções de comunicação coletiva devem corresponder, Snir et al (1995), em todos os processos. A reordenação de nós pelo compilador *OSC* é realizado somente entre aqueles nós que apresentam independência de dados e controle. A reordenação seria então evitada com a inserção de um parâmetro extra em cada uma das rotinas de ligação, entre *SISAL/IF1* e *MPI*, parâmetro este que seria usado para representar um arco de dependência entre os nós das chamadas das funções, evitando assim a reordenação.

Exemplo 33) Abaixo a interface para `MPI_Comm_rank` e `MPI_Comm_Size`.

```
int sisal_mpi_comm_size(in_depend, size, out_depend)
int in_depend, /* entrada para a dependencia entre funcoes */
    *size,       /* retorno do numero de processos          */
```

```
    *out_depend; /* saida para a dependencia entre funcoes */
{  /* chamada função MPI que retorna o numero de processos
      no comunicador MPI_COMM_WORLD                        */
   MPI_Comm_size(MPI_COMM_WORLD, size);
   *out_depend = 0;
}
int sisal_mpi_comm_rank(in_depend, rank, out_depend)
int in_depend, /* entrada para a dependencia entre funcoes */
    *rank,       /* retorno do rank do processo            */
    *out_depend; /* saida para a dependencia entre funcoes */
{
/* chamada função MPI que retorna o rank do processo
   chamador    da função no comunicador MPI_COMM_WORLD     */
   MPI_Comm_rank(MPI_COMM_WORLD, rank);
   *out_depend = 0;
}

% A partir daqui codigo SISAL utilizando sisal_mpi_comm_size
% e sisal_mpi_comm_rank.

%$c=sisal_mpi_comm_size
%$c=sisal_mpi_comm_rank
...
global sisal_mpi_comm_size(depend:integer
                           returns integer, integer)
global sisal_mpi_comm_rank(depend:integer
                           returns integer, integer)
...
dep1 := 0;
...
    rank, dep2 := sisal_mpi_comm_rank(dep1);
    ...
% a funcao que segue so executara após sisal_mpi_comm_rank
% pois depende do valor em dep2 para executar, entao dep2 e
% um arco que causa a dependencia de dados entre as duas
% funcoes

    size, dep3 := sisal_mpi_comm_size(dep2);
...
```

A diferença entre a quantidade e versatilidade da declaração de tipos em *C* e *SISAL* foi outra questão a ser resolvida. Em *C*, quando temos uma função com um parâmetro declarado como `void *`, este indica que um identificador de qualquer tipo poderá ser passado para ele. *SISAL*, por ser fortemente tipado, Cann (1992b), Skedzielewski (1991), não aceita esta forma de declaração de tipo genérico. Para contornar este problema foi necessário que, para cada um dos tipos *SISAL* possíveis de interfaceamento com *C*, fosse criada uma função para receber um tipo de dado específico. Lembrando que os tipos de *SISAL/IF1* a serem utilizados no

interfaceamento com *C* são *integer, real, character, double_real* e *arrays* unidimensionais destes tipos.

Exemplo 34) A função de comunicação coletiva de *MPI*, `MPI_Bcast` possui as seguintes funções para o interfaceamento com *SISAL/IF1*:

```
/* declaração em MPI da função MPI_Bcast */
     int MPI_Bcast(void *buffer,  ....)

/* iniciando outro modulo */
/* funcoes para interfaceamento entre SISAL/IF1 e MPI */

/* funcao que realiza broadcast de um dado do tipo inteiro */
int sisal_mpi_bcast_int(sendbuf, ...)
   int sendbuf;

/* funcao que realiza broadcast de um dado do tipo real     */
float sisal_mpi_bcast_real(sendbuf, ...)
   float sendbuf;

/* funcao para broadcast de um dado do tipo character       */
char sisal_mpi_bcast_char(sendbuf, ...)
   char  sendbuf`;

/* funcao para broadcast de um dado do tipo double_real     */
double sisal_mpi_bcast_double(sendbuf, ...)
   double  sendbuf;

/* funcao para o broadcast de um array de inteiros          */
int sisal_mpi_bcast_array_int(sendbuf, ...)
   int *sendbuf;

/* funcao para broadcast de um dado do tipo array de real   */
float sisal_mpi_bcast_array_real(sendbuf, ...)
float *sendbuf;

/* funcao para broadcast de dado do tipo array de character*/
char sisal_mpi_bcast_array_char(sendbuf, ...)
char *sendbuf;

/* função para broadcast dado do tipo array de double_real */
double sisal_mpi_bcast_array_double(sendbuf, ...)
double *sendbuf;
```

Uma listagem completa das funções para a interface entre *SISAL/IF1* e *MPI* está disponível no Apêndice A.

5.5. Transformação do programa *SISAL/IF1*

Um programa *SPMD* que segue o modelo *mestre/escravo*, Haines; Böhm (1993), possui as seguintes características:

- O processo mestre é responsável pela divisão dos *loops* paralelos em *slices*, que então serão executados em *escravos* paralelos. A distribuição dos *slices* entre os *escravos* pode ser feita em três estágios para um grande número de *escravos*, portanto criando paralelismo na fase de distribuição dos *slices*. Neste caso, chamado de distribuição multi-nível (*MLD - Multiple Level Distribution*), os *slices* são criados e distribuídos para *escravos* em paralelo. Para um pequeno número de processos *escravos*, o *mestre* pode criar e distribuir os *slices* diretamente entre os *escravos*, evitando a sobrecarga dos estágios intermediários do esquema de distribuição multi-nível. Isto é chamado distribuição de nível único. Uma vez que os *slices* tenham sido distribuídos, o *mestre* deve esperar (usando barreira de sincronização) até que o processamento de todos os *slices* tenha sido completado. Os resultados dos *slices* são reduzidos seqüencialmente para um nível único e em paralelo na distribuição multi-nível;
- Cada processo *escravo* verifica continuamente sua lista de trabalho local por *slices* a serem executados. Concluído o *slice*, o processo *escravo* envia a mensagem de conclusão para seu processo *mestre* ou *sub-mestre* atualizar o resultado global com valores computados localmente;
- A barreira de sincronização está completa quando o número de registros de conclusão recebidos pelo *mestre* é igual ao número do *slices* distribuídos para processos *escravo*;
- Um *slice* executando em um processador pode ser subdividido em um número de *threads* para habilitar a execução *multithread*. Neste modo, os *threads* são chaveados após encontrar um referência remota e iniciar o pedido. O *slice* é completado quando todos os *threads* correspondentes completarem;
- como um *slice* pode conter *loops* paralelos aninhados, um processo *escravo* pode iniciar um processo *mestre*. Assim todos processadores podem executar processos *mestre* e *escravo* e múltiplos processos *mestre* podem estar executando no mesmo processador.

Em nosso trabalho utilizamos o modelo *mestre/escravo* de processos com distribuição de *slices* em apenas um nível.

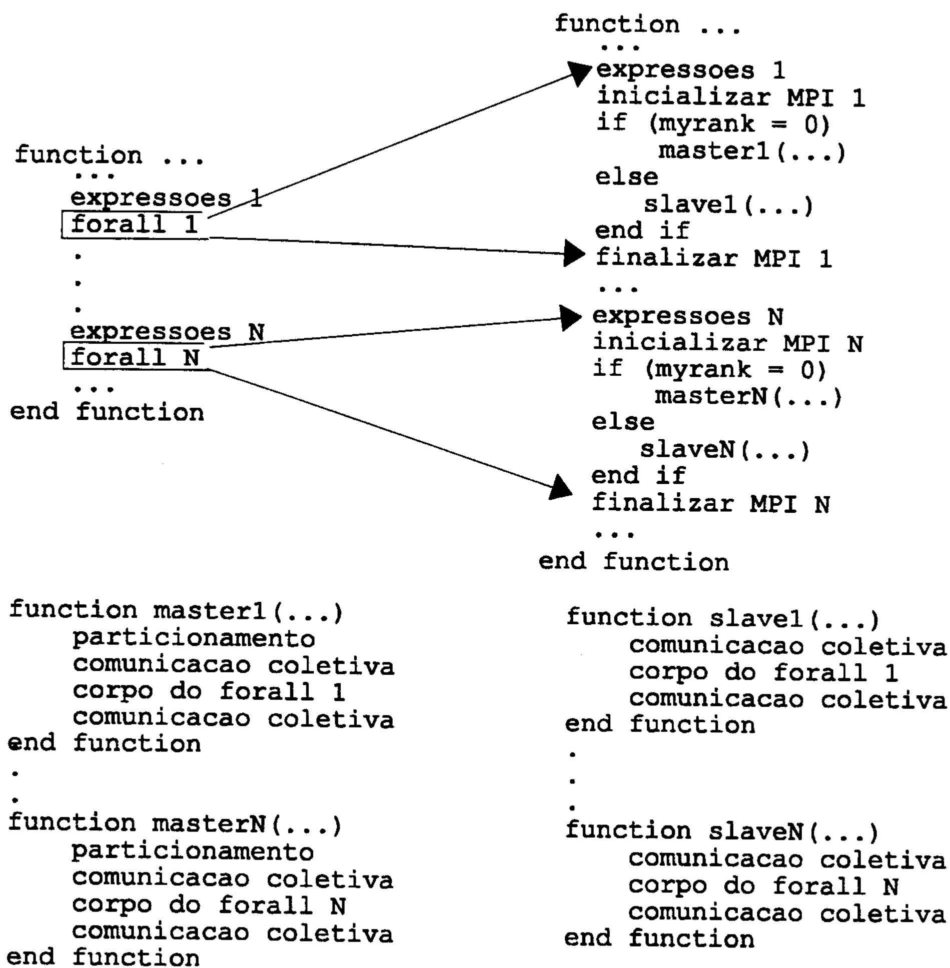


Figura 10 - Particionamento *slice* criando processos *mestre* e *escravo* no modelo *SPMD*

Podemos observar na Figura 10 acima que no início e no final englobando as alterações são realizadas operações para a inicialização e finalização de *MPI* como é exigido Message Passing Forum (1994b), Snir et al (1995).

Cada expressão *forall* foi substituída por uma expressão condicional com chamada para função *mestre* em uma condição e a função *escravo* na outra. Tanto a função *mestre* como a *escravo* possuem, além de uma fatia do *forall* original, rotinas de comunicação para a recepção e transmissão de dados. Por exemplo, os dados internos à expressão *forall* devem ser enviados a todos os processos, então a operação adequada para a tarefa é um *broadcast*. Já os índices das expressões *forall*

devem ser transmitidos por operações *scatter*, pois cada *forall* deve receber um valor diferente para o par, limite inferior e limite superior.

A forma para o cálculo do número de fatias do *forall* foi baseada nó número de processos executando, Sarkar; Cann (1990), em *MPI_*COMM_WORLD. Os índices foram calculados da seguinte maneira:

- Nº de elementos em cada fatia = total de instâncias do *forall* original / nº de processos
- Novo limite inferior = Nº de elementos em cada fatia * i + Limite inferior do *forall* original onde i é o *rank* do processo
- Novo limite superior = Novo limite inferior + Nº de elementos em cada fatia - 1

Após a expressão *forall* podemos observar novamente expressões para comunicação coletiva necessárias para reagrupar os resultados das várias fatias do *forall*. A operações a serem realizadas são *gather* ou *reduce* de acordo com a parte de retorno da expressão *forall*.

As possíveis partes de retorno de uma expressão *forall* de *SISAL* são apresentadas no Capítulo 2, seção 2.6. Tomando-as como referência, as seguintes relações foram estabelecidas:

- Expressões *forall* com retorno do tipo **`return array of`** `expressão` teriam seus resultados reagrupados com chamadas para funções *gather*

Exemplo 35) Associação da operação *gather* para os array abaixo

```
for i in 1, 10
return array of i --> chamada para sisal_mpi_gather_array_int
       array of 1.0 --> chamada para sisal_mpi_gather_array_real
end for
```

- Expressões *forall* com retorno do tipo **`return value of`** `expressão de redução` teriam os resultados reagrupados com chamadas para funções *reduce*

Exemplo 36) Associação da operação *reduce* para as reduções abaixo

```
for i in 1, 10
return value of sum i --> chamada para sisal_mpi_reduce_int
       value of sum 1.0 --> chamada para sisal_mpi_reduce_real
end for
```

Realizados todos estes estudos, definições e decisões tomadas foi possível a elaboração de um algoritmo para realizar as alterações necessárias nos programas *IF1*.

5.6. Algoritmo

O algoritmo foi desenvolvido de forma a percorrer o grafo até o final, realizando as alterações necessárias. No primeiro passo, o algoritmo procura pelos limites da primeira função que ainda não tenha percorrido. Em seguida, para cada função procura pelo primeiro *forall* não percorrido, marcando início e final caso encontre algum. Se encontrou um *forall*, realiza as alterações necessárias para a transformação em *SMPD*.

Abaixo o algoritmo para as alterações necessárias nos programas *IF1*.

```
enquanto não percorrer todo o grafo faça
   encontrar os limites da função
      marcar o início da função
      marcar o final da função

   enquanto não encontrar a marca de final da função faça
      procurar por nós forall
      se encontrou nó forall
         marcar o inicio do nó forall
         marcar o final do nó forall

         inserir chamada para a função sisal_mpi_comm_size

         inserir a expressão if para SMPD
            inserir chamada para a função sisal_mpi_comm_rank
            se o número do rank é igual ao rank do mestre
               chamada para a função mestre
            senão
               chamada para a função escravo
            fim se

         iniciar declaração para a função mestre
            calcular o limite inferior e superior do nó forall
            chamada de sisal_mpi_scatter para o limite inferior
            chamada de sisal_mpi_scatter para o limite superior
            se existe nomes de valores dentro do forall
               distribui-los com chamadas de sisal_mpi_bcast
            fim se

            copiar o forall da função original
            alterar os nomes dos limites do forall
            acertar as entradas do forall
            remover o forall original
```

```
            inserir chamadas para sisal_mpi_gather/reduce
        fim da declaração do mestre

        iniciar a declaração para a função escravo
            chamada de sisal_mpi_scatter para o limite inferior
            chamada de sisal_mpi_scatter para o limite superior


            se existe valores não constantes dentro do forall
                recebê-los com chamadas de sisal_mpi_bcast
            fim se
            copiar o nó forall do mestre
            inserir chamadas para sisal_mpi_gather/reduce
        fim da declaração do escravo
    fim se
  fim enquanto
fim enquanto
```

5.7. Implementação e execução

Baseado no algoritmo exibido anteriormente, foi implementado em linguagem *C* um programa, a fim de testar as propostas de alterações para que um programa *SISAL* possa ser executado utilizando a biblioteca *MPI.* O compilador utilizado foi o *gcc* (*GNU C*) e o programa foi nomeado temporariamente de *changes*.

Este programa toma como entrada um arquivo *IF1*, gerado a partir de um fonte *SISAL*, transforma-o num programa *SPDM*, realiza os particionamento dos *forall* juntamente com inserções para chamadas de rotinas *MPI* e retornando um arquivo também no formato *IF1*. O código *IF1* foi obtido utilizando-se o compilador *OSC* com diretiva `-IF1`, Cann (1992a), que faz com que o *OSC* finalize sua execução após a tradução de *SISAL* para *IF1* ou seja após o *front-end*, (Figura 9).

Como o código gerado por nosso programa é *IF1* podemos utilizar novamente o compilador *OSC* para realizarmos mais uma tradução. Desta vez, utilizamos o compilador *OSC* para traduzir o código *IF1* para código *C*. A princípio pode parecer estranho que não estejamos gerando o código executável diretamente. Lembremos porém que o compilador *OSC* está configurado para fazer uso apenas da sua biblioteca e não da biblioteca *MPI*. Então, este passo intermediário é necessário por hora, para que possamos então a partir do código *C* gerado e utilizando o *gcc*, realizarmos a compilação e a ligação com as bibliotecas exigidas por *SISAL* e *MPI*.

No passo seguinte, a execução do código gerado, fazemos uso do suporte fornecido pelo *LAM*.

Exemplo 37) Abaixo exibimos os passos realizados a transformação de *SISAL* para que possa executar utilizando a biblioteca *MPI*.

```
osc -IF1 forall1.sis
(tradução de SISAL para IF1)

changes forall1.if1 forallmpi1.if1
(particionamento e chamadas para MPI)
osc -C forallmpi1.if1 -O forallmpi1.c
(geração de código C)


gcc   -I/usr/local/bin -I/usr/local/lam/h -DSUN
      -DSUN4_OS -o forall1 sisal_mpi.c forallmpi1.c
      /usr/local/bin/p-srt0.o  -L/usr/local/bin
      -L/usr/local/lam/lib -lsisal -lmpi -ltstdio
      -ltools -ltrillium -largs -lt -lm
(compilação do código C)
```

onde: `sisal_mpi.c` é o arquivo com as chamadas *MPI* adaptadas para *SISAL*
`forallmpi1.c` é o arquivo gerado a partir do programa *IF1/SPMD*
`forall1` é o código executável.

As diretivas utilizadas foram obtidas unindo àquelas do compilador OSC às do compilador *hcc* de *LAM*. Entre as diretivas podemos citar `-lmpi` para a biblioteca *MPI* e `-lsisal` para a biblioteca *SISAL*.

A execução do programa segue as regras estabelecidas em *LAM*, Burns et al (1994), Pacheco (1995), Dongarra et al (1995). Primeiro, devemos criar um arquivo que liste os nomes dos computadores onde desejamos executar os programa. Sejam eles *pink.ifqsc.sc.usp.br*, *floyd.ifqsc.sc.usp.br* e *green.ifqsc.sc.usp.br*. Então podemos criar um arquivo chamado `lamhosts` contendo as seguintes linhas:

```
pink.ifqsc.sc.usp.br
floyd.ifqsc.sc.usp.br
green.ifqsc.sc.usp.br
```

Para iniciar o *LAM* em cada máquina utilizamos o comando:

```
lamboot  lamhosts
```

com os nomes dos computadores em cada linha associados com os identificadores, Burns et al (1994), `n0`, `n1` e `n2` respectivamente.

Utilizamos então estas associações para criar um outro arquivo chamado `lamschema` com as seguintes linhas :

```
foralll n0
foralll n1
foralll n2
```

para informar em quais computadores serão executados os programas.

Para executarmos a aplicação utilizamos

```
mpirun -v lamschema
```

finalizada a execução terminamos o *LAM* com

```
wipe lamhosts.
```

O método utilizado para verificar a correção dos programas com nossas modificações foi a comparação dos resultados obtidos com um programa sem as alterações.

Podemos verificar estes resultados no Apêndice B. Nele estão além, dos resultados, os códigos *IF1* dos programas que geraram os resultados e o fonte *SISAL*.

5.8. Conclusões

Apresentamos neste capítulo as várias etapas envolvidas para a realização de nosso trabalho, as quais podemos dividir em, basicamente, três grupos:

1. Estudos dos fundamentos da linguagem *SISAL* e do compilador *OSC*; linguagem de grafos acíclicos *IF1* e o padrão *MPI* .
2. Escolha do código fonte a ser manipulado (*IF1*), da estrutura a ser paralelizada (*forall*), método de particionamento (*slice*), modelo de programação (*SPMD*) e modelo de execução de processos (*master/slave*).
3. Desenvolvimento de um algoritmo para a paralelização, implementação do algoritmo e realização de testes para validar o método proposto.

Capítulo 6

6. Conclusões

O objetivo deste trabalho foi realizar um estudo das alterações necessárias em *SISAL*, para que pudéssemos fazer uso da biblioteca do padrão *MPI*. Como um dos resultados deste estudo optou-se pela manipulação do código *IF1* ao invés de *SISAL* e apresentamos propostas de alterações e adaptações nos programas fontes originais. Para a validação da nossa proposta implementamos um programa para inserir as transformações nos programas fontes e para verificar a validade das alterações realizamos execuções dos programas com as alterações. Abaixo as principais conclusões deste trabalho e sugestões para futuras pesquisas.

6.1) SISAL/IF1

A princípio estava definido que o código fonte a ser manipulado era *SISAL*, mas após estudos realizados constatamos que seria mais vantajoso a manipulação do código *IF1*. Além disso, *IF1* é uma representação direta de *SISAL* o que não traria problemas para a proposta de nosso trabalho. Constatamos que a simplicidade de *IF1*, composto de grafos nós, arcos e tipos facilitaria bastante o nosso trabalho na manipulação do código fonte. Um dos motivos para tal constatação foi que o número reduzido de estruturas de *IF1* permitiu que um pequeno número de regras de produção definisse a linguagem. Além disso, verificamos que a manipulação de um código fonte na forma de grafos permite que análises de dependência sejam realizadas simplesmente pela procura por arcos comuns entre os nós (operações). Como resultado destas propriedade *IF1* apresenta uma semântica clara e bem definida.

6-2) *MPI*

Do padrão *MPI* utilizamos principalmente as rotinas de comunicação coletiva na transmissão das mensagens além do suporte para a manipulação das mensagens e processos, o que reduz consideravelmente a abrangência dos problemas a serem resolvidos para a utilização de um sistema de passagem de mensagem. Caso este suporte não estivesse disponível necessitaríamos dispender um tempo razoável para a definição das rotinas necessárias. Além da redução dos esforços, ainda nos beneficiamos da portabilidade resultante da padronização.

6-3) Propostas de alteração

O modelo de execução de processos escolhido (*mestre/escravo*) foi uma grata surpresa pois além de sua simplicidade, adaptou-se perfeitamente ao modelo de particionamento escolhido, *slice*, tornando clara as alterações necessárias nos programas *SISAL* originais. A estrutura escolhida para ser particionada, *forall*, colaborou na clareza para a transformação de um programa *SISAL* para o *SPMD*, em virtude de ser inerentemente paralela, podendo então ser particionada diretamente.

6-3) Implementação

Apesar de todos os benefícios citados acima, nos defrontamos com a restrição do número de tipos suportados pela interface de *OSC* com a linguagem *C*, o que nos impôs uma limitação quanto aos tipos de dados que manipularíamos, conseqüentemente em relação aos tipos de programas que poderíamos utilizar para realizar nossas alterações. No entanto, este fator não impediu que pudéssemos testar as alterações propostas, ainda que com um número limitado de tipos. Podemos fazer esta afirmação pois na distribuição dos dados aos processos, distribuição dos índices do *forall* e reagrupamento dos resultados não estão associados ao tipo do dado mas à estrutura apresentada pelo *forall*.

A biblioteca para a ligação entre *SISAL* e *C* consegue contornar o problema em relação à incompatibilidade entre os tipos de *SISAL* e de *MPI*. Apesar de contornar o problemas dos tipos de *SISAL* e *MPI*, esta biblioteca não resolve

completamente o problema pois, ela deveria ser criada dinamicamente de modo a suprir todas as necessidade momentâneas e não ser fixa como foi definida. Esta necessidade se deve ao fato que sendo fixa, não poderá suportar qualquer dimensão de *array*.

Podemos verificar pelo algoritmo e pela implementação que a utilização de um código fonte de um programa na forma de grafo realmente reduz o tamanho do programa que se é escrito para a manipulação do grafo.

6-4) Trabalhos Futuros

Apresentamos como sugestão para extensões de nosso trabalho:

- Integração do programa para transformações de *SISAL* no compilador *OSC*;
- Alteração do programa para a criação de bibliotecas para interfaceamento de *SISAL* e *MPI* dinamicamente;
- Desenvolvimento de uma forma para o interfaceamento de todos os tipos de *SISAL* com *C;*
- Particionamento dos *loops forall* para mais níveis além do mais externo;
- Estudo de otimizações nas transmissões de dados;

Bibliografia

- ALMASI, G. S., AND GOTTLIEB, A., Highly Parallel Computing, Benjamin/ Cummings, 1994.

- BARENDREGT, H., AND LEEUWEN, M. V., Functional Programming and The Language TALE, Lecture Notes in Computer Science 224, Current Trends in Concurrency - Overview and Tutorial, 122-207, March, 1986.

- BURNS, G., DAOUD R., AND VAIGL, J., LAM: An Open Cluster Enviroment for MPI, Ohio Supercomputer Center, 1994, http://www.epm.ornl.gov/~walker/mpi/papers/lam-mpi.ps.Z.

- CANN, D. C., The optimizing SISAL compiler: Version 12.0, Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, April, 1992.

- CANN, D. C., SISAL 1.2: A Brief Introduction and Tutorial, UCRL-MA-110620, Lawrence Livermore National Laboratory, May, 1992.

- CANN, D. C., AND EVRIPIDOU, P., Advanced Array Optimization for High Performance Functional Language, IEEE Transaction on Parallel and Distributed Systems, 229-239, vol. 6, nº. 3, March, 1995.

- DONGARRA, J. J., OTTO, S. W., SNIR, M., AND WALKER, D., An Introduction to the MPI Standard, January, 1995, http://www.epm.ornt/~walker/mpi/papers/cacm95.ps.Z.

- FEO, J. T., AND CANN, D. C., A Report on the Sisal Language Project, Journal of Parallel and Distributed Computing 10, 349-366, December, 1990.

- FOISY, C., AND CHAILLOUX, E., Caml Fligth: a Portable SPMD Extension of ML for Distributed Memory Multiprocessor, High Performance Functional Computing, 83-96, April, 1995.

- FREEH, V. W., AND ANDREWS, G. R., fsc: A Sisal Compiler for Both Distributed- and Shared-Memory Machines, High Performance Functional Computing, 164-172, April, 1995.

- GAUDIOT, J. L., AND LEE, L. T., Multiprocessor Systems Programming in a High Level Data-Flow Language, Lecture Notes in Computer Science 259 - PARLE - Parallel Architecture and Language Europe, 134-151, June, 1987.

- GRIT, D. H., Sisal on a Message Passing Architecture, CONPAR 90 VAPP IV - Joint International Conference on a vector and Parallel Processing - Lecture Notes in Computer Science, 721-731, April, 1990.

- HAINES, M. D., Distributed Runtime Suport For Task and Data Management, Colorado State University Technical Report CS-93-110, Colorado State University, Fort Coolins, Colorado, August, 1993.

- HAINES, M., AND BÖHM, W., Task Management, Virtual Shared Memory, and Multithreading in a Distributed Memory Implementation of Sisal, PARLE 93 - Parallel Architecture and Language Europe - Lecture Notes in Computer Science - Springer Verlag, 12-23, June, 1993.

- MALARD, J., MPI: A Message Passing Interface Standard, History, Overview and Current Status, Edinburg Parallel Computing Centre, The University of Edinburg, Technology Watch Report, JIS New Technology Initiative, 1995, http://www.epcc.edu.ac.uk/epcc-tec/documents.html,

- MESSAGE PASSING FORUM, MPI: A Message Passing Interface Standard, International Journal of Supercomputer Applications, vol. 8, nos 3/4, 1994

- MESSAGE PASSING FORUM, MPI: Message Passing Interface Standard, Proc of Supercomputing 93, 878-883, 1993.

- OLDEHOEFT, R. R., CANN, D. C., Applicattive Paralelism on a Shared-Memory Multiprocessor, IEEE Software 5, 1, 62-70, January, 1988.

- PACHECO, P. S., A User's Guide to MPI, University of San Francisco, California, March, 1995, http://math.usfca.edu/pub/MPI/mpi-guide.ps.

- PANDE, S. S., AGRAWAL, D. P., AND MAUNEY, J., A Threshold Scheduling Strategy for Sisal on Distributed Memory Machines, Journal of Parallel and Distributed Computing, 21, 223-236, May, 1994

- SARKAR, V., AND CANN, D., POSC - a Partitioning and Optimizing SISAL Compiler, Proc. of the ACM International Conference on Supercomputing, 148-163, June, 1990.

- SARKAR, V., AND HENNESSY, J., Compile Time Partitioning and Scheduling of Parallel Programs, Proc. SIGPLAN 1986 - Symposium on Compiler Construction ACM, 17-26, June 1986

- SKEJELLUM, A., DOSS, N. E., AND BANGALORE, P. V., Writing Libraries in MPI, Proc. of the 1993 Scalable Parallel Libraries Conference, IEEE Computer Society Press, 166-173, October, 1993.

- SKEDZIELEWSKI, S. K., Parallel and Functional Languages and Compilers, ACM PRESS, 1991

- SKEDZIELEWSKI, S., AND GLAUERT, J., IF1 An Intermediate Form for Applicative Languages, Manual M-170, Lawrence Livermore National Laboratory, Livermore, California, January, 1985

- SKEDZIELEWSKI, S. K., AND WELCOME, M. L., Data Flow Graph Optimization in IF1, Functional Programming Language and Computer Architecture - Lecture Notes in Computer Science, 17-34, 1985

- SNIR, M., OTTO, S., HUSS-LEDERMAN, S., AND DONGARRA, J., MPI: The Complete Reference, The MIT Press, 1995
  ou http://www.netlib.org/utk/papers/mpi-book/mpi-books.ps

- STONE, H. S., High Performance Computer Architecture, Addion-Wesley, 1993.

- WELCOME, M., SKEDZIELESKI, S., YATES, R. K., AND RANELLETTI, J., IF2: Applicattive Language Intermediate Form with Explicit Memory Management, Lawrence Livermore National Laboratory, Manual M-195, Lawrence Livermore National Laboratory, Livermore, California, December, 1986.

Apêndice A

```
/***************** Interface entre rotinas SISAL e MPI *************
  Neste arquivo estao colocadas as funcoes de interfaceamento neces-
  sarias para chamada de rotinas MPI dentro de programas SISAL
 *****************************************************************/

#include <mpi.h>

/******************************************************************
 *                    Operacoes de reducao                        *
   ------------------------------------------------
    Operacao      | constante  | Operacao     | constante
   ------------------------------------------------
    MPI_MAX       |     1      | MPI_LOR      |    7
    MPI_MIN       |     2      | MPI_BOR      |    8
    MPI_SUM       |     3      | MPI_LXOR     |    9
    MPI_PROD      |     4      | MPI_BXOR     |   10
    MPI_LAND      |     5      | MPI_MAXLOC   |   11
    MPI_BAND      |     6      | MPI_MINLOC   |   12

*******************************************************************
*              Funcoes para interface entre MPI e SISAL          *

*******************************************************************
int MPI_Comm_size(MPI_Comm comm, int *size)

-------------------------------------------------------------------
global sisal_mpi_comm_size(dependence: integer
                           returns integer, integer)
-----------------------------------------------------------------*/
int sisal_mpi_comm_size(in_depend, size, out_depend)
int in_depend, /*entrada utilizada como dependencia entre funcoes */
    *size,     /* numero de processos no grupo MPI_COMM_WORLD     */
    *out_depend;/*saida a ser utilizada na dependencia das funcoes*/
{
   MPI_Comm_size(MPI_COMM_WORLD,size);
   *out_depend = 0;
}

/******************************************************************
   int MPI_Comm_rank(MPI_Comm comm, int *rank)

-------------------------------------------------------------------
   global sisal_mpi_comm_rank(dependence:integer
                              returns integer, integer)
-----------------------------------------------------------------*/
int sisal_mpi_comm_rank(in_depend, rank, out_depend)
int  in_depend,/*entrada utilizada como dependencia entre funcoes */
     *rank,    /* numero de processos no grupo MPI_COMM_WORLD     */
    *out_depend;/* saida a ser utilizada na dependencia das funcoes*/
{
   MPI_Comm_rank(MPI_COMM_WORLD,rank);
   *out_depend = 0;
}


/*****************************************************************
```

```
   int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
                 int root, MPI_Comm comm)
---------------------------------------------------------------------
global sisal_mpi_bcast_int(sendbuf, root, dependence :integer
                           returns integer, integer)
-------------------------------------------------------------------*/
int sisal_mpi_bcast_int(sendbuf, root, in_depend, recvbuf,
                        out_depend)
int  sendbuf,     /* buffer com o dado a ser distribuidos           */
     root,        /* numero do emissor da mensagem                  */
     in_depend,   /* entrada utilizada na dependencia das funcoes */
    *recvbuf,     /* buffer de recepcao da mensagem enviada         */
    *out_depend;  /* saida utilizada na dependencia das funcoes    */
{
/* Chamada para a rotina MPI_Bcast com dado de entrada inteiro      */

   MPI_Bcast(&sendbuf, 1, MPI_INT, root, MPI_COMM_WORLD);
   *recvbuf = sendbuf;
   *out_depend = 0;
}

/*------------------------------------------------------------------
global sisal_mpi_bcast_real(sendbuf: real; root, dependence: integer
                            returns real, integer)
-------------------------------------------------------------------*/
float sisal_mpi_bcast_real(sendbuf, root, in_depend,  recvbuf,
                           out_depend )
float  sendbuf;    /* buffer com o dado a ser distribuidos          */
int    root,       /* numero do emissor da mensagem                 */
       in_depend; /* entrada utilizada na dependencia das funcoes */
float *recvbuf;   /* buffer de recepcao da mensagem enviada         */
int   *out_depend; /* saida utilizada na dependencia das funcoes   */
{
/* Chamada para a rotina MPI_Bcast com dado de entrada real         */

   MPI_Bcast(&sendbuf, 1, MPI_FLOAT, root, MPI_COMM_WORLD);
   *recvbuf = sendbuf;
   *out_depend = 0;
}

/*------------------------------------------------------------------
global sisal_mpi_bcast_char(sendbuf: character;
                            root, dependence: integer
                            returns integer, character)
-------------------------------------------------------------------*/
char sisal_mpi_bcast_char(sendbuf, root, in_depend, recvbuf,
                          out_depend)
char  sendbuf;     /* buffer com o dado a ser distribuidos          */
int   root,        /* numero do emissor da mensagem                 */
      in_depend; /*entrada utilizada como dependencia das funcoes */
char *recvbuf;     /* buffer de recepcao da mensagem enviada        */
int  *out_depend; /* saida utilizada na dependencia entre funcoes */
{
 /* Chamada para a rotina MPI_Bcast com dado de entrada character */
   MPI_Bcast(&sendbuf, 1, MPI_CHAR, root, MPI_COMM_WORLD);
   recvbuf = &sendbuf;
   *out_depend = 0;
}

/*------------------------------------------------------------------
global sisal_mpi_bcast_double(sendbuf: double;
                              root, dependence: integer
```

```
                                returns integer, double)
----------------------------------------------------------------------*/
double sisal_mpi_bcast_double(sendbuf, root, in_depend, recvbuf,
                              out_depend)
double  sendbuf;  /* buffer com o dado a ser distribuidos            */
int     root,      /* numero do emissor da mensagem                  */
        in_depend; /*entrada utilizada na dependencia das funcoes */
double *recvbuf;    /* buffer de recepcao da mensagem enviada        */
int    *out_depend; /*saida utilizada na dependencia entre funcoes*/
{
/* Chamada para a rotina MPI_Bcast com dado de entrada character   */
   MPI_Bcast(&sendbuf, 1, MPI_DOUBLE, root, MPI_COMM_WORLD);
   *recvbuf = sendbuf;
   *out_depend = 0;
}

/*--------------------------------------------------------------------
global sisal_mpi_bcast_array_int(sendbuf: array[integer];
                                 count, root, dependence: integer;
                                 vd: array[integer]
                                 returns array[integer])
----------------------------------------------------------------------*/
int sisal_mpi_bcast_array_int(sendbuf, count, root, in_depend, vd,
                              recvbuf, out_depend)
int *sendbuf,      /* buffer com o dado a ser distribuidos           */
     count,        /* numero de elementos a serem distribuidos       */
     root,         /* numero do emissor da mensagem                  */
     in_depend,    /* entrada utilizada na dependencia das funcoes   */
    *vd,           /* descritor do buffer de recepcao da mensagem    */
                   /* necessario em razao de SISAL                   */
    *recvbuf,      /* buffer de recepcao da mensagem enviada         */
    *out_depend;   /* saida utilizada na dependencia entre funcoes   */

{  int i;          /* variavel contadora no loop                     */

/* Chamada de uma rotina MPI                                          */
   MPI_Bcast(sendbuf, count, MPI_INT, root, MPI_COMM_WORLD);
   *out_depend = 0;
}

/*--------------------------------------------------------------------
 global sisal_mpi_bcast_array_real(sendbuf: array[real];
                                   count, root, dependence: integer;
                                   vd: array[integer]
                                   returns array[real])
----------------------------------------------------------------------*/
float sisal_mpi_bcast_array_real(sendbuf, count, root, in_depend,
                                 vd, recvbuf, out_depend)
float *sendbuf;    /* buffer com o dado a ser distribuidos           */
int    count,      /* numero de elementos a serem distribuidos       */
       root,       /* numero do emissor da mensagem                  */
       in_depend, /* entrada utilizada na dependencia das funcoes */
      *vd;         /* descritor do buffer de recepcao da mensagem */
                   /* necessario em razao de SISAL                   */
float *recvbuf;    /* buffer de recepcao da mensagem enviada         */
int   *out_depend; /* saida utilizada na dependencia das funcoes     */

{  int i;          /* variavel contadora no loop                     */
   /* chamada de uma rotina MPI */
   MPI_Bcast(sendbuf, count, MPI_FLOAT, root, MPI_COMM_WORLD);

   for (i = 0; i < vd[4]-vd[3]; i++)
```

```
      recvbuf[i] = sendbuf[i];
   *out_depend = 0;
}

/*-------------------------------------------------------------------
global sisal_mpi_bcast_array_char(sendbuf: array[character];
                                  count, root, dependence :integer;
                                  vd: array[integer]
                                  returns array[character]; integer)
-------------------------------------------------------------------*/
char sisal_mpi_bcast_array_char(sendbuf, count, root, in_depend, vd,
                                recvbuf, out_depend)
char *sendbuf;     /* buffer com o dado a ser distribuidos          */
int   count,       /* numero de elementos a serem distribuidos      */
      root,        /* numero do emissor da mensagem                 */
      in_depend,   /* entrada utilizada na dependencia das funcoes */
     *vd;          /* descritor do buffer de recepcao da mensagem  */
                   /* necessario em razao de SISAL                  */
char *recvbuf;     /* buffer de recepcao da mensagem enviada        */
int  *out_depend;  /* saida utilizada na dependencia das funcoes    */
{  int i;          /* variavel contadora no loop                    */

/* Chamada de uma rotina MPI                                         */
   MPI_Bcast(sendbuf, count, MPI_CHAR, root, MPI_COMM_WORLD);

   for (i = 0; i < vd[4]-vd[3]; i++)
      recvbuf[i] = sendbuf[i];
   *out_depend = 0;
}

/*-------------------------------------------------------------------
global sisal_mpi_bcast_array_double(sendbuf: array[double];
                                  count, root, dependence: integer;
                                  vd: array[integer]
                                  returns array[double], out_depend)
-------------------------------------------------------------------*/
double sisal_mpi_bcast_array_double(sendbuf, count, root, in_depend,
                                    vd, recvbuf, out_depend)
double *sendbuf;    /* buffer com o dado a ser distribuidos         */
int     count,      /* numero de elementos a serem distribuidos     */
        root,       /* numero do emissor da mensagem                */
        in_depend,/* entrada utilizada na dependencia das funcoes */
       *vd;         /* descritor do buffer de recepcao da mensagem */
                    /* necessario em razao de SISAL                 */
double *recvbuf;    /* buffer de recepcao da mensagem enviada       */
int  *out_depend;   /* saida utilizada na dependencia das funcoes   */
{  int i;           /* variavel contadora no loop                   */
/* Chamada de uma rotina MPI                                         */
   MPI_Bcast(sendbuf, count, MPI_DOUBLE, root, MPI_COMM_WORLD);

   for (i = 0; i < vd[4]-vd[3]; i++)
      recvbuf[i] = sendbuf[i];
   *out_depend = 0;
}

/*******************************************************************
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)

/*-------------------------------------------------------------------
global sisal_mpi_gather_array_int(sendbuf: array[integer];
```

```
                                              sendcount, recvcount, root,
                                              dependence: integer;
                                              vd: array[integer]
                                              returns array[double], integer)
-----------------------------------------------------------------*/
sisal_mpi_gather_array_int(sendbuf, sendcount, recvcount, root,
                           in_depend, vd, recvbuf, out_depend)
int *sendbuf,    /* buffer com o dado a ser enviado para o coletor */
     sendcount,  /* numero de elementos a serem distribuidos       */
     recvcount,  /* numero de elementos a serem recebidos          */
     root,       /* numero do processo receptor da mensagem        */
     in_depend,  /* entrada utilizada na dependencia das funcoes   */
    *vd, /* vetor descritor do parametro buffer de recepcao        */
    *recvbuf,    /* buffer de recepcao das mensagens enviadas      */
    *out_depend; /* saida utilizada na dependencia das funcoes     */
{
   MPI_Gather(sendbuf, sendcount, MPI_INT, recvbuf, sendcount,
              MPI_INT, root, MPI_COMM_WORLD);
}

/*-----------------------------------------------------------------
global sisal_mpi_gather_array_real(sendbuf: array[real];
                                   sendcount, recvcount, root,
                                   dependence: integer;
                                   vd: array[integer]
                                   returns array[real], integer)
-----------------------------------------------------------------*/
float sisal_mpi_gather_array_real(sendbuf, sendcount, recvcount,
                                  root, in_depend, vd, recvbuf,
                                  out_depend)
float *sendbuf; /* buffer com o dado a ser enviado para o coletor */
int    sendcount,  /* numero de elementos a serem distribuidos    */
       recvcount,  /* numero de elementos a serem recebidos       */
       root,       /* numero do processo receptor da mensagem     */
       in_depend, /* entrada utilizada na dependencia das funcoes */
      *vd;     /* vetor descritor do parametro buffer de recepcao */
float *recvbuf;    /* buffer de recepcao das mensagens enviadas   */
int   *out_depend; /* saida utilizada na dependencia das funcoes  */
{
   MPI_Gather(sendbuf, sendcount, MPI_FLOAT, recvbuf, recvcount,
              MPI_FLOAT,root, MPI_COMM_WORLD);
   *out_depend = 0;
}

/*-----------------------------------------------------------------
global sisal_mpi_gather_array_character(sendbuf: array[character];
                                        sendcount, recvcount, root,
                                        dependence: integer;
                                        vd: array[integer]
                                   returns array[character], integer)
-----------------------------------------------------------------*/
char sisal_mpi_gather_array_char(sendbuf, sendcount, recvcount,
                                 root, in_depend, vd, recvbuf,
                                 out_depend)
char *sendbuf;  /* buffer com o dado a ser enviado para o coletor */
int   sendcount,  /* numero de elementos a serem distribuidos     */
      recvcount,  /* numero de elementos a serem recebidos        */
      root,       /* numero do processo receptor da mensagem      */
      in_depend,  /* entrada utilizada na dependencia das funcoes */
     *vd;   /* vetor descritor do parametro do buffer de recepcao */
char *recvbuf;    /* buffer de recepcao das mensagens enviadas    */
int   *out_depend; /* saida utilizada na dependencia das funcoes  */
```

```
{
   MPI_Gather(sendbuf, sendcount, MPI_CHAR, recvbuf, recvcount,
              MPI_CHAR, root, MPI_COMM_WORLD);
   *out_depend = 0;
}
/********************************************************************
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                MPI_Comm comm)

--------------------------------------------------------------------
   global sisal_mpi_scatter_array_int(sendbuf: array[integer];
                                      root, dependence: integer
                                      returns integer, integer)
------------------------------------------------------------------*/
int sisal_mpi_scatter_array_int(sendbuf, root, in_depend, recvbuf,
                                out_depend)
int  *sendbuf,     /* buffer com os dados a serem distribuidos    */
      root,        /* processo emissor dos dados                  */
      in_depend,  /* entrada utilizada na dependencia das funcoes */
     *recvbuf,     /* buffer de recepcao dos dados                */
     *out_depend; /* entrada utilizada na dependencia das funcoes */
{
   MPI_Scatter(sendbuf,1,MPI_INT, recvbuf, 1, MPI_INT, root,
               MPI_COMM_WORLD);
   *out_depend = 0;
}
/********************************************************************
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm)

--------------------------------------------------------------------
global sisal_mpi_reduce_int(sendbuf, op, root, dependence: integer;
                            returns integer, integer)
------------------------------------------------------------------*/
int sisal_mpi_reduce_int(sendbuf, op, root, in_depend, recvbuf,
                         out_depend)
int sendbuf,      /* buffer com os dados a serem distribuidos     */
    op,           /* operacao de reducao                          */
    root,         /* processo emissor dos dados                   */
    in_depend,   /* entrada utilizada na dependencia entre funcoes */
   *recvbuf,      /* buffer de recepcao dos dados                 */
   *out_depend;  /* entrada utilizada na dependencia das funcoes  */
{  switch (op) {
   case 1:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_INT,MPI_MAX,root,
                 MPI_COMM_WORLD);
      break;
   case 2:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_INT,MPI_MIN,root,
                 MPI_COMM_WORLD);
      break;
   case 3:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_INT,MPI_SUM,root,
                 MPI_COMM_WORLD);
      break;
   case 4:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_INT,MPI_PROD,root,
                 MPI_COMM_WORLD);
      break;
   case 5:
```

```
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_INT,MPI_LAND,root,
                 MPI_COMM_WORLD);
      break;
   case 6:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_INT,MPI_BAND,root,
                 MPI_COMM_WORLD);
      break;
   case 7:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_INT,MPI_LOR,root,
                 MPI_COMM_WORLD);
      break;
   case 8:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_INT,MPI_BOR,root,
                 MPI_COMM_WORLD);
      break;
   case 9:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_INT,MPI_LXOR,root,
                 MPI_COMM_WORLD);
      break;
   case 10:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_INT,MPI_BXOR,root,
                 MPI_COMM_WORLD);
      break;
   case 11:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_INT,MPI_MAXLOC,root,
                 MPI_COMM_WORLD);
      break;
   case 12:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_INT,MPI_MINLOC,root,
                 MPI_COMM_WORLD);
      break;
   };
   *out_depend = 0;
}

/*------------------------------------------------------------------
global sisal_mpi_reduce_real(sendbuf: real; op, root, dependence:
                             integer returns real, integer)
------------------------------------------------------------------*/
float sisal_mpi_reduce_real(sendbuf, op, root, in_depend, recvbuf,
                            out_depend)
float sendbuf;      /* buffer com os dados a serem distribuidos   */
int    op,          /* operacao de reducao                        */
       root,        /* processo emissor dos dados                 */
       in_depend; /* entrada utilizada na dependencia das funcoes */
float *recvbuf;     /* buffer de recepcao dos dados               */
int   *out_depend;/* entrada utilizada na dependencia das funcoes */
{  switch (op) {
   case 1:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_FLOAT,MPI_MAX,root,
                 MPI_COMM_WORLD);
      break;
   case 2:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_FLOAT,MPI_MIN,root,
                 MPI_COMM_WORLD);
      break;
   case 3:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_FLOAT,MPI_SUM,root,
                 MPI_COMM_WORLD);
      break;
   case 4:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_FLOAT,MPI_PROD,root,
```

```
                MPI_COMM_WORLD);
      break;
   case 5:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_FLOAT,MPI_LAND,root,
                 MPI_COMM_WORLD);
      break;
   case 6:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_FLOAT,MPI_BAND,root,
                 MPI_COMM_WORLD);
      break;
   case 7:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_FLOAT,MPI_LOR,root,
                 MPI_COMM_WORLD);
      break;
   case 8:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_FLOAT,MPI_BOR,root,
                 MPI_COMM_WORLD);
      break;
   case 9:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_FLOAT,MPI_LXOR,root,
                 MPI_COMM_WORLD);
      break;
   case 10:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_FLOAT,MPI_BXOR,root,
                 MPI_COMM_WORLD);
      break;
   case 11:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_FLOAT,MPI_MAXLOC,root,
                 MPI_COMM_WORLD);
      break;
   case 12:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_FLOAT,MPI_MINLOC,root,
                 MPI_COMM_WORLD);
      break;
   };
   *out_depend = 0;
   return(*recvbuf);
}

/*------------------------------------------------------------------
global sisal_mpi_reduce_double(sendbuf: double_real; op, root,
                               dependence: integer
                               returns double_real, integer)
------------------------------------------------------------------*/
double sisal_mpi_reduce_double(sendbuf,op,root,in_depend,recvbuf,
                               out_depend)
double  sendbuf;    /* buffer com os dados a serem distribuidos   */
int     op,         /* operacao de reducao                        */
        root,       /* processo emissor dos dados                 */
        in_depend; /* entrada utilizada na dependencia de funcoes */
double *recvbuf;    /* buffer de recepcao dos dados               */
int *out_depend;  /* entrada utilizada na dependencia de funcoes  */
{  switch (op) {
   case 1:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_DOUBLE,MPI_MAX,root,
                 MPI_COMM_WORLD);
      break;
   case 2:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_DOUBLE,MPI_MIN,root,
                 MPI_COMM_WORLD);
      break;
   case 3:
```

```
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_DOUBLE,MPI_SUM,root,
                 MPI_COMM_WORLD);
      break;
   case 4:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_DOUBLE,MPI_PROD,root,
                 MPI_COMM_WORLD);
      break;
   case 5:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_DOUBLE,MPI_LAND,root,
                 MPI_COMM_WORLD);
      break;
   case 6:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_DOUBLE,MPI_BAND,root,
                 MPI_COMM_WORLD);
      break;
   case 7:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_DOUBLE,MPI_LOR,root,
                 MPI_COMM_WORLD);
      break;
   case 8:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_DOUBLE,MPI_BOR,root,
                 MPI_COMM_WORLD);
      break;
   case 9:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_DOUBLE,MPI_LXOR,root,
                 MPI_COMM_WORLD);
      break;
   case 10:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_DOUBLE,MPI_BXOR,root,
                 MPI_COMM_WORLD);
      break;
   case 11:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_DOUBLE,MPI_MAXLOC,root,
                 MPI_COMM_WORLD);
      break;
   case 12:
      MPI_Reduce(&sendbuf,recvbuf,1,MPI_DOUBLE,MPI_MINLOC,root,
                 MPI_COMM_WORLD);
      break;
   };
   *out_depend = 0;
   return(*recvbuf);
}
```

Apêndice B

- **Programa que utiliza um comando de redução**

Código *SISAL*

```
define main

function main(returns integer)
   for K in 1, 10
       returns value of sum K
   end for
end function
```

Código *IF1* sem *MPI*

```
T 1 1 0     %na=Boolean
T 2 1 1     %na=Character
T 3 1 2     %na=Double
T 4 1 3     %na=Integer
T 5 1 4     %na=Null
T 6 1 5     %na=Real
T 7 1 6     %na=WildBasic
T 8 10
T 9 8 4      0
T 10 3       0     9
T 11 4       4
T 12 8       4     9
T 13 3       12    9
C$  C Faked IF1CHECK
C$  D Nodes are DFOrdered
C$  F Livermore Frontend   Version1.8
X      10     "main"      %sl=5
{ Compound    1   0
G      0      %sl=7
N 1    142    %sl=7
L             1 1    4 "1" %mk=V
L             1 2    4 "10"      %mk=V
E      1 1    0 1    11      %na=k       %mk=V
G      0      %sl=7
G      0      %sl=7
N 1    149    %sl=8
L             1 1    13 "SUM"     %mk=V
L             1 2    4 "0" %mk=V
E      0 1    1 3    11      %na=k       %mk=V %sl=7
E      1 1    0 1    4      %mk=V
} 1 0 3 0 1 2       %sl=7
E      1 1    0 1    4      %mk=V
```

Código *IF1* com *MPI*

```
CC$sisal_mpi_comm_size
CC$sisal_mpi_comm_rank
CC$sisal_mpi_scatter_array_int
CC$sisal_mpi_reduce_int
CC$sisal_mpi_bcast_int
CC$sisal_mpi_bcast_array_int
T 1 1 0     %na=Boolean
T 2 1 1     %na=Character
T 3 1 2     %na=Double
```

```
T 4 1 3    %na=Integer
T 5 1 4    %na=Null
T 6 1 5    %na=Real
T 7 1 6    %na=WildBasic
T 8 10
T 9 8 4     0
T 10 8      4      9
T 11 3      9      10
T 12 0      4
T 13 8      12     10
T 14 3      13     10
T 15 8      4      10
T 16 8      4      15
T 17 3      16     10
T 18 3      0      9
T 19 3      10     9
T 20 4      4
C$  C Faked IF1CHECK
C$  D Nodes are DFOrdered
C$  F Livermore Frontend   Version1.8
I      11     "sisal_mpi_comm_size"
I      11     "sisal_mpi_comm_rank"
I      14     "sisal_mpi_scatter_array_int"
I      17     "sisal_mpi_reduce_int"
X      18     "main"
N 1    120
L             1 1    11 "sisal_mpi_comm_rank"       %mk=V
L             1 2    4 "0" %mk=V
N 2    124
E      1 1    2 1    4      %mk=V
L             2 2    4 "0" %mk=V
N 3    129
E      2 1    3 1    1      %mk=V
{ Compound    4    1
G      0
E      0 1    0 1    4      %mk=V
G      0
N 1    120
L             1 1    18 "main.slave"    %mk=V
E      1 1    0 1    4      %mk=V
G      0
N 1    120
L             1 1    19 "main.master"   %mk=V
L             1 2    4 "1" %mk=V
L             1 3    4 "10"       %mk=V
E      1 1    0 1    4      %mk=V
} 4 1 3 0 1 2
E      3 1    4 1    4      %mk=V
E      4 1    0 1    4      %mk=V
G      19     "main.master"
N 1    120
L             1 1    11 "sisal_mpi_comm_size"       %mk=V
L             1 2    4 "0" %mk=V
N 2    135
E      0 2    2 1    4      %mk=V
E      0 1    2 2    4      %mk=V
N 3    135
E      1 1    3 1    4      %mk=V
L             3 2    4 "1" %mk=V
N 4    135
E      1 1    4 1    4      %mk=V
L             4 2    4 "1" %mk=V
```

```
N 5    141
E      2 1    5 1    4       %mk=V
L             5 2    4  "1"  %mk=V
N 6    136
E      5 1    6 1    4       %mk=V
E      1 1    6 2    4       %mk=V
N 7    140
E      6 1    7 1    4       %mk=V
L             7 2    4  "0"  %mk=V
N 8    140
E      6 1    8 1    4       %mk=V
L             8 2    4  "0"  %mk=V
N 9    129
E      8 1    9 1    1       %mk=V
{ Compound   10    1
G      0
E      0 1    0 1    4       %mk=V
G      0
N 1    135
E      0 2    1 1    4       %mk=V
E      0 3    1 2    4       %mk=V
N 2    141
E      1 1    2 1    4       %mk=V
L             2 2    4  "1"  %mk=V
N 3    122
E      2 1    3 1    4       %mk=V
E      0 4    3 2    4       %mk=V
E      3 1    0 1    4       %mk=V
G      0
N 1    135
E      0 2    1 1    4       %mk=V
E      0 3    1 2    4       %mk=V
N 2    141
E      1 1    2 1    4       %mk=V
L             2 2    4  "1"  %mk=V
N 3    122
E      2 1    3 1    4       %mk=V
E      0 4    3 2    4       %mk=V
N 4    141
E      3 1    4 1    4       %mk=V
L             4 2    4  "1"  %mk=V
E      4 1    0 1    4       %mk=V
} 10 1 3 0 1 2
E      9 1    10 1   4       %mk=V
E      0 2    10 2   4       %mk=V
E      0 1    10 3   4       %mk=V
E      1 1    10 4   4       %mk=V
{ Compound   11    0
G      0
N 1    142
L             1 1    4  "0"  %mk=V
E      0 1    1 2    4       %mk=V
E      1 1    0 4    20      %mk=V
G      0
N 1    152
E      0 4    1 1    4       %mk=V
E      0 2    1 2    4       %mk=V
N 2    141
E      1 1    2 1    4       %mk=V
E      0 3    2 2    4       %mk=V
E      2 1    0 5    4       %mk=V
G      0
```

```
N 3   107
L           3 1   4 "1" %mk=V
E      0 5  3 2   20    %mk=V
E      3 1  0 1   12    %mk=V
} 11 0 3 0 1 2
E      3 1  11 1  4     %mk=V
E      10 1 11 2  4     %mk=V
E      0 1  11 3  4     %mk=V
N 12  106
L           12 1  4 "0" %mk=V
E      4 1  12 2  4     %mk=V
E      10 1 12 3  4     %mk=V
N 13  115
E      11 1 13 1  12    %mk=V
L           13 2  4 "0" %mk=V
N 14  129
E      7 1  14 1  1     %mk=V
{ Compound  15   1
G      0
E      0 1  0 1   4     %mk=V
G      0
E      0 2  0 1   12    %mk=V
G      0
N 1   135
E      0 3  1 1   4     %mk=V
L           1 2   4 "1" %mk=V
N 2   135
E      0 3  2 1   4     %mk=V
E      0 5  2 2   4     %mk=V
N 3   135
E      0 4  3 1   4     %mk=V
E      2 1  3 2   4     %mk=V
N 4   113
E      0 2  4 1   12    %mk=V
E      1 1  4 2   4     %mk=V
E      3 1  4 3   4     %mk=V
E      4 1  0 1   12    %mk=V
} 15 1 3 0 1 2
E      14 1 15 1  4     %mk=V
E      12 1 15 2  12    %mk=V
E      1 1  15 3  4     %mk=V
E      10 1 15 4  4     %mk=V
E      6 1  15 5  4     %mk=V
N 16  120
L           16 1  14 "sisal_mpi_scatter_array_int"    %mk=V
E      13 1 16 2  12    %mk=V
L           16 3  4 "0" %mk=V
E      1 2  16 4  4     %mk=V
N 17  120
L           17 1  14 "sisal_mpi_scatter_array_int"    %mk=V
E      15 1 17 2  12    %mk=V
L           17 3  4 "0" %mk=V
E      16 2 17 4  4     %mk=V
N 18  141
E      16 1 18 1  4     %mk=V
E      17 1 18 2  4     %mk=V
N 19  135
E      18 1 19 1  4     %mk=V
L           19 2  4 "1" %mk=V
{ Compound  20   0
G      0
N 1   142
```

```
E       0 1    1 1    4       %mk=V
E       0 2    1 2    4       %mk=V
E       1 1    0 3    20      %mk=V
G       0
G       0
N 1     149
L              1 1    19 "SUM"     %mk=V
L              1 2    4 "0" %mk=V
E       0 3    1 3    20     %mk=V
E       1 1    0 1    4      %mk=V
} 20  0 3  0 1 2
E       16 1   20 1   4       %mk=V
E       19 1   20 2   4       %mk=V
N 21    120
L              21 1   17 "sisal_mpi_reduce_int"      %mk=V
E       20 1   21 2   4      %mk=V
L              21 3   4 "3" %mk=V
L              21 4   4 "0" %mk=V
E       17 2   21 5   4      %mk=V
E       21 1   0 1    4      %mk=V
G       18     "main.slave"
N 1     103
L              1 1    4 "1" %mk=V
L              1 2    4 "0" %mk=V
N 2     120
L              2 1    14 "sisal_mpi_scatter_array_int"     %mk=V
E       1 1    2 2    12     %mk=V
L              2 3    4 "0" %mk=V
L              2 4    4 "0" %mk=V
N 3     120
L              3 1    14 "sisal_mpi_scatter_array_int" %mk=V
E       1 1    3 2    12     %mk=V
L              3 3    4 "0" %mk=V
E       2 2    3 4    4      %mk=V
N 4     141
E       2 1    4 1    4      %mk=V
E       3 1    4 2    4      %mk=V
N 5     135
E       4 1    5 1    4      %mk=V
L              5 2    4 "1" %mk=V
{ Compound    6    0
G       0
N 1     142
E       0 1    1 1    4       %mk=V
E       0 2    1 2    4       %mk=V
E       1 1    0 3    20      %mk=V
G       0
G       0
N 1     149
L              1 1    19 "SUM"     %mk=V
L              1 2    4 "0" %mk=V
E       0 3    1 3    20     %mk=V
E       1 1    0 1    4      %mk=V
} 6  0 3  0 1 2
E       2 1    6 1    4       %mk=V
E       5 1    6 2    4       %mk=V
N 7     120
L              7 1    17 "sisal_mpi_reduce_int"      %mk=V
E       6 1    7 2    4      %mk=V
L              7 3    4 "3" %mk=V
L              7 4    4 "0" %mk=V
E       2 2    7 5    4      %mk=V
```

```
E     7 1   0 1   4      %mk=V
```

Resultado obtido com programa original

```
SUN SISAL 1.2 V12_9_1
55
```

Resultado obtido com programa particionado utilizando as bibliotecas *MPI*

```
2325 reduce running on n0 (o)
ld.so: warning: /usr/lib/libc.so.1.6 has older revision than
expected 8
SUN SISAL 1.2 V12_9_1
12782 reduce running on n1
55
```

• Programa que utiliza o comando de *gather*

Código *SISAL*

```
define main

function main(returns array[integer], array[integer])
   let
        z:=array_fill(0,9,0);
   in
        for i in 0,9
        returns array of i
                array of 47 * i
        end for
   end let
end function
```

Código *IF1* sem *MPI*

```
T 1 1 0     %na=Boolean
T 2 1 1     %na=Character
T 3 1 2     %na=Double
T 4 1 3     %na=Integer
T 5 1 4     %na=Null
T 6 1 5     %na=Real
T 7 1 6     %na=WildBasic
T 8 10
T 9 0   4
T 10 8  9       0
T 11 8  9       10
T 12 3  0       11
T 13 4  4
C$  C Faked IF1CHECK
C$  D Nodes are DFOrdered
C$  F Livermore Frontend   Version1.8
X       12      "main"  %sl=3
{ Compound   2   0
G       0       %sl=9
N 1     142     %sl=9
L               1 1     4 "0"   %mk=V
L               1 2     4 "9"   %mk=V
E       1 1     0 1     13       %na=i   %mk=V
G       0       %sl=9
N 1     152     %sl=10
L               1 1     4 "47"  %mk=V
E       0 1     1 2     4        %na=i   %mk=V
E       1 1     0 2     4       %mk=V
G       0       %sl=9
N 1     107     %sl=10
L               1 1     4 "1"   %mk=V
E       0 1     1 2     13       %na=i   %mk=V   %sl=9
N 3     107     %sl=11
L               3 1     4 "1"   %mk=V
E       0 2     3 2     13      %mk=V
E       1 1     0 1     9       %mk=V
E       3 1     0 2     9       %mk=V
} 2 0 3 0 1 2   %sl=9
N 3     115     %sl=10
E       2 1     3 1     9       %mk=V
L               3 2     4 "0"   %mk=V
N 4     115     %sl=11
```

```
E        2 2      4 1      9        %mk=V
L                 4 2      4 "0"    %mk=V
E        3 1      0 1      9        %mk=V
E        4 1      0 2      9        %mk=V
```

Código *IF1* com *MPI*

```
CC$sisal_mpi_comm_size
CC$sisal_mpi_comm_rank
CC$sisal_mpi_scatter_array_int
CC$sisal_mpi_gather_array_int
CC$sisal_mpi_bcast_int
CC$sisal_mpi_bcast_array_int
T 1 1 0    %na=Boolean
T 2 1 1    %na=Character
T 3 1 2    %na=Double
T 4 1 3    %na=Integer
T 5 1 4    %na=Null
T 6 1 5    %na=Real
T 7 1 6    %na=WildBasic
T 8 10
T 9 8 4     0
T 10 8      4      9
T 11 3      9      10
T 12 0      4
T 13 8      12     10
T 14 3      13     10
T 15 8      12     0
T 16 8      4      15
T 17 8      4      16
T 18 8      4      17
T 19 8      4      18
T 20 8      12     19
T 21 8      12     9
T 22 3      20     21
T 23 8      4      10
T 24 3      23     10
T 25 8      12     18
T 26 3      25     21
T 27 8      12     15
T 28 3      0      27
T 29 3      17     27
T 30 4      4
C$  C Faked IF1CHECK
C$  D Nodes are DFOrdered
C$  F Livermore Frontend   Version1.8
I     11    "sisal_mpi_comm_size"
I     11    "sisal_mpi_comm_rank"
I     14    "sisal_mpi_scatter_array_int"
I     22    "sisal_mpi_gather_array_int"
I     24    "sisal_mpi_bcast_int"
I     26    "sisal_mpi_bcast_array_int"
X     28    "main"
N 1   106
L           1 1   4 "0" %mk=V
L           1 2   4 "9" %mk=V
L           1 3   4 "0" %mk=V
N 2   120
L           2 1   11 "sisal_mpi_comm_rank"      %mk=V
```

```
L           2 2   4 "0" %mk=V
N 3   124
E     2 1   3 1   4      %mk=V
L           3 2   4 "0" %mk=V
N 4   129
E     3 1   4 1   1     %mk=V
{ Compound   5   1
G     0
E     0 1   0 1   4     %mk=V
G     0
N 1   120
L           1 1   28 "main.slave"   %mk=V
E     1 1   0 1   12    %mk=V
E     1 2   0 2   12    %mk=V
G     0
N 1   120
L           1 1   29 "main.master"  %mk=V
L           1 2   4 "0" %mk=V
L           1 3   4 "9" %mk=V
E     0 2   1 4   12    %mk=V
E     1 1   0 1   12    %mk=V
E     1 2   0 2   12    %mk=V
} 5 1 3 0 1 2
E     4 1   5 1   4     %mk=V
E     1 1   5 2   12    %mk=V
E     5 1   0 1   12    %mk=V
E     5 2   0 2   12    %mk=V
G     29    "main.master"
N 1   120
L           1 1   11 "sisal_mpi_comm_size"      %mk=V
L           1 2   4 "0" %mk=V
N 2   135
E     0 2   2 1   4     %mk=V
E     0 1   2 2   4     %mk=V
N 3   116
E     0 3   3 1   12    %mk=V
N 4   110
E     0 3   4 1   12    %mk=V
N 5   135
E     1 1   5 1   4     %mk=V
L           5 2   4 "1" %mk=V
N 6   135
E     1 1   6 1   4     %mk=V
L           6 2   4 "1" %mk=V
N 7   141
E     2 1   7 1   4     %mk=V
L           7 2   4 "1" %mk=V
N 8   136
E     7 1   8 1   4     %mk=V
E     1 1   8 2   4     %mk=V
N 9   140
E     8 1   9 1   4     %mk=V
L           9 2   4 "0" %mk=V
N 10  140
E     8 1   10 1  4     %mk=V
L           10 2  4 "0" %mk=V
N 11  129
E     10 1  11 1  1     %mk=V
{ Compound  12   1
G     0
E     0 1   0 1   4     %mk=V
G     0
```

```
N 1    135
E      0 2    1 1    4        %mk=V
E      0 3    1 2    4        %mk=V
N 2    141
E      1 1    2 1    4        %mk=V
L             2 2    4  "1"  %mk=V
N 3    122
E      2 1    3 1    4        %mk=V
E      0 4    3 2    4        %mk=V
E      3 1    0 1    4        %mk=V
G      0
N 1    135
E      0 2    1 1    4        %mk=V
E      0 3    1 2    4        %mk=V
N 2    141
E      1 1    2 1    4        %mk=V
L             2 2    4  "1"  %mk=V
N 3    122
E      2 1    3 1    4        %mk=V
E      0 4    3 2    4        %mk=V
N 4    141
E      3 1    4 1    4        %mk=V
L             4 2    4  "1"  %mk=V
E      4 1    0 1    4        %mk=V
} 12  1 3 0 1 2
E      11 1   12 1   4        %mk=V
E      0 2    12 2   4        %mk=V
E      0 1    12 3   4        %mk=V
E      1 1    12 4   4        %mk=V
{ Compound    13    0
G      0
N 1    142
L             1 1    4  "0"  %mk=V
E      0 1    1 2    4        %mk=V
E      1 1    0 4    30       %mk=V
G      0
N 1    152
E      0 4    1 1    4        %mk=V
E      0 2    1 2    4        %mk=V
N 2    141
E      1 1    2 1    4        %mk=V
E      0 3    2 2    4        %mk=V
E      2 1    0 5    4        %mk=V
G      0
N 3    107
L             3 1    4  "1"  %mk=V
E      0 5    3 2    30       %mk=V
E      3 1    0 1    12       %mk=V
} 13  0 3 0 1 2
E      5 1    13 1   4        %mk=V
E      12 1   13 2   4        %mk=V
E      0 1    13 3   4        %mk=V
N 14   106
L             14 1   4  "0"  %mk=V
E      6 1    14 2   4        %mk=V
E      12 1   14 3   4        %mk=V
N 15   115
E      13 1   15 1   12       %mk=V
L             15 2   4  "0"  %mk=V
N 16   129
E      9 1    16 1   1        %mk=V
{ Compound    17    1
```

```
G      0
E      0 1    0 1    4      %mk=V
G      0
E      0 2    0 1    12     %mk=V
G      0
N 1    135
E      0 3    1 1    4      %mk=V
L             1 2    4 "1" %mk=V
N 2    135
E      0 3    2 1    4      %mk=V
E      0 5    2 2    4      %mk=V
N 3    135
E      0 4    3 1    4      %mk=V
E      2 1    3 2    4      %mk=V
N 4    113
E      0 2    4 1    12     %mk=V
E      1 1    4 2    4      %mk=V
E      3 1    4 3    4      %mk=V
E      4 1    0 1    12     %mk=V
} 17 1 3 0 1 2
E      16 1   17 1   4      %mk=V
E      14 1   17 2   12     %mk=V
E      1 1    17 3   4      %mk=V
E      12 1   17 4   4      %mk=V
E      8 1    17 5   4      %mk=V
N 18   120
L             18 1   14 "sisal_mpi_scatter_array_int"     %mk=V
E      15 1   18 2   12     %mk=V
L             18 3   4 "0" %mk=V
E      1 2    18 4   4      %mk=V
N 19   120
L             19 1   14 "sisal_mpi_scatter_array_int"     %mk=V
E      17 1   19 2   12     %mk=V
L             19 3   4 "0" %mk=V
E      18 2   19 4   4      %mk=V
N 20   141
E      18 1   20 1   4      %mk=V
E      19 1   20 2   4      %mk=V
N 21   120
L             21 1   24 "sisal_mpi_bcast_int"       %mk=V
E      3 1    21 2   4      %mk=V
L             21 3   4 "0" %mk=V
E      19 2   21 4   4      %mk=V
N 22   135
E      20 1   22 1   4      %mk=V
L             22 2   4 "1" %mk=V
N 25   135
E      21 1   25 1   4      %mk=V
L             25 2   4 "1" %mk=V
N 26   135
E      21 1   26 1   4      %mk=V
L             26 2   4 "1" %mk=V
N 27   120
L             27 1   24 "sisal_mpi_bcast_int"       %mk=V
E      4 1    27 2   4      %mk=V
L             27 3   4 "0" %mk=V
E      21 2   27 4   4      %mk=V
{ Compound   28    0
G      0
N 1    142
E      0 1    1 1    4      %mk=V
E      0 2    1 2    4      %mk=V
```

```
E       1 1    0 3    30      %mk=V
G       0
N 1     152
L              1 1    4 "47"        %mk=V
E       0 3    1 2    4       %mk=V
E       1 1    0 4    4       %mk=V
G       0
N 1     107
L              1 1    4 "1" %mk=V
E       0 3    1 2    30      %mk=V
N 3     107
L              3 1    4 "1" %mk=V
E       0 4    3 2    30      %mk=V
E       1 1    0 1    12      %mk=V
E       3 1    0 2    12      %mk=V
} 28 0 3 0 1 2
E       18 1   28 1   4       %mk=V
E       22 1   28 2   4       %mk=V
N 30    103
L              30 1   4 "0" %mk=V
L              30 2   4 "1" %mk=V
L              30 3   4 "0" %mk=V
L              30 4   4 "0" %mk=V
L              30 5   4 "0" %mk=V
E       26 1   30 6   4       %mk=V
L              30 7   4 "0" %mk=V
E       25 1   30 8   4       %mk=V
E       4 1    30 9   4       %mk=V
N 31    115
E       28 1   31 1   12      %mk=V
E       18 1   31 2   4       %mk=V
N 32    115
E       28 2   32 1   12      %mk=V
E       18 1   32 2   4       %mk=V
N 33    120
L              33 1   26 "sisal_mpi_bcast_array_int"       %mk=V
E       0 3    33 2   12      %mk=V
E       21 1   33 3   4       %mk=V
L              33 4   4 "0" %mk=V
E       27 2   33 5   4       %mk=V
E       30 1   33 6   12      %mk=V
N 34    120
L              34 1   22 "sisal_mpi_gather_array_int"      %mk=V
E       31 1   34 2   12      %mk=V
E       19 1   34 3   4       %mk=V
E       19 1   34 4   4       %mk=V
L              34 5   4 "0" %mk=V
E       33 2   34 6   4       %mk=V
E       30 1   34 7   12      %mk=V
N 35    120
L              35 1   22 "sisal_mpi_gather_array_int"      %mk=V
E       32 1   35 2   12      %mk=V
E       19 1   35 3   4       %mk=V
E       19 1   35 4   4       %mk=V
L              35 5   4 "0" %mk=V
E       34 2   35 6   4       %mk=V
E       30 1   35 7   12      %mk=V
E       34 1   0 1    12      %mk=V
E       35 1   0 2    12      %mk=V
G       28     "main.slave"
N 1     103
L              1 1    4 "1" %mk=V
```

```
L           1 2   4 "0" %mk=V
N 2   120
L           2 1   14 "sisal_mpi_scatter_array_int"    %mk=V
E     1 1   2 2   12    %mk=V
L           2 3   4 "0" %mk=V
L           2 4   4 "0" %mk=V
N 3   120
L           3 1   14 "sisal_mpi_scatter_array_int"    %mk=V
E     1 1   3 2   12    %mk=V
L           3 3   4 "0" %mk=V
E     2 2   3 4   4     %mk=V
N 4   135
E     3 1   4 1   4     %mk=V
L           4 2   4 "1" %mk=V
N 5   135
E     3 1   5 1   4     %mk=V
L           5 2   4 "1" %mk=V
N 6   141
E     2 1   6 1   4     %mk=V
E     3 1   6 2   4     %mk=V
N 7   120
L           7 1   24 "sisal_mpi_bcast_int"      %mk=V
L           7 2   4 "0" %mk=V
L           7 3   4 "0" %mk=V
E     3 2   7 4   4     %mk=V
N 8   103
L           8 1   4 "0" %mk=V
L           8 2   4 "1" %mk=V
L           8 3   4 "0" %mk=V
L           8 4   4 "0" %mk=V
L           8 5   4 "0" %mk=V
E     5 1   8 6   4     %mk=V
L           8 7   4 "0" %mk=V
E     4 1   8 8   4     %mk=V
L           8 9   4 "0" %mk=V
N 9   135
E     6 1   9 1   4     %mk=V
L           9 2   4 "1" %mk=V
N 10  135
E     7 1   10 1  4     %mk=V
L           10 2  4 "1" %mk=V
N 11  135
E     7 1   11 1  4     %mk=V
L           11 2  4 "1" %mk=V
N 12  120
L           12 1  24 "sisal_mpi_bcast_int"      %mk=V
L           12 2  4 "0" %mk=V
L           12 3  4 "0" %mk=V
E     7 2   12 4  4     %mk=V
{ Compound  13   0
G     0
N 1   142
E     0 1   1 1   4     %mk=V
E     0 2   1 2   4     %mk=V
E     1 1   0 3   30    %mk=V
G     0
N 1   152
L           1 1   4 "47"      %mk=V
E     0 3   1 2   4     %mk=V
E     1 1   0 4   4     %mk=V
G     0
N 1   107
```

```
L           1 1   4 "1" %mk=V
E     0 3   1 2   30    %mk=V
N 3   107
L           3 1   4 "1" %mk=V
E     0 4   3 2   30    %mk=V
E     1 1   0 1   12    %mk=V
E     3 1   0 2   12    %mk=V
} 13 0 3 0 1 2
E     2 1   13 1  4     %mk=V
E     9 1   13 2  4     %mk=V
N 14  103
L           14 1  4 "0" %mk=V
L           14 2  4 "1" %mk=V
L           14 3  4 "0" %mk=V
L           14 4  4 "0" %mk=V
L           14 5  4 "0" %mk=V
E     11 1  14 6  4     %mk=V
L           14 7  4 "0" %mk=V
E     10 1  14 8  4     %mk=V
L           14 9  4 "0" %mk=V
N 15  115
E     13 1  15 1  12    %mk=V
E     2 1   15 2  4     %mk=V
N 16  115
E     13 2  16 1  12    %mk=V
E     2 1   16 2  4     %mk=V
N 17  120
L           17 1  26 "sisal_mpi_bcast_array_int"      %mk=V
E     1 1   17 2  12    %mk=V
E     7 1   17 3  4     %mk=V
L           17 4  4 "0" %mk=V
E     12 2  17 5  4     %mk=V
E     14 1  17 6  12    %mk=V
N 18  120
L           18 1  22 "sisal_mpi_gather_array_int"     %mk=V
E     15 1  18 2  12    %mk=V
E     3 1   18 3  4     %mk=V
E     3 1   18 4  4     %mk=V
L           18 5  4 "0" %mk=V
E     17 2  18 6  4     %mk=V
E     8 1   18 7  12    %mk=V
N 19  120
L           19 1  22 "sisal_mpi_gather_array_int"     %mk=V
E     16 1  19 2  12    %mk=V
E     3 1   19 3  4     %mk=V
E     3 1   19 4  4     %mk=V
L           19 5  4 "0" %mk=V
E     18 2  19 6  4     %mk=V
E     8 1   19 7  12    %mk=V
E     18 1  0 1   12    %mk=V
E     19 1  0 2   12    %mk=V
```

Resultado obtido com o programa original

```
SUN SISAL 1.2 V12_9_1
[ 0,9: # DRC=1 PRC=1
 0
 1
 2
 3
```

```
 4
 5
 6
 7
 8
 9
]
[ 0,9: # DRC=1 PRC=1
 0
 47
 94
 141
 188
 235
 282
 329
 376
 423
]
```

Resultado obtido com programa particionado utilizando as bibliotecas MPI

```
2237 teste running on n0 (o)
ld.so: warning: /usr/lib/libc.so.1.6 has older revision than
expected 8
SUN SISAL 1.2 V12_9_1
12491 teste running on n1
floyd{raul}66: [ 0,9: # DRC=1 PRC=1
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
]
[ 0,9: # DRC=1 PRC=1
 0
 47
 94
 141
 188
 235
 282
 329
 376
 423
]
```