UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E DE ESTATÍSTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Spi+ : Um Interpretador Paralelo para a Linguagem SuperPascal

Dissertação submetida ao Programa de

Pós-Graduação em Ciência da Computação

para a obtenção do grau de mestre.

Luis Fernando Fausto

Florianópolis, Setembro de 1998.

Spi+ : Um Interpretador Paralelo para a Linguagem SuperPascal

Luis Fernando Fausto

Esta dissertação foi julgada adequada para a obtenção do título de

Mestre em Ciência da Computação

especialidade **Sistemas de Computaçãò**

e aprovada em sua forma final pelo Curso de Pós-Graduação em Ciência da Computação

Prof. Dr. Luis Fernando Friedrich, Dr.
Orientador

Prof. Thadeu Botteri Corso, M.Sc.
Co-Orientador

Prof. Jorge Muniz Barreto, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Luis Fernando Friedrich, Dr. (Presidente)

Prof. Thadeu Botteri Corso, M.Sc.

Prof. Olinto J. V. Furtado, Dr.

Prof. Rômulo S. de Oliveira, Dr.)

Dedicatória

Aos meus pais e amigos.

Agradecimentos

Ao Thadeu pela oportunidade oferecida e a confiança de que tudo daria certo.

Ao Friedrich pela ajuda na elaboração do texto e o incentivo na reta final.

A Deluana pelas revisões no texto.

Aos colegas do mestrado que de uma forma ou de outra tiveram um papel importante neste trabalho.

Aos amigos de Floripa, Berlin e Madri, pelo incentivo para que este trabalho se tornasse realidade.

Índice

Lista de Figuras

Resumo

Este trabalho tem como objetivo principal realizar modificações no interpretador da linguagem paralela SuperPascal. Tais modificações visam adicionar características de criação de processos paralelos e comunicação entre processos, que atualmente não estão presentes na implementação do interpretador.

Como embasamento teórico para o trabalho é realizado um estudo dos sistemas distribuídos, dando um enfoque maior à arquitetura e aos mecanismos de comunicação para multicomputadores. Tais conceitos são pré-requisitos básicos para a compreensão das linguagens paralelas, as quais são alvo de um estudo comparativo neste trabalho.

Abstract

This work aims to implement internal changes in the interpreter of the SuperPascal language. Such changes add characteristics of parallel processes creation and communication between processes, that actually are not present in the implementation of the interpreter.

As theoretical basis for this work a study of the distributed systems is shown, giving a special attention to the architecture and communication for multicomputers. Such concepts are basic to the understanding of the parallel languages, theses languages are studed in this work.

1. INTRODUÇÃO

Computadores paralelos representam uma grande oportunidade de desenvolvimento de sistemas de alta performance, e também a resolução de grandes problemas em muitas áreas.

Durante os últimos anos, computadores massivamente paralelos, com tamanhos variando de centenas a milhares de nós processadores, têm se tornado comercialmente disponíveis. Com o passar do tempo eles ganharam um maior reconhecimento como uma poderosa ferramenta para pesquisas científicas, gerência de informações e aplicações de engenharia. Esta tendência está direcionada pelas linguagens de programação paralela e outras ferramentas que contribuem para tornar os computadores paralelos úteis para um grande número de aplicações.

Muitas linguagens foram projetadas para o desenvolvimento de aplicações em computadores paralelos. As linguagens de programação concorrente ou paralela permitem a criação de algoritmos paralelos como sendo um conjunto de ações concorrentes mapeadas em diferentes nós de um multicomputador.

Este trabalho apresenta inicialmente uma revisão sobre os sistemas distribuídos e linguagens de programação paralela, bem como a estrutura por trás destes conceitos, como por exemplo, as topologias e os mecanismos para a troca de mensagens em um multicomputador.

Após estes conceitos é apresentado uma proposta de alterações no interpretador atual da linguagem paralela SuperPascal, denominado spi. Estas modificações visam adicionar características novas ao interpretador na parte de comunicação e criação de processos paralelos.

As alterações propostas foram implementadas e validadas através da execução de programas paralelos, os resultados destas validações são apresentados na conclusão deste trabalho.

2. SISTEMAS DISTRIBUÍDOS

Neste capítulo serão apresentadas algumas definições básicas sobre sistemas distribuídos, suas principais características e como é realizada a comunicação entre processos.

"Um sistema distribuído é uma coleção de computadores independentes que para os usuários comporta-se com um único computador". Esta é a definição básica de um sistema distribuído, que pode ser refinada para atingir uma definição mais ampla e completa, formada pelas quatro motivações principais de um sistema distribuído: transparência, flexibilidade, desempenho e confiança.

2.1 TRANSPARÊNCIA

Provavelmente a característica mais desejada pelos sistemas distribuídos é a de uma imagem única, ou seja, fazer com que o sistema apresente-se ao usuário como um único sistema *time-sharing*. Um sistema que alcance este objetivo é denominado transparente.

2.2 FLEXIBILIDADE

A flexibilidade é um aspecto importante a ser considerado, pois devido a complexidade dos sistemas distribuídos uma opção por uma linha de trabalho que hoje parece correta pode mostrar-se errada em um futuro bem próximo. É necessário que se possa deixar margem a várias possibilidades de realizar uma determinada tarefa, pois desta forma uma das possíveis soluções deve ser a correta. Apesar disto, esta não é uma tarefa simples, porque esta flexibilidade exige um esforço muito maior dos projetistas para não cortar algumas funções do sistema que podem ser utilizadas no futuro.

2.3 DESEMPENHO

Flexibilidade e transparência são características requeridas para um sistema distribuído, mas nenhum sistema altamente flexível e transparente terá sucesso se não

obtiver um bom desempenho. O problema de performance é composto pelo fato de que a comunicação, processo essencial em um sistema distribuído, é tipicamente lenta. Como otimizar este processo para que ele possa se tornar mais rápido é um grande desafio computacional, e uma parte básica dos sistemas modernos.

2.4 CONFIANÇA

A centralização do processamento de informações tem a vantagem de tornar as informações menos suscetíveis aos problemas de um ambiente computacional. Quando projeta-se um sistema distribuído, deve-se tomar os cuidados necessários para garantir a integridade das informações que irão trafegar pelo sistema, pois os fatores que podem afetar a integridade das informações aumentam proporcionalmente ao grau de distribuição do sistema.

2.5 MULTICOMPUTADORES

Nesta seção serão apresentados os multicomputadores, mostrando suas características operacionais e seus vários tipos de redes de interconexão.

Multicomputadores (Figura 2.1) são máquinas paralelas compostas por nós processadores autônomos. Os multicomputadores possuem memória distribuída e têm como principal objetivo prover uma maior capacidade de processamento em relação aos sistemas monoprocessados.

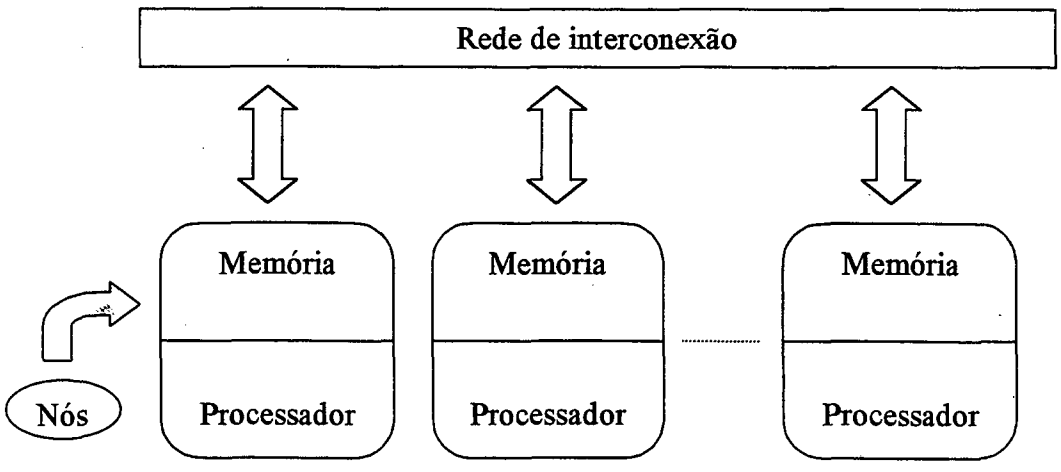

Figura 2.1: Multicomputador.

A programação de multicomputadores baseia-se na divisão de tarefas entre os nós cooperantes, que realizam funções específicas a fim de cooperar para a execução de uma determinada tarefa ou programa paralelo. Para desempenhar tais tarefas os nós comunicam-se entre si através de uma rede de interconexão.

Os multicomputadores podem ser agrupados basicamente em duas gerações distintas: multicomputadores que comunicam-se através do mecanismo de armazenamento e repasse de mensagens, *e* multicomputadores que utilizam-se da comutação de pacotes chaveados por *hardware*.

Os multicomputadores da primeira geração utilizam o esquema de armazenamento e repasse para a troca de mensagens entre os nós. Este mecanismo é implementado por *software* e denomina-se *store-and-forward* [HWA93]. Para um nó comunicar-se com outro não adjacente, a mensagem pode passar por diversos nós intermediários antes de alcançar o seu destino. Cada nó entre a origem e o destino armazena toda a mensagem em um *buffer* antes de enviá-la para o próximo nó. Quanto maior for a distância entre os nós origem e destino maior será o tempo necessário para enviar a mensagem. Este mecanismo ainda possui uma limitação no tamanho da mensagem, que não pode ser maior que os *buffers* intermediários alocados para armazená-la.

Apesar disto este mecanismo possui a qualidade de ter um baixo custo, pois como é implementado por *software*, não necessita de nenhum *hardware* específico. Com algumas modificações na sua idéia inicial, e em situações de grande carga, pode tornar-se competitivo com os mecanismos de comunicação da nova geração de multicomputadores, como mostrado em [FAU97].

Os multicomputadores de segunda geração utilizam-se de um mecanismo de comutação de pacotes chaveado por *hardware*. Cada nó possui um *hardware* específico para o roteamento de mensagens. A comunicação entre os nós não adjacentes acontece através do estabelecimento de circuitos virtuais entre os nós intermediários. Desta forma uma mensagem trafega entre o nó origem e o destino sem que haja armazenamentos intermediários, tornando o tempo necessário para o envio de uma mensagem entre nós menos suscetível à distância do que o mecanismo da geração anterior.

Para realizar a comutação de pacotes geralmente os multicomputadores utilizam redes de interconexão estáticas, como a grelha e o hipercubo, porém alguns modelos utilizam-se de redes de interconexão dinâmicas como o *crossbar*.

Uma rede de interconexão é o conjunto de elementos de chaveamento e de ligação que permitem a comunicação de dados entre processadores e memória em um multicomputador, ou entre processadores em um multicomputador [BHU87]. Quanto a sua topologia as redes de interconexão podem ser estáticas ou dinâmicas.

2.5.1 Redes de interconexão estáticas

Nas redes estáticas os elementos estão conectados através de ligações ponto-a-ponto permanentes, que não mudam durante a execução do programa. Neste caso a solução ideal seria aquela em que todos os nós possuíssem ligações entre si. O problema é que com o aumento do número de nós isto torna-se inviável, devido à grande complexidade envolvida na construção de tal estrutura.

Em função do fator complexidade encontrado nas redes estáticas, com ligações ponto-a-ponto entre todos os nós, o que se pode variar é o mecanismo responsável pelo roteamento de mensagens entre os nós intermediários, pois vai depender muito dele a velocidade que uma mensagem leva para alcançar o seu destino. As topologias de redes estáticas mais conhecidas são o anel, a grelha e o hipercubo.

2.5.1.1 Anel

O anel (Figura 2.2) é a mais simples das topologias estáticas, pois cada nó possui apenas dois canais de comunicação formando um círculo, onde a distância máxima entre dois nós é de (n/2), onde **n** é o número de nós do multicomputador.

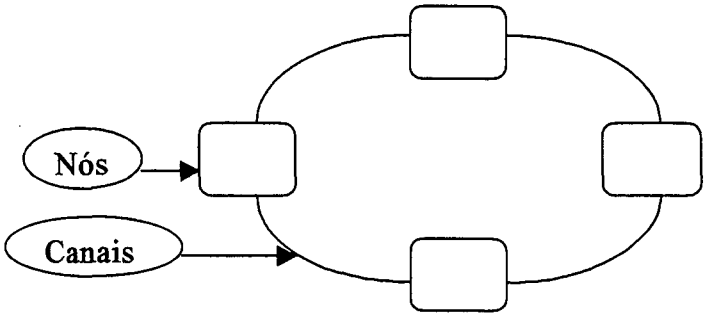

Figura 2.2: Rede de interconexão estática em anel.

2.5.1.2 Grelha

A grelha (Figura 2.3) é uma topologia onde os nós formam uma matriz quadrada **mxm** que resulta em $\mathbf{n} = \mathbf{m}^2$ nós. Cada nó possui conexão direta com todos os seus vizinhos imediatos. Neste caso a distância máxima entre dois nós é de 2(**m**-1). A grelha possui a vantagem de ter mais de um caminho para um mesmo local. Isto permite soluções para os problemas de falhas em nós intermediários e congestionamento de mensagens.

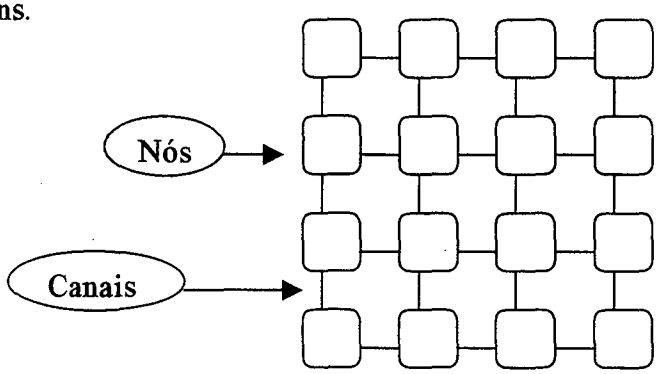

Figura 2.3: Rede de interconexão estática em grelha (4x4).

2.5.1.3 Hipercubo

O hipercubo (Figura 2.4) é uma topologia formada pelo arranjo de m cubos unidimensionais constituindo $\mathbf{n} = 2^{\mathbf{m}}$ nós, onde cada nó possui $\log_2(\mathbf{m}^2)$ canais de comunicação com os nós adjacentes. O hipercubo é uma das redes estáticas com a menor distância entre nós, sendo a distância máxima **m**.

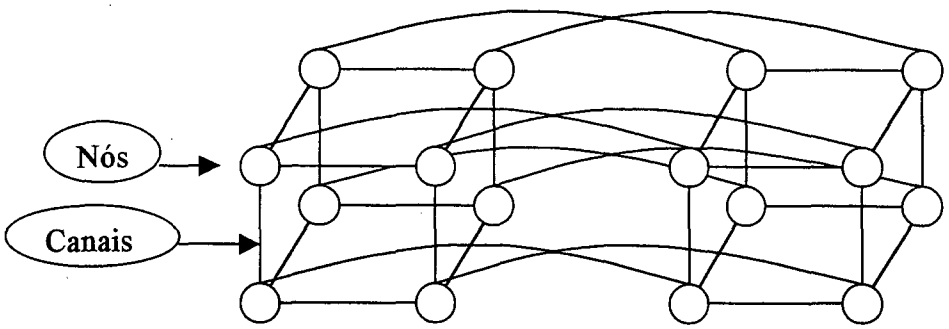

Figura 2.4: Rede de interconexão estática em hipercubo.

O hipercubo é largamente utilizado em máquinas comercias, dentre as quais podem-se destacar: iPSC, nCUBE e o CM-2.

2.5.2 Redes de interconexão dinâmica

As redes dinâmicas são formadas por canais reconfiguráveis dinamicamente para atender as necessidades dos programas paralelos, que normalmente necessitam de diferentes configurações até mesmo em suas fases de execução.

Um programa pode necessitar, por exemplo, de uma estrutura de hipercubo para a carga de dados, e logo após necessitar de uma estrutura de anel isto para a sua execução. Neste caso tanto a configuração dinâmica como a sua reconfiguração são imprescindíveis. Os tipos mais utilizados de interconexão dinâmica são o *barramento* e o *crossbar*.

2.5.2.1 Barramento

O barramento (Figura 2.5) é uma coleção de fios e conectores para a transferência de dados entre processadores, módulos de memória e periféricos ligados ao barramento. O barramento permite apenas uma transação de cada vez entre o emissor e receptor. No caso de múltiplas requisições uma deve ser executada após a outra, competindo pelo recurso. A maior vantagem do barramento sobre as demais redes dinâmicas é o seu baixo custo, porém a banda passante é limitada, e torna-se cada vez menor com o aumento do número de nós do multicomputador.

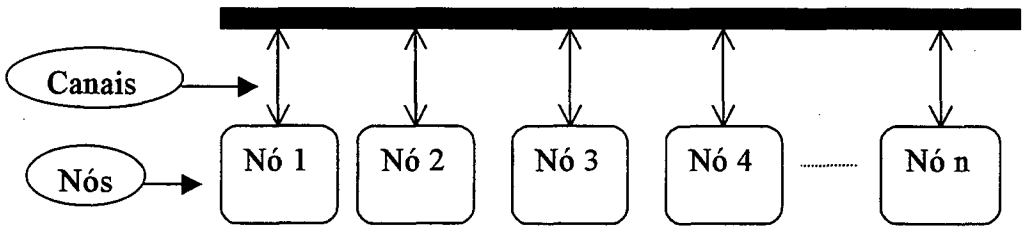

Figura 2.5: Barramento de comunicação.

2.5.2.2 Crossbar

A maior banda passante, como também a maior capacidade de interconexão, são providas pelas redes de interconexão que utilizam o *crossbar* (Figura 2.6). Um *crossbar* assemelha-se muito a uma rede telefônica, onde existem diversos pontos de cruzamento

que estabelecem conexões dinâmicas entre dois pontos. O *crossbar* é o tipo ideal de rede para uso em multicomputadores, pois é reconfigurável e não bloqueante. Ainda assim, seu custo de implementação é alto e a complexidade de seu *hardware* cresce ao quadrado do número de processadores.

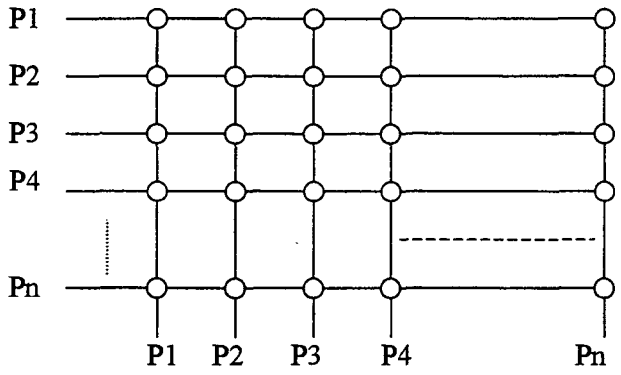

Figura 2.6: *Crossbar* NxN.

2.6 O MULTICOMPUTADOR CRUX

O multicomputador CRUX [COR93] é um projeto em desenvolvimento no Departamento de Informática e de Estatística (INE) da Universidade Federal de Santa Catarina (UFSC). O CRUX pretende suportar a execução simultânea de várias redes de processos comunicantes e de topologia variável unindo os seguintes componentes (Figura 2.7):

- Um grande número de *nós de trabalho* (cada qual com processador e memória privativa, além de vários canais de comunicação).
- Um *nó de controle* responsável pela gestão dos canais físicos (conexão/desconexão) e pela alocação de nós de trabalho para a execução das tarefas.
- Uma *rede principal* (crossbar) para enviar mensagens entre os nós de trabalho através dos canais físicos. Esta rede é configurada pelo nó de controle.

- Uma *rede auxiliar* (barramento) específica para a troca de mensagens entre os nós de trabalho e o nó de controle.

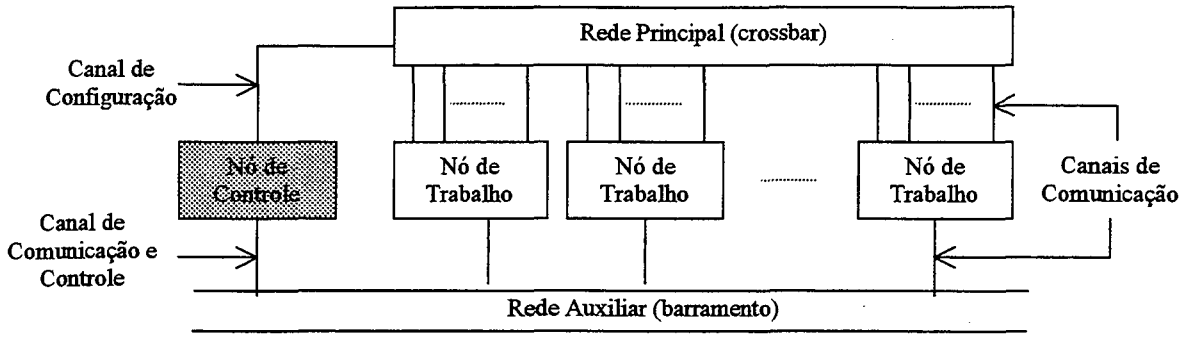

Figura 2.7: Arquitetura do multicomputador Crux.

O mecanismo de comunicação com atribuição de canais físicos por demanda constitui o núcleo do suporte para a execução de redes de processos comunicantes no CRUX. Este mecanismo é composto por dois níveis:

- Rede Principal: Transmite mensagens dos programas paralelos através de canais diretos entre os nós de trabalho onde executam seus processos.

- Rede Auxiliar: Resolve de forma simples e eficaz o problema da conexão dos canais físicos da rede principal, transmitindo mensagens de suporte à execução, entre os nós de trabalho e o nó de controle, com os pedidos de conexão/desconexão de canais físicos da rede principal.

Desta maneira, antes de iniciar uma comunicação efetiva entre dois nós de trabalho, é verificado se existe um canal direto entre eles na rede principal. Caso ele exista, a comunicação pode ser iniciada imediatamente. Caso contrário, deve ser realizado um pedido de conexão antes da comunicação efetiva entre os nós.

O mecanismo de comunicação com atribuição de canais físicos por demanda é implementado por um componente central executado no nó de controle e por componentes locais executados em cada nó de trabalho, incorporados ao núcleo do sistema operacional. O componente central é responsável pela operação da rede principal, conectando e desconectando canais físicos em resposta a solicitações dos nós

de trabalho. Os componentes locais, acessíveis através de uma biblioteca de funções de comunicação, interagem com o componente central para a realização do mecanismo de dois níveis descrito anteriormente.

O mecanismo de comunicação com atribuição de canais físicos por demanda é um procedimento completamente geral, uma vez que a duração dos canais pode variar desde o tempo de transporte de um único pacote até o da execução completa de um programa paralelo.

Para complementar o suporte à execução , um mecanismo de alocação/liberação de nós de trabalho por demanda associado à criação/destruição de processos também é provido. Da mesma maneira que o mecanismo de comunicação, esse serviço é fornecido por um componente central executado no nó de controle e por componentes locais executados em cada nó de trabalho, incorporados ao núcleo do sistema operacional.

Este trabalho de dissertação está fortemente ligado ao multicomputador CRUX, pois irá incorporar alguns dos conceitos apresentados pelo CRUX no seu desenvolvimento. Tal ligação será descrita no capítulo 5.

2.7 COMUNICAÇÃO ENTRE PROCESSOS

Comunicação é parte fundamental de um sistema distribuído, pois sem ela os processos não poderiam trocar informações necessárias para a sua execução, e os usuários não poderiam ter acesso a arquivos ou recursos remotos. Em sistemas distribuídos a comunicação é basicamente provida pela troca de mensagens e pela chamada de procedimentos remotos.

2.7.1 Troca de mensagens

A troca de mensagens é a forma mais utilizada em um sistema distribuído para a comunicação entre processos. Ela consiste em um mecanismo bastante simples, do tipo requisição/resposta, onde um processo cliente solicita uma informação através de uma requisição ao processo servidor, o qual envia uma resposta.

Existem alguns aspectos que devem ser analisados para a implementação de um mecanismo de troca de mensagens, entre eles se a comunicação será direta ou indireta, síncrona ou assíncrona, com ou sem confirmação.

A comunicação direta ocorre quando se explicita o processo ao qual se deseja enviar a mensagem. Esta é uma forma mais rápida de troca de mensagens, mas pouco flexível. A comunicação indireta surge para gerar uma maior flexibilidade, pois os processos, para comunicarem-se entre si, devem utilizar estruturas intermediárias responsáveis pelo mapeamento dos processos em caixas postais. Neste caso os processos não se comunicam diretamente com outros processos, mas com caixas postais que conferem ao sistema uma maior transparência de localidade dos processos.

A comunicação síncrona causa o bloqueio do processo emissor da mensagem até que o processo receptor esteja apto a recebê-la. Neste tipo de comunicação não existe armazenamento intermediário de mensagens, o que é uma vantagem. Por outro lado, impede que o processo emissor realize qualquer outra tarefa antes da recepção da mensagem pelo receptor, o que não é um grande problema já que normalmente os processos dependem do resultado da sua requisição para continuar o seu processamento.

A confirmação do recebimento de uma mensagem oferece maior confiabilidade aos sistemas distribuídos fracamente acoplados (redes), onde uma requisição pode viajar por distâncias longas até o receptor, gerando uma grande possibilidade de se perder no caminho pelos mais diversos motivos. Já em sistemas como multicomputadores e multiprocessadores não é tão forte esta necessidade de confirmação, pois a possibilidade da perda de uma mensagem é infinitamente menor.

Em resumo, a escolha das características a serem utilizadas depende muito dos propósitos que o sistema distribuído pretende atingir. Assim, pode-se ter desde um modelo simples, com comunicação direta e sem confirmação, até modelos mais complexos, com caixas postais e mecanismos de confirmação de mensagens.

2.7.2 Chamada de procedimento remoto

A troca de mensagens entre processos para realização de tarefas remotas possui o inconveniente de que as ações devem ser explicitadas, ou seja, quando deseja-se

acessar uma função de um determinado processo deve-se fazer uso de funções de envio e recepção de mensagens.

A chamada de procedimento remoto (RPC) torna este processo transparente para o usuário, pois ele pode acessar funções remotas como se fossem locais. Para isso, o processo local solicita a execução de um procedimento e fica bloqueado esperando pelo resultado.

Este procedimento é decomposto em mensagens que são enviadas para o processo remoto. Lá elas são executadas, e retornam ao processo solicitante o resultado da sua chamada. Do ponto de vista do usuário tudo acontece como se fosse localmente, apenas chamando um procedimento e aguardando sua finalização.

3. COMUNICAÇÃO NO UNIX

Uma das formas de comunicação mais utilizada em sistemas distribuídos é o mecanismo para troca de mensagens chamado IPC (*Interprocess Communication*). O objetivo deste mecanismo é prover meios para processos cooperantes comunicarem-se e sincronizar as suas ações. A melhor maneira de implementar tais facilidades é através de um sistema de mensagens.

3.1 SISTEMAS DE MENSAGENS

A finalidade de um sistema de mensagens é permitir que processos possam comunicar-se entre si sem a necessidade do compartilhamento de variáveis. Um sistema básico deve possuir no mínimo duas primitivas básicas para a troca de mensagens: **send**(*message*) e **receive**(*message*).

As mensagens enviadas por um processo podem ser de tamanho fixo ou variável. Quando apenas mensagens de tamanho fixo podem ser enviadas a implementação física do sistema é relativamente simples, porém esta restrição toma a tarefa de programação mais difícil. Por outro lado, mensagens de tamanho variável requerem uma implementação física mais complexa, porém facilitam a tarefa da construção de programas.

3.2 IPC NO SYSTEM V

Para aumentar a flexibilidade da comunicação entre processos, os sistemas operacionais UNIX baseados na corrente *System V* adicionaram as seguintes facilidades para a comunicação entre processos:

- Filas de mensagens: as informações que serão trocadas são colocadas em uma estrutura pré-definida. O processo que gera a mensagem especifica o tipo e coloca a mensagem em uma fila controlada pelo sistema. Os processos que acessam a fila podem usar o tipo da mensagem para realizar o recebimento seletivo de mensagens.

- Semáforos: estrutura utilizada para a troca de pequenas quantidades de dados, freqüentemente utilizados para a sincronização entre processos.

- Memória compartilhada: a informação é trocada através do acesso de espaços de dados compartilhados entre os processos. Este é o meio mais rápido para a comunicação entre processos.

Similar a um arquivo, uma IPC deve ser criada antes de ser utilizada. Cada IPC possui um dono e permissões de acesso associadas. Tais atributos definidos na criação podem ser modificados posteriormente usando chamadas de sistema. A Figura 3.1 mostra as funções disponíveis para a manipulação dos recursos IPC do *System V*.

Funcionalidade	Fila de mensagens	Semáforos	Memória compartilhada
Aloca a estrutura e ganha acesso	Msget	Semget	Shget
Controla a IPC, setando características e verificando seu status	Msgctl	Semctl	Shmctl
Operações sobre as IPCs	Msgsnd Msgrcv	Semop	Shmat Shmdt

Figura 3.1: Sinopse das chamadas de sistema sobre IPC no System V.

Semáforos e memória compartilhada não serão detalhados pois não fazem parte do escopo deste projeto. A seguir serão mostradas as características das filas de mensagens, as quais foram utilizadas na implementação deste trabalho.

3.3 FILAS DE MENSAGENS

Uma fila de mensagens é criada utilizando a chamada de sistema **msgget**. Caso não ocorra erro durante sua execução um número não negativo é retornado. Este número é o identificador da fila de mensagens e deve ser utilizado para futuras referências à mesma. Se a criação da fila falhar, é retornado o valor –1 para indicar o erro. O formato do comando **msgget** é:

Msgget(key_t key, int msgflag)

O parâmetro key é utilizado internamente para gerar um identificador único para a fila de mensagens. Já o msgflag indica as características da fila, como por exemplo, quem possui permissão de acesso à fila criada.

Após a fila criada suas permissões podem ser checadas e alteradas através da chamada de sistema **msgctl,** que possui esta sintaxe:

Msgctl(int msqid, int cmd, struct msqid_ds *buf)

O parâmetro msqid é utilizado para identificar a fila em que se deseja realizar a operação, o cmd indica o comando que deve ser executado e o buf leva as informações necessárias para a execução do comando. Existem três ações que podem ser realizadas sobre uma fila de mensagens:

- IPC_STAT: retorna em buf as informações sobre a fila de mensagens, como por exemplo, dono, modos de acesso e número de mensagens na fila.
- IPC_SET: com esta ação o usuário pode modificar características da fila de mensagens, como por exemplo, os modos de acesso.
- IPC_RMID: quando não se deseja mais utilizar a fila de mensagens deve-se retirá-la do sistema e desalocar as estruturas associadas à mesma. Isto é feito com este argumento em uma chamada da função de controle.

Para enviar mensagens para as filas utiliza-se a chamada **msgsnd**, que possui a seguinte sintaxe:

Msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflag)

O parâmetro msqid é um identificador de filas válido retornando após uma chamada de msgget. O segundo parâmetro, msgp, é um ponteiro para a mensagem a ser enviada, o qual deve ser previamente alocado e inicializado. O terceiro parâmetro, msgsz, é o tamanho da mensagem a ser enviada (valor expresso em bytes). O tamanho pode variar entre zero e o limite máximo imposto pelo sistema. O último parâmetro utilizado pelo msgsnd é o msgflg, que é usado para indicar qual ação deve ser tomada caso os limites do sistema para as filas de mensagens sejam alcançados, como por

exemplo, na chegada ao número máximo de bytes em uma fila de mensagens. Msgflg pode assumir os seguintes valores:

- IPC_NOWAIT: neste caso, quando ocorrer o alcance de um limite do sistema a mensagem, não é enviada e o controle retorna ao processo que realizou a chamada, com a variável global *errno* setada para *EAGAIN*.

- 0 : quando o msgflg for setado para zero e os limites do sistema forem alcançados, a chamada ao **msgsnd** ficará bloqueada até que o sistema possua recursos disponíveis para o envio da mensagem, a fila seja removida ou o processo que realizou a chamada receba um sinal para finalizar a sua execução.

No caso de sucesso no envio da mensagem a chamada **msgsnd** retorna o valor 0, caso contrário retorna –1.

Para a recepção de mensagens é utilizada a chamada de sistema **msgrcv**, que possui a seguinte sintaxe:

Msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflag)

O primeiro parâmetro, como em **msgsnd**, é o identificador da fila de mensagens. O segundo, msgp, é um ponteiro para a estrutura que irá receber a mensagem. A estrutura onde a mensagem será colocada deve possuir o primeiro campo do tipo *longint*, o qual irá acomodar o tipo da mensagem. O terceiro parâmetro é o tamanho máximo da mensagem, que deve ser igual ao tamanho da maior mensagem a ser recebida. O quarto , msgtyp, é o tipo da mensagem a ser recebida. Por último, o quinto parâmetro, msgflg, é utilizado para indicar qual ação deve ser tomada se a mensagem do tipo especificado não estiver na fila no momento da chamada da função, ou se o tamanho da mensagem é maior do que o especificado. Os valores pré-definidos são os seguintes:

- IPC_NOWAIT: é utilizado para indicar que a chamada de sistema **msgrcv** não deve ficar bloqueada se não houver mensagens do tipo especificado na fila.

- MSG_NOERROR: é usado para sinalizar para o **msgrcv** que as mensagens de tamanho maior do que o especificado devem ser truncadas. Se esta opção não for

utilizada e ocorrer a recepção de uma mensagem de tamanho superior ao especificado, a chamada **msgrcv** irá retornar –1 e irá setar a variável *errno* para o valor *E2BIG*, indicando que ocorreu um erro.

Estas são as chamadas necessárias para a criação de processos cliente/servidor que trocam mensagens através do uso de filas de mensagens bidirecionais.

4. LINGUAGENS DE PROGRAMAÇÃO

Atualmente existem diversos tipos de linguagens de programação, como por exemplo, as simbólicas, paralelas, orientadas a objeto, orientadas a eventos. Cada qual possui características específicas que as indicam para a resolução de problemas variados.

Neste trabalho decidiu-se mostrar apenas as paralelas por elas se encaixarem perfeitamente nos objetivos do mesmo. Além disso, elas são as mais indicadas para ambientes paralelos. Serão apresentadas as linguagens de programação paralela CSP, Occam, Joyce e SuperPascal. Elas utilizam o mecanismo de troca de mensagens para a comunicação entre os processos, que são gerados na forma de redes de processos. Estas linguagens foram escolhidas para estudo devido à semelhança entre as redes lógicas dos programas paralelos e as redes físicas dos multicomputadores.

Para ilustrar as características próprias de cada linguagem é utilizado um algoritmo único, que vai ser escrito e analisado em cada uma das linguagens apresentadas. Este algoritmo é denominado algoritmo de Miller-Rabin [HAN94], testa se um número é primo.

O algoritmo executa p testes probabilísticos do mesmo número inteiro simultaneamente através de p processos. Cada teste prova que o número é composto (não primo) ou falha sem provar nada. O algoritmo executa operações aritméticas em números naturais de comprimento variável representados por vetores de w posições (mais um dígito de *overflow*). Cada teste inicializa um gerador de números aleatórios com uma semente distinta. O processamento paralelo é organizado como uma rede em anel formada por um processo *master* e um *pipeline* de processos conectados através de canais de comunicação (Figura 4.1).

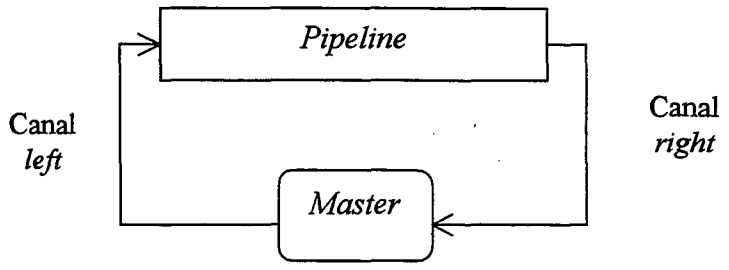

Figura 4.1: Rede de processo em anel.

O *pipeline* é composto por *p* processos chamados *nodes*, conectados por *p+1* canais de comunicação. O processo *master* envia um número através do *pipeline* e recebe *p* valores lógicos do *pipeline*, que são os resultados dos *p* testes realizados pelos *nodes*.

4.1 CSP

CSP (*Communicating Sequential Processes*) é uma linguagem baseada no modelo de troca de mensagens que foi proposto por Hoare em 1978 [HOA78]. Embora tenha influenciado o projeto de outras linguagens paralelas, CSP é essencialmente um projeto de linguagem sem implementação. As principais motivações para o seu desenvolvimento foram:

- Introduzir comandos básicos de entrada e saída na própria linguagem de programação para prover comunicação entre processos paralelos.
- Introduzir um comando paralelo para especificar execução paralela de processos seqüenciais.
- Prover comunicação síncrona direta entre os processos.
- Utilizar os comandos guardados de Dijkstra [DIJ75] associados aos comandos de entrada para introduzir e controlar o não-determinismo.

A seguir é mostrado o algoritmo de Miller-Rabin em CSP (Figura 4.2).

```
RING = [ pipeline :: PIPELINE || master :: MASTER ]

PIPELINE = [ Node( 1 ) || Node( i : 2..p-1 ) :: NODE || Node( p ) ]

Node( 1 ) = [ a : integer;
composite : boolean;

master ? a;
Node( 2 ) ! a;

....
```

```
...
executa procedimento test( a,i, composite )
Node( 2 ) ! composite;
 ]

NODE = [     a : integer;
Composite : boolean;

Node( i ) ? a;
[ i < p -> Node( i + 1 ) ! a ];

....
executa procedimento test( a,i, composite )
...

Node( i + 1) ! composite;
j : integer; j := 1;
* [ j <= i - 1 -> Node( i ) ? composite;
        Node( i + 1 ) ! composite;
        j := j + 1 ];
           ]

Node( p ) = [ a = integer;
                composite : boolean;

                Node( p ) ? a;
                Master ! a;

...
executa procedimento test( a,i, composite )
...

master ! composite;
j : integer; j := 1;
* [ j <= p - 1 -> Node( p ) ? composite;
        master ! composite;
        j :=  j + 1 ];
                ]

MASTER = [  a : integer;
composite : boolean;
...
a := valor a ser testado
...
Node( 1 ) ! a;
prime : boolean;
prime := True;
i := integer; i := 1;
* [ prime; i <= p -> Node( p ) ? composite;
```

```
        [ composite -> prime := false ];
]       i := i + 1;
        ]
```

Figura 4.2: Algoritmo de Miller-Rabin em CSP.

4.1.1 Paralelismo

Um processo consiste de um nome, variáveis locais e uma lista de comandos seqüenciais. Comandos simples são subdivididos em atribuição, entrada e saída. Comandos estruturados são subdivididos em paralelos, alternativos e repetitivos. Comandos alternativos e repetitivos utilizam os comandos guardados de Dijkstra associados aos comandos de entrada.

CSP utiliza o paralelismo explícito fornecendo um único comando paralelo (||) para criar um número fixo de processos paralelos. No algoritmo apresentado, tem-se os seguintes processos: **pipeline, master** e *p* processos **Node**. A notação **Node(i : 2..p-1) :: NODE** equivale a *p-2* processos **Node**.

A notação **expressão lógica; comandoDeEntrada -> listaDeComandos** representa um comando guardado. Comandos guardados são compostos por expressões lógicas e comandos de entrada, seguidos de uma lista de comandos.

A notação **[comandoGuardado □ comandoGuardado □ ...]** representa um comando alternativo. O término de um comando alternativo ocorre após a execução de um único comando guardado. A notação *** comandoAlternativo** representa um comando repetitivo. O comando repetitivo é responsável pela execução repetida do comando alternativo componente. O término de um comando repetitivo ocorre quando todas as guardas falham.

4.1.2 Comunicação

Um programa paralelo CSP contém uma coleção de processos que evoluem concorrentemente e se comunicam através de canais.

Os processos se comunicam através de comandos de entrada (?) e de saída (!). O processo emissor designa o processo receptor e fornece o valor a ser enviado. O processo receptor designa o processo emissor e fornece a variável para qual o valor vai ser atribuído. O mecanismo de comunicação utilizado é a comunicação síncrona direta e a associação dos processos com canais é feita de forma implícita.

Um mesmo canal pode transportar diferentes tipos de dados. No algoritmo mostrado, o processo **Node(1)** executa o comando de saída **Node(2) ! a** para enviar uma mensagem do tipo **integer** e o comando de saída **Node(2) ! composite** para enviar uma mensagem do tipo **boolean.**

Além da transferência de dados simples mostrados no algoritmo, os canais podem transportar dados estruturados utilizando construtores. No comando **Node(2) ! msg(a, composite)**, por exemplo, a mensagem transportada pelo canal contém dois tipos diferentes de dados associados pelo construtor **msg** : um **integer** e um **boolean**. Um construtor vazio como **x( )**, pode ser utilizado apenas para sincronizar dois processos sem transferir dados.

4.1.3 Ambiente de programação paralela

Como CSP é essencialmente um projeto de linguagem, não possui nenhum ambiente de programação efetivo.

4.1.4 Resumo das características de CSP

Pode-se comentar que CSP possui as seguintes limitações:

- Processos definidos estaticamente.
- Comunicação por troca de mensagens com designação direta simétrica.
- Falta de recursividade.

- Guardas de saída não são admitidas nos comandos guardados.

4.2 OCCAM

Occam é uma linguagem derivada de CSP, projetada para a programação do Transputer da Inmos, tendo sua primeira versão surgido em 1984 [INM84]. O Transputer possui grande capacidade de processamento e comunicação, fornecendo rápido chaveamento de processos e atendimento a interrupções. A relação do *hardware* com a linguagem é muito próxima, pois funções da linguagem possuem operadores correspondentes no *hardware*.

Occam tem aplicações para processamento de sinais, processamento de imagens, controle de processos, simulação, processamento em tempo real e análise numérica [BAL89]. Suas características estáticas fornecem alta eficiência para esses tipos de aplicações. Occam2 é uma extensão da linguagem Occam que surgiu em 1988 [BUR88].

A seguir, tem-se o exemplo do algoritmo de Miller-Rabin em Occam (Figura 4.3).

```
PROTOCOL channel IS BOOL, INT:
INT a :
BOOL prime :
CHAN OF channel, left, right :
PROC Node ( INT i, CHAN OF channel left, right )
        INT a, j :
        BOOL composite :
        SEQ
                Left ? a
                IF i < p
                  Right ! a
                TRUE
                  SKIP :
                ...
                executa procedimento test ( a, i, composite )
                ...
        right ! composite
        SEQ j = 1 FOR i –1
```

```
                Left ? composite
            SEQ Right ! composite
.
.

PROC pipeline ( CHAN OF channel left, right )
    [p+1] CHAN OF channel c :
    PAR
        Node( 1, left, c[ 1 ] )
        Node( p, c[ p ], right )
        PAR i = 2 FOR p-1
            Node( i, c[ i-1 ] , c[ i ] )
.
.

PROC master ( INT a, BOOL prime, CHAN OF channel left, right )
    INT i :
    BOOL composite :
    SEQ
        Left ! a
        Prime := TRUE
        SEQ i = 1 FOR p
            SEQ
                Right ? composite
                IF composite = TRUE
                    Prime := FALSE
                TRUE
                    SKIP
.
.

PAR
    Pipeline ( left, right )
    Master ( a, prime, left, right )
```

Figura 4.3: Algoritmo de Miller-Rabin em Occam.

4.2.1 Paralelismo

Toda instrução Occam é considerada um processo. Occam possui cinco processos primitivos : processos de atribuição (:=), de entrada (?), de saída (!), nulo (SKIP) e de parada (STOP). Os processos primitivos podem ser combinados para

expressar comportamentos mais complexos através de construtores da linguagem. Os construtores de Occam são : SEQ, PAR, WHILE, IF, CASE e ALT.

Compete ao programador indicar explicitamente se os processos serão combinados em seqüências através do construtor SEQ ou em paralelo através do construtor PAR.

É possível aplicar replicadores FOR aos construtores SEQ, PAR, ALT, IF com o objetivo de replicar o processo componente. Um exemplo de aplicação de replicador pode ser visto no algoritmo acima mostrado, no procedimento **pipeline**, para produzir *p-2* processos **Node**.

4.2.2 Comunicação

Um programa paralelo em Occam contém uma coleção de processos que evoluem concorrentemente e se comunicam através de canais.

O mecanismo de comunicação utilizado envolve trocas de mensagens síncronas indiretas. Esse mecanismo de comunicação é implementado no próprio *hardware* dos Transputers de forma altamente eficiente. Os canais lógicos de Occam podem ser associados aos canais físicos de comunicação dos Transputers.

A declaração **CHAN OF** descreve canais pelos quais trafegam apenas dados tipados. A declaração **PROTOCOL** descreve o formato das mensagens compostas por grupos de tipos de dados. No algoritmo, os canais **left** e **right** são canais de comunicação através dos quais são transferidas mensagens com protocolo simples composto por dois tipos de dados : lógico (**BOOL**) e numérico (**INT**).

4.2.3 Ambiente de programação paralela

O ambiente de programação paralela para Occam2, CSA *Transputer Education Kit* [CSA90], é um conjunto de ferramentas para possibilitar programação de Transputers compatíveis com o modelo T400. Ele permite que programas paralelos sejam desenvolvidos em máquinas hospedeiras para serem executados em um único Transputer ou em redes de Transputers.

A versão descrita do ambiente para a linguagem Occam2 executa sobre o sistema operacional DOS em um IBM-PC, e é composto por :

- Ambiente de produção de programas: esse ambiente é representado pelo compilador *occam*, ferramenta para a criação de bibliotecas de código chamada *ilibr*, ferramenta para a ligação de programas com bibliotecas e outros códigos compilados chamada *ilink*, ferramenta *iboot* que produz código executável e dados de carga para uma configuração específica de uma rede de Transputers. O editor utilizado para escrever programas em Occam pode ser qualquer editor disponível sobre o DOS;

- Ambiente de execução de programas: esse ambiente é representado pela ferramenta *iserver*, que carrega os programas nos Transputers usando o sistema de arquivos da máquina hospedeira e fornece suporte em tempo de execução para interface com a máquina hospedeira.

4.2.4 Resumo das características de Occam

A sintaxe de um programa em Occam possui algumas características especiais:

- Cada processo (primitivo, construtor e declaração) deve ocupar uma linha.

- A linguagem impõe indentação como forma de determinar o início e o fim de novos processos.

Pode-se comentar que Occam possui as seguintes limitações:

- Falta de recursividade para funções e procedimentos.

- Vetor de processos devem ter tamanhos constantes.

- Instruções simples devem ser escritas em linhas separadas.

- Não define mecanismos de entrada e saída como parte da linguagem.

- Não permite alocação dinâmica de variáveis.

4.3 JOYCE

Joyce é uma linguagem derivada de CSP com notação baseada em Pascal. Ela foi projetada por Brinch Hansen em 1987 [HAN87], para o ensino de princípios e técnicas de programação distribuída. A remoção de várias restrições do CSP também motivou o seu desenvolvimento. Para isso, foram introduzidos:

- Comunicação síncrona com designação indireta.
- Compartilhamento de canais.
- Variáveis do tipo porta (*port*) para acesso aos canais.
- Alfabeto de canal.
- Recursividade.
- Uso de instruções de saída nas guardas.

A seguir, tem-se o algoritmo de Miller-Rabin em Joyce (Figura 4.4).

```
Type number = array [ 0..w ] of integer;

Const p = ... número de processadores ...
Type channel = [log( boolean ), num( integer ) ];

Agent ring ( );
Var     a: number;
        Prime : boolean;
        Left, right : channel;

                Agent Node( i : integer; left, right : channel );
                Var     a : number;
                 j : integer;
                        Composite : boolean;
                Begin
                        Left ? num ( a );
                        If i < p then right ! num( a );
                        ...
executa procedimento test (a, i, composite )
...
```

```
j := 1;
while j <= i - 1 do
right ! log( composite );
begin
        left ? log( composite );
        right ! log ( composite );
        j := j  + 1;
end;
                end;

                Agent pipeline( left, right : channel );
                Type row = array[ 1..p-1 ] of channel;
                Var     c : row;
                        i : integer;
                Begin
                        i := 1;
                        While i < p do
                        Begin
                                +c[ i ];
                                i := i + 1;
                        end;
                        Node( 1, left, c[ 1 ]);
                        Node( p, c[ p - 1 ], right );
                        i := 2;
                        While i < p do
                        Begin
                                Node( i, c[ i - 1 ], c[ i ] );
                                i := i + 1;
                        End;
                End;

                Agent master( a : number; prime : boolean; left, right : channel );
                Var     i : integer;
                        Composite : boolean;
                Begin
                        Left ! num( a );
                        Prime := true;
                        i := 1;
                        While i <= p do
                        Begin
                                Right ? log( composite );
                                If composite then
                                        Prime := false;
                                i := i + 1;
                        End;
                End;

        Begin
                +left;
                +right;
```

```
        prime := true;
        pipeline( left, right );
        a := ... valor do número a ser testado ...
        master( a, prime, left, right );
end;
```

Figura 4.4: Algoritmo de Miller-Rabin em Joyce.

4.3.1 Paralelismo

Um programa Joyce consiste de procedimentos que definem processos paralelos conhecidos como agentes. A execução de um programa origina uma árvore hierárquica de processos criados e destruídos de forma dinâmica a partir de um processo inicial único, que é automaticamente criado quando o programa inicia. Os processos são criados com sintaxe semelhante a uma chamada de procedimento.

Observa-se no algoritmo descrito a existência do processo inicial **ring**, que cria dois processos filhos **pipeline** e **master**. O processo **pipeline**, por sua vez, cria p processos **node**. Tem-se, dessa forma, p+3 processos evoluindo em paralelo.

Como o número total de processos existentes é conhecido somente em tempo de execução, o mapeamento de processos em uma implementação do ambiente Joyce para uma máquina paralela deve ser feito de forma dinâmica.

4.3.2 Comunicação

Um programa paralelo Joyce contém uma coleção de processos que evoluem concorrentemente e se comunicam através de canais. O mecanismo de comunicação utilizado é a comunicação síncrona indireta.

Processos se comunicam através de símbolos transmitidos por canais. Cada canal possui um alfabeto que define um conjunto fixo de símbolos disjuntos que ele pode transportar. Um símbolo pode transportar uma mensagem de um tipo fixo. Tanto o alfabeto de símbolos quanto os tipos que estes podem transportar são conhecidos na definição do canal. Um canal é considerado um dispositivo de comunicação

compartilhado porque pode ser utilizado por dois ou mais processos, embora as comunicações ocorram entre dois processos de cada vez.

Para fazer referência ao canal, são utilizadas variáveis do tipo *port*. A definição **type Channel = [simb(integer)]**, por exemplo, define um alfabeto de canal formado pelo símbolo **simb**, que transporta uma mensagem do tipo **integer**. Um processo acessa um canal através de uma variável local do tipo *port*, como **c : Channel**. Quando um processo executa a instrução **+c**, um novo canal com o alfabeto definido pelo **Channel** é criado, e um ponteiro para o canal é atribuído à variável **c**.

Uma das estruturas de dados utilizadas para implementar canais é a fila de símbolos. Existe um par de filas para cada símbolo no alfabeto do canal: uma fila de processos emissores do símbolo (fila de saída) e uma fila de processos receptores do símbolo (fila de entrada). As filas servem para registrar pedidos de comunicação entre os processos que compartilham o canal. A presença de duas filas com pedidos de comunicação por parte dos processos possibilita o transporte bidirecional de mensagens através do canal, bem como o seu compartilhamento.

Dois processos devem executar instruções de entrada (**?**) e saída (**!**) para concretizar uma comunicação. No algoritmo descrito, o processo **master** executa a instrução de saída **left ! num(a)** para enviar a mensagem e o processo **node** executa a instrução **left ? num(a)** para receber a mensagem.

A instrução *poll* é semelhante a instrução alternativa de CSP. Entretanto, na linguagem Joyce, podem ser utilizadas tanto guardas de entrada como de saída.

Para ilustrar a utilidade dos canais compartilhados fez-se a seguinte alteração no algoritmo descrito anteriormente: canais compartilhados **left** e **right** são utilizados para realizar a tarefa de comunicação dos **nodes** com o processo **master**. Dessa forma, ao invés do valor da variável **composite**(p valores), que representa o resultado parcial feito por cada processo **node**, ser transportado através de um **pipeline** até chegar ao processo **master**, são utilizados os canais compartilhados para realizar esta tarefa. Economiza-se assim p-1 canais de comunicação.

A nova situação é representada na Figura 4.5 e no algoritmo mostrados a seguir (Figura 4.6):

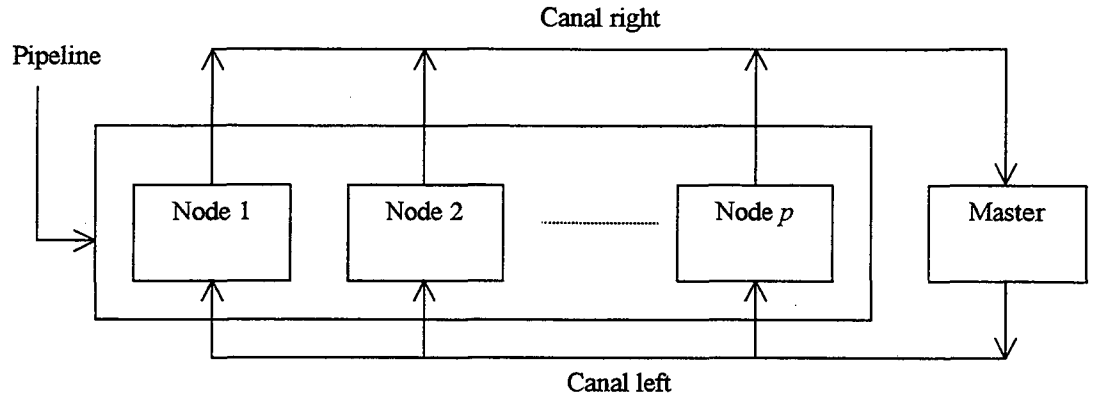

Figura 4.5: Rede de processos com canais compartilhados.

```
Type number = array[ 0..w] of integer;

Const p = ... número de processadores ...
Type channel = [ log( boolean ), num ( integer ) ];

Agent ring ( );
Var     a : number;
        Prime : boolean;
        Left, right : channel;

                Agent node( i : integer; left, right : channel );
                Var     a : number;
                        j : integer;
                        Composite : boolean;
                Begin
                        Left ? num( a );
                        ...
executa procedimento test (a, i, composite )
...
right ! log( composite );
                end;

                Agent pipeline( left, right : channel );
                Var     i : integer;
                Begin
                        i := 1;
                        While i <= p do
                        Begin
```

```
                    i := i + 1;
                End;  Node( i, left, right );
        End;

        Agent master( a : number; prime : boolean; left, right : channel );
        Var   i : integer;
              Composite : boolean;
        Begin
              Prime := true;
              i := 1;
              While i <= p do
              Begin
                    Left ! num( a );
                    Right ? log( composite );
                    If composite then
                          Prime := false;
                    i := i + 1;
              End;
        End;

Begin
      +left;
      +right;
      a := ... valor do número a ser testado ...
      prime := true;
      pipeline( left, right );
      master( a, prime, left, right );
End;
```

Figura 4.6: Algoritmo de Miller-Rabin em Joyce com canais compartilhados.

4.3.3 Ambiente de programação paralela

Os ambientes de programação paralela construídos para a linguagem Joyce são três: o primeiro ambiente para IBM-PC [HAN87b], o segundo para um hipercubo [AND89] e o terceiro para um multiprocessador [HAN89].

4.3.3.1 Ambiente para IBM-PC

O ambiente para IBM-PC é composto por:

- Ambiente de produção de programas: este ambiente é representado pelo compilador Joyce, que é um compilador de três passos escrito em Pascal. A compilação é dividida em análise léxica (*scanner*), análise sintática (*parser*) e a montagem de código objeto (*assembler*). Essas fases compõem um compilador que gera código portável interpretado por um núcleo Joyce escrito em linguagem Assembly.

- Ambiente de execução de programas: este ambiente é representado pelo núcleo Joyce, que implementa a gerência de comunicação, de memória e de processos.

Esse ambiente para um processador único foi implementado com idéias simples e eficientes para solidificar os conceitos da linguagem.

4.3.3.2 Ambiente para um multicomputador com rede de interconexão estática

O multicomputador utilizado para esse ambiente foi o hipercubo binário iPSC da Intel. Cada nó dessa máquina é conectado aos vizinhos através de canais de comunicação Ethernet. Existe um nó gerente (*cube manager*), que é um micro-computador Intel 80286, utilizado para fornecer entrada/saída e ambiente de produção de programas.

- Ambiente de produção de programas: esse ambiente é representado pelo compilador Joyce, que é um compilador de quatro passos escrito em C gerando código portável. O compilador é executado no gerente.

- Ambiente de execução de programas: esse ambiente é representado pelo núcleo Joyce, composto por um interpretador escrito em Assembly e por rotinas de comunicação e escalonamento escritas em C. Cópias do código portável são carregadas pelo núcleo em cada nó do hipercubo. Também existe o módulo de interface com o usuário, presente no nó gerente apenas quando um programa em Joyce está sendo executado. A interface representa a tarefa do sistema operacional sob o ponto de vista do programa.

Esse ambiente para um hipercubo mostrou-se ineficiente para a execução de programas Joyce, principalmente porque a criação de processos e a comunicação entre eles produz redes dinâmicas que se adaptam com dificuldade à topologia estática do multicomputador.

4.3.3.3 Ambiente para um multiprocessador

O multiprocessador utilizado para esse ambiente foi o Encore Multimax 320, que possui 18 processadores NS32332 e um barramento compartilhado que conecta os processadores com a memória compartilhada de 128 Mbytes.

- Ambiente de produção de programas: o compilador utilizado pelo multiprocessador é semelhante ao utilizado pelo ambiente IBM-PC quanto a sua forma de compilação. Existe a interface Unix Umax 4.2, uma versão Multimax para o sistema UNIX de Berkeley. Inicialmente, um usuário se comunica com um único processo Unix, chamado processo mestre. Quando o usuário decide executar um programa Joyce em p processadores, o processo mestre cria p processos Unix adicionais, conhecidos como processadores Joyce.

- Ambiente de execução de programas: esse ambiente é representado pelo núcleo para o multiprocessador. Esse núcleo, similar ao núcleo do IBM-PC, adiciona tarefas como equilíbrio de carga e *locks* (os *locks* são bloqueios para garantir a exclusão mútua no acesso às estruturas de dados compartilhados do núcleo). Esse ambiente para multiprocessador foi uma tentativa de eliminar os problemas encontrados na implementação do hipercubo.

4.3.4 Resumo das características de Joyce

Pode-se comentar que Joyce possui as seguintes limitações:

- Um processo não pode acessar variáveis globais.

- Uma mensagem não pode incluir referências a canais.

- Dois processos não podem se comunicar através do *polling* de um mesmo canal.

A primeira simplificação é a única considerada realmente importante [HAN93a].

4.4 SUPERPASCAL

SuperPascal é uma linguagem que estende um subconjunto do Pascal, com instruções determinísticas para a criação de processos paralelos e para a troca de mensagens síncronas.

As características de paralelismo são baseadas principalmente em Occam2, estendida pela inclusão de vetores de processos dinâmicos e processos paralelos recursivos.

O projeto da linguagem SuperPascal utiliza ainda características de linguagens como CSP e Joyce, e foi motivado por dois objetivos [HAN94c]:

- Simplicidade: criar uma linguagem de programação elegante para a computação científica paralela, adicionando uma quantidade mínima de conceitos à linguagem Pascal;

- Segurança: impor restrições adicionais aos conceitos de programação Pascal para permitir que um compilador verifique que processos paralelos são disjuntos, ou seja, que eles atualizam somente conjuntos disjuntos de variáveis.

A simplicidade foi atingida estendendo Pascal apenas com instruções para criação de processos e troca de mensagens síncronas para comunicação entre eles. A segurança foi garantida pela imposição de restrições adicionais nos procedimentos e funções, e omitindo algumas características do Pascal.

A seguir tem-se o exemplo do algoritmo de Miller-Rabin em SuperPascal (Figura 4.7).

```
Type number = array [ 0..w ] of integer;

Const p = ... número de processadores ...
Type channel = *( boolean, number );

Procedure master( a : number; var prime: boolean; left, right : channel );
Var     i : integer;
        Composite : boolean;

Begin
        Send( left, a );
        Prime := true;
        For i := 1 to p do
        Begin
                Receive( right, composite );
                If composite then
                        Prime := false;
        End;
End;

Procedure node( i : integer; left, right : channel );
Var     a : number;
        j : integer;
        Composite : boolean;

Begin
        Receive( left, a );
        If i < p then send( right, a );
        ...
        executa procedimento test ( a, i, composite )
        ...
        send( right, composite );
        for j := 1 to i – 1 do
        begin
                receive( left, composite );
                send( right, composite );
        end;
end;

procedure pipeline( left, right : channel );
type row = array [ 0..p ] of channel;
var     c : row;
        i : integer;

begin
        c[ 0 ] := left;
        c[ p ] := right;
        for i := 1 to p – 1 do
```

```
        forall i := 1 to p do
                node( c[ i - 1 ], c[ i ] );
end;

procedure ring( a: number; var prime : boolean );
var     left, right : channel;

begin
        open( left, right );
        parallel
                pipeline( left, right ) |
                master( a, prime, left, right );
        end;
end;
```

Figura 4.7: Algoritmo de Miller-Rabin em SuperPascal.

4.4.1 Paralelismo

As características de paralelismo da linguagem SuperPascal são representadas pelas instruções *parallel* e *forall* para a criação de processos paralelos. No algoritmo descrito, por exemplo, o procedimento **ring** utiliza a instrução *parallel* para explicitar que os procedimentos **pipeline** e **master** são executados em paralelo. O procedimento **pipeline** utiliza a instrução **forall** para explicitar que *p* processos **node** são executados em paralelo.

Existe criação dinâmica de processos através da recursividade e da possibilidade do número de processos de uma instrução *forall* ser definido em tempo de execução. Dessa forma, o mapeamento de processos, em uma implementação do ambiente SuperPascal para uma máquina paralela, deve ser feito de forma dinâmica.

4.4.2 Comunicação

Um programa SuperPascal contém uma coleção de processos que evoluem concorrentemente e se comunicam através de canais. O mecanismo de comunicação utilizado é a comunicação síncrona indireta.

Processos se comunicam através de valores chamados mensagens, transmitidos por meio de entidades chamadas canais. Processos criam canais dinamicamente e os acessam utilizando variáveis que fazem referência aos canais. Uma vez criados, os canais existem até o término do programa.

Um canal suporta mensagens de tipos diferentes, o que pode ser observado no algoritmo descrito, com a declaração **type channel = *(boolean, number)** indicando que um canal deste tipo pode transportar um valor de tipo **boolean** ou um valor de tipo **number**.

Os procedimentos necessários para a troca de mensagens são *open, send* e *receive*. *Open* é a instrução para criação de canais, *send* e *receive* são as instruções para o envio e recepção de mensagens, respectivamente.

Dois processos devem executar instruções de envio e recepção de mensagens para concretizar uma comunicação. No algoritmo descrito, o processo **master** executa a instrução de envio de mensagens **send(left, a)**, e o processo **node** executa a instrução **receive (left, a)** para a recepção dessa mensagem.

4.4.3 Ambiente de programação paralela

Foi construído um ambiente de programação paralela portável para a linguagem SuperPascal em uma estação de trabalho Sun [SUN98c] sobre o UNIX. Pode-se classificar o ambiente da seguinte forma:

- Ambiente de produção de programas: esse ambiente é representado pelo compilador SuperPascal chamado *spc*.
- Ambiente de execução de programas: esse ambiente é representado pelo interpretador *spi*, que é responsável pela execução de programas SuperPascal.

O ambiente SuperPascal é utilizado para desenvolver programas portáveis para problemas usuais na ciência da computação. Também é uma tentativa de simplificar a tarefa de programação para cientistas que, geralmente, estão mais preocupados com os

resultados numéricos do que com o aprendizado de programação em um ambiente de programação paralela, já que esse aprendizado costuma ser difícil [HAN94].

4.4.4 Resumo das limitações de SuperPascal

Pode-se comentar que SuperPascal possui as seguintes limitações:

- Parâmetros reais das chamadas de procedimento ou função e variáveis globais utilizadas no seu corpo não podem ser sinônimos.
- Procedimentos e funções recursivas não podem usar variáveis globais.
- Funções não podem atualizar variáveis globais e não podem utilizar parâmetros por referência ou procedimentos com parâmetros.
- Procedimentos e funções não podem usar procedimentos e funções como parâmetros.
- Declarações *forward* de procedimentos e funções não podem ser utilizadas.
- Tipo *pointer* é omitido.
- Instruções *goto* e *label* são omitidos.

4.5 CARACTERÍSTICAS DAS LINGUAGENS APRESENTADAS

Todas as linguagens de programação paralelas descritas são baseadas em troca de mensagens, sendo adequadas para aplicações em máquinas paralelas como multicomputadores.

CSP pode ser considerada o fundamento teórico de Occam, Joyce e SuperPascal no que diz respeito ao paralelismo e à comunicação síncrona.

Occam, por ser uma linguagem projetada especificamente para a programação de um tipo de *hardware* (Transputers), fornece alta eficiência na execução dos programas por possuir operadores em *hardware* para executar funções da linguagem de forma direta.

Joyce, mesmo sendo uma linguagem projetada para o ensino de programação distribuída, introduziu idéias novas como o compartilhamento de canais de comunicação síncronos.

SuperPascal, projetada para a elaboração de algoritmos científicos, é uma linguagem simples e segura, adequada para aplicações que seguem a disciplina produtor-consumidor.

4.5.1 Características do canais

Os canais de todas as linguagens descritas possuem capacidade zero, ou seja, utilizam o mecanismo de comunicação síncrona. Os canais de CSP, Joyce e SuperPascal possuem sentido bidirecional para o tráfego de mensagens enquanto os de Occam têm sentido unidirecional.

O acesso aos canais só é compartilhado na linguagem Joyce. Nas demais linguagens descritas eles são de acesso exclusivo dos processos por eles conectados.

O tamanho das mensagens é variável em todas as linguagens, ou seja, podem ser transferidas mensagens de tipos diferentes através do mesmo canal.

A designação é direta simétrica em CSP e indireta nas demais linguagens descritas.

A criação e destruição de canais é estática em CSP e Occam, e dinâmica em Joyce e SuperPascal. Os canais pertencem aos processos em todas as linguagens.

A seguir é mostrada uma tabela com o resumo das principais características dos canais de cada linguagem descrita (Figura 4.8).

Característica	**CSP**	**Occam**	**Joyce**	**SuperPascal**
Capacidade	Zero	Zero	Zero	Zero
Sentido	Bidirecional	Unidirecional	Bidirecional	Bidirecional
Acesso	Exclusivo	Exclusivo	Compartilhado	Exclusivo

Mensagens	Tamanho variável	Tamanho variável	Tamanho variável	Tamanho variável
Designação	Direta simétrica	Indireta	Indireta	Indireta
Criação e Destruição	Estática	Estática	Dinâmica em tempo de execução	Dinâmica em tempo de execução
Propriedade	Processos são proprietários	Processos são proprietários	Processos são proprietários	Processos são proprietários

Figura 4.8: Características dos canais das linguagens apresentadas.

5. O INTERPRETADOR SPI+

pítulo apresenta as alterações necessárias no interpretador SuperPascal para torná-lo paralelo, já que atualmente o interpretador é executado na forma seqüencial. Com isso o novo interpretador poderá utilizar de uma maneira mais eficiente os recursos das máquinas paralelas para a execução de programas paralelos.

5.1 APRESENTAÇÃO

O interpretador do SuperPascal, denominado **spi**, foi escrito na linguagem Pascal e originalmente implementado no sistema operacional SunOS da SUN MicroSystems [SUN98c]. Atualmente a sua principal restrição é não ter a característica que mais se procura quando se quer testar programas paralelos: o paralelismo.

O interpretador atual simula este paralelismo através de um escalonamento interno de processos, pelo qual são simuladas as trocas de contexto de execução entre os processos que são virtualmente criados. Hoje em dia todo programa escrito em SuperPascal e interpretado utilizando o **spi** é executado na forma seqüencial.

Outra característica bastante desejável em programas paralelos e que é simulada no interpretador SuperPascal é a troca de mensagens entre os processos. Como o **spi** não cria fisicamente os processos, não existe um mecanismo para a troca de mensagens entre os processos virtuais. A troca de mensagens ocorre através do uso de variáveis globais do interpretador.

Quando um processo executa a operação de envio de uma mensagem em SuperPascal através da função **send(canal, mensagem)**, ela é armazenada em variáveis globais até que um processo execute a operação de recebimento **receive(canal,mensagem)**. Neste momento, o conteúdo desta variável é lido pelo receptor e a troca de mensagens simulada é finalizada, liberando as variáveis globais para que outros processos possam realizar novas trocas de mensagens.

Este trabalho apresenta as alterações realizadas no interpretador SuperPascal para implementar a criação de processos paralelos e também a utilização de um

mecanismo de troca de mensagens para a comunicação entre os processos criados pelo **spi**.

Todas as modificações realizadas mantém uma compatibilidade com o interpretador atual, ou seja, um programa escrito em SuperPascal poderá executar tanto no interpretador original como no modificado sem alterações no código fonte.

É importante ressaltar que apesar do código fonte ser o mesmo, os resultados gerados pelos dois interpretadores podem ser diferentes devido às novas características adicionadas ao novo interpretador. Tais características serão explicadas no capítulo 5, quando for apresentado o novo interpretador.

A seguir será feita uma descrição dos elementos principais do funcionamento da linguagem SuperPascal: a máquina e o código.

5.2 MÁQUINA E O CÓDIGO SUPERPASCAL

O compilador SuperPascal gera código para uma máquina específica chamada **máquina SuperPascal**. O código executado por esta máquina é conhecido como **código SuperPascal**.

A máquina SuperPascal é uma máquina de pilha. Sua memória é constituída por uma seqüência de números inteiros. Os elementos da memória são conhecidos como **palavras** e os índices referentes a cada elemento são os **endereços**. As palavras armazenam o código de um programa SuperPascal com suas instruções e suas variáveis. O código, que tem tamanho fixo, é colocado no início da memória. O restante da memória é utilizada como pilha de variáveis. Durante a execução de instruções a pilha também armazena resultados temporários. A máquina possui quatro registradores de índice :

- registrador p: registrador do programa - contém o endereço da instrução atual;
- registrador b: registrador base - usado para acessar variáveis;
- registrador s: registrador de pilha - armazena o endereço do topo da pilha do processo;

- registrador t: registrador de início de memória livre - armazena o endereço da última posição de memória ocupada.

A representação da memória de uma máquina SuperPascal aparece na Figura 5.1.

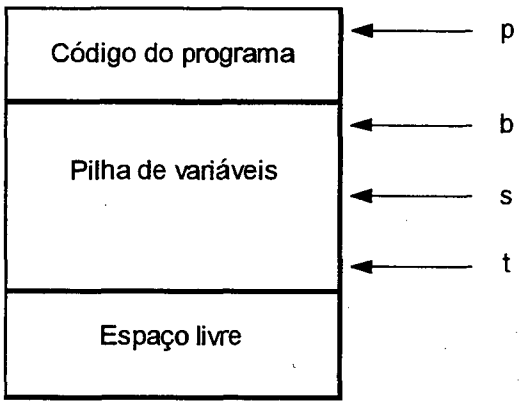

Figura 5.1: Memória da máquina SuperPascal.

A máquina SuperPascal é ideal para a linguagem SuperPascal porque :

- os operadores do código SuperPascal correspondem diretamente aos conceitos da linguagem;
- código SuperPascal de um programa tem praticamente a mesma sintaxe do próprio programa fonte. Consequentemente, o gerador de código do compilador pode ser uma extensão do analisador sintático;
- uma máquina SuperPascal implementada em *hardware* poderia executar programas SuperPascal com maior velocidade do que a maioria dos computadores tradicionais.

Como não se dispõe de uma máquina SuperPascal implementada em *hardware*, ela pode ser simulada através de um **interpretador SuperPascal**. O ambiente de programação é formado por dois componentes independentes: um ambiente de produção de programas (o compilador) e um ambiente de execução (o interpretador).

Uma desvantagem da interpretação do código é a baixa velocidade de execução. Mas a vantagem é a maior **portabilidade**: o *software* pode ser transportado de um computador para outro reescrevendo o interpretador para a máquina destino.

5.2.1 O código SuperPascal

Existem 111 instruções na máquina SuperPascal, cada qual composta por um código de operação e um número variável de operandos. Tanto os códigos de operação quanto os operandos são representados por números inteiros. Assim, o código produzido pelo compilador SuperPascal é constituído por uma seqüência de números inteiros. Os códigos de operação podem ser observados na Figura 5.2.

Operador	Código	Operador	Código	Operador	Código	Operador	Código
abs2	0	exp2	30	notequal2	60	value2	90
absint2	1	field2	31	odd2	61	variable2	91
add2	2	float2	32	open2	62	varparam2	92
addreal2	3	floatleft2	33	or2	63	write2	93
and2	4	for2	34	ordconst2	64	writebool2	94
arctan2	5	forall2	35	parallel2	65	writeint2	95
assign2	6	goto2	36	pred2	66	writeln2	96
assume2	7	grord2	37	proccall2	67	writereal2	97
case2	8	grreal2	38	procedure2	68	writestring2	98
checkio2	9	grstring2	39	process2	69	globalcall2	99
chr2	10	index2	40	program2	70	globalvalue2	100
cos2	11	ln2	41	read2	71	globalvar2	101
divide2	12	lsord2	42	readint2	72	localreal2	102
divreal2	13	lsreal2	43	readln2	73	localvalue2	103
do2	14	lsstring2	44	readreal2	74	localvar2	104
downto2	15	minus2	45	realconst2	75	ordassign2	105
endall2	16	minusreal2	46	receive2	76	ordvalue2	106
enddown2	17	modulo2	47	result2	77	realassign2	107
endio2	18	multiply2	48	round2	78	realvalue2	108
endparallel2	19	multreal2	49	send2	79	defaddr2	109
endproc2	20	neord2	50	sin2	80	defarg2	110
endprocces2	21	nereal2	51	sqr2	81		
endprog2	22	nestring2	52	sqrint2	82		
eof2	24	ngreal2	54	stringconst2	84		
eoln2	25	ngstring2	55	subreal2	85		
eqord2	26	nlord2	56	subtract2	86		
eqreal2	27	nlreal2	57	succ2	87		
eqstring2	28	nlstring2	58	to2	88	**minoperation**	**0**
equal2	29	not2	59	trunc2	89	**maxoperation**	**110**

Figura 5.2: Códigos de operação SuperPascal.

Por exemplo, o código gerado para o algoritmo simples da Figura 5.3 é apresentado na Figura 5.4. Substituindo os códigos de operações numéricos da Figura 5.2 pelos mnemônicos encontrados na Figura 5.3, colocando os argumentos entre parênteses e numerando a seqüência de execução do programa, o código SuperPascal fica mais legível (Figura 5.4) :

```
                    program Simples;
                { ----------------------------------
                  Um produtor e um consumidor.
                ----------------------------------}
const
        k = 9;

type
        tipoCanal = *(integer);

var
        canal : tipoCanal;

procedure produtor;
begin
        send(canal, k);
end;


procedure consumidor;
var
        msg : integer;

begin
        receive(canal,msg);
end;


begin
      open(canal);
      parallel
       produtor |
       consumidor
      end
end.
```

Figura 5.3: Texto de um programa simples.

```
70/1 1 5 44 1          1. program2(1, 1, 5, 44, 1)
 68 2 0 0 7 7 15       15. procedure2(2, 0, 0, 7, 7, 15)
100 4                  16. globalvalue2(4)
 9 16                  17. checkio2(16)
 64 9                  18. ordconst2(9)
 79 1 1 16             19. send2(1, 1, 16)
 18                    20. endio2
 20                    21. endproc2
 68 3 0 1 7 7 20       9.  procedure2(3, 0, 1, 7, 7, 20)
100 4                  10. globalvalue2(4)
 9 23                  11. checkio2(23)
104 4                  12. localvar2(4)
 76 1 1 23             13. receive2(1, 1, 23)
 18                    23. endio2
 20                    24. endproc2
104 4                  2.  localvar2(4)
 62 27                 3.  open2(27)
 65                    4.  parallel2
 69 4 4 10 29          5.  process2(4, 4, 10, 29)
 99 -48                14. globalcall2(-48)
 21 15 29              22. endprocess2(15, 29)
 69 5 5 10 30          6.  process2(5, 5, 10, 30)
 99 -39                8.  globalcall2(-39)
 21 5 31               25. endprocess2(5, 31)
 19 32                 7.  endparallel2(32)
 22                    26. endprog2
```

Figura 5.4: Código para o texto do programa da Figura 5.3.

5.3 SUPERPASCAL NO SOLARIS

O ambiente de programação paralela do SuperPascal no sistema operacional Solaris [SUN98b] é composto de dois módulos: **spc** e **spi.** O programa **spc** é responsável pela compilação e geração do código-objeto, que posteriormente servirá como entrada para o interpretador. O **spi** recebe um arquivo gerado pelo **spc** como entrada, que contém o código-objeto do programa paralelo que será interpretado. O **spi** é responsável pela interação com o usuário, mostrando os resultados obtidos ou solicitando a entrada de dados necessária para a execução do programa paralelo.

Os programas **spc** e **spi** foram escritos na linguagem Pascal [SUN98a] para o sistema operacional SunOS da Sun, mas são compatíveis também com o Solaris, necessitando apenas de uma recompilação de seus códigos para o funcionamento.

5.4 DESCRIÇÃO DOS MECANISMOS DE DISTRIBUIÇÃO DE PROCESSOS E TROCA DE MENSAGENS

Os mecanismos criados para o gerenciamento de processos e para a troca de mensagens são bastante simples e seguem a filosofia de implementação de Brinch Hansen [HAN98] (criador da linguagem SuperPascal), que é a de definir os procedimentos da forma mais simples e clara possível.

Todas as funções criadas para a implementação dos mecanismos acima descritos estão armazenados em uma biblioteca de funções escrita em linguagem C. Esta biblioteca, por sua vez, possui uma interface de chamadas, a qual é utilizada dentro do interpretador para fazer uso de suas funções. A criação de um módulo adicional de funções escritas em C surgiu pelo fato das funções de gerência de processos (criação, execução e remoção) e de troca de mensagens (mecanismo IPC) não possuírem uma interface de acesso para a linguagem Pascal, tendo assim que ser chamadas através da linguagem C.

Em vista das dificuldades acima citadas optou-se por deixar todo o código não-Pascal em uma biblioteca de funções separada, que no caso denomina-se *libsp.c*. Para a geração do novo interpretador, ao qual denominamos **spi+**, esta biblioteca é adicionada ao executável junto com o código Pascal do interpretador alterado.

A seguir serão apresentadas as funções criadas e que foram divididas em duas partes: na primeira parte são mostradas as funções que referem-se ao gerenciamento de processos e na segunda parte as funções associadas a troca de mensagens. Será descrito também o processo **master**, o qual é responsável pela manutenção dos canais de comunicação no **spi+**. Para a melhor compreensão da funcionalidade do processo **master** o mesmo será apresentado utilizando um exemplo de comunicação entre processos.

5.4.1 Processos

As funções utilizadas para o gerenciamento de processos no **spi+**, e que estão contidas na biblioteca *libsp.c*, são as seguintes:

- forksp: este procedimento é responsável pela criação dos processos, sendo que cada chamada resulta na criação de um processo. A criação dos processos é feita através da chamada *fork()* do sistema operacional UNIX.

- exitsp: procedimento responsável pela finalização da execução dos processos, sendo que o mesmo é chamado quando os processos terminaram as suas tarefas e desejam finalizar a sua execução. A destruição do processo é feito através da chamada *exit()* do sistema operacional UNIX.

- waitsp: procedimento responsável pela chamada da função *wait()* do sistema operacional. O processo principal do interpretador (**spi+**) é responsável pela criação de todos os processos paralelos que serão utilizados na execução do programa. Após a criação dos processos, o interpretador fica aguardando o término da execução de todos os seus filhos através de chamadas consecutivas da função waitsp. Quando todos os processos terminaram a sua execução (através da chamada da função exitsp), o processo principal termina a execução do laço de chamadas waitsp e finaliza a execução do interpretador com um todo.

5.4.2 Comunicação

As funções responsáveis pela comunicação entre processos no **spi+**, e que estão contidas na biblioteca *libsp.c*, são as seguintes:

- inittable: esta função é chamada pelo interpretador durante seu processo de inicialização para alocar uma tabela de canais. Esta tabela posteriormente será utilizada para indicar os processos que estão se comunicando, e os canais que os mesmos estão utilizando. Após a inicialização da tabela de canais ocorre a chamada da função createqueues.

- createqueues: esta função cria duas filas, usando o mecanismo de mensagens IPC do System V descrito no capítulo 3.2. Estas filas serão utilizadas para a troca de mensagens entre os processos. A primeira fila é utilizada para a comunicação entre os processos e o **master** (descrito posteriormente),e a outra é utilizada para a comunicação entre os processos. Estas filas são conhecidas apenas pelas funções da biblioteca *libsp.c* responsáveis pela troca de mensagens, não sendo acessíveis pelas demais funções do interpretador.

- freequeues: chamada responsável pela destruição das duas filas criadas no início da execução do programa SuperPascal. Sua chamada ocorre quando o interpretador vai finalizar a sua execução.

- opensp: função responsável pela alocação de um canal na tabela de canais. Este canal será utilizado para futuras comunicações entre o processo que o criou e um outro processo, que será definido durante a execução do programa paralelo. Em SuperPascal um canal é de uso exclusivo de dois processos, e uma vez criado permanece ativo até o fim da execução do programa.

- sendsp : tem a finalidade de enviar uma mensagem para um processo através de um canal especificado. O processo que executa esta chamada fica bloqueado até que o processo receptor execute a função de recebimento de mensagens (receivesp). Esta função faz uso das filas criadas na chamada createqueues para realizar o envio de mensagens para os processos.

- receivesp: é utilizada por processos para a recepção de mensagens através de um canal. Caso exista um processo realizando a execução de uma chamada sendsp, os dois processos são sincronizados e é executada a troca de mensagens. Não havendo mensagens para recepção, o processo fica bloqueado até que existam mensagens no canal para serem recebidas. As mensagens a serem recebidas ficam armazenadas nas filas criadas na função createqueues.

5.4.3 Processo Master

O processo **master** é criado no momento da execução de um programa paralelo, tendo como função o gerenciamento dos canais de comunicação. A solicitação da criação de canais pelos processos é feita ao **master**, que aloca um canal dentro da sua tabela de canais, e devolve o identificador do canal para o processo que solicitou a criação do mesmo. Abaixo é descrito um exemplo da execução de um programa paralelo, ilustrando a função do processo **master** dentro da execução dos programas no **spi+**.

O exemplo consiste em dois processos que desejam trocar informações entre si em um programa paralelo. Para que isto ocorra o seguinte roteiro deve ser seguido:

O processo que deseja enviar a mensagem (no exemplo, P1) solicita a execução da função **send**, passando como parâmetros o identificador do canal e a mensagem a ser enviada para o processo P2. Internamente o **spi+** faz o mapeamento desta chamada para a execução da função **sendsp**, com os mesmos parâmetros.

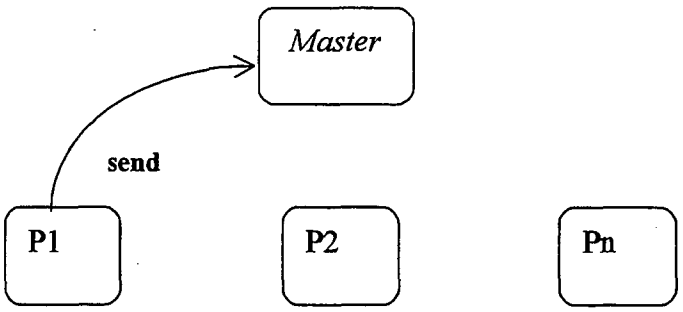

Figura 5.5 Solicitação ao master de envio de mensagem.

A função **sendsp** verifica a existência de um canal físico de comunicação entre P1 e P2. Como os dois processos nunca realizaram uma comunicação antes, tal canal não existe. Neste ponto o processo **master** entra em ação para gerenciar a alocação deste canal físico .

O **master** recebe a solicitação de que P1 deseja enviar uma mensagem para P2 (Figura 5.5) através da função **sendsp**, e faz uma alocação prévia do canal a ser utilizado. Porém a liberação do mesmo para a comunicação só ocorre quando o processo P2 manifestar a sua intenção de se comunicar com P1, ou seja, quando executar a

função **receive** (Figura 5.6). Isto é necessário pois não pode ocorrer a liberação de um canal sem que estejam estabelecidos o emissor e o receptor do canal.

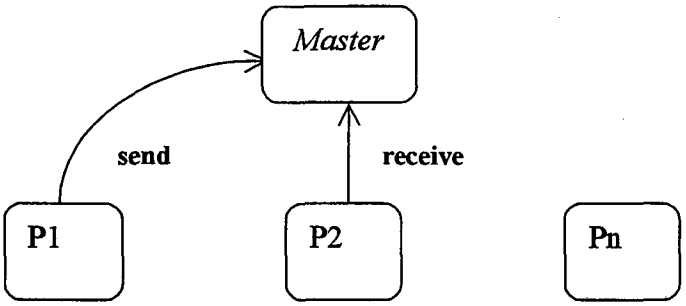

Figura 5.6: Solicitação ao master de recepção de mensagem.

A partir do momento em que P1 e P2 estão sincronizados o canal é liberado para a comunicação. Esta primeira comunicação gera um *overhead* maior que as demais pois envolve a alocação do canal. A partir da segunda comunicação entre os processos, o **master** não é mais acionado e a comunicação flui entre os dois processos de forma direta (Figura 5.7).

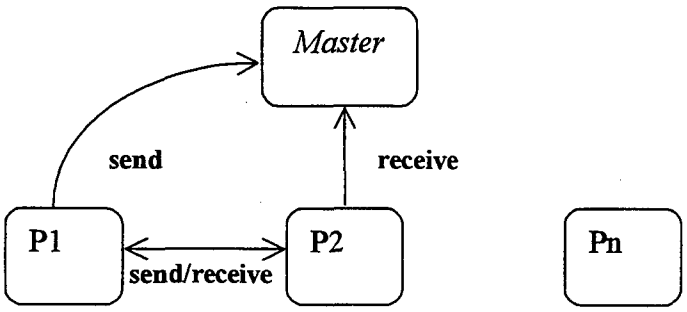

Figura 5.7: Comunicação direta entre os processos.

A estrutura de comunicação com a divisão entre canais lógicos, alocados através da função **opensp**, e canais físicos, alocados pelo **master**, foi adotada pelo **spi+** para contemplar a arquitetura da máquina CRUX, apresentada no capítulo 2.5.

Com a adoção da estrutura de comunicação entre processos do CRUX, consegue-se verificar a viabilidade da implementação do interpretador SuperPascal na topologia do CRUX. Realizando-se uma analogia com a máquina, o processo **master** corresponde ao nó de controle, os demais processos correspondem aos nós de trabalho, e os canais de comunicação correspondem aos canais físicos do CRUX. Caso estas

alterações surtam os efeitos desejados e o novo interpretador consiga um desempenho melhor que o anterior, o seu porte para a máquina CRUX será algo natural e com ótimas chances de uma melhora ainda maior na sua performance.

Sobre as filas criadas para a troca de mensagens entre os processos pode-se fazer algumas considerações que ajudarão a compreender como funciona internamente o mecanismo de troca de mensagens:

- As mensagens enviadas ou recebidas no IPC possuem a característica de serem tipadas, ou seja, a cada mensagem pode-se associar um tipo que pode ser utilizado para fazer a seleção de mensagens.

- No momento em que uma mensagem é enviada para um processo, a função **sendsp** coloca a mensagem na fila e define como sendo o tipo da mensagem o identificador do canal para o qual a mensagem foi enviada.

- Quando um processo deseja realizar uma recepção de mensagem através da função **receivesp**, é verificado na fila de mensagem se há alguma mensagem cujo tipo é igual ao canal que o processo deseja receber a mensagem. Se houver, a mensagem é retirada da fila e entregue ao processo. Caso contrário, o processo fica bloqueado até que exista alguma mensagem do tipo especificado na fila.

5.5 ALTERAÇÕES NO INTERPRETADOR SUPERPASCAL

Além da biblioteca de funções criada para adicionar as funcionalidades de gerenciamento de processos e troca de mensagens, tornou-se necessária a alteração de algumas partes do interpretador original, para que as funções criadas possam ser utilizadas.

A fim de não alterar a estrutura global do interpretador, já que o objetivo deste trabalho não é o de criar um novo interpretador, mas sim o de realizar alterações no seu comportamento, as alterações realizadas ficaram localizadas nas partes que envolvem a manipulação de processos e a troca de mensagens.

As funções do interpretador que foram alteradas são as seguintes:

- Parallel2: esta instrução é gerada quando encontra-se no código do programa SuperPascal uma chamada **parallel**. Esta instrução geralmente é utilizada para explicitar a criação de processos paralelos. A função foi alterada para salvar o contexto do interpretador antes da criação de um novo processo. Desta maneira, após o término da execução dos processos filhos o interpretador continuará sua execução do ponto onde havia parado anteriormente.

- EndParallel2: este procedimento executa a função **forksp** para cada processo da instrução **parallel**. Após isto, permanece aguardando o término dos processos filhos executando chamadas da função **waitsp** para poder prosseguir a sua execução.

- EndProcess2: foi alterado para realizar a chamada da função *exitsp*, a qual realiza a destruição do processo que a executou .

- ForAll2: esta instrução é gerada quando encontra-se no código do programa SuperPascal uma chamada **forall**. Esta instrução geralmente é utilizada para criar **n** processos paralelos em um laço. O procedimento foi alterado para armazenar o contexto atual dos registradores do interpretador e executar a função **forksp** para cada processo da instrução **forall**. Após isto, o processo permanece esperando o término da execução de seus filhos executando chamadas **waitsp** para poder retornar a sua execução normal.

- EndAll2: foi modificada para finalizar a execução do processo através de uma chamada da função **exitsp**. Esta função é chamada pelos processos criados através do comando **forall**. Os processos criados pela instrução **parallel** executam a chamada **EndProcess2**.

- Open2: instrução gerada quando encontra-se no código do programa SuperPascal uma chamada **open**. Ela é utilizada para a alocação de um canal de comunicação. Nesta função foi realizada uma alteração para gerar uma chamada ao processo **master**, solicitando que seja alocada uma entrada na tabela de canais.

- Send2: esta instrução é gerada quando encontra-se no código do programa SuperPascal uma chamada **send**. É utilizada para a o envio de uma mensagem, e foi alterada para executar a chamada da função **sendsp**.

- Receive2: esta instrução é gerada quando encontra-se no código do programa SuperPascal uma chamada **receive**. É utilizada para a recepção de uma mensagem que após a alteração realiza a chamada da função **receivesp**.

- EndProg2: esta chamada é realizada quando o interpretador está finalizando a sua execução. Nesta função foi adicionado o envio de uma mensagem ao processo **master**, solicitando que ele finalize a sua execução e libere a memória, que está sendo utilizada com as estruturas de controle e as filas (através da chamada *freequeues*) que estão sendo utilizadas para a troca de mensagens.

- Start: procedimento de inicialização do interpretador. Foi alterado para realizar a chamada da função *inittable*, para que sejam inicializadas as filas de mensagens e estruturas de controle utilizadas pelos processos do programa paralelo.

6. CONCLUSÃO

A alteração do interpretador, apesar de ter sido feita em partes específicas, influenciou profundamente o modo de execução dos programas. Inicialmente, não podia-se ter certeza de que os objetivos idealizados seriam alcançados. Porém, com o novo interpretador foram feitos vários testes que acabaram por validar os objetivos esperados, que estão descritos abaixo:

- Criação de Processos Paralelos: Foi substituída com sucesso a simulação de processos paralelos pela criação dos mesmos. Esta característica é fundamental para que se possa testar um programa paralelo. Anteriormente, com a execução seqüencial, o programa era executado sempre da mesma maneira, independentemente de fatores externos. Com as modificações efetuadas, programas que eram considerados corretos usando o interpretador seqüencial agora podem apresentar erros utilizando o spi+. Isso ocorre porque foram agregadas novas variáveis ao processo de execução, como por exemplo o escalonamento de processos e a possível utilização de vários processadores simultaneamente. Problemas como *deadlocks* podem ser detectados em programas considerados corretos pelo interpretador original.

- Comunicação Real: O mecanismo de comunicação mostrou-se eficiente na comunicação entre os processos, mesmo sendo considerado historicamente o gargalo na execução de programas paralelos. As funções tiveram implementações simples, o que tornou o processo de troca de mensagens rápido e eficiente.

- Ferramenta de Validação: Espera-se que o novo interpretador seja uma ferramenta mais útil quando se desejar testar características de paralelismo e de troca de mensagens de programas escritos na linguagem SuperPascal.

- Compatibilidade: Os programas criados para rodarem com o interpretador original executam também no interpretador alterado, sem a necessidade de alterações no código ou no processo de compilação. Isto faz com que o uso para o usuário torne-se mais amigável e atrativo. O usuário praticamente pode dispor de uma nova ferramenta para execução, já que foram alteradas

as principais partes do interpretador, sem a necessidade de recompilação de programas ou qualquer outra tarefa adicional.

- Desempenho: Brinch Hansen, em seu artigo que descreve a linguagem SuperPascal [HAN94], traz vários programas-exemplo. Tais programas foram utilizados para realizar um estudo comparativo entre o interpretador original e o alterado Apesar da máquina utilizada para testes possuir apenas 1 processador, constatou-se uma melhora de performance no tempo de execução dos programas paralelos que realizam operações de entrada e saída Os programas com poucas operaçoes de entrada e saída não tiveram uma melhoria na performance utilizando apenas 1 processador. Porém com a utilização de uma máquina com vários processadores a performace deve aumentar também para os processo com poucas operações de entrada e saída e um alto grau de processamento de informações. Alguns resultados podem ser vistos na Figura 6.1.

Programa	**spi (tempo de execução)**	**spi+ (tempo de execução)**
prime.p (*)	1967 ms	650 ms
string.p (+)	516 ms	583 ms
ring.p (*)	517 ms	482 ms
prodcons.p (+)	500 ms	583 ms
prodcons2.p(+)	523 ms	567 ms

Figura 6.1: Resultados obitdos com o spi e o spi+

Legenda:
(*) : Programas com muitas operações de entrada e saída.
(+) : Programas com poucas operações de entrada e saída.

Apesar dos resultados obtidos terem sido os esperados este trabalho teve algumas dificuldades para se tornar realidade. Com a necessidade de se usar o mecanismo de comunicação IPC, tornou-se necessária a criação de uma biblioteca de funções escrita em C, já que Pascal não possui uma interface de chamadas para o IPC.

Outra dificuldade encontrada foi a de deixar transparente ao usuário final as modificações realizadas internamente no interpretador, pois o objetivo do projeto era realizar as modificações sem um impacto no uso do interpretador. As modificações serão sentidas apenas na execução dos programas e nos resultados obtidos.

Como continuidade deste trabalho pode-se aplicar modificações semelhantes no processo de compilação dos programas, alterando o **spc**. As fases de compilação são bem distintas e podem ser executadas em paralelo por diversos processos que se comunicam através de mensagens.

REFERÊNCIAS

[ACC96] ACCETTA, M. et al., *Mach: A New Kernel Foundation for Unix Development*, Proceedings of the Summer 1986 USENIX Conference, p. 93-112, 1986.

[AND89] ANDERSEN, B., *Hypercube Experiments with Joyce*. Sigplan Notices, Vol. 24, n. 8, p. 13-22, 1989.

[BAL89] BAL, H. E., Steiner, J. G., Tanenbaum, A. S., *Programming languages for distributed computing systems.* ACM Computing Surveys, Vol. 21, p. 261-322, 1989.

[BHU87] LAXMI, N., *Interconection networks for parallel and distributed processing.* Computer, v. 20, p-9-12,1987.

[BUR88] BURNS, Alan, *Programming in Occam2.* Addison-Wesley, 1988.

[COR93] CORSO, T. B., *Ambiente para programação paralela em multicomputador.* Relatório técnico, UFSC-CTC-INE, Florianópolis, 1993.

[CSR86] COMPUTER SYSTEMS RESEARCH GROUP, *BSD UNIX Reference Manuals,* University of California at Berkeley, 1986.

[DIJ75] DIJKSTRA, E.W.,*Guarded Commands, nondeterminancy and formal derivation of programs.* Communications of ACM Vol. 18, p. 453-457, 1975.

[FAU97] FAUSTO, L. F. et al., *Store-and-Forward versus Wormhole Routing Mechanisms: A comparative Performance Evaluation in Multicomputer Networks,* Proc. Computer Summer Simulation Conference, Arlington, Virginia, USA, 1997.

[GUI82] GUILLEMONT, M,. *The Chorus Distributed Operating System: Design and Implementation,* Proceedings ACM International Synposium on Local Computer Networks, p.207-223, Firenze, 1992.

[HAN87] HANSEN, P. B., *Joyce – A Programming Language for Distributed Systems.* Software Practice and Experience, vol 17(1), p. 29-50, 1987.

[HAN87b] HANSEN, P. B., *A Joyce Implementation.* Software – Practice and Experience. Vol. 17(4), p. 267-276, 1987.

[HAN89] HANSEN, P. B., *A Multiprocessor implementation of Joyce.* Software – Practice and Experience, Vol 19(6), p. 579-592, 1989.

[HAN93a] HANSEN, P. B., *Monitors and concurrent Pascal: A personal history..* School of Computer and Information Science, Syracuse University, Syracuse, NY, 1993.

[HAN94] HANSEN, P. B., *SuperPdscal – a publication language for parallel scientific computing*, Concurrency – Practice and Experience, vol. 6(5), p. 461-483, Agosto de 1994.

[HAN94b] HANSEN, P. B., *Interference control in SuperPascal – a block-structured parallel language.* Computer Journal, vol. 37(5), p. 399-406, 1994.

[HAN98] HANSEN, P. B., *Home Page* - Disponível online: http://web.syr.edu/~pbhansen/

[HOA78] HOARE, C. A. R., *Communicating Sequential Processes.* Communications of ACM, vol. 21, n. 8, p. 666-677, agosto de 1978.

[HWA93] HWANG, K., *Advanced Computer Architecture: Parallelism, Scalability, Programmability.*McGraw Hill, 1993.

[INM84] INMOS Limited. *Occam Programming Manual.* Prentice-Hall, 1984.

[LEF89] LEFFLER, S. J. et al., *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, MA, 1989.

[SUN98a] *Sun Pascal 4.0 Answer Book-* Disponível Online: http://docs.sun.com:80/ab2/@Ab1CollToc?abcardcat=%2Fsafedir%2Fsp

ace4%2Fpkgs%2Faddoldversion%2Fab1%2FSUNWspro%2FSPROabpas%2Fab_cardcatalog;subject=languages

[SUN98b] *Sun Solaris Documentation* – Disponível Online : http://docs.sun.com:80/ab2/@Ab1CollToc?abcardcat=/safedir/space4/pkgs/addoldversion/ab1/SUNWabe/ab_cardcatalog

[SUN98c] *Sun Microsystems* – Disponível Online : http://www.sun.com

[SKI95] SKILLICORN, David B., *Programming Languages for Parallel Processing*. IEEE Computer Society Press, Los Alamitos, California, 1995.

[TAN92a] TANEMBAUM, A., *A Introduction to Amoeba,* Amsterdam: Vrije Universiteit, 1992 (Relatório Técnico).

[TAN92b] TANEMBAUM, A., *Using Sparse Capabilities in a Distributed Operating System*, Amsterdam: Vrije Universiteit, 1992 (Relatório Técnico).

ANEXO A

Neste apêndice é listado o código fonte da biblioteca *libsp.c*, a qual é utilizada pelo novo interpretador (spi+).

```
/*

Nonpascal functions used in the SuperPascal Interpreter

By: Luis Fernando Fausto - fausto@inf.ufsc.br
Last Update: 06/15/1998

*/


#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <strings.h>

/*

  New types added

*/

#define SEND          1
#define RECEIVE       2
#define TERMINATE    -1
#define MASTER        1L
#define CONNECTED     1
#define DISCONNECTED  0
#define OPEN          2
#define CONTENTION4  -2
#define MAXCHAN5     -3
#define OK            3
#define TYPE4        -2

typedef struct { long pids;
                 long pidr;
                 int cstatus;
               } tchentry;

typedef struct { long  to;
                 long  from;
                 int   chid;
                 int   typeno;
                 char  value[1000];
               } tmessage;
```

```
union itoc
{
  int i;

  struct
  {
    char c1, c2, c3, c4;
  } c;
};

/*

  C global variables

*/

int      mqid;           /* id of message queue of the master process
*/
int      pqid;           /* id of message queue of the others process
*/
tchentry chtable[128];   /* logical channels table */



/*

Debug only - Show current status of message queue

*/

void showstatus()
{
  struct msqid_ds buf;

  msgctl(mqid, IPC_STAT, &buf);
  msgctl(pqid, IPC_STAT, &buf);
}

/*

Initialize the logical channels table and the pairs table

*/

void inittable()
{
  int k;

  createqueues();
  for (k=0; k< 127; k++)
  {
    chtable[k].pids = 0;
    chtable[k].pidr = 0;
    chtable[k].cstatus = DISCONNECTED;
  }
}

/*

Destroys the two queues used by the SuperPascal interpreter
```

```
*/

void freequeues()
{
  msgctl(mqid, IPC_RMID, (struct msqid_ds *)0);
  msgctl(pqid, IPC_RMID, (struct msqid_ds *)0);
}


/*

Master process used to start the channel connections

*/

void master()
{
  tmessage message;
  int      k;
  do
  {
    msgrcv(mqid, &message, sizeof(message), MASTER , 0);
    if (message.from != TERMINATE)
    {
      switch (message.value[0]) {
       case SEND : {
        if ((chtable[message.chid].pids == 0) ||
(chtable[message.chid].pids == message.from))
        {
          chtable[message.chid].pids = message.from;
          if (chtable[message.chid].pidr != 0)
          {
            chtable[message.chid].cstatus = CONNECTED;
            message.value[0] = CONNECTED;
            message.to = message.from;
            message.from = MASTER;
            msgsnd(mqid, &message, sizeof(message), 0);
            message.to = chtable[message.chid].pidr;
            msgsnd(mqid, &message, sizeof(message), 0);
          };
        } else
        {
          message.value[0] = CONTENTION4;
          message.to = message.from;
          message.from = MASTER;
          msgsnd(mqid, &message, sizeof(message), 0);
        }
        break;
       }
       case RECEIVE : {
        if ((chtable[message.chid].pidr == 0) ||
(chtable[message.chid].pidr == message.from))
        {
          chtable[message.chid].pidr = message.from;
          if (chtable[message.chid].pids != 0)
          {
            chtable[message.chid].cstatus = CONNECTED;
            message.value[0] = CONNECTED;
            message.to = message.from;
            message.from = MASTER;
```

```
            message.to = chtable[message.chid].pids;
            msgsnd(mqid, &message, sizeof(message), 0);
          } msgsnd(mqid, &message, sizeof(message), 0);
        } else
        {
          message.value[0] = CONTENTION4;
          message.to = message.from;
          message.from = MASTER;
          msgsnd(mqid, &message, sizeof(message), 0);
        }
       }
      }
    }
  }while (message.from != TERMINATE);
  freequeues();
  exit(0);
}

/*

 Allocates a logical channel for communication

*/

int opensp()
{
 int k=0;

 while ((chtable[k].cstatus != DISCONNECTED) && (k < 127))
   k++;
 if (chtable[k].cstatus != DISCONNECTED)
  return MAXCHAN5;
 else
 {
  chtable[k].cstatus = OPEN;
  return k;
 }
}



/*

Creates the two message queues used by the SuperPascal interpreter

*/

int createqueues()
{
  if ((mqid = msgget(IPC_PRIVATE, IPC_CREAT | 0660)) == -1)
  {
    perror("msgget");
    return 1;
  }
  if ((pqid = msgget(IPC_PRIVATE, IPC_CREAT | 0660)) == -1)
  {
    perror("msgget");
    return 1;
  }
  return 0;
```

```
}*

System call fork

*/

int forksp()
{
  return fork();
}

/*

System call exit

*/

void exitsp(int status)
{
  exit(status);
}

/*

System call wait

*/

int waitsp(int status)
{
 return  wait(&status);
}

/*
Send a message to a channel
*/

int sendsp(int chid, int msg[], int typeno, unsigned int len)
{
  tmessage message;
  long mypid;
  int k, j;
  int bytes;
  union itoc i2c;

  if (chid == TERMINATE)
  {
    message.to = MASTER;
    message.from = TERMINATE;
    message.value[0] = 0;
    bytes = msgsnd(mqid, &message, sizeof(message), 0);
    if (bytes < 0)
       perror("Send Error");
    return;
  }
  if (chtable[chid].cstatus == CONNECTED)
  {
    message.to     = chid+1;
```

```
    message.typeno = typeno;
    j = 0;
    message.from = getpid();
    for (k=0;k<len;k++)
    {
      i2c.i = msg[k];
      message.value[j]   = i2c.c.c1;
      message.value[j+1]  = i2c.c.c2;
      message.value[j+2]  = i2c.c.c3;
      message.value[j+3]  = i2c.c.c4;
      j = j  + 4;
    }
    msgsnd(pqid, &message, sizeof(message), 0);
    msgrcv(pqid, &message, sizeof(message), getpid(), 0);
  }
  else
  {
    mypid = getpid();
    message.from = mypid;
    message.to = MASTER;
    message.chid = chid;
    message.value[0] = SEND;
    msgsnd(mqid, &message, sizeof(message), 0);
    msgrcv(mqid, &message, sizeof(message), mypid, 0);
    if ((message.value[0] == CONTENTION4) || (message.value[0] ==
MAXCHAN5))
    {
      return message.value[0];
    }
    else
    {
      chtable[message.chid].cstatus = CONNECTED;
      message.to     =  chid+1;
      message.from   = mypid;
      message.typeno = typeno;
      j = 0;
      for (k=0;k<len;k++)
      {
        i2c.i = msg[k];
        message.value[j]   = i2c.c.c1;
        message.value[j+1]  = i2c.c.c2;
        message.value[j+2]  = i2c.c.c3;
        message.value[j+3]  = i2c.c.c4;
        j = j  + 4;
      }
      if (msgsnd(pqid, &message, sizeof(message), 0) != -1)
      {
        msgrcv(pqid, &message, sizeof(message), mypid, 0);
      }
      else
        perror("SEND: Error sending message.\n");
    }
  }
}


/*
Receive a message from a channel
*/
                    ; errada
int receivesp(int chid, int msg[], int typeno, unsigned int len)
```

```
  tmessage message;
  long mypid;
{ int k, j;
  union itoc i2c;

  if (chtable[chid].cstatus == CONNECTED)
  {
    msgrcv(pqid, &message, sizeof(message), chid+1, 0);
    if (message.typeno != typeno)
    {
      typeno = TYPE4;
    }
    j = 0;
    for (k=0;k<len;k++)
    {
     i2c.c.c1 = message.value[j];
     i2c.c.c2 = message.value[j+1];
     i2c.c.c3 = message.value[j+2];
     i2c.c.c4 = message.value[j+3];
     msg[k] = i2c.i;
     j = j + 4;
    }
    message.to = message.from;
    message.value[0] = OK;
    msgsnd(pqid, &message, sizeof(message), 0);
  }
  else
  {
    mypid = getpid();
    message.from = mypid;
    message.to = MASTER;
    message.chid = chid;
    message.value[0] = RECEIVE;
    msgsnd(mqid, &message, sizeof(message), 0);
    msgrcv(mqid, &message, sizeof(message), mypid, 0);
   if ((message.value[0] == CONTENTION4) || (message.value[0] ==
MAXCHAN5))
    {
      return message.value[0];
    }
    else
    {
      chtable[message.chid].cstatus = CONNECTED;
      if (msgrcv(pqid, &message, sizeof(message), chid+1, 0) == 0)
        perror("RECEIVE: Error");
      j = 0;
      if (message.typeno != typeno)
      {
        typeno = TYPE4;
      }
      for (k=0;k<len;k++)
      {
       i2c.c.c1 = message.value[j];
       i2c.c.c2 = message.value[j+1];
       i2c.c.c3 = message.value[j+2];
       i2c.c.c4 = message.value[j+3];
       msg[k] = i2c.i;
       j = j + 4;
      }
      message.to = message.from;
```

```
            msgsnd(pqid, &message, sizeof(message), 0);
        }
    }   message.value[0] = OK;
    return typeno;
}
```

ANEXO B

Neste apêndice é listado o código fonte do interpretador da linguagem SuperPascal com as modificações realizadas, as quais estão indicadas no código através de comentários.

```
{        SUPERPASCAL INTERPRETER
            20 August 1993
 Copyright (c) 1993 Per Brinch Hansen

 Modified by: Luis Fernando Fausto fausto@inf.ufsc.br
 Last Update: 06/12/1998
}


program interpret(input, output);

#include "common.p"

{begin of modifications}


C procedure inittable;                external c;
  procedure master;                   external c;
C function  forksp: integer;          external c;
C procedure waitsp(status : integer); external c;
  procedure showstatus;               external c;

  {end of modifications}

  procedure run(
    var codefile: binary;
    var inpfile, outfile: text);
  const minaddr = 1;
  type
    store =
      array [minaddr..maxaddr] of
        integer;
    blocktable =
      array [1..maxblock] of integer;

  {
   External communication functions
  }
C function opensp : integer; external c;
C function receivesp(chid : integer; var msg : store;
                     typeno : integer; len : integer): integer; external c;
C function sendsp(chid : integer; msg : store;
                     typeno : integer; len : integer) : integer; external
  c;
```

```
procedure exitsp(status : integer); external c;

procedure waitsp(status : integer); external c;
var
  { permanent variables }
  b, cmax, p, ready, s,
  stackbottom, t: integer;
  running: boolean;
  st: store;
  free: blocktable;
  { temporary variables }
  bi, blockno, c, i,
  j, k, length, level,
  lineno, lower, m, n,
  si, templength,
  typeno, typeno2, upper,
  width, x, y: integer;
  dx, dy: dualreal;
  sx: string;
  cx: char;


{ begin of modifications (fausto) }
  message : store;
  bendparallel2,
  sendparallel2,
  tendparallel2 : integer;
  bendforall2,
  sendforall2,
  tendforall2 : integer;
  returnaddress : integer;
  nprocsparallel,
  nprocsforall : integer;
  status : integer;
{ end of modifications (fausto) }

{ Local procedures in run }

procedure error(lineno: integer;
  kind: phrase);
begin
  writeln('line ', lineno:4, sp,
    kind:phraselength(kind));
  running := false
end;

procedure rangeerror(
  lineno: integer);
begin
  error(lineno, range4)
end;

procedure memorylimit(
  lineno: integer);
begin
  error(lineno, maxaddr5)
end;

procedure load(
  var codefile: binary);
var i: integer;
```

```
  i := minaddr;
  while not eof(codefile)
begin and (i < maxaddr) do
      begin
        read(codefile, st[i]);
        i := i + 1
      end;
  if not eof(codefile)
    then memorylimit(1)
    else stackbottom := i
end;

procedure activate(
  bvalue, svalue, pvalue:
  integer);
begin
  svalue := svalue + 3;
  st[svalue - 2] := pvalue;
  st[svalue - 1] := bvalue;
  st[svalue] := ready;
  ready := svalue
end;

procedure select(
  lineno: integer);
begin
  if ready = 0 then
    error(lineno, deadlock4)
  else
    begin
      s := ready;
      ready := st[s];
      b := st[s - 1];
      p := st[s - 2];
      s := s - 3
    end
end;

procedure popstring(
  var value: string);
var i: integer;
begin
  s := s - maxstring;
  for i := 1 to maxstring do
    value[i] := chr(st[s + i])
end;

begin
  load(codefile);
  p := minaddr;
  running := true;
  while running do
    case st[p] of

    { VariableAccess =
        VariableName
        [ ComponentSelector ]* .
      VariableName =
        "variable" | "varparam" .
      ComponentSelector =
```

```
    "field" . }

  variable2{level, displ, Expression"index"}:
    begin
      level := st[p + 1];
      s := s + 1;
      x := b;
      while level > 0 do
        begin
          x := st[x];
          level := level - 1
        end;
      st[s] := x + st[p + 2];
      p := p + 3
    end;

  varparam2{level, displ}:
    begin
      level := st[p + 1];
      s := s + 1;
      x := b;
      while level > 0 do
        begin
          x := st[x];
          level := level - 1
        end;
      st[s] := st[x + st[p + 2]];
      p := p + 3
    end;

  index2{lower, upper, length,
      lineno}:
    begin
      lower := st[p + 1];
      i := st[s];
      s := s - 1;
      if (i < lower) or
        (i > st[p + 2]) then
          rangeerror(st[p + 4])
      else
        begin
          st[s] := st[s] +
            (i - lower) *
              st[p + 3];
           p := p + 5
        end
    end;

  field2{displ}:
    begin
      st[s] :=
        st[s] + st[p + 1];
      p := p + 2
    end;

{ StandardFunctionDesignator =
    FileFunctionDesignator |
    MathFunctionDesignator .
  FileFunctionDesignator =
    "eol" | "eoln" . }
```

```
    eof2{lineno}:
      begin
        s := s + 1;
        st[s] :=
          ord(eof(inpfile));
        p := p + 2
      end;

    eoln2{lineno}:
      begin
        s := s + 1;
        st[s] :=
          ord(eoln(inpfile));
        p := p + 2
      end;

  { MathFunctionDesignator =
      Expression [ "float" ]
        MathFunctionIdentifier .
    MathFunctionIdentifier =
      Abs | "arctan" | "chr" |
      "cos" | "exp" | "ln" |
      "odd" | "pred" | "round" |
      "sin" | Sqr | "sqrt" |
      "succ" | "trunc" .
    Abs =
     "abs" | "absint" .
    Sqr =
     "sqr" | "sqrint" . }

    float2:
      begin
        dx.a := st[s];
        s := s + 1;
        st[s - 1] := dx.b;
        st[s] := dx.c;
        p := p + 1
      end;

    abs2{lineno}:
      begin
        dx.b := st[s - 1];
        dx.c := st[s];
        dx.a := abs(dx.a);
        st[s - 1] := dx.b;
        st[s] := dx.c;
      { if overflow then
          rangeerror(st[p + 1])
        else } p := p + 2
      end;

    absint2{lineno}:
      begin
        st[s] := abs(st[s]);
      { if overflow then
          rangeerror(st[p + 1])
        else } p := p + 2
      end;

    arctan2{lineno}:
```

```
    dx.b := st[s - 1];
    dx.c := st[s];
  begin
    dx.a := arctan(dx.a);
    st[s - 1] := dx.b;
    st[s] := dx.c;
  { if overflow then
      rangeerror(st[p + 1])
    else } p := p + 2
  end;

chr2{lineno}:
  begin
    x := st[s];
    if (x < null) or (x > del)
      then
        rangeerror(st[p + 1])
      else p := p + 2
  end;

cos2{lineno}:
  begin
    dx.b := st[s - 1];
    dx.c := st[s];
    dx.a := cos(dx.a);
    st[s - 1] := dx.b;
    st[s] := dx.c;
  { if overflow then
      rangeerror(st[p + 1])
    else } p := p + 2
  end;

exp2{lineno}:
  begin
    dx.b := st[s - 1];
    dx.c := st[s];
    dx.a := exp(dx.a);
    st[s - 1] := dx.b;
    st[s] := dx.c;
  { if overflow then
       rangeerror(st[p + 1])
    else } p := p + 2
  end;

ln2{lineno}:
  begin
    dx.b := st[s - 1];
    dx.c := st[s];
    dx.a := ln(dx.a);
    st[s - 1] := dx.b;
    st[s] := dx.c;
  { if overflow then
      rangeerror(st[p + 1])
    else } p := p + 2
  end;

odd2:
  begin
    st[s] := ord(odd(st[s]));
    p := p + 1
  end;
```

```
pred2{minvalue, lineno}:
  begin
    if st[s] > st[p + 1] then
      begin
        st[s] := pred(st[s]);
        p := p + 3
      end
    else
      rangeerror(st[p + 2])
  end;

round2{lineno}:
  begin
    dx.b := st[s - 1];
    dx.c := st[s];
    s := s - 1;
    st[s] := round(dx.a);
  { if overflow then
      rangeerror(st[p + 1])
    else } p := p + 2
  end;

sin2{lineno}:
  begin
    dx.b := st[s - 1];
    dx.c := st[s];
    dx.a := sin(dx.a);
    st[s - 1] := dx.b;
    st[s] := dx.c;
  { if overflow then
      rangeerror(st[p + 1])
    else } p := p + 2
  end;

sqr2{lineno}:
  begin
    dx.b := st[s - 1];
    dx.c := st[s];
    dx.a := sqr(dx.a);
    st[s - 1] := dx.b;
    st[s] := dx.c;
  { if overflow then
      rangeerror(st[p + 1])
    else } p := p + 2
  end;

sqrint2{lineno}:
  begin
    st[s] := sqr(st[s]);
  { if overflow then
      rangeerror(st[p + 1])
    else } p := p + 2
  end;

sqrt2{lineno}:
  begin
    dx.b := st[s - 1];
    dx.c := st[s];
    dx.a := sqrt(dx.a);
    st[s - 1] := dx.b;
```

```
      { if overflow then
          rangeerror(st[p + 1])
        else } p := p + 2
      end;

    succ2{maxvalue, lineno}:
      begin
        if st[s] < st[p + 1] then
          begin
            st[s] := succ(st[s]);
            p := p + 3
          end
        else
          rangeerror(st[p + 2])
      end;

    trunc2{lineno}:
      begin
        dx.b := st[s - 1];
        dx.c := st[s];
        s := s - 1;
        st[s] := trunc(dx.a);
      { if overflow then
          rangeerror(st[p + 1])
        else } p := p + 2
      end;

  { FunctionDesignator =
     "result"
       ActualParameterPart
         "proccall" |
    StandardFunctionDesignator .
    ActualParameterPart =
      [ ActualParameter ]* .
    ActualParameter =
      Expression [ "float" ] |
      VariableAccess . }

    result2{length}:
      begin
        s := s + st[p + 1];
        p := p + 2
      end;

    proccall2{level, displ}:
      begin
        level := st[p + 1];
        s := s + 1;
        x := b;
        while level > 0 do
          begin
            x := st[x];
            level := level - 1
          end;
        st[s] := x;
        st[s + 2] := p + 3;
        p := p + st[p + 2]
      end;

  { ConstantFactor =
```

```
      "strconst" . }

    "ordconst" | "realconst" |
    ordconst2{value}:
      begin
        s := s + 1;
        st[s] := st[p + 1];
        p := p + 2
      end;

    realconst2{value}:
      begin
        st[s + 1] := st[p + 1];
        st[s + 2] := st[p + 2];
        s := s + 2;
        p := p + 3
      end;

    stringconst2{length, value}:
      begin
        length := st[p + 1];
        for i := 1 to length do
          st[s + i] :=
            st[p + i + 1];
        for i := length + 1
          to maxstring do
            st[s + i] := null;
        s := s + maxstring;
        p := p + length + 2
      end;

  { Factor =
      ConstantFactor |
      VariableAccess "value" |
      FunctionDesignator |
      Expression |
      Factor [ "not" ] . }

    value2{length}:
      begin
        length := st[p + 1];
        x := st[s];
        for i := 0 to
          length - 1 do
            st[s + i] :=
              st[x + i];
        s := s + length - 1;
        p := p + 2
      end;

    not2:
      begin
        if st[s] = ord(true)
          then
            st[s] := ord(false)
          else
            st[s] := ord(true);
        p := p + 1
      end;

  { Term =
```

```
    MultiplyingOperator ]* .
Float =
  Factor [ Factor [ Float ]
MultiplyingOperator =
  Multiply | Divide |
  "modulo" | "and" .
Multiply =
  "multiply" | "multreal" .
Divide =
  "divide" | "divreal" . }

floatleft2:
  begin
    dx.a := st[s - 2];
    s := s + 1;
    st[s] := st[s - 1];
    st[s - 1] := st[s - 2];
    st[s - 3] := dx.b;
    st[s - 2] := dx.c;
    p := p + 1
  end;

multiply2{lineno}:
  begin
    s := s - 1;
    st[s] :=
      st[s] * st[s + 1];
  { if overflow then
      rangeerror(st[p + 1])
    else } p := p + 2
  end;

divide2{lineno}:
  begin
    s := s - 1;
    st[s] :=
      st[s] div st[s + 1];
  { if overflow then
      rangeerror(st[p + 1])
    else } p := p + 2
  end;

modulo2{lineno}:
  begin
    s := s - 1;
    st[s] :=
      st[s] mod st[s + 1];
  { if overflow then
      rangeerror(st[p + 1])
    else } p := p + 2
  end;

multreal2{lineno}:
  begin
    dy.b := st[s - 1];
    dy.c := st[s];
    s := s - 2;
    dx.b := st[s - 1];
    dx.c := st[s];
    dx.a := dx.a * dy.a;
```

```
      st[s] := dx.c;
    { if overflow then
      st[s - 1] := dx.b;
        rangeerror(st[p + 1])
      else } p := p + 2
  end;

  divreal2{lineno}:
    begin
      dy.b := st[s - 1];
      dy.c := st[s];
      s := s - 2;
      dx.b := st[s - 1];
      dx.c := st[s];
      dx.a := dx.a / dy.a;
      st[s - 1] := dx.b;
      st[s] := dx.c;
    { if overflow then
        rangeerror(st[p + 1])
      else } p := p + 2
    end;

  and2:
    begin
      s := s - 1;
      if st[s] = ord(true)
        then
          st[s] := st[s + 1];
      p := p + 1
    end;

{ SimpleExpression =
    Term [ Sign ]
      [ Term [ Float ]
        AddingOperator ]* .
  Sign =
    Empty | Minus .
  Minus =
    "minus" | "minusreal" .
  AddingOperator =
    Add | Subtract | "or" .
  Add =
    "add" | "addreal" .
  Subtract =
    "subtract" |
    "subreal" . }

  minus2{(lineno}:
    begin
      st[s] := - st[s];
    { if overflow then
        rangeerror(st[p + 1])
      else } p := p + 2
    end;

  minusreal2{lineno}:
    begin
      dx.b := st[s - 1];
      dx.c := st[s];
      dx.a := - dx.a;
      st[s - 1] := dx.b;
```

```
    { if overflow then
        rangeerror(st[p + 1])
      else } p := p + 2
    end;

  add2{lineno}:
    begin
      s := s - 1;
      st[s] :=
        st[s] + st[s + 1];
    { if overflow then
        rangeerror(st[p + 1])
      else } p := p + 2
    end;

  subtract2{lineno}:
    begin
      s := s - 1;
      st[s] :=
        st[s] - st[s + 1];
    { if overflow then
        rangeerror(st[p + 1])
      else } p := p + 2
    end;

  addreal2{lineno}:
    begin
      dy.b := st[s - 1];
      dy.c := st[s];
      s := s - 2;
      dx.b := st[s - 1];
      dx.c := st[s];
      dx.a := dx.a + dy.a;
      st[s - 1] := dx.b;
      st[s] := dx.c;
    { if overflow then
        rangeerror(st[p + 1])
      else } p := p + 2
    end;

  subreal2{lineno}:
    begin
      dy.b := st[s - 1];
      dy.c := st[s];
      s := s - 2;
      dx.b := st[s - 1];
      dx.c := st[s];
      dx.a := dx.a - dy.a;
      st[s - 1] := dx.b;
      st[s] := dx.c;
    { if overflow then
        rangeerror(st[p + 1])
      else } p := p + 2
    end;

  or2:
    begin
      s := s - 1;
      if st[s] = ord(false) then
        st[s] := st[s + 1];
```

```
    end;

{ Expression = SimpleExpression
    [ SimpleExpression [ Float ]
      RelationalOperator ] .
  RelationalOperator =
    Less | Equal | Greater |
    NotGreater | NotEqual |
    NotLess.
  Less =
    "lsord" | "lsreal" |
    "lsstring" .
  Equal =
    "eqord" | "eqreal" |
    "eqstrig" | "equal" .
  Greater =
    "grord" | "grreal" |
    "grstring" .
  NotGreater =
    "ngord" | "ngreal" |
    "ngstring" .
  NotEqual =
    "neord" | "nereal" |
    "nestring" | "notequal" .
  NotLess =
    "nlord" | "nlreal" |
    "nlstring" . }

  lsord2:
    begin
      s := s - 1;
      st[s] :=
        ord(st[s] < st[s + 1]);
      p := p + 1
    end;

  eqord2:
    begin
      s := s - 1;
      st[s] :=
        ord(st[s] = st[s + 1]);
      p := p + 1
    end;

  grord2:
    begin
      s := s - 1;
      st[s] :=
        ord(st[s] > st[s + 1]);
      p := p + 1
    end;

  ngord2:
    begin
      s := s - 1;
      st[s] :=
        ord(st[s] <= st[s + 1]);
      p := p + 1
    end;
```

```
  begin
    s := s - 1;
neord2:
    st[s] :=
      ord(st[s] <> st[s + 1]);
    p := p + 1
  end;

nlord2:
  begin
    s := s - 1;
    st[s] :=
      ord(st[s] >= st[s + 1]);
    p := p + 1
  end;

lsreal2:
  begin
    dy.b := st[s - 1];
    dy.c := st[s];
    s := s - 3;
    dx.b := st[s];
    dx.c := st[s + 1];
    st[s] := ord(dx.a < dy.a);
    p := p + 1
  end;

eqreal2:
  begin
    dy.b := st[s - 1];
    dy.c := st[s];
    s := s - 3;
    dx.b := st[s];
    dx.c := st[s + 1];
    st[s] := ord(dx.a = dy.a);
    p := p + 1
  end;

grreal2:
  begin
    dy.b := st[s - 1];
    dy.c := st[s];
    s := s - 3;
    dx.b := st[s];
    dx.c := st[s + 1];
    st[s] := ord(dx.a > dy.a);
    p := p + 1
  end;

ngreal2:
  begin
    dy.b := st[s - 1];
    dy.c := st[s];
    s := s - 3;
    dx.b := st[s];
    dx.c := st[s + 1];
    st[s] := ord(dx.a <= dy.a);
    p := p + 1
  end;

nereal2:
```

```
      dy.b := st[s - 1];
      dy.c := st[s];
    begin
      s := s - 3;
      dx.b := st[s];
      dx.c := st[s + 1];
      st[s] := ord(dx.a <> dy.a);
      p := p + 1
    end;

  nlreal2:
    begin
      dy.b := st[s - 1];
*b10W dy.c := st[s];
      s := s - 3;
      dx.b := st[s];
      dx.c := st[s + 1];
      st[s] := ord(dx.a >= dy.a);
      p := p + 1
    end;

  lsstring2:
    begin
      y := s - maxstring + 1;
      s := y - maxstring;
      i := 0;
      while (i < maxstring - 1)
        and
          (st[s + i] = st[y + i])
            do i := i + 1;
      st[s] := ord(st[s + i] <
        st[y + i]);
      p := p + 1
    end;

  eqstring2:
    begin
      y := s - maxstring + 1;
      s := y - maxstring;
      i := 0;
      while (i < maxstring - 1)
        and
          (st[s + i] = st[y + i])
            do i := i + 1;
      st[s] := ord(st[s + i] =
        st[y + i]);
      p := p + 1
    end;

  grstring2:
    begin
      y := s - maxstring + 1;
      s := y - maxstring;
      i := 0;
      while (i < maxstring - 1)
        and
          (st[s + i] = st[y + i])
            do i := i + 1;
      st[s] := ord(st[s + i] >
        st[y + i]);
      p := p + 1
```

```
ngstring2:
 end;
    begin
      y := s - maxstring + 1;
      s := y - maxstring;
      i := 0;
      while (i < maxstring - 1)
        and
          (st[s + i] = st[y + i])
            do i := i + 1;
      st[s] := ord(st[s + i] <=
        st[y + i]);
      p := p + 1
    end;

nestring2:
  begin
    y := s - maxstring + 1;
    s := y - maxstring;
    i := 0;
    while (i < maxstring - 1)
      and
        (st[s + i] = st[y + i])
          do i := i + 1;
    st[s] := ord(st[s + i] <>
      st[y + i]);
    p := p + 1
  end;

nlstring2:
  begin
    y := s - maxstring + 1;
    s := y - maxstring;
    i := 0;
    while (i < maxstring - 1)
      and
        (st[s + i] = st[y + i])
          do i := i + 1;
    st[s] := ord(st[s + i] >=
      st[y + i]);
    p := p + 1
  end;

equal2{length}:
  begin
    length := st[p + 1];
    y := s - length + 1;
    s := y - length;
    i := 0;
    while (i < length - 1) and
      (st[s + i] = st[y + i])
        do i := i + 1;
    st[s] := ord(st[s + i] =
      st[y + i]);
    p := p + 2
  end;

notequal2{length}:
  begin
    length := st[p + 1];
```

```
      s := y - length;
      i := 0;
      while (i <= length - 1) and
       (st[s + i] = st[y + i])
         do i := i + 1;
      st[s] := ord(st[s + i] <>
        st[y + i]);
       p := p + 2
  end;

{ AssignmentStatement =
    VariableAccess Expression
      [ "float" ] "assign" . }

  assign2{length}:
    begin
      length := st[p + 1];
      s := s - length - 1;
      x := st[s + 1];
      y := s + 2;
      for i := 0 to
        length - 1 do
          st[x + i] :=
            st[y + i];
      p := p + 2
    end;

{ ReadStatement =
    ReadParameters |
    [ ReadParameters ]
      "readln" .
  ReadParameters =
    ReadParameter
      [ ReadParameter ]* .
    ReadParameter =
      VariableAccess Read .
    Read =
      "read" | "readint" |
      "readreal" . }

  read2{lineno}:
    begin
      read(inpfile, cx);
      st[st[s]] := ord(cx);
      s := s - 1;
      p := p + 2
    end;

  readint2{lineno}:
    begin
      read(inpfile,
        st[st[s]]);
      s := s - 1;
      p := p + 2
    end;

  readreal2{lineno}:
    begin
      read(inpfile, dx.a);
      y := st[s];
```

```
      st[y] := dx.b;
      st[y + 1] := dx.c;
      p := p + 2;
    end;

  readln2{lineno}:
    begin
      readln(inpfile);
      p := p + 2
    end;

{ WriteStatement =
    WriteParameters |
    [ WriteParameters ]
      "writeln" .
  WriteParameters =
    WriteParameter
      [ WriteParameter ]* .
  WriteParameter =
    Expression
      [ TotalWidth
        [ FracDigits ] ]
          "writereal" |
    Expression
      [ TotalWidth ]
        OtherWrite .
  OtherWrite =
    "write" | "writebool" |
    "writeint" |
    "writestring" .
  TotalWidth =
    Expression .
  FracDigits =
    Expression . }

  write2{option, lineno}:
    begin
      if st[p + 1] = ord(true)
        then
          begin
            write(outfile,
              chr(st[s - 1]):
                st[s]);
           s := s - 2
          end
        else
          begin
            write(outfile,
              chr(st[s]));
            s := s - 1
          end;
      p := p + 3
    end;

  writebool2{option, lineno}:
    begin
      if st[p + 1] = ord(true)
        then
          begin
            write(outfile,
```

```
                :st[s]);
            s := s - 2
          end (st[s - 1] = 1)
        else
          begin
            write(outfile,
              st[s] = 1);
            s := s - 1
          end;
      p := p + 3
    end;

writeint2(option, lineno):
  begin
    if st[p + 1] = ord(true)
      then
        begin
          write(outfile,
            st[s - 1]:
              st[s]);
          s := s - 2
        end
      else
        begin
          write(outfile,
            st[s]);
          s := s - 1
        end;
    p := p + 3
  end;

writereal2(option1, option2,
    lineno):
  begin
    if st[p + 1] = ord(true)
      then
        if st[p + 2] =
          ord(true) then
            begin
              s := s - 4;
              dx.b :=
                st[s + 1];
              dx.c :=
                st[s + 2];
              m :=
                st[s + 3];
              n :=
                st[s + 4];
              write(outfile,
                dx.a:m:n)
            end
          else
            begin
              s := s - 3;
              dx.b :=
                st[s + 1];
              dx.c :=
                st[s + 2];
              m :=
                st[s + 3];
```

```
                    dx.a:m)
                end
            else    write(outfile,
              begin
                s := s - 2;
                dx.b := st[s + 1];
                dx.c := st[s + 2];
                write(outfile,
                  dx.a);
              end;
          p := p + 4
        end;

      writestring2{option, lineno}:
        begin
          if st[p + 1] = ord(true)
            then
              begin
              { write(outfile,
                  sx:width) }
                width := st[s];
                s := s - 1;
                popstring(sx);
                writestring(outfile,
                  sx, width)
              end
            else
              begin
              { write(outfile, sx) }
                popstring(sx);
                writestring(outfile,
                  sx,
                  stringlength(sx))
              end;
          p := p + 3
        end;

      writeln2{lineno}:
        begin
          writeln(outfile);
          p := p + 2
        end;

    { OpenStatement =
        OpenParameters .
      OpenParameters =
        OpenParameter
          [ OpenParameter ]* .
      OpenParameter =
        VariableAccess "open" . }

      open2{lineno}: modificado
        begin
{ begin of modifications (fausto) }
          cmax := opensp;
          if cmax > maxchan then
            error(st[p + 1],
              maxchan5)
          else
            begin
```

```
              s := s - 1;
              p := p + 2
            end
          end;
{ end of modifications (fausto) }

    { ReceiveStatement =
        ReceiveParameters .
      ReceiveParameters =
        ChannelExpression
          "checkio"
            InputVariableList
              "endio" .
      ChannelExpression =
        Expression .
      InputVariableList =
        InputVariableAccess
          [ InputVariableAccess ]*
            .
      InputVariableAccess =
        VariableAccess
          "receive" . }

      checkio2{lineno}:
        begin
          c := st[s];

{begin of modifications}

          if (c < 0) or (c > cmax)
            then
              error(st[p + 1],
                channel4)
            else p := p + 2
          end;

{end of modifications}

      endio2:
        begin
          s := s - 1;
          p := p + 1
        end;

      receive2{typeno, length,
          lineno}:
        begin
          typeno := st[p + 1];
          length := st[p + 2];
          lineno := st[p + 3];
          c := st[s - 1];
{ begin of modifications (fausto) }
          typeno2 := receivesp(c, message, typeno, length);
          if typeno = typeno2 then
            begin
              x := st[s] - 1;
              s := s - 1;
              for i := 1 to length do
                st[x + i] := message[i];
              p := p + 4;
```

```
          else if typeno2 = -1 then
                error(lineno,
            end   contention4)
                else
                 error(lineno, type4);
        end;
{ end of modifications (fausto) }
    { SendStatement =
        SendParameters .
      SendParameters =
        ChannelExpression
          "checkio"
            OutputExpressionList
              "endio" .
      OutputExpressionList =
        OutputExpression
          [ OutputExpression ]* .
      OutputExpression =
        Expression "send" . }

      send2{typeno, length,                    modify
          lineno}:
        begin
          typeno := st[p + 1];
          length := st[p + 2];
          lineno := st[p + 3];
          c := st[s - length];
          { begin of modifications (fausto) }
          s := s - length;
          for i:= 1 to length do
           message[i] := st[s + i];
          typeno2 := sendsp(c, message, typeno, length);
          if typeno2 = -1  then
                error(lineno, contention4)
          else if typeno2 = -2 then
                error(lineno, type4);
          p := p + 4;
        end;
        { end of modifications (fausto) }
    { ProcedureStatement =
        ActualParameterPart
          "proccall" |
        StandardProcedureStatement .
      StandardProcedureStatement =
        ReadStatement |
        WriteStatement |
        OpenStatement |
        ReceiveStatement |
        SendStatement .
      IfStatement =
        Expression "do" Statement
          [ "goto" Statement ] .
      WhileStatement =
        Expression "do"
          Statement "goto" .
      RepeatStatement =
        StatementSequence
         Expression "do" . }

      do2{displ}:
```

```
      if st[s] = ord(true) then
        p := p + 2
      begin
        else p := p + st[p + 1];
        s := s - 1
      end;

    goto2{displ}:
      p := p + st[p + 1];

  { ForStatement =
      ForClause ForOption .
    ForClause =
      VariableAccess
        Expression "for" .
    ForOption =
      UpClause | DownClause .
    UpClause =
      Expression "to"
        Statement "endto" .
    DownClause =
      Expression "downto"
        Statement "enddown" . }

    for2: modi pcode
      begin
        st[st[s - 1]] := st[s];
        s := s - 1;
        p := p + 1
      end;

    to2{displ}:
      begin
        if st[st[s - 1]] <= st[s]
          then p := p + 2
          else
            begin
              s := s - 2;
              p := p + st[p + 1]
            end
      end;

    endto2{disp}:
      begin
        x := st[s - 1];
        st[x] := st[x] + 1;
        p := p + st[p + 1]
      end;

    downto2{displ}:
      begin
        if st[st[s - 1]] >= st[s]
          then p := p + 2
          else
            begin
              s := s - 2;
              p := p + st[p + 1]
            end
      end;

    enddown2{displ}:
```

```
          x := st[s - 1];
          st[x] := st[x] - 1;
        begin
          p := p + st[p + 1]
        end;

    { CaseStatement =
        Expression "goto"
          CaseList "case" .
      CaseList =
        StatementSequence . }

      case2{
        lineno, length, table}:
          begin
            x := st[s];
            s := s - 1;
            { binary search }
            i := 1;
            j := st[p + 2];
            while i < j do
              begin
                k :=
                 (i + j) div 2;
                m := p + 2*k + 1;
                if st[m] < x
                  then i := k + 1
                  else j := k
              end;
            m := p + 2*i + 1;
            if st[m] = x then
              p := p + st[m + 1]
            else
              error(st[p + 1],
                case4)
          end;

    { ParallelStatement =
        "parallel"
          ProcessStatementList
            "endparallel" .
      ProcessStatementList =
        ProcessStatement
          [ ProcessStatement ]* .
      ProcessStatement =
        "process"
          StatementSequence
            "endprocess" . }

      parallel2:——modificada
        begin
{ begin of modifications (fausto) }
          bendparallel2 := b;
          sendparallel2 := s;
          tendparallel2 := t;
{ end of modifications (fausto) }
          s := s + 1;
          st[s] := 0;
          p := p + 1
        end;
```

```
{ begin of modifications (fausto) }
        begin
          returnaddress := p + 2;
        endparallel2(lineno):
          nprocsparallel := st[s];
          for i := 1 to nprocsparallel do
            begin
              select(99);
              status := forksp;
              if status = 0 then
                exit;
            end;
          if status <> 0 then
            begin
              for i := 1 to nprocsparallel do
                waitsp(status);
              showstatus;
              b := bendparallel2;
              s := sendparallel2;
              t := tendparallel2;
              p := returnaddress;
            end;
        end;
{ end of modifications (fausto) }

      process2{blockno,
        templength, displ,
          lineno}:
        begin
          st[s] := st[s] + 1;
          blockno := st[p + 1];
          bi := free[blockno];
          if bi = 0 then
            begin
              bi := t + 1;
              t :=
                bi + st[p + 2] + 4
            end
          else
            free[blockno] :=
              st[bi];
          if t > maxaddr then
            memorylimit(st[p + 4])
          else
            begin
              st[bi] := b;
              st[bi + 1] := s;
              si := bi + 4;
              st[si] := blockno;
              activate(bi, si,
                p + 5);
              p := p + st[p + 3]
            end
        end;

      endprocess2{displ, lineno}:
        begin
          blockno := st[s];
          x := b;
          b := st[x];
          s := st[x + 1];
```

```
          free[blockno] := x;
          st[s] := st[s] - 1;
          st[x] := free[blockno];
{ begin of modifications (fausto) }
          exitsp(0);
{ end of modifications (fausto) }
        end;

    { ForallStatement =
        IndexVariableDeclaration
          "forall" Statement
            "endall" .
      IndexVariableDeclaration =
        Expression Expression . }

      forall2{blockno,
        templength, displ,
          lineno}:
        begin
          upper := st[s];
          lower := st[s - 1];
          if lower <= upper then
            begin
              s := s - 1;
              st[s] :=
                upper - lower + 1;
              blockno :=
                st[p + 1];
              templength :=
                st[p + 2];
              lineno := st[p + 4];
              for i := lower to
                upper do
              begin
                bi :=
                  free[blockno];
                if bi = 0 then
                  begin
                    bi := t + 1;
                    t := bi +
                      templength + 5
                  end
                else
                  free[blockno] :=
                    st[bi];
                if t > maxaddr then
                  memorylimit(lineno)
                else
                  begin
                    st[bi] := b;
                    st[bi + 1] := s;
                    st[bi + 4] := i;
                    si := bi + 5;
                    st[si] :=
                      blockno;
                    activate(bi, si,
                      p + 5)
                  end
              end;
{ begin of modifications (fausto) }
              bendforall2 := b;
```

```
                tendforall2 := t;
                returnaddress := p + st[p + 3];
                sendforall2 := s;
                nprocsforall := st[s];
                for i := 1 to nprocsforall do
                  begin
                    select(99);
                    status := forksp;
                    if status = 0 then
                      exit;
                  end;
                if status <> 0 then
                  begin
                    for i := 1 to nprocsforall do
                      waitsp(status);
                    b := bendforall2;
                    s := sendforall2;
                    t := tendforall2;
                    p := returnaddress;
                  end;
{ end of modifications (fausto) }
              end
            else { lower > upper }
              begin
                s := s - 2;
                p := p + st[p + 3]
              end
          end;

        endall2{lineno}:
          begin
{ begin of modifications (fausto) }
            exitsp(0);
{ end of modifications (fausto) }
          end;

      { AssumeStatement =
          Expression "assume" . }

        assume2{lineno}:
          begin
            x := st[s];
            s := s - 1;
            if x = ord(false) then
              error(st[p + 1],
                assume4)
            else p := p + 2
          end;

      { Statement =
          AssignmentStatement |
          ProcedureStatement |
          IfStatement |
          WhileStatement |
          RepeatStatement |
          ForStatement |
          CaseStatement |
          CompoundStatement |
          ParallelStatement |
          ForallStatement |
          AssumeStatement |
```

```
  EmptyStatement =  .
  StatementSequence =
    Statement [ Statement ]* .
  CompoundStatement =
    StatementSequence .
  Block =
    [ ProcedureDeclaration ]*
      CompoundStatement .
  ProcedureDeclaration =
    "procedure" Block
       "endproc" . }

  procedure2{blockno,
    paramlength, varlength,
      templength, displ,
        lineno}:
    begin
      st[s + 1] :=
        s - st[p + 2] - 1;
      st[s + 3] := b;
      b := s;
      blockno := st[p + 1];
      s := free[blockno];
      if s = 0 then
        begin
          s := t + 1;
          t := s + st[p + 4];
          if t > maxaddr then
            memorylimit(
              st[p + 6])
        end
      else
        free[blockno] := st[s];
      st[s] := blockno;
      p := p + st[p + 5]
    end;

  endproc2:
    begin
      blockno := st[s];
      x := s;
      s := st[b + 1];
      p := st[b + 2];
      b := st[b + 3];
      st[x] := free[blockno];
      free[blockno] := x
    end;

{ Program =
    "program" Block
      "endprog" . }

  program2{blockno,
    varlength, templength,
      displ, lineno}:
    begin
      for i := 1 to maxblock
        do free[i] := 0;
      ready := 0;
      cmax := 0;
```

```
          s := b + st[p + 2] + 4;
          t := s + st[p + 3];
          b := stackbottom;
          if t > maxaddr then
            memorylimit(st[p + 5])
          else p := p + st[p + 4]
        end;

      endprog2:
{ begin of modifications (fausto) }
        begin
          typeno2 := sendsp(-1,message,0,0);
          running := false;
        end;
{ end of modifications (fausto) }

    { localvar(displ) =
        variable(0, displ) . }

      localvar2{displ}:
        begin
          s := s + 1;
          st[s] := b + st[p + 1];
          p := p + 2
        end;

    { localvalue(displ) =
        localvar(displ)
          value(1) . }

      localvalue2{displ}:
        begin
          s := s + 1;
          st[s] :=
            st[b + st[p + 1]];
          p := p + 2
        end;

    { localreal(displ) =
        localvar(displ)
          value(2) . }

      localreal2{displ}:
        begin
          x := b + st[p + 1];
          s := s + 2;
          st[s - 1] := st[x];
          st[s] := st[x + 1];
          p := p + 2
        end;

    { globalvar(displ) =
        variable(1, displ) . }

      globalvar2{displ}:
        begin
          s := s + 1;
          st[s] :=
            st[b] + st[p + 1];
          p := p + 2
        end;
```

```
{ globalvalue(displ) =
    globalvar(displ)
      value(1) . }

  globalvalue2{displ}:
    begin
      s := s + 1;
      st[s] := st[
        st[b] + st[p + 1]];
      p := p + 2
    end;

{ ordvalue =
    value(1) . }

  ordvalue2:
    begin
      st[s] := st[st[s]];
      p := p + 1
    end;

{ realvalue =
    value(2) . }

  realvalue2:
    begin
      x := st[s];
      s := s + 1;
      st[s - 1] := st[x];
      st[s] := st[x + 1];
      p := p + 1
    end;

{ ordassign =
    assign(1) . }

  ordassign2:
    begin
      st[st[s - 1]] := st[s];
      s := s - 2;
      p := p + 1
    end;

{ realassign =
    assign(2) . }

  realassign2:
    begin
      s := s - 3;
      x := st[s + 1];
      st[x] := st[s + 2];
      st[x + 1] := st[s + 3];
      p := p + 1
    end;

{ globalcall(displ) =
    proccall(1, displ) . }

  globalcall2{displ}:
    begin
```

```
            st[s] := st[b];
            st[s + 2] := p + 2;
            p := p + st[p + 1]
          end

      end
  end { run };

  procedure readtime(
   var t:  integer);
  begin
    { A nonstandard function reads
      the processor time in ms }
    t := clock
  end;

  procedure writetime(
    var outfile: text;
    t1, t2: integer);
  begin
    { Outputs the time interval
      t2 - t1 ms in seconds }
    writeln(outfile);

  { begin of modifications }

    writeln(outfile,
      (t2 - t1 + 500),
      ' ms')

  { end of modifications }
  end;

  procedure runtime(
    var codefile: binary;
    var inpfile, outfile: text);
  var t1, t2: integer;
  begin
    readtime(t1);
    run(codefile, inpfile, outfile);
    readtime(t2);
    writetime(outfile, t1, t2)
  end;

  procedure openoutput(
    var codefile: binary;
    var inpfile: text;
    outname: phrase);
  var outfile: text;
  begin
    if outname = screen then
      begin
        writeln(output);
        runtime(codefile,
          inpfile, output);
        writeln(output)
      end
    else
      begin
        { nonstandard rewrite }
```

```
      runtime(codefile,
        inpfile, outfile)
    end;
end;

procedure openinput(
  var codefile: binary;
  inpname, outname: phrase);
var inpfile: text;
begin
  if inpname = keyboard then
    openoutput(codefile,
      input, outname)
  else
    begin
      { nonstandard reset }
      reset(inpfile, inpname);
      openoutput(codefile,
        inpfile, outname)
    end
end;

procedure start;
var codename, inpname,
  outname: phrase;
  codefile: binary;
  select: boolean;
begin

{begin of modification}
  inittable;
  if (forksp <> 0) then begin { father }
    master;
  end else begin              { child }
    write('    code = ');
    readphrase(codename);
    write('    select files? ');
    readboolean(select);
    if select then
      begin
        write('    input = ');
        readphrase(inpname);
        write('    output = ');
        readphrase(outname);
        { nonstandard reset }
        reset(codefile, codename);
        openinput(codefile,
          inpname, outname)
      end
    else
      begin
        { nonstandard reset }
        reset(codefile, codename);
        writeln(output);
        runtime(codefile, input,
          output);
        writeln(output)
      end
  end;
{end of modification}
```

```
{begin of modification}
end;
begin  start; end.

{end of modification}
```