FERRAMENTAS PARA O DESENVOLVIMENTO DE PROGRAMAS PARALELOS DISTRIBUÍDOS

Astrid Luise Harriet Hellmuth

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUACÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENCÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTACÃO.

Aprovada por:

Prof. Valmir C. Barbosa, Ph.D.
(Presidente)

Prof. Cláudio Luis de Amorim, Ph.D.

Prof. Ageu Cavalcanti Pacheco Júnior, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
JANEIRO DE 1991

HELLMUTH, ASTRID LUISE HARRIET

Ferramentas Para o Desenvolvimento de Programas
Paralelos Distribuídos [Rio de Janeiro] 1991
VIII, 65 p., 29.7 cm, (COPPE/UFRJ, M. Sc.,
Engenharia de Sistemas e Computação, 1991)
TESE — Universidade Federal do Rio de Janeiro, COPPE
1. Processamento Paralelo
I. COPPE/UFRJ II. Título(série).

Este trabalho é dedicado à Mutti e a todos os meus amigos.

Desejo agradecer a todos os meus colegas da COPPE-Sistemas, pelo apoio e estímulo, em especial ao meu Orientador, por sua compreensão e confiança, à Ana e à Lúcia por suas constantes colaborações e ao Lélio por sua inestimável ajuda.

Agradeço também a convivência com o pessoal do Laboratório de Computação Gráfica, sempre incentivadora.

Seria impossível enumerar todas as pessoas que contribuiram de diversas formas para que eu pudesse realizar este trabalho. Muitas vezes apenas com a sua distante torcida, com um sorriso, com a sua companhia, me transmitiram aquela energia que nos faz ir adiante. Sou imensamente grata a todos esses amigos queridos.

Por todos estes motivos, desejo agradecer com todo o carinho ao Cláudio Esperança por estar sempre ao meu lado.

Resumo da Tese apresentada à COPPE como parte dos requisitos necessários para a obtencão do grau de Mestre em Ciências (M. Sc.)

Ferramentas para o Desenvolvimento de Programas Paralelos Distribuídos

Astrid Luise Harriet Hellmuth

Janeiro de 1991

Orientador: Valmir C. Barbosa
Programa: Engenharia de Sistemas e Computacão

Este trabalho aborda diversos aspectos relacionados ao projeto e implementação de programas distribuídos. As principais fontes de dificuldade para o desenvolvimento de tais programas são investigadas com o objetivo de identificar procedimentos que possam ser realizados com o auxílio de ferramentas. Apresentamos, então, um ambiente de programação que oferece um modelo simples para o projeto de aplicações e um conjunto de ferramentas para geração automática do código para a aplicação descrita e seu mapeamento de maneira eficiente em uma arquitetura.

Abstract of Thesis presented to COPPE as partial fulfillment of the requirements for the degree of Master of Science (M. Sc.)

Tools for Developing Parallel Distributed Programs

Astrid Luise Harriet Hellmuth
January, 1991

Thesis Supervisor: Valmir C. Barbosa
Department: Engenharia de Sistemas e Computacão

This work covers many aspects related do the design and implementation of distributed programs. The main sources of difficulties in developing such programs are investigated in order to identify activitires that can be performed by assistant tools. A programming environment that offers a simple model for application design and a set of tools for automatic code generation and efficient mapping on an architecture is presented.

Índice

I Introdução 1

II Processamento Paralelo 3

II.1 Multiprocessadores 3

II.2 O Conceito de Processamento Concorrente 4

II.3 Concorrência e Paralelismo 4

II.4 Processos e Tarefas 5

II.5 Estados de um Processo 6

II.6 Comunicação e Sincronização 6

II.7 Sistemas Distribuídos Ponto-a-ponto 7

III Desenvolvimento de Programas Paralelos 9

III.1 A Atividade de Programação 9

III.2 Ambientes de Programação 10

III.3 Geração de Programas Paralelos 10

III.4 A Necessidade de Ferramentas de Auxílio à Programação Paralela . . 11

III.5 Características Comuns das Especificações de Algoritmos Paralelos . . 12

III.5.1 Introdução 12

III.5.2 O Grafo Descrevendo a Aplicação 13

III.5.3 O Conjunto de Procedimentos 13

III.5.4 Entrada e Saída 14

III.6 A Troca de Mensagens 14

III.7 Tipos de Mensagens 16

III.8 Armazenamento de Mensagens 17

III.9 Comunicação Síncrona e Assíncrona 19

IV Um Modelo Para Programação Distribuída **20**

IV.1 Introdução 20

IV.2 Descrição da Aplicação 20

IV.3 Modelo Para Desenvolvimento das Tarefas 21

IV.4 Modelo Para a Comunicação 22

IV.5 Prevenindo Interbloqueios 23

V O Mapeamento de Aplicações em Arquiteturas **25**

V.1 Introdução 25

V.2 O Mapeamento da Aplicação 26

V.3 Alocação de Tarefas 26

VI Usando a Linguagem OCCAM **33**

VI.1 Propriedades da Linguagem OCCAM 33

VI.2 Os Elementos da Linguagem 34

VI.3 O Programa do Usuário 40

VI.4 Execução de um Programa 43

VII O Ambiente de Programação **46**

VII.1 Introdução 46

VII.2 Funcionamento do Ambiente 46

VII.3 Programa Exemplo 48

VII.4 O Grafo da Aplicação 49

VII.5 O Grafo da Arquitetura 50

VII.6 Alocação de Tarefas 52

VII.7 Alocação de Espaço de Armazenamento 52

VII.8 Construção da Aplicação Distribuída 53

VII.9 Mecanismo de Comunicação Assíncrona 53

VII.10 Sistema de Comunicação 54

VII.11 Encaminhamento de Mensagens 55

VII.12 Multiplexadores 57

VII.13 Demultiplexadores 57

VII.14 Roteadores 58

VII.15 Terminação da Aplicação 59

VII.16 Código Gerado para o Exemplo 61

VIIIConsiderações Finais **63**

Referências Bibliográficas **63**

Lista de Figuras

II.1 Sistema Distribuído Ponto-a-Ponto 8

III.1 Interbloqueio (*Deadlock*) na Comunicação 18

IV.1 Introdução de Processos para Armazenamento de Mensagens 24

V.1 Exemplo de Grafo de Processadores 28

V.2 Exemplo de Grafo de Tarefas 30

VI.1 Exemplo de Grafo para a Aplicação 42

VI.2 Aplicação Para o Exemplo de Configuração 44

VI.3 Arquitetura Para o Exemplo de Configuração 45

VII.1O Ambiente de Programação 47

VII.2Diagrama de Funcionamento do Ambiente 48

VII.3Grafo para a Aplicação Exemplo 49

VII.4Grafo para a Arquitetura Exemplo 51

VII.5Processos Instalados em um Processador 56

Lista de Tabelas

VII.1 Esquema de Roteamento 51

VII.2 Tabela de Alocação de Tarefas 52

VII.3 Tabela de Alocação de *buffers* 53

VII.4 Mapa de Canais dos Multiplexadores 58

VII.5 Mapa de Canais dos Demultiplexadores 59

Capítulo I

Introdução

Este trabalho aborda o desenvolvimento de um ambiente de programação de aplicações para sistemas distribuídos ponto-a-ponto. Definimos um conjunto de ferramentas visando os seguintes objetivos:

1. facilitar o desenvolvimento das tarefas que compõem uma aplicação, utilizando um modelo abstrato, que permite ao programador especificar sua aplicação com um formato similar ao encontrado na literatura para descrever algoritmos distribuídos;

2. dar suporte ao modelo de comunicação entre tarefas de modo que ela ocorra de forma segura, sem a ocorrência de interbloqueios (*deadlocks*), uma vez que esta é uma das principais fontes de dificuldade encontrada na programação paralela;

3. extrair um bom desempenho da aplicação do usuário, balanceando a carga de processamento com o emprego de um algoritmo para distribuição de tarefas entre processadores;

4. livrar o programador de qualquer preocupação com relação a arquitetura da máquina onde a aplicação será executada, realizando a configuração automática da aplicação, isto é, o ambiente se encarrega de todas as operações necessárias para atribuição das tarefas aos processadores e manutenção da rede de comunicação entre tarefas.

Os conceitos básicos relacionados ao processamento paralelo são apresentados no Capítulo II.

Descrevemos as principais abordagens para o desenvolvimento de aplicações paralelas no Capítulo III. Identificamos as propriedades comuns às diversas descrições de algoritmos encontradas na literatura e as principais fontes de dificuldade encontradas durante a implementação desses algoritmos.

No Capítulo IV propomos um modelo genérico para a elaboração de programas paralelos, que permite ao programador expressar apenas as características essenciais ao comportamento de uma aplicação. A capacidade de armazenamento de mensagens trocadas entre as tarefas é uma questão crucial para a implantação do modelo de comunicação, que também analisamos nesse Capítulo.

Em seguida, no Capítulo V, expomos os problemas a serem resolvidos durante o mapeamento de aplicações em arquiteturas multiprocessadas, e descrevemos a utilização de um algoritmo para o balanceamento da carga de processamento.

Mostramos, no Capítulo VI, que a linguagem OCCAM oferece uma forma adequada para expressar o paralelismo de uma aplicação projetada de acordo com o modelo. No Capítulo VII descrevemos um ambiente para a geração automática do código correspondente à descrição fornecida nos termos do modelo de programação proposto.

Durante o projeto e implementação das ferramentas, procuramos empregar os resultados obtidos em trabalhos anteriores, reunindo-os de forma a obter um conjunto integrado de soluções para os problemas que se apresentam durante o desenvolvimento de uma aplicação distribuída. A utilização de um ambiente do tipo que propomos facilita a geração de programas corretos e confiáveis, com bom desempenho, proporcionando um aumento de produtividade ao programador.

Capítulo II

Processamento Paralelo

II.1 Multiprocessadores

Conforme computadores mais poderosos são oferecidos à comunidade de usuários, maiores se tornam as suas exigências. Existe porém um limite para a máxima velocidade alcançável com um computador baseado em um único processador. Além do mais o seu custo cresce mais rapidamente de acordo com a proximidade de tal limite. Uma solução radicalmente diferente é mover de uma arquitetura uniprocessada para uma nova arquitetura empregando paralelismo, ou seja, um certo número de processadores cooperantes [1].

A essência do paralelismo não é nova. A natureza é cheia de casos onde criaturas aparentemente frágeis e incapazes de grandes feitos, tais como formigas e abelhas, realizam façanhas incríveis através do empenho coletivo. A própria tecnologia que possibilitou o surgimento destas recentes arquiteturas é o resultado de esforços humanos coletivos: graças aos avanços da tecnologia VLSI tornou-se possível a fabricação, com baixo custo, de módulos de circuito integrado contendo diversos processadores, ou processador, memória e circuitos de comunicação.

As formas usualmente encontradas para organização desse novo tipo de arquitetura são conhecidas como SIMD e MIMD [2]. SIMD é uma abreviação para uma Seqüencia de Instruções, Múltiplas seqüências de Dados e corresponde aos chamados processadores vetoriais, que executam simultaneamente a mesma seqüência de instruções sobre diferentes conjuntos de dados. MIMD é uma abreviação para Múltiplas seqüências de Instruções, Múltiplas seqüências de Dados, i.e., podemos ter diferentes seqüências de controle executando diferentes instruções e manipulando diversos conjuntos de dados. A segunda opção é a mais flexível e corresponde aos chamados multiprocessadores.

II.2 O Conceito de Processamento Concorrente

O *processamento concorrente* é definido como o uso de diversas entidades cooperantes (idênticas ou não), trabalhando em conjunto para atingir um objetivo comum [3].

Na computação concorrente, as entidades são computadores e o objetivo pode ser um cálculo científico de larga escala (como por exemplo previsão meteorológica), ou uma aplicação de inteligência artificial (tal como ganhar um campeonato de xadrez ou controlar o sistema de defesa de uma nação).

Uma computação típica consiste de um algoritmo aplicado a um extenso conjunto de dados denominado *domínio*. A concorrência é alcançada por decomposição desse domínio, i.e, o conjunto de dados original é divido em partes (*grains*) que são atribuídas a cada computador. Os algoritmos executados nos computadores são similares aos empregados no processamento seqüencial, a menos de duas diferenças fundamentais:

- o conjunto de dados sobre o qual aplica-se o algoritmo é quantitativamente diferente, uma vez que ele corresponde a uma parte apenas do domínio;
- a necessidade de cooperação entre os computadores tende a envolver alguma forma de comunicação entre eles.

Como cada computador trata apenas de uma parcela do conjunto original de dados, espera-se que o tempo gasto para tratar todo o domínio do problema seja menor do que se fosse tratado por um único computador. Intuitivamente, a razão entre o tempo para execução seqüencial em um único computador, e o tempo gasto durante o processamento concorrente deve ser muito próximo ao número de computadores empregados.

II.3 Concorrência e Paralelismo

De acordo com o contexto, os termos **concorrente** e **paralelo** são por vezes usados com significados similares porém distintos. Diz-se que duas entidades são executadas em paralelo se, em algum instante de tempo, ambas estão realmente sendo executadas. Em contrapartida, duas entidades são descritas como concorrentes se existe a possibilidade de executá-las em paralelo. Uma linguagem de programação concorrente possui, portanto, mais de uma seqüência distinta de controle, e os objetos que possam vir a ser executados em paralelo são diretamente representados [4].

Acreditamos que tal rigor na utilização dos termos **concorrente** e **paralelo** só se faz necessário quando se deseja distinguir entre a execução de um

programa concorrente em um único processador, ou seja, quando de fato não existe paralelismo, e em diversos processadores. Por esse motivo, utilizaremos ambos os termos livremente, sem distinção, uma vez que a capacidade de execução paralela estará sempre presente.

II.4 Processos e Tarefas

Os objetos que podem ser executados simultaneamente em um ambiente MIMD são normalmente conhecidos como **processos**. De maneira informal, cada processo pode ser visto como uma instância de um programa. Assim como um circuito elétrico pode possuir várias instâncias de um certo tipo de componente, uma computação concorrente pode conter várias instâncias de um ou mais programas [5].

Outro termo empregado com freqüência como sinônimo de processo é **tarefa**, especialmente quando o tratamento do processamento paralelo é baseado na idéia de processos seqüenciais comunicantes. Por exemplo, em [6] uma aplicação é definida como uma coleção de uma ou mais tarefas (ou processos) concorrentes.

Não é raro, entretanto, encontrar o termo tarefa, na literatura técnica, designando um conjunto de processos. Uma boa descrição para este tipo de distinção entre processos e tarefas é dada em [7] onde um programa a ser executado em um multiprocessador também é visto como uma coleção de elementos, denominados tarefas. Nesse contexto, cada tarefa é uma unidade de escalonamento a ser atribuída a um ou mais processadores e pode consistir de um ou mais processos. Cada processo é uma coleção de instruções de programa, e é definido como uma unidade indivisível com respeito à alocação do processador.

Em outras situações, o termo **tarefa** não é usado de todo. É o caso da linguagem de programação concorrente OCCAM, onde um programa, em seu nível mais alto, é visto como um único processo. Um processo pode conter outros processos, variando desde uma única atribuição até um programa completo, de forma que uma estrutura hierárquica é suportada. Ou seja, processos maiores são elaborados a partir dos chamados *processos primitivos* com o auxílio de elementos denominados *construtores* [8,9].

Enfim, ao investigar estes e outros exemplos da literatura, percebemos o seguinte:

- Os termos processo e tarefa são utilizados de forma intercambiável, i.e., na grande maioria dos casos é possível substituir uma palavra pela outra sem causar ambigüidade.
- No tratamento de problemas tais como o de divisão de um procedimento em diversos processos, ou *particionamento*, e atribuição de processos a processadores, ou seja, o *escalonamento*, como por exemplo em [14] é confortável considerar:

processo como uma *unidade de computação* e

tarefa como uma *unidade de escalonamento.*

II.5 Estados de um Processo

Para dar suporte ao conceito de processo, cada processador executa um código que permite a coexistência de múltiplos processos. Assim sendo, o número de processos concorrentes envolvidos em uma única computação pode exceder em muito o número de processadores nela envolvidos.

Uma vez que tenha sido criado, um processo pode se encontrar em diversos estados. Os três estados primários para um processo são [10]:

- executando
  Quando um processo estiver usando um processador para executar instruções;
- pronto ou executável
  Quando um processo pode iniciar (ou continuar) sua execução porém não há processador disponível;
- suspenso ou bloqueado
  Quando um processo estiver esperando a ocorrência de algum evento. Por exemplo, um processo pode ser temporariamente suspenso devido à indisponibilidade momentânea de algum recurso (processador, dados de entrada, memória, comunicação e assim por diante). Nesse caso um outro processo é escalonado para execução no processador.

II.6 Comunicação e Sincronização

Apesar de cada um dos processos possuir seu código próprio e variáveis particulares, eles raramente são independentes uns dos outros. As maiores dificuldades associadas à programação paralela surgem justamente quando a interação entre os processos é tratada; isso é, quando se especifica como a transferência de dados entre os processos ocorre e como eles devem sincronizar suas ações.

Existem dois esquemas complementares de comunicação entre processos [11]:

- Memória Compartilhada
- Troca de mensagens

Sistemas de memória compartilhada requerem que processos comunicantes compartilhem algumas variáveis. Espera-se que os processos troquem informações através do uso dessas variáveis. Variáveis compartilhadas são objetos aos quais dois ou mais processos têm acesso; a partir da leitura e/ou escrita dessas variáveis os dados são passados de um processo a outro.

Sistemas de troca de mensagens permitem que os processos envolvidos na comunicação enviem seus dados de maneira explícita através de mensagens.

Associado ao ato de comunicar dados está o conceito de sincronização de processos, que descrito de maneira simplificada, corresponde à necessidade de um processo adquirir algum conhecimento acerca das atividades de um outro processo, a fim de coordenar a execução de ambos. Por exemplo, para que um processo receba uma mensagem é necessário que algum outro processo a tenha enviado.

A sincronização é uma forma especial de comunicação, na qual o dado é uma informação de controle [12]. Ela serve ao duplo propósito de assegurar o correto seqüenciamento de processos e o acesso mutuamente exclusivo a dados compartilhados. Por exemplo, os mecanismos de sincronização podem ser usados para:

- Controlar um processo produtor e um processo consumidor de forma que o processo produtor não atualize dados que ainda não foram utilizados pelo consumidor, ou impedir que o processo consumidor obtenha dados inválidos;
- Proteger os dados em um banco de dados de forma que não sejam permitidos acessos concorrentes para escrita em um mesmo registro. Esse tipo de acesso pode levar à perda de uma ou mais atualizações se dois processos executarem uma leitura em seqüencia e depois uma escrita em seqüencia dos dados atualizados.

II.7 Sistemas Distribuídos Ponto-a-ponto

A computação onde os diversos elementos processadores não compartilham memória, e, portanto, a comunicação entre processadores é realizada exclusivamente através da troca de mensagens, também é denominada de computação distribuída [13]. Um dos critérios para classificá-la é justamente quanto à forma pela qual a comunicação entre os processadores acontece. Segundo esse critério, uma divisão bem ampla pode ser feita entre sistemas distribuídos onde a comunicação se efetua por difusão (*broadcast*), e aqueles em que a comunicação se dá de forma ponto-a-ponto.

Quando a comunicação é realizada por **difusão**, todos os processadores do sistema podem se comunicar diretamente com todos os outros, através do uso de um ou mais canais de comunicação compartilhados por eles. Já os sistemas **ponto-a-ponto** podem ser representados por grafos conexos em que cada

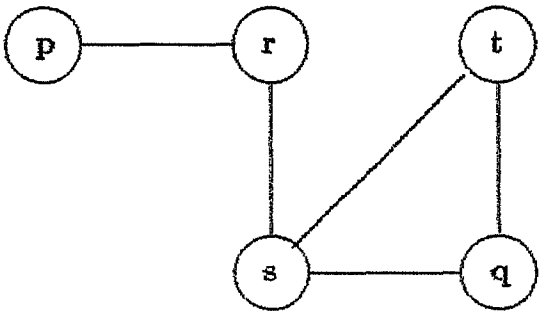

Figura II.1: Sistema Distribuído Ponto-a-Ponto

nó representa um processador e cada arco entre dois nós representa um canal de comunicação. Esse canal de comunicação é de uso exclusivo dos dois processadores representados por aqueles nós.

Essa forma de comunicação gera a necessidade de mensagens precisarem ser enviadas ao longo de rotas que passam por outros processadores que não sua origem e seu destino. Por exemplo, na figura ilustrando um sistema distribuído ponto-a-ponto, uma mensagem enviada pelo processador **p** com destino a **q** deverá, necessariamente passar pelos processadores **r** e **s**.

Neste trabalho, a discussão acerca do desenvolvimento de aplicações para sistemas MIMD de processamento paralelo, se referirá àqueles sistemas que possam ser identificados como sistemas distribuídos ponto-a-ponto.

Ω

Capítulo III

Desenvolvimento de Programas Paralelos

III.1 A Atividade de Programação

Antes de abordar o projeto de ferramentas para o auxílio à programação, é interessante analisar a atividade de programação em si, para determinar que mecanismos ou procedimentos podem ser incluídos para simplificá-la ou métodos que evitem complicações desnecessárias.

Quando nos propomos a resolver um determinado problema através do computador, precisamos, em primeiro lugar, elaborar um método ou algoritmo, que se seguido, nos dará o resultado deste problema. Podemos dizer que um algoritmo é uma seqüência de passos que manipulam informações a fim de obter um resultado. Um algoritmo é perfeitamente executável por uma pessoa, no entanto, não pode ser compreendido pelo computador. Necessário se faz traduzi-lo para alguma forma de linguagem que seja inteligível para a máquina ou linguagem de programação. Ao algoritmo assim traduzido damos o nome de programa.

O algoritmo é uma abstração descrevendo um procedimento, cuja generalidade transcende detalhes específicos de qualquer implementação. Como resultado, quando um algoritmo é especificado na literatura técnica, vários detalhes são omitidos propositadamente, ou, no máximo considerados implícitos, pois têm pouca ou nenhuma relevância na operação do algoritmo.

A programação é uma atividade de conversão de um *algoritmo* para uma forma chamada *programa*, que pode ser executada em um computador. Os detalhes omitidos precisam ser definidos no curso da programação uma vez que o computador só segue instruções explícitas. Portanto, a programação será mais fácil ou difícil, conforme a forma do algoritmo for ou não semelhante à forma do programa desejado.

Um exemplo simples seria a implementação de um algoritmo seqüencial, no qual o mecanismo de recursão é usado, através de uma linguagem de programação não-recursiva. Nesse caso, a programação se tornará difícil porque é necessário, de fato, implementar um pacote de suporte à recursão dentro dos mecanismos existentes na linguagem.

Assim sendo, é possível reduzir em grande parte as dificuldades encontradas na programação mantendo uma especificação ajustada àquela usada na literatura para descrever algoritmos, isto é, minimizando a conversão a ser realizada.

III.2 Ambientes de Programação

A forma original dos algoritmos pode, muitas vezes, parecer inatingível, uma vez que ela é revelada por uma sintaxe ou semântica pré-ordenada, dirigidas aos seus leitores e não às máquinas. Porém, freqüentemente encontramos características comuns em diversas especificações de algoritmos. A partir dessas propriedades podemos desenvolver mecanismos para o auxílio na tarefa de programação, que denominamos *ferramentas*.

Um *ambiente de programação* seria então a coleção de todas as ferramentas de linguagem e sistema operacional necessárias para suportar a programação, integradas em um único sistema [15] .

III.3 Geração de Programas Paralelos

Atualmente, a maioria das aplicações para sistemas distribuídos são desenvolvidas em um ambiente do tipo hospedeiro/máquina-alvo. Os programas são escritos em um computador hospedeiro e depois são carregados para execução no sistema alvo. O hospedeiro normalmente é uma estação de trabalho conectada ao multiprocessador através de um barramento ou de uma rede local. Vários multiprocessadores são programados dessa forma: iPSC, Cosmic Cube, NCUBE, etc [16].

Esse é o nível mais básico, e sempre presente, de ferramentas, ou seja, pacotes de programas executados no hospedeiro, que permitem a edição, compilação, depuração de aplicações e o seu carregamento na máquina-alvo para execução. Exemplos deste tipo de pacote são:

- Concurrent Workbench — pacote para desenvolvimento de programas para o iPSC, da Intel [17], contendo basicamente, compiladores C e FORTRAN, depurador e bibliotecas de rotinas vetoriais ;

- TDS (Transputer Development System) — sistema de desenvolvimento que permite a edição, compilação e execução de programas OCCAM para um transputer ou para uma rede de transputers [18].

No entanto, as facilidades oferecidas por tais sistemas não são suficientes para lidar com todos os aspectos envolvidos na programação paralela. Comparada à escrita de programas seqüenciais, a complexidade da programação para ambientes distribuídos é aumentada devido a fatores tais como: a concorrência, a comunicação e a sincronização entre os processos, e ainda à necessidade de mapear os diversos processos no conjunto de processadores disponível.

III.4 A Necessidade de Ferramentas de Auxílio à Programação Paralela

Em teoria, é fácil argumentar que um multiprocessador é mais rápido do que um computador com um único processador: divide-se uma tarefa em subtarefas, que são executadas simultaneamente em diferentes processadores, almejando alcançar um tempo de execução igual ao tempo gasto para executar a tarefa em um único processador dividido pelo número de processadores empregado na execução paralela. Embora válido, este raciocínio implica em considerar desprezível a despesa incorrida durante a decomposição e implementação de um algoritmo, o que está longe de ser uma verdade.

Existem basicamente três abordagens para o desenvolvimento de programas paralelos [14] :

- A primeira abordagem considera a paralelização um problema tão complexo que só pode ser resolvido manualmente. Por exemplo, Garg [19], divide os esforços de pesquisa para o tratamento de programas com múltiplas tarefas em dois grupos ou linhas: — manual, que deu origem a diversos sistemas para provar determinadas propriedades de algoritmos, tais como segurança, garantia de terminação, etc. ; e automática, que deu origem, por exemplo, a sistemas para análise automática de programas concorrentes, que exploram todos os comportamentos possíveis de um sistema.

  O tratamento e análise manual de programas distribuídos não é nada eficaz. Programadores tendem a cometer erros e não são muito eficientes em resolver problemas com muitas tarefas. Por exemplo, o interbloqueio entre tarefas do sistema (*system deadlock*) é um problema comum, difícil de detectar uma vez que o programa tenha sido desenvolvido.

- Outra forma de tratar o paralelismo seria através de compiladores para transformação automática de programas seqüenciais em programas paralelos. Smith [20], por exemplo, aponta como uma das dificuldades para o desenvolvimento