INSTITUTO DE FÍSICA TEÓRICA

Resolução da Equação de Laplace no Sistema Multiprocessador ACP

José Rodolfo Ferreira Xavier

INSTITUTO DE FÍSICA TEÓRICA TM-05/90

Resolução da Equação de Laplace no Sistema Multiprocessador ACP

José Rodolfo Ferreira Xavier

Janeiro/90

Tese apresentada para obtenção do grau de Mestre em Física

Orientador: Prof. Dr. Gerson Francisco

Resumo

Neste trabalho desenvolvemos algoritmos e programas para a resolução numérica de uma equação diferencial parcial (equação de Laplace) no sistema multiprocessador ACP, usando duas abordagens diferentes: resolução numérica com e sem comunicação direta entre os nós.

Os resultados indicam que (i) para matrizes, que representam os domínios de discretização, pequenas (de até 100 kbytes) deve-se utilizar um uniprocessador (no caso, foi usado o Micro VAX II); (ii) para matrizes de tamanho médio (100 - 1000 kbytes) deve-se usar o ACP, porém sem comunicação direta entre os nós; e, finalmente, (iii) para matrizes grandes (acima de 1 Mbyte) deve-se usar o programa paralelo com comunicação direta entre os processadores.

Abstract

In this work we develop algorithms and computer programs to solve a partial differential equation (Laplace's equation), numerically, on the ACP multiprocessor using two distinct approaches: only host-to-node communication and direct node-to-node communication.

The results show that: (i) for small discretization matrixes (up to 100 kbytes) no parallelization should be used (instead, the uniprocessor , the Micro VAX II, is recommended); (ii) for medium-sized matrixes (100 - 1000 kbytes) the ACP should be used, but without direct node-to-node communication; and, finally, (iii) for large matrixes (more than 1 Mbyte) the parallel program with direct node-to-node communication is the most efficient

Para Ignez e Xavier,

meus pais

Agradecimentos

Depois de meses trabalhando nesta tese me defronto com a tarefa de preparar os agradecimentos e percebo o quão difícil isto pode ser: é impressionante a quantidade de pessoas que me ajudaram, de uma maneira ou de outra, a chegar até aqui.

Devo meus agradecimentos a (entre outros):

Prof. Gerson Francisco, por ter me dado a oportunidade de participar deste projeto, pela orientação e confiança depositada em mim

Prof. Ruben Aldrovandi, diretor do IFT, pelo apoio ao projeto

Prof. Alberto Franco de Sá Santoro, pela colaboração quanto ao uso do ACP de seu laboratório, sem o qual a conclusão deste trabalho não teria sido possível

Carla, Bruno e Mariano, pelas valiosas trocas de idéias quanto ao uso do ACP, convites para seminários e muitas (mesmo!) conexões via modem ao VAX do CBPF

George E.A. Matsas, meu amigo, pela minha indicação para este projeto, pelas discussões, sugestões e leitura atenta que fez dos manuscritos

Profª. Maria Lúcia dos Santos Teles, Instituto de Física da USP, a primeira pessoa a acreditar no meu trabalho

índice

prefácio i
Sobre este trabalho ii
Por que num Instituto de Física? iii
Organização gráfica iv
Organização geral iv
Notação iv
Numeração das páginas v
Conteúdos v
Referências vi
Observação quanto às traduções vi
Modo recomendado de leitura vi

capítulo 1 - Processamento Paralelo 1
Introdução 2
Evolução dos sistemas de computadores 2
Aplicações do processamento paralelo 5
Paralelismo em sistemas com apenas 1 processador 5
Estrutura dos computadores paralelos 8
Computadores com pipeline 8
Processadores matriciais 9
Multiprocessadores 9
Classificação de acordo com a arquitetura (Flynn) 11
Referências 14

capítulo 2 - Arquitetura dos Multiprocessadores 15
Introdução 16
Multiprocessadores 16
Multiprocessadores fracamente acoplados 17
Multiprocessadores fortemente acoplados 19
Redes de interconexão 21
Rede de tempo compartilhado 21
Rede tipo crossbar 23
Memória multiporta 24
Referências 27

capítulo 3 - Concorrência nos Multiprocessadores 28
Introdução 29
Elementos de linguagem paralela 29
Detecção automática de paralelismo 32
As condições de Bernstein 34
Reestruturação de programas 36
Paralelização do algoritmo 36
Comunicação 38
Referências 38

capítulo 4 - Desenvolvendo Algoritmos Paralelos 39
Introdução 40
Conhecimento extra é importante 40
Custos de comunicação devem ser considerados 41

O algoritmo deve estar adaptado à arquitetura 41
Algoritmos para computadores MIMD 42
Classificação dos algoritmos MIMD 42
Fatores que limitam o speedup 43
Analisando a complexidade dos algoritmos MIMD 46
Referências 47

capítulo 5 - O Sistema ACP 48
Introdução 49
O Conceito do multiprocessador ACP 49
Os processadores do ACP 50
A arquitetura do sistema usado 50
Programando o ACP 52
Como rodar um programa no ACP 53
Rotinas importantes do ACP 55
O conceito de classes de nós 56
Referências 56

capítulo 6 - Equações Diferenciais Parciais 57
Introdução 58
As diferentes equações diferenciais parciais 58
A equação de Laplace em diferenças finitas 59
Resolução numérica das EDP's 60
O método de Jacobi 61
O método de Gauss-Seidel 62
O método SOR 62
Ordenamento ímpar-par 64
Aceleração de Chebyshev para o SOR 64
Referências 65

capítulo 7 - O Programa CALOR (Hospedeiro-Nós) 66
Introdução 67
Descrição geral 67
A Idéia da subdivisão do domínio 69
O balanceamento estático de carga 70
A comunicação hospedeiro-nós 72
As conversões de formato dos dados 74
Referências 75

capítulo 8 - O Programa CALOR (Nós-Nós) 76
Introdução 77
A idéia básica 77
Nós de diferentes classes 82
Comunicação direta entre os nós ACP 82
Cálculo dos endereços VME 83
Monitorações 85
Ativação e desativação dos programas dos nós 86
Formato dos dados 86
Definição do arquivo tipo UPF 87

Atribuindo programas diferentes a diferentes nós 87
Tempo máximo de execução do programa do hospedeiro 87
Tempo máximo de execução dos programas dos nós ACP 88
Referências 88

capítulo 9 - Dados e Análises 89
O problema e sua solução 90
Solução analítica 90
Solução numérica 92
Resolução num uniprocessador 93
Resolução sem comunicação entre os processadores ACP 94
Resolução com comunicação direta entre os processadores ACP 95
Comparação entre os diversos resultados 96
Comparação entre os speedups 99
Comparação entre os desempenhos dos programas paralelos 100
Número de processadores 101

capítulo 10 - Conclusões 102

apêndice A - Programando o MC68020 A
Introdução B
Programando um microprocessador B
As instruções C
Os registradores C
Modos de endereçamento E
Endereçamento direto de registradores F
Endereçamento indireto, via registrador F
Endereçamento indireto, via memória H
Endereçamento indireto, via contador de programa (PC) H
Endereçamento indireto, via memória apontada pelo PC H
Endereçamento absoluto I
Endereçamento imediato I
Endereçamento implícito I
Conjunto de instruções do MC68020 J
Transferência de dados K
Aritmética inteira L
Operações lógicas M
Deslocamentos e rotações N
Manipulação de bits individuais O
Manipulação de campos de bits P
BCD P
Controle do programa Q
Controle do sistema S
Referências S

apêndice B - Listagem de CALOR (sem ACP) T

apêndice C - Listagem de CALOR (Hospedeiro-Nós) X
CALOR_H_N.UPF Y
CALOR_H_N_HOST.for AA
CALOR_H_N_NODE.for LL

apêndice D - Listagem de CALOR (Nós-Nós) SS
NO_A_NO.UPF TT
HOSPEDEIRO.for VV
MESTRE.for2 HHH
CALOR.for1 PPP

apêndice E - Referências Utilizadas YYY

glossário I

glossário de siglas IX

índice de figuras e tabelas XII

índice remissivo XVI

prefácio

prefácio

Sobre este trabalho

O trabalho que desenvolvo aqui é o início de um programa que visa dotar o Instituto de Física Teórica (IFT) de um sistema de processamento paralelo baseado num multiprocessador.

O sistema que estou usando, descrito em detalhe no capítulo "O Sistema ACP", é um multiprocessador desenvolvido no *Fermi National Accelerator Laboratory* (FERMILAB - EUA) e se encontra no Laboratório de Física Experimental de Altas Energias do Centro Brasileiro de Pesquisas Físicas (CBPF) no Rio de Janeiro.

Enquanto o Instituto de Física Teórica tenta obter seu próprio multiprocessador, decidiu-se começar o projeto fazendo estudos sobre Processamento Paralelo e desenvolvendo algoritmos paralelos de interesse para a Física.

Assim, decidimos, nesta primeira etapa, criar e implementar algoritmos para resolução de equações diferenciais parciais (com vistas a futuros trabalhos na área de Dinâmica de Fluidos, entre outras) e analisá-los a fim de adquirir mais conhecimento, tanto teórico quanto prático, sobre a aplicação de sistemas de computação em paralelo.

O problema que abordamos foi o da resolução da equação de Laplace com condições de contorno numa região bi-dimensional retangular, pois ele tem solução bem conhecida (tanto analítica quanto numérica, ambas mostradas no capítulo "Dados e Análises") e serve aos objetivos aos quais nos propusemos, mencionados acima.

Nos capítulos subseqüentes

- apresentarei as diversas arquiteturas de computadores até chegarmos aos multiprocessadores
- mostrarei o que é Processamento Paralelo e como utilizá-lo
- introduzirei alguns métodos numéricos (não paralelos) para resolução de equações diferenciais parciais
- explicarei as nossas idéias de como resolver este tipo de problema usando um sistema multiprocessador
- apresentarei dois programas que resolvem a equação de Laplace no multiprocessador ACP
- exibirei dados relativos às medidas de desempenho dos dois programas
- analisarei os resultados obtidos a fim de entender o comportamento dos programas paralelos

Pode-se, porém, argumentar que este projeto deveria ser desenvolvido por matemáticos ou por profissionais ligados à Ciência da Computação.

Vejamos qual a contribuição que o Instituto de Física Teórica pode dar nesta área.

Por que num Instituto de Física?

Os físicos são considerados, de um modo geral, grandes consumidores de tempo de computador e a razão disto é bastante óbvia: os modelos usados, hoje em dia, na explicação dos fenômenos físicos estão cada vez mais complexos e exigem cada vez mais cálculos.

Decidiu-se, assim, dotar o Instituto de Física Teórica de um sistema de computação bastante poderoso, que pudesse suprir esta necessidade. Para isto há duas opções razoáveis: ou se compra um computador de grande porte ou se usa um sistema de processamento paralelo de baixo custo.

Ambas as alternativas acima têm suas vantagens e desvantagens:

- por um lado já existe uma gama imensa de aplicativos (programas de computador) e algoritmos que podem ser usados nos computadores de grande porte, porém estes são muito caros

- já um sistema multiprocessador custa bem menos, mas não há muitos aplicativos disponíveis e há poucos algoritmos já desenvolvidos para eles (em comparação aos existentes para os computadores convencionais)

Mas atenção! Aqui está, talvez, o fiel da balança: eis a oportunidade de se desenvolver programas de computador voltados para aplicação à Física que façam realmente uma contribuição à comunidade cientffica! Quem melhor do que os próprios físicos para saber onde concentrar os esforços computacionais que os problemas de hoje em dia exigem?

Falta somente escolher qual multiprocessador usar. Por uma série de razões† optou-se pelo sistema ACP desenvolvido no FERMILAB (veja no capítulo "O Sistema ACP" uma descrição detalhada sobre ele).

Assim, teve início, no começo de 1987, o trabalho que estamos realizando, que tem como objetivos dotar o IFT de um sistema de computação eficiente (rápido), robusto (que não exija especialistas para sua manutenção), de baixo custo e, ao mesmo tempo, desenvolver programas e algoritmos que venham a ser usados por outros cientistas em outros centros de pesquisa.

Esta é a maneira correta, a meu ver, de como se deve fazer Ciência : com cooperação mútua espírito empreendedor.

---

† descritas nos documentos "Processamento Paralelo e Aplicações - aquisição de um MicroVAX", "Projeto de Pesquisa" e "Plano de Trabalho Detalhado para Utilização do Multiprocessador Paralelo", de autoria do Prof. Gerson Francisco, do Instituto de Física Teórica.

Organização gráfica

Neste trabalho procurei usar uma notação consistente, que facilite a identificação de conceitos importantes e permita uma melhor visualização geral da organização de cada um dos tópicos apresentados.

Todos os tópicos principais estão delimitados bor 2 barras horizontais e os sub-tópicos logicamente relacionados com eles por 1 barra horizontal.

As figuras, sempre que possível (mas não obrigatoriamente), aparecem na mesma página em que são mencionadas pela primeira vez. Quando isto não for possível haverá um indicação explícita, tal como: "na figura 2 mostrada na próxima página...", ou "na figura 2 a seguir..." .

Organização geral

A organização geral deste trabalho se processa da seguinte forma:

- 1 índice
- 1 prefácio
- 10 capítulos (de 1 a 10)
- 5 apêndices (de A a E)
- 1 glossário
- 1 glossário de siglas
- 1 índice de figuras e tabelas
- 1 índice remissivo

Notação

O sublinhamento é usado para se ressaltar um conceito ou idéia.

Todas as palavras grafadas **desta forma** constam do glossário .

As palavras em *língua estrangeira* são grafadas *em itálico* (com exceção das siglas, unidades de medida, comandos de linguagens, rótulos de tabelas e de gráficos, e legendas de figura e de tabelas).

As letras que designam variáveis (geralmente referenciadas por figuras ou usadas na explicação de um algoritmo, como por exemplo a variável `MATRIZ`) ou comandos de uma linguagem (tais

como nomes de funções ou sub-rotinas, como a função `long` do FORTRAN) são impressas `desta forma`, excetuando-se quando aparecerem em legendas de figuras e de tabelas.

A numeração das figuras é composta apenas de números, sem referência ao capítulo a que pertencem. Assim, a figura 2 do capítulo 3 é denotada por "figura 2" e a figura 2 do capítulo 5 também.

As legendas das figuras e legendas de tabelas são escritas destamaneira.

Numeração das páginas

As páginas do índice estão numeradas com letras minúsculas ("`a, b...`" etc.).

O prefácio está numerado com algarismos romanos minúsculos ("`i, ii, ...`" etc.).

Os capítulos estão numerados, seqüencialmente, com algarismos arábicos ("`1, 2...`" etc.).

Os apêndices estão numerados, seqüencialmente, com letras maiúsculas ("`A, B...Z, AA, BB...`" etc.).

Glossário, glossário de siglas, índice de figuras e tabelas e índice remissivo estão numerados seqüencialmente (nesta ordem) com algarismos romanos maiúsculos ("`I, II...`" etc.)

Conteúdos

Os apêndices B, C e D contêm as listagens (em FORTRAN) dos programas criados neste trabalho. No apêndice E encontram-se listadas, em ordem alfabética por título, todas as referências usadas neste trabalho.

O glossário contém uma descrição das palavras escritas **desta forma**, listadas em ordem alfabética. No caso de ser uma palavra em ***língua estrangeira*** constará do glossário a respectiva entrada em português também.

O glossário de siglas contém a forma por extenso de todas as siglas usadas neste trabalho, listadas em ordem alfabética, excluindo-se aquelas usadas na denotação das referências. No caso de ser uma sigla em *língua estrangeira* constará a respectiva tradução, com exceção dos nomes próprios.

O índice de figuras e tabelas lista as figuras e as tabelas presentes em cada um dos capítulos e apêndices.

O índice remissivo permite se localizar tópicos de interesse especial, mencionados ao longo do trabalho, listados em ordem alfabética. No caso de ser um tópico em *língua estrangeira*, constará a respectiva entrada em português também.

Referências

As referências estão listadas ao final de cada capítulo . Elas são denotadas, no decorrer de cada um deles, por um número entre colchetes escrito um pouco acima da linha. Por exemplo, dentro de um capítulo, denota-se a referência número 3 por [3] .

Note que cada capítulo tem as suas próprias referências, o que equivale a dizer que em capítulos diferentes, referências com o mesmo número, por exemplo [3], podem representar (e geralmente representam) obras diferentes.

No apêndice E encontram-se listadas, em ordem alfabética por título, todas as referências usadas.

Observação quanto às traduções

Em várias áreas de estudo, e notadamente na área de informática, existem termos provenientes de idiomas estrangeiros que são utilizados sem serem traduzidos, pois foram, de certa forma, incorporados pelos profisionais que com eles lidam.

Parece-me preciosismo exagerado (e até mesmo injustificado) querer traduzir todos os termos estrangeiros em favor de uma pretensa nacionalização do trabalho. Pelo contrário, acredito que alguns termos até mesmo devem ser utilizados no idioma original em favor de uma melhor compreensão e identificação com trabalhos publicados nos meios internacionais.

Assim, quando julguei necessário, me preocupei em traduzir alguns termos e, pela mesma razão, usei outros no seu idioma original (geralmente inglês).

Modo recomendado de leitura

Todos os capítulos deste trabalho são autoconsistentes para trás, o que equivale a dizer que eles podem fazer referências aos capítulos anteriores, mas nunca dependem de conceitos explicados em capítulos posteriores a eles para serem entendidos.

Assim, recomendo que todos os capítulos sejam lidos na ordem em que são apresentados, pelo menos numa primeira leitura.

capítulo 1

Processamento Paralelo

Processamento Paralelo

Introdução

O trabalho que desenvolvemos aqui é o início de um programa que visa dotar o Instituto de Física Teórica de um sistema de Processamento Paralelo baseado num multiprocessador. Mas, o que é um sistema de Processamento Paralelo? O que é um multiprocessador? Estas e muitas outras questões serão respondidads daqui para a frente, mas vamos adiantar a definição de Processamento Paralelo dada pelo Prof. Hwang [1]:

- Processamento Paralelo é uma forma eficiente de processamento da informação com ênfase na exploração de eventos concorrentes no processo computacional.

Como sempre, e aqui não somos exceção, uma breve introdução histórica sobre a evolução dos sistemas de computadores certamente servirá para nos localizarmos e entender melhor as idéias apresentadas no decorrer deste trabalho.

Evolução dos sistemas de computadores

Começamos nossa abordagem histórica sobre o paralelismo na arquitetura dos computadores considerando a redução do tempo de cálculo necessário para se efetuar uma operação aritmética simples (uma multiplicação de dois números em ponto flutuante, por exemplo) desde o primeiro computador produzido comercialmente, o UNIVAC 1, em 1951. Isto está mostrado na figura 1 na próxima página.

Quanto a esta figura salientamos que há um período de aproximadamente 10 anos entre uma geração e outra de computadores (veja mais adiante quais são estas gerações) e uma diminuição do tempo de processamento de um fator 10 a cada 5 anos.

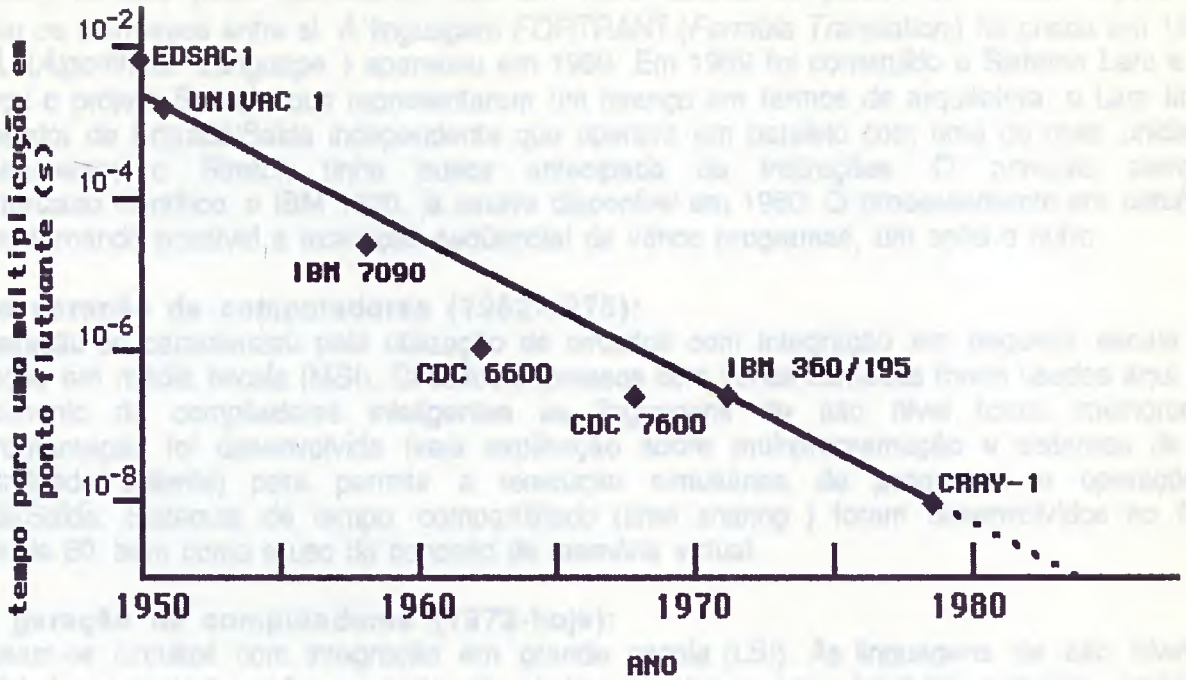


figura 1: a história da velocidade de processamento dos computadores desde 1950, mostrando um crescimento de um fator 10 a cada 5 anos

As diversas divisões, ou gerações de computadores, são determinadas levando-se em conta vários fatores tais como: a tecnologia dos computadores (válvulas, circuitos VLSI); a arquitetura dos sistemas; o modo de processamento e a linguagem que eles usam. Vale observar que estas gerações não têm realmente um limite bem definido, havendo uma certa sobreposição, o que é natural neste tipo de classificação, como mostrado na descrição abaixo.

Hoje estamos na quarta geração e já existem trabalhos em direção à construção dos computadores de quinta geração.

Vejamos a seguir uma descrição de cada uma delas:

- **Primeira geração de computadores (1938-1953):**
  a introdução do primeiro computador eletrônico analógico (1938) e do primeiro computador eletrônico digital (1953) marcam o começo da primeira geração de computadores. Relês eram usados como chaves na década de 40 e válvulas na de 50, sendo que a ligação entre elas era feita com fios. A linguagem era binária (linguagem de máquina) até a introdução do EDVAC (*Electronic Discrete Variable Automatic Computer* em 1950) que conseguia armazenar seus próprios programas, usando programas de sistema que amenizavam um pouco o fato de que se tinha que programar em linguagem de máquina.

- **Segunda geração de computadores (1952-1963):**
  aqui os transistores (inventados em 1948) e os diodos eram os elementos fundamentais na construção dos computadores. O primeiro computador digital transistorizado, TRADIC (*Transistorized Digital*

Computer), foi feito pelos laboratórios Bell em 1954, usando-se placas de circuito impresso para conectar os elementos entre si. A linguagem FORTRAN† (*Formula Translation*) foi criada em 1956 e a ALGOL (*Algorithmic Language*) apareceu em 1960. Em 1959 foi construído o Sistema Larc e a IBM começou o projeto Stretch, que representaram um avanço em termos de arquitetura: o Larc tinha um processador de Entrada/Saída independente que operava em paralelo com uma ou mais unidades de processamento; o Stretch tinha busca antecipada de instruções. O primeiro computador transistorizado científico, o IBM 1620, já estava disponível em 1960. O processamento em *batch* já era comum, tornando possível a execução seqüencial de vários programas, um após o outro.

- **Terceira geração de computadores (1962-1975):**
  esta geração se caracterizou pela utilização de circuitos com integração em pequena escala (SSI) e integração em média escala (MSI). Circuitos impressos com várias camadas foram usados aqui. Com o aparecimento de compiladores inteligentes as linguagens de alto nível foram melhoradas. A multiprogramação foi desenvolvida (veja explicação sobre multiprogramação e sistemas de tempo compartilhado adiante) para permitir a execução simultânea de programas e operações de Entrada/Saída. Sistemas de tempo compartilhado (*time sharing*) foram desenvolvidos no final da década de 60, bem como o uso do conceito de memória virtual.

- **Quarta geração de computadores (1972-hoje):**
  nela usam-se circuitos com integração em grande escala (LSI). As linguagens de alto nível foram extendidads para tratar não somente de dados escalares mas também vetoriais (usados em processadores vetoriais). A maioria dos sistemas operacionais usa tempo compartilhado e memória virtual. Já existem vários multiprocessadores a partir de 1980.

- **Quinta geração de computadores (futuro):**
  circuitos com intregração em escala muito grande (VLSI), que comportarão mais do que 10^7 transistores individuais, serão utilizados nos computadores do futuro. Espera-se que eles consigam chegar à velocidade de cálculo da ordem de 10 bilhões de instruções em ponto flutuante por segundo, ou seja, 10 Gigaflops (FLOPS = *Floating Point Instructions per Second*).

Como é possível notar, a necessidade cada vez maior de mais recursos computacionais (veja algumas áreas de aplicação adiante) faz com que sejam desenvolvidos computadores cada vez mais rápidos. Porém há um limite para a velocidade (de cálculo) que é possível obter-se com uma dada tecnologia e então se busca outra tecnologia. Mas, por trás de tudo há as inexoráveis leis da Física, que impõem um limite [2] até mesmo para as tecnologias que estão por vir.

Assim, com a tecnologia disponível, como aumentar a velocidade dos novos computadores? A resposta está no uso de arquiteturas mais adequadas, que explorem paralelismo. Por esta razão o paralelismo é tão importante.

Vejamos algumas áreas que necessitam de computadores rápidos, ótimas candidatas a usarem os computadores paralelos.

---

† A linguagem FORTRAN (FORmula TRANslation), implementada pela primeira vez em 1956, foi desenvolvida para a solução de problemas matemáticos. Foi liberada aos consumidores pela IBM Corporation em 1957. Nos anos entre 1957 e 1966 ela sofreu várias alterações e foi padronizada em 1966 pelo American National Standards Institute (ANSI). Esta versão padrão de 1966, tecnicamente conhecida como FORTRAN 66, era a quarta versão do FORTRAN, daí o nome FORTRAN IV. Nos anos seguintes, até 1977, a linguagem evoluiu e várias extensões foram adicionadas aos compiladores. Em 1977 a linguagem sofreu novamente uma padronização pelo ANSI e em 1978 surgiu o FORTRAN 77.

Aplicações do processamento paralelo

As necessidade que levaram à criação de computadores com processamento paralelo foram basicamente aquelas de execução de tarefas bastante complexas e que geralmente requerem muito tempo de cálculo, tais como cálculos em Teorias na Rede, determinação do tamanho de bacias petrolíferas, simulações etc.

Temos abaixo uma lista que, longe de estar completa, serve para ilustrar algumas aplicações dos computadores paralelos:

- Modelos de Previsão e Simulação
  - ¤ Meteorologia: previsão do tempo
  - ¤ Oceanografia: exploração dos recursos do oceano
  - ¤ Astrofísica: estudo da formação das galáxias
  - ¤ Socio-economia: controle do crime
  - ¤ Física: simulação de aceleradores de partículas

- Projetos de Engenharia e Automação
  - ¤ Cálculos por elementos finitos: barragens, espaçonaves
  - ¤ Aerodinâmica: estudo de turbulências
  - ¤ Inteligência Artificial: processamento de imagens, visão por computador, sistemas especialistas

- Exploração de fontes de energia
  - ¤ exploração sísmica: determinação de poços de petróleo
  - ¤ segurança de reatores nucleares

- Pesquisas médicas
  - ¤ tomografia: modelagem tri-dimensional do corpo
  - ¤ Engenharia Genética: pesquisa sobre mutações,

- Pesquisa Básica
  - ¤ Pesquisa Básica em Física (Mecânica Quântica e Estatística, Dinâmica dos Fluidos), Química (química de polímeros), Engenharia Eletrônica (distribuição de conexões em circuitos VLSI)

- Entretenimento
  - ¤ Animação: geração de imagens para cinema e televisão

- Treinamento
  - ¤ Navegação: simuladores de vôo (aviões, espaçonaves) e de navegação (navios)

Paralelismo em sistemas com apenas 1 processador

Por mais paradoxal que possa parecer a princípio, o paralelismo em sistemas com apenas um processador não somente existe como é muito importante. Ele pode ser obtido de várias maneiras:

- **Multiplicidade de unidades funcionais:**
  aqui constrói-se a Unidade Lógica Aritmética (ULA), presente nas UCP's, de forma que ela contenha várias unidades funcionais independentes, como por exemplo uma para adição e outra para multiplicação de dois números

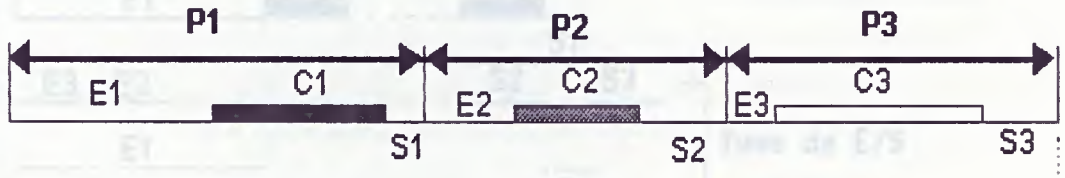
- **Paralelismo e *pipeline* dentro da UCP:**
  várias fases da execução de uma instrução são executadas em paralelo, como por exemplo busca, decodificação , busca de operandos, execução e armazenamento dos resultados

- **Operações da UCP e de E/S sobrepostas (no tempo):**
  enquanto a UCP estiver ocupada (com cálculos, por exemplo) as operações de Entrada/Saída poderão estar sendo realizadas desde que haja um controlador para isto

- **Uso de sistemas hierárquicos de memória:**
  a UCP é normalmente muito mais rápida do que um acesso à memória. Um sistema hierárquico de memória serve para diminuir esta defasagem nos tempos. A idéia básica é fazer com que a UCP sempre "veja" uma memória rápida, mesmo que para isto seja preciso colocar vários estágios de sistemas. Como exemplo, temos, em ordem decrescente de velocidades de acesso: registradores rápidos, *caches*, RAM (*Ramdom Access Memory*), discos óticos e magnéticos com cabeças fixas e memória de bolha, discos óticos e magnéticos com cabeças móveis e, finalmente, unidades de fitas magnéticas

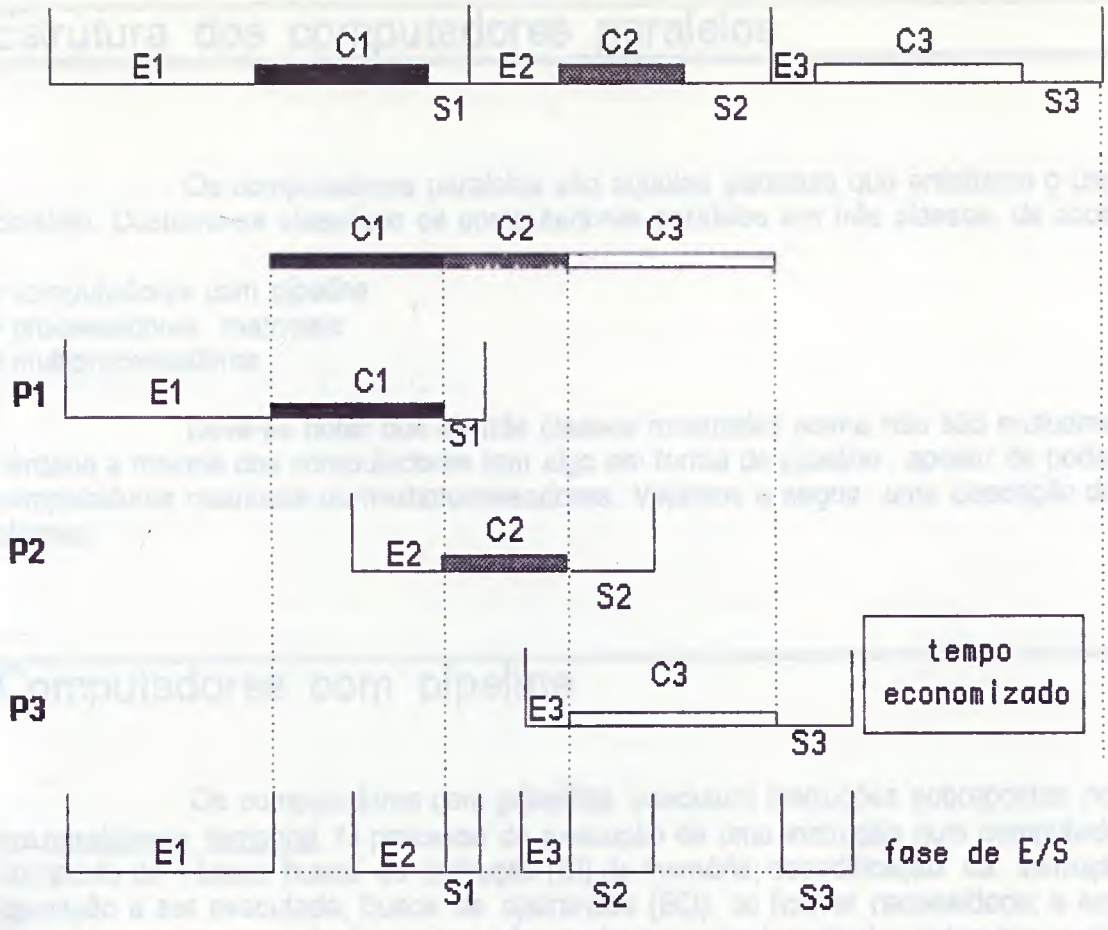
- **Multiprogramação e Compartilhamento de Tempo (*Time Sharing*):**
  ao contrário dos itens mostrados acima, multiprogramação e compartilhamento de tempo são recursos em programa (não em *hardware*) para se conseguir o processamento paralelo em máquinas com apenas um processador. Na multiprogramação, sempre que um programa precisa fazer alguma operação de Entrada/Saída, a UCP é alocada para um outro processo que dela necessitar. Mas, pode acontecer que um determinado programa use muito a UCP e isto forçará os outros programas a terem que esperar muito tempo para serem executados, mesmo que eles fossem usar a UCP por muito pouco tempo. Assim, para deixar tudo um pouco mais justo, criou-se o conceito de compartilhamento de tempo: cada programa tem o direito de usar a UCP por um determinado período de tempo (igual para todos)

Os ganhos em tempo de execução para o processamento normal (em *batch*), multiprogramação e tempo compartilhado podem ser vistos nas figura 2a, 2b e 2c, respectivamente, mostradas a seguir, onde C_i representa a execução (Cálculo) de um programa P_i; E_i representa a Entrada do Programa P_i e S_i representa a Saída de dados do Programa P_i:

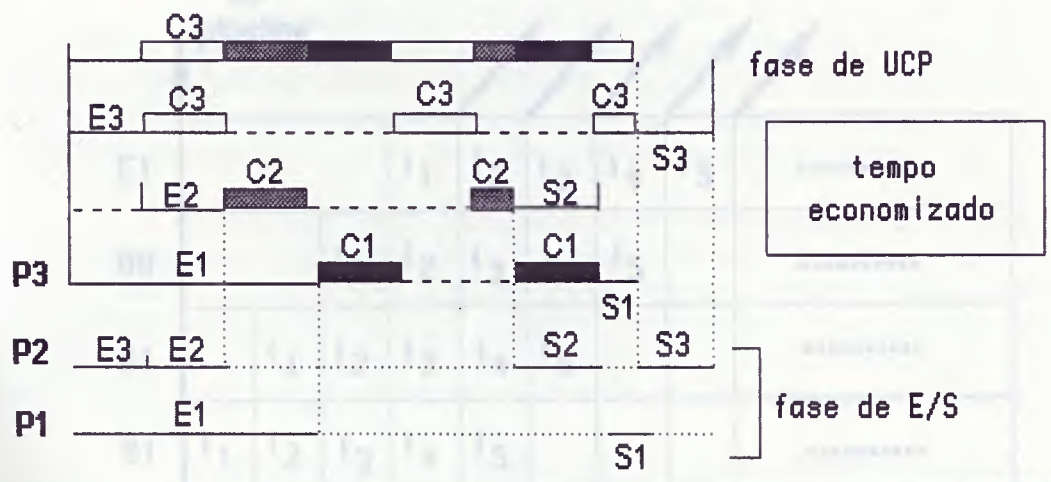


a) processamento em batch

figura 2: abordagens para se obter multiprocessamento com apenas 1 processador, usando (a) processamento em batch



b) multiprogramação



c) tempo compartilhado

figura 2 (continuação): abordagens para se obter multiprocessamento com apenas 1 processador, usando (b) multiprogramação, (c) compartilhamento de tempo

Estrutura dos computadores paralelos

Os computadores paralelos são aqueles sistemas que enfatizam o uso de processamento paralelo. Costuma-se classificar os computadores paralelos em três classes, de acordo com sua arquiteturas:

- computadores com *pipeline*
- processadores matriciais
- multiprocessadores

Deve-se notar que as três classes mostradas acima não são mutuamente exclusivas. Na verdade a maioria dos computadores tem algo em forma de *pipeline* , apesar de poder ter estruturas do tipo computadores matriciais ou multiprocessadores. Vejamos a seguir uma descrição de cada uma destas 3 classes.

Computadores com pipeline

Os computadores com *pipeline* executam instruções sobrepostas no tempo, explorando assim um paralelismo temporal. O processo de execução de uma instrução num computador normalmente é composto de 4 fases: busca da instrução (BI) da memória; decodificação da instrução (DI), que identifica a operação a ser executada; busca de operandos (BO), se houver necessidade; e execução da instrução (EI). Num computador sem *pipeline* estas 4 fases devem estar terminadas antes que a próxima instrução possa ser executada. Já num computador com *pipeline* estas 4 fases podem estar sobrepostas (no tempo), como mostrado na figura 3 abaixo:

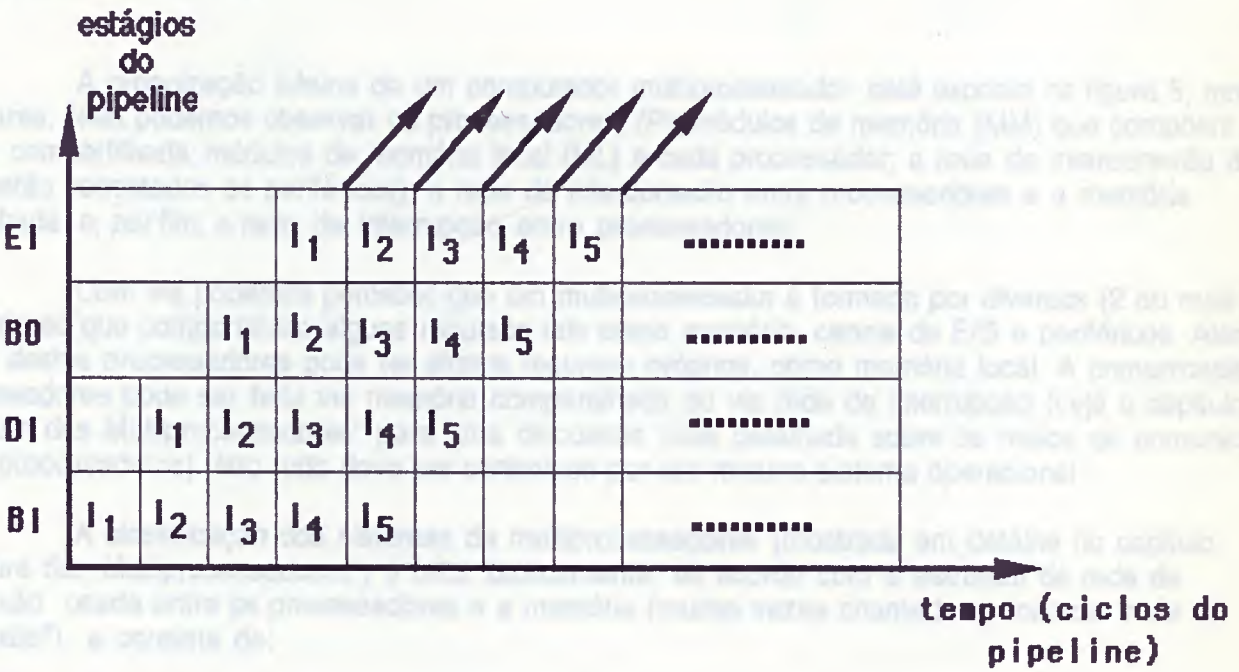


figura 3: diagrama mostrando a execução das instruções num computador com pipeline

Num computador com *pipeline* chama-se de um ciclo de instrução a soma dos vários ciclos do *pipeline* , onde cada ciclo do *pipeline* pode ser definido como o tempo gasto pelo estágio mais lento. Observe que os diversos estágios trabalham em sincronismo, sob controle de um relógio comum.

Um computador sem *pipeline* levaria, no nosso exemplo, 4 ciclos do *pipeline* para completar uma instrução. Note também que, uma vez que o *pipeline* estiver cheio haverá um resultado a cada ciclo do *pipeline* .

Processadores matriciais

Um processador matricial, também conhecido por ***array processor***, é um computador com várias unidades lógicas aritméticas (ULA's) que trabalham em paralelo, em sincronismo isto é, todas ao mesmo tempo e que estão conectadas umas às outras por meio de uma rede de interconexão. Com estas várias ULA's consegue-se um paralelismo espacial, diferente do paralelismo temporal obtido com os computadores com *pipeline* . Uma característica muito importante dos computadores matriciais são exatamente estas redes de interconexão, que podem assumir formas diferentes dependendo do problema que se quer resolver, como mostrado na figura 4 na próxima página. Como nosso interesse maior não se concentra nem nos computadores com *pipeline* nem nos processadores matriciais, e sim nos multiprocessadores, encerramos aqui esta breve introdução a este dois tipos de computadores e nos concentramos nos multiprocessadores. Recomendo ao leitor mais interessado naqueles uma consulta à referência *Parallel Computers* [3].

Multiprocessadores

A organização básica de um computador multiprocessador está exposta na figura 5, mostrada mais adiante. Nela podemos observar os processadores (P); módulos de memória (MM) que compõem a memória compartilhada; módulos de memória local (ML) a cada processador; a rede de interconexão de E/S (à qual estão conectados os periféricos); a rede de interconexão entre processadores e a memória compartilhada e, por fim, a rede de interrupção entre processadores.

Com ela podemos perceber que um multiprocessador é formado por diversos (2 ou mais) processadores que compartilham alguns recursos tais como memória, canais de E/S e periféricos. Além disto, cada um destes processadores pode ter alguns recursos próprios, como memória local. A comunicação entre os processadores pode ser feita via memória compartilhada ou via rede de interrupção (veja o capítulo "Arquitetura dos Multiprocessadores" para uma discussão mais detalhada sobre os meios de comunicação entre os processadores). Isto tudo deve ser controlado por um mesmo sistema operacional.

A classificação dos sistemas de multiprocessadores (mostrada em detalhe no capítulo "Arquitetura dos Multiprocessadores") é feita, basicamente, de acordo com a estrutura da rede de interconexão usada entre os processadores e a memória (muitas vezes chamada apenas de "rede de interconexão") e consiste de:

- barramento comum de tempo compartilhado
- rede tipo *crossbar*
- memórias multiportas

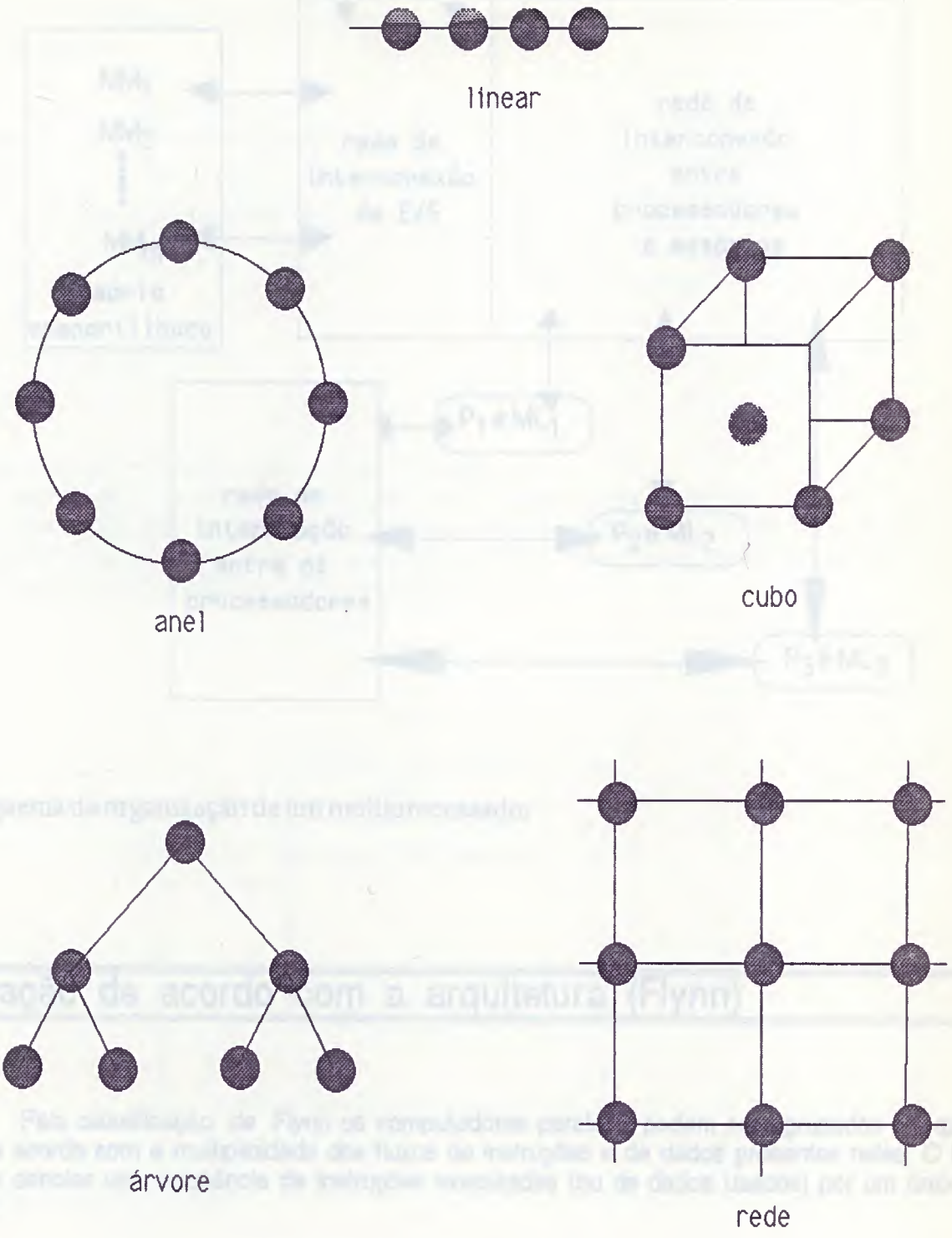


figura 4: diversas topologias para redes de interconexão. Os círculos representam os elementos processadores e as linhas representam as ligações entre eles

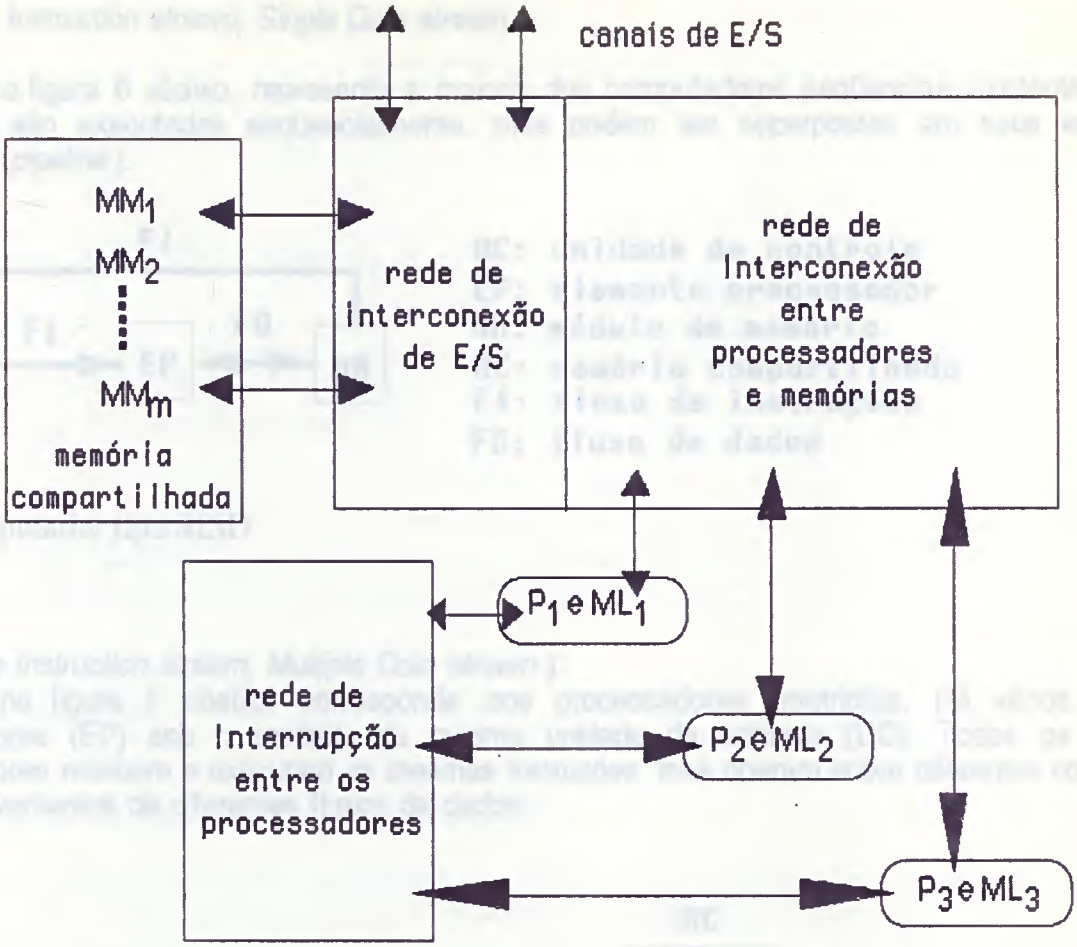


figura 5: esquema da organização de um multiprocessador

Classificação de acordo com a arquitetura (Flynn)

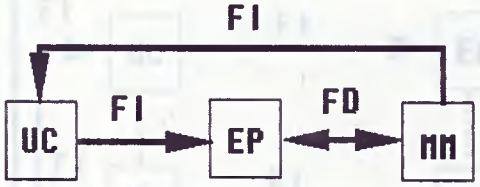
Pela classificação de Flynn os computadores paralelos podem ser agrupados em quatro categorias de acordo com a multiplicidade dos fluxos de instruções e de dados presentes neles. O termo fluxo é usado para denotar uma seqüência de instruções executadas (ou de dados usados) por um único processador:

- um **fluxo de instruções** é uma seqüência de instruções executadas por uma determinada máquina
- um **fluxo de dados** é uma seqüência de dados usada pelo fluxo de instruções

As organizações dos computadores estão baseadas na multiplicidade do *hardware* disponível para a execução dos fluxos de instruções e de dados, e são as seguintes:

• SISD (*Single Instruction stream, Single Data stream*):

mostrada na figura 6 abaixo, representa a maioria dos computadores seqüenciais existentes hoje. As instruções são executadas seqüencialmente, mas podem ser superpostas em seus estágios de execução (*pipeline*).



UC: unidade de controle
EP: elemento processador
MM: módulo de memória
MC: memória compartilhada
FI: fluxo de instruções
FD: fluxo de dados

figura 6: computador tipo SISD

• SIMD (*Single Instruction stream, Multiple Data stream*):
mostrada na figura 7 abaixo, corresponde aos processadores matriciais. Há vários elementos processadores (EP) sob o controle da mesma unidade de controle (UC). Todos os elementos processadores recebem e executam as mesmas instruções, mas operam sobre diferentes conjuntos de dados provenientes de diferentes fluxos de dados.

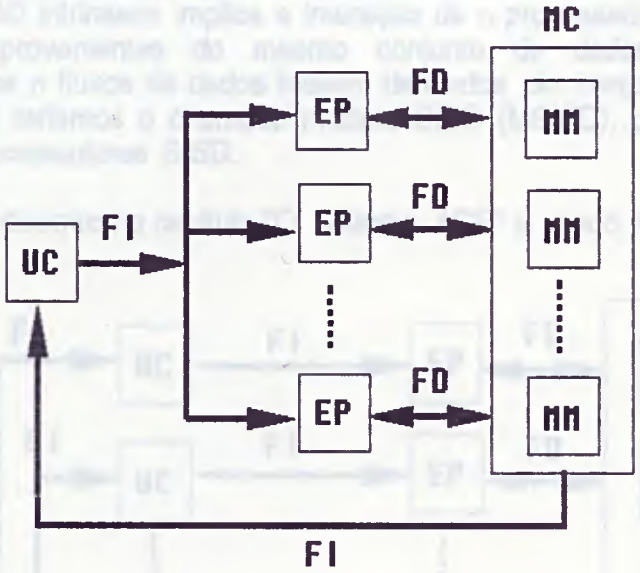


figura 7: computador tipo SIMD

• MISD (*Multiple Instruction stream, Single Data stream*):

mostrada na figura 8 a seguir, não representa nenhum computador existente. Há n processadores que recebem, cada um, diferentes instruções a serem executadas sobre o mesmo conjunto de dados e seus derivados. O resultado fornecido por um processador passa a ser a entrada do próximo processador.

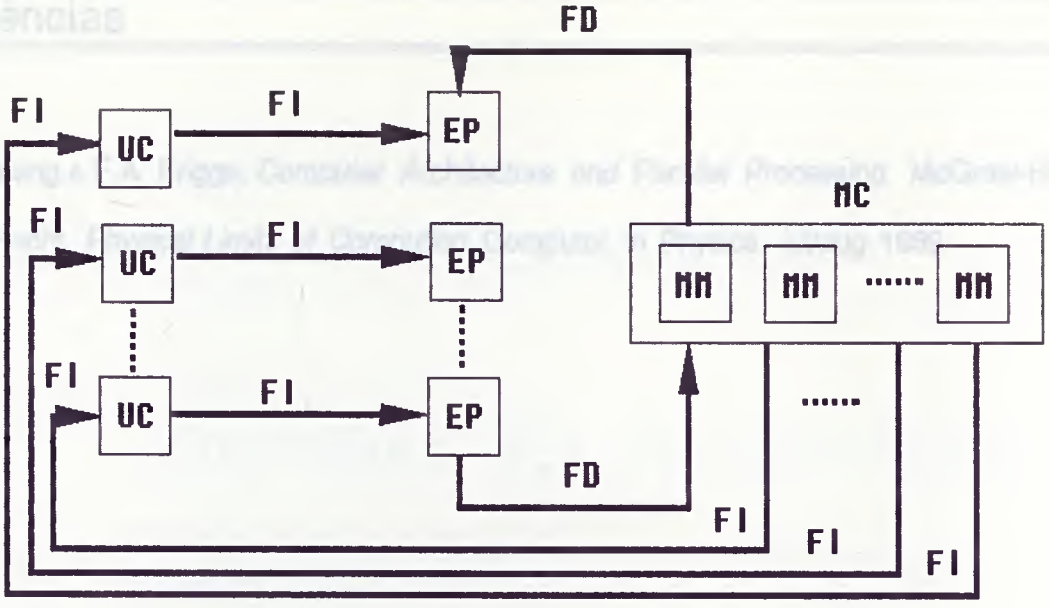


figura 8: computador tipo MISD

• MIMD (*Multiple Instruction stream, Multiple Data stream*):

mostrada na figura 9 abaixo, representa a maioria dos sistemas de multiprocessadores . Um computador tipo MIMD intrínseco implica a interação de n processadores entre si, pois todos os fluxos da memória são provenientes do mesmo conjunto de dados compartilhado por todos os processadores. Se os n fluxos de dados fossem derivados de conjuntos independentes da memória compartilhada, então teríamos o chamado múltiplo SISD (MSISD), que não é nada mais do que um conjunto de n uniprocessadores SISD.

O computador ACP, descrito no capítulo "O Sistema ACP" e usado neste trabalho, é do tipo MIMD.

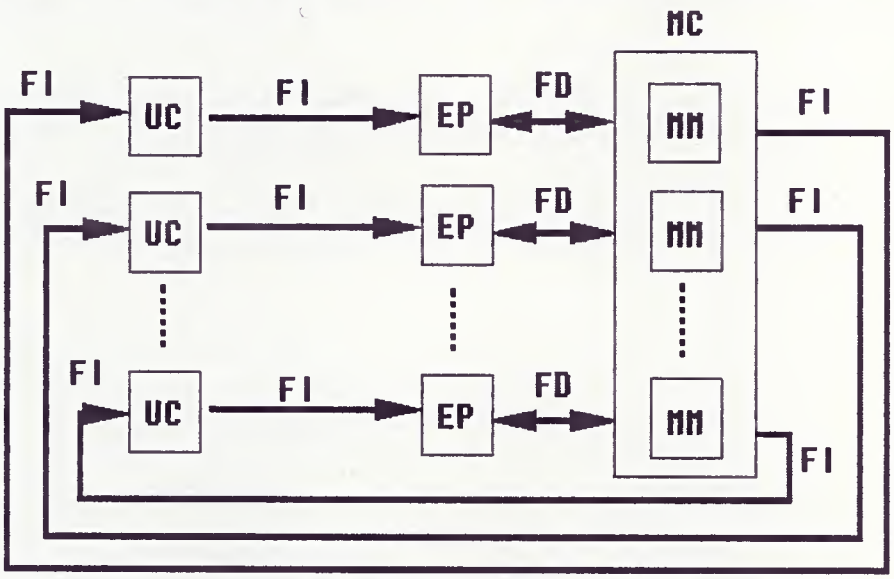


figura 9: computador tipo MIMD

Referências

[1] K. Hwang e F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1986

[2] Jeff Hecht, *Physical Limits of Computing*, Computer in Physics, jul/aug 1989

capítulo 2

Arquitetura dos Multiprocessadores

Arquitetura dos Multiprocessadores

Introdução

Este capítulo descreve os tipos de arquiteturas dos sistemas de multiprocessadores e os diferentes modos de se fazer as conexões entre os dispositivos funcionais, tais como processadores, memórias etc.

Os multiprocessadores são classificados e o problema do compartilhamento de dados entre os diversos processadores de um computador multiprocessador é abordado.

O sistema multiprocessador ACP, usado neste trabalho e descrito em detalhes no capítulo "O Sistema ACP", será mencionado na medida em que seus atributos forem sendo abordados, facilitando assim a sua classificação.

Multiprocessadores

Pretendo discutir várias maneiras de se interligar os processadores num sistema de multiprocessadores MIMD [1].

Antes de começar a descrever os sistemas de multiprocessadores [2], [3] vale a pena ressaltar a diferença entre dois tipos de processamento paralelo muito confundidos:

- **multicomputador** : é aquele sistema no qual cada elemento processador corresponde a um computador autônomo (com todas as suas características)
- **multiprocessador** : é aquele sistema no qual cada elemento processador é composto de apenas um processador (e outras unidades funcionais como memória local etc, explicadas adiante)
  e é controlado por um único sistema operacional

As principais diferenças entre eles estão na forma como são compartilhados os recursos e na cooperação na resolução dos problemas:

- num sistema multicomputador existem vários computadores autônomos que podem se comunicar entre si e que estão sob os cuidados dos respectivos sistemas operacionais

- nos sistemas multiprocessadores há somente um sistema operacional controlando todos os processadores e que fornece um ambiente adequado para a comunicação entre eles. O sistema ACP é um multiprocessador

Chamaremos de nó o conjunto formado pelo processador, memória local e interface de Entrada/Saída, presentes nos multiprocessadores. Um multiprocessador pode ser caracterizado, basicamente, por dois atributos:

- um multiprocessador é um único computador que contém vários processadores

- os processadores, ou nós, podem se comunicar entre si, em vários níveis, a fim de resolver um determinado problema. Esta comunicação pode ser feita através de áreas de memória compartilhada (comuns a todos os nós) ou através do envio de mensagens de um nó a outro, o que obviamente irá requerer algum sistema de transmissão de mensagens

Pode-se, de um modo geral, classificar os sistemas de multiprocessadores em duas categorias, de acordo com o modo como os nós se comunicam: multiprocessadores fracamente acoplados e multiprocessadores fortemente acoplados:

- num sistema de multiprocessadores fracamente acoplados a comunicação entre os diversos nós é feita pelo envio de mensagens através de um sistema de transmissão.

- já num sistema de multiprocessadores fortemente acoplados os nós se comunicam entre si através de uma memória central compartilhada, sendo que pode haver uma rede de interconexão ligando os nós à memória ou uma memória multiporta. Uma evidente limitação dos sistemas fortemente acoplados é a largura de banda da memória, que tende a ser o gargalo de todo o sistema quando o número de nós cresce muito, pois aumenta também o número de colisões quando do acesso à memória. O sistema ACP é fortemente acoplado.

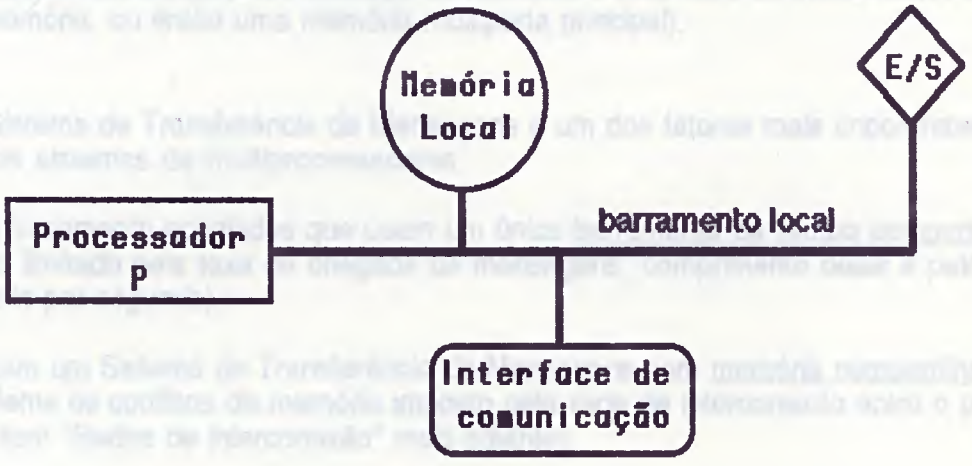
Multiprocessadores fracamente acoplados

Nos sistemas fracamente acoplados cada processador tem uma interface de Entrada/Saída e uma memória local suficientemente grande para conter o programa a ser executado e os dados a serem usados.

A comunicação entre os processos que são executados nos diferentes nós é efetuada através de um Sistema de Transferência de Mensagens.

Os sistemas fracamente acoplados são mais eficientes quando as interações entre os processos são pequenas.

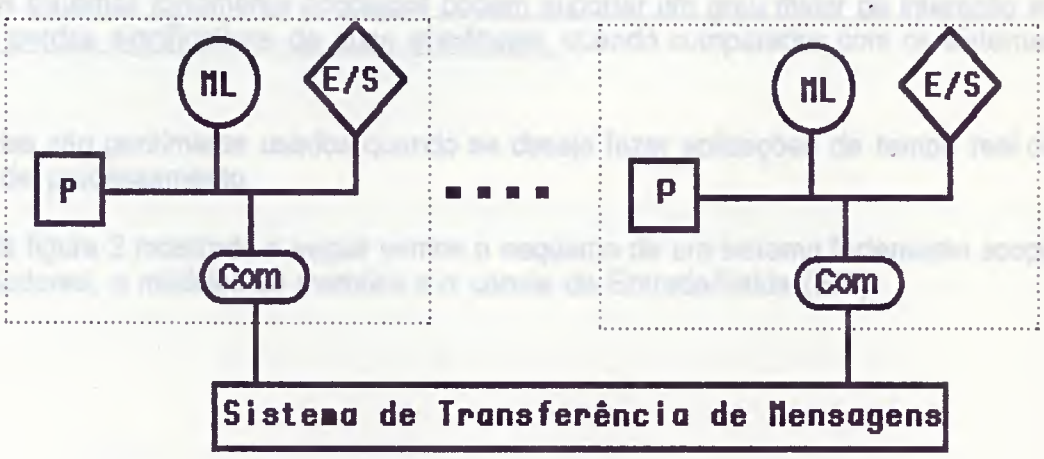
A figura 1-a abaixo, mostra um esquema de um nó de um sistema de multiprocessadores, que consiste de um processador, de uma memória local, de dispositivos de Entrada/Saída locais e de uma interface de comunicação com os outros nós (que contém uma chave e uma unidade de arbitramento).



a)

figura 1: (a) nó de um sistema multiprocessador

A figura 1-b abaixo mostra a conexão entre os nós e o Sistema de Transferência de Mensagens.



b)

figura 1: (b) conexão entre os nós e o Sistema de Transferência de Mensagens

Se dois ou mais pedidos ao Sistema de Transferência de Mensagens chegarem ao mesmo tempo, a unidade de arbitramento escolherá um dos nós de acordo com uma política de prioridades previamente determinada.

O Sistema de Transferência de Mensagens num sistema fracamente acoplado pode ser um simples barramento de tempo compartilhado ou um sistema de memória compartilhada que pode ser implementada usando-se um conjunto de módulos de memória e uma rede de interconexão entre o processador e a memória, ou então uma memória multiporta principal).

O Sistema de Transferência de Mensagens é um dos fatores mais importantes na determinação do desempenho dos sistemas de multiprocessadores:

- para os sistemas fracamente acoplados que usam um único barramento de tempo compartilhado o desempenho está limitado pela taxa de chegada de mensagens, comprimento delas e pela capacidade do barramento (em bits por segundo).

- para os que utilizam um Sistema de Transferência de Mensagens com memória compartilhada o fator limitante é o problema de conflitos de memória imposto pela rede de interconexão entre o processador e a memória (veja o item "Redes de interconexão" mais adiante).

Multiprocessadores fortemente acoplados

Nos sistemas fortemente acoplados os processadores se comunicam através de uma memória central compartilhada.

Os sistemas fortemente acoplados podem suportar um grau maior de interação entre os processos, sem perdas significativas de suas eficiências, quando comparados com os sistemas fracamente acoplados.

Eles são geralmente usados quando se deseja fazer aplicações de tempo real ou que exigem alta velocidade de processamento.

Na figura 2 mostrada a seguir vemos o esquema de um sistema fortemente acoplado, onde temos p processadores, m módulos de memória e d canais de Entrada/Saída (E/S).

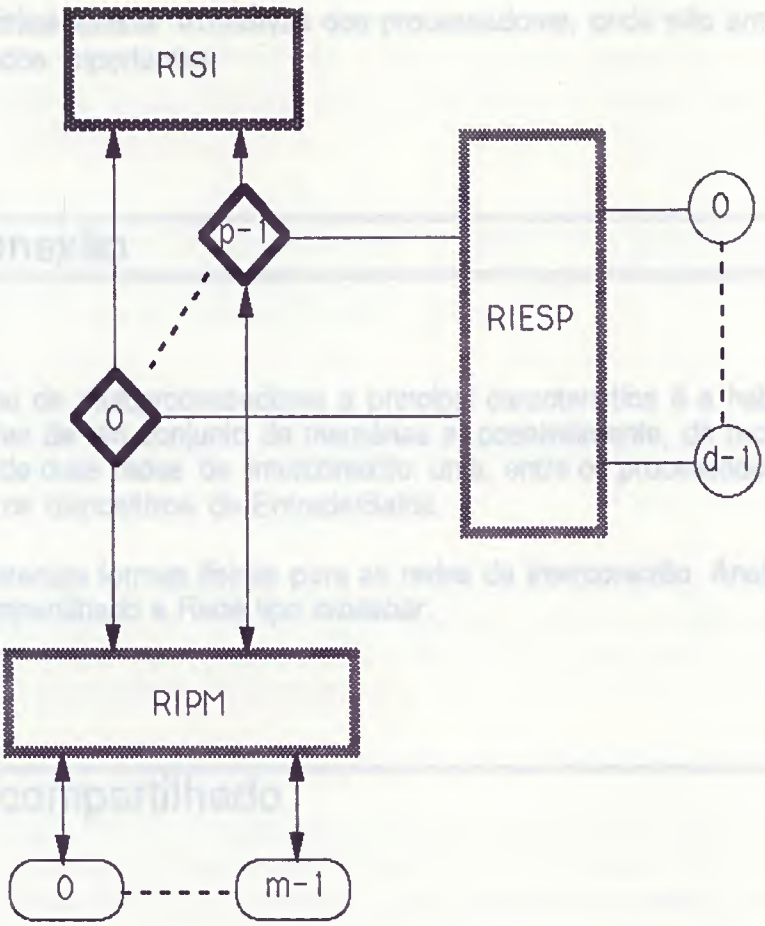


figura 2: multiprocessador fortemente acoplado

Estas unidades estão ligadas entre si por três redes de interconexão:

- Rede de Interconexão entre Processador e Memória (RIPM),
- Rede de Interconexão entre E/S e Processadores (RIESP),
- Rede de Interconexão de Sinal de Interrupção(RISI).

A RIPM é uma chave que pode conectar cada processador a cada um dos módulos de memória. Tipicamente é uma *crossbar* $p \times m$, com $p.m$ cruzamentos. Para cada um dos $p.m$ cruzamentos temos $(n+k)$ outros cruzamentos, onde n é a largura do endereço (em bits) e k é a largura da via de dados (em bits também). Assim, vemos que a complexidade desta *crossbar* é $O(p.m.(n+k))$. Se houver um grande número de processadores e um grande número de módulos de memória, o custo do *crossbar* poderá dominar o custo do sistema.

Um módulo de memória pode atender a somente um processador num dado ciclo de memória. Assim sendo, se dois ou mais processadores precisarem acessar o mesmo módulo de memória ao mesmo tempo haverá um **conflito**, que é resolvido pela RIPM. Se for necessário a RIPM poderá ser projetada para permitir o ***broadcasting*** (envio de uma mesma mensagem de uma fonte para vários destinos) de dados de um processador para vários módulos de memória. Uma outra alternativa para minimizar os conflitos de

memória é o uso de **memórias locais** exclusivas dos processadores, onde são armazenados o programa a ser executado e alguns dados importantes.

Redes de interconexão

Num sistema de multiprocessadores a principal característica é a habilidade de cada processador em compartilhar de um conjunto de memórias e, possivelmente, de recursos de Entrada/Saída. Isto é feito através do uso de duas redes de interconexão: uma, entre os processadores e a memória e a outra entre os processadores e os dispositivos de Entrada/Saída.

Existem diferentes formas físicas para as redes de interconexão. Analisaremos duas delas a seguir: Rede de tempo compartilhado e Rede tipo *crossbar*.

Rede de tempo compartilhado

O sistema de interconexão mais simples consiste de um barramento único de comunicação, ao qual estão conectadas todas as unidades funcionais, tais como processadores, memórias, interface de E/S (veja figura 3 abaixo):

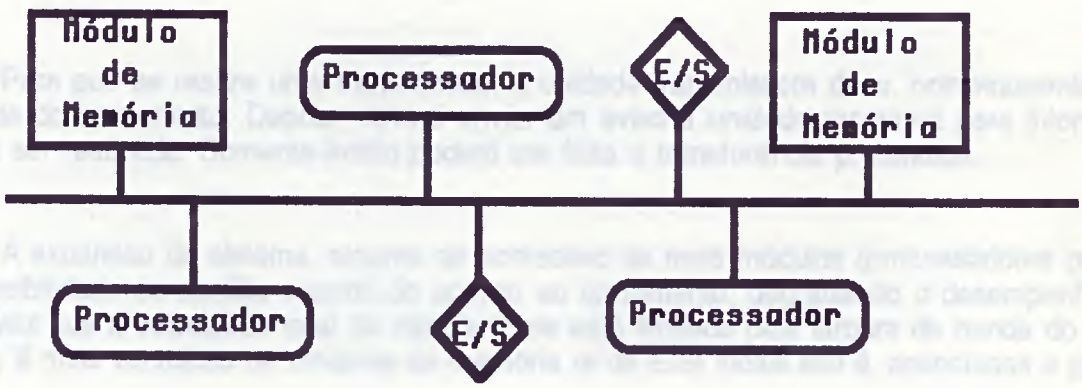


figura 3: barramento único

As operações de transferência são totalmente controladas pelas interfaces das unidades receptora e transmissora.

Uma vez que o barramento é compartilhado entre todas as unidades, deve haver um meio de se fazer um arbitramento para resolver problemas de conflito. Para isto podem ser usados vários **métodos de prioridades** , tais como: estática, fila tipo FIFO (*First In, First Out*) ou ***Daisy-Chain*** . Vemos abaixo alguns dos principais esquemas de prioridade:

- **prioridades estáticas:**
  cada dispositivo que pode solicitar o uso do barramento recebe uma prioridade que não se altera no decorrer do tempo (ou seja, estática). Quando mais de um dispositivo requerem o uso do barramento, o de prioridade mais alta é o que recebe autorização para usá-lo.

- ***Daisy-Chain*** **:**
  atribui prioridades (fixas) de acordo com a localização do dispositivo ao longo do barramento.

- **Fatias de tempo:**
  outro esquema muito comum para se fazer o arbitramento quanto ao uso do barramento, que divide a largura de banda do barramento em fatias de igual duração e oferece cada uma destas fatias a um dispositivo; se este não precisar usar o barramento então o barramento não poderá ser usado durante aquele período de tempo por nenhum outro dispositivo.

- **Prioridade dinâmica:**
  atribui, inicialmente, uma prioridade arbitrária para cada dispositivo. O de mais alta prioridade, como é de se esperar, é o que pode usar o barramento. Porém, as prioridades dos dispositivos são alteradas no decorrer do tempo, dando assim oportunidade a cada dispositivo de poder usar o barramento.

- **FIFO (*First In, First Out*):**
  aqui, os pedidos de acesso ao barramento são aprovados de acordo com suas ordens de chegada.

Para que se realize uma transmissão, a unidade transmissora deve, primeiramente, determinar a disponibilidade do barramento. Depois, deverá enviar um aviso à unidade receptora para informá-la sobre a transferência a ser realizada. Somente então poderá ser feita a transferência pretendida.

A expansão do sistema, através do acréscimo de mais módulos (processadores ou memória), aumenta a possibilidade de conflito quando do acesso ao barramento, degradando o desempenho de todo o sistema, uma vez que a velocidade total de transferência está limitada pela largura de banda do barramento. Por esta razão, é mais vantajoso ter módulos de memória (e de E/S) locais isto é, associados a cada processador .

O próximo passo para se tentar resolver este tipo de problema é ter vários barramentos bi-direcionais, como mostrado na figura 4 a seguir, que permitem múltiplas transferências simultâneas. Entretanto, isto aumenta de maneira considerável a complexidade do sistema.

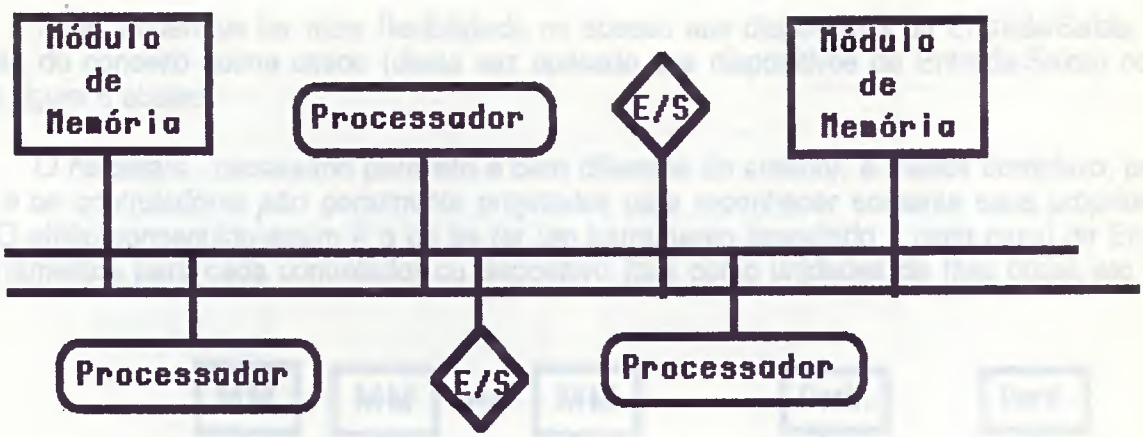


figura 4: múltiplos barramentos

Rede tipo crossbar

Se aumentarmos continuamente o número de barramentos do esquema anterior, chegaremos a um ponto no qual haverá um barramento para cada módulo de memória, como mostrado na figura 5 abaixo. A rede de conexão que irá resultar é chamada de *crossbar* .

A característica mais importante de uma rede tipo *crossbar* é a simplicidade das interfaces das unidades funcionais e a possibilidade de se fazer transferências simultâneas para todos os módulos de memória. Fica claro que estas características irão gerar um aumento na complexidade física das chaves, pois elas deverão ser capazes de resolver múltiplos pedidos de acesso à memória (geralmente resolvidos com algum tipo de prioridade pré-determinada).

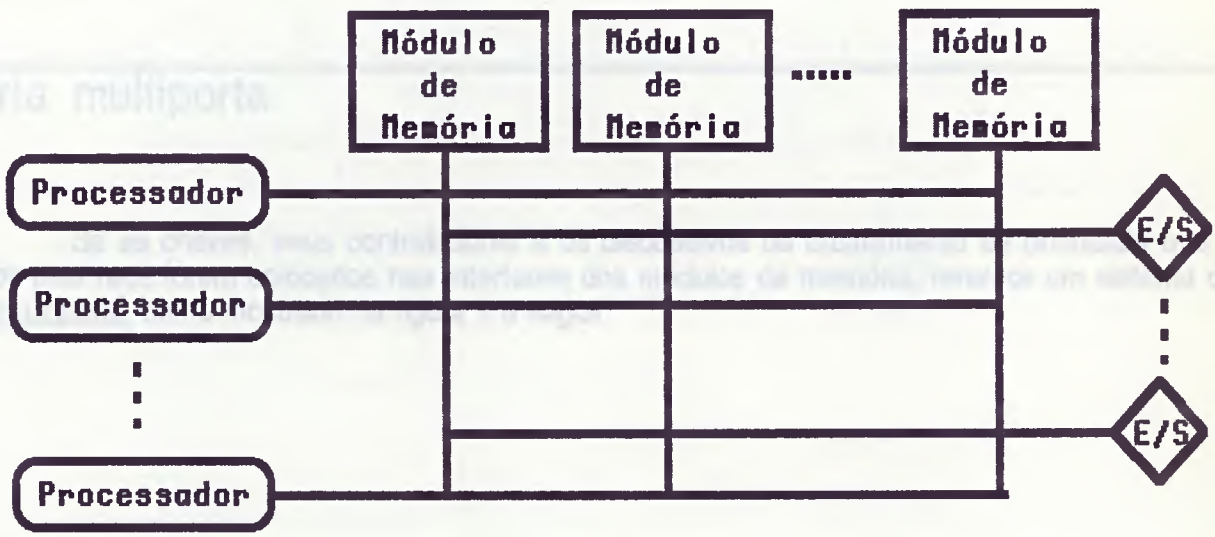


figura 5: rede tipo crossbar

Para podermos ter mais flexibilidade no acesso aos dispositivos de Entrada/Saída, utilizamos uma extensão do conceito acima citado (desta vez aplicado aos dispositivos de Entrada/Saída) como mostrado na figura 6 abaixo.

O *hardware* necessário para isto é bem diferente do anterior, e menos complexo, pois os dispositivos e os controladores são geralmente projetados para reconhecer somente seus próprios endereços. O efeito conseguido assim é o de se ter um barramento associado a cada canal de Entrada /Saída e barramentos para cada controlador ou dispositivo (tais como unidades de fitas óticas etc.).

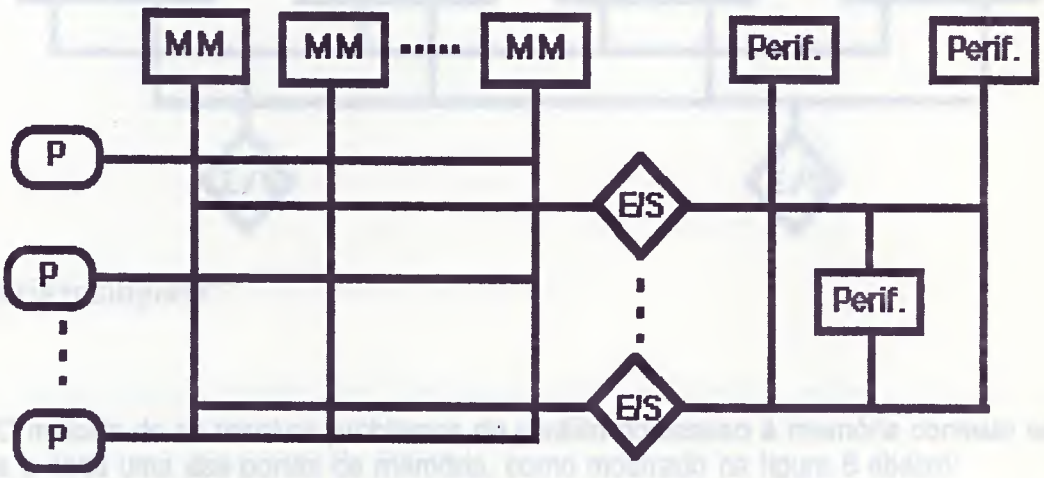


figura 6: crossbar para ligações entre processadores-memórias-E/S

A rede *crossbar* tem o potencial para ser a de melhor desempenho . Porém, por causa de sua complexidade e custo, não é muito adequada para sistemas com um grande número de processadores.

Memória multiporta

Se as chaves, seus controladores e os dispositivos de arbitramento de prioridade que estão distribuídos pela rede forem colocados nas interfaces dos módulos de memória, teremos um sistema de memória multiporta, como mostrado na figura 7 a seguir:

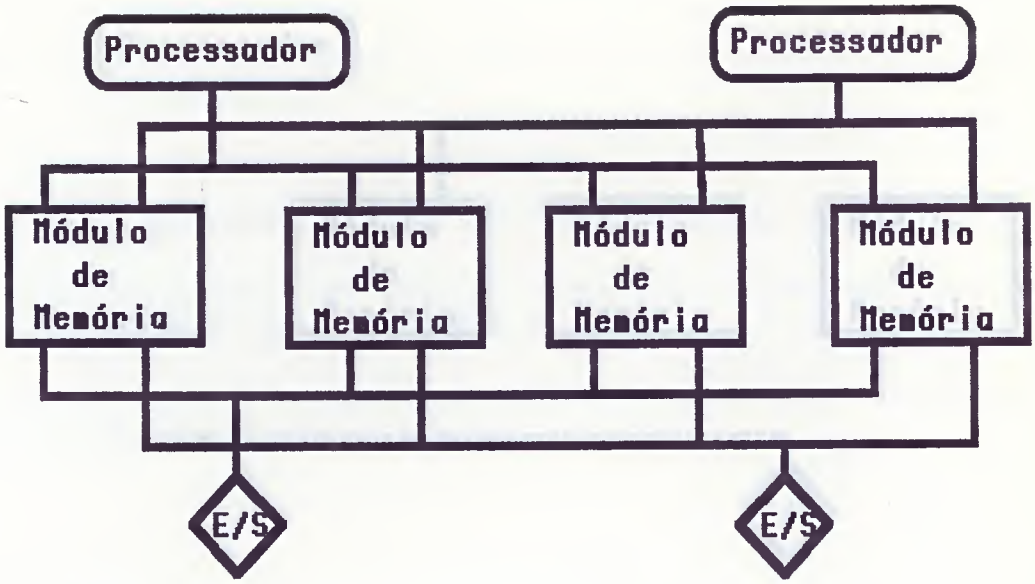


figura 7: memória multiporta

O método de se resolver problemas de conflito no acesso à memória consiste em se atribuir prioridades fixas a cada uma das portas da memória, como mostrado na figura 8 abaixo:

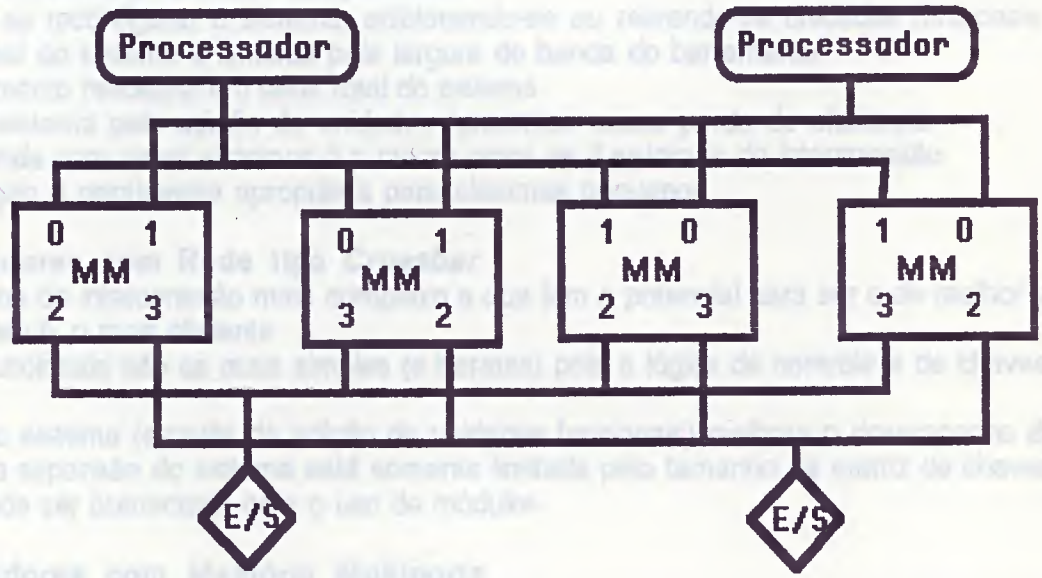


figura 8: prioridade nas portas de uma memória multiporta

A atribuição de prioridades às portas da memória permite que se conceda acesso exclusivo a determinados módulos de memória por um determinado processador, como ilustrado na figura 9 mostrada a seguir:

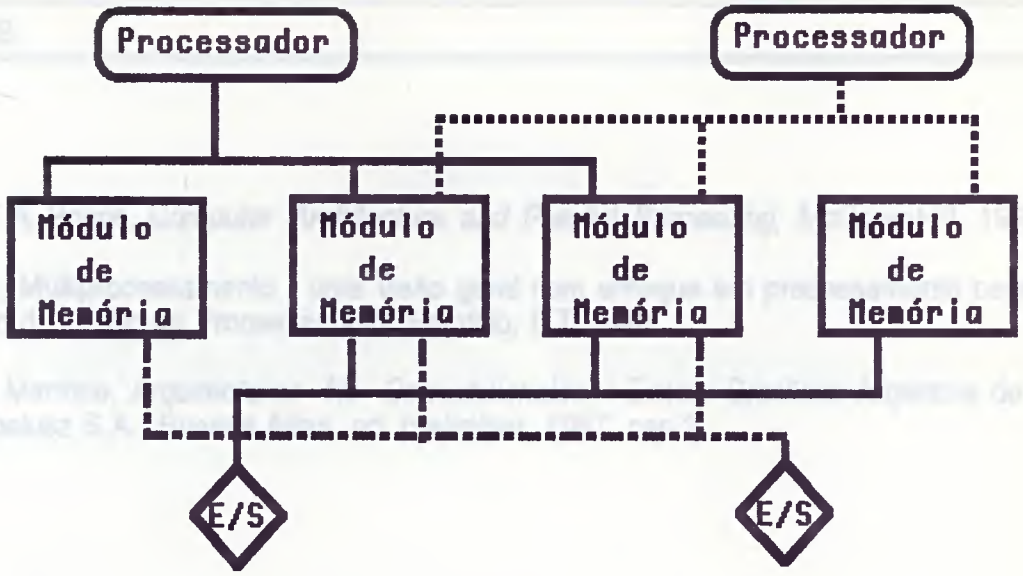


figura 9: acesso exclusivo à memória por um processador

A tabela 1abaixo mostra uma comparação entre as 3 organizações de *hardware* dos multiprocessadores acima exibidas.

Multiprocessadores com Rede de Tempo Compartilhado
- menor custo do *hardware* e menor complexidade
- facilidade em se reconfigurar o sistema, adicionando-se ou retirando-se unidades funcionais
- capacidade total do sistema é limitada pela largura de banda do barramento
- falha do barramento resultaria em falha total do sistema
- expansão do sistema pela adição de unidades funcionais causa perda de eficiência
- a eficiência obtida com estes sistemas é a menor entre os 3 sistemas de interconexão
- esta organização é geralmente apropriada para sistemas pequenos

Multiprocessadores com Rede tipo *Crossbar*
- este é o sistema de interconexão mais complexo e que tem o potencial para ser o de melhor taxa de transmissão isto é, o mais eficiente
- as unidades funcionais são as mais simples (e baratas) pois a lógica de controle e de chaveamento está na rede
- a expansão do sistema (através da adição de unidades funcionais) melhora o desempenho do sistema
- teoricamente a expansão do sistema está somente limitada pelo tamanho da matriz de chaveamento, que geralmente pode ser aumentada com o uso de módulos

Multiprocessadores com Memória Multiporta
- esta arquitetura é a que requer os módulos de memória mais caros, uma vez que devem conter toda a lógica de controle e chaveamento
- as opções de reconfiguração estão limitadas pelo número de portas
- há um potencial para uma grande taxa de transmissão isto é, grande eficiência na comunicação

tabela 1: comparação entre multiprocessadores com rede de tempo compartilhado, rede tipo crossbar e memória multiporta

Referências

[1] K. Hwang e F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1986, cap.7

[2] J.R.F. Xavier, Multiprocessamento - uma visão geral com enfoque em processamento paralelo, Comunicação do Grupo de Processamento Paralelo, IFT, 1987

[3] C. Bogni e L. Marrone, *Arquitecturas No Convencionales*, I Escola Brasileiro-Argentina de Informática, editorial Kapelusz S.A., Buenos Aires, ed. preliminar, 1987, cap.2

capítulo 3

Concorrência nos Multiprocessadores

Concorrência nos Multiprocessadores

Introdução

Este capítulo mostra como se desenvolver programas paralelos para sistemas de multiprocessadores. De um modo geral existem dois modos pelos quais programas paralelos podem ser desenvolvidos:

- manual: o paralelismo é explicitamente indicado pelo programador [1],com uso de construções adequadas. Implica a recodificação do algoritmo

- automático: fica a cargo do compilador [2] descobrir o paralelismo do programa. Aproveita codificações já existentes, como as bibliotecas

Vamos abordar este dois tópicos e mostrar como isto pode ser feito. Vale a pena notar que, por enquanto, a detecção automática de paralelismo não está disponível para o sistema ACP, com o qual trabalhamos (veja o capítulo "O Sistema ACP" para uma descrição).

Elementos de linguagem paralela

Para podermos utilizar adequadamente os sistemas de multiprocessadores precisamos de uma linguagem eficiente com a qual se possa escrever os programas que conterão as operações que desejamos sejam executadas concorrentemente, sendo que por **concorrentes** nós nos referimos àqueles dois processos nos quais a primeira operação do segundo processo começa antes do término da última operação do primeiro processo.

Uma maneira de se detectar concorrência entre dois processos é usar um conceito de estrutura em blocos. Neste caso, cada processo P_i ($i=1,n$) de uma seqüência de n processos P_1, P_2 ... P_n pode ser executado concorrentemente através do uso de construções `cobegin` e `coend`, como mostrado abaixo:

```
begin
        P0;
        cobegin P1; P2; ... ; Pn; coend
        Pn+1;
```

end

Note que cobegin e coend declaram explicitamente a parte do programa que deve ser executada concorrentemente. A figura 1 abaixo mostra o grafo de precedência do programa acima:

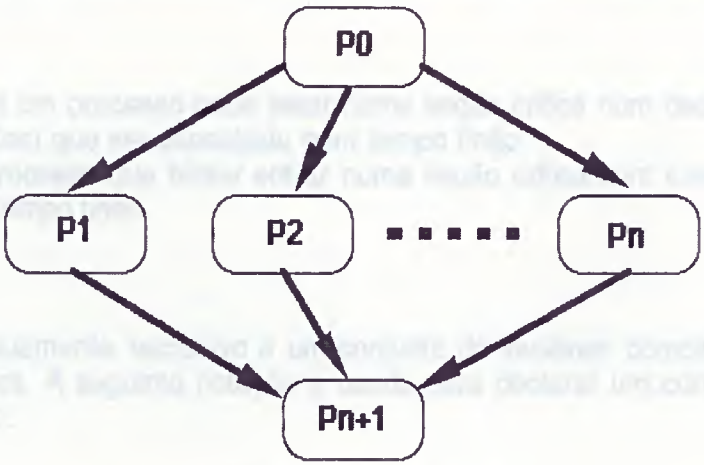


figura 1: grafo de precedência de um programa que usa as construções cobegin e coend

Neste caso o bloco delimitado por cobegin e coend somente começa a ser executado após o término do processo P_0, e o processo P_{n+1} somente tem início após o término de todos os processos P_i ($i=1,n$).

O paralelismo na execução de instruções pode ser, e com bastante freqüência é, encontrado nos laços. Por isto lançamos mão da construção parfor. Como exemplo consideremos o cálculo de C=A*B, por p processos onde A é uma matriz n x n e B e C são dois vetores n x 1:

```
s:= n/p;
parfor i:= 1 until p do begin
      for j:= (i-1)*s + 1 until s*i do begin
            C[j]:= 0;
            for k:= 1 until n do begin
                  C[j]:= C[j] + A[j,k]*B[k];
            end;
      end;
end;
```

É importante observar aqui que os processos executados concorrentemente são independentes uns dos outros ou seja, nenhuma variável alterável por um dos processos pode ser alterável por nenhum outro processo. Porém, estes processos podem acessar, mas sem alterar, variáveis comuns a todos eles.

Pode acontecer, entretanto, que processos concorrentes necessitem alterar dados comuns a todos eles e isto pode causar problemas pois se um processo começar a alterar algumas variáveis que já

estão sendo (concorrentemente) alteradas por outro processo, certamente teremos resultados com erros.

Assim, é importante manter um controle severo das **variáveis compartilhadas** (aquelas que podem ser modificadas por processos concorrentes) de maneira a permitir que um, e somente um, processo tenha acesso a elas em um dado momento da computação. Chamamos de **seções críticas** as partes dos programas que alteram variáveis compartilhadas. Existem algumas suposições que são feitas com relação às seções críticas:

- exclusão mútua: no máximo um processo pode estar numa seção crítica num dado momento
- término: uma seção crítica tem que ser executada num tempo finito
- escalonamento justo: um processo que tentar entrar numa seção crítica com certeza irá conseguir fazê-lo num tempo finito

O acesso mutuamente exclusivo a um conjunto de variáveis compartilhadas pode ser feito com o uso de algumas construções. A seguinte notação é usada para declarar um conjunto `V` de variáveis compartilhadas do tipo `TIPO`:

```
var V: shared TIPO;
```

e aí então uma seção crítica pode ser definida, usando-se o comando `csetc`, com o seguinte código:

```
csect V do P;
```

que associa um processo `P` à variável `V` e indica que o processo deve ter acesso exclusivo a ela. Fica claro que seções críticas que se refiram à mesma variável se excluem no tempo.

Seções críticas referentes a variáveis diferentes podem ser executadas em paralelo, como mostrado abaixo:

```
var V: shared TV;
var W: shared TW;
cobegin
      csect V do P1;
      csect W do P2;
coend
```

As seções críticas podem ser aninhadas, como:

```
csect V do begin
      :
      csect W do P;
end
```

Aqui, porém, temos o potencial para que apareça um **bloqueio**, no qual um ou mais processos ficam bloqueados por estarem esperando que os outros processos terminem. Por exemplo, os

processos P1 e P2 mostrados abaixo podem ficar bloqueados se P1 entrar na seção crítica Q ao mesmo tempo em que P2 entrar na R:

```
cobegin
     P1: csect Q do csect R do S;
     P2: csect R do csect Q do T;
coend
```

e a razão é que quando o processo P_1 tentar entrar na seção crítica R ele será bloqueado pois P_2 já estará lá e quando o processo P_2 tentar entrar na seção crítica Q ele será bloqueado pois P_1 já estará lá!

Como regra, um bloqueio sempre acontece quando 2 processos entram em suas seções críticas em ordem oposta um do outro e criam uma situação na qual cada processo fica esperando o outro terminar. Porém isto somente acontece se inferirmos que um recurso não pode ser liberado por um processo que estiver esperando pela alocação de outro recurso. Assim, uma maneira simples de se evitar os bloqueios é desenvolver um algoritmo que faça primeiro a liberação dos recursos de um processo e depois verifique a possibilidade de alocação de novos recursos.

Os métodos vistos aqui de se obter paralelismo nos programas envolvem a codificação do algoritmo usando estruturas próprias para o paralelismo. Isto pode ser muito bom para algoritmos que estejam sendo criados pela primeira vez, mas certamente não é muito interessante quando já se dispõe de bibliotecas prontas (como a CERNLIB ou IMSL, por exemplo) que já estão codificadas de forma seqüencial e necessitariam ser recodificadas.

Neste caso, seria mais aconselhável deixar a cargo do compilador a tarefa de encontrar as estruturas passíveis de paralelização, aliviando assim o trabalho do programador. É exatamente desta detecção automática de paralelismo que trataremos no item seguinte.

Detecção automática de paralelismo

Podemos usar o termo "paralelismo" de várias maneiras, ou em vários níveis. O paralelismo dentro de um programa pode existir desde a nível de instruções das linguagens (como visto anteriormente) até a nível de micro-operações.

Num ambiente de multiprocessadores MIMD o interesse geral está centralizado no paralelismo entre os processos, onde por processo se entende uma linha de programa ou um grupo de instruções de um programa.

O paralelismo nos processos pode existir numa hierarquia de níveis. Por exemplo, um grupo de instruções é considerado como sendo um processo de primeiro nível. Uma sub-rotina que é chamada pelo programa principal é um processo de segundo nível. Se esta rotina chamar outra sub-rotina esta passa a ser um processo de terceiro nível e assim por diante. Assim, podemos representar um programa por uma hierarquia de níveis, como mostrado na figura 2 a seguir, onde cada bloco representa um processo:

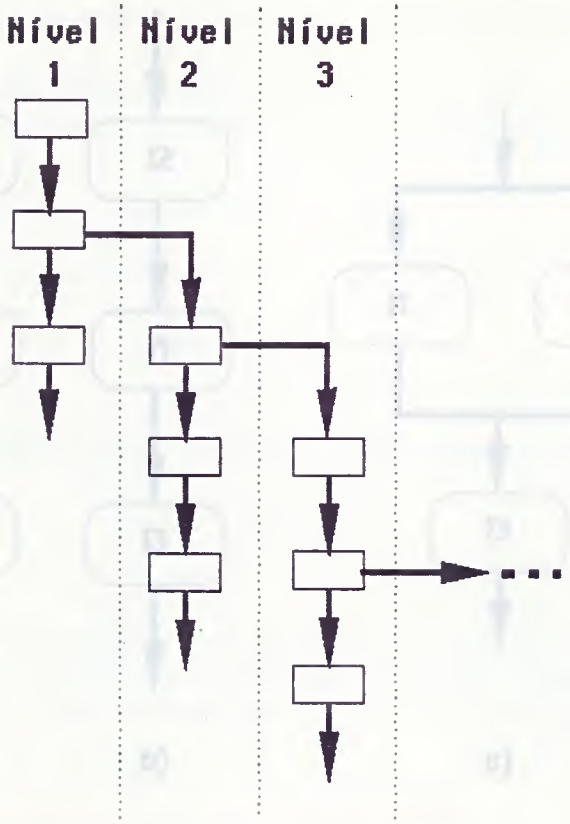


figura 2: representação hierárquica de um programa

Uma vez definidos os vários níveis de um programa, resta-nos reconhecer os processos que possam ser executados em paralelo dentro de um mesmo nível.

Já vimos, no item anterior, como podem ser explicitamente declaradas, pelo programador, partes de um programa a serem executadas em paralelo. Mas se desejarmos que isto seja feito independentemente do programador isto é, de forma automática, então precisaremos analisar o programa-fonte para então reconhecer suas partes paralelizáveis.

O principal fator para a detecção automática de paralelismo entre os processos de um mesmo nível é a dependência dos dados.

Considere as instruções `Ii` do programa seqüencial mostrado na figura 3-a a seguir: se a execução da instrução `I3` for independente da ordem em que `I1` e `I2` forem executadas (figuras 3-a e 3-b) então dizemos haver paralelismo entre as instruções `I1` e `I2`. Portanto elas podem ser executadas em paralelo, como mostrado na figura 3-c.

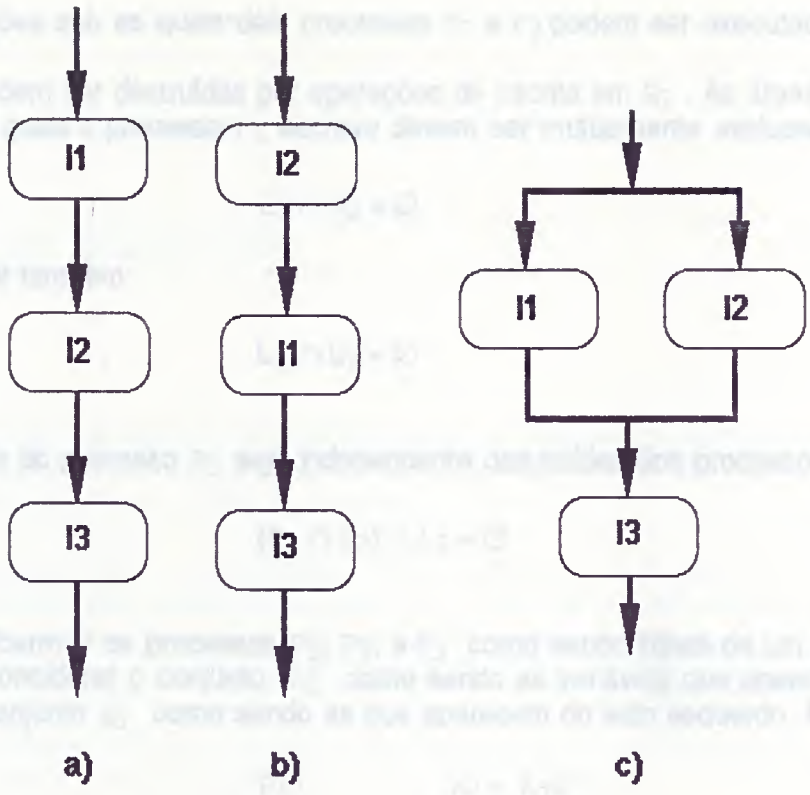


figura 3: execuções (a, b) seqüenciais e (c) paralela de um programa

Esta comutatividade é uma condição necessária, mas certamente não é suficiente, para que duas instruções sejam executadas em paralelo: pode acontecer que existam duas instruções que possam ser executadas em qualquer ordem, porém não paralelamente. Pode-se perguntar, então, quando dois processos seqüenciais podem ser executados em paralelo. O próximo item responde a esta pergunta.

As condições de Bernstein

As **condições de Bernstein**, dadas a seguir, devem ser satisfeitas para que dois processos seqüenciais possam ser executados em paralelo. Elas estão baseadas em dois conjuntos diferentes de variáveis de cada um dos processos P_i:

- o conjunto (L_i) de leitura do processo P_i, que representa o conjunto de todas as posições de memória para as quais a primeira operação em P_i que as envolve é uma leitura;
- o conjunto (E_i) de escrita do processo P_i, que representa o conjunto de todas as posições de memória alteradas por P_i.

As condições sob as quais dois processos P_1 e P_2 podem ser executados em paralelo são:

- posições de L_1 não podem ser destruídas por operações de escrita em E_2 . As áreas de memória das quais o processo P_1 lê e nas quais o processo P_2 escreve devem ser mutuamente exclusivas, ou seja:

$$L_1 \cap E_2 = \varnothing$$

- por simetria devemos ter também:

$$L_2 \cap E_1 = \varnothing$$

- é preciso que a entrada do processo P_3 seja independente das saídas dos processos P_1 e P_2, ou seja:

$$(E_1 \cap E_2) \cap L_3 = \varnothing$$

Se identificarmos os processos P_1, P_2, e P_3 como sendo linhas de um programa em linguagem de alto nível, podemos considerar o conjunto L_i como sendo as variáveis que aparecem do lado direito do sinal de atribuição e o conjunto E_i como sendo as que aparecem do lado esquerdo. Por exemplo, se tivermos:

```
P1:        X:= A+B
P2:        Y:= C+D
P3:        Z:= X+Y
```

é fácil notar que P_1 e P_2 podem ser executados em paralelo, mas P_3 terá que esperar pelo término de P_1 e P_2. De fato, tendo em mente as condições de Bernstein, podemos ver que:

$$L_1 = \{A,B\}; \qquad E_1 = \{X\}$$
$$L_2 = \{C,D\}; \qquad E_2 = \{Y\}$$
$$L_3 = \{X,Y\}; \qquad E_3 = \{Z\}$$

e portanto:

$$L_1 \cap E_2 = \varnothing$$
$$L_2 \cap E_1 = \varnothing$$
$$(E_1 \cap E_2) \cap L_3 = \varnothing$$

e então concluímos que P_1 e P_2 podem (realmente) ser executados em paralelo e podemos reescrever o programa da seguinte forma:

```
begin
    cobegin
        X:= A+B;
        Y:= C+D;
    coend
    Z:= X+Y;
end
```

É baseado nas condições de Bernstein que funciona a maioria dos programas detetores automáticos de paralelismo em programas-fonte escritos em linguagem de alto nível.

Reestruturação de programas

O problema de se decompor um programa em várias partes paralelas foi abordado anteriormente. O processamento paralelo permite a execução mais rápida, em geral, dos programas e a melhor utilização dos recursos de um sistema de multiprocessadores.

Existem vários fatores da arquitetura dos computadores paralelos que afetam a decomposição de um programa em forma paralela. Alguns deles são: velocidade de processamento, tempo de acesso à memória, largura de banda do barramento etc. Dependendo da arquitetura usada, diferentes aspectos do sistema podem se transformar no maior empecilho para a obtenção de um alto grau de concorrência.

Hoje, a utilização eficaz de muitos sistemas multiprocessadores está limitada pela falta de uma metodologia para ser usada no desenvolvimento dos programas paralelos.

Vejamos aqui uma abordagem metodológica no desenvolvimento de programas paralelos. No capítulo "O Programa CALOR (Hospedeiro-Nós)" e no capítulo "O Programa CALOR (Nós-Nós)" podem ser vistos dois exemplos que usam a metodologia apresentada no item a seguir.

Paralelização do algoritmo

A principal técnica para se paralelizar um programa é, como esperado, decompor os algoritmos em partes paralelas.

Duas características são importantes nesta decomposição : **partição** e **atribuição** . A partição é a divisão, propriamente dita, do algoritmo em partes distintas. A atribuição se refere à distribuição destas partes para os diferentes processadores.

Consideremos o exemplo de um programa para um sistema com apenas um processador e vejamos como ele pode ser decomposto e reestruturado para funcionar num sistema multiprocessador. O exemplo que vamos usar é um programa para fazer o histograma de uma imagem digital isto é, dada uma imagem digital onde cada ponto pode assumir um determinado valor de nível de cinza, queremos determinar quantos pontos têm cada um dos níveis de cinza. Supomos que cada ponto da imagem possa ter seu nível de cinza representado por um número entre 0 e 255. Seja `HISTOG[0..255]` um vetor que conterá a contagem de cada um dos 256 níveis de cinza da imagem. A imagem será representada pela matriz retangular `IMAGEM[0..m-1,0..n-1]` de `m` linhas e `n` colunas.

O programa serial poderia ser:

```
var IMAGEM[0..m-1,0..n-1]: integer;
var HISTOG[0..255]: integer;
initial HISTOG[0..255]:= 0
for i:= 0 until m-1 do begin
     for j:= 0 until n-1 do
          HISTOG[IMAGEM[i,j]]:= HISTOG[IMAGEM[i,j]] + 1;
```

A complexidade temporal deste algoritmo é O(m.n), que pode ser reestruturado se notarmos que a operação é iterativa e que uma linha da imagem é varrida no laço interno.

Uma vez que temos m linhas na imagem podemos dividir o programa em s segmentos iguais e não superpostos (supondo m divisível por p), sendo que cada um destes segmentos contém s = m/p linhas da imagem. Assim, cada um dos p processadores do multiprocessador faz o histograma de s linhas da imagem.

Usando a construção parfor vista anteriormente temos uma possível paralelização do algoritmo como:

```
var HISTOG[0..255]: integer;
initial HISTOG[0..255]:= 0

var LHISTOG[1..p, 0..255]: integer;
initial LHISTOG[1..p, 0..255]:= 0

parfor i:= 0 until p do begin
  var IMAGEM[(i-1)*s..s*i-1, 0..n-1]:integer;
  var LHISTOG[i, 0..255]: integer;
  for k:= (i-1)*s until s*i-1 do
    for j:= 0 until n-1 do
      LHISTOG[i,IMAGEM[k,j]]:= LHISTOG[i,IMAGEM[k,j]]+1;
end;

for j:= 0 until 255 do
  for i:= 1 until p do
    HISTOG[j]:= HISTOG[j] + LHISTOg[i,j;
```

Agora a complexidade temporal do algoritmo é O(s.n) isto é, um ganho teórico na velocidade de processamento da ordem de p, ou seja, do número de processadores, como esperado. Porém este valor nunca é alcançado na prática devido a vários fatores relacionados com as sincronizações que devem ocorrer e que não se percebe no algoritmo, pois dependem da parte física do computador.

Em geral uma implementação eficiente de um algoritmo depende muito da arquitetura da máquina na qual ele irá rodar. Mudanças sutis na arquitetura podem causar grandes mudanças no desempenho de um dado algoritmo.

O melhor desempenho de um algoritmo somente pode ser alcançado quando ele for codificado tendo-se em mente a arquitetura da máquina na qual se deseja rodá-lo.

A decomposição paralela eficiente do algoritmo depende do conhecimento da arquitetura do computador por parte do programador e da codificação do algoritmo de modo a se usar todo o potencial da máquina.

Comunicação

A relação entre a computação e a comunicação entre os processadores é um fator muito importante no desenvolvimento dos programas paralelos. O modo pelo qual os dados são distribuídos entre os processadores pode ter um efeito significativo na quantidade de comunicação requerida por um programa e, portanto, no seu tempo de execução.

Com o aumento da velocidade do processador em relação à velocidade de acesso à memória, torna-se cada vez mais crítica a reorganização dos algoritmos de forma a minimizar o número de referências aos dados.

Com a diminuição da capacidade da memória compartilhada (veja o capítulo "Arquitetura dos Multiprocessadores") o algoritmo deve ser decomposto de forma a minimizar o número de acessos à memória de massa (discos, principalmente). Assim é desejável se fazer o maior número de operações possível com os dados antes de se buscar novos dados dos discos.

Os capítulos "O Programa CALOR (Hospedeiro-Nós)" e "O Programa CALOR (Nós-Nós)" apresentam duas abordagens diferentes para a resolução do mesmo problema, sendo que este tem a comunicação entre o computador hospedeiro e os nós diminuída em relação àquele.

Referências

[1] K. Hwang e F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1986

[2] F.C. Mokarzel, Reestruturação Automática de Programas Seqüenciais para Processamento Paralelo, II Simpósio Brasileiro de Arquitetura de Computadores- Processamento Paralelo, 1988

capítulo 4

Desenvolvendo Algoritmos Paralelos

Desenvolvendo Algoritmos Paralelos

Introdução

Há pelo menos três maneiras de se desenvolver um algoritmo paralelo para se resolver um determinado problema:

- detectar o paralelismo inerente a um determinado algoritmo seqüencial já existente
- inventar um novo algoritmo (paralelo)
- adaptar um algoritmo paralelo que já resolve um problema similar

Cada uma destas maneiras tem seu momento certo para ser usada. A menos que sejamos a primeira pessoa no mundo a resolver um determinado problema, um algoritmo seqüencial já existirá: ao invés de reinventarmos a roda, poderemos aproveitar o trabalho já feito por outras pessoas. Talvez este algoritmo seqüencial possa ser paralelizado de uma maneira simples! Talvez não.

Porém, paralelizar algoritmos seqüenciais às cegas pode ser um grande erro: alguns algoritmos seqüenciais simplesmente não têm nenhuma paralelização possível. Assim, algoritmos paralelos feitos a partir destes algoritmos seqüenciais exibirão, necessariamente, um desempenho baixo.

Também a arquitetura do computador em uso irá geralmente requerer que uma abordagem particular seja feita. Portanto, algumas vezes é preciso desenvolver os algoritmos paralelos desde o início.

Conhecimento extra é importante

Suponhamos que estejamos tentando encontrar um algoritmo paralelo para resolver um determinado problema. Se existir um algoritmo seqüencial que resolve este problema podemos tentar usá-lo como ponto de partida (o que certamente soa bastante natural!). Se, todavia, o algoritmo seqüencial não for particularmente paralelizável, talvez então precisemos usar algum conhecimento extra sobre o problema a fim de poder usá-lo. Por exemplo, considere o problema de se encontrar a soma de n números inteiros. Sem dúvida nenhuma salta à mente um algoritmo seqüencial para resolver este problema:

```
begin
      SOMA:= 0;
      for i:= 1 to N do
            SOMA:= SOMA + Ai;
      endfor
end
```

Suponha que n é igual a 4. Assim, as adições acima serão feitas na seguinte ordem:

$$[\ (A_0 + A_1) + A_2\ ] + A_3$$

Este é um processo intrinsecamente seqüencial?

Dado o modo como os parênteses e colchetes estão agrupados, a soma de A_0 e A_1 deve ser encontrada antes que A_2 possa ser adicionada ao subtotal. Da mesma maneira, A_3 não pode ser adicionada à soma até que A_2 seja adicionada a A_0+A_1.

Entretanto nós sabemos (eis o nosso conhecimento extra sobre o problema!) que adições podem ser efetuadas em paralelo. Sabemos também, que a adição inteira em computadores digitais é associativa (ou quase, pelo menos). Por causa desta associatividade, podemos reescrever esta expressão:

$$(A_0 + A_1) + (A_2 + A_3)$$

Este agrupamento mostra que enquanto os dois primeiros elementos estão sendo somados, o segundo par pode estar sendo somado também.

Custos de comunicação devem ser considerados

É um erro ignorar os custos de comunicação ao se determinar a complexidade de um algoritmo paralelo pois pode ser que os custos de comunicação sejam maiores do que a complexidade computacional isto é, que mais tempo seja gasto enviando-se os dados aos processadores do que realmente manipulando-se estes dados.

Por exemplo, suponha que leve u unidades de tempo para se fazer uma adição em ponto flutuante num multiprocessador particular e que leve 100.u unidades de tempo para passar um número em ponto flutuante de um processador para outro. Suponha agora que temos que somar n números em ponto flutuante num sistema que contenha p processadores e suponha que estes números estejam, inicialmente, na memória local de qualquer processador que desejarmos.

Se $n < 101$, então não importa o número p de processadores, será sempre mais rápido somar estes n números num único processador do que fazer as adições em paralelo, uma vez que 100 adições podem ser feitas no mesmo tempo que 1 comunicação.

O algoritmo deve estar adaptado à arquitetura

O desempenho de um mesmo algoritmo pode ser radicalmente diferente em diferentes arquiteturas, algumas vezes devido a atrasos por causa da comunicação, como mencionado acima.

Outras vezes alguns outros fatores podem estar envolvidos, como por exemplo a sincronização nos processadores matriciais [1],[2] (veja também o capítulo "Arquitetura dos Multiprocessadores"): sabemos que a sincronização é automática nos processadores matriciais (pois eles trabalham em sincronismo).

Já nos multiprocessadores (uma outra arquitetura!) a sincronização deve ser feita através de programa e portanto é menos eficiente.

Assim, devemos ter em mente para qual tipo de arquitetura estamos fazendo um determinado programa paralelo sob pena de, não agindo assim, prejudicarmos a eficiência do programa.

Algoritmos para computadores MIMD

Quando se fala em algoritmos para computadores paralelos tipo MIMD surgem, imediatamente, três dúvidas:

- quais tipos de algoritmos existem
- como se pode estimar o desempenho deles
- como é afetado este desempenho

Neste item trataremos destas e de outras questões.

Classificação dos algoritmos MIMD

A paralelização de algoritmos em modelos com arquitetura tipo MIMD (*Multiple Instruction stream, Multiple Data stream*) pode ser dividida em três categorias:

- algoritmos *pipelined*
- algoritmos particionados (ou sincronizados)
- algoritmos assíncronos.

Algoritmo *pipelined* é um conjunto ordenado de segmentos no qual a saída de cada segmento passa a ser a entrada do seu sucessor. A entrada do algoritmo serve de entrada ao primeiro segmento e a saída do último segmento é a saída do algoritmo. Como em todos os sistemas do tipo *pipeline* , todos os segmentos devem produzir seus resultados à mesma taxa, sob pena de o segmento mais lento se tornar o gargalo do sistema. Um compilador paralelo cujas fases individuais fossem atribuídas a diferentes processadores seria um exemplo de um algoritmo deste tipo [3].

Um **algoritmo sistólico** é um tipo especial de algoritmo *pipelined* que tem três atributos que o distingüem daquele: (i) o fluxo de dados é regular; (ii) os dados podem fluir em mais de uma direção e (iii) os cálculos feitos em cada segmento são essencialmente idênticos. Da mesma forma que nos algoritmos *pipelined*, os algoritmos sistólicos requerem sincronização implícita entre os processos que produzem os dados e os processos que os usam.

Ao contrário dos algoritmos *pipelined*, onde os vários processadores têm tarefas diferentes, nos **algoritmos particionados** há o compartilhamento das tarefas: um problema é subdividido em vários sub-problemas, que são resolvidos pelos vários processadores. As soluções dos sub-problemas são então agrupadas para gerar a solução do problema original. Isto irá requerer que haja uma sincronização entre os processadores; por isto estes algoritmos são chamados também de algoritmos sincronizados. Os algoritmos referenciados nos capítulos "O Programa CALOR (Hospedeiro-Nós)" e "O Programa CALOR (Nós-Nós)" são particionados.

Como exemplo de algoritmo particionado considere o problema de se somar `k.p` valores num sistema de `p` processadores: uma maneira natural de se calcular esta soma é igualar a zero o valor de uma variável global (que conterá o valor final da soma) e criar `p` processos, um para cada processador, sendo que cada processo soma um conjunto distinto de `k` números (lembre-se de que mostramos acima que somas podem ser paralelizadas de uma maneira natural). Quando um processo termina de calcular a sua sub-soma, ele adiciona este número à variável global. Os processos, é claro, devem ter acesso exclusivo à variável global enquanto estiverem atualizando seu valor. Após somar seus sub-totais à variável global, os processos terminam. Depois que todos os processos tiverem terminado, a variável global conterá a resposta do problema.

Os **algoritmos assíncronos** são aqueles que trabalham sem sincronização entre os processos. Todos os processos podem estar trabalhando para atingir o mesmo objetivo (como no caso do particionamento), ou podem ter tarefas diferentes (como no caso do *pipeline*), mas o elemento essencial é que nenhum processo nunca tem que esperar que outro processador lhe forneça seus dados de entrada. Pelo contrário, eles são caracterizados pela habilidade dos processadores em trabalharem com os dados mais recentemente disponíveis. Esta característica não-determinista faz com que a predição de seus desempenhos seja difícil.

Fatores que limitam o speedup

Speedup de um algoritmo paralelo é a razão entre os tempos de execução do melhor algoritmo seqüencial e o do algoritmo paralelo.

Vários fatores podem contribuir para limitar o *speedup* alcançável por um determinado algoritmo paralelo num modelo MIMD.

Um deles, bastante evidente, é o tamanho do problema: se não houver trabalho suficiente para ser feito pelos processdores disponíveis, então qualquer algoritmo paralelo exibirá um *speedup* prejudicado. Este fenômeno, chamado de **Efeito Amdahl**, explica por que o *speedup* é uma função crescente do tamanho do problema dado. A figura 1 a seguir ilustra o Efeito Amdahl:

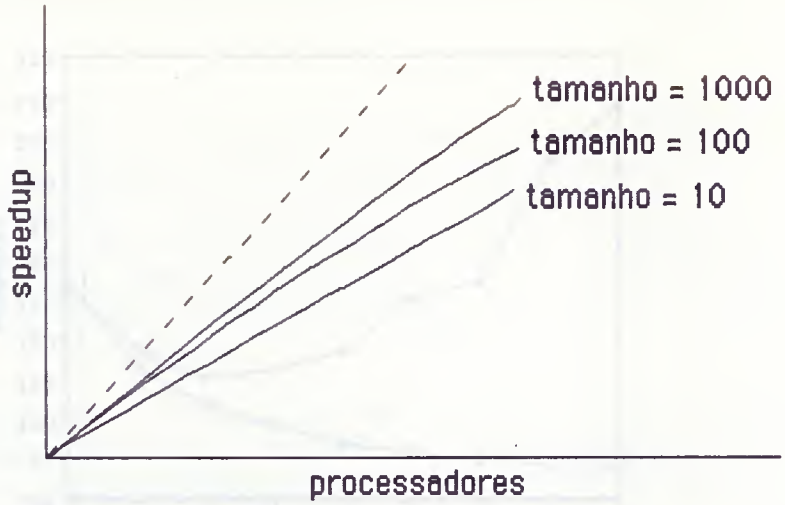


figura 1: Efeito Amdahl: como regra geral o speedup é uma função crescente do tamanho do problema

Os códigos seqüenciais limitam o *speedup* dos três tipos de algoritmos (**Lei de Amdahl** [3]).

Há ainda um tipo mais sutil de limitação causada pelo uso de códigos seqüenciais em algoritmos particionados, como é o caso do algoritmo usado neste trabalho: se uma parte do algoritmo precisar ser executada seqüencialmente por um dos `p` processadores, então os restantes `p-1` processadores deverão esperar pela conclusão da parte seqüencial.

Os processadores que trabalham num algoritmo não devem estar muito vinculados, particularmente quanto à contenção dos processadores por recursos. Muita contenção por um único recurso limita o *speedup* dos algoritmos *pipelined*, particionados e assíncronos tanto quanto a presença de códigos seqüenciais. A contenção por recursos de dados é chamada de bloqueamento do programa.

Um problema similar pode ocorrer num nível mais fundamental: se as instruções usadas por todos os processadores estiverem numa única memória compartilhada, então a contenção dos processadores por esta memória implicará um limite máximo do *speedup* alcançável por qualquer algoritmo paralelo.

Uma experiência [3] feita na Universidade de Carnegie-Mellon demonstra o efeito de se manter todo o código numa única memória (veja a figura 2 a seguir). Na figura 2 os pontos representam os tempos necessários para que os algoritmos paralelos encontrem as raízes de uma função, em função do número de processadores e:

• a linha com pontos em forma de losangos cheios representa uma situação na qual o código para cada processador estava armazenado na mesma página de memória (compartilhada). Observe que a contenção desta página de memória única foi tão alta que 7 processadores demoraram mais tempo para resolver o problema do que somente 1 processador.

• a linha com pontos em forma de quadrados vazios representa uma situação na qual o código para cada processador foi colocado em páginas de memória exclusivas a cada processador.

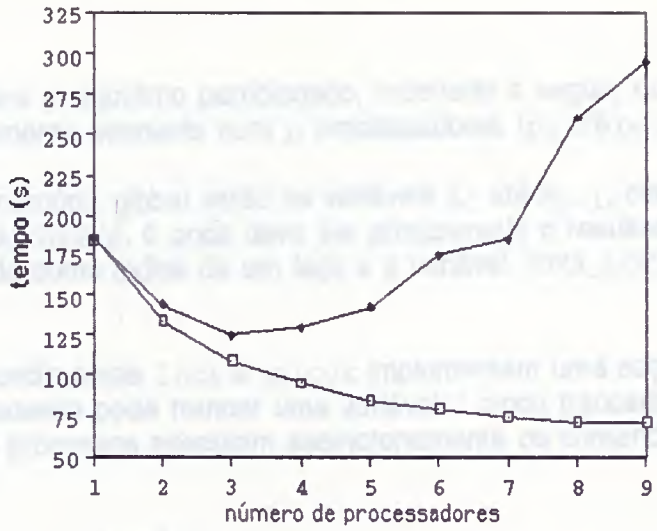


figura 2: a contenção dos processadores por uma única memória, a qual contém todo o código a ser utilizado pelos processadores, limita o speedup

A situação ideal é aquela na qual cada processador tem a sua memória local (exclusiva), onde são armazenados o código, as variáveis locais e as constantes. Desta maneira a memória compartilhada somente é acessada para se descobrir o valor de uma variável que seja compartilhada por todos os processos e esteja sujeita a modificações. Em outras palavras, a memória compartilhada somente é usada para comunicação e sincronização de processos.

A **carga de trabalho** deve ser bem distribuída entre todos os processadores, sob pena de sobrecarregar muito alguns e deixar ociosos outros. Tipicamente, a carga de trabalho não é problema para os algoritmos particionados que estejam sendo executados em computadores fortemente acoplados.

Em geral, contudo, a distribuição do trabalho (ou balanceamento de carga) é uma tarefa crítica. Existem duas políticas gerais de alocação, ou decomposição:

- estática, na qual as tarefas e suas relações de precedência são conhecidas antes da execução
- dinâmica, onde as tarefas são geradas durante a execução do programa

A vantagem da decomposição estática é que ela permite a pré-alocação das tarefas para processadores e isto pode reduzir a quantidade de comunicação entre eles. A vantagem da decomposição dinâmica é que ela facilita manter todos os processadores ocupados.

Os algoritmos que fazem relativamente poucas operações entre as comunicações exibem baixo desempenho em computadores MIMD.

Finalmente, deve ser usado o algoritmo mais eficiente possível. É fato já conhecido que o algoritmo seqüencial mais rápido não gera, necessariamente, o mais eficiente algoritmo paralelo.

Analisando a complexidade dos algoritmos MIMD

Considere o algoritmo particionado, mostrado a seguir, que calcula a soma de n valores num multiprocessador fortemente acoplado com p processadores (p_0 até p_{n-1}).

Numa memória global estão as variáveis A_1 até A_{p-1}, contendo os valores a serem somados. A variável global SOMA_GLOBAL é onde deve ser armazenado o resultado final. Cada processador P_i tem uma variável local J_i usada como índice de um laço e a variável $SOMA\_LOCAL_i$, que contém o sub-total do processador.

Os procedimentos lock e unlock implementam uma seção crítica; o trancamento é uma ação atômica e nenhum processo pode trancar uma variável J ainda trancada. O comando for all é executado para p processos. Os processos executam assincronamente os comandos dentro da estrutura for all e terminam no endfor:

```
begin
  SOMA_GLOBAL:= 0;
  for all Pi, onde 0 ≤ i ≤ p-1 do
    SOMA_LOCAL:= 0
    for J:= 1 to n step p do
      SOMA_LOCAL:= SOMA_LOCAL + Aj
    endfor
    lock(SOMA_GLOBAL)
    SOMA_GLOBAL:= SOMA_GLOBAL + SOMA_LOCAL
    unlock(SOMA_GLOBAL)
  endfor
end
```

Vamos determinar a complexidade temporal deste algoritmo paralelo e tentar determinar quantos processadores que devem ser usados para resolvê-lo. Primeiro, quanto tempo leva para se criar os processos? Se o primeiro processo cria os restantes p-1 processos sozinho, a complexidade temporal da criação do processo é O(p).

Supondo que haja p processadores e p memórias, e que cada processo rode no seu próprio processador, então não há conflito de memória e o tempo gasto para se ter as somas locais é O(n/p).

Uma vez que SOMA_GLOBAL é atualizada dentro de uma seção crítica, somente 1 processo pode acessá-la num determinado instante. Assim os processos têm que esperar para usar este recurso e a complexidade temporal desta atualização é O(p).

No endfor final é onde a sincronização ocorre. Se a sincronização for feita por intermédio de um semáforo global [1], então novamente cada processo tem que ter acesso exclusivo a ele e a complexidade temporal da sincronização se torna O(p).

Assim, a complexidade deste algoritmo é $O(p + n/p)$. O mínimo da função $p + n/p$ ocorre quando $p=\sqrt{n}$.

Desta forma, o tempo de execução do algoritmo paralelo é minimizado quando $\sqrt{n}$ processadores são utilizados.

Referências

[1] K.Hwang e F.A.Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1986

[2] R.W.Hockney e C.R.Jesshope, *Parallel Computers*, Adam Hilger Ltd., Bristol, 1986, cap.3

[3] M.J.Quinn, *Designing Efficient Algorithms For Parallel Computers*, McGraw-Hill, 1987

capítulo 5

O Sistema ACP

O Sistema ACP

Introdução

Este capítulo descreve o sistema de multiprocessadores desenvolvido no *Fermi National Accelerator Laboratory* (FERMILAB) pelo programa *Advanced Computer Program* (ACP).

O sistema é modular e está baseado em placas processadoras de baixo custo que incluem 2 Mbytes de memória RAM. Estes módulos seguem o padrão VME e rodam programas de reconstrução de experiências (em FORTRAN) à velocidade perto da de um VAX 11/780.

Neste capítulo descrevemos o multiprocessador ACP do Laboratório de Física Experimental de Altas Energias (LAFEX), sob coordenação do Professor Alberto Santoro, no Centro Brasileiro de Pesquisas Físicas (CBPF-RJ), no qual desenvolvemos a parte prática do nosso trabalho.

Por fim apresentamos, de uma maneira bastante resumida, porém suficiente para dar uma idéia geral, a maneira de se programar o ACP, bem como algumas de suas rotinas mais importantes.

O Conceito do multiprocessador ACP

Em contraste com as necessidades sentidas pela Ciência da Computação e pelos projetos de processadores paralelos comerciais, o ACP foi dirigido a problemas com estruturas paralelas claramente identificadas. Isto permitiu que se adotasse a estratégia de se começar o projeto com o paralelismo mais simples possível, onde cada processador trabalha de uma maneira completamente independente dos outros, sem nenhuma comunicação entre eles. Isto porque o problema de reconstrução de eventos (para o qual foi, inicialmente, desenvolvido o ACP) requer somente que os dados de cada evento da experiência sejam processados separadamente uns dos outros, mas da forma mais rápida possível.

Com o passar do tempo mais generalidade está sendo acrescentada [1] ao sistema para permitir que problemas com graus de paralelismo mais acentuados sejam resolvidos, através da comunicação entre os nós.

Não é muito difícil de se imaginar que a comunicação entre os processadores é um fator importante na resolução de problemas tais como teorias na rede [2], equações diferenciais, simulação de fluidos etc. Um estudo feito no *California Institute of Technology* [3] mostrou que configurações em forma de hipercubos (nas quais os processadores estão ligados aos seus vizinhos num espaço de n dimensões) são bastante eficientes na resolução de, basicamente, todos os problemas científicos.

Na figura 1 (na próxima página) temos um esquema geral do computador ACP, que é composto de:

- um computador-hospedeiro (neste caso um MicroVAX II)
- processadores (denotados por CPU) ligados entre si por um barramento VME e ao computador-hospedeiro por um barramento denotado por ***branch bus***
- outros módulos de controle, explicados em detalhe no item "A arquitetura do sistema usado".

O *hardware* é acompanhado de programas básicos que cuidam da simulação, compilação, descarregamento de rotinas nos nós, depuração dos programas dos nós, comunicação entre o computador-hospedeiro e os nós ACP e, com um certo cuidado por parte do programador, da comunicação entre os nós também. Vejamos a seguir como são compostos este nós.

Os processadores do ACP

Foram produzidos dois projetos de UCP: um é baseado no microprocessador 68020 da Motorola e o outro no 32100 da AT&T. Cada projeto inclui o respectivo co-processador de ponto flutuante e 2 Mbytes de memória RAM local (expansível até 8 Mbytes). A freqüência em ambos os projetos é de 16 MHz.

A UCP baseada no 68020 roda programas em FORTRAN-77 compilados com o compilador da Absoft e a UCP baseada no 32100 usa o compilador da Philon. A linguagem FORTRAN foi escolhida por permitir o aproveitamento das bibliotecas já existentes e é a única disponível atualmente.

Os desempenhos das duas UCP's dependem muito dos programas; para a reconstrução de eventos eles variam entre 60% e 80% de um VAX (aqui 1 VAX equivale ao modelo VAX 11/780 com acelerador de ponto flutuante e FORTRAN VMS 4.x). Algumas rotinas da CERNLIB, quando codificadas em linguagem de máquina do 68020 (ou 32100), fazem com que o desempenho alcance 97% de um VAX.

Outros microprocessadores e compiladores estão sendo estudados e, inclusive, sendo adotados nas gerações mais novas do ACP [1].

A arquitetura do sistema usado

Uma vez que o IFT ainda não tem o seu próprio ACP, foi-nos autorizado usar o do Laboratório de Física Experimental de Altas Exergias (LAFEX) no Centro Brasileiro de Pesquisas Físicas (CBPF-RJ). Um diagrama do multiprocessador ACP lá existente pode ser visto na figura 1, mostrada a seguir.

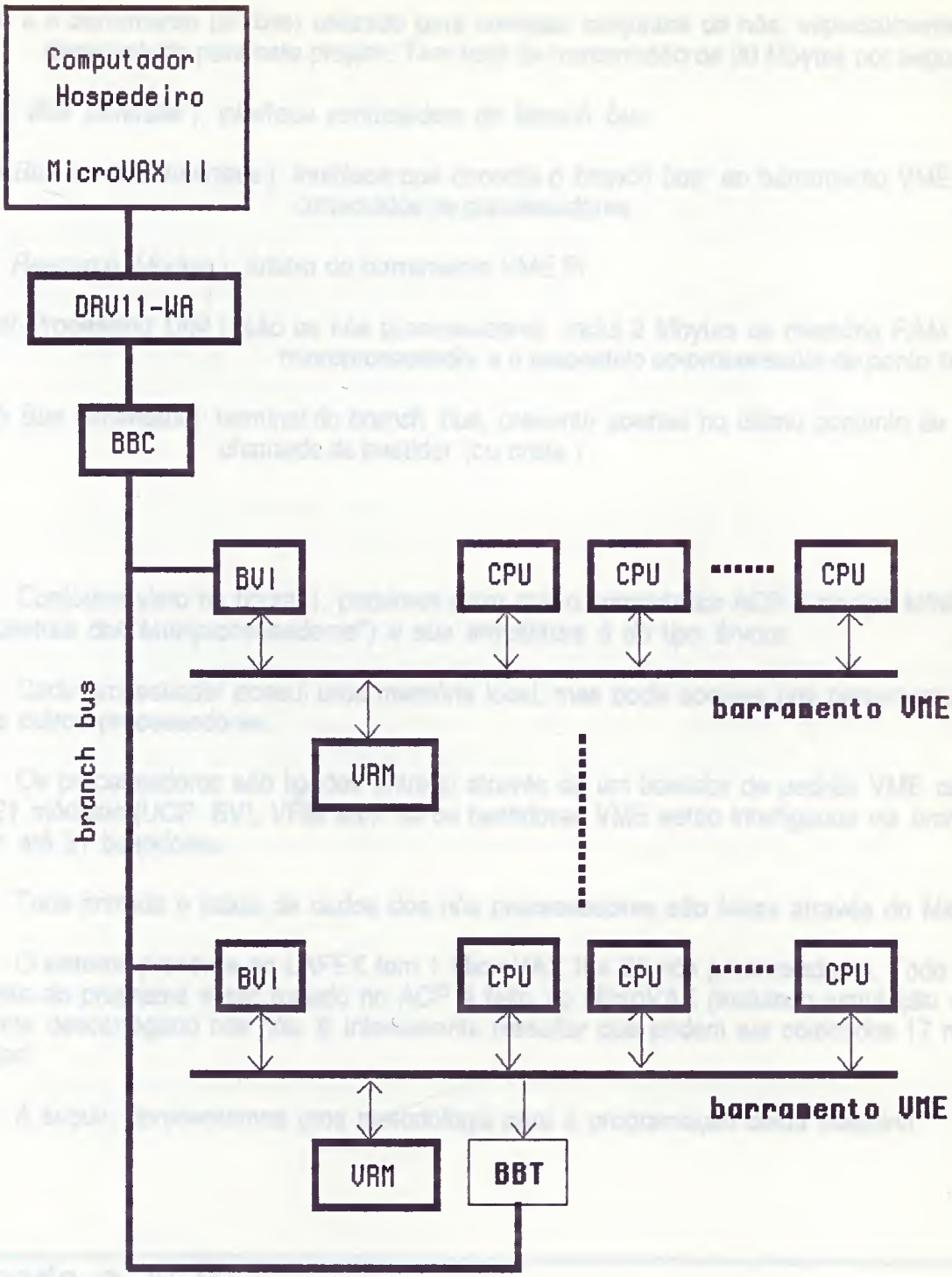


figura 1: esquema geral de um computador ACP

Na figura 1 acima temos diversas siglas e seus significados são:

- DRV11-WA [4] : interface paralela de 16 bits (DMA) da DEC para o sistema Q-Bus, com o qual se acessa o MicroVAX.

- *Branch Bus*: é o barramento (32 bits) utilizado para conectar conjuntos de nós, especialmente desenvolvido para este projeto. Tem taxa de transmissão de 20 Mbytes por segundo.
- BBC (*Branch Bus Controler*): interface controladora do *branch bus*.
- BVI (*Branch Bus to VME Interface*): interface que conecta o *branch bus* ao barramento VME, ao qual estão conectados os processadores.
- VRM (*VME Resource Module*): árbitro do barramento VME [5].
- CPU (*Central Processing Unit*): são os nós processadores. Inclui 2 Mbytes de memória RAM local, o microprocessador e o respectivo co-processador de ponto flutuante.
- BBT (*Branch Bus terminator*): terminal do *branch bus*, presente apenas no último conjunto de nós, chamado de bastidor (ou *crate*).

Conforme visto na figura 1, podemos dizer que o computador ACP é do tipo MIMD (veja o capítulo "Arquitetura dos Multiprocessadores") e sua arquitetura é do tipo árvore.

Cada processador possui uma memória local, mas pode acessar (via barramento VME) as memórias dos outros processadores.

Os processadores são ligados entre si através de um bastidor de padrão VME capaz de suportar até 21 módulos (UCP, BVI, VRM etc). Já os bastidores VME estão interligados via *branch bus*, que pode interligar até 31 bastidores.

Toda entrada e saída de dados dos nós processadores são feitas através do MicroVAX.

O sistema presente no LAFEX tem 1 MicroVAX II e 21 nós processadores. Todo o desenvolvimento do programa a ser rodado no ACP é feito no MicroVAX (incluindo simulação e depuração) e posteriormente descarregado nos nós. É interessante ressaltar que podem ser colocados 17 nós ACP num mesmo bastidor.

A seguir, apresentamos uma metodologia para a programação desta máquina.

Programando o ACP

Para podermos rodar um programa no ACP [6] é preciso, antes de mais nada, dividi-lo em duas partes: uma irá rodar no computador-hospedeiro e outra em cada um dos nós. Um programa paralelo típico, a ser rodado no ACP, conterá as 3 partes mostradas abaixo:

- iniciação geral de variáveis importantes
- um laço principal, que coordenará o programa que rodará nos nós
- recolhimento dos resultados e finalização

Evidentemente, a parte que mais consome tempo de UCP deve ser escolhida de tal forma a rodar nos nós, deixando a cargo do computador-hospedeiro a função de coordenação dos nós (se esta não for possível de ser executada pelos próprios nós e isto é possível, porém requer cuidados especiais) e funções de Entrada/Saída.

A passagem dos parâmetros para os nós deve ser feita através de blocos de dados (COMMON BLOCKS definidos na linguagem FORTRAN) previamente definidos e não através de argumentos de sub-rotinas (porém, dentro de um nó os parâmetros são passados normalmente como argumentos de sub-rotinas).

Dois modos de se passar, no ACP, dados para os processadores estão mostrados a seguir:

- transmissão de evento: na qual dados (os eventos, da Físical) são passados a nós individuais ;
- transmissão comum: na qual os mesmos dados são enviados a todos os nós.

Note que não se menciona aqui a comunicação direta entre nós, o que será tema do capítulo "O programa CALOR (Nós-Nós)", por se tratar de um uso especial do ACP.

O programa que roda nos nós deverá terminar com um comando RETURN e poderá escrever somente na unidade 9 usando, por exemplo, o comando WRITE(9,*) ou uma de suas variantes (veja o manual da linguagem FORTRAN para sintaxe completa do comando WRITE), se bem que isto não é desejável a não ser em fase de depuração e testes do programa. Todas as escritas na unidade 9 feitas pelos nós serão colocadas em um *buffer* e automaticamente lidas e impressas pelo hospedeiro quando um evento for lido do nó. Quaisquer outros tipos de Entrada/Saída feitos pelos nós são considerados ilegais .

Todos os programas que usarem o sistema ACP deverão incluir o comando:

```
include 'ACP$AREA:USER_COMMON.INC'
```

que irá cuidar das definições usadas pelo ACP.

Além disso é necessário chamar a rotina ACP_INIT antes que qualquer rotina que use o ACP seja chamada, bem como chamar a rotina de finalização ACP_EXIT no final do programa do hospedeiro.

Como rodar um programa no ACP

Como mencionado anteriormente, o programa que se quer rodar no ACP deve ser dividido em duas partes: uma que roda no computador-hospedeiro e outra que roda nos nós. A parte que roda nos nós deve ser escrita em FORTRAN-77 puro enquanto que a parte que roda no MicroVAX pode usar extensões do VAX para o FORTRAN.

Após ter feito a divisão mencionada acima deve-se criar um arquivo (***User's Parameter File***, ou UPF) que instruirá o sistema ACP sobre qual programa rodar (no hospedeiro e nos nós), quantos nós alocar e outras informações importantes.

A seguir resumimos alguns comando obrigatórios de se ter nos arquivos tipo UPF. Uma decrição detalhada e completa pode ser encontrada na referência *ACP Users's Guide for Utilities* [6].

- `SYSTEM = CBPF`
  deve ser a primeira linha de um arquivo UPF e especifica qual o sistema que vai ser usado. Por exemplo, se se quisesse usar o simulador em vez do ACP real do CBPF, trocaríamos a palavra `CBPF` por `SIMULATOR`.

- `HOST SOURCE = nome_do_arquivo`
  este comando indica o nome do arquivo fonte (em FORTRAN) que contém o programa a ser rodado no computador-hospedeiro (*host*).

- `NODE SOURCE = nome_do_arquivo`
  idem para o programa a ser rodado nos nós.

- `NODE MAIN SUBROUTINE = nome_da_sub-rotina`
  aqui especifica-se o nome da rotina que deve ser chamada (no nó) sempre que o computador-hospedeiro autorizar o nó a rodar seu programa.

- `NODE BLOCKS = 1:nome_do _bloco1, ... ,n:nome_do_blocon`
  com este comando são especificados o número e o nome de todos os `n` blocos dos nós que terão seus dados lidos (ou escritos) pelo hospedeiro. Estes são os `COMMON BLOCKS` da linguagem FORTRAN sobre os quais nos referimos anteriormente.

- `CLASS 1:NODE = n 68020`
  indica que `n` nós do tipo 68020 (em contraposição aos nós do tipo 32100) serão usados pelo programa. No caso de se querer usar o simulador deve-se substituir o número `68020` pela palavra `SIMULATOR`.

Após ter preparado os 3 arquivos necessários (no mínimo) para rodar seu programa (ou seja, o programa fonte em FORTRAN que roda no hospedeiro, o que roda nos nós e o arquivo UPF associado), o usuário emite o comando:

```
ACP$MULTICOMP nome_do_arquivo_tipo_UPF
```

que irá compilar os dois programas em FORTRAN, uni-los com as rotinas ACP necessárias e prepará-los para rodar (no sistema especificado pelo arquivo tipo UPF que lhe serviu como parâmetro).

Então, para rodar seu programa no ACP (ou no simulador, dependendo da primeira linha do arquivo tipo UPF usado no comando `ACP$MULTICOMP`) o usuário apenas precisa emitir o comando:

```
RUN nome_do_arquivo_tipo_UPF
```

que iniciará (`RUN`) os programs no computador-hospedeiro e nos nós processadores do ACP.

Rotinas importantes do ACP

O controle dos (e a comunicação com os) nós ACP é feito através do uso de uma série de rotinas e funções disponíveis para o usuário. Uma descrição detalhada delas pode ser encontrada na referência *ACP Software User's Guide for Event Oriented Processing* [7].

A seguir mostramos as mais importantes:

- ACP_INIT
  esta rotina carrega o programa a ser rodado nos nós, estabelece um canal de comunicação entre o computador-hospedeiro e os nós e inicia alguns parâmetros a serem usados por outras rotinas. Ela deve ser a primeira rotina a ser invocada antes de se chamar qualquer outra rotina do ACP.

- ACP_SENDEVENT(ARRAY,LENGTH,SEND_DONE)
  ela localiza um nó disponível, transmite os dados deste evento ao bloco número 1 (veja o comando NODE BLOCKS do arquivo UPF no item anterior) e autoriza o nó a rodar seu programa. Existe uma variação desta rotina na qual se pode especificar (i) para qual bloco se quer mandar os dados, (ii) para qual nó e (iii) se o nó deve rodar seu programa ou esperar até outra ordem: foi esta variação que foi usada no programa CALOR, explicado no capítulo "O Programa CALOR (Hospedeiro-Nós)". ARRAY é o nome da matriz (FORTRAN) que contém os dados a serem transmitidos, LENGTH é o tamanho do bloco a ser passado (em palavras, de 32 bits) e SEND_DONE é uma variável lógica que assume .TRUE. se um nó foi encontrado e .FALSE. caso contrário.

- ACP_GETEVENT(ARRAY,LENGTH,GET_DONE)
  ela localiza um nó que tenha terminado de rodar seu programa, recebe os dados deste evento no bloco número 2 (veja o comando NODE BLOCKS do arquivo UPF no item anterior) e informa ao sistema que este nó está disponível para receber mais eventos. Existe uma variação desta rotina na qual se pode especificar (i) de qual bloco se quer receber os dados, (ii) de qual nó e (iii) se o nó deve ser considerado disponível para receber outros eventos ou não: foi esta variação que foi usada no programa CALOR, explicado no capítulo "O Programa CALOR (Hospedeiro-Nós)". ARRAY é o nome da matriz (FORTRAN) que contém os dados a serem transmitidos, LENGTH é o tamanho do bloco a ser passado (em palavras, de 32 bits) e GET_DONE é uma variável lógica que assume .TRUE. se um nó foi encontrado e .FALSE. caso contrário.

- ACP_EXIT
  esta rotina executa todos os procedimentos de saída para o sistema ACP. Deve ser chamada no final do programa do usuário.

O conceito de classes de nós

Como já visto, existem vários tipos de nós processadores, como por exemplo aqueles que usam o microprocessador 68020 e aqueles que usam o 32100.

Outras diferenças podem existir entre os diversos nós processadores como, por exemplo, a quantidade de memória RAM local ou quanto ao programa que ele deve rodar (sim! É possível rodar um programa diferente em cada nó! Veja o capítulo "O Programa CALOR (Nós-Nós)").

Tendo isto em mente é possível tratar nós diferentes de maneiras diferentes. Para tal usa-se a noção de classe: dois nós são ditos pertencer à mesma classe se quando um nó for usado o outro puder ser usado em seu lugar. A especificação dos nós, no arquivo tipo UPF, pode também conter informações quanto às classes às quais eles pertencem.

No capítulo "O programa CALOR (Hospedeiro-Nós)" são feitas referências a rotinas que necessitam da informação da classe dos nós sobre os quais elas atuam. Mais detalhes podem ser encontrados na referência *ACP Software User's Guide for Event Oriented Processing* [7] listada a seguir.

Referências

[1] B. Schulze e R. Valois, Segunda Geração de Processadores Paralelos do "*Advanced Computer Program* " (ACP), II Simpósio Brasileiro de Arquiteturas de Computadores - Processamento Paralelo, 1988

[2] E. Eichten et al, *Lattice Gauge Theory and the ACP*, FERMILAB internal communication, May 6, 1987

[3] G.C. Fox e S. Otto, *Algorithms for concurrent processors*, Physics Today, pag.50, May/84

[4] *VAX System and Options Catalog*, 1987 January-March, cap.3

[5] D. Jones, *VMEbus Primer*, Electronics and Wireless World, pag.8, December/86

[6] *ACP Users's Guide for Utilities*, Advanced Computer Program, FERMILAB

[7] *ACP Software User's Guide for Event Oriented Processing*, Advanced Computer Program, FERMILAB, revision 3.0, December 4, 1987

capítulo 6

Equações Diferenciais Parciais

Equações Diferenciais Parciais

Introdução

Este capítulo descreve alguns métodos numéricos utilizados na solução de Equações Diferenciais Parciais (EDP) a valores de contorno usando diferenças finitas e aborda, em especial, o Método SOR (do inglês, *Simultaneous Over-Relaxation*) utilizado nos programas para computador desenvolvidos neste trabalho e mostrados nos capítulos "O Programa CALOR (Hospedeiro-Nós)" e "O Programa CALOR (Nós-Nós)".

Inicialmente introduzimos a equação usada neste trabalho, a equação de Laplace, desenvolvendo-a em termos de diferenças finitas.

A seguir apresentamos vários métodos numéricos para se resolvê-la, encerrando com a apresentação do método SOR e de dois métodos que aceleram sua convergência: ordenamento ímpar-par e aceleração de Chebyshev [1].

Os números (N) de iterações necessárias para se chegar a uma mesma solução são comparados para todos os métodos abordados, mostrando assim a relação entre seus desempenhos.

As diferentes equações diferenciais parciais

As equações diferenciais parciais são geralmente classificadas [2] em 3 grupos: elípticas , hiperbólicas e parabólicas .

Um exemplo de uma equação hiperbólica é a equação de propagação de uma onda unidimensional :

$$\frac{\partial^2 u}{\partial t^2} = v^2 \cdot \frac{\partial^2 u}{\partial x^2} \qquad \text{[eq.1]}$$

sendo que v é a velocidade de propagação da onda.

Um exemplo de uma equação parabólica é a equação da difusão:

$$\frac{\partial u}{\partial t} = -\frac{\partial}{\partial x}\left[D . \frac{\partial u}{\partial x} \right] \qquad \text{[eq.2]}$$

onde D é o coeficiente de difusão.

Por fim, um exemplo de uma equação elíptica é a equação de Poisson:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \qquad \text{[eq.3]}$$

onde $f(x,y)$ é o termo que representa a fonte. Supondo $f = 0$ temos a equação de Laplace:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \qquad \text{[eq.4]}$$

Esta classificação das equações diferenciais parciais em elípticas, hiperbólicas e parabólicas não é muito importante do ponto de vista computacional. Mais interessante é fazer uma distinção em termos de como elas evoluem:

- as hiperbólicas e as parabólicas basicamente representam problemas a valores iniciais
- as elípticas descrevem problemas a valores de contorno

Naqueles casos as equações descrevem evoluções temporais a partir de um determinado tempo inicial, enquanto que com as elípticas são descritas soluções estáticas da função `u` numa determinada região de interesse e elas são resolvidas numericamente iterando-se uma determinada solução até que ela convirja para a solução do problema (por isto a idéia de "relaxação").

A equação de Laplace em diferenças finitas

Quando uma função `q(x)` e suas derivadas são funções contínuas e finitas em `x`, então pelo Teorema de Taylor podemos escrever as seguintes expressões, onde as linhas (') representam as derivadas com relação à variável `x`:

$$q(x+h) = q(x) + hq'(x) + 1/2\ h^2q''(x) + 1/6\ h^3q'''(x) + \ldots$$

$$q(x-h) = q(x) - hq'(x) + 1/2\ h^2q''(x) - 1/6\ h^3q'''(x) + \ldots$$

Somando-se as duas expressões anteriores resulta:

$$q(x+h) + q(x-h) = 2q(x) + h^2 q''(x) + O(h^4)$$

onde $O(h^4)$ denota os termos contendo potências de h de ordem maior ou igual a 4. Supondo que h seja pequeno podemos escrever:

$$q''(x) = \{q(x+h) - 2q(x) + q(x-h)\}/h^2 \qquad \text{[eq.5]}$$

Suponha agora que u é uma função de duas variáveis, por exemplo, de x e y. Se

- dividirmos o plano x-y em retângulos de lados Δx e Δy, nas direções x e y respectivamente
- fizermos com que as coordenadas definidas pelas intersecções das linhas deste reticulado sejam denotadas

  por $x=l.\Delta x$ e $y=c.\Delta y$;
- denotarmos o valor da função u no ponto $P=(x=l.\Delta x, y=c.\Delta y)$ por $u_{l,c}$;

então, usando a equação de diferenças finitas para a segunda derivada de uma função [eq.5], podemos reescrever a equação de Laplace [eq.4] como:

$$u_{l,c} = \frac{\{u_{l-1,c} + u_{l+1,c} + u_{l,c-1} + u_{l,c+1}\}}{4} \qquad \text{[eq.6]}$$

que é a equação de Laplace em forma discretizada (diferenças finitas) que usaremos nos próximos capítulos.

Resolução numérica das EDP's

A resolução numérica das equações diferenciais parciais é um assunto bastante vasto e não seria possível, nem sequer desejável, apresentar aqui todos os métodos existentes, com todo o rigor, por mais tentador que isto possa parecer. Referências sobre eles existem em grande número [3], [4], [5], [6].

Pelo contrário, parece-nos muito mais útil apresentar o método no qual nos baseamos para resolver, num sistema de multiprocessadores, em paralelo, a equação de Laplace, apresentando também uma breve idéia dos métodos que o antecederam, de uma maneira lógica e direta.

Assim, podemos citar os métodos de Jacobi, Gauss-Seidel e SOR, que serão explicados a seguir, sempre aplicados à equação de Laplace.

O método de Jacobi

Denotando-se a n-ésima aproximação da equação de Laplace discretizada $u_{l,c}$ [eq.6] por $u^n_{l,c}$ então o método de Jacobi fornece a (n+1)-ésima aproximação de $u_{l,c}$ como sendo:

$$u^{n+1}_{l,c} = 1/4 \{u^n_{l-1,c} + u^n_{l+1,c} + u^n_{l,c-1} + u^n_{l,c+1}\} \qquad \text{[eq.7]}$$

que nada mais é do que uma média nos 4 vizinhos, o que pode ser visualizado na figura 1 abaixo:

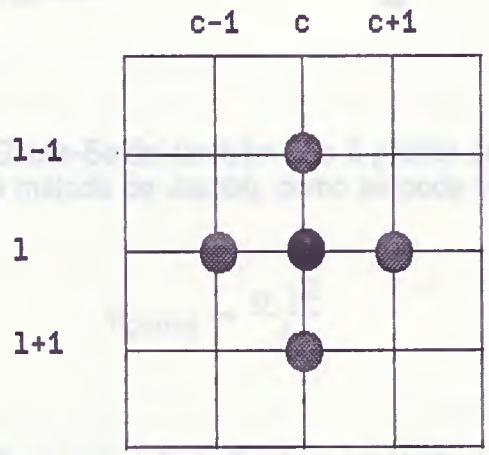


figura 1: cálculo do elemento $u_{l,c}$ em função dos seus 4 vizinhos mais próximos

Porém, sabe-se que o método de Jacobi não é prático pois converge muito lentamente. Mas quanto exatamente é "muito lentamente"? Para responder a esta pergunta podemos pensar em quantas iterações (N) será preciso efetuar para que o resultado tenha uma precisão de 10^{-p}:

$$N_{jacobi} \approx \frac{p.L^2}{2} \qquad \text{[eq.8]}$$

supondo-se que se está resolvendo o problema num reticulado de L x L pontos.

Assim, vemos que o número de iterações é proporcional ao número de pontos do reticulado (L^2), no método de Jacobi.

O método de Gauss-Seidel

O método de Gauss-Seidel é a mais próxima extensão do método de Jacobi que se pode imaginar: ao invés de se usar, para o cálculo do elemento u^{n+1}, somente elementos da iteração anterior (u^n), usam-se os elementos da iteração $n+1$ que já estiverem disponíveis.

Se os cálculos forem feitos, por exemplo, a partir da primeira linha e prosseguirem ao longo dela (da mesma forma que se lê um texto em português), temos a seguinte expressão para a $(n+1)$-ésima aproximação de $u_{l,c}$:

$$\left(u_{l,c}^{n+1}\right)_{gauss} = \frac{\{u_{l-1,c}^{n+1} + u_{l+1,c}^{n} + u_{l,c-1}^{n+1} + u_{l,c+1}^{n}\}}{4} \qquad \text{[eq.9]}$$

Porém o método de Gauss-Seidel também não é prático pois converge muito lentamente (apesar de ser mais rápido do que o método de Jacobi), como se pode ver pela expressão:

$$N_{gauss} \approx \frac{p.L^2}{4} \qquad \text{[eq.10]}$$

Note que há uma melhora de um fator 2 sobre o método de Jacobi, mas o número de iterações no método de Gauss-Seidel continua sendo proporcional ao número de pontos do reticulado (L^2).

O método SOR

O método SOR (*Simultaneous Over-Relaxation*) consiste em se fazer uma correção ao método de Gauss-Seidel da seguinte forma:

$$\left(u_{l,c}^{n+1}\right)_{SOR} = \omega.\left(u_{l,c}^{n+1}\right)_{gauss} + (1-\omega).u_{l,c}^{n} \qquad \text{[eq.11]}$$

onde ω é o parâmetro do método SOR e o índice "gauss" denota o valor calculado no método de Gauss-Seidel [eq.9].

Para a equação que estamos resolvendo (a equação de Laplace) pode-se calcular o seguinte valor ótimo para o parâmetro do método SOR [2]:

$$\omega = \frac{2}{\left[1 + \sqrt{1 - \rho_{jacobi}^{2}}\right]} \qquad \text{[eq.12]}$$

e

$$\rho_{jacobi} = 1 - \frac{\pi^{2}}{2L^{2}} \qquad \text{[eq.13]}$$

De maneira bem diferente dos métodos de Jacobi e de Gauss-Seidel, o número (N) de iterações necessárias para que a solução no Método SOR tenha precisão 10^{-p} é dado por:

$$N_{sor} \approx \frac{p.L}{3} \qquad \text{[eq.14]}$$

Comparando esta equação com as [eq.8] e [eq.10] notamos que há uma melhora substancial, pois naquelas o número de iterações é proporcional ao quadrado da dimensão L do reticulado enquanto que aqui é proporcional a L.

Uma vez que os programas desenvolvidos neste trabalho não usam reticulados quadrados (veja os capítulos "O Programa CALOR (Hospedeiro-Nós)" e "O Programa CALOR (Nós-Nós)") mostramos que caso o reticulado fosse retangular (L x C), teríamos para ρ_{jacobi} a seguinte expressão [3] :

$$\rho_{jacobi} = \frac{\cos\left(\frac{\pi}{C}\right) + \left(\frac{\Delta x}{\Delta y}\right)^{2} \cos\left(\frac{\pi}{L}\right)}{1 + \left(\frac{\Delta x}{\Delta y}\right)^{2}} \qquad \text{[eq.15]}$$

onde Δx e Δy são os espaçamentos nas direções x e y, respectivamente.

Ordenamento ímpar-par

Um ponto que merece certa atenção é a ordem como são calculados os novos pontos do reticulado: o modo mais simples, por exemplo, é o de se efetuar os cálculos de cima para baixo e da esquerda para a direita, da mesma forma como se lê um texto em português.

Porém, podemos dividir (imaginariamente) o reticulado em pontos pares e ímpares (como um tabuleiro de xadrez): as equações [eq.9] e [eq.11] nos mostram que os pontos pares somente dependem dos pontos ímpares e os ímpares somente dependem dos pares. Assim, podemos executar uma "semi-varrida" no reticulado atualizando os pontos ímpares e depois uma outra "semi-varrida" atualizando os pontos pares usando os novos valores dos pontos ímpares.

Este é o chamado ordenamento ímpar-par.

Os programas CALOR (veja os capítulos "O Programa CALOR (Hospedeiro-Nós)" e "O Programa CALOR (Nós-Nós)") implementam este tipo de ordenamento em seus cálculos.

Aceleração de Chebyshev para o SOR

Outro ponto que merece especial atenção é a convergência do método SOR. Sabe-se que, enquanto ω [eq.13] é o valor ótimo assintótico para o parâmetro de relaxação, ele não é necessariamente uma boa escolha para seu valor inicial.

A aceleração de Chebyshev usa o ordenamento ímpar-par mostrado acima e muda o valor de ω a cada semi-varrida de acordo com as expressões:

$$\omega^{(0)} = 1$$

$$\omega^{(1/2)} = \frac{2}{\left[1 + \sqrt{1 - \frac{\rho_{jacobi}^{2}}{2}}\right]}$$

e assim até:

$$\omega^{(n+1/2)} = \frac{1}{\left[1 - \sqrt{1 - \frac{\rho_{jacobi}^{2} \cdot \omega^{(n)}}{4}}\right]} \quad \text{para } n = 1/2,\ 1 \ldots$$

onde há a propriedade de que:

$$\omega^{(\infty)} \Rightarrow \omega_{ótimo}$$

Tendo entendido o método SOR com aceleração de Chebyshev, temos condições de entender os algoritmos usados na resolução, em paralelo, da equação de Laplace, apresentados nos capítulos "O Programa CALOR (Hospedeiro-Nós)" e "O Programa CALOR (Nós-Nós)".

Referências

[1] W.H. Press et al, *Numerical Recipes - The Art of Scientific Computing*, Cambridge University Press, 1987

[2] G.D. Smith, *Numerical solution of partial differential equations - finite difference methods*, Clarendon Press - Oxford, 1978, cap. 1 e 5

[3] G.E. Forsythe, *Finite-Difference Methods for Partial Differential Equations*, John Wiley & Sons, 1969

[4] R.Courant and D.Hilbert, *Methods of Mathematical Physics*, vol.2, Interscience, 1962

[5] B. Carnahan et al, *Applied Numerical Methods*, John Wiley & Sons, 1969, cap.7

[6] S.L.Ross, *Differential Equations*, John Wiley & Sons, 2nd edition, 1974

capítulo 7

O Programa CALOR (Hospedeiro-Nós)

O Programa CALOR (Hospedeiro-Nós)

Introdução

Este capítulo descreve o programa CALOR, desenvolvido e analisado neste trabalho, abordando a idéia básica por trás do seu algoritmo.

O termo "Hospedeiro-Nós" serve para diferenciar o programa tratado aqui (que usa extensivamente a comunicação entre o computador-hospedeiro e os nós processadores) daquele apresentado no capítulo "O Programa CALOR (Nós-Nós)", o qual usa um dos nós ACP como gerente, substituindo o MicroVAX II nesta tarefa.

Tanto as rotinas que rodam no computador-hospedeiro quanto as que rodam nos nós ACP são apresentadas, dando-se maior ênfase à parte de comunicação entre elas.

Também é descrito o modo como foi feita a paralelização do algoritmo, com tratamento minucioso da implementação da comunicação entre o computador-hospedeiro e os nós ACP.

Inclui-se nesta discussão a apresentação das estruturas de dados necessárias à comunicação hospedeiro-nós, além da apresentação das rotinas do sistema ACP usadas para tal.

No apêndice C encontram-se as listagens dos programas: o primeiro roda no computador-hospedeiro (CALOR_H_N_HOST.for) e o segundo nos nós ACP (CALOR_H_N_NODE.for).

Descrição geral

O programa CALOR (Hospedeiro-Nós) resolve, em paralelo, a equação de Laplace, usando o método SOR com aceleração de Chebyshev (veja o capítulo "Equações Diferenciais Parciais") num conjunto de nós do ACP, sendo que o controle do programa e a transmissão dos resultados (de e para os nós) está a cargo exclusivamente do computador-hospedeiro .

A idéia básica do algoritmo usado é a subdivisão do domínio no qual se deseja resolver a equação de Laplace: o domínio é subdividido (pelo hospedeiro) em tantas partes (ou subdomínios) quantos forem os nós ACP a serem usados na resolução do problema e distribuídas para eles. A seguir, cada um dos nós resolve a equação dentro do seu subdomínio. Numa etapa posterior, as bordas são trocadas entre nós vizinhos (ou seja, as condições de contorno para cada processador são redistribuídas) e o processo se repete até que uma condição de parada seja satisfeita.

O programa CALOR é composto de duas partes: uma (também chamada CALOR) que roda no computador-hospedeiro e outra que roda em cada um dos nós ACP:

• a parte que roda no computador-hospedeiro (o programa principal) é responsável pela iniciação geral, transmissão de parâmetros importantes para os nós, recebimento dos resultados dos nós e controle geral da execução do programa

• a parte que roda nos nós (sub-rotina) é encarregada da resolução da equação de Laplace propriamente dita

Abaixo temos um esquema geral do funcionamento do programa:

```
CALOR (roda no hospedeiro)
        * entrada dos parâmetros e iniciação
        * balanceamento estático de carga (subdivisão do domínio)
        * transmissão de parâmetros importantes a cada nó
        * repetir
            para cada um dos nós fazer
                transmitir suas bordas e
                autorizar o começo dos cálculos
            para cada um dos nós fazer
                receber um valor para usar como critério de parada e
                receber as bordas recém-calculadas pelo nó
          até que o critério de parada seja satisfeito
        * para cada um dos nós fazer
            receber os resultados dos subdomínios
        * imprimir os resultados


SORCHE (roda nos nós ACP)
        * se for a primeira chamada fazer
            iniciação geral e
            determinar seu subdomínio e
            determinar as condições de contorno
        * determinar em que posição está este nó
            se for superior então receber 1 borda (a inferior)
            se for do meio então receber 2 bordas (inferior e superior)
            se for inferior então receber 1 borda (a superior)
        * resolver a equação em seu subdomínio
        * calcular critério de parada e transmiti-lo
        * calcular próximos parâmetros do SOR
            se for superior então transmitir 1 borda ( a superior )
            se for do meio então transmitir 2 bordas (inferior e superior)
            se for inferior então transmitir 1 borda (a inferior)
        * arrumar variáveis necessárias
```

Você deve ter observado que a determinação das condições de contorno é feita pelos próprios nós e não pelo computador-hospedeiro (que, como dito anteriormente, é responsável, entre outras coisas, pela iniciação geral). Isto se deve ao fato de querermos minimizar a comunicação entre o hospedeiro e os nós ACP e, uma vez que os nós têm velocidades comparáveis à do hospedeiro (veja o capítulo "O Sistema ACP"), não há prejuízos.

A Idéia da subdivisão do domínio

O algoritmo usado na resolução em paralelo do problema está baseado no fato de se poder resolver o problema original subdividindo-se seu domínio em vários subdomínios e então se resolver o problema, em paralelo, em cada um dos subdomínios menores.

Uma vez que o método SOR com aceleração de Chebyshev já foi visto no capítulo "Equações Diferenciais Parciais", vamos agora mostrar como foi feita esta subdivisão do domínio original (discretizado), para depois podermos explicar o porquê de se fazer comunicação entre o hospedeiro e os nós.

Suponha que o domínio original (aquele no qual se deseja resolver a equação de Laplace) seja representado pela figura 1, mostrada abaixo:

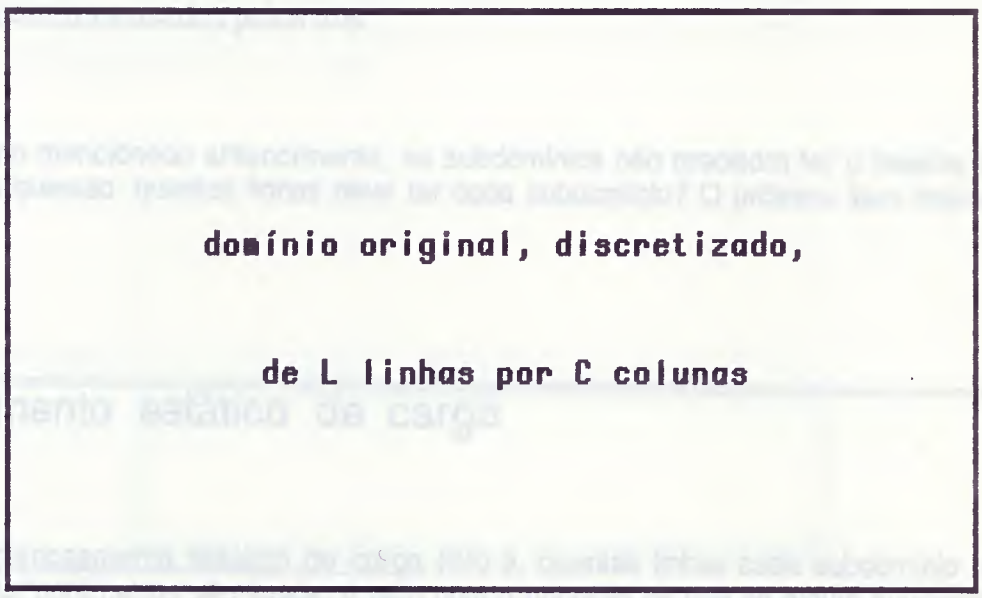


figura 1: domínio original, discretizado

Note que a figura 1 representa um domínio já discretizado , de L linhas por C colunas, com condições de contorno representadas pela borda da figura.

Ao invés de resolvermos o problema no domínio acima com um único processador, podemos resolvê-lo em paralelo, com o uso de vários processadores, bastando para isto subdividir o domínio original em vários (tantos quantos forem os processadores a serem usados) subdomínios . Cada um destes recém-criados subdomínios tem o mesmo número de colunas que o domínio original (C) mas não tem o mesmo número de linhas (L). Aliás, dependendo do número de linhas do domínio original e do número de processadores usados, os subdomínios terão números diferentes de linhas uns dos outros. Veja a figura 2 a seguir, onde o domínio original foi subdividido em p subdomínios:

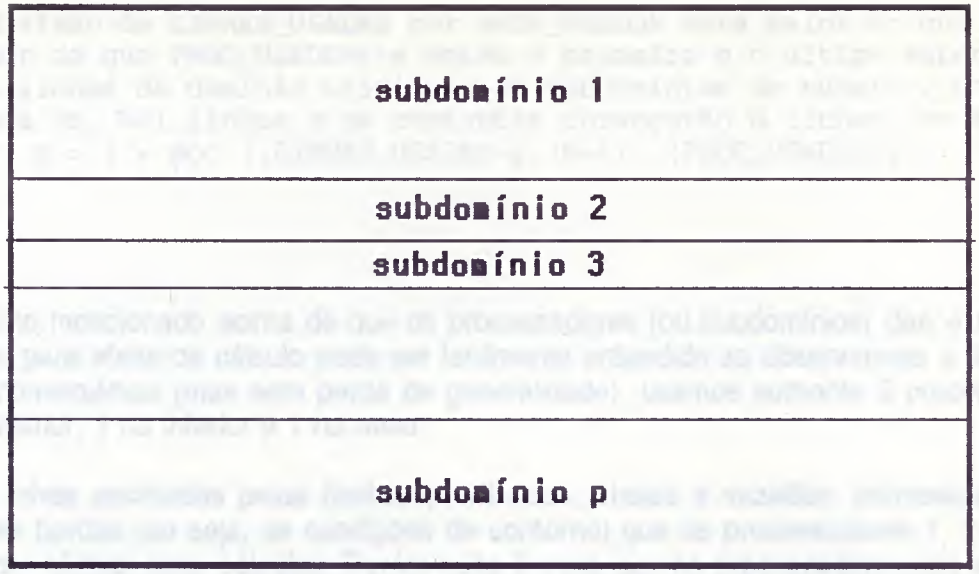


figura 2: os p subdomínios usados pelos nós

Como mencionado anteriormente, os subdomínios não precisam ter o mesmo número de linhas. Mas então surge a questão: quantas linhas deve ter cada subdomínio? O próximo item responde a esta pergunta.

O balanceamento estático de carga

O balanceamento estático de carga (isto é, quantas linhas cada subdomínio recebe, uma vez que todos os demais parâmetros são iguais) é feito com o princípio de que se algum subdomínio tiver que ter linhas a mais, estes deverão ser os subdomínios das extremidades (pois, para efeito de cálculos, eles têm uma borda a menos. Veja discussão mais abaixo).

Além deles, havendo ainda necessidade de alguns subdomínios terem mais linhas, estes deverão ser escolhidos seqüencialmente (pois não há motivo para não ser assim) a partir do subdomínio número 2. Formalizando em termos de regras mais explícitas, temos:

• se o número de linhas do domínio original (LINHAS_USADAS) for divisível pelo número de processadores (PROC_USADOS), então cada subdomínio abrangerá exatamente N linhas do domínio original, com N = LINHAS_USADAS div PROC_USADOS.Senão,

• se o resto da divisão de LINHAS_USADAS por PROC_USADOS for igual a 1,então o primeiro subdomínio abrangerá N+1 linhas do domínio original e os demais subdomínios abrangerão, cada um, N linhas, com N dado acima. Senão,

• se o resto da divisão de LINHAS_USADAS por PROC_USADOS for igual a 2, então o primeiro e o último subdomínios abrangerão, cada um, N+1 linhas do domínio original e os demais subdomínios abrangerão,cada um, N linhas, com N dado acima. Senão,

• o resto da divisão de LINHAS_USADAS por PROC_USADOS será maior do que 2 (e certamente menor do que PROC_USADOS) e então o primeiro e o último subdomínios abrangerão N+1 linhas do domínio original; os subdomínios de número 2 até m também abrangerão, cada um, N+1 linhas e os restantes abrangerão N linhas,com N dado acima e com m dado por: m = 1 + mod {[LINHAS_USADAS-2.(N+1)],(PROC_USADOS-2)}.

O fato mencionado acima de que os processadores (ou subdomínios) das extremidades têm uma linha a menos para efeito de cálculo pode ser facilmente entendido se observarmos a figura 3 mostrada abaixo onde, por conveniência (mas sem perda de generalidade), usamos somente 3 processadores isto é, 1 na extremidade superior, 1 na inferior e 1 no meio:

As linhas apontadas pelas flechas pontilhadas, cheias e vazadas, representam, respectivamente, as bordas (ou seja, as condições de contorno) que os processadores 1, 2 e 3 precisam conhecer para poder efetuar seus cálculos. Porém salta à vista que os processadores das extremidades (neste caso, 1 e 3) já têm todas as linhas menos 1 (respectivamente a marcada pela flexa pontilhada inferior, e a marcada pela flexa vazada superior) enquanto que o processador do meio (o de número 2) precisa ainda receber 2 linhas (marcadas pelas flexas cheias).

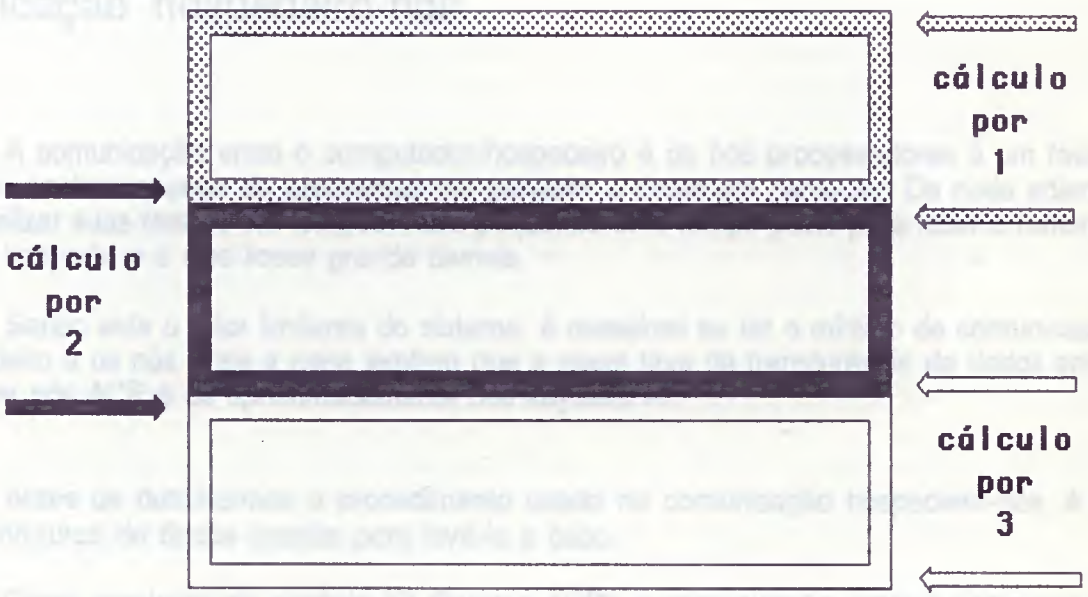


figura 3: cálculo do número de linhas

Isto significa que os processadores localizados nas extremidades têm que fazer seus cálculos em 1 linha a menos do que os processadores localizados no meio. Lembre-se (conforme mostrado no capítulo "Equações Diferenciais Parciais") de que os valores nas bordas não são recalculados, mas somente o interior das matrizes, pois aquelas representam as condições de contorno do problema (supostas fixas).

Assim, quando do balanceamento de carga, é natural dar-se prioridade para um maior carregamento (em termos de linhas que estiverem sobrando) dos processadores das extremidades, exatamente o que determinam as regras formalizadas acima. A sub-rotina CARGA, definida no programa CALOR (na parte que roda no computador-hospedeiro), implementa o algoritmo mostrado acima.

Na primeira iteração é fácil saber as condições de contorno de cada um dos processadores, mas pode-se perguntar quais "condições de contorno" os processadores usam nas iterações posteriores.

A idéia aqui é bastante simples:

- na iteração n, a borda inferior (superior) a ser usada pelo processador p é a linha 2 ($N\_ELEMENTOS_{p-1}$) do processador p+1 (p-1) calculada na iteração n-1, onde $N\_ELEMENTOS_p$ é o número de linhas que tem o processador p. Fica subentendido que o processador 1 não tem alterada sua borda superior e que o último processador não tem alterada sua borda inferior, pois são as condições de contorno do domínio original, supostas fixas.

O fato de a borda de um processador ser calculada por outro é o ponto mais delicado da resolução em paralelo do problema, uma vez que ele requer a transmissão de valores calculados por um processador para (até dois) outros, realizada através da comunicação entre o computador-hospedeiro e os nós ACP.

A comunicação hospedeiro-nós

A comunicação entre o computador-hospedeiro e os nós processadores é um fator importantíssimo no desempenho de programas em paralelo, e deste em particular. De nada adiantaria ter nós capazes de realizar suas tarefas em tempos muito pequenos se o tempo gasto para fazer a transmissão dos dados entre o hospedeiro e eles fosse grande demais.

Sendo este o fator limitante do sistema, é desejável se ter o mínimo de comunicação possível entre o hospedeiro e os nós. Vale a pena lembrar que a maior taxa de transferência de dados entre o hospedeiro e os nós ACP é de aproximadamente 500 kbytes/s [2].

Antes de detalharmos o procedimento usado na comunicação hospedeiro-nós, é interessante observar as estruturas de dados usadas para levá-la a cabo.

Como mostrado no capítulo "O Sistema ACP", a comunicação entre o computador-hospedeiro e os nós ACP é feita através de blocos de variáveis tipo *common* (COMMON BLOCKS normais do FORTRAN) definidas da mesma maneira no programa principal (CALOR) e na sub-rotina que roda nos nós (SORCHE) e declaradas no arquivo tipo UPF (*User's Parameter File*) correspondente.

No arquivo tipo UPF (veja apêndice C) associado ao programa CALOR temos a seguinte definição de blocos a serem usados na comunicação Hospedeiro-Nós:

```
node blocks = 1:PARAME, 2:BORDAS, 3:MATRIZ, 4:DIFERE, 5:VARRE
```

sendo que os blocos descritos acima referem-se às seguintes estruturas de dados:

- PARAME
  N_LINHAS : número de linhas em que foi subdividido o domínio original.
  N_COLUNAS: número de colunas em que foi subdividido o domínio original.
  N_DO_PROC: indica por qual número o nó é reconhecido pelo hospedeiro.
  POSICAO : indica a posição da sub-matriz do nó com relação à matriz original.
  PTR_SUP : variável que aponta, na matriz domínio, onde começa a parte destinada ao nó.
  PTR_INF : variável que aponta, na matriz domínio, onde termina a parte destinada ao nó.

- BORDAS
  DUAS_BORDAS: vetor que contém 2 bordas (de tamanho HOST_COLUNAS cada) e que são trocadas pelos nós.

- MATRIZ
  M_ANT: sub-matriz do nó. Contém a resposta do problema, resolvido no nó.

- DIFERE
  EPSILON: número calculado pelo nó, usado como critério de parada.

- VARRE
  ENE: indica a iteração. Usado pelo SOR com aceleração de Chebyshev.

Os blocos PARAME e VARRE são usados para fazer a comunicação unidirecional no sentido hospedeiro para os nós ACP; o bloco DIFERE para a comunicação unidirecional no sentido nós ACP para o hospedeiro e, finalmente, os blocos BORDAS e MATRIZ são usados na comunicação bidirecional.

Para se transmitir um bloco de dados do hospedeiro para um nó específico (como é o caso no programa CALOR), usamos a rotina ACP_SENDBLOCK (como mostrado no capítulo "O Sistema ACP", a rotina ACP_SENDEVENT envia um bloco de dados para o primeiro nó disponível que encontrar e não para um nó específico).

Para se receber um bloco de dados de um determinado nó usamos a rotina ACP_GETBLOCK.

As sintaxes completas de ambas as rotinas ACP_SENDBLOCK e ACP_GETBLOCK estão explicadas na referência *ACP Software User's Guide for Event Oriented Processing* [1]. A seguir mostramos apenas com quais parâmetros chamar estas rotinas para alguns casos por nós utilizados:

- enviar um BLOCO de dados correspondente a uma MATRIZ de dados de um determinado TIPO, por exemplo, números reais, de TAMANHO determinado a um PROCESSADOR específico pertencente a uma certa CLASSE e ainda avisar este determinado processador que ele ainda não deve iniciar seu programa pois, por exemplo, mais dados lhe serão enviados:

```
ACP_SENDBLOCK(BLOCO,MATRIZ,TAMANHO,PROCESSADOR,CLASSE,ACP$NO_GO,SEND_DONE)
```

- idem acima, mas avisando o processador que ele deve começar a executar seu programa:

```
ACP_SENDBLOCK(BLOCO,MATRIZ,TAMANHO,PROCESSADOR,CLASSE,ACP$GO,   SEND_DONE)
```

- receber um BLOCO de dados correspondente a uma MATRIZ de dados de um determinado TIPO, por exemplo, números reais, de TAMANHO determinado, de um PROCESSADOR específico pertencente a uma certa CLASSE e ainda avisar este determinado processador que ele ainda não deve ser considerado disponível para receber mais tarefas:

```
ACP_GETBLOCK(BLOCO,MATRIZ,TAMANHO,PROCESSADOR,CLASSE,ACP$MORE_BLOCKS,GET_DONE)
```

- idem acima, mas avisando o processador que ele já pode ser considerado disponível para receber mais tarefas:

```
ACP_GETBLOCK(BLOCO,MATRIZ,TAMANHO,PROCESSADOR,CLASSE,ACP$LAST_BLOCK, GET_DONE)
```

As 2 rotinas mostradas acima (ACP_SENDBLOCK e ACP_GETBLOCK) são bastante poderosas e ainda assim simples de serem usadas: note que apenas trocando 2 constantes pré-definidas (o par ACPGO / ACPNO_GO e o par ACP$MORE_BLOCKS / ACP$LAST_BLOCK) tem-se controle total da execução (ou não) da rotina que roda no nó ACP.

As variáveis lógicas SEND_DONE e GET_DONE têm o mesmo significado mostrado anteriormente (veja o capítulo "O Sistema ACP"), bem como a definição do termo "classe".

As conversões de formato dos dados

Como visto anteriormente, transmitir (receber) dados para (de) um determinado nó é bastante fácil, porém pode-se perguntar como eles são transmitidos uma vez que, como sabemos, os nós ACP usam um determinado tipo de microprocessador diferente daquele do computador-hospedeiro, sendo que ambos têm, evidentemente, formatos de dados diferentes.

A resposta é surpreendentemente simples: a transmissão (recepção) é feita sobre os dados da maneira que eles estão, sem modificação em seus formatos. Cabe ao programa principal (o que roda no hospedeiro) ou ao programa do nó, ou até mesmo a ambos, converter os dados para formatos com os quais possam lidar.

Isto permite que, uma vez fixado o formato que se quer usar (por exemplo o do nó ACP), se o hospedeiro somente precisar receber dados de um processdor e transmiti-los a outro(s) sem necessidade de interpretá-los, ele poderá estar sempre trabalhando com os dados no formato do nó, eliminando assim constantes conversões entre formatos de dados. Isto, aliás, foi usado no programa CALOR.

Existem dois pares de rotinas (um que roda no hospedeiro e outro que roda nos nós ACP) que realizam conversões de formatos de dados. A seguir temos uma breve descrição delas. Uma descrição mais detalhada e de como se faz a conversão entre formatos do VAX e do MC68020 (IEEE) pode ser encontrada na referência *ACP Software User's Guide for Event Oriented Processing* [1]:

• conversão hospedeiro→nó ACP:

roda no hospedeiro: `ACP_CONVERT_H_N`
roda no nó ACP : `ACPCHN`

• conversão nó ACP→hospedeiro:

roda no hospedeiro: `ACP_CONVERT_N_H`
roda no nó ACP : `ACPCNH`

que têm como parâmetros (entre outros) a matriz que contém os dados a serem convertidos e o tipo dos dados (por exemplo, números reais).

Referências

[1] *ACP Software User's Guide for Event Oriented Processing*, Advanced Computer Program, FERMILAB, Revision 3.0, December 4, 1987

[2] J.R.Biel, *Introduction to ACP software for event oriented processing*, curso dado no CBPF, april 4, 1988

capítulo 8

O Programa CALOR (Nós-Nós)

O Programa CALOR (Nós-Nós)

Introdução

Neste capítulo é descrito o programa HOSPEDEIRO, que resolve a equação de Laplace num multiprocessador ACP, implementando comunicação direta entre os nós processadores.

Além do algoritmo usado na resolução da equação de Laplace propriamente dita, são descritas as rotinas do sistema ACP usadas na comunicação direta entre os nós processadores.

No apêndice D encontram-se as listagens dos programas: o primeiro roda no computador-hospedeiro (HOSPEDEIRO.for); o segundo nos vários nós-de-cálculo (CALOR.for1), e o terceiro no processador-mestre (MESTRE.for2). Veja no próximo item a definição de processador-mestre e de nós-de-cálculo

A idéia básica

Até agora a resolução da equação de Laplace no ACP foi feita com uma grande participação do computador-hospedeiro (veja o capítulo "O programa CALOR (Hospedeiro-Nós)"): era ele quem fazia a constante troca de bordas entre os nós processadores, o que requeria uma grande comunicação entre ele e os nós, degradando desta forma o desempenho geral do programa.

Assim sendo, decidimos reduzir esta comunicação (a fim de tentar obter um menor tempo de processamento) e resolvemos substituir o computador-hospedeiro, nas tarefas em que isto fosse possível, por um dos processadores ACP, chamado de **nó-mestre** (ou **processador-mestre**).

A seguir, nas figura 1 a 3, temos os diagramas de blocos dos programas que rodam (i) no computador-hospedeiro, (ii) no nó-mestre e (iii) nos nós-de-cálculo, respectivamente.

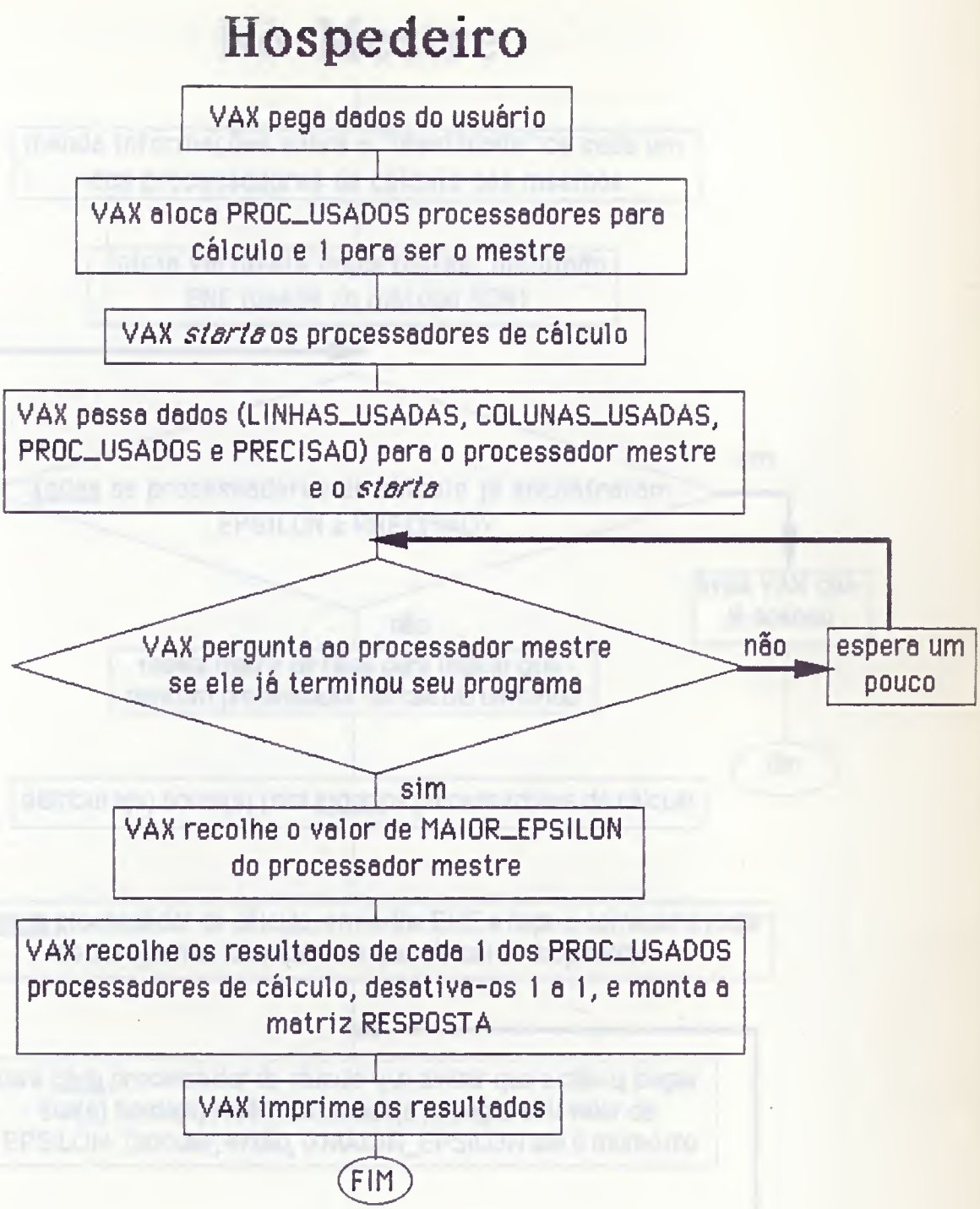


figura 1: diagrama de blocos do programa que roda no computador-hospedeiro

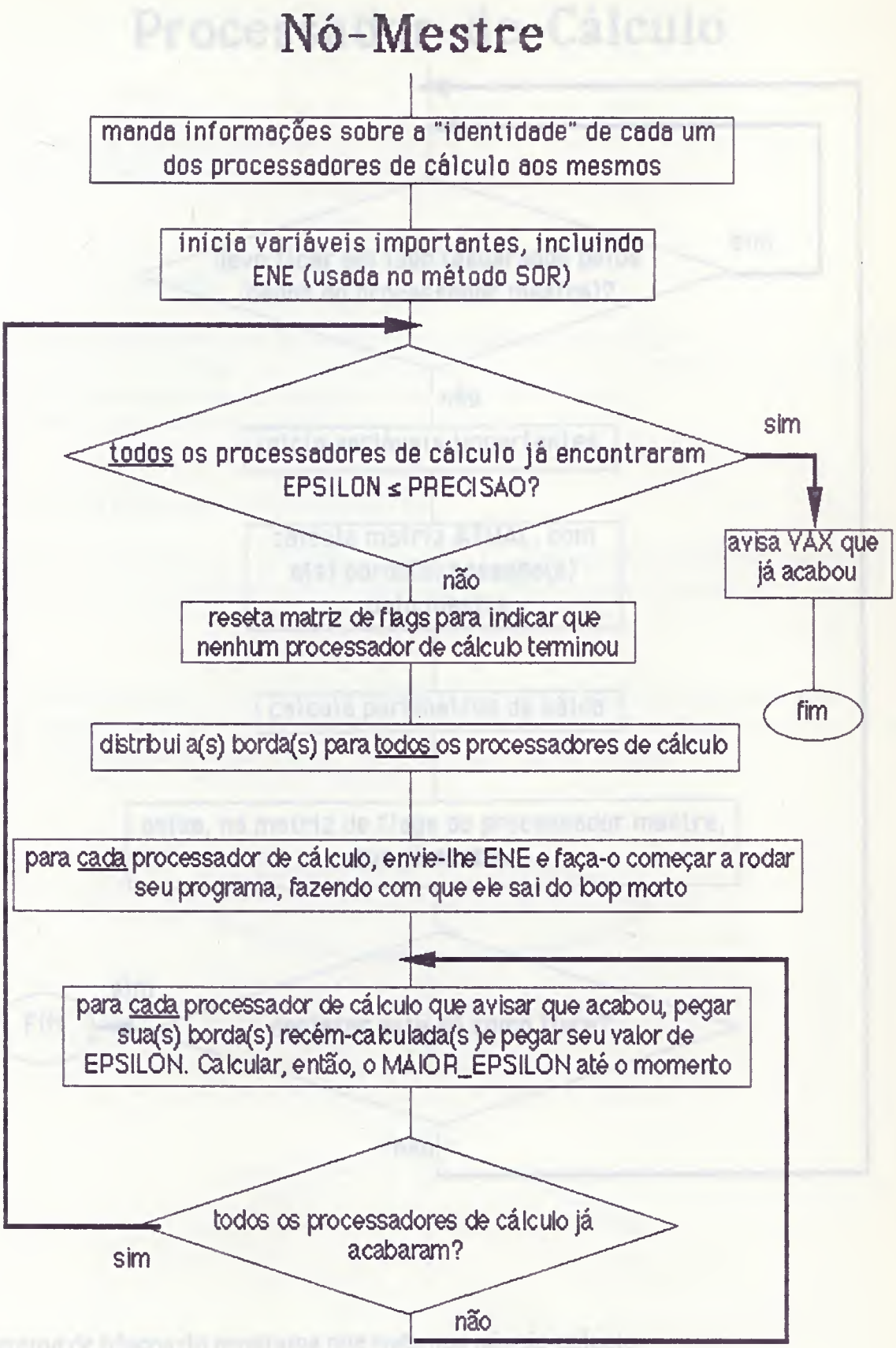


figura 2: diagrama de blocos do programa que roda no processador-mestre

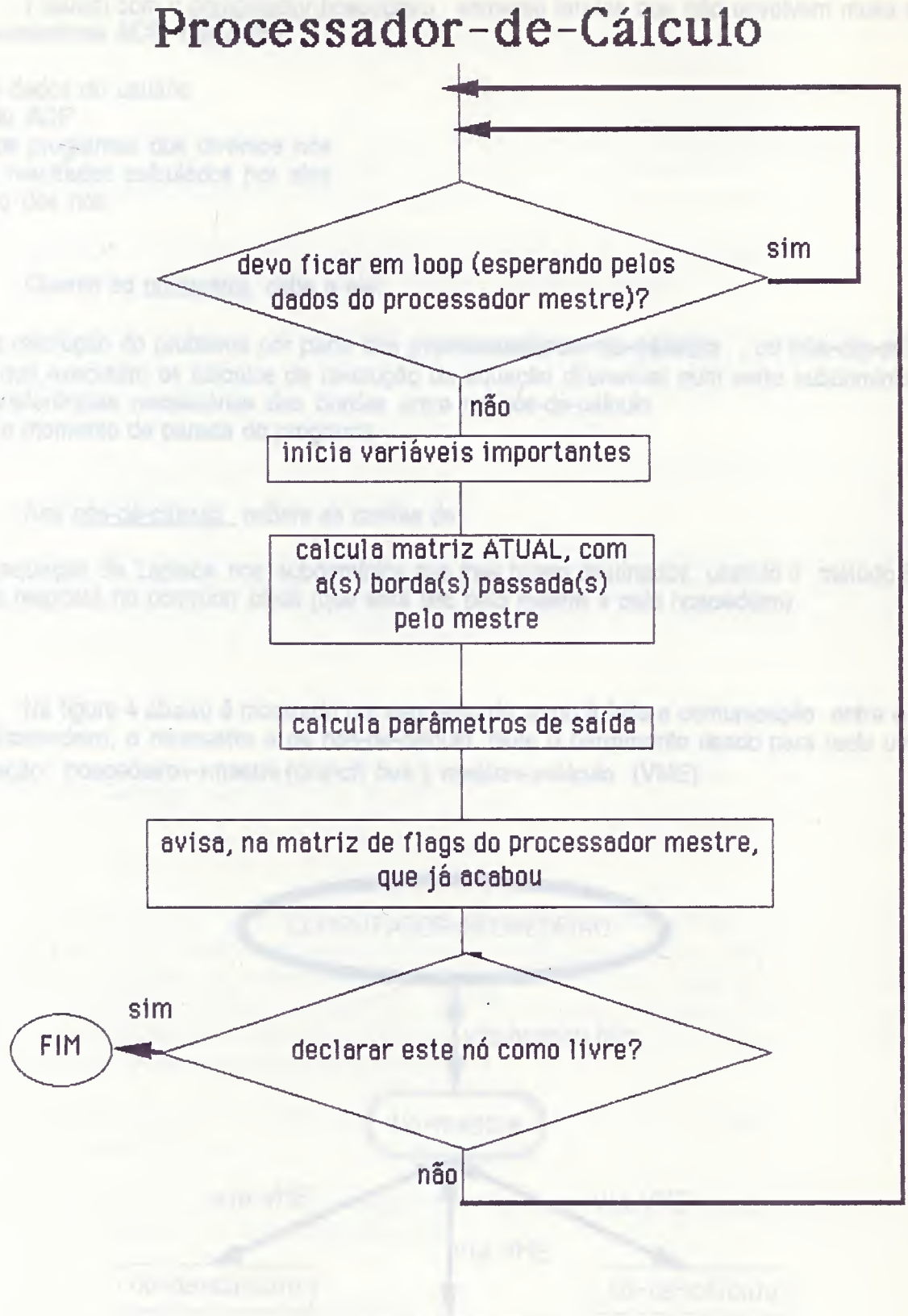


figura 3: diagrama de blocos do programa que roda nos nós-de-cálculo

Ficaram com o computador-hospedeiro somente tarefas que não envolvem muita comunicação com os processadores ACP, tais como:

- entrada de dados do usuário
- iniciação do ACP
- ativação dos programas dos diversos nós
- coleta dos resultados calculados por eles
- desativação dos nós

Quanto ao nó-mestre, cabe a ele:

- monitorar a resolução do problema por parte dos processadores-de-cálculo , ou nós-de-cálculo (os nós ACP que executam os cálculos da resolução da equação diferencial num certo subdomínio)
- fazer as transferências necessárias das bordas entre os nós-de-cálculo
- determinar o momento de parada do programa

Aos nós-de-cálculo cabem as tarefas de:

- resolver a equação de Laplace nos subdomínios que lhes forem destinados, usando o método SOR
- colocar sua resposta no *common block* (que será lido pelo mestre e pelo hospedeiro)

Na figura 4 abaixo é mostrado um esquema de como é feita a comunicação entre o computador-hospedeiro, o nó-mestre e os nós-de-cálculo. Note o barramento usado para cada um dos tipos de comunicação: hospedeiro↔mestre (*branch bus*), mestre↔cálculo (VME).

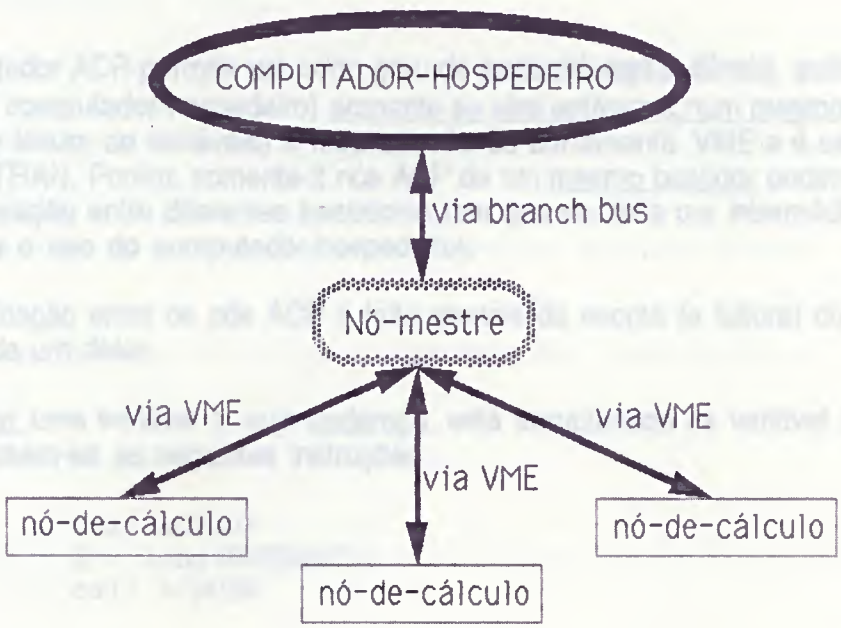


figura 4: esquema da comunicação entre o computador-hospedeiro, o nó-mestre e os nós-de-cálculo

Nós de diferentes classes

Até agora nos referimos ao nó-mestre e aos nós-de-cálculo sem dizer explicitamente quem eles são. Do ponto de vista físico (isto é, de *hardware*) não há diferença entre quem é quem. Já do ponto de vista lógico há: o nó-mestre executa um programa que controla a execução dos programas dos nós-de-cálculo.

Isto equivale a dizer que os nós-de-cálculo pertencem à uma determinada classe e o nó-mestre a outra (veja a discussão sobre classe de nós no capítulo "O Sistema ACP").

No início do programa são alocados `PROC_ALOCADOS` processadores (conforme definido no arquivo tipo UPF e na constante `PROC_ALOCADOS` nos programas) para uso. Dos `PROC_ALOCADOS` processadores alocados, o programa permite que, no máximo, `PROC_ALOCADOS-1` sejam usados como processadores-de-cálculo, pois sempre reserva 1 para ser o nó-mestre.

Isto significa que se o usuário desejar resolver seu problema dividindo o domínio original em `r` subdomínios, ele deverá alocar `r+1` processadores ACP.

Vale a pena ressaltar que todos os PROC_ALOCADOS processadores têm que estar num mesmo bastidor (veja a razão disto no item "Comunicação nó-nó" na próxima página). Assim, conforme explicado no capítulo "O Sistema ACP", podemos, no máximo, dividir o domínio original em 16 subdomínios, pois num mesmo bastidor cabem, no máximo, 17 nós ACP.

Comunicação direta entre os nós ACP

O computador ACP permite um certo grau de **comunicação direta entre os nós** (isto é, sem o envolvimento do computador-hospedeiro) somente se eles estiverem num mesmo bastidor [1]. Esta comunicação (escrita ou leitura de variáveis) é feita através do barramento VME e é executada com o uso da instrução `long` do FORTRAN. Porém, somente 2 nós ACP de um mesmo bastidor podem se comunicar ao mesmo tempo: a comunicação entre diferentes bastidores tem que ser feita por intermédio do *branch bus* (e, conseqüentemente, com o uso do computador-hospedeiro).

A comunicação entre os nós ACP é feita através da escrita (e leitura) de variáveis diretamente na memória local de cada um deles.

Para se ler uma variável `I`, cujo endereço está armazenado na variável `ENDERECO`, a partir de um outro processador, usam-se as seguintes instruções:

```
call ACPSUP
I = long(ENDERECO)
call ACPUSR
```

com `I` e `ENDERECO` inteiros.

As chamadas de ACPSUP e ACPUSR servem para habilitar e desabilitar, respectivamente, a leitura (ou escrita) de variáveis fora de um determinado nó ACP, a partir de um outro nó.

Porém, como a função long somente lê números de 4 *bytes*, são necessárias 2 leituras para se ler um número real (de 8 *bytes*). Assim, para se ler a variável R (real, de 8 *bytes*) usa-se, por exemplo, o seguinte código:

```
real*8 R
integer RAUX(2), ENDERECO
equivalence (R, RAUX)
      :
call ACPSUP
RAUX(1) = long(ENDERECO)
RAUX(2) = long(ENDERECO+4)
call ACPUSR
```

Note que lançamos mão da instrução equivalence, o que torna possível o acesso, como real*8, aos 2 números inteiros lidos com o uso das instruções long. Observe também o incremento adequado (neste caso, 4) no valor do argumento da segunda instrução long.

De modo análogo, a escrita da variável I na variável cujo endereço é ENDERECO se dá com:

```
call ACPSUP
long(ENDERECO) = I
call ACPUSR
```

Cálculo dos endereços VME

Para localizarmos o endereço de um certo *common block* precisamos calcular 2 valores:

- o endereço (VME) do nó de onde queremos ler (ou escrever)
- o deslocamento, com relação ao endereço deste nó, do *common block* em questão

O endereço (VME) de um nó pode ser calculado com o uso da função:

```
ACP_NODE_VME_ADDRESS(NO)
```

que aceita como único parâmetro o número do nó (NO) de interesse e retorna seu endereço (no barramento VME) como um número inteiro.

Já o deslocamento, com relação ao endereço (VME) de um nó, de um *common block* é calculado pela função:

```
ACP_NODE_BLOCK_OFFSET(A_CLASSE, O_BLOCO)
```

que tem como parâmetros a classe (A_CLASSE) do nó e o *common block* (O_BLOCO) que se deseja acessar. Note que o número do *common block* passado como parâmetro aqui deve ser o mesmo que o definido no arquivo tipo UPF (*User's Parameter File*) correspondente.

Desta forma, o endereço (ENDERECO) da variável A (veja sua definição abaixo), localizada no nó NO pertencente à classe CLASSE, é calculado com:

```
integer A(22)
common /AJUDA/ A
parameter (AJUCOM = 1)
      :
ENDERECO = ACP_NODE_VME_ADDRESS(NO) +
           ACP_NODE_BLOCK_OFFSET(CLASSE, AJUCOM)
```

sendo que o arquivo UPF correspondente tem a linha:

```
                :
node blocks = 1:AJUCOM ...
                :
```

Supondo que a figura 5, mostrada abaixo, represente a memória de m nós ACP, notamos que as funções ACP_NODE_VME_ADDRESS e ACP_NODE_BLOCK_OFFSET nada mais nos fornecem do que um apontador para estas estruturas de dados e um deslocamento a partir dele, respectivamente:

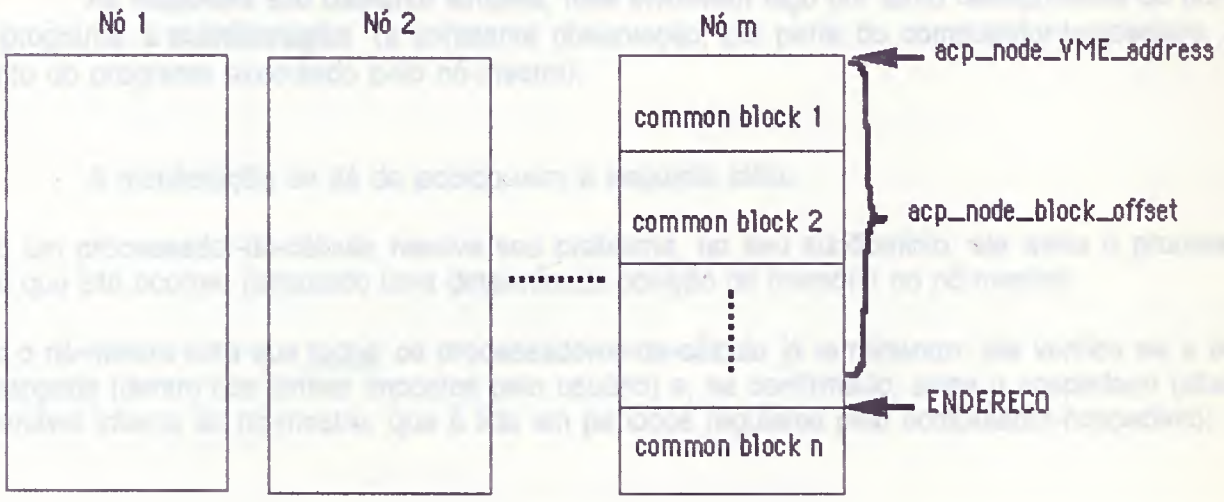


figura 5: esquema da memória dos nós ACP, mostrando como se calcula o endereço de uma certa variável, como usodas funções ACP_NODE_VME_ADDRESS e ACP_NODE_BLOCK_OFFSET

Observe, porém, que ENDERECO somente representa o endereço da primeira variável do *common block* (no exemplo acima, somente da variável A(1)). As variáveis seguintes devem ser acessadas através de incrementos adequados no valor de ENDERECO.

Uma vez que o método de se calcular os endereços (descrito acima) tem que ser feito pelo computador-hospedeiro (pois as funções ACP_NODE_VME_ADDRESS e ACP_NODE_BLOCK_OFFSET somente estão disponíveis para ele) e levando-se em conta que são os nós ACP que deles farão uso (para implementar a comunicação direta entre os nós) é necessário que estes endereços sejam conhecidos pelos nós, tanto o mestre quanto os de cálculo. Para isto, é transmitido (pelo hospedeiro, para todos os nós ACP) um *common block* (USADOS, como mostrado na listagem que se encontra no apêndice D) que contém o endereço da primeira variável de todos os *common blocks* a serem referenciados, além de outras informações importantes.

Monitorações

Como dito anteriormente, a maior parte das tarefas executadas pelo programa é levada a cabo pelos nós ACP (tanto os de cálculo quanto o mestre). Porém, ficou a cargo do computador-hospedeiro a importante tarefa do recolhimento dos resultados, o que evidentemente só pode ser feito quando os nós-de-cálculo já tiverem encontrado a solução do problema.

Mas aí então surgem as questões:

- quem determina se os nós-de-cálculo já encontraram a solução?
- quando é que os resultados podem ser recolhidos pelo computador-hospedeiro?

As respostas são bastante simples, mas envolvem algo um tanto desagradável do ponto de vista do programa: a **monitoração** (a constante observação, por parte do computador-hospedeiro, do andamento do programa executado pelo nó-mestre).

A monitoração se dá de acordo com a seguinte idéia:

- quando um processador-de-cálculo resolve seu problema, no seu subdomínio, ele avisa o processador-mestre que isto ocorreu (alterando uma determinada posição de memória no nó-mestre)

- quando o nó-mestre nota que todos os processadores-de-cálculo já terminaram, ele verifica se a solução já foi alcançada (dentro dos limites impostos pelo usuário) e, se confirmado, avisa o hospedeiro (alterando uma variável interna ao nó-mestre, que é lida em períodos regulares pelo computador-hospedeiro).

Vale a pena ressaltar que a monitoração é usada pois não há nenhuma maneira de se gerar uma interrupção no hospedeiro (vinda de um nó ACP). Se isto fosse possível, poderíamos colocar o programa do hospedeiro em hibernação e ele seria acordado por uma interrupção gerada pelo nó-mestre, assim que este verificasse que os nós-de-cálculo tivessem resolvido o problema.

Ativação e desativação dos programas dos nós

Os programas dos nós ACP têm sua execução ativada através do computador-hospedeiro com o uso da instrução ACP_SENDBLOCK (como descrito no capítulo "O Programa CALOR (Hospedeiro-Nós)"). Daí para a frente quem monitora os nós-de-cálculo é o processador-mestre.

Porém, uma vez que um nó ACP (no caso o nó-mestre) não pode ativar um programa de um outro nó (no caso, dos nós-de-cálculo), não podemos permitir que os processadores-de-cálculo cheguem ao fim de seus programas (à instrução RETURN, mais especificamente), pois aí seus programas terminariam e eles teriam que ser novamente ativados pelo hospedeiro, aumentando a comunicação hospedeiro-nós, exatamente o que desejamos minimizar (veja no item "Tempo máximo de execução dos programas dos nós ACP", mais adiante, uma modificação que tem que ser acrescentada ao arquivo UPF quando desejamos que os programas dos nós ACP rodem por muito tempo).

Assim, a estratégia usada é colocar os nós-de-cálculo num laço e, através de um aviso dado pelo mestre, liberá-los para começar seus cálculos, mas com a condição de fazê-los voltar a este laço assim que terminá-los.

Isto é bastante claro, mas pode-se perguntar quem cuida da desativação dos programas dos nós (tanto os de cálculo quanto o mestre). Se nos lembrarmos de que é o computador-hospedeiro quem recolhe os resultados dos diversos subdomínios (ou seja, dos diversos nós-de-cálculo) então notaremos que esta seria uma ótima oportunidade para desativar os nós-de-cálculo (e o mestre também!). De fato, a instrução ACP_GETBLOCK (veja o capítulo "O Programa CALOR (Hospedeiro-Nós)"), usada para a coleta dos resultados do nó-mestre (e dos processadores-de-cálculo), é chamada com o parâmetro ACP$LAST_BLOCK, a fim de desativar todos os nós.

Formato dos dados

Uma vez que toda a parte de cálculo e controle dos resultados se dá nos processadores ACP, o formato dos dados utilizado é o entendido por eles. Isto elimina as conversões entre formatos, as quais são feitas somente quando o hospedeiro:

- ativa os processadores ACP
- lê (do nó-mestre) a variável que indica que a solução foi alcançada
- recolhe os resultados dos diversos subdomínios

Como a conversão de inteiros é nil (veja a referência *ACP Software User's Guide for Event Oriented Processing* [2]), há na verdade somente a conversão dos resultados (que são números reais) obtidos pelos nós-de-cálculo e do valor MAIOR_EPSILON encontrado pelo nó-mestre (que não é importante e poderia ser retirado do código do computador-hospedeiro, pois serve apenas para indicar o término do programa ao usuário: contém a maior diferença na última iteração do método SOR).

Definição do arquivo tipo UPF

Já foi dito no capítulo "O Sistema ACP" que o arquivo tipo UPF (*User's Parameter File*) serve para configurar o sistema a ser usado, tal como especificar o número de nós ACP a ser alocado, os *common blocks* a serem usados, os programas a serem carregados etc. Vejamos agora outras informações que precisamos sejam transmitidas ao sistema ACP para implementarmos a comunicação direta entre os nós.

Atribuindo programas diferentes a diferentes nós

Como vimos, o nó-mestre e os nós-de-cálculo executam programas diferentes. Quais são estes programas é uma das informações que deve estar presente no arquivo UPF correspondente. Uma vez que os nós-de-cálculo têm sempre o mesmo programa (no caso, `CALCULO.for1`) e o nó-mestre um outro programa (no caso, `MESTRE.for2`), é necessário separar estes nós em 2 classes:

- os nós-de-cálculo pertecem à classe 1
- o nó-mestre pertence à classe 2

daí a presença dos sufixos "`.for1`" e "`.for2`" nos nomes dos arquivos do programa dos nós-de-cálculo e do nó-mestre, respectivamente.

Desta forma, para se especificar o programa que cada classe irá executar, usamos os seguintes comandos no arquivo UPF:

```
class 1:node source = CALCULO.for1
class 2:node source = MESTRE.for2
```

onde o sufixo `forn` indica que o programa deve ser executado nos processadores pertencentes à classe `n`.

Tempo máximo de execução do programa do hospedeiro

Como mencionado anteriormente, o computador-hospedeiro fica num laço à espera do aviso (dado pelo nó-mestre) de que a resposta do problema foi encontrada. Este laço é conseguido com o seguinte código (veja a listagem completa do programa HOSPEDEIRO no apêndice D):

```
GET_DONE = .false.
do while (.not.GET_DONE)
call acp_getblock(MAIEPS,MAIOR_EPSILON,SIZE,NO_MESTRE,CLASS_M,ACP$LAST_BLOCK,GET_DONE)
end do
```

A variável GET_DONE somente assumirá o valor .TRUE. quando o nó-mestre devolver ao hospedeiro o valor de MAIOR_EPSILON ou seja, somente quando o nó-mestre terminar seu programa. Entretanto, pode acontecer que o programa do nó-mestre demore muito tempo para terminar (isto vai depender, essencialmente, do tempo que os processadores de cálculo levarem para resolver o problema ou seja, do tamanho da matriz original e da precisão requerida).

Porém, um dos *defaults* do *software* básico do ACP determina que a execução da rotina acp_getblock tem que obter sua resposta num tempo máximo, sob pena de abortar o programa do hospedeiro, com a apresentação da mensagem de erro: %ACP-USER_ERROR, Host process timeout.

Desta forma, torna-se necessário modificar este tempo máximo de espera de um acp_getblock. Para isto, inclui-se no arquivo UPF a seguinte instrução:

```
host time = T
```

onde T é o tempo máximo (em segundos) de espera pela função acp_getblock, estimado pelo programador.

Tempo máximo de execução dos programas dos nós ACP

Uma vez que os programas dos nós (tanto os dos processadores-de-cálculo quanto o do nó-mestre) somente são desativados uma única vez (quando o hospedeiro recolhe os resultados), pode acontecer que estes nós fiquem executando seus programas por um tempo longo demais (maior do que 900 segundos).

Acontece, porém, que um dos *defaults* do *software* básico do ACP determina que qualquer nó ACP que executar um programa por mais de 900 segundos seja automaticamente desativado. Desta forma, torna-se necessário modificar este tempo máximo de execução de programas dos nós ACP. Para isto, inclui-se no arquivo UPF a seguinte instrução:

```
node time = T
```

onde T é o tempo máximo (em segundos) de execução do programa, estimado pelo programador. Se não se sabe o tempo que o programa do nó ACP gastará, pode-se colocar um valor muito elevado em T, digamos 9000.

Referências

[1] J.R.Biel, *Introduction to ACP Software for Event Oriented Processing*, curso dado no CBPF, april 4, 1988

[2] *ACP Software User's Guide for Event Oriented Processing*, Advanced Computer Program, FERMILAB, Revision 3.0, december 4, 1987

capítulo 9

Dados e Análises

O problema e sua solução

O problema que abordamos com os três programas desenvolvidos neste trabalho (veja os apêndices B, C e D), a resolução da equação de Laplace numa região bi-dimensional retangular, com condições de contorno, tem suas soluções analítica e numérica apresentadas a seguir.

Solução analítica

A fim de poder verificar a exatidão das soluções numéricas encontradas pelos programas de computador já mencionados, decidimos obter também a solução analítica da equação diferencial por eles resolvida.

Note que ao se resolver equações diferenciais mais complexas (do que $\nabla^2 u(x,y)=0$, por exemplo) através de métodos numéricos, nem sempre é possível se obter suas formas analíticas também (vale lembrar que esta foi uma das razões pelas quais trabalhamos com a equação de Laplace, como já mencionado no apêndice).

Desejamos resolver a equação diferencial parcial:

$$\frac{\partial^2 v(x,y)}{\partial x^2} + \frac{\partial^2 v(x,y)}{\partial y^2} = 0 \qquad \text{[eq.1]}$$

numa região bi-dimensional de dimensões LxC, com as seguintes condições de contorno:

$$v(x,0) = 0 \qquad \text{[eq.2]}$$
$$v(x,L) = 0 \qquad \text{[eq.3]}$$

$$v(0,y) = 0 \qquad \text{[eq.4]}$$
$$v(C,y) = T \qquad \text{[eq.5]}$$

Usando o método de separação de variáveis, supomos $v(x,y)$ da forma:

$$v(x,y) = X(x).Y(y)$$

obtemos as duas equações:

$$\frac{1}{X}.\frac{d^2X}{dx^2} = \lambda \qquad \text{[eq.6]}$$

$$\frac{1}{Y}.\frac{d^2Y}{dy^2} = -\lambda \qquad \text{[eq.7]}$$

que somente satisfarão as condições de contorno se $\lambda<0$.

Assim, substituindo as equações [eq.2] a [eq.5] nas equações [eq.6] e [eq.7], e usando a ortonormalidade da função seno no intervalo $(0,L)$, obtemos:

$$v(x,y) = \sum_{-\infty}^{\infty} \frac{4T}{(2n-1)\pi}.\frac{\operatorname{senh}\left((2n-1)\pi.\frac{x}{L}\right).\operatorname{sen}\left((2n-1)\pi.\frac{y}{L}\right)}{\operatorname{senh}\left((2n-1)\pi.\frac{C}{L}\right)}.$$

A fim de que a solução acima esteja de acordo com as condições de contorno (veja figura 1 adiante) usadas pelos programas mostrados nos apêndices B, C e D, mudamos a escala (no caso, de temperatura) de acordo com as expressões:

$$T = -100 \quad e$$

$$u(x,y) = v(x,y) + 100$$

Desta forma obtemos finalmente:

$$u(x,y) = 100 - \frac{400}{\pi}\sum_{-\infty}^{\infty} \frac{1}{(2n-1)}.\frac{\operatorname{senh}\left((2n-1)\pi.\frac{x}{L}\right).\operatorname{sen}\left((2n-1)\pi.\frac{y}{L}\right)}{\operatorname{senh}\left((2n-1)\pi.\frac{C}{L}\right)}. \qquad \text{[eq.8]}$$

que é a solução analítica da equação [eq.1] com as condições de contorno:

$$u(x,0) = 100 \ (°C) \qquad \text{[eq.9]}$$
$$u(x,L) = 100 \ (°C) \qquad \text{[eq.10]}$$
$$u(0,y) = 100 \ (°C) \qquad \text{[eq.11]}$$
$$u(C,y) = 0 \ (°C) \qquad \text{[eq.12]}$$

Solução numérica

No item anterior mostramos a resolução analítica [eq.8] da equação de Laplace [eq.1] num domínio retangular de dimensões LxC e com condições de contorno dadas pelas equações [eq.9] a [eq.12].

Na figura 1 abaixo está apresentada a resposta numérica encontrada pelos programas de computador (descritos nos capítulos "O Programa CALOR (Hospedeiro-Nós)" e "O Programa CALOR (Nós-Nós)", e mostrados nos apêndices B, C e D).

Nela, a altura (ao longo do eixo "temperatura") representa a temperatura (em °C, por exemplo) em cada um dos pontos da região bi-dimensional LxC delimitada pela base da figura.

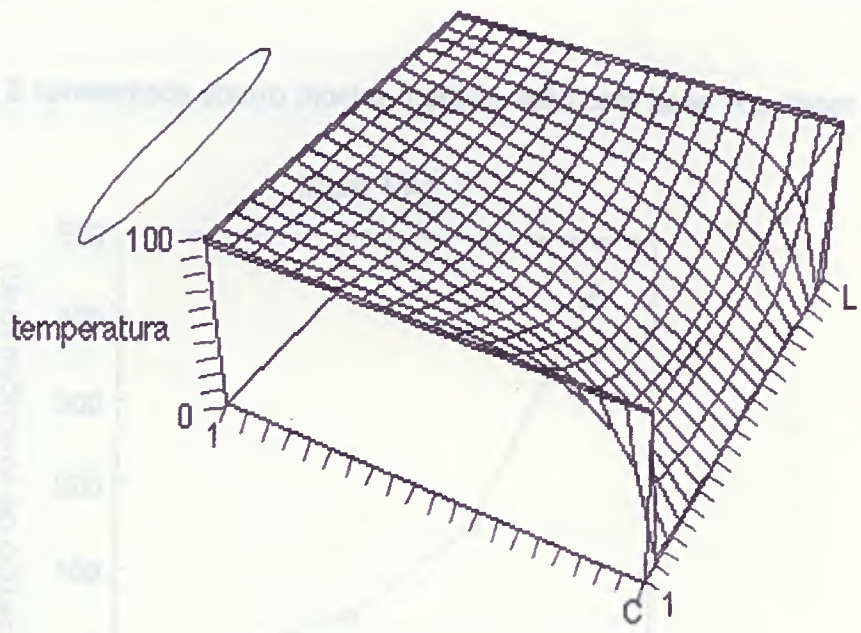


figura 1: condições de contorno e solução da equação de Laplace

Resolução num uniprocessador

A seguir temos, mostrados na tabela 1 e na figura 2, os dados referentes aos tempos de diversas execuções do programa "CALOR_SEM_ACP", cuja listagem encontra-se no apêndice B. Este programa resolve a equação de Laplace, usando o método descrito no capítulo "Equações Diferenciais Parciais", num computador com apenas 1 processador. O computador usado foi o MicroVAX II do Instituto de Física Teórica, rodando o sistema operacional VMS versão 4.6 e VAX FORTRAN versão 4.0. Para cada execução do programa, onde foram fixados o número de linhas da matriz original (igual a 400) a ser resolvida e a precisão requerida (igual a 1.0), variou-se o número de colunas do domínio original.

Na tabela 1 mostrada abaixo, para cada número de colunas usadas no domínio original (coluna 1), são fornecidos os tempos de execução do programa em segundos (coluna 2) e em minutos (coluna 3). Os valores com o menor tempo de processamento são sucedidos pelo símbolo (<) e os valores com o maior tempo de processamento são sucedidos pelo símbolo (>).

Número de Linhas 400		Precisão 1.0
número de colunas	tempo (s)	tempo (min)
100	246 (<)	4.1 (<)
200	2928	48.8
300	9425	157.1
400	26600 (>)	443.3 (>)

tabela 1: tempo de processamento (em segundos e minutos), usando o MicroVAX II

A figura 2 apresentada abaixo mostra o gráfico feito com base nos dados contidos na tabela 1.

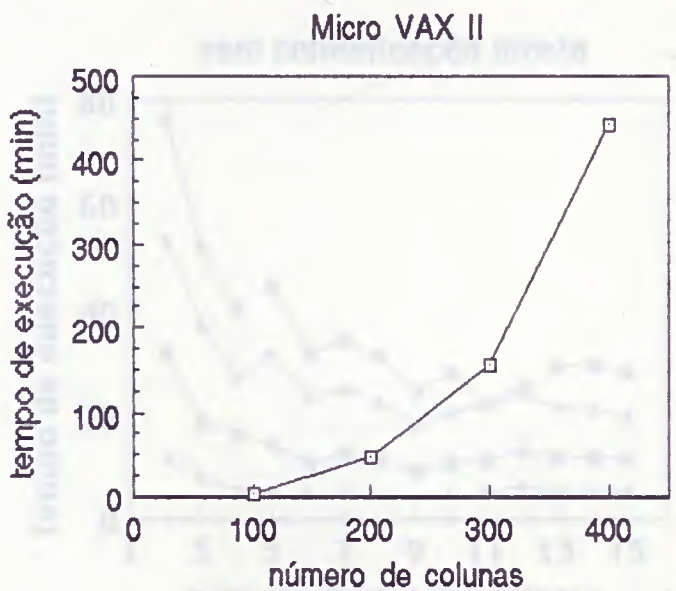


figura 2: tempo de processamento em função do número de colunas do domínio original (MicroVAX II)

Resolução sem comunicação entre os processadores ACP

A seguir temos, mostrados na tabelas 2 e na figura 3, os dados referentes aos tempos de diversas execuções do programa "CALOR_H_N", descrito no capítulo "O Programa CALOR (Hospedeiro-Nós)" e cuja listagem encontra-se no apêndice C. Para cada execução do programa, onde foram fixados os números de linhas (igual a 400) e de colunas da matriz original a ser resolvida, variou-se o número de processadores usados concorrentemente na resolução do problema. A precisão requerida foi a mesma em todas as execuções (igual a 1.0). Na tabela 2 mostrada abaixo, para cada número de processadores usados na resolução do problema (coluna 1) são fornecidos os tempos de execução (em segundos, nas colunas 2 a 5) do programa para cada um dos valores do número de colunas. A figura 3 mais adiante mostra o gráfico feito com base na tabela 2.

Número de processadores	Número de Colunas =100 tempo (s)	Número de Colunas = 200 tempo (s)	Número de Colunas = 300 tempo (s)	Número de Colunas = 400 tempo (s)
2	668 (>)	1894 (>)	3166 (>)	4577 (>)
3	453	1110	2238	3045
4	351	940	1616	2424
5	305	862	1901	2671
6	270	639	1392	1886
7	296	761	1484	2061
8	284	684	1340	1862
9	237 (<)	558 (<)	1073 (<)	1472
10	271	643	1220	1680
11	320	682	1318	1306 (<)
12	360	764	1437	1504
13	298	694	1297	1729
14	307	712	1249	1764
15	319	697	1175	1691

tabela 2: tempo de processamento (em segundos) sem comunicação direta entre os nós ACP

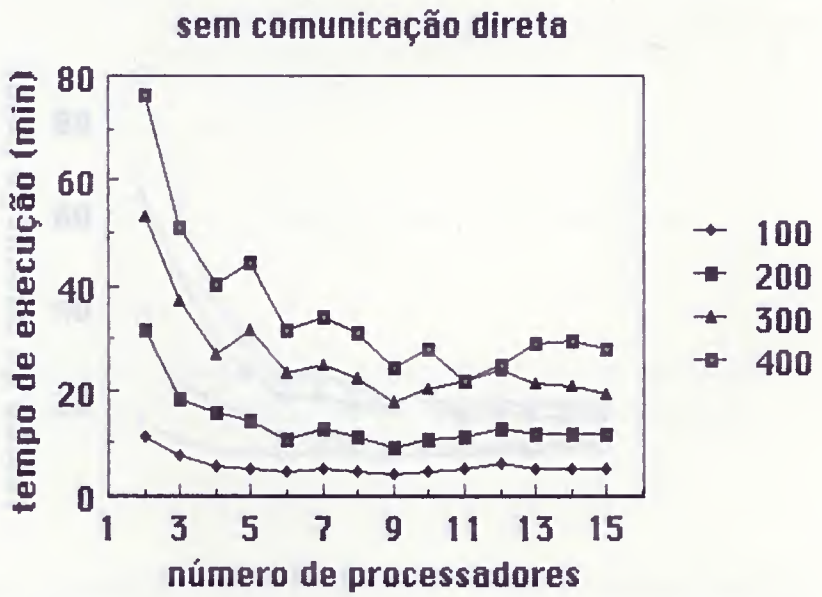


figura 3: tempo de processamento (em minutos) sem comunicação direta entre os nós ACP

Resolução com comunicação direta entre os processadores ACP

A seguir temos, mostrados na tabela 3 e na figura 4, os dados referentes aos tempos de diversas execuções do programa "HOSPEDEIRO", descrito no capítulo "O Programa CALOR (Nós-Nós)" e cuja listagem encontra-se no apêndice D. Para cada execução do programa, onde foram fixados os números de linhas (igual a 400) e de colunas da matriz original a ser resolvida variou-se o número de processadores usados concorrentemente na resolução do problema. A precisão requerida foi a mesma em todas as execuções (igual a 1.0). Na tabela 3 mostrada abaixo, para cada número de processadores usados na resolução do problema (coluna 1), são fornecidos os tempos de execução (em segundos, nas colunas 2 a 5) do programa para cada um dos valores do número de colunas. A figura 4 mais adiante mostra o gráfico feito com base na tabela 3.

Número de processadores	Número de Colunas =100 tempo (s)	Número de Colunas = 200 tempo (s)	Número de Colunas = 300 tempo (s)	Número de Colunas = 400 tempo (s)
2	932 (>)	2251 (>)	3841 (>)	5360 (>)
3	696	1392	2902	3773
4	570	1231	2035	2592
5	512	981	1593	2008
6	480	726	1233	1519
7	524	746	1256	1511
8	514	694	1151	1384
9	463 (<)	590	1090	1200
10	513	668	1344	1260
11	486	573 (<)	958 (<)	1136 (<)
12	542	632	1019	1210
13	531	633	987	1138
14	575	679	977	1167
15	593	644	989	1137

tabela 3: tempo de processamento (em segundos) com comunicação direta entre os nós ACP

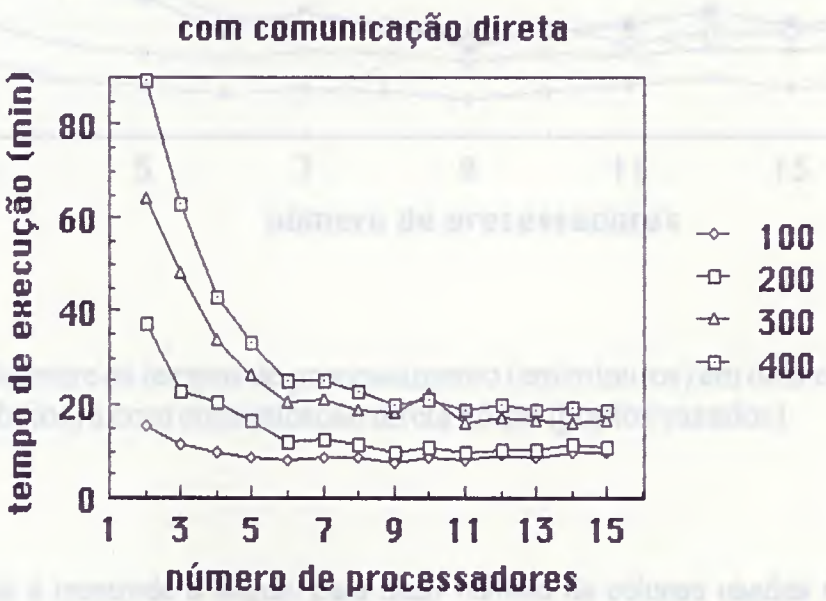


figura 4: tempo de processamento (em minutos) com comunicação direta entre os nós ACP

Comparação entre os diversos resultados

A seguir, na figura 5, temos o gráfico com todos os dados referentes aos tempos de execução (em minutos) num sistema multiprocessador sem comunicação direta entre os processadores e num sistema multiprocessador com comunicação direta entre os processadores (dados extraídos das tabelas 2 e 3).

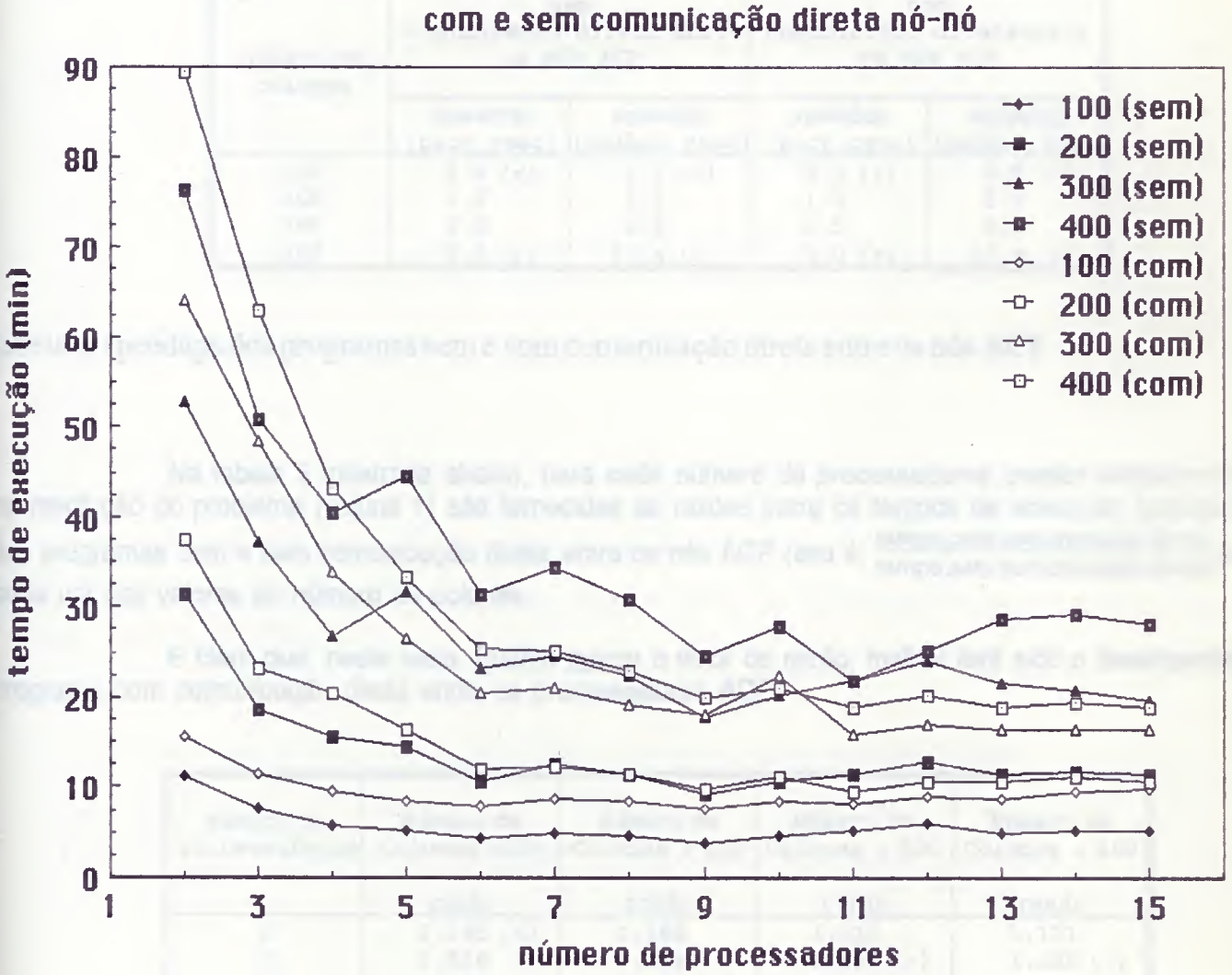


figura 5: comparação entre os tempos de processamento (em minutos) em dois casos: sem comunicação direta nó-nó (pontos cheios) e com comunicação direta nó-nó (pontos vazados)

Na tabela 4 mostrada a seguir, para cada número de colunas usadas no domínio original (coluna1) são fornecidas as razões entre os tempos de processamento no MicroVAX II e os tempos de

processamento em paralelo (ou seja, $\frac{\text{tempo.no.Micro.VAX.II}}{\text{tempo.em.paralelo}}$) para os piores (colunas 2 e 4) e melhores (colunas 3 e 5) casos das tabelas 2 e 3. Como antes, as entradas com o menor valor da razão são sucedidas pelo símbolo (<) e as entradas com o maior valor da razão são sucedidas pelo símbolo (>).

É claro que quanto maior o valor desta razão, melhor terá sido o desempenho dos programas paralelos. Na verdade, estes valores representam os *speedups* dos programas (veja o capítulo "Desenvolvendo Algoritmos Paralelos").

número de colunas	sem comunicação direta entre os nós ACP		com comunicação diretaentre os nós ACP	
	speedup (pior caso)	speedup (melhor caso)	speedup (pior caso)	speedup (melhor caso)
100	0.4 (<)	1.0 (<)	0.3 (<)	0.5 (<)
200	1.6	5.3	1.3	5.1
300	3.0	8.8	2.5	9.8
400	5.8 (>)	20.4 (>)	5.0 (>)	23.4 (>)

tabela 4: speedups dos programas sem e com comunicação direta entre os nós ACP

Na tabela 5 mostrada abaixo, para cada número de processadores usados concorrentemente na resolução do problema (coluna 1) são fornecidas as razões entre os tempos de execução (colunas 2 a 5) dos programas com e sem comunicação direta entre os nós ACP (isto é, $\frac{\text{tempo.com.comunicação.direta}}{\text{tempo.sem.comunicação.direta}}$) para cada um dos valores do número de colunas.

É claro que, neste caso, quanto menor o valor da razão, melhor terá sido o desempenho do programa com comunicação direta entre os processadores ACP.

Número de processadores	Número de Colunas =100 razão	Número de Colunas = 200 razão	Número de Colunas = 300 razão	Número de Colunas = 400 razão
2	1.395 (<)	1.188	1.213	1.171
3	1.536	1.254	1.297 (>)	1.239 (>)
4	1.624	1.310 (>)	1.259	1.069
5	1.679	1.138	0.838	0.752
6	1.778	1.136	0.886	0.805
7	1.770	0.980	0.846	0.733
8	1.810	1.015	0.859	0.743
9	1.954 (>)	1.057	1.016	0.815
10	1.893	1.039	1.102	0.750
11	1.519	0.840	0.727	0.870
12	1.506	0.827 (<)	0.709 (<)	0.805
13	1.782	0.912	0.761	0.658 (<)
14	1.873	0.954	0.782	0.662
15	1.859	0.924	0.842	0.672

tabela 5: razões entre os tempos de processamento com e sem comunicação direta nó-nó

Nas tabela 6 e 7 mostadas a seguir temos as razões entre os tempos de processamento para vários números de processadores e o menor tempo de processamento (ou seja, $\frac{\text{tempo}}{\text{menor.tempo}}$), para um dado número de colunas, nos casos sem e com comunicação direta, respectivamente.

Aqui, quando os resultados das razões (para alguns números de processadores) estiverem perto uns dos outros, o problema poderá ser resolvido usando-se os números de processadores associados, sem perdas significativas nos desempenhos. Por exemplo, no caso de comunicação direta com matrizes 400x300 (tabela 7, coluna 5), poderiam ser usados 11, 13 ou 15 nós ACP, indistintamente, que o tempo de execução seria basicamente o mesmo.

Número de processadores	Número de Colunas =100 tempo/menor tempo	Número de Colunas = 200 tempo/menor tempo	Número de Colunas = 300 tempo/menor tempo	Número de Colunas = 400 tempo/menor tempo
2	2.819 (>)	3.394 (>)	2.951 (>)	3.505 (>)
3	1.911	1.989	2.086	2.332
4	1.481	1.685	1.506	1.856
5	1.287	1.545	1.772	2.045
6	1.139	1.145	1.297	1.444
7	1.249	1.364	1.383	1.578
8	1.198	1.226	1.249	1.426
9	1.000 (<)	1.000 (<)	1.000 (<)	1.127
10	1.143	1.152	1.137	1.286
11	1.350	1.222	1.228	1.000 (<)
12	1.519	1.369	1.339	1.152
13	1.257	1.244	1.209	1.324
14	1.295	1.276	1.164	1.351
15	1.346	1.249	1.095	1.295

tabela 6: razões entre os vários tempos de processamento e o menor tempo de processamento, sem comunicação direta nó-nó

Número de processadores	Número de Colunas =100 tempo/menor tempo	Número de Colunas = 200 tempo/menor tempo	Número de Colunas = 300 tempo/menor tempo	Número de Colunas = 400 tempo/menor tempo
2	2.013 (>)	3.928 (>)	4.009 (>)	4.718 (>)
3	1.503	2.429	3.029	3.321
4	1.231	2.148	2.124	2.282
5	1.106	1.712	1.663	1.768
6	1.037	1.267	1.287	1.337
7	1.132	1.302	1.311	1.330
8	1.110	1.211	1.201	1.218
9	1.000 (<)	1.030	1.138	1.056
10	1.108	1.166	1.403	1.109
11	1.050	1.000 (<)	1.000 (<)	1.000 (<)
12	1.171	1.103	1.064	1.065
13	1.147	1.105	1.030	1.002
14	1.242	1.185	1.020	1.027
15	1.281	1.124	1.032	1.001

tabela 7: razões entre os vários tempos de processamento e o menor tempo de processamento, com comunicação direta nó-nó

Comparação entre os speedups

Observando a tabela 4 podemos notar alguns fatos interessantes com relação aos tempos de processamento num uniprocessador e nos sistemas paralelos (ou seja, *speedups* com e sem comunicação direta entre os nós ACP):

a) nos casos em que o número de colunas é igual a 100:
ou seja, usando-se matrizes relativamente pequenas. Percebe-se que para matrizes pequenas quase não há vantagem em se usar a paralelização (sem comunicação direta: *speedup* igual a 1.0, no melhor caso), pois há muita comunicação de dados com relação aos cálculos propriamente ditos. Isto fica ainda mais crítico quando se tenta usar a comunicação direta, pois o desempenho do programa cai ainda mais, chegando a levar o dobro do tempo que o programa num uniprocessador (com comunicação direta: *speedup* de somente 0.5, no melhor caso).

b) nos casos em que o número de colunas é igual a 200:
aumentando-se um pouco o tamanho da matriz nota-se a superioridade do algoritmo paralelo na resolução deste tipo de equação diferencial: o uniprocessador demora 5 vezes mais para resolver o mesmo problema. Porém, a comunicação direta entre os processadores ACP ainda não se mostra vantajosa, pois com ela consegue-se um *speedup* de 5.1 (no melhor caso) enquanto que o programa que não a implementa tem um *speedup* um pouco maior (igual a 5.3). Apesar destes números estarem próximos um do outro, é possível notar a tendência de aumento do *speedup* do programa com comunicação direta em relação ao *speedup* do programa que não a implementa

c) nos casos em que o número de colunas é igual a 300:
aqui permanece a tendência (já esperada) de altos *speedups*. O interessante é que o programa com comunicação direta já é mais vantajoso do que o programa sem ela (no melhor caso), mas esta vantagem não é muito grande (aproximadamente 11%)

d) nos casos em que o número de colunas é igual a 400:
com grandes matrizes (neste caso, de dimensão igual a 400 x 400) acentua-se ainda mais a vantagem do algoritmo que implementa a resolução em paralelo. Além disto, a comunicação direta se mostra mais vantajosa do que a resolução sem ela (chega-se a *speedup* de 23.4 com comunicação direta, contra 20.4 sem ela).

Assim, concluímos que:

- para matrizes pequenas (menores do que 100 kbytes) não há vantagem em se resolver o problema num processador paralelo: é preferível usar um uniprocessador. Somente para matrizes maiores devem ser usados os programas paralelos

- para matrizes de tamanho médio (entre 100 e 1000 kbytes) e grandes (maiores do que 1 Mbyte ou seja, provavelmente todas as vezes em que se for resolver um problema real) deve-se usar as versões paralelizadas dos programas, que utilizam o ACP

Comparação entre os desempenhos dos programas paralelos

Observando a figura 5 e a tabela 5 é possível se fazer algumas observações com relação aos desempenhos dos programas com e sem comunicação direta:

a) nos casos em que o número de colunas é igual a 100:
ou seja, usando-se matrizes (que representam o domínio de discretização) relativamente pequenas. Aqui, o fato de se usar a comunicação direta piorou o desempenho total do programa. Nota-se que o programa com comunicação direta é sempre mais lento do que o programa controlado somente pelo computador-hospedeiro

b) nos casos em que o número de colunas é igual a 200:
aumentando um pouco o tamanho das matrizes (agora de dimensão 400 x 200), nota-se que o programa com comunicação direta ainda teve seu desempenho pior do que o programa que não a implementa, porém a diferença entre estes desempenhos diminui à medida em que aumenta o número de processadores. Aqui se começa a notar uma tendência de melhora do desempenho do programa que usa comunicação direta com o aumento da complexidade do problema (ou seja, com o aumento do tamanho das matrizes)

c) nos casos em que o número de colunas é igual a 300:
aqui comprovamos a tendência mencionada no item anterior. Com pouca paralelização (isto é, com até 4 processadores trabalhando em paralelo) a comunicação direta entre os nós não se mostra vantajosa. Porém, com uma maior paralelização observa-se nitidamente a vantagem da comunicação direta: o tempo de processamento desta forma é (geralmente) menor do que o tempo de processamento do programa sem comunicação direta (e quando isto não acontece, para 9 e 10 nós ACP, a diferença não é grande: aproximadamente 5%)

d) nos casos em que o número de colunas é igual a 400:
aumentando ainda mais o tamanho das matrizes fica clara a vantagem de se usar a comunicação direta entre os processadores: com exceção dos casos com baixa paralelização (com até 4 processadores trabalhando concorrentemente) o tempo de processamento sem comunicação direta foi bem maior do que o tempo de processamento do programa com comunicação direta

Assim, concluímos que:

- para matrizes de tamanho médio (entre 100 e 1000 kbytes) deve-se usar o ACP, porém sem comunicação direta entre os processadores
- para matrizes grandes (maiores do que 1 Mbyte) deve-se usar o ACP com comunicação direta, sendo que o número mínimo de nós alocados deve estar em torno de 12

Número de processadores

Ainda com base na figura 5, nota-se uma tendência de estabilização do tempo de processamento com o aumento do número de processadores, sendo que esta estabilização é mais suave para o programa que implementa a comunicação entre os nós ACP.

É claro que está tendência somente aparece devido ao número de processadores usados ser pequeno: certamente haverá um aumento no tempo de processamento se forem usados muitos processadores, pois haverá um momento em que a maior parte deles estará ociosa e muito tempo será gasto na comunicação dos dados. Por outro lado podemos perceber que deve haver um número ótimo de processadores com o qual é feita a menor comunicação de dados com relação ao tempo em que cada um deles permanece ocupado.

Com base na tabelas 6 e 7 podemos afirmar que, tomando por referência este número-ótimo-de-processadores:

- se for diminuído o número de processadores então os nós ficarão sobrecarregados (deixando o barramento ocioso) e demorarão mais tempo para resolver o problema
- se for aumentado o número de processadores, então os nós ficarão desocupados muito cedo (provavelmente enquanto um nó anterior estiver transmitindo seus dados de volta ao mestre, seja ele o computador-hospedeiro, seja um nó-mestre ACP) e o barramento ficará sobrecarregado

Parece estar claro que este número-ótimo-de-processadores deve depender do tamanho das matrizes envolvidas nos cálculo: para matrizes pequenas este número deve ser atingido com menos processadores do que para o caso de matrizes grandes. Isto acontece, de fato. Se observarmos a figura 5 (ou as tabelas 2 e 3), notaremos que para determinados números de processadores há mínimos nas curvas que representam o tempo de processamento, confirmando a previsão de que estes mínimos devem aparecer para números de processadores cada vez maiores quando as dimensões das matrizes forem aumentadas.

Assim, concluímos que:

- quanto maiores forem as matrizes, maior deverá ser o número de processadores a serem usados concorrentemente na resolução do problema
- para matrizes pequenas (menores do que 100 kbytes) deve-se usar, no máximo, 9 nós ACP
- para matrizes de tamanho médio (entre 100 e 1000 kbytes) e grandes (maiores do que 1 Mbyte) deve-se usar, no mínimo, 12 processadores ACP
- uma vez atingido um determinado número-ótimo-de-processadores, não será vantajoso se aumentar ainda mais o número de processadores

capítulo 10

Conclusões

Conclusões

Com este trabalho começamos nossos estudos sobre Processamento Paralelo aplicado à Física, e com ele foi possível tirar algumas conclusões:

a) é importante o uso de multiprocessamento na resolução de equações diferenciais parciais (no caso específico, usamos a equação de Laplace), pois a paralelização do algoritmo que as resolve pode ser obtida de forma a usar eficientemente os diversos processadores (deve ser ressaltado que o método usado na resolução da equação de Laplace pode servir de base para algoritmos de resolução de outros problemas tais como sistemas lineares, entre outros) ou seja, há um paralelismo intrínseco neste tipo de problema

b) é vantajoso se desenvolver algoritmos paralelos e programas para multiprocessadores (de aplicação em Física), pois consegue-se melhoras nos desempenhos altamente compensadoras (conseguimos obter uma diminuição no tempo de processamento com relação ao Micro VAX II de até 23 vezes)

c) quanto à paralelização do algoritmo para a resolução da equação diferencial parcial propriamente dito, podemos afirmar que:

- para matrizes (que representam o domínio de discretização) menores do que 100 kbytes não há vantagem em se usar o ACP, sendo recomendável somente o uso do Micro VAX II
- para matrizes de tamanho entre 100 e 1000 kbytes deve-se usar o ACP na resolução do problema, porém sem comunicação direta entre os nós
- para matrizes maiores do que 1 Mbyte também se deve usar o ACP, porém com comunicação direta entre os processadores

d) quanto ao sistema ACP, podemos notar que há uma melhora dos desempenhos dos programas quando se usa a comunicação direta entre os nós, e isto acontece devido ao fato de que com a implementação da comunicação direta (ou seja, quando o programador trabalha num nível mais baixo) as transferências de dados são mais eficientes

apêndice A

Programando o MC68020

Programando o MC68020

Introdução

O objetivo deste apêndice é mostrar como olhar para um programa em **linguagem *assembly*** e entender o que está acontecendo. Ele não vai ensinar a programar em linguagem *assembly*.

Neste apêndice será introduzido o microprocessador MC68020 (usado no computador paralelo ACP) do ponto de vista de um programador em linguagem de alto nível (como FORTRAN ou Pascal), que certamente irá precisar depurar códigos compilados do ACP, mais cedo ou mais tarde. Não será apresentada a parte física do microprocessador .

Não se exige nenhum conhecimento anterior de programação em linguagem *assembly*. Supomos apenas que se saiba programar numa linguagem de alto nível assim como os princípios básicos de numeração nos sistemas binário e hexadecimal (representados aqui por números precedidos pelo cifrão: $).

Para melhor entender a diferença entre conseguir ler um programa em linguagem *assembly* e saber escrever um programa nesta linguagem, imagine que você está tentando ler (e entender!) uma sentença escrita numa língua que você não conhece (alemão, por exemplo). Se conhecer algumas palavras da sentença, você poderá entender o significado dela, mesmo sem saber conjugar os verbos, declinar corretamente os adjetivos ou conhecer outros pontos gramaticais importantes. Por exemplo, mesmo sem saber alemão você talvez consiga entender o que significa "*Interview mit zwei japanischen Studenten* ", mesmo que não consiga formulá-la por si próprio.

Programando um microprocessador

O microprocessador MC68020 (como todos os outros microprocessadores que existem) somente entende linguagem *assembly* (na verdade, somente **linguagem de máquina**, mas uma está tão próxima da outra que usaremos o termo linguagem *assembly* para designar as duas indistintamente, a menos que a diferenciação se faça necessária).

Toda comunicação com o MC68020 tem que ser feita somente nesta linguagem. Qualquer construção de alto nível (`if`, `while`, `for` e todas as outras) somente pode ser executada neste código nativo do microprocessador.

Os programas **compiladores** transformam as construções em linguagem de alto nível neste código nativo dos microprocessadores. Daí a razão de os compiladores serem específicos para cada processador (ou para cada família de processadores).

As instruções

A capacidade mais fundamental de um microprocessador é a sua habilidade em aceitar e executar uma **instrução** . Estas instruções são armazenadas na memória do computador, seja em RAM (*Random Access Memory*) ou em ROM (*Read Only Memory*).

Cada instrução do MC68020 (geralmente) inclui **operandos** , que são os valores usados pelo microprocessador para executar sua tarefa. Por exemplo, o MC68020 tem uma instrução que soma dois números: estes dois números são os operandos da instrução de soma. Por outro lado, ele também tem uma instrução que não faz absolutamente nada, que serve para se gastar tempo, seja lá pela razão que for (e existem muitas delas): esta instrução não tem nenhum operando.

As instruções que o MC68020 entende executam várias operações. Por exemplo, um conjunto delas é usado para mover dados na memória, outro conjunto executa operações matemáticas, outro conjunto controla o fluxo do programa etc. Todos este conjuntos serão abordados mais adiante.

Uma vez que as instruções são armazenadas na memória, elas têm representações numéricas. Por exemplo, a instrução que move dados de um local para outro é codificada pelo número $11F8.

Todas as instruções do MC68020 são compostas de 2 *bytes*, como a mostrada acima, mas algumas precisam, como dito anteriormente, de *bytes* adicionais para especificar os operandos. Se usarmos a instrução $11F8 para mover um dado (*byte*) de um lugar para outro, precisaremos especificar os endereços-fonte e endereços-destino deste *byte* a ser movido logo após a instrução propriamente dita.

A linguagem *assembly* atribui nomes às diferentes instruções. Por exemplo, as instruções que movem dados recebem o nome de `MOVE`. Estes nomes são chamados de **mnemônicos** porque eles são mais facilmente lembráveis (pelos humanos) do que coisas como $11F8.

Os mnemônicos não têm nenhum significado para o microprocessador. Eles são sempre traduzidos para suas representações numéricas ($11F8, no caso do mnemônico MOVE).

Os registradores

Além da memória do computador (RAM e ROM), o MC68020 utiliza uma pequena (mas importantíssima) memória própria, chamada de **registradores** .

Da mesma forma que com a RAM ou ROM, o microprocessador pode escrever nos registradores (e ler deles). Esta memória consiste de 16 registradores, de 4 *bytes* cada um. Eles estão divididos em 2 grupos de 8 registradores cada: os **registradores de dados** (representados por `D0` até `D7`) e os **registradores de endereço** (representados por `A0` até `A7`).

O microprocessador tem instruções específicas para lidar com os registradores. Por exemplo, a instrução `MOVE` também pode ser usada para mover dados dos (ou para os) registradores. A vantagem de se usar registradores é que eles são mais rápidos do que a RAM.

Dos 16 registradores disponíveis no MC68020, 1 tem significado especial: o registrador de endereços A7. Ele controla a **pilha**, que é uma estrutura de dados do tipo último-que-entra-é-o-primeiro-que-sai (LIFO, *last in first out*) usada para armazenamento de variáveis locais e globais, passagem de parâmetros, variáveis temporárias de um programa (criadas quando se chama uma sub-rotina, por exemplo), endereços de retorno, valores salvos de outras variáveis (para posterior re-utilização) etc. O registrador A7, por apontar para esta pilha, recebe o nome especial de SP (*Stack Pointer*, ou ponteiro da pilha). O SP sempre aponta para o local onde o próximo dado a ser acrescentado irá entrar.

Um fato que merece ser destacado é que a pilha cresce em direção aos endereços mais baixos da memória. Isto significa que o chamado **topo da pilha** tem, na verdade, o menor endereço de toda a pilha.

Além destes 16 registradores (de dados e de endereços), o microprocessador tem ainda 2 outros registradores importantes: **contador do programa** e **registrador de estado**.

O contador do programa (abreviado por PC, de *Program Counter*) contém o endereço da próxima instrução a ser executada. Ele é automaticamente incrementado da quantia adequada para que seja executada a próxima instrução.

O registrador de estado (*Status Register*) mantém várias informações importantes sobre o microprocessador. A parte mais importante do registrador de estado é o *byte* menos significativo, que contém informações (nos chamados ***flags***, de 1 bit de comprimento) sobre o resultado da última instrução que foi executada. Um dos bits deste *byte*, chamado bit Z, é colocado em 1 sempre que a última instrução executada tenha gerado um zero (por exemplo, se o número zero foi transferido, com MOVE; ou se uma soma resultou no valor zero; etc). Existem 5 bits como ele, chamados de códigos de condição (*condition codes*) e este *byte* do registrador de estado é conhecido como registrador de códigos de condição (CCR, *Condition Code Register*). Os códigos de condição são os seguintes:

- C (*Carry Bit*) : contém o *carry* ("vai-um") do bit mais significativo produzido pela última instrução
- X (*Extend Bit*) : é sempre igual ao bit C
- Z (*Zero Bit*) : tem valor 1 sempre que a operação executada gera um 0, e tem o valor 0 caso contrário
- N (*Negative Bit*) : tem valor 1 sempre que um resultado é negativo e 0 caso contrário
- V (*Overflow Bit*) : tem valor 1 sempre que o resultado de uma operação tem magnitude maior do que pode ser representada pelo operando-destino

O MC68020 contém um conjunto de instruções que realizam várias ações dependendo dos códigos de condição e que servem para que o programa possa tomar rumos diferentes dependendo dos resultados de outras operações (é o análogo à construção if das linguagens de alto nível). Por exemplo, há uma instrução que muda o fluxo do programa para uma parte diferente do programa se o *flag* Z contiver o valor 1.

Veremos, no item "Conjunto de instruções do MC68020" mais adiante, as principais instruções deste microprocessador.

Modos de endereçamento

Como dito anteriormente, a maioria das instruções do MC68020 requer operandos, que são os análogos dos parâmetros nas linguagens de alto nível. Além de especificar seus operandos, a maioria das instruções permite que especifiquemos o tamanho deles, geralmente *bytes*, **palavras** (*words*, compostas de 2 *bytes*) ou **palavras-longas** (*longwords* , compostas de 4 *bytes*).

Em partes diferentes de um programa escrito em *assembly* os operandos serão encontrados de maneiras diferentes. Por exemplo, se desejarmos somar `2103` ao valor armazenado no registrador `D3`, haverá 2 operandos: o valor `2103` e o registrador `D3`. Cada um destes operandos será representado em linguagem *assembly* por meio de um **modo de endereçamento** particular, onde por modo de endereçamento de uma instrução entendemos o método usado por ela para calcular o endereço de seus operandos.

No exemplo acima (somar `2103` ao registrador `D3`) escrevemos a instrução correspondente, em *assembly*, como sendo `ADD.W #2103, D3` . O nome da instrução é `ADD`. O sufixo "`.W`" após o nome da instrução indica que o tamanho do operando, neste caso `2103`, é de 1 palavra (*word*). Se fosse o caso poderíamos ter usado o sufixo "`.B`" (*byte*) ou o sufixo "`.L`" (*longword*). Uma instrução sem nenhum sufixo é entendida como trabalhando com operandos de tamanho de 1 palavra (2 *bytes*). O símbolo "`#`" precedendo o número `2103` indica que este operando está especificado no modo de endereçamento **imediato**, o que equivale a dizer que o valor especificado na instrução (o `2103`) é o valor real a ser usado: ele não é uma posição de memória ou a especificação de um registrador, mas sim o valor em si.O segundo operando desta instrução é um registrador (no caso `D3`). Este modo de endereçamento, contendo simplesmente o nome do registrador, é chamada de modo **direto do registrador de dados**. Simplesmente significa que a instrução especifica um registrador a ser usado como operando. Por falar nisto, o primeiro operando (neste caso o número `#2103`) é chamado de **operando-fonte** e o segundo (neste caso o registrador `D3`) de **operando-destino**.

Estes 2 modos de endereçamento, imediato e direto de registrador de dados, são 2 dos 18 modos de endereçamento disponíveis no MC68020. Alguns deste modos são usados muito raramente, mas existem, caso se precise deles. Vale a pena notar que nem todas as instruções apresentam todos os modos de endereçamento, mas para a maioria da instruções há os modos de endereçamento mais desejáveis.

Uma observação interessante: se o sufixo "`.W`" for usado, a palavra é obtida do *byte* endereçado e do *byte* imediatamente posterior a ele; se o sufixo "`.L`" for usado, os *bytes* usados são o endereçado pela instrução e os 3 posteriores a ele.

A seguir exporemos os 18 modos de endereçamento disponíveis para o MC68020, classificados em 8 categorias:

- endereçamento direto de registradores
- endereçamento indireto, via registrador
- endereçamento indireto, via memória
- endereçamento indireto, via contador de programa (PC)
- endereçamento indireto, via posição de memória apontada pelo PC
- endereçamento absoluto
- endereçamento imediato
- endereçamento implícito.

Endereçamento direto de registradores

Neste modo de endereçamento, o operando é um dos 16 registradores do MC68020. No exemplo do item anterior, vimos um modo de endereçamento direto do registrador usando um registrador de dados (D0 a D7).

Há um outro modo de endereçamento direto de registrador, o modo **direto do registrador de endereços** . Ele funciona de maneira análoga ao modo direto do registrador de dados, exceto que um registrador de endereço (A0 a A7) é usado como operando. Um exemplo é a instrução `ADD A2, D4`, que soma o valor encontrado em A2 com o valor armazenado em D4, colocando o resultado em D4. Nesta instrução o primeiro operando está especificado pelo modo direto do registrador de endereços e o segundo operando pelo modo direto do registrador de dados.

Endereçamento indireto, via registrador

É muito freqüente acontecer que o programador queira calcular um endereço baseado num endereço que já esteja num registrador de endereços (isto é, de uma maneira indireta). O MC68020 possui cinco modos de endereçamento para se especificar endereços desta maneira.

O primeiro destes modos é chamado de **indireto, via registrador de endereços**. Nele o endereço efetivo é simplesmente o conteúdo do registrador de endereços especificado. Por exemplo, a instrução `MOVE.L $145,(A0)` causará a transferência da palavra-longa armazenada na posição de memória `$145` para a posição de memória cujo endereço está contido no registrador A0. Assim, se A0 contiver o número `$150489` antes de esta instrução ser executada, o conteúdo da posição de memória `$145` será copiado na posição de memória `$150489`.

Quando um programa manipula objetos na pilha (discutida anteriormente) o ponteiro da pilha (SP) tem que ser atualizado constantemente para refletir a posição atual do topo da pilha. Antes que um novo objeto seja colocado na pilha, o SP tem que ser decrementado para criar espaço para este novo objeto. Quando um objeto é removido da pilha o SP tem que ser incrementado de tantos *bytes* quanto for o tamanho deste objeto.

Uma vez que manipulação de pilha é muito usada, o MC68020 tem um modo de endereçamento que combina as duas operações de colocar um objeto na pilha e decrementar o SP, numa só instrução.

Consegue-se acrescentar um objeto à pilha com um modo de endereçamento chamado de **indireto, com pré-decremento, via registrador de endereços**. Quando um programa usa este modo, automaticamente o ponteiro da pilha é decrementado de forma a criar espaço para o objeto, que é colocado na pilha. A instrução `MOVE.W D0,-(A7)` fará duas coisas: primeiro, o ponteiro da pilha (registrador A7) será decrementado de 2 *bytes* (por causa do sufixo "`.W`", que significa 2 *bytes*) para criar espaço para o objeto a ser acrescentado à pilha; segundo, a palavra armazenada no registrador D0 será colocada na pilha. O sinal de menos (-) indica que se trata de pré-decremento . Este processo pode ser visto na figura 1 mostrada a seguir.

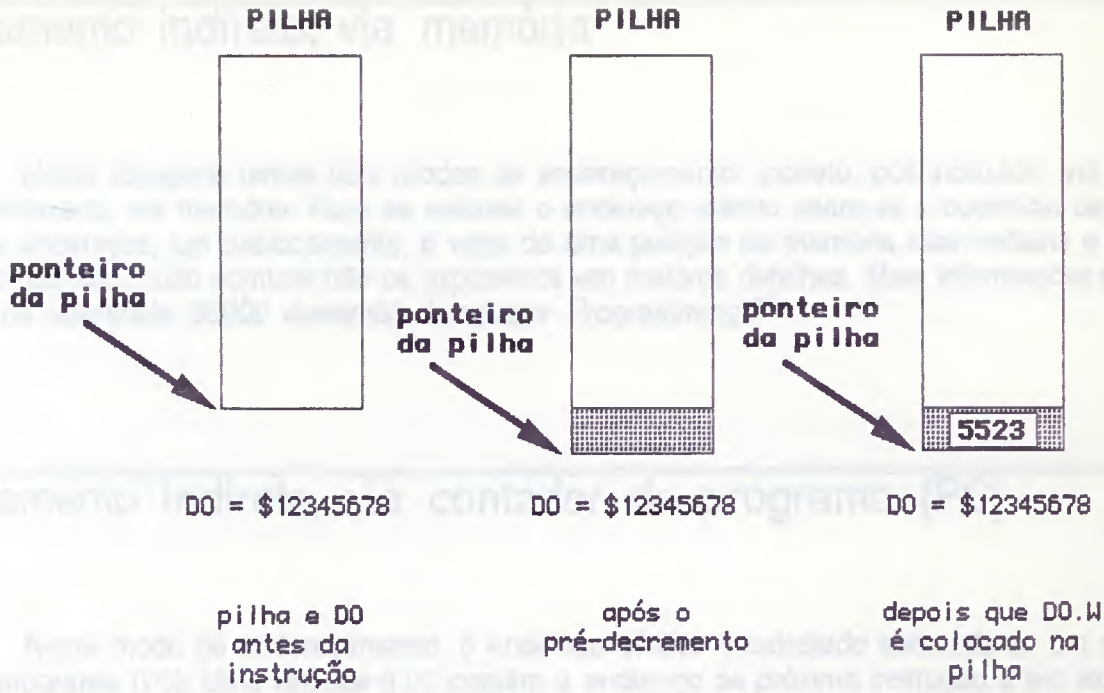


figura 1: modo de endereçamento indireto, com pré-decremento, via registrador de endereços

Observe que o ponteiro da pilha é automaticamente decrementado de 2 *bytes* pois este é o tamanho do operando. Se o operando fosse uma palavra-longa (denotada pelo sufixo ".L" junto à instrução) o ponteiro da pilha seria decrementado de 4 *bytes*. Porém, se o operando tiver o tamanho de 1 *byte* (sufixo ".B") o ponteiro da pilha será decrementado de 2 *bytes*. Isto garante que a pilha sempre comece em endereços pares.

O MC68020 tem um modo de endereçamento correspondente para se retirar objetos da pilha: **indireto, com pós-incremento, via registrador de endereços**. Neste modo a instrução é executada em primeiro lugar e só então é que o ponteiro da pilha é incrementado. Como exemplo vejamos a instrução `MOVE.L (A7)+,(A2)`. Ela move uma palavra-longa (".L"), situada no topo da pilha (apontada por `A7`) para o registrador `A2`. Após a transferência o ponteiro da pilha é incrementado de 4 *bytes* (".L"). O sinal de mais (+) indica que se trata de pós-incremento .

Os outros modos de endereçamento indireto com registrador são parecidos com os modos indiretos com relação ao PC, discutidos anteriormente. Eles são endereçamento **indireto, com deslocamento, via registrador de endereços** e endereçamento **indireto, com indexação e deslocamento, via registrador de endereços**. Estes modos permitem que se especifique um registrador de endereços e 1 ou 2 (respectivamente) valores a serem somados ao registrador de endereços para se obter o endereço efetivo.

Por exemplo, se tivermos a instrução `MOVE.W 8(A6),12(A1,D0.W)`, o endereço efetivo do operando-fonte (especificado pelo modo de endereçamento indireto, com deslocamento, via registrador de endereços) será calculado somando-se `8` ao valor armazenado no registrador `A6`. Já o endereço efetivo do operando-destino (especificado pelo modo de endereçamento indireto, com indexação e deslocamento, via registrador de endereço) será calculado somando-se `12`, o conteúdo de `A7` e o conteúdo de `D0`.

Endereçamento indireto, via memória

Nesta categoria temos dois modos de endereçamento: indireto, pós-indexado, via memória e indireto, pré-indexado, via memória. Para se calcular o endereço efetivo usam-se o conteúdo de um registrador de endereços, um deslocamento, o valor de uma posição de memória intermediária e um fator de escala. Como não são muito comuns não os exporemos em maiores detalhes. Mais informações podem ser encontrados na referência *68000 Assembly Language Programming* [1].

Endereçamento indireto, via contador de programa (PC)

Neste modo de endereçamento, o endereço efetivo é calculado somando-se um valor ao contador do programa (PC). Uma vez que o PC contém o endereço da próxima instrução a ser executada, o endereçamento relativo ao contador do programa sempre funciona, não importando em qual endereço da memória o programa foi carregado.

O modo mais comum de endereçamento relativo ao PC é o endereçamento **indireto, com deslocamento, via PC**. Neste modo a instrução contém um deslocamento que é somado ao PC gerando o endereço efetivo. Por exemplo, a instrução `MOVE.W 234(PC),D4` copiará a palavra que está a 234 *bytes* da instrução sendo executada para o registrador D4.

Um outro modo de endereçamento relativo ao PC é o endereçamento **indireto, com indexação e deslocamento, via PC**. Neste modo a instrução não somente fornece um deslocamento que é somado ao PC como também soma um outro valor, o índice, que pode ser qualquer registrador de dados ou de endereços. Em *assembly* ele tem a seguinte forma: `JSR 1989(PC,A4.W)`. O endereço efetivo é calculado somando-se o valor de PC ao deslocamento (neste caso 1989) e ao índice, que neste caso é a palavra armazenada no registrador A4. Este modo de endereçamento é usado, por exemplo, pela construção case do Pascal: o programa carrega o seletor do case num registrador (usando o exemplo acima, no registrador A4) e executa a instrução JMP (salto incondicional) com endereçamento indireto, com indexação e deslocamento, via PC.

Endereçamento indireto, via memória apontada pelo PC

Neste modo de endereçamento, como nos outros modos indiretos, o endereço efetivo é calculado usando-se uma posição de memória intermediária. Aqui, entretanto, ao invés de se usar um registrador de endereços, usa-se o valor atual do contador do programa (PC). Nesta categoria existem dois modos de endereçamento: indireto, pós-indexado, via posição de memória apontada pelo PC e indireto, pré-indexado, via posição de memória apontada pelo PC. Mais uma vez, como não são muito comuns não os exporemos em maiores detalhes, que podem ser encontrados na referência *68000 Assembly Language Programming* [1].

Endereçamento absoluto

No modo de endereçamento absoluto o endereço efetivo é o endereço real especificado na instrução. Existem dois modos absolutos de endereçamento: **absoluto curto** e **absoluto longo**, sendo que a única diferença entre eles é o intervalo dos valores permitidos. O modo de endereçamento curto permite que sejam usados endereços entre $0000 e $FFFF; no longo, o intervalo vai de $00000000 até $FFFFFFFF. O modo de endereçamento curto requer apenas mais uma palavra por instrução, enquanto que o longo requer mais duas palavras.

Como exemplo temos a instrução: `MOVE.L $2103,D2`. Nela o primeiro operando (fonte) usa o modo de endereçamento curto enquanto que o segundo operando (destino) usa o endereçamento direto do registrador de dados. Observe que o sufixo ".L" na instrução `MOVE` indica o tamanho do operando: uma palavra-longa será movida do endereço `$2103` para o registrador `D2`.

Endereçamento imediato

Discutimos anteriormente o endereçamento **imediato**. Ele significa que o dado sobre o qual age a instrução é especificado logo após a instrução propriamente dita: não há nenhum endereço efetivo para ser calculado. A instrução `MOVE.B #$89,-$135(A5)` coloca o valor do operando-fonte (`$89`) na posição de memória `$135` antes de `A5`. O símbolo "#" antes do operando-fonte indica que se trata de endereçamento imediato.

Uma variação deste modo é o endereçamento imediato rápido: funciona como o anterior, com a exceção de que o operando-fonte tem que caber em apenas 1 *byte* (para algumas instruções ele deve caber em apenas 3 bits). Quando se usa este modo de endereçamento o valor do operando-fonte é colocado junto da instrução isto é, ele passa a fazer parte da instrução propriamente dita. A instrução `MOVEQ #0,D3` moverá o valor zero para o registrador `D3` e ocupará somente 2 *bytes*. É claro que se poderia usar a instrução `MOVE` no lugar da `MOVEQ`, mas a vantagem é que `MOVEQ` é mais rápida.

Endereçamento implícito

O último tipo de endereçamento é o endereçamento de registrador **implícito**: aqui a instrução propriamente dita contém o endereço do operando e os operandos são registradores. O endereço efetivo está implícito na instrução. Por exemplo, a instrução `RTS` (*Return from Subroutine*) remove o endereço que está no topo da pilha, incrementa o ponteiro da pilha de 4 *bytes* (pois endereços têm 4 *bytes*) e coloca o endereço recém removido da pilha no contador do programa, fazendo com que o programa sofra um desvio no seu fluxo e recomece em outro ponto (que é o ponto imediatamente após a chamada da sub-rotina, como era de se esperar). O ponteiro da pilha é o operando implícito desta instrução.

Algumas instruções chegam mesmo a não ter nenhum operando, como é o caso da instrução NOP (*No Operation*).

Na tabela 1 , mostrada abaixo, temos um resumo simplificado dos modos de endereçamento do MC68020:

Tipo de Endereçamento	Modo	Exemplo
direto de registradores	registrador de dados	MOVE D0,D1
	registrador de endereços	MOVE A0,A1
indireto via registrador	via registrador de endereços (RE)	MOVE D0,(A0)
	pós-incremento, via RE	MOVE (A7)+,D0
	com pré-decremento, via RE	MOVE D0,-(A7)
	com deslocamento, via RE	MOVE $12(A6),D0
	com indexação e deslocamento, via RE	MOVE $12(A6,D1),D0
indireto via memória	pós-indexado	MOVE ([$10,A0],D0,$6),D1
	pré-indexado	MOVE ([$10,A0,D0],$6),D1
indireto via PC	com indexação	MOVE (-$22,PC),D0
	com indexação e deslocamento	MOVE ($1000,PC,D0),D1
indireto via mem apontada por PC	pós-indexado	MOVE ([$800,PC],D0,$5),D1
	pré-indexado	MOVE D1,([$800,PC,D0],$5)
absoluto	absoluto curto	MOVE $1024,D0
	absoluto longo	MOVE $12345678,D0
imediato	imediato	MOVE #1989,D0
	imediato rápido	MOVEQ #5,D0
implícito	NOP	

tabela 1: modos de endereçamento do MC68020

Conjunto de instruções do MC68020

O MC68020 contém um conjunto de instruções que pode ser dividido em 9 categorias:

- Transferência de dados
- Aritmética inteira
- Operações lógicas
- Deslocamentos e rotações
- Manipulação de bits individuais
- Manipulação de campos de bits
- BCD
- Controle do programa
- Controle do sistema

A seguir discutiremos cada uma destas 9 categorias, mas não iremos abordar todas as instruções, somente aquelas mais freqüentemente usadas pelo compiladores pois, afinal de contas, nosso objetivo é conseguir ler (e entender) a saída de um compilador quando necessitarmos depurar um programa. Para uma discussão detalhada das instruções disponíveis para MC68020 recomendamos a leitura das referências *68000 Assembly Language Programming* [1], 68000 Microprocessador [2] e *Programming the*

68000 [3] que tratam da família de microprocessadores 68000, incluindo as instruções somente disponíveis para o MC68020.

Transferência de dados

As instruções de transferência de dados servem para se copiar dados entre posições de memória e registradores.

A instrução mais comum deste tipo é `MOVE`: ela serve para se copiar dados entre 2 posições de memória, entre dois registradores ou entre uma posição de memória e um registrador, em qualquer sentido. Todos os modos de endereçamento são permitidos para o operando-fonte e quase todos para o destino. O dado a ser transferido pode ser um *byte*, uma palavra ou uma palavra-longa. Muitas variações da instrução `MOVE` realizam funções especiais. Entre elas podemos destacar as seguintes:

- `MOVEA`: move um valor para um registrador de endereços
- `MOVEM`: move múltiplos registradores para uma posição de memória
- `MOVEQ`: move um valor, geralmente de até 8 bits e implícito na instrução, para um registrador de dados

É interessante notar que uma atribuição numa linguagem de alto nível (por exemplo `X=21389`, em FORTRAN) geralmente é compilada para uma instrução `MOVE`.

Outras instruções de transferência de dados incluem `LEA` (*Load Effective Address*), que calcula um endereço efetivo e o carrega num registrador de endereços; `PEA` (*Push Effective Address*), que coloca um endereço efetivo na pilha; `LINK` e `UNLK`, usadas para se trabalhar com estruturas de dados chamadas de *stack frames* (empregadas na criação de variáveis locais quando da invocação de uma sub-rotina[1]); `EXG` (*Exchange*), que troca os valores entre dois registradores; e `SWAP`, que troca os valores entre as palavras mais significativa e menos significativa de um registrador de dados.

Na tabela 2 abaixo temos um resumo das operações de transferência de dados:

Instrução	Descrição
EXG	troca o conteúdo de 2 registradores
LEA	carrega um endereço efetivo num registrador
LINK	cria um *stackframe* para uma sub-rotina
MOVE	copia dados entre memórias e registradores
PEA	coloca um endereço efetivo na pilha
SWAP	troca as palavras mais e menos significativas de um registrador
UNLK	destroi um *stackframe* de uma sub-rotina

tabela 2: operações de transferência de dados

Aritmética inteira

As instruções de aritmética inteira servem para se realizar operações matemáticas (em complemento de 2) sobre os dados nos registradores e na memória. Em geral, adição, subtração, multiplicação e divisão numa linguagem de alto nível produzem estes tipos de instruções quando os operandos são inteiros.

As instruções de aritmética inteira incluem ADD, que soma dois operandos e coloca o resultado no segundo operando. A instrução ADD básica requer que um dos operandos seja um registrador de dados; o outro operando pode ser especificado por qualquer um dos modos de endereçamento já mostrados. Uma variação, ADDA (*Add Address*), permite que o operando-destino seja um registrador de endereços. Há também a instrução ADDI (*Add Immediate*) para adição de um valor imediato, bem como ADDQ (*Add Quick*), que é uma maneira rápida de se somar um valor imediato (contanto que ele esteja restrito ao intervalo 0 a 7, inclusive, pois precisa caber em 3 bits).

Da mesma forma, há um conjunto correspondente de instruções para subtrações: SUB, SUBA, SUBI e SUBQ.

Há duas instruções para se fazer multiplicações: MULU e MULS. A instrução MULU (*Unsigned Multiply*) multiplica duas palavras, sem sinal. Uma delas tem que ser um registrador de dados. O resultado é uma palavra-longa, sem sinal, e é armazenada no segundo operando. A outra instrução de multiplicação, MULS (*Signed Multiply*), é semelhante à MULU com a única diferença que os operandos (palavras) e o resultado (palavra-longa) são tratados como valores (com sinal) em complemento de dois. Como antes, o operando-destino tem que ser um registrador de dados.

Analogamente, há ainda um par de instruções para divisões: DIVU e DIVS. A instrução DIVU divide uma palavra-longa (operando-destino) por uma palavra (operando-fonte), deixando o resultado no operando-destino, sendo que o quociente é colocado na palavra menos significativa e o resto na mais significativa. Como na multiplicação, o operando-destino tem que ser um registrador de dados. A instrução DIVS é análoga, porém trata os números como sendo representados em complemento de dois (isto é, com sinal).

A instrução CMP (*Compare*) compara dois número e atualiza os códigos de condição adequadamente, dependendo do resultado da comparação. Um dos operandos tem que ser um registrador de dados, se se quiser usar a versão genérica da instrução CMP. Existem, porém, variações dela, como CMPA (compara um valor com um registrador de endereços), CMPI (compara um valor com um número imediato) e CMPM (compara os valores apontados por dois registradores de endereços). A comparação pode ser entendida como uma subtração (onde o resultado da subtração não é armazenado) da seguinte forma: DESTINO-FONTE.

Outra instrução que altera os valores dos códigos de condição é TST (*Test*), que compara um valor com o número zero e atualiza os códigos de condição adequadamente.

As instruções CMP (e sua variações) e TST geralmente são seguidas de uma instrução de desvio do fluxo do programa. Elas são geradas quando são compiladas as construções `for-endfor`, `if-endif`, `do-while` dos programas em linguagem de alto nível.

A instrução NEG (*Negate*) é usada para se negar um operando (convertê-lo para seu complemento de dois). Matematicamente corresponde a se multiplicar o número por -1. Com relação aos números na forma binária, corresponde a se inverter todos os seus bits e então somar 1.

Às vezes é preciso converter um determinado tipo de número em outro, por exemplo quando um programa em Pascal soma um inteiro (`integer`, de 2 *bytes*) a um inteiro-longo (`longinteger`, de 4 *bytes*), o programa tem que primeiro converter o inteiro para um inteiro-longo e então executar a soma. Para esclarecer, suponha que o valor inteiro seja `-14` e que o inteiro-longo seja `12`. Em hexadecimal estes números têm, respectivamente, as representações `$FFF2` e `$0000000C`. Antes que eles possam ser somados, o valor inteiro tem que ser convertido para o formato do inteiro-longo, que seria `$FFFFFFF2`. O MC68020 possui uma instrução que faz este tipo de conversão de formatos automaticamente: `EXT` (*Sign Extend*). Ela automaticamente copia o bit de mais alta ordem do operando (que é o bit de sinal) em todos os bits da palavra mais significativa, convertendo uma palavra numa palavra-longa, quer seu valor seja positivo ou negativo.

A instrução `CLR` (*Clear*) é usada para se colocar todos os bits do operando em zero. Se os sufixos "`.B`" ou "`.W`" forem usados, somente os bits correspondentes serão colocados em zero.

Na tabela 3 abaixo temos um resumo das instruções de aritmética inteira:

Instrução	Descrição
ADD	soma os operandos e deixa o resultado no operando-destino
CLR	coloca zero nos bits do operando
CMP	compara os operandos e atualiza os códigos de condição
DIVS	divide destino pelo fonte (com sinal), resultado no destino
DIVU	divide destino pelo fonte (sem sinal), resultado no destino
EXT	extende o sinal do operando
MULS	multiplica destino e fonte (com sinal), resultado no destino
MULU	multiplica destino e fonte (sem sinal), resultado no destino
NEG	nega (multiplica por -1) o operando
SUB	subtrai fonte do destino
TAS	testa e modifica um *byte*
TST	compara operando com zero e atualiza os códigos de condição

tabela 3: instruções de aritmética inteira

Operações lógicas

As instruções lógicas definidas para o MC68020 permitem que se faça as 4 operações lógicas básicas: **e-lógico** , **ou-lógico** , **não-lógico** e **ou-exclusivo-lógico** .

A instrução `AND` faz um e-lógico entre dois operandos (bit a bit) e armazena o resultado no operando-destino. Ela tem muitas variações, sendo que a mais comum é a `ANDI` (*And Immediate*) que é usada quando um dos operandos é um valor imediato.

A instrução `OR` faz um ou-lógico entre dois operandos (bit a bit) e armazena o resultado no operando-destino. Ela também tem muitas variações, sendo que a mais comum é a ORI (*Or Immediate*) que é usada quando um dos operandos é um valor imediato.

A instrução `EOR` faz um ou-exclusivo-lógico entre dois operandos (bit a bit) e armazena o resultado no operando-destino. Ela também tem muitas variações, sendo que a mais comum é a `EORI` (*Exclusive Or Immediate*) que é usada quando um dos operandos é um valor imediato.

A quarta instrução lógica é NOT, que inverte todos os bits do operando: os que tinham valor 0 passam a ter valor 1 e vice-versa.

Na figura 2 abaixo temos as tabelas-verdade das instruções AND, OR e EOR, que fornecem o resultado destas instruções em função do dois operandos sobre os quais elas atuam.

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

EOR	0	1
0	0	1
1	1	0

figura 2: tabelas-verdade das instruções AND, OR e EOR

Na tabela 4 abaixo temos um resumo das instruções lógicas:

Instrução	Descrição
AND	executa um e-lógico bit a bit, entre os operandos
OR	executa um ou-lógico bit a bit, entre os operandos
EOR	executa um ou-exclusivo-lógico bit a bit, entre os operandos
NOT	inverte todos os bits do operando

tabela 4: instruções lógicas

Deslocamentos e rotações

As instruções de deslocamento fazem com que os bits de um operando sejam deslocados (tanto para a direita quanto para a esquerda). Como conseqüência do uso delas alguns bits são "perdidos" quando são deslocados para "fora" do operando. Já com as instruções de rotação, os bits que são deslocados para fora do operando são re-inseridos nele pelo outro lado.

Existem quatro instruções de deslocamento de bits: ASL (*Arithmetic Shift Left*), ASR (*Arithmetic Shift Right*), LSL (*Logical Shift Left*) e LSR (*Logical Shift Right*).

As instruções aritméticas (ASL e ASR) servem para se multiplicar ou dividir (por potências de 2) números inteiros, pois multiplicar um número por 2 é a mesma coisa que deslocar sua representação binária para a esquerda (com ASL) enquanto que dividir um número por 2 é a mesma coisa que deslocar sua representação binária para a direita (com ASR).

Já as instruções lógicas (LSL e LSR) fazem praticamente a mesma coisa, com a exceção de que as aritméticas preservam o bit de sinal enquanto que as lógicas não. O bit que "sobra" é colocado no bit C do CCR, enquanto que o bit que ficou "vazio" é preenchido com 0.

As instruções de rotação fazem com que os bits de um operando sejam deslocados (seja para a direita ou para a esquerda) de modo que o bit que "sobra" é colocado no bit que ficou "vazio". As instruções de rotação geralmente não são usadas na geração de códigos pelos compiladores.

Na tabela 5 abaixo temos um resumo das instruções de deslocamento e de rotação:

Instrução	Descrição
ASL	deslocamento aritmético para a esquerda
ASR	deslocamento aritmético para a direita
LSL	deslocamento lógico para a esquerda
LSR	deslocamento lógico para a direita
ROL	rotação para a esquerda
ROR	rotação para a direita
ROXL	rotação para a esquerda com extensão
ROXR	rotação para a direita com extensão

tabela 5: instruções de deslocamento e de rotação

Manipulação de bits individuais

Há um conjunto de instruções para se testar e manipular bits individuais dentro dos operandos: `BTST` (*Bit Test*), `BSET` (*Bit Test and Set*), `BCLR` (*Bit Teste and Clear*) e `BCHG` (*Bit Test and Change*).

A instrução `BTST` é usada para se determinar se o valor de um bit é `0` ou `1`. O código de condição `Z` (também chamado de *flag* `Z`) é modificado de acordo com resultado: ele é colocado em `1` se o bit testado for `0` e é colocado em `0` se o bit testado for `1`. Com a instrução `BTST #3,D3`, o quarto bit (bit número `3`) do registrador `D3` é testado. Como ele é igual a `1` (como pode ser visto na figura 3 abaixo) o *flag* `Z` do código de condição é colocado em `0`.

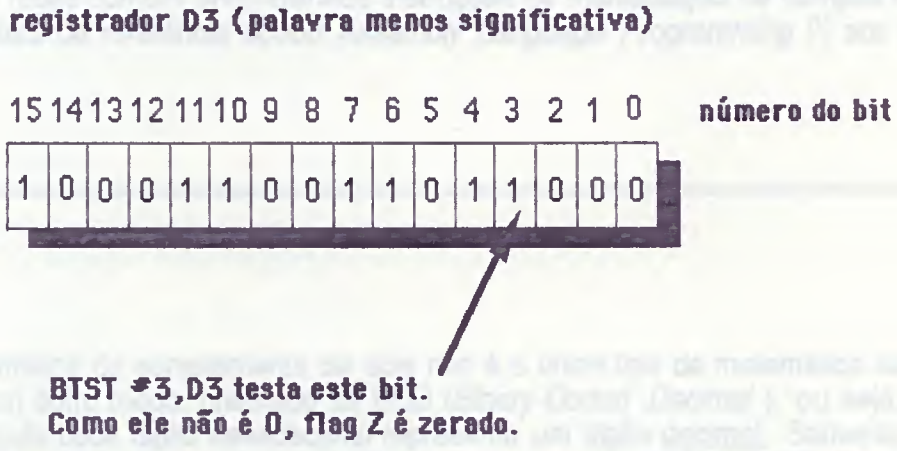


figura 3: instrução bit test

A instrução BSET primeiro testa o bit e atualiza o *flag* Z, da mesma forma que a instrução anterior, porém, após o teste, ela muda o valor do bit testado para 1.

A instrução BCLR é semelhante à instrução BSET, com a única diferença que após o teste ela muda o valor do bit testado para 0.

A instrução BCHG também testa o bit e modifica o *flag* Z de acordo com o resultado do teste. Porém, após o teste, ela inverte o valor do bit testado isto é, se ele tinha o valor 0 esta instrução modifica seu valor para 1 e vice-versa.

É interessante saber que o fato de as instruções BSET e BCLR testarem o bit do operando antes de alterá-lo (para 1 e 0, respectivamente) não tem muita importância: geralmente o compilador, ou o programador, somente deseja alterar o bit, sem se importar com o valor que ele tinha antes da alteração.

Na tabela 6 abaixo temos um resumo das instruções de manipulação de bits:

Instrução	Descrição
BCHG	testa o bit, modifica o *flag* Z e inverte o bit testado
BCLR	testa o bit, modifica o *flag* Z e coloca 0 no bit testado
BSET	testa o bit, modifica o *flag* Z e coloca 1 no bit testado
BTST	testa o bit, modifica o *flag* Z

tabela 6: instruções de manipulação de bits

Manipulação de campos de bits

Além de permitir a manipulação de bits individuais, o MC68020 tem um conjunto de instruções para se manipular conjuntos de bits, chamados de campos de bits. Este campos podem ter um tamanho de até 32 bits. Como não é muito comum encontrarmos instruções de manipulação de campos de bits, recomendamos a leitura da referência *68000 Assembly Language Programming* [1] aos mais interessados.

BCD

A aritmética de complemento de dois não é o único tipo de matemática com que o MC68020 consegue lidar: há um outro modo, chamado de BCD (*Binary-Coded Decimal*), ou seja, decimal codificado em binário. Neste modo cada dígito hexadecimal representa um dígito decimal. Somente valores hexadecimais entre $0 e $9 são permitidos; os dígitos hexadecimais de $A até $F são considerados ilegais por não serem usados no sistema decimal. Nesta representação, por exemplo, o valor hexadecimal $28 representa o número decimal 28.

No MC68020 existem instruções para somar, subtrair e mudar o sinal de números no sistema BCD. Este modo não é muito utilizado uma vez que o modo que usa complemento de 2 é mais eficiente. Para uma discussão detalhada sobre o sistema BCD recomendamos a leitura da referência *68000 Assembly Language Programming* [1].

Controle do programa

O MC68020 tem um conjunto de instruções que permite que o fluxo do programa seja alterado. Estas instruções são o análogo do `GO TO` e `CALL` do FORTRAN: elas fazem com que a execução do programa continue em algum outro ponto.

Em linguagem *assembly* há três tipos de instruções que controlam o fluxo do programa:

- instruções condicionais
- instruções incondicionais
- instruções de retorno

A primeira instrução condicional é o desvio condicional: esta instrução testa os códigos de condição de acordo com uma certa condição: se o teste for verdadeiro ocorre um desvio para outra parte do programa; se for falso a instrução seguinte é executada. A forma geral desta instrução é escrita como `Bcc`, que significa "desvie baseado nos códigos de condição". Na verdade há 14 tipos diferentes de desvio, sendo que cada um deles testa um tipo de condição. Por exemplo, a instrução `BEQ` significa "desvie se igual" (*Branch If Equal*).

Os 14 tipos diferentes de desvio podem ser vistos na tabela 7 a seguir:

Instrução	Descrição
BCC	desvia se o *flag* C (*carry*) tiver 0
BCS	desvia se o *flag* C (*carry*) tiver 1
BEQ	desvia se igual (*flag* Z igual a 1)
BGE	desvia se maior ou igual
BGT	desvia se maior
BHI	desvia se os *flags* C e Z tiverem 0
BLE	desvia se menor ou igual
BLS	desvia se os *flags* C ou Z tiverem 1
BLT	desvia se menor
BMI	desvia se negativo (*flag* N igual a 1)
BNE	desvia se diferente
BPL	desvia se positivo (*flag* N igual a 0)
BVC	desvia se não há *overflow* (*flag* V igual a 0)
BVS	desvia se há *overflow* (*flag* V igual a 1)

tabela 7: instruções de desvio baseado nos códigos de condição

Geralmente uma instrução Bcc sucede uma instrução de comparação, que modifica os códigos de condição. Por exemplo, um programa pode comparar dois valores para saber se eles são iguais e então efetuar um desvio (se forem) com a instrução CMP seguida de BEQ.

Os compiladores usam as instruções Bcc para implementar os comandos "se-então-senão": primeiro é feita a comparação; então se a comparação falha há um desvio para depois do "então" bem para o comando "senão", se houver. Desvios condicionais depois de uma instrução CMP também são usados para se testar os limites num laço (do-end do, no FORTRAN), por exemplo.

Há ainda uma forma mais sofisticada de desvio, chamada de DBcc (*Decrement and Branch on Condition Code*). Esta instrução sozinha realiza, na verdade, 3 funções: ela testa uma condição; se a condição for falsa um registrador de dados é decrementado, e se este registrador de dados não contiver -1, um desvio é feito. Se a condição inicial for satisfeita (verdadeira) ou se o registrador decrementado contiver -1 então não há desvio e a próxima instrução é executada. Isto significa que podemos ter um laço que pode ser finalizado de duas maneiras: ou a condição é satisfeita ou o valor do registrador de dados é menor do que zero.Esta instrução geralmente é usada pelos compiladores para otimizar os laços. As condições para o desvio podem ser quaisquer das mostradas na tabela 7 acima, e mais outras duas: DBT (*Decrement and Branch until True*), que sempre dá como falso o resultado da comparação; e DBF (*Decrement and Branch until False*), que sempre executará o desvio, significando que apenas quando o registrador de dados contiver um valor menor do que 0 o laço terminará. Ela também é escrita, mais freqüentemente, como DBRA (*Decrement and Branch Always*).

O terceiro e último tipo de instrução condicional é Scc (*Set Byte on Condition Code*). Ela testa uma dada condição; se o resultado for verdadeiro o *byte* especificado como sendo o operando tem todos os seus bits colocados em 1 ($FF); se a condição falhar ele tem todos os seus bits colocados em 0 ($00). Além dos 14 tipos de teste mostrados na tabela 7, há ainda o ST (*Set If True*) que sempre coloca os bits do operando em 1; e SF (*Set If False*) que sempre coloca os bits do operando em 0.

O primeiro tipo de instrução incondicional é BSR (*Branch To Subroutine*), que é o equivalente à chamada de sub-rotina do FORTRAN (call). Ela causa um desvio incondicional (que não depende de nenhuma condição) para outro ponto do programa, mas antes do desvio ela armazena o endereço da próxima instrução na pilha. Isto permite que a sub-rotina chamada possa retornar para o programa que a chamou, executando a instrução RTS(ver sua descrição mais adiante).

Outra instrução incondicional é JSR (*Jump To Subroutine*), que é muito parecida com a BSR. Da mesma forma que a BSR, a instrução JSR guarda na pilha o endereço de retorno e executa um desvio incondicional. A única diferença entre elas é o modo de endereçamento disponível para os operandos. A instrução JSR permite que o endereço efetivo seja especificado por qualquer um de muitos modos de endereçamento. Já a BSR necessita que o modo de endereçamento usado seja relativo ao contador do programa (PC).

A terceira instrução incondicional, chamada de BRA (*Branch Always*), é similar às instruções Bcc. Ela é usada para causar um desvio para outro local do programa, sem depender de condição nenhuma. Ao contrário de BRS e JSR, ela não armazena na pilha o endereço de retorno.

A quarta e última instrução incondicional, da mesma forma que BRA, causa um desvio incondicional sem retorno. Esta instrução, chamada de JMP (*Jump*), somente difere da BRA nos modos de endereçamento disponíveis. A instrução JMP permite o mesmo conjunto de modos de endereçamento que JSR, enquanto que BRA necessita de endereçamento relativo ao contador do programa (PC) .

A instrução de retorno é RTS (*Return From Subroutine*), que retira o endereço do topo da pilha e o coloca no contador do programa (PC).

Algumas instruções de controle do sistema podem ser vistas na tabela 8 abaixo:

Instrução	Descrição
Bcc	desvia de acordo com os códigos de condição
DBcc	testa a condição, decrementa um contador e desvia de acordo com os códigos de condição
Scc	modifica um *byte* de acordo com os códigos de condição
BRA	desvia sempre (com endereçamento relativo ao PC)
BSR	desvia para a sub-rotina
JMP	desvia sempre
JSR	desvia para a sub-rotina (com endereçamento relativo ao PC)
RTS	retorne da sub-rotina

tabela 8: instruções de controle do programa

Controle do sistema

As instruções de controle são aquelas que afetam o estado do sistema. Normalmente somente uma delas aparece nos códigos gerados pelo compilador: CHK (*Check*). Ela é usada pelos compiladores para verificar se os intervalos (por exemplo os elementos de um vetor) gerados em tempo real são compatíveis com a definição dada à variável sendo verificada (como, por exemplo, qundo o programa-fonte é compilado com a instrução para verificação de tamanho de vetores e matrizes). Ela necessita de dois operandos: o primeiro (fonte) contém o endereço efetivo que armazena o maior valor possível para o intervalo; o segundo operando tem que ser um registrador de dados que contém o valor sendo verificado.

Algumas instruções de controle do sistema podem ser vistas na tabela 9 abaixo:

Instrução	Descrição
BKP	estabelece *breakpoint*
CHK	*trap* para operando fora dos limites
RESET	executa um *reset* nos dispositivos
STOP	pára o processador
TRAP	*trap* via tabela de exceções

tabela 9: instruções de controle do sistema

Referências

[1] L.Leventhal et al., *68000 Assembly Language Programming*, second edition, McGraw-Hill,1986

[2] W. Cramer e G.Kane, 68000 Microprocessador, McGraw-Hill, 1986

[3] S.Williams, *Programming the 68000*, SYBEX, 1985

apêndice B

Listagem de CALOR (sem ACP)

CALOR_SEM_ACP

Esta é a listagem do programa que roda no computador convencional (MicroVAX II), com apenas 1 processador

Inclui:

- CALOR_SEM_ACP: programa principal
- sub-rotina ctes_iniciais
- sub-rotina poe_A_em_B
- sub-rotina imprime

```
      program calor_sem_ACP
      implicit none

      integer MAX_L, MAX_C
      parameter (MAX_L = 600, MAX_C = 600)
      real*8 ANTERIOR(MAX_L,MAX_C), ATUAL(MAX_L,MAX_C)
      real*8 AUXILIAR, ROJAC2, OMEGAENE, RESID, PRECIS, MAIOR_EPSILON, AUX1
      real*4 TEMPO
      integer L, C, NUM_LINHAS, NUM_COLUNAS, ENE
      logical PARA, OUT, FIM
      real*8 ZERO, MEIO, UM, DOIS, QUARTO, PI, QUATRO
      parameter (ZERO=0.0d0, MEIO=0.5d0, UM=1.0d0, DOIS= 2.0d0)
      parameter (QUARTO=0.25d0, PI=3.1415926535897930d0, QUATRO=4.0d0)

C PONTO DE ENTRADA
2     call parametros_iniciais(NUM_LINHAS,NUM_COLUNAS,PRECIS, OUT,FIM)
      if (FIM) go to 1
      call cond_iniciais(ANTERIOR,NUM_LINHAS,NUM_COLUNAS)
      write(*,*) ' '
      write(6,*) 'Linhas usadas = ', NUM_LINHAS
      write(6,*) 'Colunas usadas = ', NUM_COLUNAS
      write(6,*) 'Precisao = ', PRECIS
      TEMPO = secnds(0.0)
      ENE = 0
      PARA = .false.
      ROJAC2 =((DCOS(PI/NUM_LINHAS)+DCOS(PI/NUM_COLUNAS))*MEIO)**DOIS
      OMEGAENE = 1

      do while (.not.(PARA))
        ENE = ENE + 1
        call poe_A_em_B(ANTERIOR,ATUAL,NUM_LINHAS,NUM_COLUNAS)
C Resolve pelo SOR
        do C = 2, NUM_COLUNAS-1
          do L = 2, NUM_LINHAS-1
            if (MOD(L+C,2) .eq. MOD(ENE,2)) then
              RESID = ANTERIOR(L-1,C) + ANTERIOR(L+1,C) +
                      ANTERIOR(L,C-1) + ANTERIOR(L,C+1)
                      - QUATRO * ANTERIOR(L,C)
              ATUAL(L,C) = ATUAL(L,C) + OMEGAENE*RESID*QUARTO
            end if
          end do
        end do
C Calcula se pode parar
        MAIOR_EPSILON = 0.0
        do C = 2, NUM_COLUNAS-1
          do L = 2, NUM_LINHAS-1
            MAIOR_EPSILON = MAIOR_EPSILON +   abs( ATUAL(L,C) - ANTERIOR(L,C) )
          end do
        end do
        if (MAIOR_EPSILON .le. PRECIS) PARA = .true.
C Atualiza valores importantes
        call poe_A_em_B(ATUAL,ANTERIOR,NUM_LINHAS,NUM_COLUNAS)
        if (ENE . eq. 1) then
          OMEGAENE = UM / (UM - MEIO*ROJAC2)
        else
          OMEGAENE = UM / (UM - QUARTO * ROJAC2 * OMEGAENE)
        end if
      end do

C FINALIZACAO
      TEMPO = secnds(TEMPO)
      write(6,*) 'Tempo total de execucao = ',TEMPO,' s'
      if (OUT) then
        write(6,*) 'Matriz resposta:'
        call imprime(ATUAL,NUM_COLUNAS,NUM_LINHAS)
      end if
      go to 2
1     stop
      end
```

```
      subroutine ctes_iniciais(NUM_LINHAS,NUM_COLUNAS,PREC,O,F)
      integer MAX_L, MAX_C, NUM_LINHAS,NUM_COLUNAS
      parameter (MAX_L = 600, MAX_C = 600))
      real*8 PREC, AUX
      logical O,F
C Numero de LINHAS
      WRITE(*,*)'Numero de linhas? (3 a',MAX_L,')'
      READ(*,*) NUM_COLUNAS
      if (NUM_COLUNAS.lt. 0) FIM = .true.
      if (NUM_COLUNAS.gt. MAX_L) NUM_COLUNAS= MAX_L
      if (NUM_COLUNAS.lt. 3) NUM_COLUNAS= 3
C Numero de COLUNAS
      WRITE(*,*)'Numero de colunas? (3 a',MAX_C ,')'
      READ(*,*) NUM_COLUNAS
      if (NUM_COLUNAS.lt. 0) FIM = .true.
      if (NUM_COLUNAS.gt. MAX_C ) NUM_COLUNAS= MAX_C
      if (NUM_COLUNAS.lt. 3) NUM_COLUNAS= 3
C PRECISAO
      WRITE(*,*)'Precisao entre 2 iteracoes consecutivas?'
      READ(*,*) PREC
C Imprime matriz resposta no final (0=nao, <>0=sim)
      O = .false.
      WRITE(*,*)'Imprimir matriz resposta no final ( <>0 => SIM)'
      READ(*,*) AUX
      if (AUX .ne. 0) O = .true.
C Fim do programa?
      F = .false.
      WRITE(*,*)'Quer terminar com o programa? (0 -> sim)'
      READ(*,*) AUX
      if (AUX .eq. 0) F = .true.
      return
      end



      subroutine poe_A_em_B(A,B,NUM_LINHAS,NUM_COLUNAS)
      integer MAX_L, MAX_C, L, C, NUM_LINHAS, NUM_COLUNAS
      parameter (MAX_L = 600, MAX_C = 600))
      real*8 A(MAX_L,MAX_C), B(MAX_L,MAX_C)
      do L = 1, NUM_LINHAS
        do C = 1, NUM_COLUNAS
          B(L,C) = A(L,C)
        end do
      end do
      return
      end


      subroutine imprime(MATRIZ, NUM_LINHAS, NUM_COLUNAS)
      integer MAX_L, MAX_C, L, C, NUM_LINHAS, NUM_COLUNAS
      parameter (MAX_L = 600, MAX_C = 600))
      real*8 A(MAX_L,MAX_C)
      if (NUM_COLUNAS.le. 20) then
        do L = 1, NUM_LINHAS
          WRITE(6,1) (A(L,C), C = 1,NUM_COLUNAS)
        end do
      end if
      if (NUM_COLUNAS.gt. 20) then
        do L = 1, NUM_LINHAS
          WRITE(6,*) 'Linha ',L
          do C = 1,NUM_COLUNAS
            WRITE(6,2) C, A(L,C)
          end do
        end do
      end if
      return
1     format(1X,<NUM_COLUNAS>f4.0)
2     format(1X,i5,f4.0)
      end
```

apêndice C

Listagem de CALOR (Hospedeiro-Nós)

CALOR_H_N.UPF

Este arquivo contém as instruções para que os programas do computador hospedeiro e o dos nós processadores sejam compilados. É o arquivo tipo UPF -

Usam-se 12 placas ACP do Centro Brasileiro de Pesquisas Físicas (CBPF).

```
system = CBPF
host source = CALOR_H_N_HOST.for
node source = CALOR_H_N_NODE.for
node main subroutine = SORCHE
node blocks = 1:PARAME, 2:BORDAS, 3:MATRIZ, 4:DIFERE, 5:VARRE
class 1: nodes = 12 68020
```

CALOR_H_N_HOST.for

Esta é a listagem do programa que roda no computador hospedeiro .

Inclui:

- CALOR_H_N: programa principal
- sub-rotina ctes_iniciais
- sub-rotina cond_iniciais
- sub-rotina imprime
- sub-rotina carga

```
      Program CALOR_H_N
      implicit none
C-----------------------------------------------------------------------
C Carrega arquivo de definições para uso do ACP
      include 'ACP$AREA:USER_COMMON.INC'
C-----------------------------------------------------------------------

C
C Versao do programa
C
      character*11 VERSAO
      parameter (VERSAO = 'VERSAO 6.1')

C
C Numero maximo de processadores usados neste programa
C
      integer N_PROC
      parameter (N_PROC = 10)


C
C Maior matriz possivel de ser resolvida por este programa
C
      integer HOST_LINHAS, HOST_COLUNAS
      parameter (HOST_LINHAS  = 400)
      parameter (HOST_COLUNAS = 400)

C
C Maior matriz possivel de ser resolvida por cada um dos nos
C
      integer NO_LINHAS, NO_COLUNAS
      parameter (NO_LINHAS  = HOST_LINHAS/2+2)
      parameter (NO_COLUNAS = HOST_COLUNAS)

C
C Variaveis para referenciar partes especificas das matrizes nos nos
C
      integer SUPERIOR, INFERIOR, INTERNO
      parameter (SUPERIOR = 1)
      parameter (INFERIOR = 2)
      parameter (INTERNO  = 3)


C
C Variaveis tipo COMMON, para comunicaçao com os nos
C
      integer PARAME, BORDAS, MATRIZ, DIFERE, VARRE
      parameter (PARAME = 1)
      parameter (BORDAS = 2)
      parameter (MATRIZ = 3)
      parameter (DIFERE = 4)
      parameter (VARRE  = 5)

C S_ representa # de palavras de 32 bits
      integer  S_PARAME, S_BORDAS, S_MATRIZ, S_DIFERE, S_VARRE
      parameter (S_PARAME = 6)
      parameter (S_BORDAS = 2* NO_COLUNAS*2)
      parameter (S_MATRIZ = NO_LINHAS*NO_COLUNAS*2)
      parameter (S_DIFERE = 1)
      parameter (S_VARRE  = HOST_LINHAS*HOST_COLUNAS*2)


      integer N_LINHAS, N_COLUNAS, N_DO_PROC, POSICAO, PTR_SUP_NO, PTR_INF_NO
      common /PARAME/ N_LINHAS,          !# de linhas deste RUN
     &                N_COLUNAS,                !# de colunas deste RUN
     &                N_DO_PROC,                !identifica o processador
     &                POSICAO,                  !localiza o processador na matriz
     &                PTR_SUP_NO,               !PTR_SUPERIOR deste no' N_DO_PROC
     &                PTR_INF_NO                !PTR_INFERIOR deste no' N_DO_PROC
```

```
      real*8 DUAS_BORDAS(2*NO_COLUNAS) !inferior:[1..NO_COLUNAS] e
      common/BORDAS/DUAS_BORDAS        !superior[NO_COLUNAS+1..2*NO_COLUNAS]

      real*8 M_ANT(NO_LINHAS, NO_COLUNAS)
      common/MATRIZ/M_ANT

      real*8 EPSILON
      common/DIFERE/EPSILON

      integer ENE
      common/VARRE/ENE     !iteraçao. N=0, 1, 2 ...

C
C Variaveis locais
C
      real*4 TEMPO

      real*8 ATUAL(HOST_LINHAS, HOST_COLUNAS),   !matriz resposta
     &       MATRIZ_DO_NO(NO_LINHAS, NO_COLUNAS),
     &       MAIOR_EPSILON,
     &       PRECISAO

      integer VAR_PARAME(S_PARAME),       ! constantes de cada no
     &        SIZE,                       !tamanho dos blocos a serem enviados
     &        NO,                         !para referenciar os nos
     &        CLASS,                      !classe (unica) dos nos
     &        PTR_INFERIOR(N_PROC),       !borda enviada para os nos
     &        PTR_SUPERIOR(N_PROC),!      borda enviada para os nos
     &        AUXILIAR_SUP, AUXILIAR_INF,
     &        L, C,
     &        LINHAS_USADAS, COLUNAS_USADAS,!dados do usuario
     &        PROC_USADOS,                  !dados do usuario
     &        RESTO, QUOCIENTE,
     &        OUTPUT,                     !imprimir resultado no fim. (0=nao)
     &        ENEAUX,

      logical PARA,                       !.true. caso fim do programa
     &        SEND_DONE, GET_DONE,        !para se lidar com o ACP
     &        FIM                                   !terminar com programa em batch

      real*8 ZERO, MEIO, UM, QUATRO
      parameter (ZERO=0.0d0, MEIO=0.5d0, UM=1.0d0, QUATRO=4.0d0)

C=====================================================================
C                  PONTO DE ENTRADA
C=====================================================================
      call ACP_INIT
      CLASS = 1

C
C ponto de entrada secundario, para rodar o programa varias vezes
C
1     WRITE(6,*)'-------------------------------------------------'
      WRITE(6,*)VERSAO
      WRITE(6,*)'Para parar com o programa, entre com pelo menos uma'
      WRITE(6,*)' das 4 condiçoes abaixo como sendo negativa:'
      WRITE(6,*)' '
      call ctes_iniciais(LINHAS_USADAS, COLUNAS_USADAS, PRECISAO, PROC_USADOS, FIM, OUTPUT)
C
C acaba se FIM=.true.
C
      if (FIM) go to 2

      WRITE(6,*)' '
      WRITE(6,*)'Linhas usadas = ',LINHAS_USADAS
      WRITE(6,*)'Colunas usadas= ',COLUNAS_USADAS
      WRITE(6,*)'Precisao usada= ',PRECISAO
      WRITE(6,*)'Procs. usados = ',PROC_USADOS
```

```
C
C note que ATUAL fica com dados no formato dos NOS processadores
C
      call cond_iniciais(ATUAL, LINHAS_USADAS, COLUNAS_USADAS)
      SIZE = S_ATUAL
      call acp_convert_h_n(ATUAL, ATUAL, SIZE, ACP_R_8, CLASS)



C=====================================================================
C ======
C FASE 0
C ======
C                              INICIACAO
C
C=====================================================================

      TEMPO = secnds(0.0)

C
C Prepara vetores que apontam para as bordas
C

      call carga(PTR_INFERIOR,PTR_SUPERIOR,LINHAS_USADAS,PROC_USADOS)


C=====================================================================
C ======
C FASE 1
C ======
C                       INICIACAO DOS PROCESSADORES
C
C=====================================================================


C
C Manda os 6 parametros importantes para todos os processadores
C

      SIZE = S_PARAME
      do NO = 1, PROC_USADOS
        VAR_PARAME(1) = LINHAS_USADAS
        VAR_PARAME(2) = COLUNAS_USADAS
        VAR_PARAME(3) = NO

        VAR_PARAME(4) = INTERNO
        if (NO .eq. 1)           VAR_PARAME(4) = SUPERIOR
        if (NO .eq. PROC_USADOS) VAR_PARAME(4) = INFERIOR
        VAR_PARAME(5) = PTR_SUPERIOR(NO)
        VAR_PARAME(6) = PTR_INFERIOR(NO)

        call acp_convert_h_n(VAR_PARAME,VAR_PARAME,SIZE,ACP_I_4,CLASS)
        SEND_DONE = .FALSE.
        do while (.not.SEND_DONE)
          call acp_sendblock(PARAME, VAR_PARAME, SIZE, NO, CLASS, ACP$NO_GO, SEND_DONE)
        end do
      end do



C=====================================================================
C                          PARTE PRINCIPAL
C=====================================================================
      PARA = .FALSE.
      EXEAUX = 1
```

```
      do while (.not.PARA)

C=====================================================================
C ======
C FASE 2
C ======
C        para cada um dos PROC_USADOS blocos em que foi subdividido o dominio
C          distribua as bordas para os respectivos processadores
C          faça os processadores calcularem as suas partes
C        fim
C=====================================================================

C
C Distribua as bordas para os processadores
C
       SIZE = S_VARRE                !conta o # de iteracoes
       call acp_convert_h_n(ENEAUX, ENE, SIZE, ACP_I_4, CLASS)
       do NO = 1, PROC_USADOS
         SIZE = S_BORDAS
         if (NO .eq. 1) then                    !Borda superior
           AUXILIAR_SUP = PTR_SUPERIOR(NO)
           do C = 1, COLUNAS_USADAS
             DUAS_BORDAS(C+NO_COLUNAS) = ATUAL(AUXILIAR_SUP,C)
           end do
         end if
         if (NO .eq. PROC_USADOS) then          !Borda inferior
           AUXILIAR_INF = PTR_INFERIOR(NO)
           do C = 1, COLUNAS_USADAS
             DUAS_BORDAS(C) = ATUAL(AUXILIAR_INF,C)
           end do
         end if

         SEND_DONE = .FALSE.
         do while (.not.SEND_DONE)
           call acp_sendblock(BORDAS, DUAS_BORDAS, SIZE, NO,CLASS, ACP$NO_GO, SEND_DONE)
         end do

C
C Faça o processador NO calcular sua parte
C
         SIZE = S_VARRE
         SEND_DONE = .FALSE.
         do while (.not.SEND_DONE)
           call acp_sendblock(VARRE, ENE, SIZE, NO,CLASS,  ACP$GO, SEND_DONE)
         end do
       end do


C=====================================================================
C ======
C FASE 3
C ======
C        para cada um dos PROC_USADOS blocos em que foi subdividido o dominio
C          pegue o resultado EPSILON dos processadores que ja terminaram
C          e calcule o MAIOR_EPSILON
C        fim
C=====================================================================
       SIZE = S_DIFERE
       MAIOR_EPSILON = ZERO
       do NO = 1, PROC_USADOS
         GET_DONE = .FALSE.
         do while (.not.GET_DONE)
           call acp_getblock(DIFERE, EPSILON, SIZE, NO,CLASS, ACP$MORE_BLOCKS, GET_DONE)
         end do
         call acp_convert_n_h(EPSILON, EPSILON, SIZE, ACP_R_8, CLASS)
         MAIOR_EPSILON = MAIOR_EPSILON +    EPSILON
         end if
       end do
```

```
C=================================================================
C ======
C FASE 4
C ======
C       calcule se pode parar de calcular e se puder entao faça
C            para cada 1 dos PROC_USADOS blocos em que foi dividido o dominio
C               pegue as submatrizes dos respectivos processadores, juntando-as
C               ... para formar a matriz resposta ATUAL
C            senao
C               pegue os resultados das bordas dos respectivos processadores e
C               avise os processadores que eles estaao livres (ready) e
C               aumenta N de 1
C               recomece na fase 2
C=================================================================

C Verifica se ja pode parar. Se puder, monte a matriz resposta de volta

      if (MAIOR_EPSILON .le. PRECISAO) then
        PARA = .TRUE.
        SIZE = S_ATUAL
        call acp_convert_n_h(ATUAL, ATUAL, SIZE, ACP_R_8, CLASS)
        SIZE = S_MATRIZ
        do NO = 1, PROC_USADOS
          L = 2
          AUXILIAR_INF = PTR_INFERIOR(NO) - 1
          AUXILIAR_SUP = PTR_SUPERIOR(NO) + 1

          GET_DONE = .FALSE.
          do while (.not.GET_DONE)
            call acp_getblock(MATRIZ,MATRIZ_DO_NO,SIZE, NO,CLASS,ACP$LAST_BLOCK, GET_DONE)
          end do
          call acp_convert_n_h(MATRIZ_DO_NO, MATRIZ_DO_NO, SIZE,  ACP_R_8, CLASS)

          do while (AUXILIAR_SUP .le. AUXILIAR_INF)
            do C = 1, COLUNAS_USADAS
               ATUAL(AUXILIAR_SUP,C) = MATRIZ_DO_NO(L,C)
            end do
            AUXILIAR_SUP = AUXILIAR_SUP + 1
            L = L + 1
          end do

        end do
C Ainda nao pode parar, portanto pegue os resultados das bordas dos
C   respectivos processadores e continue
        else
          ENEAUX = ENEAUX + 1
          SIZE = S_BORDAS
          do NO = 1, PROC_USADOS
            GET_DONE = .FALSE.

            do while (.not.GET_DONE)
              call acp_getblock(BORDAS,DUAS_BORDAS,SIZE,NO,CLASS,ACP$LAST_BLOCK, GET_DONE)
            end do
            if (NO .ne. 1) then                    !Borda inferior
              AUXILIAR_SUP = PTR_SUPERIOR(NO) + 1
              do C = 1, COLUNAS_USADAS
                ATUAL(AUXILIAR_SUP,C) = DUAS_BORDAS(C)
              end do
            end if
            if (NO .ne. 1PROC_USADOS) then         !Borda superior
              AUXILIAR_INF = PTR_INFERIOR(NO) - 1
              do C = 1, COLUNAS_USADAS
                ATUAL(AUXILIAR_SUP,C) = DUAS_BORDAS(C+NO_COLUNAS)
              end do
            end if
          end do
        end if

      end do  !while (.not.PARA)
```

```
C=====================================================================
C       FINALIZACAO
C=====================================================================

        TEMPO = secnds(TEMPO)

        if (OUTPUT .ne. 0) then
          WRITE(6,*)'-------------------'
          WRITE(6,*)'- Matriz resposta -'
          WRITE(6,*)'-------------------'
          call imprime (ATUAL, LINHAS_USADAS, COLUNAS_USADAS)
        end if
        WRITE(6,*)' '
        WRITE(6,*)'tempo total de execuçao= ',TEMPO, 's'
        WRITE(6,*)'-----------------------------------------'
        WRITE(6,*)' '


C
C Volta para rodar de novo, pois nenhuma das constantes iniciais foi <0
C
        go to 1

C
C Fim do programa, pois uma das constantes iniciais foi negativa
C

2       call ACP_EXIT
        stop 'PROGRAMA DE J.RODOLFO F.XAVIER - IFT - S.PAULO'
        end
```

```
      subroutine ctes_iniciais(LINHAS_USADAS,COLUNAS_USADAS,PRECISAO, PROC_USADOS,FIM,OUTPUT)

C Comum ao programa principal
        integer N_PROC
        parameter (N_PROC = 10)

        integer HOST_LINHAS, HOST_COLUNAS
        parameter (HOST_LINHAS  = 400)
        parameter (HOST_COLUNAS = 400)
C Saida
        integer LINHAS_USADAS, COLUNAS_USADAS, PROC_USADOS, OUTPUT
        real*8 PRECISAO
        logical FIM
C
C Qualquer numero negativo implica fim do programa
C
        FIM = .false.
C
C Numero de LINHAS
C
        WRITE(*,*)'Numero de linhas? (3 a',HOST_LINHAS,')'
        READ(*,*) LINHAS_USADAS
        if (LINHAS_USADAS .lt. 0) FIM = .true.
        if (LINHAS_USADAS .gt. HOST_LINHAS) LINHAS_USADAS = HOST_LINHAS
        if (LINHAS_USADAS .lt. 3) LINHAS_USADAS = 3
C
C Numero de COLUNAS
C
        WRITE(*,*)'Numero de colunas? (3 a',HOST_COLUNAS,')'
        READ(*,*) COLUNAS_USADAS
        if (COLUNAS_USADAS .lt. 0) FIM = .true.
        if (COLUNAS_USADAS .gt. HOST_COLUNAS) COLUNAS_USADAS = HOST_COLUNAS
        if (COLUNAS_USADAS .lt. 3) COLUNAS_USADAS = 3
C
C PRECISAO
C
        WRITE(*,*)'Precisao entre 2 iteracoes consecutivas?'
        READ(*,*) PRECISAO
        if (PRECISAO .lt. 0) FIM = .true.
C
C Numero de PROCESSADORES
C
        WRITE(*,*)'Numero de processadores? (2 a',N_PROC,')'
        READ(*,*) PROC_USADOS
        if (PROC_USADOS .lt. 0) FIM = .true.
        if (PROC_USADOS .gt. N_PROC) PROC_USADOS = N_PROC
        if (PROC_USADOS .lt. 2) PROC_USADOS = 2
C
C Imprime matriz resposta no final (0=nao, <>0=sim)
C

        WRITE(*,*)'Imprimir matriz resposta no final ( <>0 => SIM)'
        READ(*,*) OUTPUT
        if (OUTPUT .lt. 0) FIM = .true.
        if (OUTPUT .ne. 0) OUTPUT = 1

        return
        end
```

```
      subroutine cond_iniciais(MATRIZ, LINHAS_USADAS, COLUNAS_USADAS)

C Comum ao programa principal
      integer HOST_LINHAS, HOST_COLUNAS
      parameter (HOST_LINHAS  = 400)
      parameter (HOST_COLUNAS = 400)

C Entrada
      integer LINHAS_USADAS, COLUNAS_USADAS

C Saida
      real*8 MATRIZ(HOST_LINHAS, HOST_COLUNAS)

C Interna
      LINHA, COLUNA

C parte interna
      do LINHA = 2, LINHAS_USADAS-1
        do COLUNA = 2, COLUNAS_USADAS-1
          MATRIZ(LINHA,COLUNA) = 0.0
        end do
      end do

C bordas esquerda e direita
      do LINHA = 1, LINHAS_USADAS
        MATRIZ(LINHA,1) = 100.0
        MATRIZ(LINHA,COLUNAS_USADAS) = 100.0
      end do

C bordas superior e inferior
      do COLUNA = 1, COLUNAS_USADAS
        MATRIZ(1,COLUNA) = 100.0
        MATRIZ(LINHAS_USADAS,COLUNA) = 100.0
      end do

      return
      end
```

```
      subroutine imprime(MATRIZ, LINHAS_USADAS, COLUNAS_USADAS)

C Comum ao programa principal
      integer HOST_LINHAS, HOST_COLUNAS
      parameter (HOST_LINHAS  = 400)
      parameter (HOST_COLUNAS = 400)


C Entrada
      integer LINHAS_USADAS, COLUNAS_USADAS
      real*8 MATRIZ(HOST_LINHAS, HOST_COLUNAS)

C Interna
      L, C

      if (COLUNAS_USADAS .le. 20) then
        do L = 1, LINHAS_USADAS
          WRITE(6,1) (MATRIZ(L,C), C = 1,COLUNAS_USADAS)
        end do
      end if

      if (COLUNAS_USADAS .gt. 20) then
        do L = 1, LINHAS_USADAS
          WRITE(6,*) 'Linha ',L
          do C = 1,COLUNAS_USADAS
            WRITE(6,2) C, MATRIZ(L,C)
          end do
        end do
      end if

      return

1     format(1X,<COLUNAS_USADAS>f4.0)
2     format(1X,i5,f4.0)

      end
```

```
      subroutine carga(PTR_INF,PTR_SUP,LINHAS_USADAS,PROC_USADAS)

C Comum ao programa principal
      integer N_PROC
      parameter (N_PROC = 10)

C Entrada
      integer LINHAS_USADAS, PROC_USADAS

C Saida
      integer PTR_INF(N_PROC), PTR_SUP(N_PROC)

C Internas
      integer QUOCIENTE, RESTO, PARA_POR, INC_INF, NO

      QUOCIENTE = LINHAS_USADAS/PROC_USADOS
      RESTO     = MOD(LINHAS_USADAS,PROC_USADOS)

      PTR_INF(1) = QUOCIENTE + 1
      if (RESTO .ne. 0) PTR_INF(1) = QUOCIENTE + 2

      PARA_POR = 0
      if (RESTO .gt. 2) PARA_POR = RESTO - 2

      do NO = 2, PROC_USADOS
        INC_INF = 0
        if (PARA_POR .gt. 0) then
          PARA_POR = PARA_POR - 1
          INC_INF = 1
        end if
        PTR_INF(NO) = PTR_INF(NO-1) + QUOCIENTE + INC_INF
        PTR_SUP(NO) = PTR_INF(NO-1) - 1
      end do
      PTR_SUP(1) = 1
      PTR_INF(PROC_USADOS) = LINHAS_USADAS

      return
      end
```

CALOR_H_N_NODE.for

Esta é a listagem do programa que roda nos processadores ACP .

Inclui:

- SORCHE sub-rotina principal, acessada pelo programa do hospedeiro
- função CONINI

```
      subroutine SORCHE
      implicit none

      integer HOSTLINHAS, HOSTCOLUNAS
      parameter (HOSTLINHAS  = 400)
      parameter (HOSTCOLUNAS = 400)

      integer NO_LINHAS, NO_COLUNAS
      parameter (NOLINHAS  = HOSTLINHAS/2+2)
      parameter (NOCOLUNAS = HOSTCOLUNAS)

C
C Variaveis para referenciar partes especificas das matrizes nos nos
C
      integer SUPERIOR, INFERIOR, INTERNO
      parameter (SUPERIOR = 1)
      parameter (INFERIOR = 2)
      parameter (INTERNO  = 3)

C
C Variaveis tipo COMMON para comunicaçao com o hospedeiro
C

      integer LINUSADAS, COLUSADAS, NDOPROC, POSICAO, PTRSUP, PTRINF
      common /PARAME/ LINUSADAS,
     &                COLUSADAS,
     &                NDOPROC,
     &                POSICAO,
     &                PTRSUP,
     &                PTRINF


      real*8 BORDA(2*NOCOLUNAS)
      common/BORDAS/BORDA


      real*8 MANT(NOLINHAS, NOCOLUNAS)
      common/MATRIZ/MANT


      real*8 EPSILON
      common/DIFERE/EPSILON


      integer ENE
      common/VARRE/ENE


C Variaveis locais
      integer L, C, NELEMENTOS, AUX

      real*8 ROJAC2,
     &       OMEGAENE,
     &       RESID,
     &       ATUAL(NOLINHAS,NOCOLUNAS)

C Funçao usada
      real*8 CONINI


C Constantes
      real*8 ZERO, MEIO, UM, DOIS, QUARTO, PI, QUATRO
      parameter (ZERO=0.0d0, MEIO=0.5d0, UM=1.0d0, DOIS= 2.0d0)
      parameter (QUARTO=0.25d0, PI=3.141592653589793d0, QUATRO=4.0d0)
```

```
C=====================================================================
C                      PONTO DE ENTRADA
C=====================================================================
C
C Calcula parametros de iniciaçao
C
      NELEMENTOS = PTRINF-PTRSUP+1
      ROJAC2 =((DCOS(PI/NELEMENTOS)+DCOS(PI/COLUSADAS))*MEIO)**DOIS

      if (ENE .eq. 1) then
        L = 1
        do 900 AUX = PTRSUP, PTRINF
          do 901 C = 1, COLUSADAS
            MANT(L,C) = CONINI(AUX, C, LINUSADAS, COLUSADAS)
901       continue
          L = L + 1
900     continue

        OMEGAENE = UM
        do 903 L = 2, NELEMENTOS-1
          do 904 C = 2, COLUSADAS-1
            ATUAL(L,C) = ZERO
904       continue
903     continue
      end if

C
C Despachador principal
C
      if (POSICAO .eq. SUPERIOR) GO TO 1
      if (POSICAO .eq. INTERNO)  GO TO 2
      if (POSICAO .eq. INFERIOR) GO TO 3

C=====================================================================
C ## 1 ## PROCESSADOR E BORDA SUPERIOR
C=====================================================================

C
C Coloca a borda INFERIOR, passada por outro processador
C
1     do 10 C = 1, COLUSADAS
        MANT(NELEMENTOS,C) = BORDA(C)
10    continue

C
C Coloca as condiçoes de contorno em ATUAL
C
      do 11 C = 1, COLUSADAS
        ATUAL(1,C) = MANT(1,C)
        ATUAL(NELEMENTOS,C) = MANT(NELEMENTOS,C)
11    continue
      do 12 L = 2, NELEMENTOS-1
        ATUAL(L,1) = MANT(L,1)
        ATUAL(L,COLUSADAS) = MANT(L,COLUSADAS)
12    continue

C
C Calcula, usando diferenças finitas, com MANT, gerando MATUAL
C
      do 13 C = 2, COLUSADAS-1
        do 14 L = 2, NELEMENTOS-1
          if (MOD(L+C,2) .eq. MOD(ENE,2)) then
            RESID = MANT(L-1,C)+MANT(L+1,C)+MANT(L,C-1) + MANT(L,C+1)-QUATRO * MANT(L,C)
            ATUAL(L,C) = ATUAL(L,C) + OMEGAENE*RESID*QUARTO
          end if
14      continue
13    continue
```

```
C
C Arruma o vetor de borda, a ser devolvido
C

        do 15 C = 1, COLUSADAS
          BORDA(C+NOCOLUNAS) = ATUAL(NELEMENTOS-1,C)
15      continue

C
C Vai calcular os parametros de saida
C

        GO TO 4


C=====================================================================
C ## 2 ## PROCESSADOR E INTERNO
C=====================================================================

C
C Arruma as bordas passadas pelos 2 processadores vizinhos
C

2       do 20 C = 1, COLUSADAS
          MANT(1,C)           = BORDA(C+NOCOLUNAS)
          MANT(NELEMENTOS,C) = BORDA(C)
20      continue

C
C Coloca as condiçoes de contorno em ATUAL
C
        do 21 C = 1, COLUSADAS
          ATUAL(1,C) = MANT(1,C)
          ATUAL(NELEMENTOS,C) = MANT(NELEMENTOS,C)
21      continue

        do 22 L = 2, NELEMENTOS-1
          ATUAL(L,1) = MANT(L,1)
          ATUAL(L,COLUSADAS) = MANT(L,COLUSADAS)
22      continue

C
C Calcula, usando diferenças finitas, com MANT, gerando MATUAL
C

        do 23 C = 2, COLUSADAS-1
          do 24 L = 2, NELEMENTOS-1
            if (MOD(L+C,2) .eq. MOD(ENE,2)) then
              RESID = MANT(L-1,C)+MANT(L+1,C)+MANT(L,C-1) + MANT(L,C+1)-QUATRO * MANT(L,C)
              ATUAL(L,C) = ATUAL(L,C) + OMEGAENE*RESID*QUARTO
            end if
24        continue
23      continue

C
C Arruma os vetores de borda, a serem devolvido
C

        do 25 C = 1, COLUSADAS
          BORDA(C)           = ATUAL(2,C)
          BORDA(C+NOCOLUNAS) = ATUAL(NELEMENTOS-1,C)
25      continue

C
C Vai calcular os parametros de saida
C

        GO TO 4
```

```
C=================================================================
C ## 3 ## PROCESSADOR E BORDA INFERIOR
C=================================================================

C
C Coloca a borda SUPERIOR, passada por outro processador
C

3       do 30 C = 1, COLUSADAS
          MANT(1,C) = BORDA(C+NOCOLUNAS)
30      continue

C
C Coloca as condiçoes de contorno em ATUAL
C
        do 31 C = 1, COLUSADAS
          ATUAL(1,C) = MANT(1,C)
          ATUAL(NELEMENTOS,C) = MANT(NELEMENTOS,C)
31      continue

        do 32 L = 2, NELEMENTOS-1
          ATUAL(L,1) = MANT(L,1)
          ATUAL(L,COLUSADAS) = MANT(L,COLUSADAS)
32      continue


C
C Calcula, usando diferenças finitas, com MANT, gerando MATUAL
C

        do 33 C = 2, COLUSADAS-1
          do 34 L = 2, NELEMENTOS-1
            if (MOD(L+C,2) .eq. MOD(ENE,2)) then
              RESID = MANT(L-1,C)+MANT(L+1,C)+MANT(L,C-1) + MANT(L,C+1)-QUATRO * MANT(L,C)
              ATUAL(L,C) = ATUAL(L,C) + OMEGAENE*RESID*QUARTO
            end if
34        continue
33      continue

C
C Arruma o vetor de borda, a ser devolvido
C

        do 35 C = 1, COLUSADAS
          BORDA(C) = ATUAL(2,C)
35      continue

C
C Vai calcular os parametros de saida
C

        GO TO 4

C=================================================================
C ## 4 ## CALCULA PARAMETROS DE SAIDA
C=================================================================
C
C Calcula valor de EPSILON
C

4       EPSILON = ZERO
        do 40 C = 2, COLUSADAS-1
          do 41 L = 2, NELEMENTOS-1
            EPSILON = EPSILON +   ABS( ATUAL(L,C)-MANT(L,C) )
41        continue
40      continue
```

```
C
C Poe matriz ATUAL na ANTERIOR, para funcionar na proxima iteraçao
C

      do 42 C = 1, COLUSADAS
        do 43 L = 1, NELEMENTOS
          MANT(L,C) = ATUAL(L,C)
43      continue
42    continue



C
C Arruma o valor de OMEGAENE, para funcionar na proxima iteraçao
C

      if (ENE .eq. 1) then
        OMEGAENE = UM / (UM - MEIO*ROJAC2)
      else
        OMEGAENE = UM / (UM - QUARTO * ROJAC2 * OMEGAENE)
      end if

      return
      end
```

```
      real*8 function CONINI(LINHA,COLUNA, MAXLIN,MAXCOL)

      integer LINHA, COLUNA, MAXLIN, MAXCOL

C
C tudo 0.0, menos nas bordas de cima, de baixo e da esquerda (100.0)
C

       if (LINHA .ne. 1) GO TO 1
       CONINI = 100.0
       GO TO 10

1      if (LINHA .ne. MAXLIN) GO TO 2
       CONINI = 100.0
       GO TO 10

2      if (COLUNA .ne. 1) GO TO 3
       CONINI = 100.0
       GO TO 10

3      if (COLUNA .ne. MAXCOL) GO TO 4
       CONINI = 0.0
       GO TO 10

4      CONINI = 0.0

10     return
       end
```

apêndice D

Listagem de CALOR (Nós-Nós)

NO_A_NO.UPF

Este arquivo contém as instruções para que os programas do computador hospedeiro e o dos nós processadores sejam compilados. É o arquivo tipo UPF

Usam-se 13 placas ACP do Centro Brasileiro de Pesquisas Físicas (CBPF): 1 como nó-mestre e 12 como nós-de-cálculo.

```
system = CBPF
host source = HOSPEDEIRO.for
node time = 3000

node blocks = 1:PARCIA,  2:USADOS, 3:PRECIS, 4:MAIEPS
node blocks = 5:EMLOOP,  6:FLAGS,  7:BORDAS, 8:MISCEL
node blocks = 9:EPSILO, 10:VARRE

class 1: nodes = 12 68020
class 1: node source = CALOR.for1
class 1: node main subroutine = CALOR

class 2: nodes = 1 68020
class 2: node source = MESTRE.for2
class 2: node main subroutine = MESTRE
```

HOSPEDEIRO.for

Esta é a listagem do programa que roda no computador-hospedeiro

Inclui:

- HOSPEDEIRO programa principal
- sub-rotina pede_dados
- função CONINI: estabelece as condições iniciais do domínio
- sub-rotina balanceamento_de_carga
- sub-rotina imprime

```
      program HOSPEDEIRO

      implicit none
      include 'ACP$AREA:USER_COMMON.INC'

C NUMERO MAXIMO DE PROCESSADORES ALOCADOS POR ESTE PROGRAMA
      integer PROC_ALOCADOS
      parameter (PROC_ALOCADOS = 13)

C DIMENSOES DA MAIOR MATRIZ POSSIVEL DE SER RESOLVIDA POR ESTE PROGRAMA
      integer HOST_LINHAS, HOST_COLUNAS
      parameter (HOST_LINHAS  = 400)
      parameter (HOST_COLUNAS = 400)

C DIMENSOES DA MAIOR MATRIZ PARCIAL QUE PODE SER RESOLVIDA POR UM NO
      integer NO_LINHAS, NO_COLUNAS
      parameter (NO_LINHAS  = HOST_LINHAS/2+2)
      parameter (NO_COLUNAS = HOST_COLUNAS)

C=====================
C VARIAVEIS TIPO COMMON
C=====================
C PARA COMUNICACAO ENTRE HOST E NO-MESTRE
      integer   PARCIA,   USADOS,   PRECIS,   MAIEPS,   EMLOOP
      integer S_PARCIA, S_USADOS, S_PRECIS, S_MAIEPS, S_EMLOOP
      parameter (PARCIA = 1) ! matriz PARCIAl com o resultado de 1 no
      parameter (USADOS = 2) !dados fornecidos pelo usuario
      parameter (PRECIS = 3) !PRECISao fornecida pelo usuario
      parameter (MAIEPS = 4) !MAIor EPSilon calculado pelos nos-de-calculo
      parameter (EMLOOP = 5) !poe nos-de-calculo EM LOOP


      real*8 PARCIAL(NO_LINHAS, NO_COLUNAS)
      parameter (S_PARCIA = NO_LINHAS*NO_COLUNAS* 2) ! S=tamanho em palavras de 32 bits
      COMMON /PARCIA/ PARCIAL


      integer LINHAS_USADAS,
     &        COLUNAS_USADAS,
     &        PROC_USADOS,
     &        ENDERECOS_MAIEPS(PROC_ALOCADOS),
     &        ENDERECOS_EMLOOP(PROC_ALOCADOS),
     &        ENDERECO_FLAGS,
     &        ENDERECOS_BORDAS(PROC_ALOCADOS),
     &        ENDERECOS_MISCEL(PROC_ALOCADOS),
     &        ENDERECOS_EPSILO(PROC_ALOCADOS),
     &        ENDERECOS_VARRE(PROC_ALOCADOS)
      parameter (S_USADOS = (4+6*PROC_ALOCADOS)* 1)
      COMMON /USADOS/ LINHAS_USADAS,
     &                COLUNAS_USADAS,
     &                PROC_USADOS,
     &                ENDERECOS_MAIEPS,
     &                ENDERECOS_EMLOOP,
     &                ENDERECO_FLAGS,
     &                ENDERECOS_BORDAS,
     &                ENDERECOS_MISCEL,
     &                ENDERECOS_EPSILO,
     &                ENDERECOS_VARRE
      real*8 PRECISAO
      parameter (S_PRECIS = 1* 2)
      COMMON /PRECIS/ PRECISAO


      real*8 MAIOR_EPSILON
      parameter (S_MAIEPS = 1* 2)
      COMMON /MAIEPS/ MAIOR_EPSILON
```

```
      integer FICA_EM_LOOP, NENHUM_LOOP, LOOP_INICIAL
      parameter (S_EMLOOP = 1* 1)
      COMMON /EMLOOP/ FICA_EM_LOOP
      parameter (NENHUM_LOOP  = 0)
      parameter (LOOP_INICIAL = 1)


C PARA COMUNICACAO ENTRE NO-MESTRE E NOS-DE-CALCULO
      integer  FLAGS,    BORDAS,   MISCEL,   EPSILO,   VARRE
      integer S_FLAGS, S_BORDAS, S_MISCEL, S_EPSILO, S_VARRE
      parameter (FLAGS   = 6)
      parameter (BORDAS  = 7
      parameter (MISCEL  = 8)
      parameter (EPSILO  = 9)
      parameter (VARRE   = 10)


      integer FLAGS_FIM_DE_CALCULO(PROC_ALOCADOS),
     &        FIM_DE_CALCULO, NAO_FIM_DE_CALCULO, JA_RECEBEU
      parameter (S_FLAGS  = PROC_ALOCADOS* 1)
      COMMON /FLAGS/ FLAGS_FIM_DE_CALCULO
      parameter (FIM_DE_CALCULO = 0)
      parameter (NAO_FIM_DE_CALCULO = 1)
      parameter (JA_RECEBEU = 2)


      real*8 DUAS_BORDAS(2*NO_COLUNAS)
      parameter (S_BORDAS = 2*NO_COLUNAS* 2)
      COMMON /BORDAS/ DUAS_BORDAS


      integer POSICAO,
     &        PTRSUP,
     &        PTRINF,
     &        ESTENO,
     &        SUPERIOR, INTERNO, INFERIOR
      parameter (S_MISCEL = 4* 1)
      COMMON /MISCEL/ POSICAO,
     &                PTRSUP,
     &                PTRINF,
     &                ESTENO
      parameter (SUPERIOR = 1)
      parameter (INTERNO  = 2)
      parameter (INFERIOR = 3)


      real*8 EPS_DO_NO
      parameter (S_EPSILO = 1* 2)
      COMMON /EPSILO/ EPS_DO_NO


      integer ENE
      parameter (S_VARRE = 1* 1)
      COMMON /VARRE/ENE



C=====================================================
C VARIAVEIS E CONSTANTES LOCAIS AO PROGRAMA DO HOSPEDEIRO
C=====================================================

      real*8 RESPOSTA(HOST_LINHAS, HOST_COLUNAS)
      real*8 CONINI

      logical  SEND_DONE, GET_DONE
      logical QUER_FIM, TEM_OUTPUT
```

```
      integer CLASS_M, CLASS_C
      integer SIZE
      integer NO_MESTRE
      integer NO, LINHA, COLUNA
      integer VETOR_USADOS(S_USADOS)
      integer PTR_INFERIOR(PROC_ALOCADOS),
     &        PTR_SUPERIOR(PROC_ALOCADOS),
     &        AUXILIAR_INF, AUXILIAR_SUP
      integer OFFSET


      real*4 TEMPO



C=====================================================================
C PONTO DE ENTRADA PRINCIPAL
C=====================================================================

      call ACP_INIT
      CLASS_C = 1
      CLASS_M = 2

      NO_MESTRE = PROC_ALOCADOS


C=====================================================================
C PONTO DE ENTRADA SECUNDARIO
C=====================================================================
1      call pede_dados(LINHAS_USADAS,COLUNAS_USADAS,PROC_USADOS,PRECISAO,TEM_OUTPUT,
     &                QUER_FIM)


C-----------------------------------------------
C Termina o programa se uma das condiçoes foi negativa
C-----------------------------------------------
      if (QUER_FIM) go to 2

      write(6,*)' '
      write(6,*)'Linhas usadas =',LINHAS_USADAS
      write(6,*)'Colunas usadas =',COLUNAS_USADAS
      write(6,*)'Processadores de calculo usados =',PROC_USADOS
      write(6,*)'Precisao entre 2 iteraçoes consecutivas =',PRECISAO


C--------------------------
C Inicia variaveis deste run.
C--------------------------
      TEMPO = secnds(0.0)



C-------------------------------------------------------------------------------
C Faz o balanceamento estatico de carga, a ser usado quando for ler os resultados dos nos.
C-------------------------------------------------------------------------------
      call balanceamento_de_carga(PTR_INFERIOR,PTR_SUPERIOR, LINHAS_USADAS,PROC_USADOS)
```

```
C------------------
C Monta VETOR_USADOS
C------------------
      VETOR_USADOS(1) = LINHAS_USADAS
      VETOR_USADOS(2) = COLUNAS_USADAS
      VETOR_USADOS(3) = PROC_USADOS
      OFFSET = 3


      do NO = 1, PROC_ALOCADOS
        VETOR_USADOS(OFFSET+NO) = acp_node_VME_address(NO) + acp_node_block_offset(CLASS_M,MAIEPS)
      end do
      OFFSET = OFFSET + PROC_ALOCADOS

      do NO = 1, PROC_ALOCADOS
        VETOR_USADOS(OFFSET+NO) = acp_node_VME_address(NO) + acp_node_block_offset(CLASS_C,EMLOOP)
      end do
      OFFSET = OFFSET + PROC_ALOCADOS

      VETOR_USADOS(OFFSET+1) = acp_node_VME_address(NO_MESTRE) + acp_node_block_offset(CLASS_M,FLAGS)
      OFFSET = OFFSET + 1

      do NO = 1, PROC_ALOCADOS
        VETOR_USADOS(OFFSET+NO) = acp_node_VME_address(NO) + acp_node_block_offset(CLASS_C,BORDAS)
      end do
      OFFSET = OFFSET + PROC_ALOCADOS

      do NO = 1, PROC_ALOCADOS
        VETOR_USADOS(OFFSET+NO) = acp_node_VME_address(NO) + acp_node_block_offset(CLASS_C,MISCEL)
      end do
      OFFSET = OFFSET + PROC_ALOCADOS
      do NO = 1, PROC_ALOCADOS
        VETOR_USADOS(OFFSET+NO) = acp_node_VME_address(NO) + acp_node_block_offset(CLASS_C,EPSILO)
      end do
      OFFSET = OFFSET + PROC_ALOCADOS


      do NO = 1, PROC_ALOCADOS
        VETOR_USADOS(OFFSET+NO) = acp_node_VME_address(NO) + acp_node_block_offset(CLASS_C,VARRE)
      end do
      OFFSET = OFFSET + PROC_ALOCADOS



C-------------------------------------------------------------
C Envia dados importantes para os nos de calculo e para o mestre
C-------------------------------------------------------------
      SIZE = S_USADOS
      call acp_convert_h_n(VETOR_USADOS, VETOR_USADOS, SIZE, ACP_I_4, CLASS_C)

C para os nos de calculo
      do NO = 1, PROC_USADOS
        SEND_DONE = .false.
        do while (.not.SEND_DONE)
          call acp_sendblock(USADOS, VETOR_USADOS, SIZE, NO, CLASS_C, ACP$NO_GO,SEND_DONE)
        end do
      end do
      call acp_convert_n_h(VETOR_USADOS, VETOR_USADOS, SIZE, ACP_I_4, CLASS_C)
      call acp_convert_h_n(VETOR_USADOS, VETOR_USADOS, SIZE, ACP_I_4, CLASS_M)
C ... e para o no mestre
      SEND_DONE = .false.
      do while (.not.SEND_DONE)
        call acp_sendblock(USADOS,VETOR_USADOS,SIZE,NO_MESTRE,CLASS_CM,ACP$NO_GO, SEND_DONE)
      end do
```

```
C-------------------------------------------------------------------------------
C Prepara os nos de calculo de tal forma que seus programas nao sejam desativados (ENE = 1)
C Poe todos os nos-de-calculo em loop e os starta
C Arruma variaveis do mestre e o starta
C-------------------------------------------------------------------------------

C avisa para os nos de calculo nao destivarem seus programas
      SIZE = S_VARRE
      ENE = 1 ! Nao desativar os nos de calculo
      call acp_convert_h_n(ENE, ENE, SIZE, ACP_I_4, CLASS_C)
      do NO = 1, PROC_USADOS
        SEND_DONE = .false.
        do while (.not.SEND_DONE)
          call acp_sendblock(VARRE, ENE, SIZE, NO, CLASS_C, ACP$NO_GO, SEND_DONE)
        end do
      end do


C Poe todos os nos-de-calculo em loop e os starta
      SIZE = S_EMLOOP
      FICA_EM_LOOP = LOOP_INICIAL
      call acp_convert_h_n(FICA_EM_LOOP, FICA_EM_LOOP, SIZE, ACP_I_4, CLASS_C)
      do NO = 1, PROC_USADOS
        SEND_DONE = .false.
        do while (.not.SEND_DONE)
          call acp_sendblock(EMLOOP, FICA_EM_LOOP, SIZE, NO, CLASS_C, ACP$GO, SEND_DONE)
        end do
      end do


C Starta o mestre (aproveitando, manda PRECISAO)
      SIZE = S_PRECIS
      call acp_convert_h_n(PRECISAO, PRECISAO, SIZE, ACP_R_8, CLASS_M)
      SEND_DONE = .false.
      do while (.not.SEND_DONE)
        call acp_sendblock(PRECIS, PRECISAO, SIZE, NO_MESTRE, CLASS_M, ACP$GO, SEND_DONE)
      end do


C-------------------------------------
C Arruma as condiçoes iniciais do dominio.
C-------------------------------------
      do LINHA = 1, LINHAS_USADAS
        do COLUNA = 1, COLUNAS_USADAS
          RESPOSTA(LINHA,COLUNA) = CONINI(LINHA,COLUNA, LINHAS_USADAS,COLUNAS_USADAS)
        end do
      end do



C-------------------------------------------------------------------------------
C Fica esperando que o no-mestre sinalize que os processadores-de-calculo ja encontraram a soluçao
C Recolhe MAIOR_EPSILON do no mestre
C Recolhe matrizes parciais de todos os PROC_USADOS nos de calculo e os libera
C-------------------------------------------------------------------------------

C Recolhe MAIOR_EPSILON do no-mestre. E o aviso de que a resposta do problema foi encontrada
      SIZE = S_MAIEPS
      GET_DONE = .false.
      do while (.not.GET_DONE)
        call acp_getblock(MAIEPS,MAIOR_EPSILON,SIZE, NO_MESTRE,CLASS_M, ACP$LAST_BLOCK,
     &                    GET_DONE)
      end do
      call acp_convert_n_h(MAIOR_EPSILON, MAIOR_EPSILON, SIZE, ACP_R_8, CLASS_M)
```

```
C Recolhe matrizes parciais de todos os PROC_USADOS nos de calculo e os libera
      SIZE = S_PARCIA
      do NO = 1, PROC_USADOS
        LINHA = 2
        AUXILIAR_INF = PTR_INFERIOR(NO) - 1
        AUXILIAR_SUP = PTR_SUPERIOR(NO) + 1

        GET_DONE = .false.
        do while (.not.GET_DONE)
          call acp_getblock(PARCIA,PARCIAL,SIZE, NO,CLASS_C, ACP$LAST_BLOCK, GET_DONE)
        end do
        call acp_convert_n_h(PARCIAL, PARCIAL, SIZE, ACP_R_8, CLASS_C)

        do while (AUXILIAR_SUP .le. AUXILIAR_INF)
          do COLUNA = 1, COLUNAS_USADAS
            RESPOSTA(AUXILIAR_SUP,COLUNA) = PARCIAL(LINHA,COLUNA)
          end do
          AUXILIAR_SUP = AUXILIAR_SUP + 1
          LINHA = LINHA + 1
        end do
        call acp_declare_node_ready(NO)
      end do


C------------
C Finalizaçao
C------------
       TEMPO = secnds(TEMPO)
       if (TEM_OUTPUT) call imprime(RESPOSTA, LINHAS_USADAS, COLUNAS_USADAS)

       write(6,*)'----------------------'
       write(6,*)'Tempo total de execuçao =',TEMPO,'segundos'
       write(6,*)'----------------------'

C-------------------------------------------------------------------------
C Volta para rodar de novo, pois nenhum dos dados do usuario foi negativo.
C-------------------------------------------------------------------------
       go to 1


C-------------------------------------------------------------
C Termina o programa, pois um dos dados do usuario foi negativo.
C-------------------------------------------------------------
2      call ACP_exit
       stop 'Programa de J.RODOLFO F.XAVIER - IFT - S.PAULO'
       end
```

```
      subroutine pede_dados(LINHAS_USADAS, COLUNAS_USADAS, PROC_USADOS, PRECISAO, TEM_OUTPUT, QUER_FIM)

C
C Comuns ao programa principal
C
      integer PROC_ALOCADOS
      parameter (PROC_ALOCADOS = 13)

      integer HOST_LINHAS, HOST_COLUNAS
      parameter (HOST_LINHAS = 400)
      parameter (HOST_COLUNAS = 400)

C Saida
      integer LINHAS_USADAS, COLUNAS_USADAS, PROC_USADOS
      real*8  PRECISAO
      logical QUER_FIM, TEM_OUTPUT

C Interna
      integer PROC_CALCULO, AUX_IMPRIME

C
C Ponto de entrada
C
      write(*,*)'-----------------------------------------------'
      write(*,*)'Para parar o programa, entre com pelo menos 1 das'
      write(*,*)' condiçoes abaixo como sendo negativa'
      write(*,*)' '

C
C Qualquer numero negativo implica fim do programa
C
      QUER_FIM = .false.

C
C Entra com o numero de linhas
C
      write(*,*)'Numero de linhas? (3 a',HOST_LINHAS,')'
      read(*,*) LINHAS_USADAS
      if (LINHAS_USADAS .lt. 0) QUER_FIM = .true.
      if (LINHAS_USADAS .lt. 3)           LINHAS_USADAS = 3
      if (LINHAS_USADAS .gt. HOST_LINHAS) LINHAS_USADAS = HOST_LINHAS

C
C Entra com o numero de colunas
C
      write(*,*)'Numero de colunas? (3 a',HOST_COLUNAS,')'
      read(*,*) COLUNAS_USADAS
      if (COLUNAS_USADAS .lt. 0) QUER_FIM = .true.
      if (COLUNAS_USADAS .lt. 3)           COLUNAS_USADAS = 3
      if (COLUNAS_USADAS .gt. HOST_COLUNAS) COLUNAS_USADAS = HOST_COLUNAS

C
C Reserva 1 processador para ser o mestre
C
      PROC_CALCULO = PROC_ALOCADOS - 1


C
C Entra com o numero de processadores para calculo
C
      write(*,*)'Processadores de calculo? (2 a', PROC_CALCULO,')'
      read(*,*) PROC_USADOS
      if (PROC_USADOS .lt. 0) QUER_FIM = .true.
      if (PROC_USADOS .lt. 2)           PROC_USADOS = 2
      if (PROC_USADOS .gt. PROC_CALCULO) PROC_USADOS = PROC_CALCULO
```

```
C
C Entra com a precisao para parada
C
      write(*,*)'Precisao entre 2 iteraçoes consecutivas?'
      read(*,*) PRECISAO
      if (PRECISAO .lt. 0.0) QUER_FIM = .true.

C
C Usuario vai querer imprimir a matriz resposta no fim do programa?
C
      TEM_OUTPUT = .false.
      write(*,*)'Deseja imprimir a matriz-resposta (<>0 -> sim)?'
      read(*,*) AUX_IMPRIME
      if (AUX_IMPRIME .lt. 0) QUER_FIM = .true.
      if (AUX_IMPRIME .ne. 0) TEM_OUTPUT = .true.

      return
      end
```

```
      real*8 function CONINI(L,C, LINHAS_USADAS,COLUNAS_USADAS)

C Entrada
      integer L,C, LINHAS_USADAS,COLUNAS_USADAS

      CONINI = 0.0
      if (L .eq. 1)              CONINI = 100.0
      if (L .eq. LINHAS_USADAS)  CONINI = 100.0
      if (C .eq. 1)              CONINI = 100.0
      if (C .eq. COLUNAS_USADAS) CONINI =   0.0

      return
      end
```

```
      subroutine balanceamento_de_carga(PTR_INFERIOR,PTR_SUPERIOR, LINHAS_USADAS,PROC_USADOS)

C
C Comuns ao programa principal
C
      integer PROC_ALOCADOS
      parameter (PROC_ALOCADOS = 13)

C Entrada
      integer LINHAS_USADAS, PROC_USADOS

C Saida
      integer PTR_INFERIOR(PROC_ALOCADOS), PTR_SUPERIOR(PROC_ALOCADOS)

C Interna
      integer QUOCIENTE, RESTO, PARA_POR, INC_INF


      QUOCIENTE = LINHAS_USADAS/PROC_USADOS
      RESTO     = mod(LINHAS_USADAS,PROC_USADOS)

      PTR_INFERIOR(1) = QUOCIENTE + 1
      if (RESTO .ne. 0) PTR_INFERIOR(1) = QUOCIENTE + 2

      PARA_POR = 0
      if (RESTO .gt. 2) PARA_POR = RESTO - 2

      do NO = 2, PROC_USADOS
        INC_INF = 0
        if (PARA_POR .gt. 0) then
          PARA_POR = PARA_POR - 1
          INC_INF = 1
        end if
        PTR_INFERIOR(NO) = PTR_INFERIOR(NO-1) + QUOCIENTE + INC_INF
        PTR_SUPERIOR(NO) = PTR_INFERIOR(NO-1) - 1
      end do
      PTR_SUPERIOR(1) = 1
      PTR_INFERIOR(PROC_USADOS) = LINHAS_USADAS

      return
      end
```

```
      subroutine imprime_resposta(RESPOSTA, LINHAS_USADAS, COLUNAS_USADAS)

      integer HOST_LINHAS, HOST_COLUNAS
      parameter (HOST_LINHAS = 400)
      parameter (HOST_COLUNAS = 400)

      integer LINHAS_USADAS, COLUNAS_USADAS
      real*8 RESPOSTA(HOST_LINHAS,HOST_COLUNAS)


      write(6,*)'--------------------'
      write(6,*)'- Matriz Resposta -'
      write(6,*)'--------------------'
      if (COLUNAS_USADAS .le. 20) then
        do LINHA = 1, LINHAS_USADAS
          write(6,1) (RESPOSTA(LINHA,COLUNA), COLUNA = 1,COLUNAS_USADAS)
        end do
      end if
      if (COLUNAS_USADAS .gt. 20) then
        do LINHA = 1, LINHAS_USADAS
          write(6,*)'Linha',LINHA
          do COLUNA = 1, COLUNAS_USADAS
            write(6,2) COLUNA, RESPOSTA(LINHA,COLUNA)
          end do
        end do
      end if

      return
1     format(1X,<COLUNAS_USADAS>f4.0)
2     format(1X,I5,f4.0)
      end
```

MESTRE.for2

Esta é a listagem do programa que roda no computador-mestre .

Inclui:

- sub-rotina MESTRE: sub-rotina principal
- sub-rotina balanceamento_de_carga
- função CONINI: estabelece as condições iniciais do domínio

```
      subroutine MESTRE

      implicit none


C NUMERO MAXIMO DE PROCESSADORES ALOCADOS POR ESTE PROGRAMA
      integer PROC_ALOCADOS
      parameter (PROC_ALOCADOS = 13)

C DIMENSOES DA MAIOR MATRIZ POSSIVEL DE SER RESOLVIDA POR ESTE PROGRAMA
      integer HOST_LINHAS, HOST_COLUNAS
      parameter (HOST_LINHAS  = 400)
      parameter (HOST_COLUNAS = 400)

C DIMENSOES DA MAIOR MATRIZ PARCIAL QUE PODE SER RESOLVIDA POR UM NO
      integer NO_LINHAS, NO_COLUNAS
      parameter (NO_LINHAS  = HOST_LINHAS/2+2)
      parameter (NO_COLUNAS = HOST_COLUNAS)

C=====================
C VARIAVEIS TIPO COMMON
C=====================
C PARA COMUNICACAO ENTRE HOST E NO-MESTRE
      integer   PARCIA,   USADOS,   PRECIS,   MAIEPS,   EMLOOP
      integer S_PARCIA, S_USADOS, S_PRECIS, S_MAIEPS, S_EMLOOP
      parameter (PARCIA = 1) ! matriz PARCIAl com o resultado de 1 no
      parameter (USADOS = 2) !dados fornecidos pelo usuario
      parameter (PRECIS = 3) !PRECISao fornecida pelo usuario
      parameter (MAIEPS = 4) !MAIor EPSilon calculado pelos nos-de-calculo
      parameter (EMLOOP = 5) !poe nos-de-calculo EM LOOP


      real*8 PARCIAL(NO_LINHAS, NO_COLUNAS)
      parameter (S_PARCIA = NO_LINHAS*NO_COLUNAS* 2) ! S=tamanho em palavras de 32 bits
      COMMON /PARCIA/ PARCIAL


      integer LINHAS_USADAS,
     &        COLUNAS_USADAS,
     &        PROC_USADOS,
     &        ENDERECOS_MAIEPS(PROC_ALOCADOS),
     &        ENDERECOS_EMLOOP(PROC_ALOCADOS),
     &        ENDERECO_FLAGS,
     &        ENDERECOS_BORDAS(PROC_ALOCADOS),
     &        ENDERECOS_MISCEL(PROC_ALOCADOS),
     &        ENDERECOS_EPSILO(PROC_ALOCADOS),
     &        ENDERECOS_VARRE(PROC_ALOCADOS)
      parameter (S_USADOS = (4+6*PROC_ALOCADOS)* 1)
      COMMON /USADOS/ LINHAS_USADAS,
     &                COLUNAS_USADAS,
     &                PROC_USADOS,
     &                ENDERECOS_MAIEPS,
     &                ENDERECOS_EMLOOP,
     &                ENDERECO_FLAGS,
     &                ENDERECOS_BORDAS,
     &                ENDERECOS_MISCEL,
     &                ENDERECOS_EPSILO,
     &                ENDERECOS_VARRE
      real*8 PRECISAO
      parameter (S_PRECIS = 1* 2)
      COMMON /PRECIS/ PRECISAO


      real*8 MAIOR_EPSILON
      parameter (S_MAIEPS = 1* 2)
      COMMON /MAIEPS/ MAIOR_EPSILON
```

```
      integer FICA_EM_LOOP, NENHUM_LOOP, LOOP_INICIAL
      parameter (S_EMLOOP = 1* 1)
      COMMON /EMLOOP/ FICA_EM_LOOP
      parameter (NENHUM_LOOP  = 0)
      parameter (LOOP_INICIAL = 1)


C PARA COMUNICACAO ENTRE NO-MESTRE E NOS-DE-CALCULO
      integer  FLAGS,    BORDAS,   MISCEL,   EPSILO,   VARRE
      integer S_FLAGS, S_BORDAS, S_MISCEL, S_EPSILO, S_VARRE
      parameter (FLAGS   = 6)
      parameter (BORDAS  = 7
      parameter (MISCEL  = 8)
      parameter (EPSILO  = 9)
      parameter (VARRE   = 10)


      integer FLAGS_FIM_DE_CALCULO(PROC_ALOCADOS),
     &        FIM_DE_CALCULO, NAO_FIM_DE_CALCULO, JA_RECEBEU
      parameter (S_FLAGS  = PROC_ALOCADOS* 1)
      COMMON /FLAGS/ FLAGS_FIM_DE_CALCULO
      parameter (FIM_DE_CALCULO = 0)
      parameter (NAO_FIM_DE_CALCULO = 1)
      parameter (JA_RECEBEU = 2)


      real*8 DUAS_BORDAS(2*NO_COLUNAS)
      parameter (S_BORDAS = 2*NO_COLUNAS* 2)
      COMMON /BORDAS/ DUAS_BORDAS


      integer POSICAO,
     &        PTRSUP,
     &        PTRINF,
     &        ESTENO,
     &        SUPERIOR, INTERNO, INFERIOR
      parameter (S_MISCEL = 4* 1)
      COMMON /MISCEL/ POSICAO,
     &                PTRSUP,
     &                PTRINF,
     &                ESTENO
      parameter (SUPERIOR = 1)
      parameter (INTERNO  = 2)
      parameter (INFERIOR = 3)


      real*8 EPS_DO_NO
      parameter (S_EPSILO = 1* 2)
      COMMON /EPSILO/ EPS_DO_NO


      integer ENE
      parameter (S_VARRE = 1* 1)
      COMMON /VARRE/ENE


C
C VARIAVEIS LOCAIS AO PROCESSADOR MESTRE
C

       integer VETORMISCEL(SMISCEL)
       integer I, J, C, AUXSUP, AUXINF, NO
       integer FALTALER
       integer     AUXEPSDONO(SEPSILO)
       equivalence (AUXEPSDONO, EPSDONO)

       integer     AUXDUASBORDAS(SBORDAS)
       equivalence (AUXDUASBORDAS, DUASBORDAS)
```

```
      integer PTRINFERIOR(PROCALOCADOS), PTRSUPERIOR(PROCALOCADOS)
      integer SIZE, ENDERECO

      real*8 MATRIZDEBORDAS(SBORDAS/2, PROCALOCADOS)
      real*8 CONINI

      logical PARA

C=========================
C PONTO DE ENTRADA PRINCIPAL
C=========================

C Manda parametros importantes para todos os processadores

      call balanceamentodecarga(PTRINFERIOR,PTRSUPERIOR, LINHASUSADAS,PROCUSADOS)

      SIZE = SMISCEL
      do 1 NO = 1, PROCUSADOS
        VETORMISCEL(1) = INTERNO
        if (NO .eq. 1)           VETORMISCEL(1) = SUPERIOR
        if (NO .eq. PROCUSADOS)  VETORMISCEL(1) = INFERIOR
        VETORMISCEL(2) = PTRSUPERIOR(NO)
        VETORMISCEL(3) = PTRINFERIOR(NO)
        VETORMISCEL(4) = NO

        ENDERECO = ENDERECOSMISCEL(NO)
        call acpsup
        do 2 I = 1, SIZE
          long(ENDERECO) = VETORMISCEL(I)
          ENDERECO = ENDERECO + 4
2       continue
        call acpusr
1     continue

C----------------------
C iniciar MATRIZDEBORDAS
C----------------------

      do 3 NO = 1, PROC_USADOS
        AUXINF = PTRINF(NO)
        AUXSUP = PTRSUP(NO)
        do 4 C = 1, COLUNASUSADAS
          DUASBORDAS(C,NO)          = CONINI(AUXINF,C, LINHASUSADAS,COLUNASUSADAS)
          DUASBORDAS(C+NOCOLUNA,NO) = CONINI(AUXSUP,C, LINHASUSADAS,COLUNASUSADAS)
4       continue
3     continue


C=============
C LOOP PRINCIPAL
C=============

      PARA = .FALSE.
      ENE = 1
1111  if (PARA) go to 9999

C----------------------
C Reseta matriz de flags.
C----------------------

     do 5 NO = 1, PROCUSADOS
       FLAGSFIMDECALCULO(NO) = NAOFIMDECALCULO
5    continue
```

```
C-------------------------------------------------------
C Distribua as bordas para os processadores e os starta.
C-------------------------------------------------------
      do 6 NO = 1, PROCUSADOS
        do 7 C = 1, 2* NOCOLUNAS
          DUASBORDAS(C) = MATRIZDEBORDAS(C,NO)
7       continue
        ENDERECO = ENDERECOSBORDAS(NO)
        call acpsup
        do 8 I = 1, 2* NOCOLUNAS
          long(ENDERECO)   = AUXDUASBORDAS(2*I-1)
          long(ENDERECO+4) = AUXDUASBORDAS(2*I)
          ENDERECO = ENDERECO + 8
8       continue
        call acpusr
C Envia o valor de ENE
        ENDERECO = ENDERECOSVARRE(NO)
        call acpsup
        long(ENDERECO) = ENE
        call acpusr
C Faca o processador NO calcular sua parte
        ENDERECO = ENDERECOSEMLOOP(NO)
        call acpsup
        long(ENDERECO) = NENHUMLOOP
        call acpusr
6     continue

C-------------------------------------------------------------------------------------
C Comeca a prestar atencao para ver se os processadores de calculo ja resolveram o problema
C-------------------------------------------------------------------------------------
      MAIOREPSILON = 0.0
      FALTALER = PROCUSADOS
2222  if (FALTALER .eq. 0) go to 3333
      do 9 NO = 1, PROCUSADOS
        if (FLAGSFIMDECALCULO(NO) .ne. JARECEBEU) then
          if (FLAGSFIMDECALCULO(NO) .eq. FIMDECALCULO) then
            FLAGSFIMDECALCULO(NO) = JARECEBEU
            ENDERECO = ENDERECOSEPSILO(NO)
            call acpsup
            AUXEPSDONO(1) = long(ENDERECO)
            AUXEPSDONO(2) = long(ENDERECO+4)
            call acpusr
            MAIOREPSILON = MAIOREPSILON +    EPSDONO
            ENDERECO = ENDERECOSBORDAS(NO)
            call acpsup
            do 10 C = 1, 2* NOCOLUNAS
              AUXDUASBORDAS(2*C-1) = long(ENDERECO)
              AUXDUASBORDAS(2*C)   = long(ENDERECO+4)
              ENDERECO = ENDERECO + 8
10          continue
            call acpusr

            if (NO .ne. 1) then
              do 11 C = 1, NOCOLUNAS
                MATRIZDEBORDAS(C,NO-1) = DUASBORDAS(C)
11            continue
            end if
            if (NO .ne. PROCUSADOS) then
              do 12 C = NOCOLUNAS+1, 2* NOCOLUNAS
                MATRIZDEBORDAS(C,NO+1) = DUASBORDAS(C)
12            continue
            end if
            FALTALER = FALTALER - 1
          end if
        end if
9     continue
      go to 2222
3333  ENE = ENE + 1
```

```
C-------------------------------------------------------------------------------
C Verifica se ja pode parar. Se puder avisa os nos de calculo para se dasativarem
C (mandando ENE = 0)
C-------------------------------------------------------------------------------
      if (MAIOREPSILON .le. PRECISAO) then
        ENE = 0
        do 13 NO = 1, PROC_USADOS
          ENDERECO = ENDERECOSVARRE(NO)
          call acpsup
          long(ENDERECO) = ENE
          call acpusr
13      continue
        PARA = .TRUE.
      end if
      go to 1111



C---------------
C FIM DO PROGRAMA
C---------------
9999  return
      end
```

```
      subroutine balanceamentodecarga(PTRINFERIOR,PTRSUPERIOR, LINHASUSADAS,PROCUSADOS)

C
C Comuns ao programa principal
C
      integer PROCALOCADOS
      parameter (PROCALOCADOS = 10)

C Entrada
      integer LINHASUSADAS, PROCUSADOS

C Saida
      integer PTRINFERIOR(PROCALOCADOS), PTRSUPERIOR(PROCALOCADOS)

C Interna
      integer QUOCIENTE, RESTO, PARAPOR, INCINF


      QUOCIENTE = LINHASUSADAS/PROCUSADOS
      RESTO     = mod(LINHASUSADAS,PROCUSADOS)

      PTRINFERIOR(1) = QUOCIENTE + 1
      if (RESTO .ne. 0) PTRINFERIOR(1) = QUOCIENTE + 2

      PARAPOR = 0
      if (RESTO .gt. 2) PARAPOR = RESTO - 2

      do NO = 2, PROCUSADOS
        INCINF = 0
        if (PARAPOR .gt. 0) then
          PARAPOR = PARAPOR - 1
          INCINF = 1
        end if
        PTRINFERIOR(NO) = PTRINFERIOR(NO-1) + QUOCIENTE + INCINF
        PTRSUPERIOR(NO) = PTRINFERIOR(NO-1) - 1
      end do
      PTRSUPERIOR(1) = 1
      PTRINFERIOR(PROCUSADOS) = LINHASUSADAS

      return
      end
```

```
      real*8 function CONINI(L,C,  MAXLINHAS,MAXCOLUNAS)

C Entrada
      integer L,C, MAXLINHAS,MAXCOLUNAS

      CONINI = 0.0
      if (L .eq. 1)          CONINI = 100.0
      if (L .eq. MAXLINHAS)  CONINI = 100.0
      if (C .eq. 1)          CONINI = 100.0
      if (C .eq. MAXCOLUNAS) CONINI =   0.0

      return
      end
```

CALOR.for1

Esta é a listagem do programa que roda nos nós-de-cálculo .

Inclui:

- sub-rotina CALOR: sub-rotina principal
- função CONINI: estabelece as condições iniciais do domínio

```
      subroutine CALOR

      implicit none


C NUMERO MAXIMO DE PROCESSADORES ALOCADOS POR ESTE PROGRAMA
      integer PROC_ALOCADOS
      parameter (PROC_ALOCADOS = 13)

C DIMENSOES DA MAIOR MATRIZ POSSIVEL DE SER RESOLVIDA POR ESTE PROGRAMA
      integer HOST_LINHAS, HOST_COLUNAS
      parameter (HOST_LINHAS  = 400)
      parameter (HOST_COLUNAS = 400)

C DIMENSOES DA MAIOR MATRIZ PARCIAL QUE PODE SER RESOLVIDA POR UM NO
      integer NO_LINHAS, NO_COLUNAS
      parameter (NO_LINHAS  = HOST_LINHAS/2+2)
      parameter (NO_COLUNAS = HOST_COLUNAS)

C====================
C VARIAVEIS TIPO COMMON
C====================
C PARA COMUNICACAO ENTRE HOST E NO-MESTRE
      integer   PARCIA,   USADOS,   PRECIS,   MAIEPS,   EMLOOP
      integer S_PARCIA, S_USADOS, S_PRECIS, S_MAIEPS, S_EMLOOP
      parameter (PARCIA = 1) ! matriz PARCIAl com o resultado de 1 no
      parameter (USADOS = 2) !dados fornecidos pelo usuario
      parameter (PRECIS = 3) !PRECISao fornecida pelo usuario
      parameter (MAIEPS = 4) !MAIor EPSilon calculado pelos nos-de-calculo
      parameter (EMLOOP = 5) !poe nos-de-calculo EM LOOP


      real*8 PARCIAL(NO_LINHAS, NO_COLUNAS)
      parameter (S_PARCIA = NO_LINHAS*NO_COLUNAS* 2) ! S=tamanho em palavras de 32 bits
      COMMON /PARCIA/ PARCIAL


      integer LINHAS_USADAS,
     &        COLUNAS_USADAS,
     &        PROC_USADOS,
     &        ENDERECOS_MAIEPS(PROC_ALOCADOS),
     &        ENDERECOS_EMLOOP(PROC_ALOCADOS),
     &        ENDERECO_FLAGS,
     &        ENDERECOS_BORDAS(PROC_ALOCADOS),
     &        ENDERECOS_MISCEL(PROC_ALOCADOS),
     &        ENDERECOS_EPSILO(PROC_ALOCADOS),
     &        ENDERECOS_VARRE(PROC_ALOCADOS)
      parameter (S_USADOS = (4+6*PROC_ALOCADOS)* 1)
      COMMON /USADOS/ LINHAS_USADAS,
     &                COLUNAS_USADAS,
     &                PROC_USADOS,
     &                ENDERECOS_MAIEPS,
     &                ENDERECOS_EMLOOP,
     &                ENDERECO_FLAGS,
     &                ENDERECOS_BORDAS,
     &                ENDERECOS_MISCEL,
     &                ENDERECOS_EPSILO,
     &                ENDERECOS_VARRE
      real*8 PRECISAO
      parameter (S_PRECIS = 1* 2)
      COMMON /PRECIS/ PRECISAO


      real*8 MAIOR_EPSILON
      parameter (S_MAIEPS = 1* 2)
      COMMON /MAIEPS/ MAIOR_EPSILON
```

```
      integer FICA_EM_LOOP, NENHUM_LOOP, LOOP_INICIAL
      parameter (S_EMLOOP = 1* 1)
      COMMON /EMLOOP/ FICA_EM_LOOP
      parameter (NENHUM_LOOP  = 0)
      parameter (LOOP_INICIAL = 1)


C PARA COMUNICACAO ENTRE NO-MESTRE E NOS-DE-CALCULO
      integer  FLAGS,    BORDAS,   MISCEL,   EPSILO,   VARRE
      integer S_FLAGS, S_BORDAS, S_MISCEL, S_EPSILO, S_VARRE
      parameter (FLAGS   = 6)
      parameter (BORDAS  = 7
      parameter (MISCEL  = 8)
      parameter (EPSILO  = 9)
      parameter (VARRE   = 10)


      integer FLAGS_FIM_DE_CALCULO(PROC_ALOCADOS),
     &        FIM_DE_CALCULO, NAO_FIM_DE_CALCULO, JA_RECEBEU
      parameter (S_FLAGS  = PROC_ALOCADOS* 1)
      COMMON /FLAGS/ FLAGS_FIM_DE_CALCULO
      parameter (FIM_DE_CALCULO = 0)
      parameter (NAO_FIM_DE_CALCULO = 1)
      parameter (JA_RECEBEU = 2)


      real*8 DUAS_BORDAS(2*NO_COLUNAS)
      parameter (S_BORDAS = 2*NO_COLUNAS* 2)
      COMMON /BORDAS/ DUAS_BORDAS


      integer POSICAO,
     &        PTRSUP,
     &        PTRINF,
     &        ESTENO,
     &        SUPERIOR, INTERNO, INFERIOR
      parameter (S_MISCEL = 4* 1)
      COMMON /MISCEL/ POSICAO,
     &                PTRSUP,
     &                PTRINF,
     &                ESTENO
      parameter (SUPERIOR = 1)
      parameter (INTERNO  = 2)
      parameter (INFERIOR = 3)


      real*8 EPS_DO_NO
      parameter (S_EPSILO = 1* 2)
      COMMON /EPSILO/ EPS_DO_NO


      integer ENE
      parameter (S_VARRE = 1* 1)
      COMMON /VARRE/ENE


C
C VARIAVEIS INTERNAS A ESTE PROGRAMA
C
       integer LINHA, COLUNA, NELEMENTOS, AUX
       integer MEUFLAG


       real*8 ROJAC2,
     &        OMEGAENE,
     &        RESID,
     &        ATUAL(NOLINHAS,NOCOLUNAS)
```

```
C Função usada
      real*8 CONINI


C Constantes
      real*8 ZERO, MEIO, UM, DOIS, QUARTO, PI, QUATRO
      parameter (ZERO=0.0d0, MEIO=0.5d0, UM=1.0d0, DOIS= 2.0d0)
      parameter (QUARTO=0.25d0, PI=3.141592653589793d0, QUATRO=4.0d0)


C=====================================================================
C PONTO DE ENTRADA PRINCIPAL
C=====================================================================

C---------------------------------------------------------------------------------------
C Verifica se deve terminar o programa: sim caso ENE=0 e nao, caso contrario
C Verifica se deve ficar em loop: sim caso foi startado mas ainda nao recebeu as bordas do mestre
C---------------------------------------------------------------------------------------

1000  if (ENE .eq. 0) go to 3000
      if (FICAEMLOOP .eq. NENHUMLOOP) go to 2000
      go to 1000


C=====================================================================
C PONTO DE ENTRADA SECUNDARIO
C=====================================================================

C Calcula parametros do método SOR

2000    NELEMENTOS = PTRINF-PTRSUP+1
        ROJAC2 =((DCOS(PI/NELEMENTOS)+DCOS(PI/COLUSADAS))*MEIO)**DOIS

C Calcula o endereco do flag deste no, na matriz de flags do no mestre. Multiplica ESTENO por 4, pois 4 é o # de bytes
de um inteiro

        MEUFLAG = ENDERECOFLAGS + 4*(ESTENO-1)


C Calcula parametros de iniciaçao

      if (ENE .eq. 1) then
        LINHA = 1
        do 900 AUX = PTRSUP, PTRINF
          do 901 COLUNA = 1, COLUNASUSADAS
            PARCIAL(LINHA,COLUNA) = CONINI(AUX,COLUNA, LINHASUSADAS,COLUNASUSADAS)
901       continue
          LINHA = LINHA + 1
900     continue

902     OMEGAENE = UM
        do 903 LINHA = 2, NELEMENTOS-1
          do 904 COLUNA = 2, COLUSADAS-1
            ATUAL(LINHA,COLUNA) = ZERO
904       continue
903     continue
      end if


C---------------------
C Despachador principal
C---------------------
      if (POSICAO .eq. SUPERIOR) GO TO 1
      if (POSICAO .eq. INTERNO ) GO TO 2
      if (POSICAO .eq. INFERIOR) GO TO 3
```

```
C=================================================================
C ## 1 ## PROCESSADOR E BORDA SUPERIOR
C=================================================================

C-----------------------------------------------
C Coloca a borda INFERIOR, passada por outro processador
C-----------------------------------------------

1     do 10 COLUNA = 1, COLUNASUSADAS
        PARCIAL(NELEMENTOS,COLUNA) = DUASBORDAS(COLUNA)
10    continue

C--------------------------------------
C Coloca as condiçoes de contorno em ATUAL
C--------------------------------------

      do 11 COLUNA = 1, COLUNASUSADAS
        ATUAL(1         ,COLUNA) = PARCIAL(1         ,COLUNA)
        ATUAL(NELEMENTOS,COLUNA) = PARCIAL(NELEMENTOS,COLUNA)
11    continue
      do 12 LINHA = 2, NELEMENTOS-1
        ATUAL(LINHA,1) = PARCIAL(LINHA,1)
        ATUAL(LINHA,COLUNASUSADAS) = PARCIAL(LINHA,COLUNASUSADAS)
12    continue

C-------------------------------------------------------------
C Calcula, usando diferenças finitas, com PARCIAL, gerando MATUAL
C-------------------------------------------------------------

      do 13 COLUNA = 2, COLUNASUSADAS-1
        do 14 LINHA = 2, NELEMENTOS-1
          if (MOD(LINHA+COLUNA,2) .eq. MOD(ENE,2)) then
            RESID = PARCIAL(LINHA-1,COLUNA) + PARCIAL(LINHA+1,COLUNA) +
     &              PARCIAL(LINHA,COLUNA-1) + PARCIAL(LINHA,COLUNA+1) -
     &              QUATRO * PARCIAL(LINHA,COLUNA)
            ATUAL(LINHA,COLUNA) = ATUAL(LINHA,COLUNA) + OMEGAENE*RESID*QUARTO
          end if
14      continue
13    continue

C--------------------------------------
C Arruma o vetor de borda, a ser devolvido
C--------------------------------------

      do 15 COLUNA = 1, COLUNASUSADAS
         DUASBORDAS(COLUNA+NOCOLUNAS) = ATUAL(NELEMENTOS-1,COLUNA)
15  continue


C----------------------------------
C Vai calcular os parametros de saida
C----------------------------------

      GO TO 4


C=================================================================
C ## 2 ## PROCESSADOR E INTERNO
C=================================================================

C-----------------------------------------------
C Arruma as bordas passadas pelos 2 processadores vizinhos
C-----------------------------------------------

2     do 20 COLUNA = 1, COLUNASUSADAS
        PARCIAL(1,COLUNA)          = DUASBORDAS(COLUNA+NOCOLUNAS)
        PARCIAL(NELEMENTOS,COLUNA) = DUASBORDAS(COLUNA)
20    continue
```

```
C--------------------------------------
C Coloca as condiçoes de contorno em ATUAL
C--------------------------------------

      do 21 COLUNA = 1, COLUNASUSADAS
        ATUAL(1         ,COLUNA) = PARCIAL(1         ,COLUNA)
        ATUAL(NELEMENTOS,COLUNA) = PARCIAL(NELEMENTOS,COLUNA)
21    continue
      do 22 LINHA = 2, NELEMENTOS-1
        ATUAL(LINHA,            1) = PARCIAL(LINHA,            1)
        ATUAL(LINHA,COLUNASUSADAS) = PARCIAL(LINHA,COLUNASUSADAS )
22    continue


C-------------------------------------------------------------
C Calcula, usando diferenças finitas, com PARCIAL, gerando MATUAL
C-------------------------------------------------------------

      do 23 COLUNA = 2, COLUNASUSADAS-1
        do 24 LINHA = 2, NELEMENTOS-1
          if (MOD(LINHA+COLUNA,2) .eq. MOD(ENE,2)) then
            RESID = PARCIAL(LINHA-1,COLUNA) + PARCIAL(LINHA+1,COLUNA) +
     &              PARCIAL(LINHA,COLUNA-1) + PARCIAL(LINHA,COLUNA+1) -
     &              QUATRO * PARCIAL(LINHA,COLUNA)
            ATUAL(LINHA,COLUNA) = ATUAL(LINHA,COLUNA) + OMEGAENE*RESID*QUARTO
          end if
24      continue
23    continue

C-----------------------------------------
C Arruma os vetores de borda, a serem devolvidos
C-----------------------------------------

      do 25 COLUNA = 1, COLUNASUSADAS
        DUASBORDAS(COLUNA          ) = ATUAL(2           ,COLUNA)
        DUASBORDAS(COLUNA+NOCOLUNAS) = ATUAL(NELEMENTOS-1,COLUNA)
25    continue

C---------------------------------
C Vai calcular os parametros de saida
C---------------------------------

      GO TO 4


C=================================================================
C ## 3 ## PROCESSADOR E BORDA INFERIOR
C=================================================================
C-----------------------------------------------
C Coloca a borda SUPERIOR, passada por outro processador
C-----------------------------------------------

3     do 30 COLUNA = 1, COLUNASUSADAS
        PARCIAL(1,COLUNA) = DUASBORDAS(COLUNA+NOCOLUNAS)
30    continue

C--------------------------------------
C Coloca as condiçoes de contorno em ATUAL
C--------------------------------------

      do 31 COLUNA = 1, COLUNASUSADAS
        ATUAL(1         ,COLUNA) = PARCIAL(1         ,COLUNA)
        ATUAL(NELEMENTOS,COLUNA) = PARCIAL(NELEMENTOS,COLUNA)
31    continue
      do 32 LINHA = 2, NELEMENTOS-1
        ATUAL(LINHA,            1) = PARCIAL(LINHA,            1)
        ATUAL(LINHA,COLUNASUSADAS) = PARCIAL(LINHA,COLUNASUSADAS)
32    continue
```

```
C-------------------------------------------------------------
C Calcula, usando diferenças finitas, com PARCIAL, gerando MATUAL
C-------------------------------------------------------------

      do 33 COLUNA = 2, COLUSADAS-1
        do 34 LINHA = 2, NELEMENTOS-1
          if (MOD(LINHA+COLUNA,2) .eq. MOD(ENE,2)) then
            RESID = PARCIAL(LINHA-1,COLUNA) + PARCIAL(LINHA+1,COLUNA) +
     &              PARCIAL(LINHA,COLUNA-1) + PARCIAL(LINHA,COLUNA+1) -
     &              QUATRO * PARCIAL(LINHA,COLUNA)
            ATUAL(LINHA,COLUNA) = ATUAL(LINHA,COLUNA) + OMEGAENE*RESID*QUARTO
          end if
34      continue
33    continue


C---------------------------------------
C Arruma o vetor de borda, a ser devolvido
C---------------------------------------

      do 35 COLUNA = 1, COLUNASUSADAS
        DUASBORDAS(COLUNA) = ATUAL(2,COLUNA)
35    continue

C-----------------------------------
C Vai calcular os parametros de saida
C-----------------------------------

      GO TO 4


C=================================================================
C ## 4 ## CALCULA PARAMETROS DE SAIDA
C=================================================================

C---------------------------------
C Calcula valor de EPSILON neste no.
C---------------------------------

4     EPSDONO = ZERO
      do 40 COLUNA = 2, COLUNASUSADAS-1
        do 41 LINHA = 2, NELEMENTOS-1
          EPSDONO = EPSDONO +   ABS( ATUAL(LINHA,COLUNA)-PARCIAL(LINHA,COLUNA) )
41      continue
40    continue

C-------------------------------------------------------------
C Poe matriz ATUAL na PARCIAL, para funcionar na proxima iteraçao.
C-------------------------------------------------------------

      do 42 COLUNA = 1, COLUNASUSADAS
        do 43 LINHA = 1, NELEMENTOS
          PARCIAL(LINHA,COLUNA) = ATUAL(LINHA,COLUNA)
43      continue
42    continue


C-------------------------------------------------------------
C Arruma o valor de OMEGAENE, para funcionar na proxima iteraçao.
C-------------------------------------------------------------

      if (ENE .eq. 1) then
        OMEGAENE = UM / (UM - MEIO*ROJAC2)
      else
        OMEGAENE = UM / (UM - QUARTO * ROJAC2 * OMEGAENE)
      end if
```

```
C-------------------------------------------------------------------
C Avisa o mestre (através da matriz de flags) que ja acabou seus calculos.
C-------------------------------------------------------------------

      call acpsup
      long(MEUFLAG) = FIMDECALCULO
      call acpusr

C-------------------------------------------------------------------------------
C Seta seu flag para ficar em loop esperando pelas novas bordas mandadas pelo mestre
C  e vai para o loop.
C-------------------------------------------------------------------------------

      FICAEMLOOP = LOOPINICIAL
      go to 1000


C---------------
C FIM DO PROGRAMA
C---------------

3000  return
      end
```

```
      real*8 function CONINI(L,C, LINHASUSADAS,COLUNASUSADAS)

C Entrada
      integer L,C, LINHASUSADAS,COLUNASUSADAS

      CONINI = 0.0
      if (L .eq. 1)             CONINI = 100.0
      if (L .eq. LINHASUSADAS)  CONINI = 100.0
      if (C .eq. 1)             CONINI = 100.0
      if (C .eq. COLUNASUSADAS) CONINI =   0.0

      return
      end
```

apêndice E

Referências Utilizadas

Referências Utilizadas

A

- *ACP Software User's Guide for Event Oriented Processing*, Advanced Computer Program, FERMILAB, Revision 3.0, December 4, 1987
- *ACP Users's Guide for Utilities*, Advanced Computer Program, FERMILAB
- G.C. Fox e S. Otto, *Algorithms for concurrent processors*, Physics Today, pag.50, May/84
- B. Carnahan et al, *Applied Numerical Methods*, John Wiley & Sons, 1969, cap.7
- C. Bogni e L. Marrone, *Arquitecturas No Convencionales*, I Escola Brasileiro-Argentina de Informática, editorial Kapelusz S.A., Buenos Aires, ed. preliminar, 1987, cap.2
- B. Schulze e R. Valois, Segunda Geração de Processadores Paralelos do "*Advanced Computer Program* " (ACP), II Simpósio Brasileiro de Arquiteturas de Computadores - Processamento Paralelo, 1988

C

- K. Hwang e F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1986

D

- M.J.Quinn, *Designing Efficient Algorithms For Parallel Computers*, McGraw-Hill, 1987,
- S.L.Ross, *Differential Equations*, John Wiley & Sons, 2nd edition, 1974

F

- G.E. Forsythe, *Finite-Difference Methods for Partial Differential Equations*, John Wiley & Sons, 1969

I

- J.R.Biel, *Introduction to ACP software for event oriented processing*, curso dado no CBPF, april 4, 1988
- E. Eichten et al, *Lattice Gauge Theory and the ACP*, FERMILAB internal communication, May 6, 1987

M

- R.Courant and D.Hilbert, *Methods of Mathematical Physics*, vol.2, Interscience, 1962
- J.R.F. Xavier, Multiprocessamento - uma visão geral com enfoque em processamento paralelo, Comunicação do Grupo de Processamento Paralelo, IFT, 1987

N

- W.H. Press et al, *Numerical Recipes - The Art of Scientific Computing*, Cambridge University Press, 1987
- G.D. Smith, *Numerical solution of partial differential equations - finite difference methods*, Clarendon Press - Oxford, 1978, cap. 1 e 5

P

- R.W.Hockney e C.R.Jesshope, *Parallel Computers*, Adam Hilger Ltd., Bristol, 1986, cap.3
- S.Williams, *Programming the 68000*, SYBEX, 1985
- Jeff Hecht, *Physical Limits of Computing*, Computer in Physics, jul/aug 1989

R

- F.C. Mokarzel, Reestruturação Automática de Programas Seqüenciais para Processamento Paralelo, II Simpósio Brasileiro de Arquitetura de Computadores- Processamento Paralelo, 1988

V

- *VAX System and Options Catalog*, 1987 January-March, cap.3
- D. Jones, *VMEbus Primer*, Electronics and Wireless World, pag.8, December/86

extras

- L.Leventhal et al., *68000 Assembly Language Programming*, second edition, McGraw-Hill,1986
- W. Cramer e G.Kane, 68000 Microprocessador, McGraw-Hill, 1986

glossário

A

Absoluto curto: modo de endereçamento do MC68020 no qual o operando é especificado na própria instrução. Somente podem ser usados endereços entre $0000 e $FFFF.

Absoluto longo: modo de endereçamento do MC68020 no qual o operando é especificado na própria instrução e no qual podem ser usados endereços entre $00000000 e $FFFFFFFF.

Algoritmos assíncronos: são aqueles que trabalham sem sincronização entre os processos. Todos os processos podem estar trabalhando para atingir o mesmo objetivo ou podem ter tarefas diferentes mas o elemento essencial é que nenhum processo nunca tem que esperar que outro processador lhe forneça seus dados de entrada.

Algoritmo particionado: algoritmo paralelo no qual há uma distribuição das tarefas entre os vários processadores.

Algoritmo *pipelined* : algoritmo paralelo no qual a saída de um segmento passa a ser a entrada do próximo segmento. A entrada do algoritmo serve de entrada ao primeiro segmento e a saída do último segmento é a saída do algoritmo.

Algoritmo sistólico: tipo especial de algoritmo *pipelined* com três características que o distingüe daquele: (i) o fluxo de dados é regular, (ii) os dados podem fluir em mais do que uma direção e (iii) os cálculos feitos em cada segmento são essencialmente idênticos.

Assembler: programa que converte instruções escritas em linguagem *assembly* para linguagem de máquina. Em português, programa montador.

Atribuição : é a distribuição de partes de um algoritmo (que sofreu uma partição) entre vários processadores.

Array processor: é um computador com vários elementos processadores que operam em paralelo e em sincronismo. Em português, processador matricial.

B

Barramento tronco: barramento usado pelo sistema ACP, que conecta vários bastidores VME ao computador-hospedeiro. Em inglês, *branch bus*.

Bloqueio : estado no qual um ou mais processos ficam quando esperam pela liberação de algum recurso. Em inglês, *deadlock* .

Branch bus: barramento usado pelo sistema ACP, que conecta vários bastidores VME ao computador-hospedeiro. Em português, barramento tronco.

Broadcasting : envio de uma mensagem de uma fonte a vários destinos, ao mesmo tempo.

Buffer: área de memória usada para se guardar dados que serão usados posteriormente.

C

Carga de trabalho: é a maneira como as tarefas são distribuídas entre os diversos processadores num computador paralelo. É importante pois uma má distribuição das tarefas pode sobrecarregar alguns processadores e deixar outros ociosos.

Códigos de condição: um conjunto de *flags* no microprocessador que fornece informações sobre a última instrução executada, tais como se um operando ou resultado era zero etc.

Compilador : programa que traduz instruções escritas em linguagem de alto nível (FORTRAN, Pascal, C etc.) para linguagem de máquina.

Comunicação direta: leitura ou escrita de variáveis nos nós ACP sem a participação do computador-hospedeiro.

Concorrentes : são aqueles dois processos nos quais a primeira operação do segundo processo começa antes do término da última operação do primeiro processo.

Condições de Bernstein: são as condições que dois processos seqüenciais têm que satisfazer para que possam ser executados em paralelo.

Conflito : situação na qual dois ou mais nós precisam acessar o mesmo módulo de memória ao mesmo tempo.

Contador do programa: um registrador do microprocessador que contém o endereço da próxima instrução a ser executada. É comumente abreviado por PC.

Crossbar: é a rede de interconexão na qual há um barramento conectando cada um dos processadores a cada um dos módulos (memória, interface de E/S etc).

D

Daisy-chain : esquema de prioridade no qual cada dispositivo recebe uma prioridade (fixa) de acordo com sua posição ao longo do barramento.

Deadlock : estado no qual um ou mais processos ficam quando esperam pela liberação de algum recurso. Em português, bloqueio.

Direto do registrador de dados: modo de endereçamento do MC68020 no qual o operando é o conteúdo de um registrador de dados.

Direto do registrador de endereços: modo de endereçamento do MC68020 no qual o operando é o conteúdo de um registrador de endereços.

E

Efeito Amdahl: mostra que, em geral, o *speedup* é uma função crescente do tamanho do problema. Isto se deve ao fato de que para problemas muito pequenos a paralelização não é eficiente.

E-lógico : operação na qual o bit do operando-fonte e o do operando-destino devem ter o valor 1 a fim de que o resultado também valha 1.

Escrita direta: escrita de variáveis nos nós ACP sem a participação do computador-hospedeiro.

F

Flag: bit que contém um valor que representa um determinado estado entre 2 possíveis.

Fluxo de dados: seqüência de dados usada por um fluxo de instruções.

Fluxo de instruções: seqüência de instruções executada por uma máquina.

I

Imediato: modo de endereçamento do MC68020 no qual o operando é especificado imediatamente após a instrução.

Implícito: modo de endereçamento do MC68020 no qual a instrução propriamente dita contém o endereço do operando e os operandos são registradores. O endereço efetivo está implícito na instrução.

Indireto, com deslocamento, via PC: modo de endereçamento do MC68020 no qual o endereço do operando é especificado somando-se um deslocamento ao valor atual do PC.

Indireto, com deslocamento, via registrador de endereços: modo de endereçamento do MC68020 no qual o endereço do operando é especificado somando-se um deslocamento ao valor de um registrador de endereços.

Indireto, com indexação e deslocamento, via PC: modo de endereçamento do MC68020 no qual o endereço do operando é especificado somando-se um deslocamento e um índice (que pode ser um registrador de dados ou de endereços) ao valor atual do PC. Pode ser usado, por exemplo, numa construção do tipo `case`, do Pascal.

Indireto, com indexação e deslocamento, via registrador de endereços: modo de endereçamento do MC68020 no qual o endereço do operando é especificado somando-se um deslocamento e um índice (que pode ser um registrador de dados ou de endereços) ao valor de um registrador de endereços. Pode ser usado, por exemplo, numa construção do tipo `case`, do Pascal.

Indireto, com pré-decremento, via registrador de endereços: modo de endereçamento do MC68020 no qual o operando é apontado pelo registrador de endereços. Antes de calcular o endereço do operando esta instrução decrementa (isto é, diminui de 2 bytes) o registrador de endereços especificado. É muito utilizado para se acrescentar objetos à pilha.

Indireto, com pós-incremento, via registrador de endereços: modo de endereçamento do MC68020 no qual o operando é apontado pelo registrador de endereços. Depois de calcular o endereço do operando esta instrução incrementa (isto é, aumenta de 2 bytes) o registrador de endereços especificado. É muito utilizado para se retirar objetos da pilha.

Indireto, via registrador de endereços: modo de endereçamento do MC68020 no qual o operando é apontado pelo registrador de endereços.

Instrução: um comando que é entendido e executado pelo microprocessador.

J

José Rodolfo Ferreira Xavier: o autor desta tese!

L

Lei de Amdahl: lei que prediz que códigos seqüenciais limitam o *speedup* alcançável por um algoritmo paralelo.

Leitura direta: leitura de variáveis nos nós ACP sem a participação do computador-hospedeiro.

Linguagem *assembly*: linguagem para se trabalhar com um microprocessador, formada por mnemônicos. É um meio termo entre linguagem de máquina e linguagens de alto nível. Veja também *assembler*.

Linguagem de máquina: linguagem que o microprocessador consegue reconhecer, formada apenas por números.

`Long:` instrução (FORTRAN) que permite a leitura (e escrita) de variáveis diretamente no barramento VME.

M

Memória de acesso aleatório: é a memória volátil de um computador isto é, aquela que armazena informações somente enquanto o computador está ligado. Em inglês, *random access memory*. Note que os nomes em português e em inglês são bastante infelizes, uma vez que esta memória, na verdade, não tem nada de aleatório. É normalmente chamada de RAM.

Memória apenas de leitura: é a memória não-volátil do computador isto é, aquela que continua armazenando informações mesmo quando o computador está desligado. Em inglês, *read only memory*. É comumente chamada de ROM.

Memória local: memória que somente pode ser acessada por um determinado nó num sistema multiprocessador.

Memória multiporta: memória que contém unidades funcionais para chaveamento e arbitramento na sua interface.

Métodos de prioridades: são métodos usados para se resolver problemas de conflitos.

Mnemônico : um código que se parece com a linguagem normal (dos humanos!) e que representa uma instrução do microprocessador. Os programas montadores (*assemblers*) convertem programas escritos com mnemônicos para instruções em linguagem de máquina.

Modo de endereçamento: qualquer um dos muitos modos usados pelo microprocessador para determinar seu operando.

Monitoração : a constante observação, por parte do computador-hospedeiro, do programa executado pelo nó-mestre.

Montador : programa que converte instruções escritas em linguagem *assembly* para linguagem de máquina. Em inglês, *assembler*.

Multicomputador : sistema no qual cada elemento processador corresponde a um computador autônomo (com todas as suas características).

Multiprocessador : sistema no qual cada elemento processador é composto de apenas um processador (e outras unidades funcionais como memória local etc.) e é controlado por um único sistema operacional.

N

Não-lógico : operação na qual os bits do operando são invertidos isto é, os que tinham valor 0 passam a ter valor 1 e vice-versa.

Nó: conjunto formado, basicamente, por (i) um processador, (ii) interface de Entrada/Saída e (iii) memória local. Pode incluir outras unidades funcionais como co-processador de ponto flutuante etc.

Nó-de-cálculo : o nó ACP que executa os cálculos da resolução da equação de Laplace num certo sub-domínio. Também chamado de processador-de-cálculo.

Nó-mestre: nó ACP que monitora a resolução do problema por parte dos nós-de-cálculo. Também chamado de processador-mestre.

O

Operando : um valor usado por uma instrução do microprocessador para que ele execute sua função. Um operando é o análogo de um parâmetro numa linguagem de alto nível.

Operando-destino : o segundo operando (numa instrução de 2 operandos, ou o primeiro numa instrução de apenas 1 operando) de uma instrução do microprocessador.

Operando-fonte : o primeiro operando de uma instrução do microprocessador.

Ou-lógico : operação na qual tanto o bit do operando-fonte quanto o do operando-destino, ou ambos, deve ter o valor 1 a fim de que o resultado também tenha o valor 1.

Ou-exclusivo-lógico : operação na qual o bit do operando-fonte ou o do operando-destino, mas não ambos, deve ter o valor 1 a fim de que o resultado também valha 1.

P

Palavra : designa o tamanho natural dos operandos do microprocessador. No caso do MC68020 a palavra tem o comprimento de 2 bytes.

Palavra-longa : designa uma palavra de comprimento duplo. No caso do MC68020 a palavra-longa tem o comprimento de 4 bytes.

Partição : é a divisão de um algoritmo em várias partes, a fim de que ele possa ser resolvido com o uso de um multiprocessador.

Pilha : estrutura de dados do tipo último-que-entra-é-o-primeiro-que-sai (LIFO, *last in first out*) usada para armazenamento de variáveis locais e globais, passagem de parâmetros, endereços de retorno etc.

Pipeline : tipo de arquitetura na qual há várias unidades funcionais dispostas seqüencialmente, de forma que a saída fornecida por uma unidade funcional passa a ser a entrada da próxima unidade, como numa linha de montagem.

Processador-de-cálculo : o nó ACP que executa os cálculos da resolução da equação de Laplace num certo sub-domínio. Também chamado de nó-de-cálculo.

Processador matricial: é um computador paralelo com vários elementos processadores que operam em paralelo e em sincronismo. Em inglês, *array processor*.

Processador-mestre : nó ACP que monitora a resolução do problema por parte dos nós-de-cálculo. Também chamado de nó-mestre.

Processamento paralelo: forma eficiente de processamento da informação com ênfase na exploração de eventos concorrentes no processo computacional.

R

Random access memory: é a memória volátil de um computador isto é, aquela que armazena informações somente enquanto o computador está ligado. Em português, memória de acesso aleatório. Note que os nomes em inglês e em português são bastante infelizes, uma vez que esta memória, na verdade, não tem nada de aleatório. É comumente chamada de RAM.

Read only memory: é a memória não-volátil do computador isto é, aquela que continua armazenando informações mesmo quando o computador está desligado. Em português, memória apenas de leitura. É comumente chamada de ROM.

Registrador: uma posição de memória dentro do próprio microprocessador.

Registrador de dados: registrador usado para armazenamento de um número.

Registrador de endereço: registrador usado para armazenamento de um endereço.

Registrador de estado: um registrador do microprocessador que inclui o registrador de condição de estado e várias informações sobre o estado atual do microprocessador.

S

Seção crítica: aquela parte de um programa na qual as variáveis compartilhadas são alteradas.

Speedup: é a razão entre o tempo de execução do melhor algoritmo seqüencial e o do algoritmo paralelo associado.

T

Topo da pilha: posição de memória que corresponde ao menor endereço da pilha, uma vez que a pilha cresce no sentido dos endereços mais altos para os mais baixos. É o endereço apontado pelo apontador da pilha (SP, *stack pointer*).

U

User's Parameters File : arquivo que contém instruções ao compilador do ACP, tais como o número de nós que se deseja alocar, os nomes dos arquivos-fonte que contêm os programas a serem executados etc. É comumente abreviado por UPF.

V

Variável compartilhada: aquela variável que pode ser modificada por processos concorrentes.

glossário de siglas

A

An: *Address Register* n. Em Português, Registrador de Endereços n, com n=0...7.
ACP: *Advanced Computer Program*. Em português, Programa de Computador Avançado.
ALGOL: *Algorithmic Language* . Em português, Linguagem Algorítmica.

B

BBC: *Branch Bus Controler.* Em português, Controlador do Barramento Tronco.
BBT: *Branch Bus Terminator.* Em português, Terminação para o Barramento Tronco.
BCD: *Binary-Coded Decimal*. Em português, Decimal Codificado em Binário.
BI: Busca de Instrução.
BO: Busca de Operando.
BVI: *Branch Bus to VME Interface.* Em português, Interface entre o Barramento Tronco e o barramento VME.

C

CBPF: Centro Brasileiro de Pesquisas Físicas, no Rio de Janeiro.
CCR: *Condition Code Register.* Em português, Registrador do Código de Condição.
CERNLIB: biblioteca de rotinas científicas, muito usada pelos físicos.
CPU: *Central Processing Unit.* Em português, Unidade Central de Processamento.

D

Dn: *Data Register* n. Em Português, Registrador de Dados n, com n=0...7.
DEC: *Digital Equipment Corporation.*
DI: Decodificação da Instrução.

E

EI: Execução da Instrução.
EDP; Equação Diferencial Parcial.
EDVAC : *Electronic Discrete Variable Automatic Computer.*

F

FERMILAB: *Fermi National Accelerator Laboratory*, nos EUA.
FIFO: *First In, First Out.* Em português, Primeiro que Entra, Primeiro que Sai.
FLOPS: *Floating-point Instructions per Second.* Em português, Instruções em Ponto Flutuante por Segundo.
FORTRAN: Formula Translation .

I

IBM: *International Business Machine.*
IEEE: *Institute of Electrical and Electronic Engineers.*
IFT: Instituto de Física Teórica, em São Paulo.
IMSL: biblioteca de rotinas matemáticas, muito usada pelos físicos.

L

LAFEX : Laboratório de Física Experimental de Altas Energias, do CBPF.

M

MIMD: *Multiple Instruction stream, Multiple Data stream.* Em português, Fluxo de Instruções múltiplo, Fluxo de Dados múltiplo.
MISD: *Multiple Instruction stream, Single Data stream.* Em português, Fluxo de Instruções múltiplo, Fluxo de Dados único.
MSI: *Medium Scale Integration.* Em português, Integração em Média Escala.

P

PC: *Program Counter*. Em português, Contador do Programa.

R

RAM: *Random Access Memory.* Em português, Memória de Acesso Aleatório.
RIESP: Rede de Interconexão entre Entrada/Saída e Processador.
RIPM: Rede de Interconexão entre Processador e Memória.
RISI: Rede de Interconexão de Sinal de Interrupção.
ROM: *Read Only Memory.* Em português, Memória Apenas de Leitura.

S

SIMD: *Single Instruction stream, Multiple Data stream.* Em português, Fluxo de Instruções único, Fluxo de Dados múltiplo.
SISD: *Single Instruction stream, Single Data stream.* Em português, Fluxo de Instruções único, Fluxo de Dados único.
SOR: *Simultaneous Over-Relaxation method.*
SP: *Stack Pointer*. Em português, Ponteiro da Pilha.
SSI: *Small Scale Integration*. Em português, Integração em Pequena Escala.

T

TRADIC: *Transistorized Digital Computer.*

U

UCP: Unidade Central de Processamento.
ULA: Unidade Lógica e Aritmética.

V

VLSI: *Very Large Scale Integration.* Em português, Integração em Escala Muito Grande.
VME: *Versa Module Eurocard.*
VRM: *VME Resource Module.* Em português, Módulo de Recursos para o VME.

índice de figuras e tabelas

índice de figuras e tabelas

capítulo 1 Processamento Paralelo

figura 1: a história da velocidade de processamento dos computadores desde 1950, mostrando um crescimento de um fator 10 a cada 5 anos - 3
figura 2: abordagens para se obter multiprocessamento com apenas 1 processador, usando (a) processamento em batch - 6
figura 2 (continuação): abordagens para se obter multiprocessamento com apenas 1 processador,usando (b) multiprogramação, (c) compartilhamento de tempo - 7
figura 3: diagrama mostrando a execução das instruções num computador com pipeline - 8
figura 4: diversas topologias para redes de interconexão. Os círculos representam os elementos processadores e as linhas representam as ligações entre eles - 10
figura 5: esquema da organização de um multiprocessador - 11
figura 6: computador tipo SISD - 12
figura 7: computador tipo SIMD - 12
figura 8: computador tipo MISD - 13
figura 9: computador tipo MIMD - 13

capítulo 2 Arquitetura dos Multiprocessadores

figura 1: (a) nó de um sistema multiprocessador - 18
figura 1: (b) conexão entre os nós e o Sistema de Transferência de Mensagens - 18
figura 2: multiprocessador fortemente acoplado - 20
figura 3: barramento único - 21
figura 4: múltiplos barramentos - 23
figura 5: rede tipo crossbar - 23
figura 6: crossbar para ligações entre processadores-memórias-E/S - 24
figura 7: memória multiporta - 25
figura 8: prioridade nas portas de uma memória multiporta - 25
figura 9: acesso exclusivo à memória por um processador - 26

tabela 1: comparação entre multiprocessadores com rede de tempo compartilhado, rede tipo crossbar e memória multiporta - 26

capítulo 3 Concorrência nos Multiprocessadores

figura 1: grafo de precedência de um programa que usa as construções cobegin e coend - 30
figura 2: representação hierárquica de um programa - 33
figura 3: execuções (a, b) seqüenciais e (c) paralela de um programa - 34

capítulo 4 Desenvolvendo Algoritmos Paralelos

figura 1: Efeito Amdahl: como regra geral o speedup é uma função crescente do tamanho do problema - 44
figura 2: a contenção dos processadores por uma única memória, a qual contém todo o código a ser utilizado pelos processadores, limita o speedup - 45

capítulo 5 O Sistema ACP

figura 1: esquema geral de um computador ACP - 51

capítulo 6 Equações Diferenciais Parciais

figura 1: cálculo do elemento $u_{l,c}$ em função dos seus 4 vizinhos mais próximos - 61

capítulo 7 O Programa CALOR (Hospedeiro-Nós)

figura 1: domínio original, discretizado - 69
figura 2: os p sub-domínios usados pelos nós - 70
figura 3: cálculo do número de linhas - 71

capítulo 8 O Programa CALOR (Nós-Nós)

figura 1: diagrama de blocos do programa que roda no computador-hospedeiro - 78
figura 2: diagrama de blocos do programa que roda no processador-mestre - 79
figura 3: diagrama de blocos do programa que roda nos nós-de-cálculo - 80
figura 4: esquema da comunicação entre o computador-hospedeiro, nó-mestre e os nós-de-cálculo - 81
figura 5: esquema da memória dos nós ACP, mostrando como se calcula o endereço de uma certa variável, com o uso das funções ACP_NODE_VME_ADDRESS e ACP_NODE_BLOCK_OFFSET - 84

capítulo 9 Dados e Análises

figura 1: condições de contorno e solução da equação de Laplace - 92
figura 2: tempo de processamento em função do número de colunas do domínio original (MicroVAX II) - 93
figura 3: tempo de processamento (em minutos) sem comunicação direta entre os nós ACP - 94
figura 4: tempo de processamento (em minutos) com comunicação direta entre os nós ACP - 95
figura 5: comparação entre os tempos de processamento (em minutos) em dois casos: sem comunicação direta nó-nó (pontos cheios) e com comunicação direta nó-nó (pontos vazados) - 96

tabela 1: tempo de processamento (em segundos e minutos), usando o MicroVAX II - 93
tabela 2: tempo de processamento (em segundos) sem comunicação direta entre os nós ACP - 94
tabela 3: tempo de processamento (em segundos) com comunicação direta entre os nós ACP - 95
tabela 4: speedups dos programas sem e com comunicação direta entre os nós ACP - 97
tabela 5: razões entre os tempos de processamento com e sem comunicação direta nó-nó - 97
tabela 6: razões entre os vários tempos de processamento e o menor tempo de processamento, sem comunicação direta nó-nó - 98
tabela 7: razões entre os vários tempos de processamento e o menor tempo de processamento, com comunicação direta nó-nó - 98

apêndice A Programando o MC68020

figura 1: modo de endereçamento indireto, com pré-decremento, via registrador de endereços - G
figura 2: tabelas-verdade das instruções AND, OR e EOR - N
figura 3: instrução bit test - O

tabela 1: modos de endereçamento do MC68020 - J
tabela 2: operações de transferência de dados - K
tabela 3: instruções de aritmética inteira - M
tabela 4: instruções lógicas - N
tabela 5: instruções de deslocamento e de rotação - O
tabela 6: instruções de manipulação de bits - P
tabela 7: instruções de desvio baseado nos códigos de condição - Q
tabela 8: instruções de controle do programa - S
tabela 9: instruções de controle do sistema - S

índice remissivo

A

absoluto curto I
absoluto longo I
aceleração de Chebyshev 64
acesso mutuamente exclusivo 31
ACP$AREA:USER_COMMON.INC 53
ACP$GO 74
ACP$LAST_BLOCK 74, 86
ACP$MORE_BLOCKS 74
ACP$MULTICOMP 54
ACP$NO_GO 74
ACP_CONVERT_H_N 75
ACP_CONVERT_N_H 75
ACP_EXIT 53, 55
ACP_GETBLOCK 74, 86
ACP_GETEVENT 55
ACP_INIT 53, 55
ACP_NODE_BLOCK_OFFSET 83
ACP_NODE_VME_ADDRESS 83
ACP_SENDBLOCK 73, 86
ACP_SENDEVENT 55
ACPCHN 75
ACPCNH 75
ACPSUP 83
ACPUSR 83
acrescentar um objeto à pilha F
ADD M
ADDA L
ADDI L
ADDQ L
Algoritmo *pipelined* 42
algoritmo sistólico 43
algoritmos assíncronos 43
algoritmos particionados 43
alocação de novos recursos 32
alta velocidade de processamento 19
ANDI M
apêndices v
aplicações de tempo real 19
apontador 84
argumentos de sub-rotinas 53
aritmética de complemento de dois P
aritmética inteira L
armazenamento dos resultados 6
arquitetura da máquina 38
arquitetura do computador 38
arquiteturas dos sistemas de multiprocessadores 16
arquivo tipo UPF 72, 87
array processor 9
ASL O
ASR O
ativar um programa 86
atribuição 36, K

B

balanceamento estático de carga 70
barramento 81
barramento de tempo compartilhado 9,19
barramento único 21
barramento VME 82
bastidor 52, 82
BCC Q, S
BCD P
BCHG P
BCLR P
BCS Q
BEQ Q
BGE Q
BGT Q
BHI Q
bibliotecas 50
bibliotecas prontas 32
Binary-Coded Decimal P
binário B
BKP S
BLE Q
blocos de dados 53
bloqueio 31, 32
BLS Q
BLT Q
BMI Q
BNE Q
borda de um processador 72
bordas 67
BPL Q
BRA R
branch bus 50, 52, 82
Branch Bus Controler 52
Branch Bus terminator 52
Branch Bus to VME Interface 52
breakpoint S
broadcasting 20
BSET P
BSR R
BTST P
buffer 53
busca 6
busca da instrução 8
busca de operandos 6, 8
BVC Q
BVS Q

C

call R
CALOR (Hospedeiro-Nós) 67
CALOR_H_N AA

CALOR_SEM_ACP U
capítulos v
característica não-determinista 43
CARGA 71
carga de trabalho 45
Carry Bit D
CBPF ii
Central Processing Unit 52
Centro Brasileiro de Pesquisas Físicas ii
CERNLIB 50
chave 20
CHK S
ciclo de instrução 9
ciclo de memória 20
Ciência iii
CLASS 54
class 1 87
class 2 87
classe 56, 74
classe 1 87
classe 2 87
classificação das equações diferenciais parciais 59
classificação de Flynn 11
CLR M
CMP M
cobegin 29
codificação 32
codificação do algoritmo 38
coeficiente de difusão 59
coend 29
colisões quando do acesso à memória 17
common block 83
COMMON BLOCKS 53, 72
Compartilhamento de Tempo 6
compilação 50
compiladores B, R
complexidade temporal 37, 46
computador ACP 50
computador-hospedeiro 50, 67, 77
computadores com *pipeline* 8
comunicação 77, 81
comunicação bidirecional 73
comunicação direta entre os nós 53, 82
comunicação entre o computador-hospedeiro e os nós ACP 50, 68
comunicação entre os nós 50
comunicação unidirecional 73
comutatividade 34
concorrentes 29
condição de parada 67
condições de Bernstein 34
condições de contorno 67, 71
conflito 20
conflitos de memória 19, 20
contador do programa D
contenção dos processadores por recursos 44
controle S
convergência do método SOR 64
conversão de formatos 74, M
coordenação 53
código nativo dos microprocessadores B
códigos de condição D, R
crate 52
crossbar 20, 23
custos de comunicação 41

D

Daisy-Chain 22
DBF R
DBRA R
DBT R
decodificação 6
decodificação da instrução 8
decomposição 36
decomposição paralela eficiente do algoritmo 38
defaults do software básico do ACP 88
definições usadas pelo ACP 53
dependência dos dados 33
depuração 52
desativação dos programas dos nós 86
descarregamento de rotinas nos nós 50
desempenho 24, 41, 45, 77
desempenho de programas em paralelo 72
desempenho dos sistemas de multiprocessadores 19
deslocamento 83, N
desvio condicional Q
detecção automática de paralelismo 29, 33
dinâmica 45
direto do registrador de dados E
direto do registrador de endereços F
DIVS M
DIVU M
domínio original 69
DRV11-WA 51

E

e-lógico M
Efeito Amdahl 43
elípticas 58
endereçamento imediato I
endereçamento indireto, com deslocamento, via PC H
endereçamento indireto, com deslocamento, via registrador de endereços G

endereçamento indireto, com indexação e deslocamento, via PC H
endereçamento indireto, com indexação e deslocamento, via registrador de endereços G
endereço (VME) do nó 83
endereços I
endereços pares G
endereços-destino C
endereços-fonte C
entrada e saída de dados dos nós processadores 52
EOR N
EORI M
equação de Laplace 59, 67, 77
equação de Laplace com condições de contorno numa região bi-dimensional retangular ii
equação de Laplace em forma discretizada 60
equação de Poisson 59
equação de propagação de uma onda unidimensional 58
equação elíptica 59
equação parabólica 59
equações diferenciais parciais 58
equivalence 83
especificação dos nós 56
estática 45
estrutura da rede de interconexão 9
estrutura em blocos 29
estruturas de dados 72
execução 6
execução da instrução 8
EXG K
EXT M
Extend Bit D

F

Fatias de tempo 22
Fermi National Accelerator Laboratory ii
FERMILAB ii
FIFO 22
figuras iv
físicos iii
fluxo 11
fluxo de dados 11, 43
fluxo de instruções 11
fluxo do programa Q
fonte 59
formato dos dados 74, 86
fortemente acoplado 17
FORTRAN 50, B
função CONINI LL, VV, HHH, PPP

G

ganho teórico na velocidade de processamento 37
ganhos em tempo de execução 6
gerações de computadores 3
GET_DONE 74
glossário iv, v
glossário de siglas v
grafo de precedência 30

H

hexadecimal B
hibernação 85
hiperbólicas 58
histograma 36, 37
HOSPEDEIRO VV
HOST SOURCE 54
host time 88

I

imagem digital 36
imediato E
implementação eficiente de um algoritmo 37
incrementos 84
indireto, com pós-incremento, via registrador de endereços G
indireto, com pré-decremento, via registrador de endereços F
indireto, via registrador de endereços F
Instituto de Física Teórica ii, iii
instrução C
instrução de comparação R
instrução incondicional R
instruções lógicas M
integração em grande escala 4
integração em média escala 4
integração em pequena escala 4
interior das matrizes 71
interrupção no hospedeiro 85
intregração em escala muito grande 4
índice v
índice de figuras e tabelas v
índice remissivo v

J

JMP R
JSR R

L

Laboratório de Física Experimental de Altas Energias ii
laços 30, R
largura de banda da memória 17
largura de banda do barramento 22, 36
LEA K
legendas das figuras v
legendas de tabelas v
Lei de Amdahl 44
letras que designam variáveis iv
liberação dos recursos 32
linguagem 29
linguagem *assembly* B
linguagem de alto nível B
linguagem de máquina B
LINK K
listagem do programa que roda nos processadores ACP LL
listagem do programa que roda no computador-hospedeiro VV
listagem do programa que roda no computador-mestre HHH
listagem do programa que roda no computador convencional (MicroVAX II), com apenas 1 processador U
listagem do programa que roda no computador hospedeiro AA
listagem do programa que roda no nós-de-cálculo PPP
língua estrangeira iv, v
localização do dispositivo ao longo do barramento 22
long 82
LSL O
LSR O

M

maior tempo de processamento 93, 94, 95
manipulação de pilha F
melhor desempenho de um algoritmo 38
memória central compartilhada 17, 19
memória compartilhada 9, 17
memória global 46
memória multiporta 24
memória RAM local 56
memória virtual 4
memórias locais 21
memórias multiportas 9
menor tempo de processamento 93, 94, 95
método de Gauss-Seidel 62
método de Jacobi 61
método SOR 62
método SOR com aceleração de Chebyshev 67
métodos de prioridades 22
métodos numéricos 58, 90
microprocessador B
microprocessador MC68020 B
mnemônicos C
modelos iii
modo de endereçamento particular E
momento de parada 81
monitoração 85
MOVE K
MOVEA K
MOVEM K
MOVEQ K
módulos de memória 9
módulos de memória local 9
MULS M
multicomputador 16
multiplicidade de unidades funcionais 6
multiprocessador 9, 16
multiprocessadores 4, 8, 13, 16
Multiprocessadores com Memória Multiporta 26
Multiprocessadores com Rede de Tempo Compartilhado 26
Multiprocessadores com Rede tipo *Crossbar* 26
multiprocessadores fortemente acoplados 17
multiprocessadores fracamente acoplados 17
multiprogramação 4, 6
MULU M

N

não-lógico M
NEG M
Negative Bit D
nil 86
nível de cinza 36
NODE BLOCKS 54, 55, 84
NODE MAIN SUBROUTINE 54
NODE SOURCE 54
node time 88
NOT N
notação iv
nó 17
nó de um sistema de multiprocessadores 18
nó-mestre 77, 82
nós-de-cálculo 81, 82
numeração das figuras v

O

Operações da UCP e de E/S sobrepostas 6

operações matemáticas L
operando-destino E
operando-fonte E
operandos C
OR N
ordenamento ímpar-par 64
organização geral deste trabalho iv
ORI M
ortonormalidade 91
ou-exclusico-lógico M
ou-lógico M
Overflow Bit D

P

palavras E
palavras-longas E
parabólicas 58
paralelismo 32
Paralelismo e *pipeline* dentro da UCP 6
paralelismo espacial 9
paralelismo temporal 8
parâmetro do método SOR 62, 63
parfor 30
partição 36
Pascal B
passagem dos parâmetros para os nós 53
PEA K
pilha D
pipeline 8
placas processadoras 49
política de prioridades 19
ponteiro da pilha D
pontos pares e ímpares 64
potências de 2 N
pós-incremento G
prefácio v
pré-decremento F
Primeira geração de computadores 3
primeira variável do *common block* 84
Prioridade dinâmica 22
prioridades estáticas 22
processador matricial 8, 9, 12
processador-mestre 77
processadores 9
processadores-de-cálculo 81
Processamento Paralelo 2
programa paralelo típico 52
programador em linguagem de alto nível B
programar em linguagem *assembly* B

Q

Quarta geração de computadores 4
Quinta geração de computadores 4

R

razões entre os tempos de execução 97
razões entre os tempos de processamento 96
receber um bloco de dados 73
recolhimento dos resultados 85
reconstrução de eventos 49
rede de interconexão 9
rede de interconexão de E/S 9
Rede de Interconexão de Sinal de Interrupção 20
Rede de Interconexão entre E/S e Processadores 20
Rede de Interconexão entre Processador e Memória 20
rede de interconexão entre processadores e a memória compartilhada 9
rede de interrupção entre processadores 9
rede tipo crossbar 9
redes de interconexão 21
referências vi
registrador de estado D
registrador implícito I
registradores C
registradores de dado C
registradores de endereço C
RESET S
reset S
resolução numérica das equações diferenciais parciais 60
resolver a equação de Laplace nos subdomínios 81
resposta numérica 92
retirar objetos da pilha G
RETURN 53, 86
ROL O
ROR O
rotação N
ROXL O
ROXR O
RTS R
RUN 54

S

seções críticas 31
Segunda geração de computadores 3
SEND_DONE 74
SF R

simulação 50, 52
sinal de mais (+) G
sinal de menos (-) F
sincronização 42, 43, II
sintaxes 73
sistema de memória compartilhada 19
Sistema de Transferência de Mensagens. 17
sistema multiprocessador com comunicação direta entre os processadores 96
sistema multiprocessador sem comunicação direta entre os processadores 96
sistema operacional 16
sistemas hierárquicos de memória 6
solução analítica 90, 91
soluções analítica e numérica 90
soluções numéricas 90
SORCHE LL
Speedup 43, 97
ST R
stack frame K
stack pointer D
STOP S
SUB L, M
SUBA L
subdivisão do domínio 67
subdomínios 67, 69
subdomínios das extremidades 70
subdomínios) das extremidades 71
SUBI L
sublinhamento iv
SUBQ L
sub-rotina balanceamento_de_carga VV, HHH
sub-rotina CALOR PPP
sub-rotina carga AA
sub-rotina cond_iniciais AA
sub-rotina ctes_iniciais U, AA
sub-rotina imprime U, AA, VV
sub-rotina MESTRE HHH
sub-rotina pede_dados VV
sub-rotina poe_A_em_B U
sub-rotina principal, acessada pelo programa do hospedeiro LL
sufixo forn 87
sufixos 87
SWAP K
SYSTEM 54

T

tabelas-verdade N
TAS M
temperatura 91, 92
tempo compartilhado 4
tempo de acesso à memória 36
tempo máximo de espera de um acp_getblock. 88
tempo máximo de execução de programas dos nós ACP 88
tempos de execução 93, 94, 95, 96
Teorema de Taylor 59
Terceira geração de computadores 4
testar e manipular bits individuais O
topo da pilha D
tópicos de interesse especial v
tópicos principais iv
tradução v
transferência de dados K
transferências 81
transmissão comum 53
transmissão de evento 53
transmissão de valores 72
transmitir um bloco de dados 73
TRAP S
trap S
troca de bordas 77
TST M

U

UCP 50
unidade 9 53
unidade de arbitramento 19
UNLK K
UPF 87
User's Parameter File 54, 84, 87

V

valor ótimo 63
valores de contorno 59
valores iniciais 59
vantagem da decomposição dinâmica 45
vantagem da decomposição estática 45
variáveis compartilhadas 31
VAX 11/780 49
velocidade de processamento 36
velocidade de propagação da onda 58
VME Resource Module 52

Z

Zero Bit D

extras

.for1 87
.for2 87

32100 50, 56
68020 50, 56

Curiosidades

A título de curiosidade, apresento aqui alguns dados sobre a confecção deste trabalho:

- ☐ Todos os capítulos, apêndice etc., foram criados no computador Apple Macintosh II, com o editor de textos **Microsoft Word 4.0** e impressos na impressora ImageWriter II, a cores. O tempo de impressão de 1 cópia completa é de cerca de 18 horas

- ☐ A impressão da capa foi feita com um patch (**Adobe Type Manager**) sobre as rotinas do QuickDraw e fontes em formato outline, a fim de se obter maior resolução

- ☐ As fórmulas matemáticas foram criadas com o DA: **Expressionist™ 1.11**

- ☐ As figuras foram criadas pelos programas: **Cricket Draw** e **Mac Draw II**

- ☐ Os dados apresentados em forma de tabelas (no capítulo "Dados e Análises") foram obtidos diretamente (via modem, usando-se o emulador de terminais **VersaTerm PRO**), do MicroVAX II do Laboratório de Física Experimental de Altas Energias do CBPF, no Rio de Janeiro; e foram tabulados e graficados pelo programa **Cricket Graph**

- ☐ Os dados da solução numérica da equação de Laplace (usados na confecção do gráfico 3D do capítulo "Dados e Análises") foram gerados no Macintosh II, por uma versão em **Turbo Pascal** do programa mostrado no apêndice B, e demoraram 264 segundos para serem calculados (usando-se uma matriz 20x20 e precisão igual a 0.01)

- ☐ O gráfico 3D da solução numérica da equação de Laplace foi gerado pelo programa **Mathematica™** e modificado no Cricket Draw

- ☐ Foram usadas 10 fontes diferentes, com tamanhos de 9 até 96 pontos: Helvetica, Times, Courier, **Chicago**, Geneva, Symbol (ω) e a fonte própria da ImageWriter II (nas listagens dos programas)

- ☐ O tamanho total de 1 cópia eletrônica deste trabalho, incluindo texto e figuras, é de 1215 kbytes