PARALELIZAÇÃO DE UM MÉTODO DE REFINAMENTO DE VALORES PRÓPRIOS NUMA ARQUITECTURA DE MEMÓRIA DISTRIBUÍDA

Tese de Mestrado

Faculdade de Engenharia da Universidade do Porto

Outubro de 1995

Aos meus Pais.

PARALELIZAÇÃO DE UM MÉTODO DE REFINAMENTO DE VALORES PRÓPRIOS NUMA ARQUITECTURA DE MEMÓRIA DISTRIBUÍDA

Tese de Mestrado

em

Engenharia Electrotécnica e de Computadores

na Área de

Informática Industrial

Maria Joana Monteiro de Carvalho Peres

(jperes@fe.up.pt)

Tese Orientada pela

Professora Doutora Maria Filomena Dias D'Almeida,

Professora Associada do Departamento de Engenharia Mecânica e Gestão Industrial da Faculdade de Engenharia da Universidade do Porto.

Resumo

As necessidades crescentes de resolver problemas de cada vez maior dimensão levou ao florescimento da computação paralela. Dado que os actuais supercomputadores são bastante dispendiosos e acessíveis a um número reduzido de utilizadores surgiu assim a alternativa de usar uma rede de computadores como uma máquina paralela. Com efeito, existem na maioria das universidades e centros de investigação, estações de trabalho integradas em rede, oferecendo uma quantidade bastante apreciável de poder computacional, o qual não é utilizado em períodos mortos.

Para ilustrar as possibilidades de uma destas redes, nomeadamente uma *'Farm'* constituída por quatro *'DECAlphas AXP'* ligados em rede *FDDI* através dum *'GIGAswitch'*, e usando o sistema de programação *PVM*, tomou-se um programa de cálculo de valores próprios de operadores, por refinamento, que tem a particularidade de ter como núcleo computacional, essencialmente produtos matriz - vector ou matriz - matriz, de grandes dimensões, e que se torna portanto muito adaptado a este tipo de arquitectura.

Esta tese toma como referência uma versão paralela deste problema para máquinas de memória partilhada e adapta-a à arquitectura de memória distribuída acima referida, fazendo de seguida uma análise do desempenho.

Palavras Chave:

Cálculo de Valores Próprios por Refinamento;

Sistemas Paralelos de Memória Distribuída;

PVM.

Agradecimentos

Começo por agradecer à minha orientadora, a Profª. Maria Filomena Dias d'Almeida, do Departamento de Engenharia Mecânica e Gestão Industrial da Faculdade de Engenharia da Universidade do Porto pelo tema actual que me propôs, pelos conhecimentos que me transmitiu, e pelo auxílio e disponibilidade que sempre demonstrou para o fazer.

Queria também deixar aqui expresso o meu profundo reconhecimento à Profª. Lígia Ribeiro e ao Prof. Fernando Nunes Ferreira, responsáveis pelo CICA - Centro de Informática Prof. Correia de Araújo da Faculdade de Engenharia da Universidade do Porto -, por me terem facilitado a utilização dos recursos disponíveis do CICA, e pelo apoio e incentivo necessário que sempre me deram para a prossecução deste trabalho.

Também não queria deixar de agradecer às funcionárias do CICA pela compreensão e carinho demonstrado ao longo do mestrado.

Finalmente, queria agradecer a todos os familiares e amigos que ao longo deste ano sempre me apoiaram e me incentivaram a realizar este trabalho.

Nomenclatura

Termo	*Equação que o Define*	*Significado*
ϕ_n	(2.24)	Base do subespaço invariante maximal de T_n.
ψ_n	(2.25)	Uma base do subespaço invariante maximal do operador adjunto de T_n.
Σ_n	(2.21)	Operador resolvente reduzida de T definido num espaço de Banach.
A_m	(2.23)	A matriz A_m de dimensão m *x* m representa uma aproximação T_m de T.
A_i	–	Bloco de i linhas da matriz A_m, ou bloco de i colunas da matriz A_m, ou bloco de i linhas e i colunas da matriz A_m, conforme o contexto.
A_mX	–	Produto da matriz A_m de dimensão *mxm* pela matriz X de dimensão *ma x ma*.
c_T	(5.10)	Carga total de um dado problema.
c_i	(5.11)	Carga a atribuir ao processador i.
E_p	(5.9)	Eficiência de um programa.
$nlin_i$	(5.12)	Número de linhas da matriz A_m a atribuir ao processador i.
P_n	(2.19)	Projecção espectral do operador T.
S_p	(5.8)	*"Speedup"* de um programa.
T_1	–	Tempo que demora a execução de um programa num só processador.
T	(2.9)	Operador integral de Fredholm.
T_n	(2.23)	Aproximação inicial de Fredholm do operador T.
T_p	–	Tempo que demora a execução de um programa em p processadores.

t_d	–	Tempo de discretização das matrizes A_m e A_n.
t_{dcfny}	(5.7)	Tempo total do refinamento dos valores próprios.
t_e	–	Tempo da equação de Sylvester.
t_{es}	(5.4)	Tempo da entrada de SIGMAN.
t_I	–	Tempo de inicialização do algoritmo.
t_i	(5.5)	Tempo da *i*ésima iteração do algoritmo.
t_p	–	Tempo do produto A_mX.
t_r	(5.3)	Tempo do resíduo.
t_s	–	Tempo de SIGMAN.
t_{ti}	(5.6)	Tempo total das iterações necessárias ao algoritmo.
v_i	–	Velocidade do processador *i*.

Índice

Lista das Figuras

Lista das Tabelas

Capítulo 1

Introdução

Na última década, o computador paralelo excedeu o poder computacional dos supercomputadores vectoriais, tornando-se assim uma importante ferramenta na ciência, na engenharia e na indústria [Chandy91]. A sua popularidade está a crescer, por exemplo, nas seguintes áreas: na ciência dos computadores, em sistemas de base de dados, em análises financeiras. Contudo, o computador paralelo é a excepção e o computador sequencial a norma.

Nesta década, com os avanços do *'software'* paralelo e da tecnologia das comunicações vai ser menos difícil desenvolver programas paralelos eficientes, levando menos programadores a procurar soluções sequenciais.

Uma rede de estações de trabalho pode ser vista como uma máquina paralela potencial oferecendo uma quantidade bastante apreciável de poder computacional que não é utilizado em períodos mortos e que se encontra acessível a uma maior população de utilizadores. De facto, hoje em dia com o desenvolvimento de ambientes de programação em rede, a paralelização de programas em redes de computadores torna-se muito mais simples e acessível.

A necessidade de mais poder computacional também vai estimular o uso do processamento paralelo. Algumas aplicações, como a previsão meteorológica, requerem tanto esforço computacional em tão pouco tempo que até os

supercomputadores mais poderosos são inadequados. À medida que a supercomputação se torna uma grande fatia da investigação científica, a computação paralela vai crescer. Será mesmo, a única opção para certas aplicações.

1.1 - Objectivos

Para ilustrar as possibilidades de uma destas redes, nomeadamente uma *'Farm'* constituída por quatro *'DECAlphas AXP'* ligados em rede *FDDI* através dum *'GIGAswitch'*, e usando o sistema de programação *PVM*, tomou-se um programa de cálculo de valores próprios de operadores, por refinamento, que tem a particularidade de ter como núcleo computacional, essencialmente produtos matriz - vector ou matriz - matriz, de grandes dimensões, e que se torna portanto muito adaptado a este tipo de arquitectura.

Os valores próprios de operadores são em geral obtidos por discretização destes seguida de resolução de um problema de valores próprios matricial. A precisão destes processos depende directamente da dimensão da matriz mas o tempo de cálculo aumenta com ela. Mediante utilização da Análise Funcional foram estabelecidas fórmulas de refinamento de valores próprios aproximados que permitem obter precisões correspondentes ao cálculo de um problema de valores próprios de grande dimensão, sem o resolver directamente, usando apenas a matriz de grande dimensão para produtos matriz - vector ou matriz - matriz, operações altamente paralelizáveis.

Como ponto de partida tomou-se uma versão paralela deste problema para máquinas de memória partilhada. Esta tese adapta esta versão paralela a uma arquitectura de memória distribuída através do sistema de programação *PVM* e faz uma análise do desempenho obtido.

1.2 - Estrutura da Tese

Esta tese encontra-se estruturada do seguinte modo:

O Capítulo 2 começa por apresentar algumas definições e resultados dos espaços de Banach, dos operadores definidos nestes, e da Teoria Espectral, necessários à introdução do método de refinamento de valores próprios. Posteriormente, apresenta

a fórmula de refinamento de valores próprios múltiplos a paralelizar e descreve alguns detalhes do algoritmo que implementa a referida fórmula seguindo a descrição feita num anterior trabalho, concretamente em [Almeida93].

O Capítulo 3 aborda questões relativas à programação paralela em redes de computadores. A primeira parte deste capítulo apresenta os aspectos fundamentais da programação por passagem de mensagens. Posteriormente, descreve sumariamente o *'software' PVM*, o qual se enquadra neste tipo de programação, e que, de entre outros sistemas de programação em redes de computadores, foi o escolhido para a prossecução deste trabalho. O Capítulo 3 termina com um alerta para algumas questões que a programação paralela em redes acarreta.

O Capítulo 4 descreve o ambiente computacional que deu suporte ao trabalho prático realizado nesta tese. Este capítulo começa por descrever as características mais relevantes dos elementos constituintes da *'Farm'*, nomeadamente, dos quatro *'DECAlphas AXP'* e do *'GIGAswitch'*. O Capítulo 4 termina com a apresentação do sistema operativo que equipa as máquinas da *'Farm'* e com a descrição sucinta das bibliotecas utilizadas na implementação do algoritmo: o sistema operativo *DEC OSF/1*, a biblioteca *DXML* de rotinas matemáticas e uma biblioteca de rotinas para medição de tempos de execução de programas.

O Capítulo 5 ilustra as possibilidades da *'Farm'* na paralelização do algoritmo proposto no Capítulo 2, tomando como referência uma versão paralela deste problema para máquinas de memória partilhada. A primeira parte deste capítulo faz um estudo da versão sequencial deste problema seguindo as opções tomadas em [Almeida93]. A segunda parte deste capítulo descreve a estratégia de paralelização pela qual se optou, em função do estudo feito na primeira parte. A terceira parte apresenta os resultados obtidos da paralelização do algoritmo na *'Farm'* e estuda o desempenho obtido a nível de *"speedup"* e eficiência. A quarta parte deste capítulo faz um estudo comparativo do desempenho obtido no algoritmo, a nível de *"speedup"* e eficiência, para várias distribuições dos dados consideradas. Este capítulo termina com uma proposta de optimização do algoritmo através de um esquema de balanceamento de carga.

O Capítulo 6 encerra este texto. Nele se descrevem as conclusões mais relevantes a que se chegou e se sugerem perspectivas de trabalho futuro.

Capítulo 2

Definição do Problema

O objectivo deste trabalho é a paralelização de um programa de refinamento de valores próprios aproximados de operadores [Almeida93], [Vieira91], em ambientes de memória distribuída.

Consideraremos o problema de valores próprios:

$$T\varphi = \lambda\varphi \qquad (2.1),$$

em que T é um operador integral compacto, $\lambda \in \mathbf{C}$, $\varphi \in X$, onde X é um espaço de Banach.

Para descrever o método de refinamento de valores próprios é, pois, necessário introduzir algumas definições e resultados matemáticos.

2.1 - Definições e Resultados Preliminares

2.1.1 - Espaços de Banach

Seja E um espaço vectorial sobre um corpo **K** real ou complexo.

DEFINIÇÃO 2.1: Um *operador linear* é uma transformação linear de E em E.

DEFINIÇÃO 2.2: Uma *norma* em E é uma função $\|.\|: E \to \mathrm{R}^+$ que satisfaz as seguintes condições:

- $\forall x \in E, \|x\| = 0 \Leftrightarrow x = 0$ (2.2);
- $\forall x \in E \ \forall \alpha \in \mathrm{K}, \|\alpha x\| = |\alpha| \cdot \|x\|$ (2.3);
- $\forall x, y \in E, \|x + y\| \leq \|x\| + \|y\|$ (2.4).

DEFINIÇÃO 2.3: Um *produto interno* em E é uma função de duas variáveis (x,y), definidas por:

- $\forall x, y, z \in E \ \forall \alpha, \beta \in \mathrm{K}, (\alpha x + \beta y, z) = \alpha(x, z) + \beta(y, z)$ (2.5);
- $\forall x, y \in E (x, y) = \overline{(y, x)}$ (2.6);
- $\forall x \in E, (x, x) = 0 \Leftrightarrow x = 0$ (2.7).

DEFINIÇÃO 2.4: Uma sucessão $(x_n)_{n \in \mathrm{N}}$ é de Cauchy se e só se

$$\forall \varepsilon > 0 \ \exists n_0 \in \mathrm{N} \ \forall m, n \in \mathrm{N}: m, n \geq n \Rightarrow \|x_m - x_n\| < \varepsilon \quad (2.8).$$

DEFINIÇÃO 2.5: Um *espaço vectorial normado* é um espaço vectorial sobre o qual está definida uma norma.

Um *espaço completo* é um espaço vectorial em que toda a sucessão de Cauchy é convergente.

Um *espaço de Banach*, X, é um espaço normado completo.

Um *espaço de Hilbert*, H, é um espaço completo com um produto interno e em que a norma deriva do produto interno.

RESULTADO 2.1: Num espaço de Banach um subconjunto é completo se e só se é fechado.

Num espaço linear normado qualquer subespaço de dimensão finita é completo.

2.1.2 - Operadores Definidos em Espaços de Banach

Seja T um operador definido num espaço de Banach complexo. Seguidamente apresentaremos algumas definições:

Sejam X e Y dois espaços de Banach e S um subconjunto de X.

DEFINIÇÃO 2.6: Um operador T diz-se *compacto* se e só se transforma cada subconjunto limitado de X num subconjunto relativamente compacto de Y com fecho topológico compacto.

DEFINIÇÃO 2.7: O *espaço dos operadores lineares contínuos com domínio X e valores em Y* é denotado por $\mathcal{L}(X, Y)$. Se Y coincidir com X, denota-se apenas por $\mathcal{L}(X)$.

RESULTADO 2.2: $T \in \mathcal{L}(X, Y)$ é compacto se e só se a imagem $\{Tx_n\}$ de qualquer sucessão limitada $(x_n)_{n\in\mathbb{N}}$ de X tem uma subsucessão convergente em Y.

RESULTADO 2.3: Operadores lineares contínuos em espaços de Banach de dimensão finita são compactos.

RESULTADO 2.4: Seja X um espaço de Banach e T um operador compacto de X em X. Então, para cada valor próprio λ de T só existe um número finito de vectores próprios linearmente independentes.

DEFINIÇÃO 2.8: O *operador identidade* em X é denotado por I_X.

DEFINIÇÃO 2.9: O *operador integral de Fredholm T*, definido em $X = C[a, b]$ e de núcleo K: $[a, b] \times [a, b] \rightarrow \mathbf{C}$ é um operador definido por:

$$\forall x \in C[a, b]\ \forall t \in [a, b],\ Tx(t) = \int_a^b K(t,s)x(s)ds \qquad (2.9).$$

Se o núcleo $K(t,s)$ é contínuo no seu domínio então o operador integral de Fredholm é compacto.

DEFINIÇÃO 2.10: Uma *forma semilinear* é uma função $f: X \rightarrow \boldsymbol{C}$ tal que:

$$\forall x, y \in X\ \forall \alpha, \beta \in C, f(\alpha x + \beta y) = \overline{\alpha} f(x) + \overline{\beta} f(y) \qquad (2.10).$$

O *espaço adjunto* de X, denotado por X^*, é o conjunto das formas semilineares limitadas sobre X.

DEFINIÇÃO 2.11: A forma semilinear $f(x)$ identifica-se com o *produto escalar* definido em X^* x X e denotado por $<f, x>$. Este designa-se por forma sesquilinear, isto é, é uma função linear na primeira variável e semilinear na segunda.

Se X for um espaço de Hilbert então X^* coincide com X e o produto escalar $<,>$ com o produto interno $(,)$ de espaço de Hilbert.

DEFINIÇÃO 2.12: Define-se operador adjunto $T^*: X \to X$ por:

$$\forall x \in X \forall y \in X^*, \langle Tx, y\rangle = \langle x, T^* y\rangle \tag{2.11}$$

RESULTADO 2.5: Se $T = T^*$ então todos os valores próprios são reais e os correspondentes vectores próprios formam um conjunto ortonormal.

Seja $(T_n)_{n\in\mathbb{N}}$ uma sucessão de operadores lineares pertencentes a $\mathcal{L}(X, Y)$, onde X e Y são dois espaços vectoriais normados.

DEFINIÇÃO 2.13: Diz-se que, [Chatelin83], $(T_n)_{n\in\mathbb{N}}$ *converge para T de modo pontual* e representa-se por $T_n \longrightarrow T$, se e só se:

$$T_n \longrightarrow T \Leftrightarrow \lim_{n\to+\infty} T_n x = Tx, \ \forall x \in X \tag{2.12}.$$

DEFINIÇÃO 2.14: Diz-se que, [Chatelin83], $(T_n)_{n\in\mathbb{N}}$ *converge para T de modo uniforme ou em norma* e representa-se por $T_n \xrightarrow{u} T$, se e só se:

$$T_n \xrightarrow{u} T \Leftrightarrow \lim_{n\to+\infty} \|T_n - T\| = 0 \tag{2.13}.$$

DEFINIÇÃO 2.15: Diz-se que, [Chatelin83], $(T_n)_{n\in\mathbb{N}}$ *converge para T de modo colectivamente compacto* e representa-se por $T_n \xrightarrow{cc} T$, se e só se:

- $T_n \longrightarrow T$, isto é, T_n converge para T de modo pontual;
- e o conjunto $\bigcup_{n=1}^{+\infty} (T - T_n)C$, onde C é uma bola unitária em X, tem fecho compacto em X.

2.1.3 - Definições da Teoria Espectral

DEFINIÇÃO 2.16: O *conjunto resolvente* do operador T é denotado por $\rho(T)$ e definido por:

$$\rho(T) = \left\{ z \in \mathbf{C} : (T - z\mathrm{I}_X)^{-1} \in \mathcal{L}(X) \right\} \qquad (2.14).$$

O espectro de T, $\sigma(T)$, é o conjunto complementar em $\mathbf{C}$ de $\rho(T)$ [Chatelin83].

DEFINIÇÃO 2.17: O *operador resolvente* é definido por:

$$R(T,z) = (T - z\mathrm{I}_X)^{-1}, \text{ para } z \in \rho(T) \qquad (2.15).$$

$R(T,z)$ tem domínio X e contradomínio Dom$T$ para $z \in \rho(T)$. Quando não houver ambiguidade denota-se apenas por $R(z)$ [Chatelin83].

DEFINIÇÃO 2.18: O *raio espectral* do operador T , denotado por $r_\sigma(T)$, é definido por [Chatelin83]:

$$r_\sigma(T) = \inf\left\{ \left\| T^k \right\|_X^{1/k} : k \in \mathbf{N} \right\} \qquad (2.16).$$

DEFINIÇÃO 2.19: Um operador limitado $P \in \mathcal{L}(X)$ é uma *projecção* se e só se $P^2 = P$. Associado a P temos a decomposição $X = M \oplus N$ onde M e N são subespaços fechados de X e definidos por $M = PX = \mathrm{Ker}(I_X - P)$ e $N = \mathrm{Ker}P = (I_X - P)X$.

RESULTADO 2.6: Existe uma e uma só projecção sobre M segundo N tal que $\forall x \in N\ P(x) = 0$ e $\forall x \in M\ P(x) = x$. Tal projecção é ortogonal se e só se

$$\|x - Px\|_2 = \min_{y \in M} \|x - y\|_2 \qquad (2.17).$$

DEFINIÇÃO 2.20: Seja $\overline{\lambda}$ um valor próprio de T^*, então $\exists \psi \neq 0 : T^* \psi = \lambda \psi$, onde ψ é um vector próprio de T^* associado a $\overline{\lambda}$, ou seja, $\psi^H T = \lambda \psi^H$. Ao vector ψ chama-se *vector próprio à esquerda de T associado a* λ.

DEFINIÇÃO 2.21: A *multiplicidade algébrica* de um valor próprio λ, denotada por *ma*, é a multiplicidade de λ como raiz do polinómio característico.

DEFINIÇÃO 2.22: O *espaço próprio*, denotado por E_λ, é o conjunto dos vectores próprios associados ao valor próprio λ mais o vector nulo, isto é,

$$E_\lambda = \ker(T - \lambda I) = \{x \in X : (T - \lambda I)x = 0\} \qquad (2.18).$$

DEFINIÇÃO 2.23: A *multiplicidade geométrica* de um valor próprio λ, denotada por *mg*, é a dimensão do espaço próprio a ele associado.

DEFINIÇÃO 2.24: A *projecção espectral* do operador T associada ao valor próprio λ é definida por [Chatelin83]:

$$P = \frac{-1}{2\pi i}\int_{\Gamma(\lambda)} R(z)\,dz,\ P \in \mathcal{L}(X) \qquad (2.19).$$

DEFINIÇÃO 2.25: O *subespaço invariante maximal* do operador T associado ao valor próprio λ, é denotado por M e é definido por:

$M = PX$.

M é tal que dim M = dim PX = ma.

DEFINIÇÃO 2.26: O *operador resolvente reduzida* é definido por [Chatelin83], [Kato66]:

$$S = S(\lambda) = \lim_{z \to \lambda} R(z)(I - P) \qquad (2.20),$$

donde:

$$S = \frac{1}{2\pi i}\int_{\Gamma(\lambda)} \frac{R(z)\,dz}{z - \lambda} \qquad (2.21).$$

2.2 - Método de Refinamento de Valores Próprios

Dado o problema de valores próprios (2.1) em que T é um operador integral de Fredholm compacto, o programa que se pretende aqui paralelizar em ambientes de memória distribuída, calcula uma aproximação incial pelo método de Fredholm e refina-a segundo a fórmula baseada na correcção do resíduo que foi estabelecida por M. Ahues, F. d'Almeida e M. Telias, inicialmente para valores próprios simples [Ahues82].

A comparação deste método com outros métodos de refinamento de valores próprios deste tipo de operadores, detalhes de implementação e análise de custos encontra-se em [Almeida84]. A implementação para o caso de valores próprios múltiplos mencionada nos trabalhos anteriores requer cuidados adicionais nomeadamente a necessidade de trabalhar com uma base do subespaço invariante maximal em vez dos vectores próprios conforme descrito em [Ahues90]. Essa implementação foi efectuada por M. I. Vieira [Vieira91] para aproximações iniciais obtidas por diversos métodos (Galerkin, Fredholm, Nystron, Sloan). Posteriormente F. d'Almeida [Almeida93] fez a paralelização da fórmula de refinamento que vai ser estudada aqui no caso de se partir de uma aproximação inicial de Nystron (o caso de se partir de uma aproximação inicial de Fredholm é muito semelhante). Seguindo a descrição feita no artigo referido a fórmula a paralelizar é:

$$\begin{cases} \xi_0 = \phi_n \\ \chi_k = (T\xi_k)(< T\xi_k, \psi_n >)^{-1}, k = 0,1,\ldots \\ \xi_{k+1} = \chi_k - \Sigma_n F(\chi_k) \end{cases} \qquad (2.22),$$

em que:

- T_n é uma aproximação inicial de Fredholm definida por:

$$T_n x = \sum_{i=1}^{n} \left(\sum_{j=1}^{n} \left(w_j K(s_i, s_j) x(s_j) \right) e_i \right) \qquad (2.23),$$

  onde $\{e_i\}$ é a base das funções chapéu do subespaço X_n das funções contínuas parcelarmente lineares correspondentes a uma partição de [0,1], $\{w_j\}_j$ e $\{s_j\}_j$, j = 1, ..., n os pesos e os pontos duma regra de quadratura aproximada. A matriz $w_j K(s_i, s_j)$ é a chamada matriz de Fredholm.

- ϕ_n é uma base do subespaço invariante maximal de T_n;

- ψ_n é uma base do subespaço invariante maximal do operador adjunto de T_n;

- Σ_n é o operador resolvente reduzida definido em X (ver Definição 2.26) cujo resultado será calculado pela rotina SIGMAN (ver [Almeida93]);

- $F(x)$ é tal que $F(x) = Tx - x < Tx, \psi_n >= 0$.

O algoritmo que vamos paralelizar só difere do referido em [Almeida93] no cálculo da projecção espectral $P_n x$ e de $\sum_n w$, dados x e w, que são realizados pelas rotinas PNX e SIGMAN. As respectivas fórmulas no caso de Fredholm são semelhantes às de Nystron por isso transcrevemos aqui a parte do artigo [Almeida93] referente a Detalhes de Algoritmo com as modificações relativas ao cálculo de PNX e SIGMAN.

2.2.1 - Detalhes do Algoritmo

O algoritmo desenvolvido por [Almeida84] para calcular este método, na prática, usa uma discretização de T representada pela matriz A_m de ordem $m >> n$ para substituir o operador T na expressão (2.22). Foi provado em [Almeida84] que, neste caso, as iterações vão convergir para os elementos próprios de T_m. Logo, é importante utilizar um valor de m grande para obter uma boa aproximação dos valores próprios de T, depois do refinamento iterativo.

O algoritmo começa por calcular duas discretizações iniciais:

- a matriz A_m de dimensão m x m que representa uma aproximação T_m de T;
- a matriz A_n de dimensão n x n que representa uma aproximação T_n de T.

Seguidamente, são calculados os valores próprios μ_1, μ_2, ..., μ_{ma} pelo método QR. O vector u_n vai conter ma colunas que correspondem aos vectores próprios à direita associados a μ_1, μ_2, ..., μ_{ma} e v_n contém ma colunas que correspondem aos vectores próprios à esquerda associados a μ_1, μ_2, ..., μ_{ma}.[1]

A seguir, são calculados os prolongamentos ϕ_n e ψ_n de u_n e v_n, respectivamente, ao subespaço de dimensão m tal que:

$$\phi_n = p\, u_n \qquad (2.24),$$

e

$$\psi_n = r^* v_n \qquad (2.25),$$

[1]Se $ma = 1$, λ_n, u_n, v_n, são o valor próprio, o vector próprio à direita e o vector próprio à esquerda, respectivamente.

onde p é o operador prolongamento do subespaço X_n no subespaço X_m e r o operador restrição do subespaço X_m no subespaço X_n.

A iteração começa por determinar uma base actualizada do subespaço invariante e por sua vez um λ_n actualizado de acordo com a expressão (2.22). A iteração usa dois subalgoritmos PNX e SIGMAN para calcular a projecção espectral P_n e a resolvente reduzida Σ_n.

Seja X o espaço de todas as funções contínuas definidas em [0,1] e $\pi_n : X \rightarrow X_n$ uma projecção de interpolação no subespaço X_n das funções contínuas parcelarmente lineares correspondentes a uma partição de [0,1].

O cálculo de $P_n x$ é dado por $\phi_n < rx, v_n >$ e o valor $y = \Sigma_n w$ é dado pela resolução da seguinte equação:

$$(I - P_n)T_n y - y\theta_n = (I - P_n)w = w - \phi_n < rw, v_n > \qquad (2.26),$$

com $\theta_n = diag(\mu_1, \mu_2, ..., \mu_{ma})$, que pode ser representada esquematicamente por:

$$\begin{matrix}(n\,\text{linhas}) \\ ((m-n)\,\text{linhas})\end{matrix}\begin{bmatrix}\pi_n(I-P_n)T_n & 0 \\ 0 & 0\end{bmatrix}*\begin{bmatrix}\pi_n y \\ (I-\pi_n)y\end{bmatrix}-\begin{bmatrix}\pi_n y \\ (I-\pi_n)y\end{bmatrix}*\begin{bmatrix}\theta_n\end{bmatrix} =$$

$$= \begin{bmatrix}\pi_n w \\ (I-\pi_n)w\end{bmatrix} - \begin{bmatrix}\pi_n\phi_n < rw, v_n > \\ (I-\pi_n)\phi_n < rw, v_n >\end{bmatrix} \qquad (2.27),$$

donde resulta o seguinte sistema de equações:

$$\begin{cases}\pi_n(I-P_n)T_n\pi_n y - \pi_n y\theta_n = \pi_n w - \pi_n\phi_n < rw, v_n > \\ -(I-\pi_n)y\theta_n = (I-\pi_n)w - (I-\pi_n)\phi_n < rw, v_n >\end{cases} \qquad (2.28).$$

A primeira equação deste sistema é uma equação de Sylvester da forma:

$$AX - XB = C \qquad (2.29)^2,$$

onde:

- $A = (I - P_n)T_n$

[2] A é uma matriz de dimensão n x n e B de dimensão ma x ma.

- $B = \theta_n$
- $C = \pi_n w - \pi_n \phi_n < rw, v_n >$

que vai ter como solução $\pi_n y$. A partir da segunda equação obtém-se $(I - \pi_n)y$, sendo assim possível determinar o valor de y, visto que:

$$y = \pi_n y + (I - \pi_n) y \qquad (2.30).$$

2.3 - Síntese

Este capítulo introduziu algumas definições e resultados dos espaços de Banach, dos operadores definidos nestes espaços, e da Teoria Espectral, necessários à introdução do método de refinamento de valores próprios múltiplos. Posteriormente descreveu-se sucintamente este método de refinamento e foi apresentada a fórmula de refinamento a paralelizar. Por fim, descreveram-se os detalhes do algoritmo que a implementa seguindo a descrição feita em [Almeida93].

Ficou assim definido o problema a tratar nesta tese, o qual começará a ser abordado no próximo capítulo.

Capítulo 3

Computação Paralela em Redes de Computadores

As necessidades crescentes de resolver problemas de cada vez maior dimensão levou ao florescimento da computação paralela. No entanto, os actuais supercomputadores são bastante dispendiosos e acessíveis a um número restrito de utilizadores. Por outro lado, existem na maioria das universidades e centros de investigação, estações de trabalho integradas em rede, oferecendo uma quantidade bastante apreciável de poder computacional. Além disso, este poder computacional não é, em geral, utilizado à noite ou em períodos mortos durante o dia. Essa rede de computadores pode ser vista como uma máquina paralela potencial.

As estações de trabalho, sendo de uso geral, podem trabalhar independentemente ou cooperar entre si na execução de uma aplicação paralela. Para aquelas aplicações essencialmente paralelas e de computação intensiva que correm em estações de trabalho, pode-se prever que paralelizá-las leva a melhores desempenhos sem os inconvenientes e as dificuldades que a mudança para um ambiente de desenvolvimento diferente e normalmente muito menos amigável, inevitavelmente acarreta.

Já existe muito trabalho realizado em computadores de memória comum cuja paralelização em memória distribuída exige um esforço adicional visto que a

comunicação entre processadores tem que ser feita explicitamente por passagem de mensagens.

De facto, hoje em dia, com o desenvolvimento de ambientes de programação em rede, sobretudo baseados no modelo de passagem de mensagens - tais como, por exemplo, o *PVM* [Sunderam90], *'Linda'* [Carriero89], *'Express'* [Flower91], *P4* [Butler94], *MPI* [MPIF94] -, a paralelização de programas em redes de computadores torna-se muito mais simples. Mais ainda, dado que alguns destes sistemas são do domínio público eles são acessíveis a qualquer programador. Muitos outros sistemas com características semelhantes aos citados existem. A panorâmica oferecida em [Turcotte93] é um bom ponto de partida para os leitores com interesses nesta matéria.

O sistema de passagem de mensagens adoptado no âmbito do trabalho que esta tese expõe, foi o *PVM*. E isto porque o *PVM* é um sistema robusto e garante um elevado grau de portabilidade [Sunderam94]. Em [Sukup94] faz-se uma comparação bastante interessante entre os sistemas atrás referidos (excepto o *MPI* pois não estava desenvolvido até à data), onde, do ponto de vista do autor, o *PVM* é o mais conveniente em vários aspectos, nomeadamente, a nível de instalação e escrita de programas.

Por outro lado, por ter sido adoptado em numerosas instituições espalhadas pelo mundo, o *PVM* tornou-se quase um *'standard'* de programação paralela. Para além, da qualidade cientifica que já é costume esperar em *'software'* do domínio público, este fenómeno deve-se, em grande parte, ao facto do *'software' PVM* e a respectiva documentação serem suportadas por uma equipa de trabalho muito competente, o que faz com que seja de uma qualidade profissional comparável à do *'software'* comercial. Esta equipa é formada por Jack Dongarra e Al Geist do Laboratório Nacional de *'Oak Ridge'*, Vaidy Sunderam da Universidade de *'Emory'*, Robert Mancheck e Weicheng Jiang da Universidade de *'Tennessee'*, e Adam Beguelin da Universidade de *'Carnegie Mellon'*.

Este capítulo é fundamentalmente dedicado a programação paralela por passagem de mensagens em redes de computadores, sendo composto por três partes: a primeira, introduz os conceitos mais importantes no âmbito da programação por passagem de mensagens; a segunda, descreve sumariamente o sistema *PVM*; a última,

alerta para alguns cuidados a observar no âmbito da programação paralela em redes de computadores.

3.1 - Programação por Passagem de Mensagens

Num ambiente de rede, dado que os processadores não partilham o mesmo espaço de memória, a comunicação entre processos adjudicados a processadores de máquinas distintas só pode ser efectuada pela troca de mensagens.

Entende-se que um processo é basicamente um programa a correr, isto é, é uma entidade activa, controlada por um programa e que necessita de um processador para poder executar-se [Marques90].

Existem diversos conjuntos de primitivas que dão suporte à comunicação remota entre processadores. Os mais comuns baseiam-se em ***passagem de mensagens*** - *"Message Passing"* - ou em ***chamadas a procedimentos remotos*** - *"Remote Procedure Call"*.

A passagem de mensagens entre processos é uma extensão da comunicação entre processadores no âmbito dos sistemas centralizados. A chamada de um procedimento remoto corresponde a uma evolução do modelo de passagem de mensagens. Isto é, do ponto de vista do cliente, a maioria das interacções com outros processos resume-se ao envio de uma mensagem, solicitando um serviço remoto, seguida de uma espera pela mensagem de resposta. Estas operações acrescidas da formatação e desformatação da mensagem de resposta, podem ser encapsuladas no interior de uma rotina que, do ponto de vista do programador, poderá ser chamada como qualquer outra. A única diferença, é que esta tem a particularidade de ser executada não no espaço de endereçamento local, mas sim no espaço de outro processo [Marques90].

Comparando o modelo de programação por passagem de mensagens com o modelo de programação por chamada a procedimentos remotos, o primeiro tem a importante vantagem de apresentar um maior grau de flexibilidade no sentido em que não esconde os mecanismos da troca de mensagens, alargando assim o alcance das opções de comunicação disponíveis a um programador de aplicações. O segundo tem a vantagem de se inserir com toda a naturalidade na metodologia habitual de

programação mas ao esconder do programador todos os detalhes das mensagens limita o paralelismo. Portanto, o modelo normalmente adoptado para paralelizar aplicações é o modelo por passagem de mensagens.

3.1.1 - Conceitos Gerais

O paradigma da programação por passagem de mensagens consiste em resolver um dado problema à custa de vários processos, os quais residem em processadores distintos e comunicam entre si por mensagens - Figura 3.1.

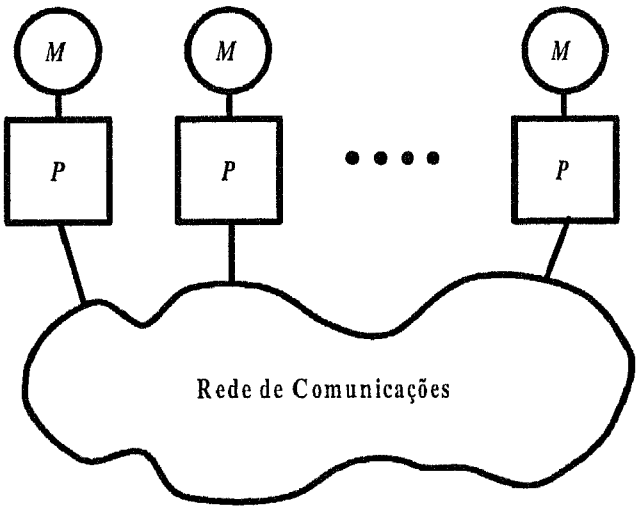

Figura 3.1 - Paradigma da programação por passagem de mensagens.

Daí o ponto essencial do paradigma de programação por passagem de mensagens: é que os processos comunicam entre si por troca de mensagens. Para os programas que correm em cada processador, as operações de passagem de mensagens são simples chamadas a rotinas.

O paradigma da programação por passagem de mensagens tornou-se bastante popular nos tempos que correm. Uma das razões para este facto, é o grande número de plataformas que suportam este tipo de modelo de programação. Programas escritos no estilo de passagem de mensagens podem correr em multiprocessadores de memória distribuída ou partilhada, redes de estações de trabalho ou mesmo em máquinas uniprocessador. A principal vantagem de se contar com este paradigma, é que o programador sabe que ao evoluir de um sistema sequencial para um sistema

distribuído, os seus programas e algoritmos são, em princípio, portáveis, para qualquer arquitectura que suporte o modelo de programação por passagem de mensagens.

Tipicamente, os programas desenvolvidos sobre uma plataforma do tipo de passagem de mensagens são organizados mediante um de dois modelos: o modelo *SPMD* - *"Single Program Multiple Data"* - e o modelo *MPMD* - *"Multiple Program Multiple Data"*. No modelo *SPMD* existe apenas um só programa que corre em todos os processadores. No modelo *MPMD*, um conjunto de programas escravos executa trabalho para um ou mais programas mestre.

3.1.2 - Modelo *SPMD*

No modelo *SPMD* todos os processadores executam em simultâneo o mesmo programa, porém, sobre dados diferentes - Figura 3.2.

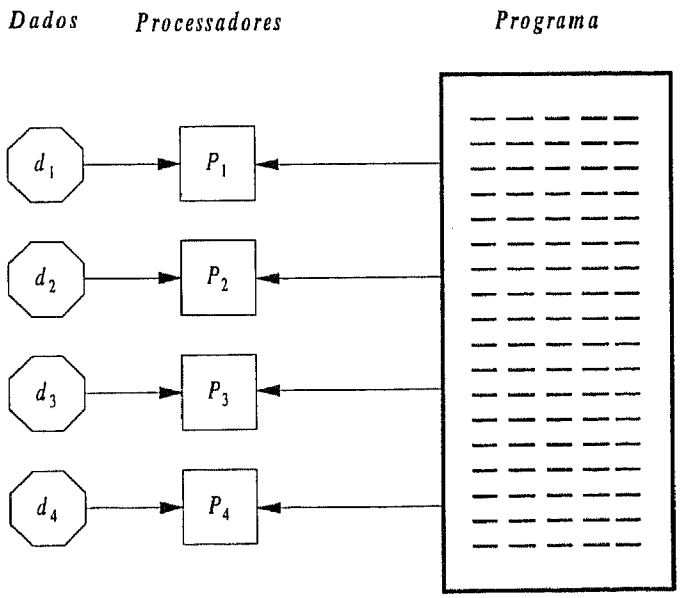

Figura 3.2 - Modelo *SPMD*.

Na prática, o programador pode alterar esta perspectiva adjudicando acções particulares a um ou mais processadores mediante *'ifs'*. Por exemplo, é possível ter-se um processo ***controlador*** que faz uma tarefa diferente (ler, verificar e distribuir dados iniciais) da de um processo ***trabalhador*** que poderá, por exemplo, efectuar cálculos com os dados que recebeu - Figura 3.3.

```
PROGRAM
IF (processo .eq. processo controlador) THEN
        CALL CONTROLADOR (argumentos)
ELSE
        CALL TRABALHADOR (argumentos)
ENDIF
END
```

Figura 3.3 - Extracto do código de um programa baseado no modelo *SPMD*.

3.1.3 - Modelo *MPMD*

O modelo *MPMD*, também chamado modelo mestre/escravo, tem por base um programa distinto a correr em cada processador - Figura 3.4.

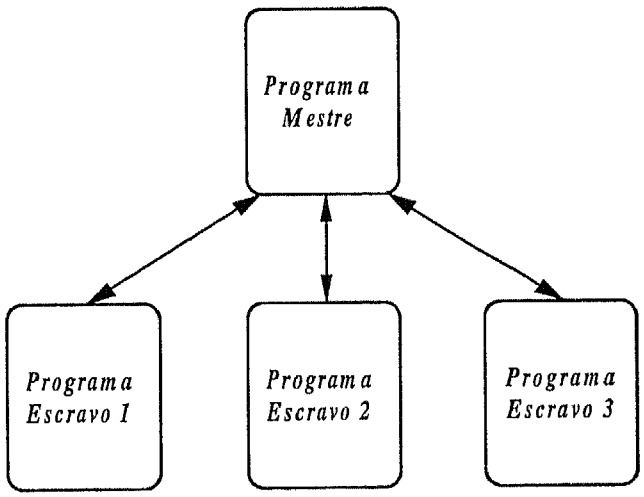

Figura 3.4 - Modelo *MPMD*.

Estes operam sobre os diferentes dados que lhes foram distribuídos. Quando, para resolver um determinado problema é possível dividi-lo em vários subproblemas independentes, é teoricamente possível construir um programa segundo este modelo. Na prática, podemos ter uma situação análoga ao exemplo anterior; contudo, em vez de termos programas iguais, temos programas diferentes: um ou mais programas mestres que correspondem a processos controladores e um ou mais programas escravos que correspondem a processos trabalhadores.

3.1.4 - Mensagens

A transmissão de uma mensagem ocorre quando há interesse em que os valores de variáveis de um programa passem para variáveis de outro programa. A mensagem consiste num conjunto de parâmetros nos quais se contam os dados a trocar. Para o sistema de passagem de mensagens ser capaz de fazer este transporte, é necessário especificar o que encerra o envio de uma mensagem. Regra geral, a seguinte informação tem de ser entregue ao sistema de passagem de mensagens [MacDonald94]:

- Que processador pretende enviar a mensagem;
- Onde estão os dados na memória afecta ao processador emissor;
- Que tipo de dados interessa enviar;
- Quantos dados interessa enviar;
- A que processador(s) se destina a mensagem;
- Onde deverão ficar guardados os dados no(s) processador(es) receptor(es);
- Que quantidade de dados está o processador receptor preparado para receber.

Além de entregar dados, o sistema de passagem de mensagens tem que providenciar alguma informação sobre o progresso das comunicações entre processos. Isso permite verificar se tudo corre como o esperado, permitindo ainda desencadear acções de excepção caso isso não se verifique.

No envio de uma mensagem estão envolvidos três mecanismos [Tanenbaum92]:

- o ***acesso*** - Para um programa enviar uma mensagem é necessário sinalizar a sua interacção junto do sistema de passagem de mensagens;
- o ***endereço*** - Todas as mensagens têm de ser endereçadas. Significa isto que todas elas têm que levar consigo alguma informação anexa por forma a que o sistema de passagem de mensagens saiba a quem as deve entregar;

- a ***recepção*** - Para que a recepção de uma mensagem seja feita de uma forma correcta é importante que o processo receptor tenha capacidade para a receber. Assim, se por exemplo, o *'buffer'* onde vão ser colocados os dados recebidos é mais pequeno que a mensagem enviada, o sistema de passagem de mensagens pode truncá-la ou rejeitá-la.

3.1.5 - Comunicação Ponto-a-Ponto

A forma mais simples de transmitir dados é a chamada ***comunicação ponto-a-ponto***. Este tipo de comunicação envolve sempre dois únicos processos. A mensagem é enviada dum processo emissor para um processo receptor, tendo apenas estes dois processos necessidade de conhecer a existência desta mensagem.

O envio de uma mensagem pode interactuar com a execução dum subprograma de várias maneiras. Concretamente, se o envio de mensagens é ***síncrono*** ou ***assíncrono***, e também se o envio de mensagens é ***bloqueante*** ou ***não-bloqueante***.

A comunicação em modo ***síncrono*** implica o conhecimento por parte do processo emissor que a mensagem foi recebida pelo processo receptor; isto é, a comunicação não termina enquanto a mensagem não for recebida. Na comunicação em modo ***assíncrono*** apenas há conhecimento que a mensagem foi enviada.

As operações de comunicações ***bloqueantes*** retornam da chamada à respectiva subrotina apenas quando a operação termina. A operações ***não-bloqueantes*** retornam logo após a chamada da subrotina, possibilitando assim que o subprograma prossiga de imediato. Mais tarde, o subprograma pode testar do término da operação não-bloqueante.

3.1.6 - Comunicação Colectiva

A ***comunicação colectiva*** distingue-se da comunicação ponto-a-ponto pelo facto de envolver sempre um grupo de processos. Entende-se por comunicação colectiva uma operação que requer a cooperação de todos os processos do mesmo

grupo. Estes tipos de operações colectivas existem na maior parte dos sistemas de passagem de mensagens.

Em termos gerais existem três tipos de operações colectivas [MacDonald94]:

- ***barreira*** - Esta operação sincroniza os processos. A barreira não envolve qualquer troca de informação; apenas bloqueia o processo que a chama até todos os membros do grupo terem chamado a rotina *barreira*;

- ***difusão*** - Esta operação é uma comunicação do tipo um-para-muitos. Um processo envia uma mensagem para um grupo de processos com uma única operação;

- ***operações de aglutinação*** - Uma operação de aglutinação caracteriza-se por tomar itens de dados de vários processos e os reduzir a um único item, o qual fica normalmente disponível a todos os participantes do grupo. Por exemplo, se tivermos vários valores espalhados pelos processos, podemos somá-los através de uma única operação de aglutinação.

Todas estas operações podem ser construídas a partir de primitivas de comunicações ponto-a-ponto. Porém, é preferível utilizar rotinas próprias para o efeito, caso estas se encontrem disponíveis, dado que em princípio estarão optimizadas.

3.2 - O Sistema *PVM*

Um dos sistemas de programação por passagem de mensagens mais utilizado e difundido é o *PVM - "Parallel Virtual Machine"*.

O *PVM* é uma ferramenta de programação em rede ou em multiprocessadores para processamento paralelo. Este sistema foi desenvolvido por Jack Dongarra e Al Geist do Laboratório Nacional de *'Oak Ridge'*, Vaidy Sunderam da Universidade de *'Emory'*, Robert Mancheck e Weicheng Jiang da Universidade de *'Tennessee'*, e Adam Beguelin da Universidade de *'Carnegie Mellon'* [Geist94a].

O *PVM* é um conjunto integrado de ferramentas de *'software'* e de bibliotecas que emulam um ambiente de computação concorrente de uso geral, flexível e

heterogéneo, a partir de uma rede de computadores de arquitecturas diversas. O principal objectivo do sistema *PVM* é o de permitir que este conjunto de computadores possa ser usado cooperativamente para programação concorrente ou paralela. Este conjunto de computadores heterogéneo, que emula uma máquina paralela de memória distribuída, é designado por *máquina virtual*, enquanto que os seus elementos constituintes são designados por *nós*.

Os princípios em que o *PVM* se baseia são os seguintes [Geist94b]:

- ***Existência de uma Tabela de nós configurável pelo utilizador*** - As tarefas computacionais da aplicação correm num conjunto de máquinas seleccionadas pelo utilizador para uma determinada corrida do programa escrito em *PVM*. Desta tabela de nós podem fazer parte computadores uniprocessador ou multiprocessador, incluindo computadores de memória partilhada e distribuída. Esta tabela de nós pode ser alterada dinamicamente por adição ou remoção de máquinas (uma importante característica no âmbito da tolerância a falhas);

- ***Acesso transparente ao 'hardware'*** - Os programas podem ver o '*hardware*' como simples elementos de processamento, ou então explorar as diferentes capacidades de cada máquina inscrita na tabela de nós. Esta segunda hipótese permite atribuir determinadas tarefas às máquinas mais apropriadas à realização das mesmas;

- ***Computação baseada em processos*** - A unidade de paralelismo em *PVM* é a *tarefa*; a maior parte das vezes coincidente com um processo em *UNIX* mas nem sempre. Uma tarefa é um segmento de código sequencial em que a comunicação e a computação se alternam. A atribuição de uma única tarefa a um processador não é forçosa em *PVM*; múltiplas tarefas podem ser executadas no mesmo processador;

- ***Modelo explícito de passagem de mensagens*** - No âmbito do *PVM*, conjuntos de tarefas cooperam entre si pelo envio e recepção de mensagens explícito. A extensão das mensagens é limitada apenas pela quantidade de memória disponível;

- ***Heterogeneidade*** - O sistema *PVM* suporta heterogeneidade em termos de máquinas, redes e aplicações. Relativamente à passagem de mensagens, o *PVM* permite que as mensagens encerrem diferentes tipos de dados e possam ser trocadas entre máquinas com diferentes tipos de representação de dados;

- ***Suporte de Multiprocessadores*** - O *PVM* utiliza as facilidades nativas de passagem de mensagens nos multiprocessadores para tirar vantagem do '*hardware*' subjacente. Os vendedores fornecem muitas vezes a sua própria versão optimizada do *PVM* para os sistemas que disponibilizam, que, mesmo assim, pode comunicar com a versão do *PVM* do domínio público.

3.2.1 - Componentes do *PVM*

Esta subsecção pretende apenas focar alguns aspectos do funcionamento do *PVM*, de molde a permitir dar a conhecer, as particularidades mais relevantes deste sistema do ponto de vista dos objectivos desta tese. Os leitores interessados em conhecer mais em pormenor os detalhes do *PVM*, devem consultar o respectivo manual [Geist94a]. Pode também ser útil a leitura do relatório técnico do CICA de P. Vasconcelos: "Experiência com PVM e ScaLAPACK" [Vasconcelos94].

O *PVM* é composto essencialmente por duas partes. A primeira parte é um '*daemon*'[1] chamado '*pvmd*' que corre em cada um dos computadores que fazem parte da máquina virtual. O '*pvmd*' não executa qualquer computação, servindo apenas de encaminhador e controlador de mensagens. Ele é o ponto de contacto entre cada nó, providenciando ainda o controlo e a autenticação de processos, bem como a detecção de falhas. Para usar o *PVM*, é necessário configurar primeiro a máquina virtual, definindo uma tabela de nós. Os '*daemons*' são inicializados em cada um dos nós e cooperam entre si emulando a máquina virtual. A aplicação *PVM* pode, de seguida, ser executada a partir da linha de comando da '*shell*' de qualquer um dos nós.

Múltiplos utilizadores podem criar a sua própria máquina virtual sem interferir umas com as outras. Além disso, o mesmo utilizador pode correr várias aplicações *PVM* numa mesma máquina virtual.

[1] Um '*daemon*' é um processo que corre em "*background*", sem estar associado a nenhum terminal ou '*login shell*' em particular. Os '*daemons*' dão suporte a eventos esporádicos, ou executam tarefas periódicas.

A segunda parte do sistema é uma biblioteca de rotinas de interface com o *PVM*: a *'libpvmd'*. Esta biblioteca contém primitivas necessárias à cooperação entre tarefas numa aplicação *PVM*. Concretamente, a *'libpvmd'* contém rotinas de suporte à passagem de mensagens, à distribuição de processos, à coordenação entre tarefas e à alteração da máquina virtual. Internamente, esta biblioteca de rotinas interactua com os *'daemons' 'pvmd'* dos diferentes nós. As linguagens de programação suportadas pelo *PVM* são *C*, *C++* e *FORTRAN*.

Para além de fornecer rotinas para comunicação ponto-a-ponto, o *PVM* fornece também rotinas para comunicação colectiva, tais como[2]: *'pvmfjoingroup'*, *'pvmflvgroup'*, *'pvmfgettid'*, *'pvmfgetinst'*, *'pvmfgetsize'*, *'pvmfbcast'*, *'pvmfbarrier'*, *'pvmfscatter'*, *'pvmfgather'*, *'pvmfreduce'*. Porém, estas não se enquadram no âmbito desta tese, pelo que não serão aqui descritas. Portanto, a presente subsecção apenas foca a comunicação ponto-a-ponto. Para mais pormenores ver [Geist94a].

Conforme foi dito na Subsecção 3.1.5, a comunicação ponto-a-ponto envolve apenas duas tarefas. O modelo de comunicação assume que qualquer tarefa pode enviar uma mensagem para qualquer outra tarefa e, que não há limite para o tamanho ou número destas mensagens. Enquanto que todos os nós têm limitações na memória física disponível, o que limita o potencial tamanho do *'buffer'*, o modelo de comunicação não se restringe às limitações particulares das máquinas e assume que existe memória disponível em quantidade suficiente.

As subsecções seguintes pretendem dar uma panorâmica geral de como o *PVM* organiza e implementa este modelo de comunicação.

3.2.2 - Endereçamento, Protocolos e Formatação dos Dados

O sistema *PVM* utiliza um número inteiro de 32 bits para endereçar os *'daemons' 'pvmd'* dos diferentes nós e as tarefas, no seio da máquina virtual. Este parâmetro é designado por *identificador de tarefa* (*TID*). O *TID* permite pois identificar de forma única uma tarefa no interior da máquina virtual. Assim, o *TID* é análogo ao identificador de processo (*PID*) no âmbito do sistema *UNIX*. Dado que os

[2] As rotinas do *PVM* citadas nesta tese, destinam-se à programação em *FORTRAN*; porém, existem rotinas equivalentes às citadas desenvolvidas em linguagem *C*.

TIDs têm que ser únicos, eles são atribuídos pelo *'pvmd'* local, não podendo portanto ser fornecidos pelo utilizador.

O *PVM* contém diversas rotinas que dão o valor do *TID* para permitir que um programa *PVM* possa identificar cada uma das tarefas no sistema. O identificador de tarefa é sobretudo necessário para enviar e receber mensagens e também para testar da chegada de alguma mensagem.

Outro aspecto que importa aqui referir, é o tipo de protocolos de comunicação utilizados no *PVM*. Dada a divulgação dos protocolos *UDP* - *"User Datagram Protocol"* [Postel81b] - e *TCP* - *"Transmission Control Protocol"* [Postel81a] -, e por razões de portabilidade, estes foram os escolhidos pelos projectistas do *PVM*.

O protocolo *UDP* é um protocolo de comunicação orientado para o monólogo, enquanto que o protocolo *TCP* é um protocolo de comunicação orientado para o diálogo.

O primeiro utiliza mecanismos para garantir a correcta chegada de dados ao destino, o que o torna num protocolo fiável. O segundo não usa mecanismos que confirmem a chegada de mensagens ao destino. Isto significa que pode haver perda de mensagens, duplicação de mensagens, ou então que estas podem chegar fora de ordem ao destino. Daí ser normalmente considerado um protocolo pouco fiável.

Existem três ligações possíveis entre os elementos constituintes de uma máquina virtual: entre os *'daemons' 'pvmd'* dos diferentes nós, entre o *'pvmd'* e as suas tarefas, e entre as tarefas.

Os *'daemons'* dos diferentes nós comunicam entre si através de *'sockets' UDP*. Nestes, foram introduzidos alguns mecanismos para tornar o protocolo *UDP* mais fiável. As tarefas comunicam com o seu *'pvmd'* através duma ligação *TCP*.

A comunicação entre tarefas é normalmente estabelecida através dos *'daemons'* locais de cada tarefa, mas o *PVM* permite estabelecer uma ligação *TCP* directa entre duas tarefas, aumentando assim o desempenho da comunicação. A Figura 3.5 descreve esquematicamente a arquitectura da comunicação sobre uma hipotética rede de computadores heterogéneos.

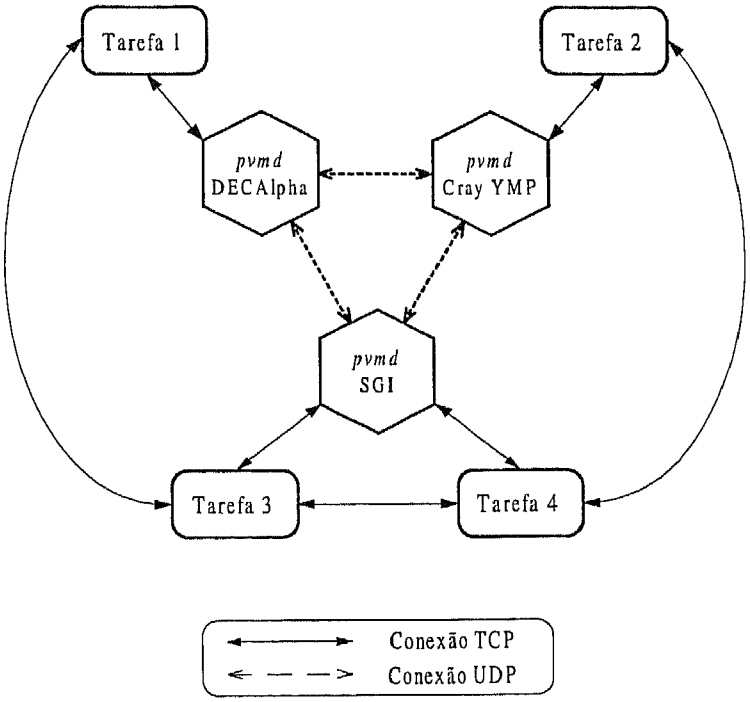

Figura 3.5 - Arquitectura da comunicação.

Do ponto de vista do utilizador, a selecção do tipo de ligação entre tarefas - isto é, se é feita directamente ou via *'daemons'* - é conseguida pela simples chamada da rotina *'pvmfsetopt'*, definindo a opção PVMROUTE tal como mostra a Figura 3.6.

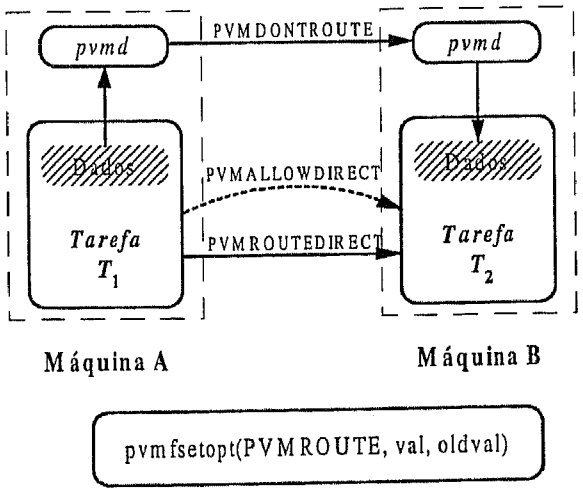

Figura 3.6 - Modos de comunicação em *PVM*.

Outro facto importante a referir, é a forma como o sistema *PVM* lida com os dados a transferir:

Se a máquina virtual for constituída por nós com diferentes tipos de representação de dados, então o sistema codifica os dados no formato *XDR* - *"External Data Representation"* [SUN87] -, de modo a garantir que os dados enviados pela tarefa emissora são correctamente recebidos pela tarefa receptora.

Se os diversos nós da máquina virtual utilizam a mesma representação de dados, então não é necessário qualquer tipo de codificação para garantir a correcta entrega de dados à tarefa receptora. Esta possibilidade melhora o desempenho da transmissão.

Haja ou não codificação dos dados, estes, por defeito, são copiados da memória do processador emissor para um *'buffer'*. Porém, quando não é necessário codificar os dados, o *PVM* permite copiar os dados directamente da memória do processador emissor para um *'buffer'* afecto ao processador receptor, sem passar por um *'buffer'* afecto ao processador emissor - Figura 3.7. Com esta estratégia é possível aumentar significativamente a velocidade da transmissão de uma mensagem, nomeadamente quando os dados a transferir são numerosos e densos.

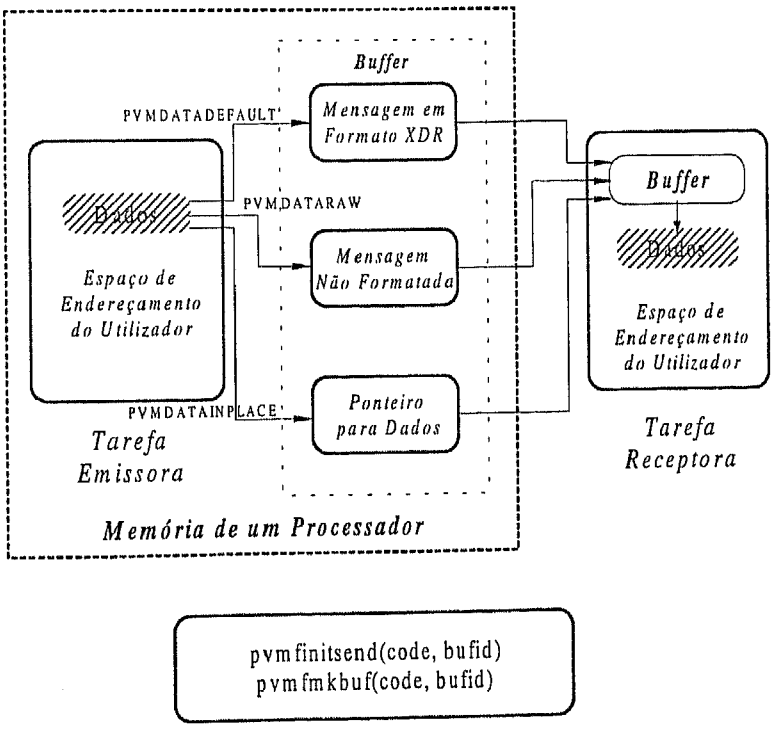

Figura 3.7 - Modos de formatação dos dados em *PVM*.

3.2.3 - Envio e Recepção de Mensagens

As mensagens trocadas entre as tarefas geridas pelo *PVM*, são, por defeito, copiadas para um *'buffer'*, e só depois enviadas ou recebidas. A Figura 3.7 introduzida na subsecção anterior permite também ilustrar este procedimento.

O envio de mensagens em *PVM* é assíncrono, pois o modo de envio é independente da recepção física da mensagem pela tarefa receptora, podendo assim continuar a executar código subsequente.

O envio de uma mensagem em *PVM* é executado em três passos: inicialização de um *'buffer'*, empacotamento dos dados e, por fim, etiquetagem e envio da mensagem.

O sistema *PVM* tem três rotinas destinadas ao envio de mensagens: a *'pvmfsend'*, a *'pvmfpsend'* e a *'pvmfmcast'*. Todas são assíncronas, no sentido em que a execução da tarefa emissora é retomada logo que o *'buffer'* esteja apto a ser reutilizado. A rotina *'pvmfmcast'* permite enviar uma mensagem para várias tarefas receptoras excluindo a tarefa emissora, não sendo por isso considerada pelos projectistas do *PVM* uma rotina para comunicação colectiva. Esta rotina tem a desvantagem da mensagem ser encaminhada através dos *'daemons' 'pvmd'*, não permitindo a transmissão directa entre tarefas, no entanto, tem a vantagem de numa chamada só enviar uma mensagem para várias tarefas.

A diferença entre as rotinas *'pvmfsend'* e *'pvmfpsend'*, é que enquanto a primeira implica a inicialização de um *'buffer'* e o empacotamento dos dados através de outras duas rotinas do *PVM*, a segunda envia uma mensagem numa única chamada ao sistema *PVM*; isto é, executa a inicialização de um *'buffer'*, o empacotamento dos dados e o envio da mensagem, através duma única primitiva do sistema *PVM*. Porém, a rotina *'pvmfpsend'* pressupõe que os dados não precisam de ser codificados.

A recepção de uma mensagem compreende duas etapas: receber a mensagem com implícita criação de um *'buffer'*, e desempacotar os dados segundo a ordem por que foram empacotados. O *PVM* permite filtrar a recepção de uma mensagem segundo a origem e/ou tipo da mesma (etiqueta).

Existem quatro rotinas para receber mensagens no âmbito do *PVM*: a rotina *'pvmfrecv'*, a *'pvmfprecv'*, a *'pvmfnrecv'* e a *'pvmftrecv'*. As duas primeiras são rotinas bloqueantes, a terceira é não-bloqueante e a quarta é um misto de recepção bloqueante e não-bloqueante. A segunda recebe e desempacota os dados duma só vez, enquanto que a primeira requer a posterior chamada de outra rotina do *PVM* para o desempacotamento dos dados. A terceira rotina pode ser chamada múltiplas vezes para verificar se determinada mensagem chegou enquanto executa trabalho útil entre chamadas. Finalmente, a rotina *'pvmftrecv'* permite especificar um intervalo de tempo de espera da recepção de uma mensagem. Isto é, a tarefa só bloqueia durante esse período tempo.

A recepção bloqueante de mensagens é assim apenas necessária quando se torna indispensável a sincronização entre tarefas.

O modelo de comunicação garante que a ordem cronológica de chegada de mensagens é preservada.

3.2.4 - Exemplo de uma Aplicação *PVM*

O procedimento geral para escrever uma aplicação em *PVM* é o seguinte: um utilizador escreve um ou mais programas em *C*, *C++* ou *FORTRAN 77* os quais contêm chamadas às rotinas da biblioteca do *PVM*. No âmbito do *PVM*, cada tarefa é vista como um programa. Posteriormente, estes programas são compilados de acordo com a arquitectura de cada uma das máquinas da tabela dos nós, sendo os ficheiros executáveis colocados num local acessível ao *PVM*. Para correr a aplicação, o utilizador inicia manualmente uma das tarefas num dos nós. Esta tarefa inicia subsequentemente as outras tarefas. Daí resulta tipicamente uma colecção de tarefas activas que fazem cálculos locais e trocam mensagens entre si para em paralelo resolver um dado problema.

Normalmente, os programas em *PVM* são organizados mediante os modelos *SPMD* ou *MPMD*.

A Figura 3.8 é um exemplo básico dum programa *PVM* do tipo *SPMD* escrito em *FORTRAN* e, como tal, consiste num programa único. O programa começa por fazer o

acesso ao sistema *PVM* e por lançar três executáveis iguais a ele próprio na máquina virtual.

```
      program spmd
      include "fpvm3.h"
      integer nprocs, etiqueta1, etiqueta2
      parameter( nprocs = 3, etiqueta1 = 1, etiqueta2 = 2 )
      integer meutid,info,iproc,ptid,numt,i
      integer tids(0:nprocs)
c
c*** Ligação ao PVM. ***
c*** Qual é o meu TID ? ***
      call pvmfmytid( meutid )
c
c*** Tenho um pai ? ****
      call pvmfparent(tids(0))
c
c*** Sou eu o pai? ****
      if( tids(0) .lt. 0 ) then
        call pvmfspawn('spmd',PVMTASKDEFAULT,'*',nprocs,
     >                 tids(1),numt)
        tids(0) = meutid
        if( numt .ne. nprocs ) stop "Problema no SPAWN"
c
        call pvmfinitsend( PVMDATADEFAULT, info )
        call pvmfpack( integer4, tids(1), nprocs, 1, info )
        call pvmfmcast( nprocs, tids(1), etiqueta1, info )
c
        do i = 1, nprocs
          call pvmfrecv( tids(i), etiqueta2, info )
          call pvmfunpack( integer4, iproc, 1, 1, info )
          print*,'Recebi uma mensagem do processador ',iproc
        end do
      else
        call pvmfrecv( -1, etiqueta1, info )
        call pvmfunpack( integer4, tids(1), nprocs, 1, info )
        do i = 1, nprocs
          if( meutid .eq. tids(i) ) iproc = i
        end do
        call pvmfinitsend( PVMDATARAW, info )
        call pvmfpack( integer4, iproc, 1, 1, info )
        call pvmfsend( tids(0), etiqueta2, info )
      endif
c*** Saída do PVM ***
      call pvmfexit(info)
      end
```

Figura 3.8 - Programa "spmd.f".

Seguidamente, mediante *'ifs'*, atribui procedimentos ao processo controlador e aos processos trabalhadores. Estes, por sua vez, enviam um inteiro ao processo controlador. No fim, desliga-se do *PVM*.

A Tabela 3.1 descreve detalhadamente o programa em causa.

Processo Controlador	*Processos Trabalhadores*
Inclui o ficheiro "fpvm3.h" de definições de variáveis do *PVM*.	
Define variáveis necessárias ao programa;	
Faz o acesso ao *PVM*: *'pvmfmytid'*;	
Lança três executáveis iguais a ele próprio na máquina virtual: *'pvmfspawn'*;	Inclui o ficheiro "fpvm3.h" de definições de variáveis do *PVM*;
Verifica se tudo correu bem;	Define variáveis necessárias ao programa;
Cria um *'buffer'* no formato *XDR*, empacota os *TIDs*, etiqueta a mensagem e envia-a aos processos trabalhadores;	Faz o acesso ao *PVM*: *'pvmfmytid'*;
	Recebe a mensagem por filtragem do número da etiqueta e desempacota os dados;
	Atribui um número inteiro "iproc" ao seu processador em função do seu *TID*;
Recebe a mensagem e desempacota os dados;	Cria um *'buffer'* no formato *RAW*, empacota o inteiro "iproc", etiqueta e envia a mensagem ao processo controlador;
Sai do PVM: *'pvmfexit'*.	Sai do *PVM*: *'pvmfexit'*.

Tabela 3.1 - Descrição do programa "spmd.f".

As Figuras 3.9 e 3.10 são um exemplo ilustrativo dum programa *PVM* do tipo *MPMD* escrito em *FORTRAN*.

Neste caso, temos dois programas distintos: o programa correspondente à tarefa mestre e um outro programa correspondente às tarefas escravas. A tarefa mestre lança as tarefas escravas na máquina virtual, as quais, por sua vez, enviam uma *'string'* à tarefa mestre. No fim, desligam-se do *PVM*.

A Tabela 3.2 descreve detalhadamente os programas em causa.

```
      program mestre
      include "fpvm3.h"
      integer nprocs, etiqueta
      parameter( nprocs = 3, etiqueta = 1 )
      integer meutid,info,numt,i
      integer tids(0:nprocs)
      character*10 buf
c
c*** Ligação ao PVM. ***
c*** Qual é o meu TID ? ***
      call pvmfmytid( meutid )
      print*,"Eu sou o",meutid
c
      call pvmfspawn("ola_mestre",PVMTASKDEFAULT,'*',nprocs,
     <              tids(1),numt)
c
      if( numt .ne. nprocs ) stop "Problema no SPAWN"
c
      do i = 1, nprocs
        call pvmfrecv( tids(i), etiqueta, info )
        call pvmfunpack( string, buf, 10, 1, info )
        print*,buf ' de ' ,tids(i)
      end do
c
c*** Saída do PVM ***
      call pvmfexit(info)
      end
```

Figura 3.9 - Programa mestre "ola.f".

```
      program escravo
      include "fpvm3.h"
      integer nprocs, etiqueta
      parameter( nprocs = 3, etiqueta = 1 )
      integer info,ptid
      integer tids(0:nprocs)
c
      call pvmfparent(ptid)
c
      call pvmfinitsend(PVMDATADEFAULT,info))
      call pvmfpack(string,'Ola mestre',10,1,info)
      call pvmfsend(ptid, etiqueta,info)
c*** Saída do PVM ***
c
      call pvmfexit(info)
      end
```

Figura 3.10 - Programa escravo "ola_mestre.f".

Tarefa Mestre	*Tarefas Escravas*
Inclui o ficheiro "fpvm3.h" de definições de variáveis do *PVM*.	
Define variáveis necessárias ao programa;	
Faz o acesso ao *PVM*: *'pvmfmytid'*;	
Lança três tarefas escravas "ola_mestre" na máquina virtual: *'pvmfspawn'*;	Inclui o ficheiro "fpvm3.h" de definições de variáveis do *PVM*;
Verifica se tudo correu bem;	Define variáveis necessárias ao programa;
	Faz o acesso ao *PVM*: *'pvmfparent'*;
Recebe a mensagem de todas as tarefas escravas e desempacota os dados;	Cria um *'buffer'* no formato *XDR*, empacota uma *string*, etiqueta a mensagem e envia-a à tarefa mestre;
Sai do PVM: *'pvmfexit'*.	Sai do *PVM*: *'pvmfexit'*.

Tabela 3.2 - Descrição dos programas "ola.f" e "ola_mestre.f".

De acordo com [Geist94b], não existem para um utilizador de *PVM* limitações teóricas relativas ao tipo de paradigma de programação. Qualquer controlo específico ou estrutura de dependências pode ser implementada no sistema *PVM* usando devidamente as suas primitivas. No entanto, existem alguns aspectos relativamente aos quais o programador deve estar atento, seja qual for o sistema de programação baseado em passagem de mensagens que utilize.

3.3 - Questões no Âmbito da Programação Paralela em Redes

Quando se programa em redes de computadores há vários factores que podem degradar o desempenho de uma aplicação paralela. Nesse sentido, e para tirar o melhor partido da paralelização de uma aplicação, é necessário ter em conta alguns aspectos intrínsecos às redes. As próximas subsecções descrevem alguns destes aspectos e algumas técnicas utilizadas para fazer face a estes, com o intuito de obter o melhor desempenho possível neste tipo de ambientes.

3.3.1 - Considerações Gerais Sobre o Desempenho

O primeiro aspecto a considerar é a ***granularidade das tarefas*** [Hwang93]. Esta pode ser definida como a razão entre o número de bytes recebidos por uma tarefa

e o número de operações de vírgula flutuante que a tarefa em causa executa. Tendo em consideração a velocidade de computação das máquinas que constituem a máquina virtual e a largura de banda da rede que as liga, o programador deve calcular um limite mínimo para a granularidade das tarefas, que compõem a sua aplicação. Quanto maior for a granularidade maior é o "*speedup*"[3]; Porém, a percentagem de paralelismo decresce com o aumento da granularidade.

O segundo aspecto a ter em conta, é o ***número de mensagens*** enviadas. Podemos situar-nos entre dois modelos extremos: um elevado número de mensagens de pequena dimensão, ou um reduzido número de mensagens de grande dimensão. Este último modelo, apesar de reduzir o tempo consumido na inicialização de mensagens, pode não conduzir a um decréscimo do tempo total de execução. Existem casos em que mensagens pequenas podem ser intercaladas com cálculo, permitindo assim minorar o '*overhead*'[4] de comunicação. A possibilidade de intercalar comunicação com computação, bem como a definição do número óptimo de mensagens a enviar, são parâmetros próprios de cada aplicação.

O terceiro aspecto a ter em consideração, é o tipo de paralelismo que melhor se adequa à aplicação: o ***paralelismo funcional*** ou o ***paralelismo dos dados*** [Hwang93]. O paralelismo funcional corresponde a ter diferentes computadores da máquina virtual a executar diferentes tarefas. Por exemplo, um supercomputador vectorial pode resolver parte do problema que se proporciona à vectorização, um multiprocessador pode resolver outra parte do problema que se proporciona à paralelização e uma estação de trabalho gráfica pode ser utilizada para visualizar os resultados em tempo-real. Cada máquina executa diferentes funções (mas, normalmente, sobre os mesmos dados).

O paralelismo dos dados corresponde à divisão dos mesmos e à sua distribuição por todas as máquinas constituintes da máquina virtual. Os diferentes processadores executam um conjunto de operações, normalmente iguais, transformando cada

[3] O "*speedup*" quantifica a aceleração do tempo de execução de uma aplicação paralela.

[4] O '*overhead*' consiste na sobrecarga temporal e/ou estrutural introduzida por acções ou informações que não fazem parte intrínseca da operação ou dos dados, mas que oferecem suporte a um processo de computação. Tipicamente, o '*overhead*' aumenta o tempo de processamento de uma aplicação; porém, em geral, essa sobrecarga é inevitável. No contexto das comunicações entre processadores ou processos, por exemplo, os mecanismos necessários à verificação de erros é uma forma de '*overhead*', e têm por finalidade possibilitar que os programas de transmissão e recepção se possam certificar da correcção dos dados.

conjunto de dados e trocando informação entre si até que o problema em causa seja resolvido. Este modelo é bastante popular em multiprocessadores de memória distribuída, resultando daí a adjudicação de um único programa a diversas máquinas, o que pode resultar em centenas de processadores a executar uma mesma tarefa. Muitos algoritmos do domínio da matemática aplicada foram desenvolvidos usando o modelo do paralelismo dos dados. Entre eles contam-se: algoritmos próprios da Álgebra Linear, algoritmos dedicados à resolução de Equações de Derivadas Parciais e algoritmos destinados à manipulação de matrizes, nomeadamente, em aplicações de processamento de imagem.

Claro que, no âmbito do *PVM*, ambos os modelos citados no parágrafo anterior se podem agrupar num modelo híbrido que explore as capacidades de cada máquina.

3.3.2 - Considerações Sobre a Rede

A partilha da rede com outros utilizadores é um dos aspectos a ter em consideração quando se deseja correr uma aplicação paralela numa rede de máquinas. Este ambiente multiutilizador e multitarefa afecta de uma forma complexa o desempenho da comunicação e, consequentemente, a execução de um programa paralelo.

Considerem-se primeiro os efeitos de diferentes ***velocidades de computação*** e as diferentes capacidades de ***memória RAM*** de cada nó da máquina virtual. Este fenómeno é típico num ambiente heterogéneo, onde, normalmente existem grandes diferenças no desempenho de cada uma das máquinas disponíveis. Assim, se por exemplo, dividirmos um problema em tarefas iguais e o pusermos a correr em cada uma das máquinas, a máquina mais lenta vai atrasar a conclusão do programa. Além disso, se as tarefas trocarem informação entre si (o que é típico), as máquinas mais rápidas vão ser obrigadas a esperar pelos dados das máquinas mais lentas, havendo assim uma degradação do desempenho do sistema global.

Um segundo aspecto a considerar são os efeitos da ***latência da transmissão*** de mensagens através da rede. Por exemplo, numa rede *'Ethernet'* ou *FDDI - "Fiber Distributed Data Interface"* - as mensagens são enviadas em série; isto é, uma de cada vez. Ora isto provoca latências significativas na troca das mensagens entre tarefas.

Compete ao utilizador determinar se deve ou não compensar este fenómeno no seu algoritmo.

Um terceiro aspecto a considerar, é o facto do ***desempenho computacional*** e o ***tráfego na rede*** estarem constantemente a alterar-se à medida que outros utilizadores partilham estes recursos. Uma aplicação pode obter um bom *"speedup"* num determinado instante e uns minutos mais tarde obter um *"speedup"* péssimo. O padrão normal de sincronização durante a corrida duma aplicação pode ser afectado por outras aplicações que se servem da rede causando a espera de dados por parte de determinadas tarefas. No pior dos casos, um erro de sincronização pode existir numa aplicação e só ocorrer quando a carga dinâmica da máquina flutua duma determinada forma. Dado que estas condições são difíceis de reproduzir, este tipo de erros dificilmente é detectado, portanto as medidas de desempenho devem ser feitas só com 1 utilizador.

Muitas destas considerações sobre a rede podem ser tomadas em linha de conta incorporando alguma forma de balanceamento de carga numa aplicação paralela.

3.3.3 - Balanceamento de Carga

O ***balanceamento de carga*** é a distribuição de trabalho computacional a diferentes processadores de modo a equilibrar a carga dos mesmos em função das suas características próprias. Esta distribuição é assim feita em função da velocidade de *CPU* das diferentes máquinas, das características de ligação *IPC* - *"Interprocess Communication"* - e da quantidade da memória *RAM* disponível nas máquinas em jogo. Esta distribuição de carga computacional pretende minimizar o tempo de inactividade (*"idle time"*) de cada processador. Diz-se que uma aplicação paralela está bem balanceada quando, em cada passo da aplicação, cada processador tem adjudicada aproximadamente a mesma quantidade útil de computação e de comunicação que os demais [Hwang93].

De acordo com [Schmidt94], o balanceamento de carga pode ser o aspecto mais importante para melhorar o desempenho duma aplicação paralela em redes de estações de trabalho. Existem diversos esquemas de balanceamento de carga para programas

paralelos. Os dois esquemas descritos de seguida são os mais usados em computação em rede [Goscinski91].

O método mais simples é o ***balanceamento estático de carga***. Neste método, o problema é dividido à partida e as tarefas são atribuídas aos processadores uma só vez. A divisão dos dados pode ocorrer antes do processo arrancar, ou a divisão pode ser feita num passo anterior da aplicação. A extensão ou o número de tarefas atribuídas a uma dada máquina pode variar tendo em conta as diferentes potencialidades computacionais das diferentes máquinas em jogo. Como todas as tarefas podem estar activas desde o início, elas podem comunicar e cooperar umas com as outras. Numa rede leve, isto é, com pouca carga, este tipo de método pode ser bastante efectivo.

Quando as cargas computacionais variam ao longo do tempo, é conveniente adoptar um método de ***balanceamento dinâmico de carga***. O método mais popular é o chamado ***'Pool of Tasks Paradigm'***. Este método é normalmente implementado num programa do tipo mestre/escravo onde o mestre cria e mantém uma pilha de tarefas, que vai atribuindo aos programas escravos assim que estes ficam inactivos. A pilha é usualmente implementada como uma fila de espera e, se as tarefas variarem em tamanho, as maiores são colocadas junto ao topo da pilha. Com este método todos os escravos são mantidos ocupados enquanto existirem tarefas na pilha. Como as tarefas começam e terminam em tempos arbitrários, este método é apropriado para aplicações que não requeiram comunicação entre os programas escravos e que apenas comuniquem com o programa mestre e com ficheiros distribuídos na rede.

Ultimamente tem sido proposto um terceiro esquema de balanceamento dinâmico de carga: o da ***auto-avaliação e redistribuição de carga***. Este método, que não utiliza um programa mestre, requer que em determinados momentos todos os processos reexaminem e redistribuam as suas cargas. Por exemplo, na resolução de Equações de Derivadas Parciais, cada passo pode ser balanceado estaticamente e, entre cada passo e o seguinte pode-se examinar a forma como o problema se alterou, redistribuindo nessa altura a grelha de pontos [Thévenin95]. Há diversas variações deste esquema. Algumas implementações nunca sincronizam com todas as tarefas em jogo, mas apenas distribuem o excesso de carga pelas tarefas vizinhas. Outras implementações esperam até que uma tarefa avise que a sua carga excedeu uma determinada tolerância, fazendo-se nessa altura uma redistribuição em vez de se esperar um determinado intervalo de tempo.

3.4 - Síntese

Este capítulo focou os aspectos fundamentais relativos à programação por passagem de mensagens. Posteriormente, descreveu sumariamente o *'software' PVM*, - o qual se enquadra neste tipo de modelo -, e que de entre outros sistemas de computação em redes de computadores, foi o escolhido para a prossecução deste trabalho. Por fim, alertou para alguns dos problemas que a programação paralela em redes acarreta.

As questões referidas neste capítulo são naturalmente importantes enquanto enquadradas no ambiente computacional onde o trabalho prático, que será descrito em posterior capítulo desta tese, foi realizado. A descrição do ambiente computacional deste trabalho será precisamente o objectivo do próximo capítulo.

Capítulo 4

Ambiente Computacional

Este capítulo descreve sucintamente as principais características do ambiente computacional usado no desenvolvimento desta tese.

O núcleo do ambiente computacional que aqui se refere, é a *'Farm ALPHA AXP'* instalada no *CICA*[1]. Este sistema - que ao longo desta tese será sistematicamente designado por *'Farm'* - é constituído por um conjunto de quatro servidores *'ALPHA'*, interligados por um comutador de fibra óptica. Esta *'Farm'*, quando munida de *'software'* apropriado, como por exemplo o *PVM*, toma a forma de uma máquina paralela virtual, permitindo assim a execução de aplicações paralelas e distribuídas.

4.1 - O *'hardware'*: *'DECAlphas AXP'* e *'GIGAswitch'*

Os sistemas *'ALPHA AXP'* produzidos pela empresa *'Digital Equipment Corporation'* - daqui por diante simplesmente designada por *'Digital'* - são máquinas baseadas numa arquitectura *RISC* - *"Reduced Instruction Set Computer"* - de endereçamento de 64-bit. Os servidores incluídos na *'Farm'* sobre o qual o presente trabalho foi desenvolvido, são 4 sistemas *'ALPHA AXP'*, iguais dois a dois. Concretamente, a *'Farm'* em causa reúne 2 máquinas *'DECAlpha AXP 3000/500'* e 2

[1] *CICA* - Centro de Informática Prof. Correia de Araújo da Faculdade de Engenharia da Universidade do Porto.

máquinas *'DECAlpha AXP 3000/600'*. Tanto um como o outro tipo de máquinas são baseados numa arquitectura uniprocessador. As 4 máquinas que compõem a *'Farm'* estão ligadas por uma rede *FDDI* através dum *'GIGAswitch'*. A Figura 4.1 descreve esquematicamente a topologia da *'Farm'*.

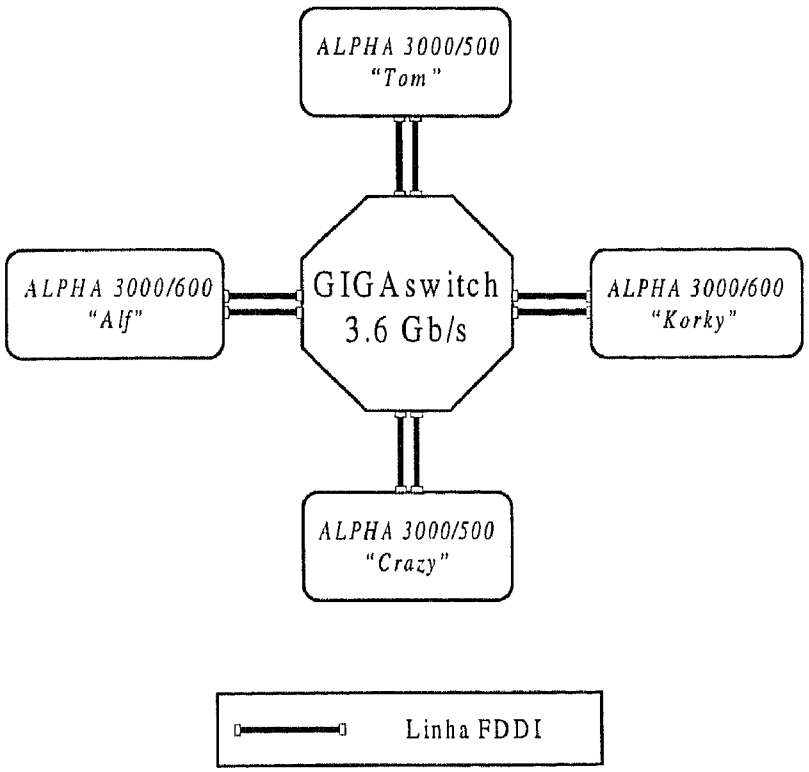

Figura 4.1 - Esquema da *'Farm'* do CICA.

O *'DECAlpha AXP 3000/500'* é uma máquina cujo período de relógio é de 6,7 ns com um processador cujo desempenho de pico é de 150 Mflops [DEC500]. As máquinas deste tipo que integram a *'Farm'* que serviu de base ao desenvolvimento deste trabalho, possuem uma memória *'cache'* com uma capacidade de 512 Kbyte e estão dotados de 128 Mbyte de memória *RAM*.

O *'DECAlpha AXP 3000/600'* é uma máquina mais rápida e mais moderna do que a *'DECAlpha AXP 3000/500'*. Assim, o período do relógio desta máquina é de 5,7 ns com um processador cujo desempenho de pico é de 175 Mflops [DEC600]. As máquinas deste tipo que integram a *'Farm'* em causa, possuem uma memória *'cache'* com uma capacidade de 2 MByte e estão dotadas de 64 Mbyte de memória *RAM*.

Note-se que as máquinas *'DECAlpha AXP 3000/600'* apresentam um desempenho teoricamente superior ao das máquinas *'DECAlpha AXP 3000/500'* devido, em grande parte, às características intrínsecas do tipo de processador, que as equipam, como sejam a frequência do relógio e a capacidade da memória *'cache'*. Porém as máquinas *'DECAlpha AXP 3000/600'* que integram a *'Farm'* têm menos memória *RAM* do que as *'DECAlpha AXP 3000/500'*.

Esta diferença na capacidade da memória *RAM* entre estes dois tipos de máquinas (64 Mbyte para a *'DECAlpha AXP 3000/600'* e 128 Mbyte para a *'DECAlpha AXP 3000/500'*) é susceptível de conduzir a desempenhos melhores por parte das máquinas *'DECAlpha AXP 3000/500'*, nomeadamente, quando está em causa a resolução de problemas que comportam um volume de dados que não pode ser armazenado, na sua totalidade, na memória *RAM* das máquinas *'DECAlpha AXP 3000/600'*, mas que pode contudo ser armazenado na memória *RAM* das máquinas *'DECAlpha AXP 3000/500'*.

O *'GIGAswitch'* que integra a *'Farm'* foi produzido pela empresa *'Digital'*. Este *'GIGAswitch'* é um comutador de barramento cruzado muito rápido, constituído por uma matriz de 36 portas [Souza94]. As máquinas atrás descritas estão ligadas ao *'GIGAswitch'* através de linhas *FDDI*, obtendo-se assim um canal de comunicação com uma largura de banda de 100 Mbit/s.

O *'GIGAswitch'*, ao permitir a transmissão e recepção simultânea de mensagens entre os elementos constituintes da *'Farm'*, fornece um aumento significativo da largura de banda relativamente a uma rede de máquinas interligadas directamente por linhas *FDDI*. Esta largura de banda é escalável com o aumento do número dos elementos constituintes da *'Farm'*, no sentido em que ao acrescentar mais máquinas à *'Farm'*, a largura de banda agregada aumenta na mesma proporção. Neste caso, o *'GIGAswitch'* da *'Farm'* do CICA oferece uma largura de banda agregada máxima de 3.6 Gigabit/s, pois permite 36 conversações simultâneas. Outra particularidade do *'GIGAswitch'*, é a de tornar a latência da comunicação entre processadores muito diminuta, uma particularidade extremamente importante do ponto de vista do desempenho de um sistema computacional na execução de aplicações paralelas [Souza94].

4.2 - Sistema Operativo e Bibliotecas

Todas as máquinas instaladas na *'Farm'* estão equipadas com o sistema operativo *DEC OSF/1* Versão 3.0 [DECOSF/1]. Este sistema operativo é um sistema multiutilizador e multitarefa, alicerçado num núcleo (*'kernel'*) de arquitectura avançada de 64-bit. Este, por sua vez, é baseado na concepção do *'kernel' 'Mach'* Versão 2.5 da Universidade de *'Carnegie Mellon'*, sobre o qual a *'Digital'* acrescentou componentes da *"Berkeley Software Distribution"* (*BSD*) Versões 4.3 e 4.4, e outros produtos tanto do domínio público como da própria *'Digital'*. O sistema operativo *'DEC OSF/1'* Versão 3.0 é compatível com as interfaces de programação do sistema '*Berkeley* 4.3' e do Sistema V [DECOSF/1].

Todas as ferramentas que aqui se descrevem correm sobre este sistema operativo. A presente secção descreve precisamente os aspectos mais importantes dessas ferramentas.

As bibliotecas de rotinas matemáticas para aplicações científicas são uma importante ferramenta para computação de elevado desempenho, no sentido em que ajudam a reduzir o custo da computação, aumentando, consequentemente, a produtividade de qualquer centro de cálculo. A biblioteca *DXML* [DXML94] - *"Digital eXtended Math Library"* - foi construída e adaptada pela *'Digital'*, a partir de rotinas já existentes, com o intuito de fornecer um elevado desempenho nos sistemas *'ALPHA'*.

A *'Digital'* divide as rotinas da biblioteca em quatro áreas distintas [Kamath94]:

- *BLAS* - Subrotinas Básicas de Álgebra Linear. Estas incluem as *BLAS 'standard'* (*BLAS* 1 [Lawson79], *BLAS* 2 [Dongarra88] e *BLAS* 3 [Dongarra90]) e mais alguns melhoramentos implementados pela *'Digital'*;

- *LAPACK* - *PACK*age de Álgebra Linear [Anderson92]. Este grupo de rotinas inclui solucionadores de sistemas de equações lineares e de cálculo de valores próprios;

- Processamento de Sinal - Este grupo inclui rotinas destinadas à resolução de Transformadas Rápidas de Fourier (*FFT*s), e à convolução e à correlação de funções [DXML94];

- Solucionadores de Sistemas Lineares Esparsos - Este grupo de rotinas inclui solucionadores directos [Duff86] e iterativos [Barrett93].

A maioria das rotinas da biblioteca *DXML* foram construídas a partir das referências acima citadas e propositamente optimizadas para a arquitectura *'ALPHA'*.

Todas as rotinas de todos os subcomponentes da *DXML* seguem as convenções da linguagem de programação *FORTRAN*, no sentido em que a biblioteca *DXML* assume os *'standards'* do *FORTRAN* no que respeita à passagem de argumentos e ao armazenamento dos dados. Por exemplo, as matrizes são guardadas na memória por coluna.

O código fonte dos subcomponentes da *DXML* foi escrito em *FORTRAN* e em *C*.

No seu conjunto, a biblioteca *DXML* contém quase 400 rotinas. Muitas destas rotinas estão disponíveis em quatro versões: precisão simples real, precisão dupla real, precisão simples complexa e precisão dupla complexa. Estas rotinas podem ser chamadas tanto de programas escritos em *FORTRAN* como em *C*, desde que se tenha em linha de conta a diferença típica destas duas linguagens no tocante à forma como o armazenamento de matrizes é feito.

Todas as rotinas dos quatro subcomponentes da biblioteca *DXML* foram talhadas para tirar vantagem das características do sistema *'ALPHA'*. Esta preocupação inclui, por exemplo, o número de registos disponíveis de vírgula flutuante, a capacidade da memória *'cache'* de dados primária e secundária, e do tamanho de página. Esta optimização envolveu também alterações sobre as rotinas do domínio público relativamente às estruturas de dados e ao uso de novos algoritmos de modo a reestruturar a computação para lidar de maneira eficaz com a hierarquia de memória.

Algumas técnicas gerais são usadas em todos os subcomponentes da *DXML* por forma a atingir um melhor desempenho computacional. Estas técnicas incluem [Daydé91], [Hwang93]:

- Evitar os ciclos por forma a optimizar a utilização das *'pipelines'* de vírgula flutuante;

- Acesso aos dados usando computação do tipo *'stride-1'*, isto é, de incremento 1.

- Gestão eficaz da hierarquia de memória baseada no princípio da localidade de referência, nomeadamente através de:

    - Reutilização, sempre que possível, dos dados armazenados nos registos do processador por forma a minimizar as transferências de informação entre este e a memória;

    - Gestão eficiente das *'caches'* de dados de molde a maximizar as referências locais e a reutilização dos dados. Por exemplo, os algoritmos em causa foram estruturados para operar em sub-blocos de matrizes devidamente dimensionados para permanecerem na *'cache'* até que todas as operações que envolvem os dados do sub-bloco terminem;

    - Algoritmos que minimizam a ocorrência de falta de páginas e perdas no *'buffer'* de tradução de endereços, evitando-se assim transferências excessivas de informação entre memória e disco;

Os leitores mais interessados neste assunto poderão constatar, através do trabalho de C. Kamath [Kamath94], que estas optimizações conduzem a desempenhos óptimos. O trabalho em causa cita valores de desempenho das rotinas da *DXML* para a arquitectura *'ALPHA'* de uma maneira geral significativamente superiores às rotinas equivalentes do domínio público. Concretamente, no caso da rotina *DGEMM* das *BLAS*3, - que executa o produto entre duas matrizes -, dá conta da obtenção de um desempenho aproximadamente cinco vezes superior ao da rotina equivalente do domínio público.

Dado que no âmbito deste trabalho foram utilizadas algumas rotinas *BLAS*, importa descrever com algum detalhe esta subcomponente.

Conforme foi referido anteriormente, as rotinas *BLAS* subdividem-se em *BLAS* de nível 1, *BLAS* de nível 2 e *BLAS* de nível 3.

As rotinas *BLAS* de nível 1 operam apenas com vectores e escalares. Dada a natureza dos dados que envolvem estas operações, as rotinas *BLAS* de nível 1 têm uma granularidade particularmente baixa. Exemplos deste tipo de rotinas são o produto interno, o índice do elemento máximo de um vector, a soma dos valores absolutos dos elementos de um vector e a soma do produto de um escalar por um vector com outro vector.

As rotinas *BLAS* de nível 2 incluem operações de maior granularidade que as *BLAS* 1, pois executam operações que envolvem matrizes e vectores; por exemplo, o produto matriz-vector e a solução de sistemas de equações triangulares. Estas rotinas suportam vários tipos de armazenamento dos dados tais como o genérico, o simétrico, a banda, e o empacotado.

As rotinas *BLAS* de nível 3 executam operações que envolvem matrizes, o que corresponde a operações de maior granularidade que as das *BLAS* 2. Estas rotinas incluem, por exemplo, o produto matriz-matriz e a resolução de sistemas de equações triangulares com múltiplos termos independentes. Quando particularizadas, estas operações são definidas para matrizes que podem ser genéricas, simétricas ou triangulares.

As rotinas matemáticas da *DXML* utilizadas neste trabalho foram a *DGEMM* das *BLAS*3 e a *DSCAL* das *BLAS*1.

Dada a falta de procedimentos de apoio ao desenvolvimento deste trabalho no âmbito da medição de tempos, foi propositadamente criada por J. Peres e P. Vasconcelos [Peres95], uma biblioteca de rotinas para colmatar esta lacuna. Esta biblioteca é constituída basicamente por rotinas que medem tempo de *CPU*, tempo de sistema e tempo real (tempo de relógio).

Esta biblioteca de tempos é constituída por 2 conjuntos de rotinas perfazendo no total 5 rotinas. Concretamente, foi criado o conjunto de três rotinas: a *tcpu*, a *tsys*, a *treal*; e um segundo conjunto de rotinas construídas à custa das 3 primeiras: a *tcr* e a *ttotal*.

A rotina *tcpu* retorna o tempo de *CPU*; a rotina *tsys* retorna o tempo de sistema; finalmente, a rotina *treal* retorna o tempo real. Para medir um determinado tempo, basta chamar a rotina apropriada antes e depois do troço de código cujo tempo de corrida se pretende avaliar, calculando-se de seguida a diferença dos tempos obtidos.

A rotina *tcr* retorna o tempo de *CPU* e o tempo real; e a rotina *ttotal* retorna o tempo de *CPU*, de sistema e real. Analogamente, para medir o tempo pretendido, chama-se as rotinas antes e depois do troço de código a contabilizar e toma-se a diferença entre os respectivos parâmetros.

A biblioteca de medição de tempos foi escrita em linguagem *C* mas inclui uma interface *C-FORTRAN*. Isto permite que as rotinas desta biblioteca possam ser chamadas a partir de programas escritos numa quer noutra linguagem.

Código em *C*	Interface *C-FORTRAN*

`double tcpu()`
`{`
`  double msecond;`
`  struct rusage tw;`
`  getrusage (RUSAGE_SELF,&tw);`
`  msecond = tw.ru_utime.tv_sec +`
`tw.ru_utime.tv_usec * 1.0e-6;`
`  return msecond;`
`}`
 	
`double treal()`
`{`
`  double msecond;`
`  struct timeval tv;`
`  gettimeofday (&tv,0);`
`  msecond = tv.tv_sec +`
`tv.tv_usec * 1.0e-6;`
`  return msecond;`
`}`
 	
`void tcr (cpu, real)`
`double *cpu, *real;`
`{`
`extern double tcpu(), treal();`
`  *cpu = tcpu();`
`  *real = treal();`
`}`
	`double tcpu_()`
	`{`
	`  return tcpu();`
	`}`
	
	`double treal_()`
	`{`
	`  return treal();`
	`}`
	
	`void tcr_(cpu, real)`
	`double *cpu, *real;`
	`{`
	`  tcr(&cpu,`
	`&real);`
	`}`

Figura 4.2 - Extracto do código fonte das rotinas *tcpu*, *treal*, *tcr*.

O código destas rotinas é baseado em chamadas às funções do sistema operativo *'DEC OSF/1'*, *'gettimeofday'* e *'getrusage'*, tal como é ilustrado no código apresentado na Figura 4.2.

A resolução que é possível obter na medida dos tempos é de 10^{-6} segundos, isto é, 1 microsegundo. Esta resolução é perfeitamente aceitável no âmbito do desenvolvimento deste trabalho.

4.3 - Síntese

Este capítulo focou as particularidades mais relevantes do ambiente computacional que serviu de suporte ao trabalho que esta tese expõe. Concretamente, ele descreveu tanto as características do *'hardware'* como das bibliotecas utilizadas.

A começar este capítulo, descreveu-se sucintamente a *'Farm'* de *'ALPHAS'* do *CICA*, nomeadamente os aspectos mais importantes das máquinas dos tipos *'DECAlpha AXP 3000/500'* e *'DECAlpha AXP 3000/600'* e do *'GIGAswitch'*, que a integram. Posteriormente, foi sumariamente apresentado o sistema operativo *DEC OSF/1* que equipa as máquinas da *'Farm'*, e fez-se uma descrição sucinta das bibliotecas de rotinas utilizadas: a biblioteca *DXML* de rotinas matemáticas e uma biblioteca de rotinas para a medição de tempos de execução de programas.

As questões abordadas neste capítulo são essenciais do ponto de vista do enquadramento do trabalho experimental realizado no âmbito desta tese, trabalho esse que o próximo capítulo descreve.

Capítulo 5

Implementação e Experimentação

A implementação do método da correcção do resíduo para o caso de valores próprios múltiplos, mencionada nos artigos do Capítulo 2, requer cuidados adicionais, nomeadamente a necessidade de trabalhar com uma base do subespaço invariante maximal em vez dos vectores próprios conforme descrito em [Ahues90]. Essa implementação foi efectuada por M. I. Vieira [Vieira91] para aproximações iniciais obtidas por diversos métodos (Galerkin, Fredholm, Nystron, Sloan). Posteriormente, F. d'Almeida [Almeida93] fez a paralelização da fórmula de refinamento, que vai ser estudada aqui, no caso de se partir de uma aproximação inicial de Nystron. Neste trabalho apresentam-se resultados numéricos da implementação de um programa de refinamento em que se parte de aproximações iniciais de Fredholm. Vai começar-se por descrever o núcleo do operador:

O núcleo do operador de Fredholm utilizado em todos os exemplos numéricos desta tese é o núcleo definido pela expressão:

$$K(s,t) = 0.25 - 0.5*|t-s| \text{ com } s,t \in [0,1] \qquad (5.1),$$

cujos valores próprios teóricos são dados pela expressão:

$$\left\{[(2k-1)\pi]^{-2}, k \in N\right\} \qquad (5.2),$$

e têm multiplicidade 2.

Seguindo uma das opções tomadas para o valor de n em [Almeida93], tomou-se $n = 17$ e, variou-se o valor de m, onde $m \in \{1025, 2049, 3073, 4097\}$, tendo sempre em vista que ao aumentar o valor de m os valores próprios aproximados vão ser mais precisos mas também nunca esquecendo que o algoritmo vai ficar mais pesado. Seguindo, também, as opções tomadas em [Almeida93], como critério de paragem do algoritmo, exigiu-se que a norma infinito do resíduo, não fosse superior a uma tolerância *tol* previamente fixada em 10^{-11}, e todos os cálculos foram feitos em dupla precisão, o que corresponde a uma precisão de aproximadamente 15 casas decimais.

Todos os exemplos desta tese referem-se ao refinamento do primeiro valor próprio do núcleo definido pela expressão 5.1. A precisão pretendida foi atingida sempre ao fim de duas iterações.

Seguindo, também, as opções tomadas em [Almeida93] calculou-se o tempo gasto nas seguintes fases do algoritmo:

1ª) A Fase da discretização das aproximações T_n e T_m de T, obtendo-se as matrizes A_n e A_m, respectivamente;

2ª) A Fase de inicialização que inclui o cálculo dos valores próprios da matriz A_n pelo método QR, prolongamento e renormalização dos vectores próprios a serem refinados e o cálculo das matrizes necessárias para a equação de Sylvester;

3ª) A Fase do refinamento iterativo dos valores próprios aproximados, que corresponde ao subprograma DCFNY do programa de refinamento, onde cada iteração inclui:

- Cálculo de χ_k, dado pela expressão (2.22), que envolve um produto $A_m X$ [1] e outros pequenos cálculos, nomeadamente, produtos de matrizes de dimensão menor e produtos de escalares por vectores;

- Cálculo da entrada para SIGMAN, isto é, o cálculo de $F(\chi_k)$ que envolve também o produto $A_m X$ e outros pequenos cálculos,

[1] X representa uma matriz de dimensão m x ma, onde neste caso $ma = 2$.

nomeadamente, produtos de matrizes de pequena dimensão e produtos de escalares por vectores;

- Resolução da equação de Sylvester, pelo algoritmo de Bartels-Stewart [Bartels72];
- Prolongamento da solução da equação de Sylvester a um subespaço de dimensão m;
- Cálculo do resíduo e norma infinito do resíduo;

Contabilizou-se o tempo real para estas diferentes fases da seguinte forma:

- Tempo de discretização - t_d - que corresponde à 1ª fase do algoritmo;
- Tempo de inicialização - t_I - que corresponde à 2ª fase do algoritmo;
- Tempo da *i*ésima iteração - t_i - onde não incluímos o cálculo do resíduo;
- Tempo do produto $A_m X$ - t_p;
- Tempo da equação de Sylvester - t_e;
- Tempo da entrada de SIGMAN - t_{es};
- Tempo de SIGMAN - t_s;
- Tempo do resíduo - t_r;
- Tempo total de iteração - t_{ti};
- Tempo total do refinamento, isto é, o tempo necessário para se obterem os valores próprios refinados - t_{dcfny}.

Com base nestas definições tem-se que:

$$t_r \approx t_p \qquad (5.3);$$

$$t_{es} \approx 2*t_p \quad (5.4);$$

$$t_i \approx t_{es} + t_e + t_s \quad (5.5);$$

$$t_{ti} = \sum_{i=1}^{k} t_i \text{, onde } k \text{ é o número total de iterações} \quad (5.6);$$

$$t_{dcfny} \approx t_{ti} + k*t_r \quad (5.7).$$

Correu-se o programa para o valor de m = 1025, nos dois tipos de máquinas da *'Farm'*. As Tabelas 5.1 e 5.2 ilustram os tempos obtidos.

t_d	t_l	t_1	t_r	t_2	t_r	t_{ti}	t_{dcfny}	t_p	t_e	t_{es}	t_s
0,370	0,018	0,434	0,211	0,427	0,210	0,860	1,285	0,209	0,002	0,422	0,008

Tabela 5.1 - Tempos reais, em segundos, obtidos na máquina *'DECAlpha 3000/600'* para m = 1025.

t_d	t_l	t_1	t_r	t_2	t_r	t_{ti}	t_{dcfny}	t_p	t_e	t_{es}	t_s
0,383	0,018	0,492	0,240	0,487	0,240	0,979	1,464	0,239	0,003	0,480	0,009

Tabela 5.2 - Tempos reais, em segundos, obtidos na máquina *'DEC Alpha 3000/500'* para m = 1025.

Da análise das Tabelas 5.1 e 5.2 verifica-se que o que torna pesado o algoritmo são essencialmente os produtos de matrizes do tipo A_mX, como já antes tinha sido concluído teoricamente em [Almeida84] e experimentalmente em [Almeida93].

Para se obter os valores próprios refinados para este caso foram necessários 6 produtos[2] do tipo A_mX. A percentagem de tempo gasto nestes produtos foi de aproximadamente 99% do tempo t_{dcfny}. Dados estes resultados parece natural investir

[2] O programa usou 6 produtos, pois o cálculo de um produto A_mX necessário para o cálculo do resíduo não foi aproveitado na iteração seguinte, como poderia ter sido feito.

na paralelização do código correspondente a estes produtos. Vai passar-se a descrever a estratégia de paralelização adoptada.

5.1 - Estratégia de Paralelização

O algoritmo que se pretende paralelizar está representado esquematicamente na Figura 5.1.

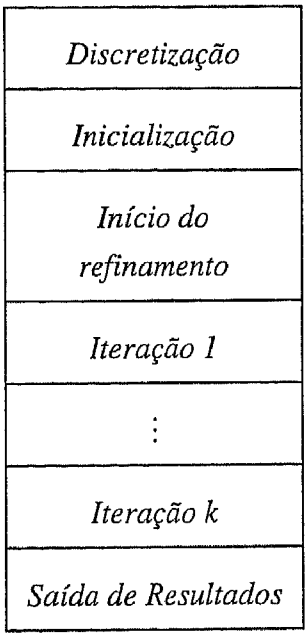

Figura 5.1 - Esquema do algoritmo sequencial.

Dada a natureza do algoritmo descrita na secção anterior, optou-se por uma estratégia de paralelização que se enquadra num modelo híbrido, isto é, conjuga o modelo do paralelismo dos dados com o modelo de paralelismo funcional.

A sua implementação foi baseada no modelo *MPMD*, isto é, adoptou-se o esquema mestre/escravo. O mestre, além de fazer o mesmo trabalho que os escravos, tem a seu cargo todas as inicializações, difunde informação, recebe resultados intermédios, efectua cálculos intermédios com esses resultados, e, por fim, recebe os resultados finais. Os escravos, por sua vez, fazem cálculos e enviam resultados ao mestre. Só existe comunicação entre mestre e escravos, não havendo qualquer troca de informação entre escravos.

A questão fulcral do algoritmo paralelo assenta no facto da matriz A_m ser densa e ser conhecida à partida, e de não sofrer qualquer alteração dos seus valores ao longo do algoritmo. Esta particularidade do algoritmo torna desnecessária a distribuição dos valores da matriz ou blocos da matriz pelos escravos. Isto, evita o envio de um extenso volume de dados do mestre para os escravos, o que acarretaria, um tempo substancial despendido na comunicação, o que inevitavelmente conduziria a uma degradação no desempenho da execução da aplicação paralela.

Assim, a discretização da matriz A_m é inicialmente dividida equitativamente pelos p processadores. Concretamente, tanto o mestre como os escravos, calculam localmente a discretização da matriz A_m da parte correspondente ao bloco linha A_i, $0 \leq i \leq p$, como mostra a Figura 5.2:

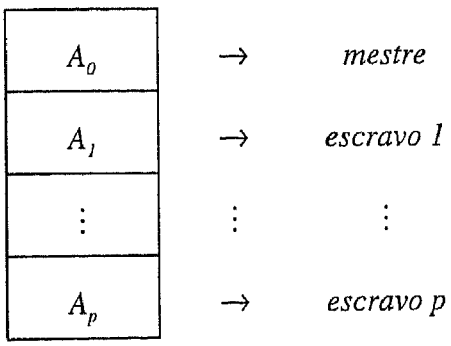

Figura 5.2 - Distribuição dos dados pelos p processadores em blocos linha.

Dado que todos os produtos do tipo A_mX se processam no bloco de código correspondente a t_{dcfny}, importa descrever o padrão de comunicação/computação em cada iteração, onde foi incluído o cálculo do resíduo. A Figura 5.3 ilustra esse padrão se se dispuser de dois processadores.

Se se dispuser de mais processadores para resolver este problema, o padrão de comunicação/computação não se alterará, pois não existe, comunicação entre escravos. A única diferença é que o mestre difundirá e receberá informação, de e para vários escravos.

Da Figura 5.3 depreende-se que este algoritmo é muito síncrono, o que poderá, eventualmente, ser uma desvantagem na obtenção de um bom desempenho computacional.

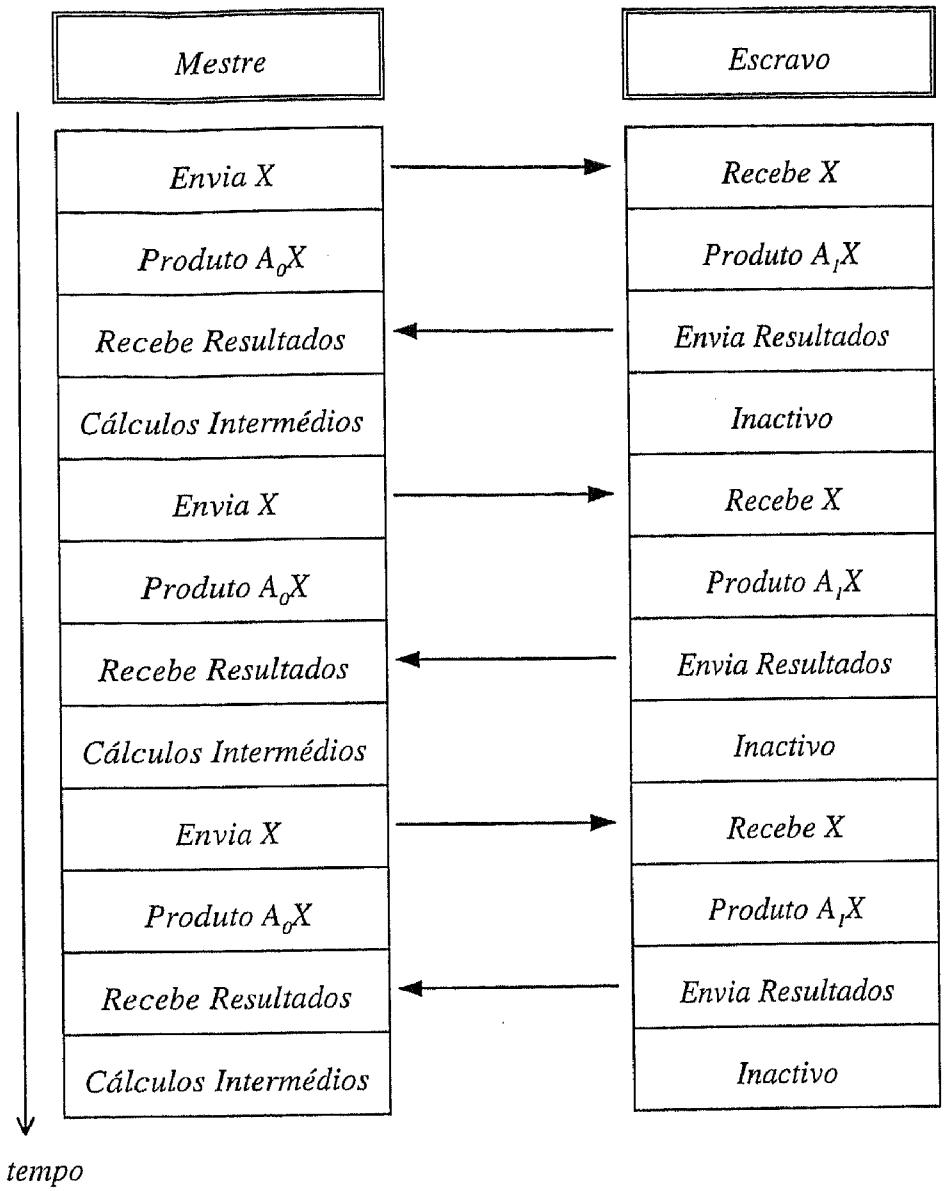

Figura 5.3 - Padrão de comunicação/computação em cada iteração k no modelo mestre/escravo com 2 processadores.

5.2 - Resultados Experimentais

Nesta secção apresenta-se todos os resultados experimentais obtidos no ambiente computacional descrito no Capítulo 4.

Estes resultados foram obtidos nas seguintes condições:

- Monopolização da *'Farm'* pela aplicação em causa, por razões referidas na Subsecção 3.3.2 do Capítulo 3;

- Encaminhamento directo das mensagens entre tarefas *PVM*, por forma a minimizar o tempo de comunicação do programa;

- Os dados não foram formatados, pois todas as máquinas da *'Farm'* têm a mesma representação de dados.

Dada a diferença nas características das máquinas que constituem a *'Farm'* (os nós "korky" e "alf" com 64 Mbyte de memória *RAM* e uma frequência de relógio de 175 MHz; e os nós "crazy" e "tom" com 128 Mbyte de memória *RAM* e uma frequência de relógio de 150 MHz) e, no sentido de estudar o desempenho do algoritmo este foi testado em todas as configurações possíveis da máquina virtual.

Consideram-se aqui dois casos: o **CASO I** que corresponde a fixar-se o programa mestre no nó "korky" e donde deriva o seguinte mapa de máquinas virtuais possíveis - Tabela 5.3:

Máquina Virtual	*Nós*
*MVK*1	"korky", "alf"
*MVK*2	"korky", "tom"
*MVK*3	"korky", "alf", "tom"
*MVK*4	"korky", "tom", "crazy"
*MVK*5	"korky", "tom", "alf", "crazy"

Tabela 5.3 - Mapa das máquinas virtuais possíveis para o CASO I.

e o **CASO II** que resulta de se fixar o programa mestre no nó "crazy", obtendo-se assim o seguinte mapa de máquinas virtuais possíveis - Tabela 5.4:

Máquina Virtual	*Nós*
*MVC*1	"crazy", "tom"
*MVC*2	"crazy", "korky"
*MVC*3	"crazy", "tom", "korky"
*MVC*4	"crazy", "korky", "alf"
*MVC*5	"crazy", "korky", "alf", "tom"

Tabela 5.4 - Mapa das máquinas virtuais possíveis para o CASO II.

Executou-se o algoritmo sequencial tanto no nó "korky" como no nó "crazy", e tomaram-se os tempos obtidos como ponto de comparação com os tempos obtidos na versão paralela, para o CASO I e CASO II, respectivamente.

As Tabelas 5.5 e 5.6 mostram os tempos obtidos na paralelização do produto de matrizes do tipo $A_m X$ para os vários valores de m, para os dois casos. Estes tempos são valores médios. O tempo do produto inclui o tempo de comunicação passado no envio de X e na recolha dos resultados enviados pelos escravos.

m	Sequencial	*MVK*1	*MVK*2	*MVK*3	*MVK*4	*MVK*5
1025	0,209	0,116	0,134	0,093	0,096	0,068
2049	0,840	0,434	0,508	0,353	0,367	0,255
3073	75,335	0,985	1,135	0,737	0,750	0,573
4097	136,347	54,315	49,181	1,395	1,693	0,997

Tabela 5.5 - Tempos reais, em segundos, correspondentes a t_p para o CASO I.

m	Sequencial	*MVC*1	*MVC*2	*MVC*3	*MVC*4	*MVC*5
1025	0,239	0,176	0,176	0,111	0,109	0,075
2049	0,967	0,506	0,496	0,359	0,345	0,260
3073	2,215	1,169	1,077	0,754	0,742	0,591
4097	164,411	2,061	49,810	1,334	1,279	1,102

Tabela 5.6 - Tempos reais, em segundos, correspondentes a t_p para o CASO II.

No sentido de compreender melhor os resultados obtidos na versão paralela, analisou-se em detalhe os tempos obtidos na versão sequencial do algoritmo:

É nítido que para os valores de $m = 1025$ e $m = 2049$ os tempos obtidos no CASO I são melhores. Já era de esperar que assim fosse pois o nó "korky" tem um processador mais rápido que o nó "crazy".

Outro pormenor curioso é o que se observa quando se aumenta m de 2049 para 3073: no CASO I o tempo real obtido é muito mais elevado que no CASO II. A explicação para este fenómeno está na memória *RAM* existente em cada um dos nós, sendo a do nó "korky" metade da do nó "crazy": a matriz A_m, com $m = 2049$, corresponde a um espaço de memória de aproximadamente 32 Mbyte, enquanto que para $m = 3073$, o sistema tem de reservar um espaço de memória na ordem dos 72 Mbyte, o qual não é possível armazenar na sua totalidade na memória *RAM* do nó "korky". Consequentemente, o sistema necessita de efectuar operações de *'swaping'* e *'paging'*, isto é, tem que aceder ao disco duro. Ora, este acesso ao disco duro vai penalizar muito os tempos reais obtidos, aliás como se pode constatar pelos valores representados nas Tabelas 5.5 e 5.6, onde o tempo de execução para $m = 3073$ no nó "korky" aumenta exponencialmente relativamente ao tempo obtido para $m = 2049$.

No caso de $m = 4097$, constata-se que a matriz A_m não é armazenável na sua totalidade na memória *RAM* de nenhum dos nós, obtendo-se um melhor tempo para o nó que tem menos memória *RAM* disponível mas que tem um processador mais rápido e mais memória *'cache'*.

Depois desta análise, fica-se com a impressão que a memória *RAM* é um factor que poderá influenciar bastante os tempos obtidos na versão paralela do algoritmo para certos valores de m.

Para ter uma visão integrada dos valores representados nas Tabelas 5.5 e 5.6 representamo-los graficamente nas Figuras 5.4 e 5.5, respectivamente.

Numa primeira análise destes gráficos, chega-se à conclusão que, tanto no CASO I como no CASO II, o tempo real do cálculo do produto diminui ao acrescentar mais processadores.

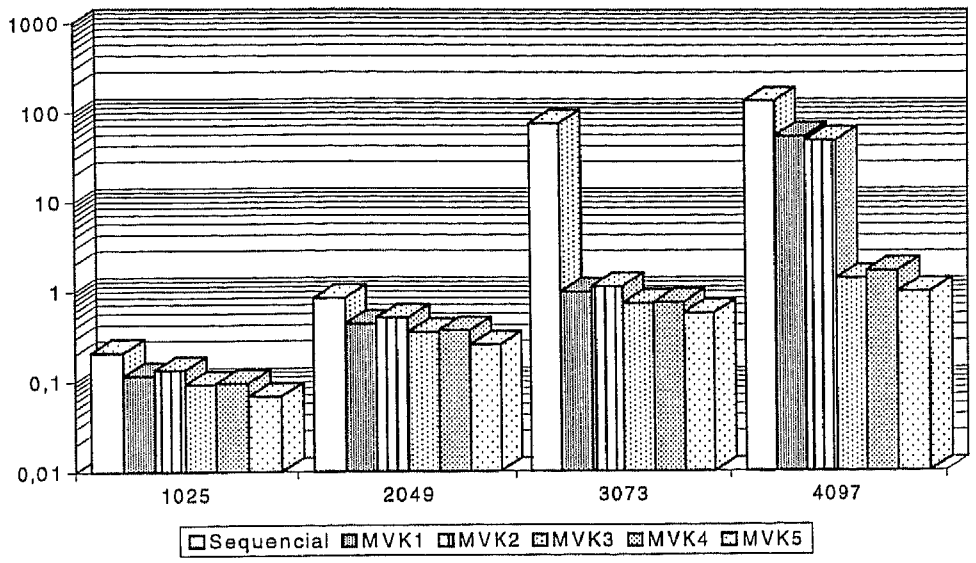

Figura 5.4 - Tempos reais, em segundos, correspondentes a t_p para o CASO I.

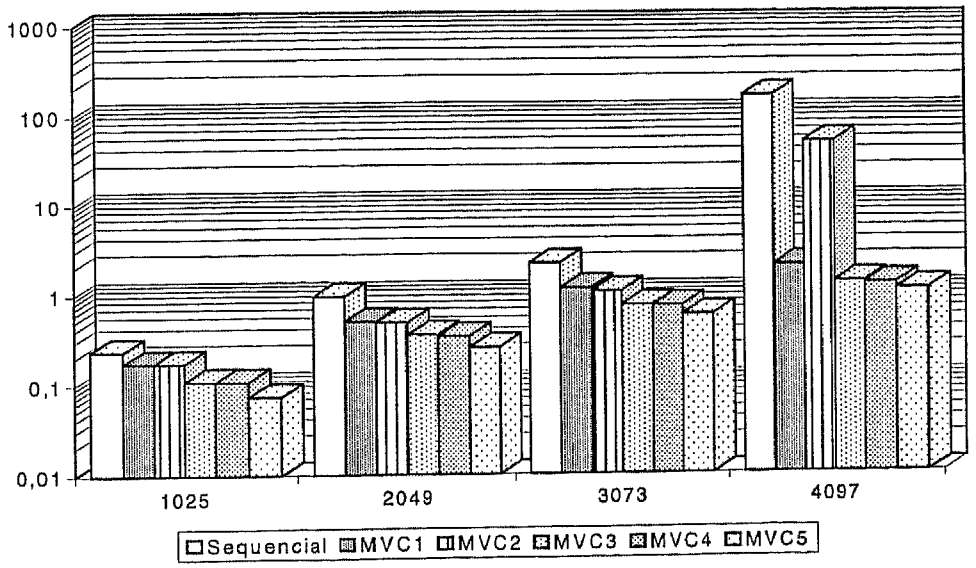

Figura 5.5 - Tempos reais, em segundos, correspondentes a t_p para o CASO II.

Ao analisar os tempos obtidos para as máquinas virtuais constituídas por dois nós, verifica-se que estes são semelhantes. No entanto, destacam-se dois pormenores:

- Se se compararem as duas máquinas virtuais constituídas por nós com características homogéneas, a *MVK*1 e a *MVC*1, obtêm-se melhores resultados

naquela que tem processadores mais rápidos, desde que a totalidade dos dados possa ser armazenada na memória *RAM* dos nós (a segunda coluna do gráfico da Figura 5.4, para cada valor de m, é sempre mais pequena que a segunda coluna do gráfico da Figura 5.5, a não ser para $m = 4097$). Esta ressalva é importante, pois como se pode verificar na Tabela 5.5, para $m = 4097$ e para a máquina virtual *MVK*1, o tempo obtido é de 54,315 segundos, o qual é muito superior ao tempo obtido para o mesmo valor de m para *MVC*1, - 2,061 segundos -, onde existe memória *RAM* suficiente para armazenar os dados;

- Outro facto interessante, é constatar que os tempos obtidos para *MVK*2 são ligeiramente superiores aos obtidos para *MVK*1 (a segunda coluna do gráfico da Figura 5.4, para cada valor de m, é sempre mais pequena que a terceira coluna do mesmo gráfico, a não ser para $m = 4097$). Este facto deriva da heterogeneidade a nível da velocidade dos processadores da máquina virtual *MVK*2, onde o processador do escravo sendo mais lento vai atrasar a execução do cálculo. Por exemplo, para $m = 3073$, obteve-se o seguinte tempo para o cálculo do produto A_1X- Tabela 5.7:

	*MVK*1	*MVK*2
$A_1 X$	0,953	1,084

Tabela 5.7 - Tempos reais, em segundos, correspondentes a A_1X em *MVK*1 e *MVK*2.

As máquinas virtuais constituídas por 3 processadores são todas heterogéneas. Por essa razão, é de supor que se obtenham tempos semelhantes, tanto no CASO I como no CASO II, dado que, conforme foi referido na Subsecção 3.3.2, são os processadores mais lentos que condicionam o tempo de execução dum programa paralelo. Note-se que, *MVK*3 e *MVC*4 são constituídas por elementos com as mesmas características; e por sua vez *MVK*4 e *MVC*3 também. Com efeito, se compararmos os tempos obtidos nas 4 máquinas virtuais constituídas por três nós, constata-se que os tempos obtidos são semelhantes (as quartas e quintas colunas dos gráficos das Figuras 5.4 e 5.5 são aproximadamente do mesmo tamanho, excepto no caso de $m = 4097$,

onde a quinta coluna do gráfico da Figura 5.4 é maior que as outras). Se se detalhar, verifica-se que:

- Os tempos obtidos para *MVK*3 são ligeiramente inferiores aos tempos obtidos na máquina virtual *MVK*4. Isto deve-se, provavelmente, ao facto dos escravos da máquina virtual *MVK*3 correrem em nós com processadores de velocidades diferentes, sendo o mais rápido igual ao do mestre. Assim, um dos escravos acaba o cálculo parcial do produto aproximadamente ao mesmo tempo que o mestre podendo enviar logo de seguida os resultados. Isto é, o mestre não fica bloqueado à espera dos resultados dos escravos como no caso da máquina *MVK*4 onde os escravos correm em nós que têm processadores iguais mas mais lentos que o do mestre;

- Constata-se, também, que os tempos obtidos na máquina virtual *MVC*3 são ligeiramente superiores aos da máquina virtual *MVC*4. Note-se que, para *MVC*3 os dois escravos correm em nós com características diferentes, ao contrário dos escravos da máquina virtual *MVC*4. Supõe-se, que, por essa razão, no caso de *MVC*4, o mestre quando acaba de calcular o seu produto parcial pode logo de seguida receber os produtos parciais dos escravos, visto que estes já os calcularam e, provavelmente, já os enviaram. No caso de *MVC*3, o mestre acaba o seu cálculo parcial aproximadamente ao mesmo tempo que um dos escravos, podendo este facto, eventualmente, atrasar a recepção dos resultados parciais dos escravos.

A última observação a fazer diz respeito às máquinas virtuais com 4 processadores, a *MVK*5 e a *MVC*5, onde a única diferença está no nó onde corre o programa mestre. Como se pode verificar obtêm-se, também, tempos semelhantes (as sextas colunas dos gráficos referidos atrás são do mesmo tamanho). No entanto, verifica-se, pelas Tabelas 5.5 e 5.6, que quando o programa mestre corre no nó "korky" os tempos obtidos são ligeiramente melhores. Supõe-se que a justificação para este facto, se encontra na recepção dos resultados parciais dos escravos por parte do mestre.

Depois deste estudo exaustivo dos tempos obtidos no cálculo do produto de matrizes do tipo A_mX vai analisar-se os tempos obtidos na execução do subprograma DCFNY, o qual conforme foi já referido na Secção 5.1, envolve 6 produtos do tipo A_mX.

As Tabelas 5.8 e 5.9 apresentam os tempos obtidos. Verifica-se que o tempo t_{dcfny} obtido para os vários valores de m para os dois casos estudados, CASO I e II, é aproximadamente igual a 6 vezes o tempo t_p. Este facto permite que todas as conclusões tiradas para as tabelas anteriores podem ser transportadas para aqui.

m	Sequencial	*MVK*1	*MVK*2	*MVK*3	*MVK*4	*MVK*5
1025	1,285	0,747	0,855	0,621	0,637	0,513
2049	5,087	2,695	3,166	2,235	2,330	1,678
3073	420,798	6,067	7,241	4,832	4,785	3,901
4097	781,852	327,841	297,257	8,910	10,742	6,566

Tabela 5.8 - Tempos reais, em segundos, correspondentes a t_{dcfny} para o CASO I.

m	Sequencial	*MVC*1	*MVC*2	*MVC*3	*MVC*4	*MVC*5
1025	1,464	1,121	1,119	0,739	0,731	0,536
2049	5,834	3,155	3,086	2,292	2,207	1,729
3073	13,348	7,336	6,640	4,840	4,843	4,184
4097	976,801	13,006	387,348	8,486	8,210	6,708

Tabela 5.9 - Tempos reais, em segundos, correspondentes a t_{dcfny} para o CASO II.

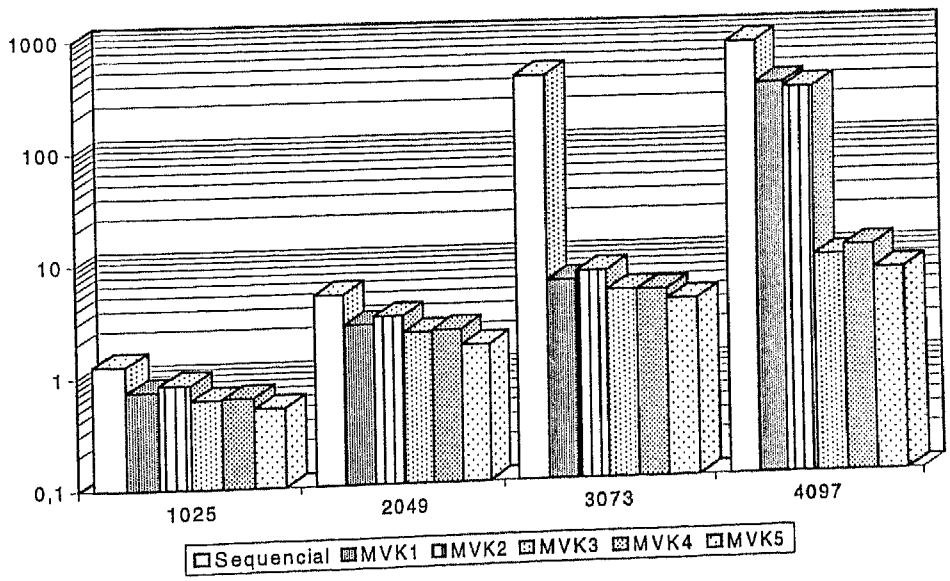

Figura 5.6 - Tempos reais, em segundos, correspondentes a t_{dcfny} para o CASO I.

Da mesma forma, se se comparar os gráficos das Figuras 5.6 e 5.7 com os gráficos das Figuras 5.4 e 5.5, respectivamente, constata-se que apresentam um mesmo padrão.

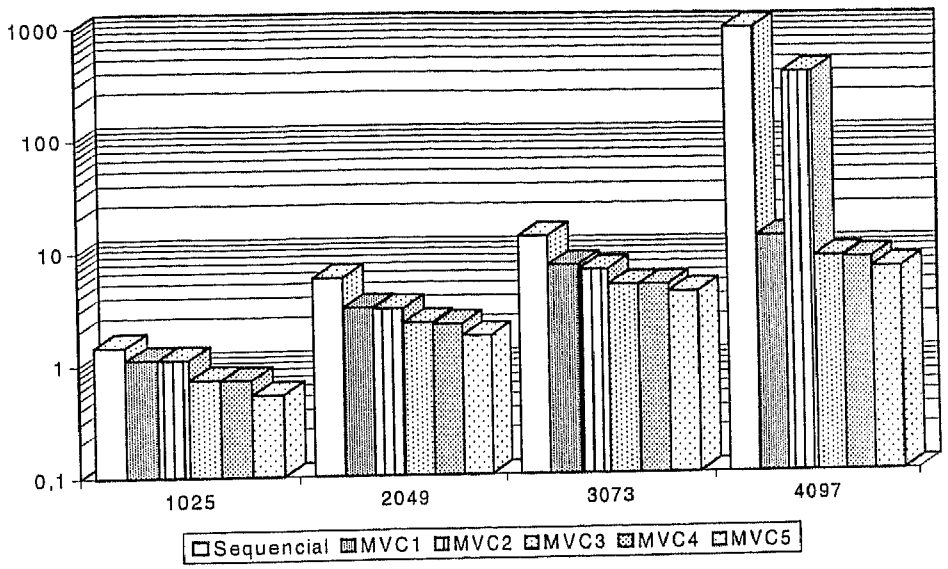

Figura 5.7 - Tempos reais, em segundos, correspondentes a t_{dcfny} para o CASO II.

Seguidamente, faz-se uma análise do desempenho deste algoritmo.

5.3 - Desempenho

As medidas de desempenho, o *"speedup"* e a eficiência, foram definidas matematicamente da seguinte forma:

Define-se *"speedup"*, S_p, pela expressão [Quinn87]:

$$S_p = \frac{T_1}{T_p} \qquad (5.8),$$

e, a eficiência E_p, pela expressão[Quinn87]:

$$E_p = \frac{S_p}{p} \qquad (5.9),$$

onde:

- T_1 é o tempo que demora a execução dum programa num só processador;
- T_p é o tempo que demora a execução dum programa em p processadores.

A melhor eficiência possível é 1, implicando que o melhor *"speedup"* é tal que $S_p = p$ [Hwang93].

A eficiência mínima corresponde ao caso do programa ser executado sequencialmente num só processador. A eficiência máxima é alcançada quando todos os p processadores estão sempre a ser utilizados ao longo do período de execução do programa.

As Tabelas 5.10 e 5.11 representam os valores de *"speedup"* obtidos na paralelização do produto de matrizes do tipo A_mX para os vários valores de m, para o CASO I e II, respectivamente. Estes valores de *"speedup"* foram calculados a partir dos melhores tempos obtidos na versão sequencial para cada valor de m.

m	*Sequencial*	*MVK1*	*MVK2*	*MVK3*	*MVK4*	*MVK5*
1025	1	1,8	1,6	2,3	2,2	3,1
2049	1	1,9	1,7	2,4	2,3	3,3
3073	1	**2,2**	2,0	3,0	3,0	3,9
4097	1	**2,5**	**2,8**	**97,7**	**80,5**	**136,8**

Tabela 5.10 - *"Speedups"* obtidos no produto A_mX para o CASO I.

m	*Sequencial*	*MVC1*	*MVC2*	*MVC3*	*MVC4*	*MVC5*
1025	1	1,2	1,2	1,9	1,9	2,8
2049	1	1,7	1,7	2,3	2,4	3,2
3073	1	1,9	**2,1**	2,9	3,0	3,8
4097	1	**66,2**	**2,7**	**102,2**	**106,6**	**123,8**

Tabela 5.11 - *"Speedups"* obtidos no produto A_mX para o CASO II.

Todos os valores de *"speedup"* superiores ao número de processadores correspondem a falsos *"speedups"*, pois o *"speedup"* máximo é dado por $S_p = p$ (estes estão assinalados na Tabela 5.11 a carregado). Estes resultaram da falta de memória

RAM que afectou o tempo de execução da versão sequencial. Os falsos *"speedups"* para $m = 3073$, para os dois casos, supõe-se que resulta de se ter escolhido o tempo da versão sequencial correspondente ao nó "crazy".

Da observação das Tabelas 5.10 e 5.11, conclui-se que se obtiveram valores de *"speedup"* muito bons e semelhantes, nos dois casos considerados. Com efeito, podem transportar-se para aqui as conclusões tiradas sobre os tempos obtidos no cálculo do produto A_mX. Acrescente-se apenas, que a maior diferença nos valores de *"speedup"* obtidos, para os dois casos, verifica-se para $m = 1025$, sendo os do CASO I superiores aos do CASO II.

A partir das Tabelas 5.10 e 5.11 construíram-se os gráficos das Figuras 5.8 e 5.9, os quais representam a eficiência obtida na paralelização do produto de matrizes do tipo A_mX para os vários valores de m, para os CASOS I e II, respectivamente.

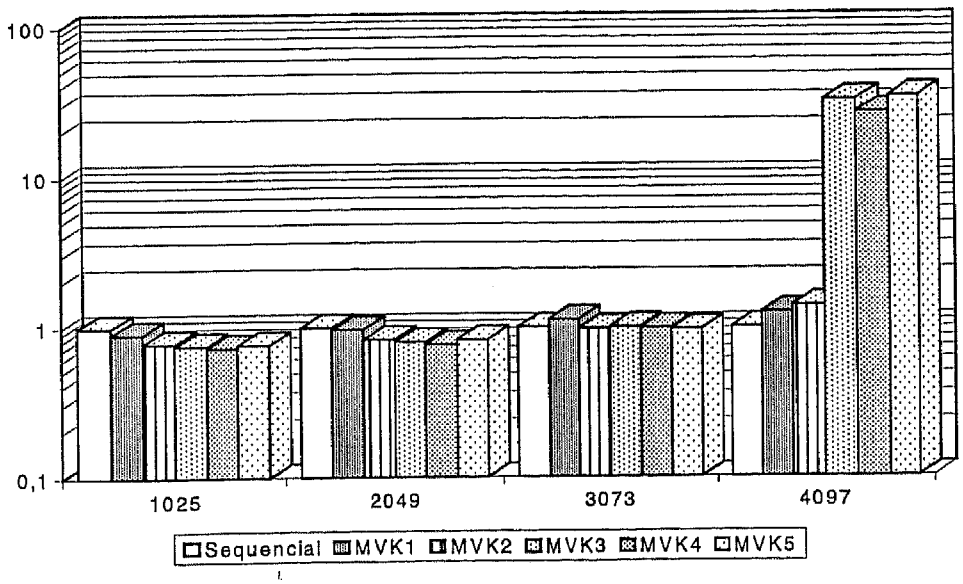

Figura 5.8 - Eficiência obtida no produto A_mX para o CASO I.

Da observação dos referidos gráficos conclui-se que tanto num caso como no outro, à medida que se aumenta o valor de m, a eficiência tende para 1. Ora, este facto, tem a ver com o quociente tempo de computação/tempo de comunicação. Para valores de m baixos, o tempo de cálculo é pequeno pelo que o tempo de comunicação é preponderante face a ele. Em contra partida, para valores de m grandes, o tempo de cálculo vai ser preponderante relativamente ao tempo de comunicação. Donde o

quociente tempo de computação/tempo de comunicação vai aumentar à medida que o m cresce.

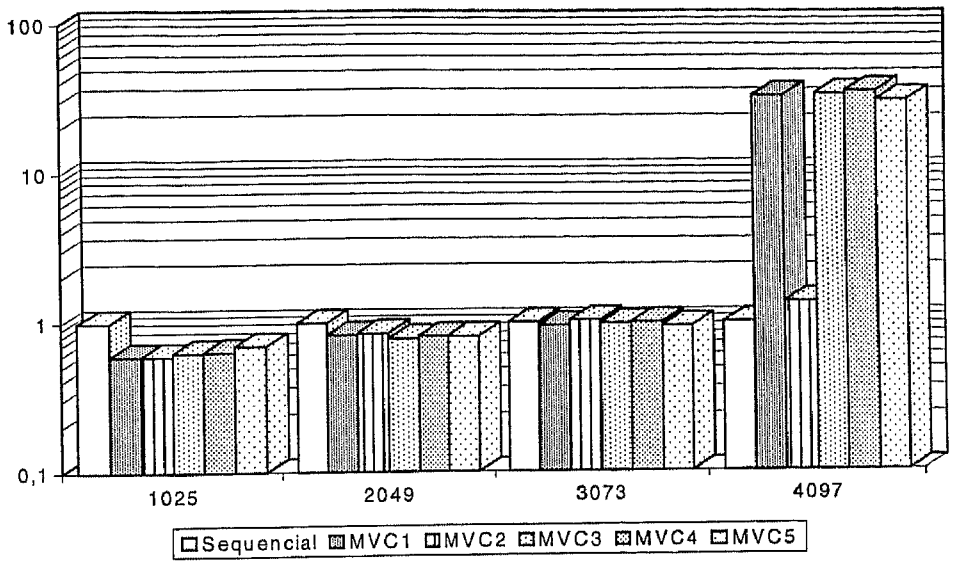

Figura 5.9 - Eficiência obtida no produto $A_m X$ para o CASO II.

Ainda relativamente às Figuras 5.8 e 5.9, verifica-se, que se obtêm eficiências semelhantes. Contudo, as eficiências encontradas no CASO I são ligeiramente melhores que as do CASO II (as colunas da Figura 5.8 são ligeiramente maiores que as colunas da Figura 5.9). As colunas das referidas figuras que são superiores a 1 correspondem aos falsos *"speedups"* mencionados atrás.

Um outro ponto que vale a pena investigar é se se observa um comportamento semelhante ao descrito, no caso do subprograma DCFNY. Nesse sentido, apresentam-se nas Tabelas 5.12 e 5.13 os valores de *"speedup"* encontrados.

m	*Sequencial*	*MVK1*	*MVK2*	*MVK3*	*MVK4*	*MVK5*
1025	1	1,7	1,5	2,1	2,0	2,5
2049	1	1,9	1,6	2,3	2,2	3,0
3073	1	**2,2**	1,8	2,8	2,8	3,4
4097	1	**2,4**	**2,6**	**87,7**	**72,8**	**119,1**

Tabela 5.12 - *"Speedups"* obtidos no subprograma DCFNY para o CASO I.

m	*Sequencial*	*MVC1*	*MVC2*	*MVC3*	*MVC4*	*MVC5*
1025	1	1,1	1,1	1,7	1,8	2,4
2049	1	1,6	1,6	2,2	2,3	2,9
3073	1	1,8	2,0	2,8	2,8	3,2
4097	1	**60,1**	2,0	**92,1**	**95,2**	**116,6**

Tabela 5.13 - *"Speedups"* obtidos no subprograma DCFNY para o CASO II.

Note-se, que neste caso, também há falsos *"speedups"*, os quais derivam dos mencionados atrás.

Os gráficos das Figuras 5.10 e 5.11 representam a eficiência correspondente.

Por comparação com os gráficos anteriores verifica-se que apresentam o mesmo padrão. Porém, as eficiências são ligeiramente inferiores às correspondentes aos produtos A_mX. A justificação para este facto, encontra-se na inactividade dos escravos entre os cálculos dos seus produtos parciais. Isto é, o mestre no subprograma DCFNY efectua cálculos entre os produtos A_mX, e por isso, os escravos vão ter um tempo de espera até que o mestre termine os cálculos intermédios e lhes envie mais trabalho (ver Figura 5.3).

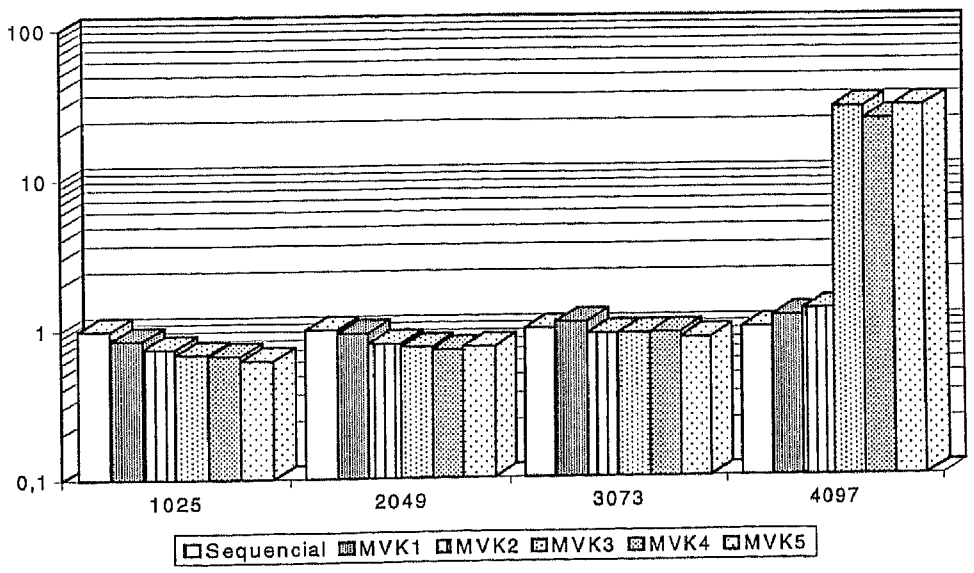

Figura 5.10 - Eficiência obtida no subprograma DCFNY para o CASO I.

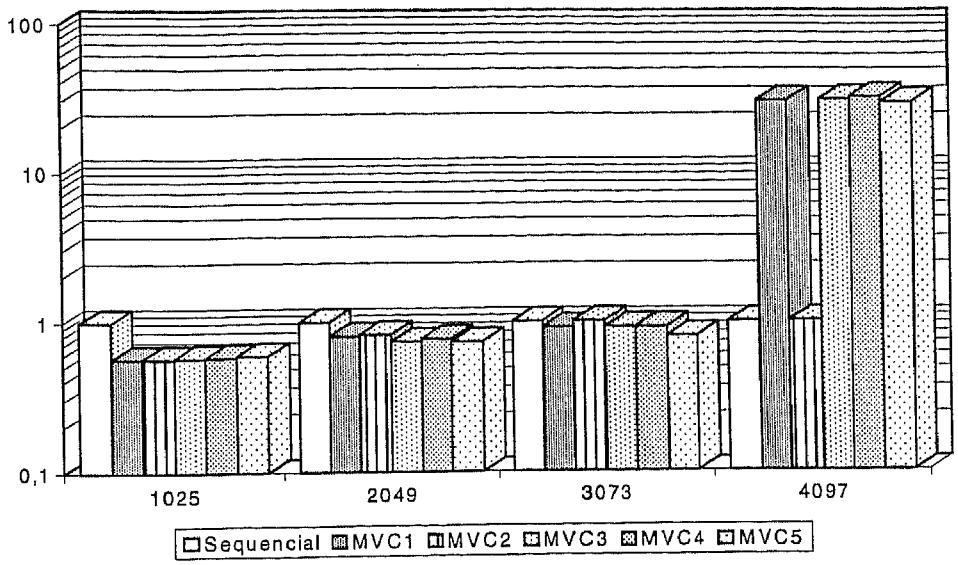

Figura 5.11 - Eficiência obtida no subprograma DCFNY para o CASO II.

Dados estes resultados elegeu-se a melhor máquina virtual a nível de desempenho para o algoritmo em causa em função do valor de m e do número de processadores - Tabela 5.14:

m	2	3	4
1025	*MVK*1	*MVK*3	*MVK*5
2049	*MVK*1	*MVK*3 e *MVC*4	*MVK*5
3073	*MVK*1	*MVK*3, *MVK*4, *MVC*3 e *MVC*4	*MVK*5
4097	*MVC*1	*MVC*4	*MVK*5

Tabela 5.14 - Máquina virtual onde foi obtido o melhor desempenho para cada valor de m.

Consequentemente, foram escolhidas preferencialmente as máquinas virtuais constituídas por nós com processadores mais rápidos, com excepção do caso de m = 4097 por razões que têm a ver com as diferenças nas capacidades de memória *RAM* dos nós.

Refira-se também, que os falsos *"speedups"* obtidos, demonstram que de facto, quando um problema não é resolúvel numa arquitectura sequencial, dada a dimensão do problema em função das capacidades da máquina, vale, efectivamente, a pena, paralelizar o problema.

Os resultados descritos nesta subsecção são bastante satisfatórios. Contudo poder-se-ia ter partido de uma divisão diferente da discretização da matriz A_m, por exemplo uma divisão por blocos de colunas ou por blocos. A próxima subsecção vai encarregar-se deste estudo.

5.4 - Outras Distribuições

Como a matriz A_m é densa, consideraram-se as seguintes distribuições da discretização da matriz A_m ilustradas nas Figuras 5.12 e 5.13:

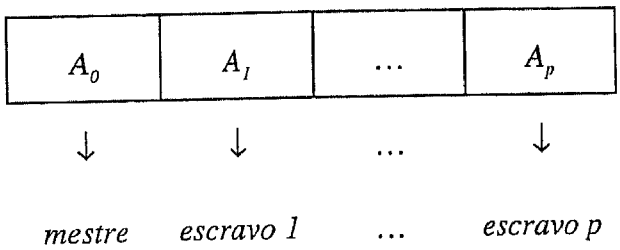

Figura 5.12 - Distribuição dos dados pelos *p* processadores em blocos de colunas.

mestre A_0	*escravo 1* A_1	...	*escravo i* A_i
...	...	...	...
escravo l A_l	*escravo l+1* A_{l+1}	...	*escravo p* A_p

Figura 5.13 - Distribuição dos dados pelos *p* processadores por blocos.

Neste estudo apenas se considerou o caso da máquina virtual *MVK5* pois, o propósito não é fazer um estudo exaustivo para cada uma das máquinas virtuais, mas sim ilustrar o comportamento do algoritmo em causa para estas distribuições.

O padrão de comunicação/computação ilustrado na Figura 5.3 mantém-se para estas duas distribuições. Contudo, o mestre tem que efectuar alguns cálculos adicionais. Concretamente, ele tem que somar os produtos parciais que receber dos escravos. No caso dos blocos de colunas, o mestre soma todos os resultados parciais;

no caso dos blocos soma dois a dois, pois só foi considerada a máquina virtual constituída por 4 processadores.

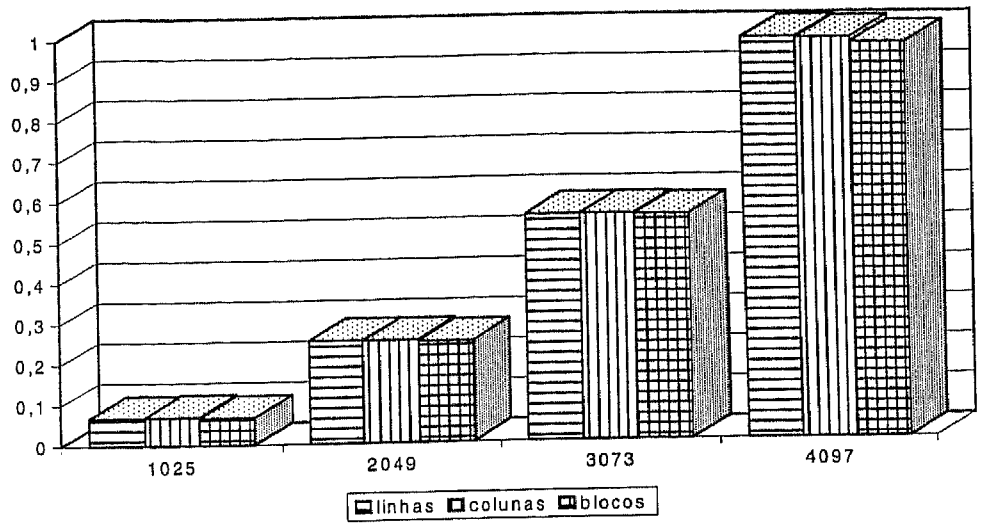

Figura 5.14 - Tempos reais, em segundos, do produto A_mX correspondentes às diferentes divisões da matriz A_m pelos p processadores.

Constata-se, pela Figura 5.14 que não existem diferenças significativas nos tempos dos produtos A_mX para as diferentes distribuições dos dados pelos p processadores. Aliás, como acontece com o tempo de execução correspondente ao subprograma DCFNY dado pelo gráfico da Figura 5.15:

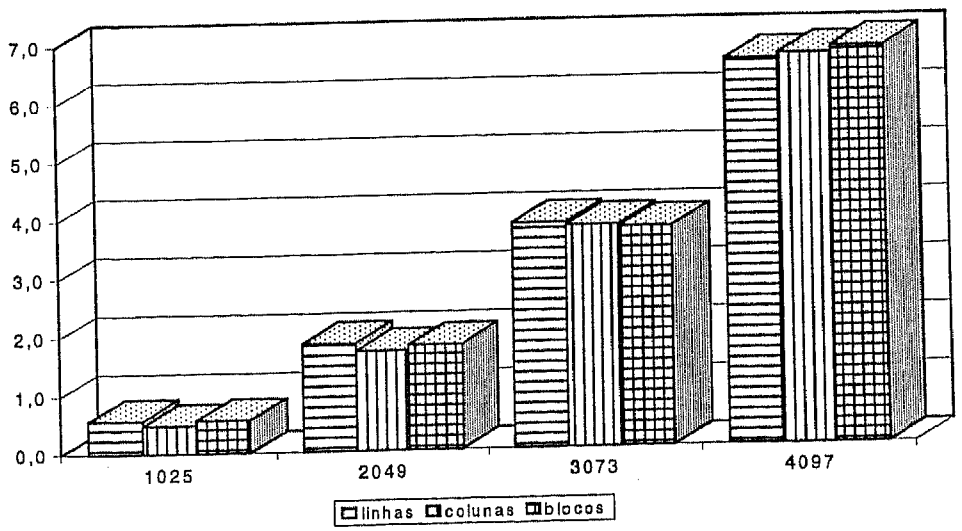

Figura 5.15 - Tempos reais, em segundos, do subprograma DCFNY correspondentes às diferentes divisões da matriz A_m pelos p processadores.

Seguidamente, estudou-se o desempenho encontrado para o algoritmo em causa para as diferentes distribuições. As Tabelas 5.15 e 5.16 ilustram os valores de *"speedup"* obtidos tanto para o cálculo do produto $A_m X$ como para o cálculo do subprograma DCFNY. Observando as referidas tabelas constata-se que não existem diferenças significativas para as distribuições consideradas.

m	Linhas	Colunas	Blocos
1025	3,1	3,0	3,1
2049	3,3	3,3	3,4
3073	3,9	3,9	3,9
4097	137,4	137,7	139,5

Tabela 5.15 - *"Speedups"* obtidos no cálculo do produto $A_m X$ em *MVK5* para as distribuições consideradas.

m	Linhas	Colunas	Blocos
1025	2,5	2,7	2,3
2049	3,0	3,0	2,8
3073	3,4	3,5	3,5
4097	119,1	116,4	114,4

Tabela 5.16 - *"Speedups"* obtidos no cálculo do subprograma DCFNY em *MVK5* para as distribuições consideradas.

5.5 - Optimização

Conforme foi já atrás referido, o que torna o algoritmo pesado são os produtos do tipo $A_m X$. Como a matriz A_m é conhecida à partida, a carga computacional destes produtos pode ser atribuída aos processadores à priori.

O facto da *'Farm'* ser constituída por máquinas heterogéneas, nomeadamente a nível da velocidade dos processadores, torna possível melhorar o desempenho do programa através de um esquema de balanceamento de carga.

Por essa razão, ensaiou-se um esquema de balanceamento estático de carga de modo a equilibrar o trabalho atribuído a cada processador de acordo com a velocidade dos mesmos.

O trabalho a atribuir a cada tarefa foi estabelecido da seguinte forma:

Dada a carga total c_T queremos distribuí-la pelos p processadores de forma a cada processador trabalhar o mesmo tempo, isto é, de forma a minimizar o tempo inactivo de cada processador. A carga c_i a atribuir a cada processador tem de ser proporcional à velocidade do processador v_i, isto é, $c_i = k*v_i$, com $k > 0$.

Donde:

$$c_T = k*\sum_{i=1}^{p} v_i \Rightarrow k = \frac{c_T}{\sum_{i=1}^{p} v_i} \qquad (5.10),$$

portanto, a carga a atribuir a cada processador é dada por:

$$c_i = \frac{c_T}{\sum_{j=1}^{p} v_j} * v_i \qquad (5.11).$$

Seja v_1 a velocidade dos processadores dos nós "korky" e "alf" e v_2 a velocidade dos processadores dos nós "crazy" e "tom". De acordo com o Capítulo 4, a relação entre elas é de 1,16, isto é, $v_1 = 1{,}16*v_2$.

Como se pretende dividir a matriz A_m em blocos de linhas, da Equação 5.6 resulta que:

$$nlin_i = \frac{m}{\sum_{j=1}^{p} v_j} v_j \qquad (5.12).$$

Assim, por exemplo, o número de linhas do bloco de linhas A_i da matriz A_m a atribuir ao processador i, no caso de $m = 4097$ e na máquina virtual *MVK5*, é ilustrado na Tabela 5.17:

	Carga igual (n.º de linhas de A_m)	Carga balanceada (n.º de linhas de A_m)
"korky"	1025	1101
"alf"	1024	1100
"tom"	1024	948
"crazy"	1024	948

Tabela 5.17 - Atribuição do número de linhas da matriz A_m a cada processador em função das velocidades dos mesmos no caso de m = 4097 e no caso de *MVK5*.

Estabelecido o trabalho a atribuir a cada processador, reestruturou-se o algoritmo por forma a introduzir este balanceamento de carga. Neste estudo excluiu-se o CASO II visto que, em geral, se obtiveram melhores desempenhos nas máquinas virtuais do CASO I. O caso de m = 4097 só foi considerado quando, após o balanceamento de carga, o problema cabia na memória *RAM* das máquinas da *'Farm'*. A máquina virtual *MVK1*, também foi excluída, pois é constituída por nós homogéneos.

Os resultados obtidos no cálculo do produto estão representados na Tabela 5.18:

m	*MVK2*	*MVK3*	*MVK4*	*MVK5*
1025	0,120	0,080	0,084	0,065
2049	0,459	0,305	0,319	0,242
3073	1,034	0,684	0,715	0,527
4097	—	—	—	0,932

Tabela 5.18 - Tempos reais, em segundos, obtidos no cálculo do produto A_mX com balanceamento de carga.

Se se comparar estes resultados com os obtidos na Tabela 5.5 da Secção 5.2, verifica-se que estes são melhores, tal como era de supor. Vai agora analisar-se, se de facto, o trabalho foi bem equilibrado nas várias máquinas virtuais e para os vários valores m. As Tabelas de 5.19 a 5.22 representam a percentagem de utilização de cada processador para cada uma das máquinas virtuais consideradas.

m	1025	2049	3073
"korky"	50,29	50,65	50,41
"tom"	49,71	49,35	49,59

Tabela 5.19 - Percentagem de utilização de cada processador no cálculo do produto A_mX com balanceamento de carga para *MVK2*.

m	1025	2049	3073
"korky"	33,09	33,99	33,93
"tom"	31,80	33,16	33,09
"crazy"	35,10	32,85	32,98

Tabela 5.20 - Percentagem de utilização de cada processador no cálculo do produto A_mX com balanceamento de carga para *MVK3*.

m	1025	2049	3073
"korky"	32,21	33,92	33,91
"tom"	34,34	33,09	33,06
"crazy"	33,46	33,00	33,03

Tabela 5.21 - Percentagem de utilização de cada processador no cálculo do produto A_mX com balanceamento de carga para *MVK4*.

m	1025	2049	3073	4097
"korky"	23,86	25,44	25,41	25,09
"alf"	21,94	24,82	24,97	25,08
"tom"	27,06	24,89	24,86	24,97
"crazy"	27,14	24,85	24,76	24,85

Tabela 5.22 - Percentagem de utilização de cada processador no cálculo do produto A_mX com balanceamento de carga para *MVK5*.

Da análise destas tabelas chega-se à conclusão que o trabalho foi bem equilibrado, havendo um pequeno desequilíbrio apenas para $m = 1025$ nas máquinas virtuais *MVK3*, *MVK4*, e *MVK5*.

Com efeito os valores de *"speedup"* obtidos, tanto para o cálculo do produto como para o cálculo do subprograma DCFNY, são melhores. Note-se que os valores de *"speedup"* obtidos na máquina virtual *MVK*2 são inferiores aos da máquina virtual *MVK*1, como era de supor.

A partir das Tabelas 5.24 e 5.25 construíram-se os gráficos das Figuras 5.16 e 5.17 as quais representam os valores da eficiência encontrada.

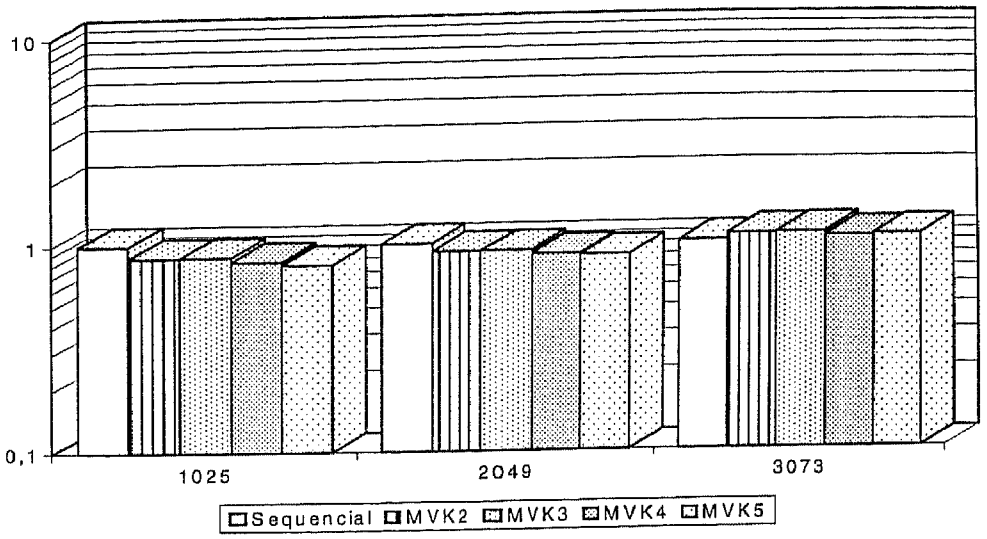

Figura 5.16 - Eficiência obtida no cálculo do produto $A_m X$ após o balanceamento de carga.

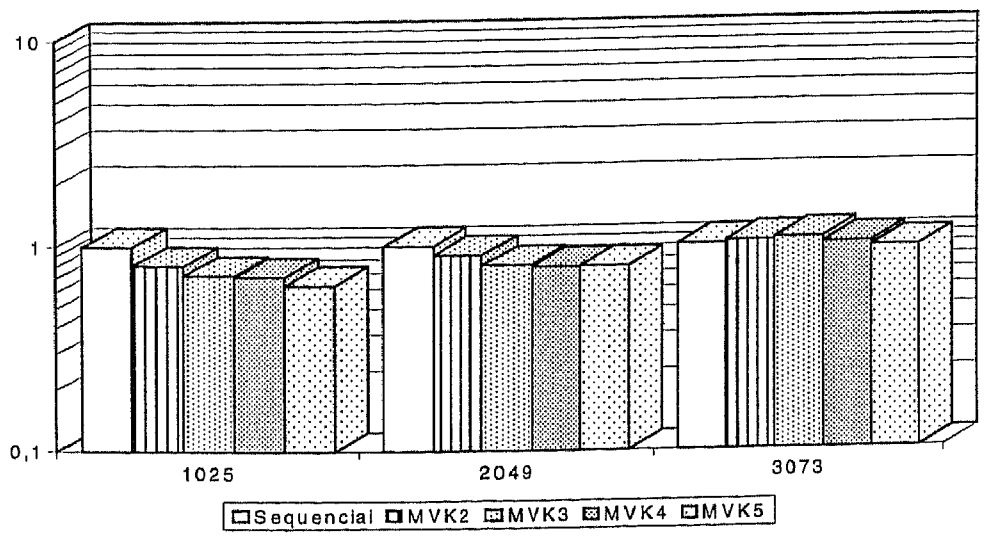

Figura 5.17 - Eficiência obtida no subprograma DCFNY após o balanceamento de carga.

Constata-se que após o balanceamento de carga as colunas se aproximam mais de 1.

5.6 - Síntese

Este capítulo ilustrou algumas das possibilidades da *'Farm'* na paralelização do algoritmo proposto no Capítulo 2.

A primeira parte deste capítulo fez um estudo da versão sequencial deste problema seguindo as opções tomadas em [Almeida93]. Posteriormente, descreveu a estratégia de paralelização pela qual se optou, em função do estudo feito na primeira parte. Seguidamente, apresentou os resultados obtidos na paralelização do algoritmo na *'Farm'* e estudou o desempenho encontrado a nível de *"speedup"* e eficiência. Depois, estudou, comparativamente, o desempenho obtido no algoritmo, a nível de *"speedup"* e eficiência, para as distribuições dos dados consideradas: por linhas, por blocos de colunas e por blocos. Por fim, propõe um esquema de balanceamento estático de carga com o intuito de optimizar o algoritmo em causa.

Capítulo 6

Conclusões

O propósito desta tese era ilustrar as possibilidades duma rede de computadores através de uma aplicação de cálculo de valores próprios por refinamento. Este objectivo foi atingido tomando como referência uma versão paralela deste problema para uma arquitectura de memória partilhada.

Dada a natureza do algoritmo proposto para a resolução deste problema, optou-se por um modelo de paralelização do tipo *MPMD*. Apesar do algoritmo ser bastante síncrono, o *PVM* comportou-se bem, não introduzindo grande entropia (isto é, *'overheads'* provocados pela comunicação) na execução do programa paralelo visto que os resultados obtidos foram bastante bons. O balanceamento de carga proposto para optimização do referido problema prova que, dada uma rede de computadores heterogénea, concretamente a nível da velocidade dos processadores, conduz a melhorias no desempenho da aplicação em causa.

Esta tese conclui que valeu a pena paralelizar o problema em causa numa arquitectura de memória distribuída, concretamente numa *'Farm'* de *'DECAlphas AXP'*. Os desempenhos obtidos no Capítulo 5 demonstram claramente os benefícios desta paralelização neste tipo de arquitecturas: os valores de *"speedup"* estão muito próximos do número de processadores existentes e, consequentemente, a eficiência obtida é próxima da máxima.

Os *"speedups"* superiores ao número de processadores existentes (os falsos *"speedups"* referidos no Capítulo 5) mostram, que vale a pena também, paralelizar a aplicação quando as características das máquinas são limitadas para a resolução do problema em causa, isto é, para grandes dimensões da matriz A_m.

No seguimento desta tese, e como trabalho futuro, afigura-se interessante desenvolver um algoritmo que assente noutro tipo de modelo, nomeadamente no modelo *SPMD*. Seria também interessante construir um algoritmo menos síncrono podendo, desse modo, tirar eventualmente maior partido da arquitectura em causa. A comparação do desempenho obtido no modelo *MPMD* com o *SPMD* é uma boa perspectiva de trabalho futuro.

Referências Bibliográficas

[Ahues82] Ahues, M., d'Almeida, Telias, M.
"On the Defect Correction Method with Applications to Iterative Refinement Techniques".
IMAG, Raport Recherche, No 324.
Université de Grenoble.
1982.

[Ahues90] Ahues, M., Aranciba, S., Telias, M.
"Rayleigh - Schrödinger Series for Defective Spectral Elements of Compact Operators in Banach Spaces. First part: Theoretical Aspects".
Numerical Funct. Anal. and Optimiz. 11 (9&10), 839-850.
1990-1991.

[Almeida84] d'Almeida, F.
"*Problemas de Valores Próprios de Operadores Integrais Compactos. Métodos de Refinamento de Soluções Aproximadas*".
Tese de Doutoramento em Análise Numérica.
Faculdade de Ciências da Universidade do Porto.
Novembro 1984.

[Almeida93] d' Almeida, F.
"*Iterative refinement of eigenelements of compact integral operators using BLAS 3 routines on the ALLIANT VFX/80 and on the ALLIANT FX/2800*", Technical Report TR-PA-93-11. Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique (CERFACS).
1993.

[Anderson92] Anderson, E. et al.
"*LAPACK Users' Guide*".
Society for Industrial and Applied Mathematics [SIAM].
Philadelphia, 1992.

[Bartels72] Bartels, R. and Stewart, G.
"*Solution of the Equation* $AX + XB = C$".
Communications of the ACM, No. 15, Pp. 820-826.
1972.

[Barret93] Barret, R. et al.
"Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods".
Society for Industrial and Applied Mathematics [SIAM].
Philadelphia, 1993.

[Butler94] Butler, R. and Lusk, E.
"User's Guide to the p4 Parallel Programming System".
Technical Report ANL-92/17 (Revised).
Mathematics and Computer Science Division, Argonne National Laboratory.
April 1994.

[Carriero89] Carriero, N. and Gelernter, D.
"LINDA in Context".
Communications of the ACM. Vol. 32, No. 4. Pp. 444-458.
April 1989.

[Chandy91] Chandy, K. and Kesselman, C.
"Parallel Programming in 2001".
IEEE Software. November 1991. Pp. 11-20.

[Chatelin83] Chatelin, F.
"Spectral Approximation of Linear Operators".
Academic Press, New York,1983.

[Daydé91] Daydé, M., Duff, I.
"Use of Level 3 BLAS in LU Factorization in a Multiprocessing Environment on Three Vector Multiprocessors: the Alliant FX/80, the CRAYY-2 and the IBM 3090 VF".
The International Journal of Supercomputer Applications, Vol 5, No. 3, Pp.92-110, 1991.

[DEC500] DEC 3000 Model 500/500S AXP
"Owner's Guide"
Digital Equipment Corporation, Maynard, Massachusetts.

[DEC600] DEC 3000 Model 600/600S AXP
"Owner's Guide"
Digital Equipment Corporation, Maynard, Massachusetts.

[DECOSF/1] DEC OSF/1 Versão 3.0
"Technical Overview"
Digital Equipment Corporation, Maynard, Massachusetts.

[Dongarra88] Dongarra, J., DuCroz, S., Hammarling, S. and Hanson, R.
"An Extended Set of FORTRAN Basic Linear Algebra Subprograms".
ACM Transactions on Mathematical Software, Vol. 14 No. 1, Pp. 1-17.
March 1988.

[Dongarra90] Dongarra, J., DuCroz, S., Hammarling, S. and Duff, I.
"A Set of Level 3 Basic Linear Algebra Subprograms".
ACM Transactions on Mathematical Software, Vol. 16 No. 1, Pp. 1-17.
March 1990.

[DXML94] *"Digital eXtended Math Library Reference Manual"*.
Order N° AA-QONHB-TE.
Digital Equipment Corporation Maynard, Massachusetts.

[Duff86] Duff, I., Erisman, A. and Reid, J.
"Direct Methods for Sparse Matrices".
New York, Oxford University Press, 1986.

[Flower91] Flower, J., Kolawa, A. and Bharadwaj. S.
"The Express Way to Distributed Processing".
Supercomputing Review, Pp. 54-55.
May 1991.

[Geist94a] Geist, A., Beguelim, A., Dongarra, J., Jiang, W., Mancheck, R.
and Sunderam, V.
"PVM 3 User's Guide and Reference Manual".
Technical Report ORNL/TM-12187, Oak Ridge National Laboratory.
May 1994.

[Geist94b] Geist, A.
"PVM: A User's Guide and Tutorial for Networked Parallel Computing".
MIT Press, 1994.

[Goscinski91] Goscinski, A.
"Distributed Operating Systems: The Logical Design".
Addison-Wesley Publishers, Ltd., 1991.

[Hwang93] Hwang, K.
"Advanced Computer Architecture:Parallelism, Scalability, Programmability".
McGraw-Hill, Inc, 1993.

[Kamath94] Kamath, C., Ho, R. and Mantley, D. P.
"DXML: A High-performance Scientific Subroutine Library".
Digital Technical Journal, Vol. 6, N°3, Summer 1994.

[Kato66] Kato, T.
"Pertubation Theory for Linear Operators"
Springer Verlag, 1966.

[Lawson79] Lawson, C., Hanson, R., Kincaid, D. and Krogh, F.
"Basic Linear Algebra Subprograms for Fortran Usage".
ACM Transactions on Mathematical Software, Vol. 5 No. 3, Pp. 308-323.
June 1991.

[MacDonald94] MacDonald, N., Minty, E., Harding, T., Brown, S.
"Wrting Message Passing Parallel Programs with MPI".
Course Notes of a Two-Day Course.
Edinburgh Parallel Computing Centre.
The University of Edinburgh, 1994.

[Marques90] Marques, J. e Guedes, P.
"Fundamentos de Sistemas Operativos"
Editorial Presença, Colecção Informática e Computadores, 1990.

[MPIF94] Message Passing Interface Forum
"*MPI: A Message-Passing Interface Standard*".
International Journal of Supercomputer Aplications and High Performance Computing, 8(3/4), 1994.

[Peres95] Peres, J. e Vasconcelos, P.
"*Biblioteca de Tempos*".
Documento Técnico do Centro de Informática Prof. Correia de Araújo (CICA), a publicar.
Faculdade de Engenharia da Universidade do Porto.
Março 1995.

[Postel81a] Postel. J.
"*Transmission Control Protocol*".
Technical Report RFC 793, Information Sciences Institute.
September 1981.

[Postel81b] J. Postel.
"*User Datagram Protocol*".
Technical Report RFC 768. Information Sciences Institute,
September 1981.

[Quinn87] Quinn, M.
"*Designing Efficient Algorithms for Parallel Computers*".
McGraw Hill, Computer Science Series.
1987.

[Schmidt94] Schmidt, B. and Sunderam, V.
"*Empirical Analysis of Overheads in Cluster Environments*".
Concurrency: Pratice and Experience. Vol. 6, No. 1. Pp. 1-32.
February 1994.

[Souza94] Souza, R. J., Krishmakumar, P.G., Özveren, Simcoe, R. J., Spinney, B. A., Thomas, R. E., Walsh, R. J.
"*GIGAswitch System: A High-performance Packet-switching Platform*".
Digital Technical Journal, Vol. 6, N°1, Winter 1994.

[Sukup94] Sukup, F.
"*Efficiency Evaluation of Some Parallelization Tools on a Workstation Cluster Using the NAS Parallel Benchmarks*".
Technical Report ACPC/ TR 94-2, Computer Center,
Vienna University of Technology.
January 1994.

[SUN87] "*XDR: External Data Representation Standard*".
RFC 1014, Sun Microsystems, Inc.
June 1987.

[Sunderam90] Sunderam, V., Geist, G., Dongarra, J. and Manchek, R.
" *PVM: A Framework for Parallel Distributed Computing*".
Concurrency: Pratice and Experience, 2(4), Pp. 315-339.
December 1990.

[Sunderam94] Sunderam, V., Geist, G., Dongarra, J. and Manchek, R.
"*The PVM Computing System: Evolution, Experiences, and Trends*".
Parallel Computing, 1994.

[Tanenbaum92] Tanenbaum,Andrew S.
"*Modern Operating Systems*".
Prentice-Hall, Inc., 1992.

[Thévenin95] Thévenin, D., Behrendt, F., Maas, U. and Warnatz, J.
"*Simulation of Reacting Flows with a Portable Parallel Code Using Dynamic Load Balancing*".
Proceedings of High-Performance Computing and Networking, International Conference Exhibition, Pp 378-383, Milan, Italy.
May 1995.

[Turcotte93] Turcotte, L.
"*A Survey of Software Environments for Exploting Networked Computing Resources*".
Draft Report, Mississippi State University. Jackson, Mississippi.
January 1993.

[Vasconcelos94] Vasconcelos, Paulo B.
"*Experência com PVM e ScaLAPACK*".
Documento Técnico do Centro de Informática Prof. Correia de Araújo (CICA), CICA I01.
Faculdade de Engenharia da Universidade do Porto.
Junho 1994.

[Vieira91] Vieira, I.
"*Refinamento de Valores Próprios de Operadores Integrais ou Diferenciais*".
Tese de Mestrado em Engenharia Electrotécnica e de Computadores (Área de Sistemas).
Faculdade de Engenharia da Universidade do Porto.
Maio 1991.