



MINISTÉRIO DA CIÊNCIA E TECNOLOGIA

INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

INPE-10297-TDI/916

**UM PROCESSO OTIMIZADO DE PRODUÇÃO DE MAPAS DA
RADIÇÃO CÓSMICA DE FUNDO EM MICROONDAS**

José Oscar Fernandes

Dissertação de Mestrado do Curso da Pós-Graduação em Computação Aplicada,
orientada pelos Drs. Airam Jônatas Preto e Stephan Stephany,
aprovada em 27 de março de 2003

519.863

FERNANDES, J. O.

Um processo otimizado de produção de mapas da radiação cósmica de fundo em microondas / J. O. Fernandes. – São José dos Campos: INPE, 2003.

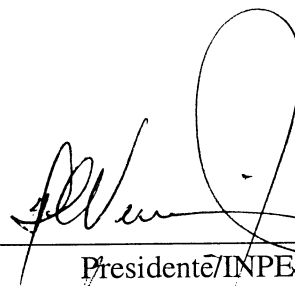
97p. – (INPE-10297-TDI/916).

1.Radiação Cósmica de Fundo em Microondas (RCFM). 2.Sistemas de processamento paralelo. 3.Redução de dados. 4.FORTRAN (linguagem de programação). 5.Programação de sistemas computacionais. I.Título.

Instituto Nacional de Pesquisas Espaciais - INPE / Sao Jose
dos Campos - Sao Paulo

Aprovado pela Banca Examinadora em cumprimento a requisito exigido para a obtenção do Título de **Mestre em Computação Aplicada.**

Dr. Haroldo Fraga de Campos Velho



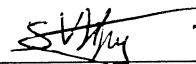
Presidente/INPE, SJCampos-SP

Dr. Airam Jônatas Preto



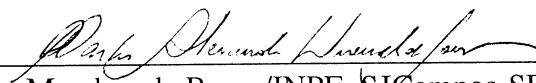
Orientador/INPE, SJCampos-SP

Dr. Stephan Stephany



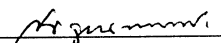
Orientador/INPE, SJCampos-SP

Dr. Carlos Alexandre Wuensche de Souza



Membro da Banca/INPE, SJCampos-SP

Dr. Newton de Figueiredo Filho



Membro da Banca
Convidado UNIFEI, Itajubá-MG

Candidato: José Oscar Fernandes

São José dos Campos, 27 de março de 2003.

“Estamos sós e sem desculpas,
é o que traduzirei dizendo que o homem está condenado a ser livre.
Condenado, porque não se criou a si próprio,
e no entanto livre, porque uma vez lançado ao mundo,
é responsável por tudo quanto fizer.”

(Jean Paul Sartre)

À Glêda e Raul,
dedico.

AGRADECIMENTOS

A meus pais, agradeço pelo incentivo e apoio em todas as etapas de minha vida.

A meus irmãos, que sempre estiveram comigo.

Aos Drs. Airam Jônatas Preto e Stephan Stephany, pela orientação e pelos anos de aprendizagem e colaboração.

Aos colegas do Setor de Lançamento de Balões, pela amizade e pela grande ajuda na realização deste curso.

Aos colegas do grupo de Cosmologia, pela infraestrutura e suporte oferecido para a realização deste estudo.

RESUMO

Este trabalho propõe uma estratégia de otimização de desempenho para o software de produção de mapas da Radiação Cósmica de Fundo em Microondas (RCFM), onde se busca a minimização do tempo de processamento utilizando uma arquitetura paralela de memória distribuída. Inicialmente, um código desenvolvido em Fortran foi portado para Fortran 90 e *High Performance Fortran* (HPF), em uma implementação baseada em paralelismo de dados, sendo executado em uma arquitetura paralela de memória compartilhada. O código foi convenientemente dividido em trechos e foi instrumentado para prover informações dos tempos de execução de cada trecho. A análise dos tempos de execução mostrou um “gargalo” de desempenho nas rotinas que implementam a convolução de matrizes, o que é feito por um algoritmo *Fast Fourier Transform* (FFT) bidimensional. Esta rotina é a maior consumidora de tempo de CPU devido à distribuição de dados entre os processadores. No algoritmo FFT, a matriz de convolução deve ser transposta. Tipicamente, um bloco de colunas desta matriz é atribuído a cada processador, mas no cálculo da matriz transposta, blocos de linhas da mesma matriz devem ser lidos. Isto é claramente, uma operação pouco eficiente quando se utiliza o HPF. A rotina FFT foi portada para *Message Passing Interface* (MPI), combinando paralelismo de dados e tarefas. A arquitetura paralela de memória distribuída utilizada é um *beowulf cluster* de 16 nós, sendo cada nó um computador Linux IA-32. O uso de MPI para melhorar o desempenho do HPF foi proposto por Foster et al. em “Double standards: bringing task parallelism to HPF via the Message Passing Interface” e foi utilizado em uma rotina 2D-FFT com o objetivo de proporcionar uma comunicação mais eficiente entre os nós no cálculo de uma matriz transposta. Este trabalho apresenta as avaliações do método utilizado, que pode ser aplicado a outras rotinas consumidoras de tempo de CPU. A análise dos tempos de execução fornece indícios para se conseguir um melhor balanceamento de carga e menor tempo de comunicação entre os nós.

AN OPTIMIZED PROCESS OF PRODUCTION OF COSMIC MICROWAVE BACKGROUND RADIATION MAPS

ABSTRACT

The current work describes the optimization and parallelization of the software for the production of Cosmic Microwave Background (CMB) radiation maps in a distributed memory architecture. Initially, the Fortran code was ported to Fortran 90 and to High Performance Fortran (HPF) and run in a SMP (Shared Memory Processor) machine. In order to provide timing information, calls to operating system timing routines were imbedded in the code. Analysis of timing information shows performance bottlenecks in the matrix convolution routine, which is done by a two-dimensional Fast Fourier Transform (FFT) algorithm. This is a time consuming routine due to data distribution among processors. Due to the FFT algorithm the convolution matrix has to be transposed. Typically, a block of columns of this matrix is assigned to each processor but, in order to calculate the transposed matrix, block of lines of the same matrix must be read. This was clearly an inefficient issue running HPF on that machine. The FFT routine was ported to Message Passing Interface (MPI) mixing task and data parallelism. The used distributed memory machine is a Beowulf cluster, each node being a Linux IA-32 computer. The use of MPI to enhance HPF performance was already proposed by Foster et al. in "Double standards: bringing task parallelism to HPF via the Message Passing Interface" and was used in the 2D FFT routine in order to provide a more efficient communication between nodes in the calculation of the transposed matrix. This work presents the evaluations of the used method, and the same approach could be used in other time consuming routines of the CMB code. The timing analysis provides clues to enhance load balancing and data communication between nodes.

SUMÁRIO

	Pág.
LISTA DE FIGURAS	
LISTA DE TABELAS	
LISTA DE SIGLAS E ABREVIATURAS	
CAPÍTULO 1 - INTRODUÇÃO	21
CAPÍTULO 2 - PRODUÇÃO DE MAPAS DA RCFM.....	25
2.1 - Formalismo tradicional para produção de mapas da RCFM.....	29
2.2 - Modelo do céu.....	32
CAPÍTULO 3 - PROGRAMAÇÃO PARALELA.....	37
3.1 - Arquiteturas de alto desempenho.....	38
3.2 - Arquiteturas de máquinas paralelas.....	39
3.3 - Paradigmas de programação paralela.....	45
3.4 - Ambientes de programação paralela.....	49
3.4.1 - Fortran 90/95.....	49
3.4.2 - High Performance Fortran – HPF.....	51
3.4.3 - Message Passing Interface – MPI.....	55
CAPÍTULO 4 - TÉCNICAS DE OTIMIZAÇÃO DE CÓDIGO.....	57
4.1 - Otimização de programas seqüenciais.....	58
4.2 - Monitoração de programas paralelos.....	64
4.3 - Instrumentação para medidas de tempo.....	65
CAPÍTULO 5 - OTIMIZAÇÃO DA PRODUÇÃO DE MAPAS DA RCFM.....	67
5.1 - Otimização da aplicação seqüencial.....	69
5.2 - Programação baseada em paralelismo de dados.....	74

5.2.1 - Convolução de matrizes usando FFT.....	74
5.2.2 - Execução em uma máquina paralela de memória compartilhada.....	77
5.2.3 - Execução em uma máquina paralela de memória distribuída.....	79
5.3 - Programação baseada em paralelismo de dados e tarefas.....	81
CAPÍTULO 6 - CONCLUSÕES.....	85
REFERÊNCIAS BIBLIOGRÁFICAS.....	89
APÊNDICE A - Trecho da subrotina usando paralelismo de dados.....	93
APÊNDICE B - Trecho da subrotina usando paralelismo de dados e tarefas.....	95

LISTA DE FIGURAS

	Pág.
2.1 - Esquema ilustrativo do processo de redução de dados da RCFM.....	25
2.2 - Plataforma HACME.....	27
2.3 - Mapas do experimento HACME e satélite COBE.....	28
2.4 - Estratégia de apontamento do telescópio.....	28
2.5 - Modelo do céu.....	32
2.6 - Matriz de apontamento.....	33
2.7 - Matriz do filtro <i>white noise</i>	33
2.8 - Matriz de correlação de ruído.....	34
2.9 - Matriz de covariância de ruído inversa.....	34
3.1 - Arquitetura paralela de memória compartilhada.....	41
3.2 - Arquitetura paralela de memória distribuída.....	44
3.3 - Uso de FORALL e INDEPENDENT.....	54
4.1 - Subrotina <i>timer</i>	66
4.2 - Instrumentação do programa.....	66
5.1 - Diagrama em blocos do programa <i>doicvm</i>	70
5.2 - Uso de operações de redução.....	71
5.3 - Convolução de matrizes usando 2D-FFT.....	75
5.4 - Implementação HPF de 2D-FFT.....	76
5.5 - Tempos de computação e comunicação em máquina de memória compartilhada.....	78
5.6 - Tempo da rotina de convolução de matrizes.....	80
5.7 - Tempos de computação e comunicação em máquina de memória distribuída.....	81
5.8 - Implementação HPF/MPI de 2D-FFT.....	82
5.9 - Comparativo do tempo de execução das implementações HPF e HPF/MPI..	84

LISTA DE TABELAS

	Pág.
2.1 - Requisitos computacionais para o algoritmo de produção de mapas.....	31
5.1 - Tempos de execução seqüencial das malhas internas.....	70
5.2 - Tempos de execução seqüencial do trecho 3.....	72
5.3 - Comparativo entre versões seqüenciais usando F77 e F90.....	73
5.4 - Execução seqüencial do trecho 5.....	73
5.5 - Comparação dos tempos da execução seqüencial e paralela usando HPF.....	77
5.6 - Medidas dos tempos da rotina de convolução.....	78
5.7 - Tempo total da rotina de convolução usando HPF.....	79
5.8 - Tempos parciais da rotina de convolução usando HPF.....	80
5.9 - Comparativo entre os tempos das rotinas HPF e HPF/MPI.....	83

LISTA DE SIGLAS E ABREVIATURAS

1D-FFT	one-dimensional FFT
2D-FFT	two-dimensional FFT
ACE	Advanced Cosmic Explorer
BEAST	Background Emission Anisotropy Scanning Telescope
CAP	Curso de Pós-Graduação em Computação Aplicada do INPE
CC-NUMA	Cache Coherent NUMA
CISC	Complex Instruction Set Computers
COBE	Cosmic Background Explorer
COMA	Cache-Only Memory Access
COW	Cluster of Workstations
CPU	Central Processing Unit
DAS	Divisão de Astrofísica do INPE
ESA	European Space Agency
FFT	Fast Fourier Transform
FFTW	The Fast Fourier Transforms in the West
FLOP	Floating Point Operation
GB	Giga Byte = 1.024 MB = 1.073.741.824 bytes
HACME	HEMT Advanced Cosmic Microwave Explorer
HEMT	High Electron-Mobility Transistor amplifier
HPF	High Performance Fortran
INPE	Instituto Nacional de Pesquisas Espaciais
KB	Kilo Byte = 1.024 bytes
LAC	Laboratório Associado de Computação e Matemática Aplicada do INPE
LHS	Left Hand Side
μ K	Micro Kelvin. (1,0 E -6 Kelvin)
μ s	Microsegundos. (1,0 E -6 segundos)
m	Metro
MAP	Microwave Anisotropy Probe
MB	Mega Byte = 1.024 KB = 1.048.576 bytes

MHz	Mega Hertz (1,0 E 6 Hertz)
ms	Milisegundos (1,0 E -3 segundos)
MIMD	Multiple-Instruction Multiple-Data
MISD	Multiple-Instruction Single-Data
MPI	Message Passing Interface
MPP	Massive Parallel Processors
NASA	National Aeronautics and Space Administration
NCC- NUMA	Non-Cache Coherent NUMA
n_d	Número de observações
NORMA	NO Remote Memory Access
NOW	Network of Workstations
n_p	Número de pixels
NUMA	Non-uniform Memory Access
PC	Personal Computer
PVM	Parallel Virtual Machine
RAM	Random Access Memory
RCFM	Radiação Cósmica de Fundo em Microondas
RHS	Right Hand Side
RISC	Reduced Instruction Set Computers
SC-NUMA	Software Coherent NUMA
SIMD	Single-Instruction Multiple-Data
SISD	Single-Instruction Single-Data
SLB	Setor de Lançamento de Balões
SMP	Symmetric Multi Processors
SPMD	Single-Program Multiple-Data
STD	Série Temporal de Dados
TB	Tera Byte = 1.024 GB = 1.099.511.627.776 bytes
UMA	Uniform Memory Access
VLIW	Very Long Instruction Word

CAPÍTULO 1

INTRODUÇÃO

A Radiação Cósmica de Fundo em Microondas (RCFM) é um dos principais objetivos de pesquisa em Cosmologia, sendo uma das evidências observacionais que sustentam o modelo do *Big Bang* como modelo cosmológico padrão. A energia liberada após o evento teria se arrefecido com a expansão do universo e a observação dessa energia residual é uma ferramenta importante para o entendimento da origem, estruturas e evolução do universo.

Denomina-se radiação cósmica de fundo a qualquer contribuição de intensidade ou brilho do céu que não possa ser associada a fontes individuais, galácticas ou extragalácticas, conhecidas. Sua origem pode ser devida a objetos muito distantes, acreditando-se ser o Universo Jovem a fonte dessa radiação. A faixa de microondas é a região do espectro eletromagnético onde a radiação cósmica de fundo é mais intensa.

Para o estudo da RCFM, instrumentos no solo e a bordo de balões estratosféricos ou satélites vêm coletando enormes conjuntos de dados, que requerem sistemas de alta capacidade de processamento para sua redução, visualização e análise; sistemas que sejam rápidos e que suportem a variedade de pacotes de software de análise disponíveis para essas aplicações.

A produção de mapas é uma etapa intermediária do processo de análise de dados da RCFM, que consiste na transformação de enormes séries temporais de dados (10^6 a 10^{10} pontos) fornecidas pelos instrumentos de observação em alguns parâmetros de interesse cosmológico, que permitem estabelecer, com precisão cada vez maior, as condições iniciais do universo.

Com o contínuo aprimoramento da instrumentação para aquisição de dados e a necessidade de se produzir mapas com melhor resolução, o estudo da RCFM representa

um sério desafio computacional. O tratamento e análise de dados requerem equipamentos de ponta e minimização do tempo de processamento, estimulando o aperfeiçoamento dos métodos e algoritmos existentes.

A técnica clássica de produção de mapas utilizada neste trabalho, descrita por Wright [1], apresenta duas características importantes: o tempo de processamento é proporcional ao número de dados disponíveis e a memória principal exigida é proporcional ao número de pixels com o qual se deseja confeccionar o mapa. Com os instrumentos embarcados em satélites, como os satélites *Microwave Anisotropy Probe* (MAP) da NASA lançado em junho de 2001 e PLANCK da ESA previsto para 2007, esperam-se obter séries temporais da ordem de 10^{10} pontos e mapas com 10^6 pixels, que requerem não somente maior capacidade computacional, mas também uma quebra de paradigma nas técnicas atuais de análise de dados. Técnicas como processamento paralelo, redes neurais e algoritmos genéticos estão entre as novas ferramentas exploradas pela comunidade científica para a solução destes problemas. O objetivo principal da exploração de novas técnicas é conseguir maior precisão e velocidade na visualização, análise e modelagem da imensa quantidade de dados gerados por experimentos a bordo de satélites e testar novas técnicas ainda não usuais na Astronomia, mas já usuais e eficientes em outros ramos da Ciência. Em ambos os casos, as exigências computacionais implicam em sistemas com grande capacidade de memória e velocidade para tratamento de uma grande quantidade de dados.

A incapacidade dos algoritmos utilizados atualmente em lidar com as técnicas de análise de dados em escalas de tempo adequadas motivou o estudo de implementações alternativas e a paralelização dos procedimentos. Este trabalho descreve a investigação de uma solução para este problema através da otimização e reestruturação de um código seqüencial utilizado na análise de dados do experimento *HEMT Advanced Cosmic Microwave Explorer* (HACME) e o emprego de programação paralela visando à minimização do tempo de processamento. O trabalho iniciou-se com a otimização de um programa codificado originalmente em Fortran, portando-o para Fortran 90, sendo o código instrumentado para medidas de tempo de execução e posterior análise.

Investigou-se a viabilidade do uso de uma implementação paralela baseada em paralelismo de dados, utilizando *High Performance Fortran* (HPF). Em trechos do programa onde o paralelismo de dados não foi eficiente, como nas rotinas de convolução de matrizes usando *Fast Fourier Transform* (FFT), foi investigado o uso de uma implementação mista combinando paralelismo de tarefas e dados. O paralelismo de tarefas foi implementado com a utilização de funções *Message Passing Interface* (MPI) em um ambiente baseado em paralelismo de dados usando HPF, com o objetivo de se minimizar o tempo de comunicação entre os processadores envolvidos.

A plataforma utilizada para o estudo, otimização e implementação paralela da produção de mapas foi uma máquina paralela de memória distribuída do INPE-DAS (máquina *cluster*), composta por 16 nós baseados na arquitetura Intel IA-32, sistema operacional Linux, sendo os nós interligados por um comutador e uma rede padrão *Fast Ethernet*. Esta é uma configuração muito comum, denominada *beowulf cluster*.

Este trabalho foi organizado em 6 capítulos. O capítulo 2 descreve alguns conceitos sobre o método para produção de mapas da RCFM. O capítulo 3 apresenta conceitos sobre programação paralela, uma breve descrição das arquiteturas, paradigmas e ambientes de programação paralela e algumas características das linguagens de programação Fortran 90, HPF e da biblioteca de funções MPI. O capítulo 4 apresenta conceitos sobre algumas técnicas de otimização, consideradas clássicas, tanto para programas seqüenciais como paralelos. O capítulo 5 apresenta os resultados obtidos durante a otimização da implementação seqüencial, a investigação de uma implementação paralela baseada em paralelismo de dados e os resultados de uma implementação mista combinando paralelismo de tarefas e dados. O capítulo 6 apresenta algumas considerações, conclusões e possíveis caminhos de estudos futuros para a solução dos problemas encontrados na produção de mapas da RCFM.

CAPÍTULO 2

PRODUÇÃO DE MAPAS DA RCFM

A Cosmologia atual busca o entendimento do processo de formação de galáxias, aglomerados de galáxias e demais estruturas do Universo. Seu provável causador seria a instabilidade gravitacional causada por flutuações na densidade de matéria no Universo primordial, que teriam causado flutuações de temperatura na distribuição angular da RCFM, observadas hoje por diversos experimentos a bordo de balões, satélites e no solo. A existência da RCFM é uma das evidências observacionais que sustentam o modelo do *Big Bang* como modelo cosmológico padrão [2][3].

A produção de mapas é um estágio essencial do processo de análise de dados da RCFM, que consiste na transformação de enormes séries temporais de dados (STD) fornecidas pelos instrumentos de observação em alguns parâmetros de interesse cosmológico. A partir de uma STD se obtém uma estimativa do mapa do céu numa certa frequência. Após a obtenção de mapas em diversas frequências calcula-se o espectro de potência da RCFM. Conhecendo-se o espectro de potência estimam-se alguns parâmetros cosmológicos, como o parâmetro de densidade Ω , o parâmetro de densidade bariônica Ω_b , o parâmetro de Hubble h e a constante cosmológica Λ , parâmetros estes que permitem confrontar os modelos cosmológicos com resultados experimentais.

A Figura 2.1 ilustra o processo de redução de dados da RCFM.

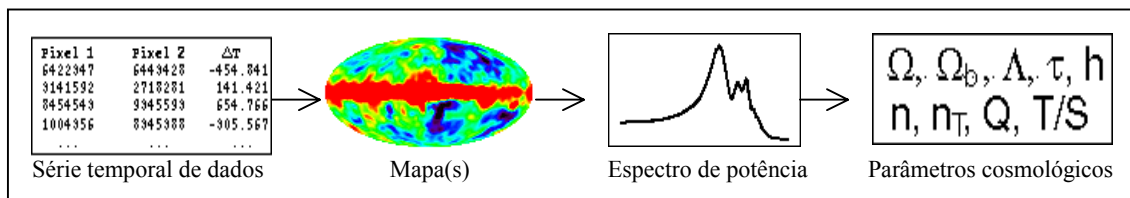


FIGURA 2.1 – Esquema ilustrativo do processo de redução de dados da RCFM.

Uma STD [4] é uma série cronológica das observações realizadas no decorrer de um dado experimento, consistindo em uma tabela com as indicações das coordenadas das regiões apontadas pelo instrumento e os sinais medidos em cada região, que podem ser associados às diferenças de temperatura ou às próprias temperaturas de cada região.

Um mapa da RCFM consiste na representação gráfica da distribuição de temperatura do céu obtida após a manipulação da respectiva série. Os mapas de RCFM representam enormes quantidades de dados sem perda substancial de informação e ocupam um lugar de destaque na metodologia de análise de dados em Cosmologia experimental. Para a obtenção de uma boa relação sinal/ruído, cada região do céu é observada inúmeras vezes, causando a coleta deste grande conjunto de dados.

A implementação em estudo foi desenvolvida para o experimento *HEMT Advanced Cosmic Microwave Explorer* (HACME) [5][6], cujo objetivo foi medir flutuações de temperatura da RCFM nas regiões de γ Ursae Minoris e α Leonis. Paralelamente, o HACME serviu de protótipo para testar a instrumentação que fez parte do experimento *Background Emission Anisotropy Scanning Telescope* (BEAST) [7], projeto desenvolvido em parceria pelo Departamento de Física da Universidade da Califórnia em Santa Bárbara (UCSB), o *Jet Propulsion Laboratory* (JPL) do Instituto de Tecnologia da Califórnia, o Instituto Nacional de Física Nuclear (INFN) e o Departamento de Física da Universidade de Roma, o Instituto de Astrofísica Espacial e Física Cósmica do Conselho Nacional de Pesquisas (IASF-CNR) em Bolonha, a Universidade de Milão, os Departamentos de Astronomia e Física da Universidade de Illinois, o Departamento de Física e Química da Universidade Federal de Itajubá (UNIFEI) e a Divisão de Astrofísica do INPE.

O BEAST consiste em um telescópio gregoriano não-axial [7][8] projetado para medir flutuações da RCFM, produzindo mapas com uma resolução angular de $0,33^\circ$ e $0,5^\circ$ e com sensibilidade de $10 \mu\text{K}$ a $20 \mu\text{K}$ por pixel em 40 GHz e em 30 GHz, respectivamente. A configuração óptica consiste em um espelho primário parabólico (semi-eixo maior igual a 2,2 m) e um sub-refletor elipsoidal (semi-eixo maior igual a

0,8 m). Um espelho plano giratório faz o feixe descrever uma trajetória elíptica no céu, que combinada com o movimento azimutal da gôndola, possibilitou a obtenção de mapas bidimensionais da RCFM. Após a realização de dois vôos em balões no ano de 2000, o telescópio foi instalado em uma base de pesquisas da Universidade da Califórnia (WMRS - *White Mountain Research Station*), coletando dados de julho a dezembro de 2001 e em alguns períodos de 2002.

A Figura 2.2 apresenta uma ilustração de um experimento para medidas da RCFM em vôos de balão, constituído de um telescópio, sistema de aquisição de dados e eletrônica associada.

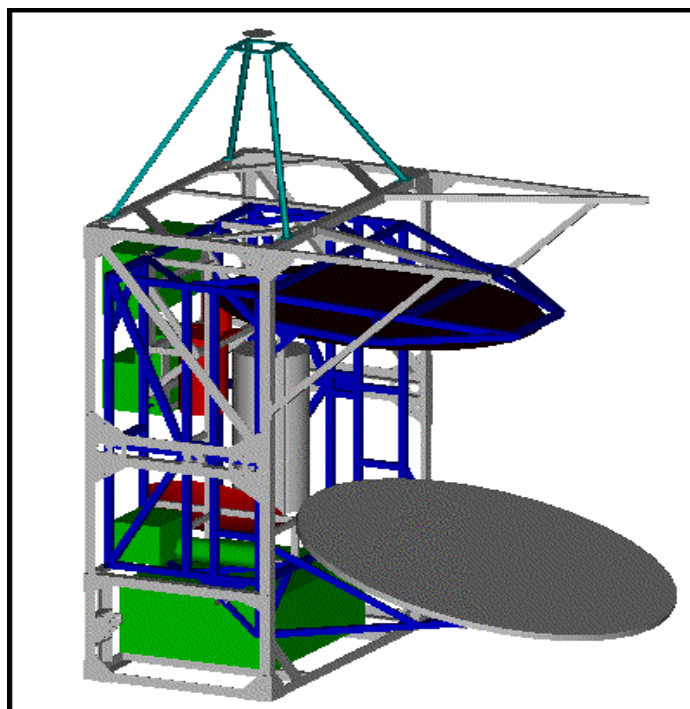


FIGURA 2.2 – Plataforma HACME.

A Figura 2.3 apresenta uma comparação entre mapas produzidos pelo experimento HACME e um mapa produzido pelo satélite *Cosmic Background Explorer* (COBE). O mapa em torno da estrela γ Ursae Minoris possui 3082 pixels, o mapa em torno da estrela α Leonis 2050 pixels e o mapa produzido pelo satélite COBE 6144 pixels.

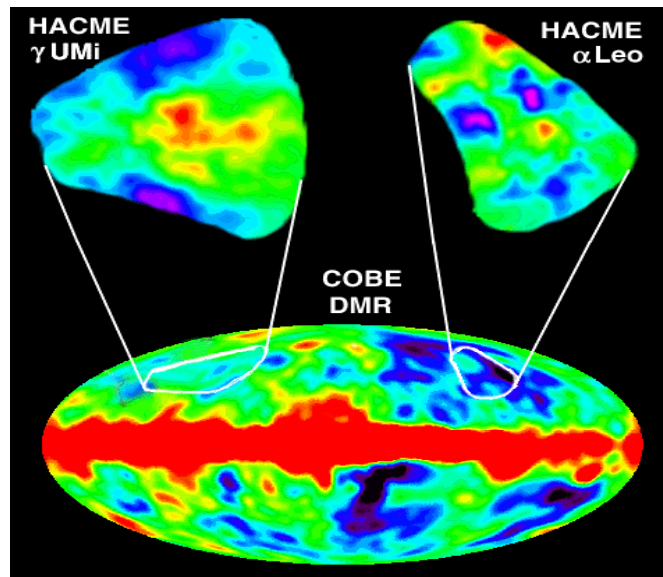


FIGURA 2.3 – Mapas do experimento HACME e satélite COBE.
 FONTE: [5].

A estratégia de observação do experimento é representada na Figura 2.4. Há três movimentos considerados nesta estratégia: o movimento do espelho plano, o movimento da gôndola que executa uma varredura em azimute e a rotação do céu. A composição destes três movimentos permite observar uma certa região do céu. Durante um vôo em balão, o telescópio é apontado para algumas regiões de interesse, que são monitoradas durante horas, gerando um grande volume de dados, que demandam sistemas de alta capacidade de processamento para sua redução e análise.

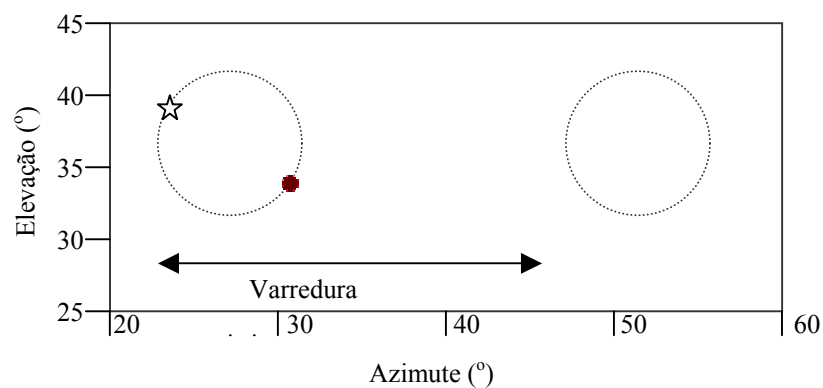


FIGURA 2.4 – Estratégia de apontamento do telescópio.

2.1 – Formalismo Tradicional para Produção de Mapas da RCFM

Na redução e análise dos dados da RCFM, métodos lineares podem ser utilizados para a produção dos mapas. Um método linear de produção de mapas pode ser expresso como $m = W d$, onde m é uma estimativa do mapa, d uma série temporal de dados e W denota uma matriz $n_p \times n_d$ que caracteriza o método, sendo n_p o número de pixels do mapa e n_d o número de pontos da série temporal. Tegmark [4] descreve 8 métodos lineares para produção de mapas da RCFM, dentre eles o método utilizado na redução de dados deste experimento.

O método para produção de mapas desenvolvido pela equipe do satélite COBE e adotado para vários outros experimentos [9][10][11], e neste trabalho denominado método COBE, pode ser descrito através dos seguintes passos:

Adota-se um sistema de coordenadas astronômicas (θ, ψ) para determinação de pontos sobre a esfera celeste. Divide-se a região do céu que se pretende mapear em sub-regiões, aproximadamente iguais e mutuamente excludentes, e a cada uma delas associa-se um pixel. Conhecendo-se a posição em que o telescópio estava apontado em cada observação pode-se construir uma matriz P de dimensões $n_d \times n_p$ que representa a estratégia de apontamento do experimento. Esta matriz de apontamento indica as coordenadas do pixel p observado em cada instante t , tal que $P = 1$ se $(\theta_t, \psi_t) \in p$ e $P = 0$, caso contrário. Assim, os elementos da matriz P são todos iguais a 0, exceto um único 1 em cada linha da matriz. Na seção 2.2, é apresentado um modelo do céu com 14 pixels para ilustração do método.

As n_d observações do céu são agrupadas em uma série temporal d , sendo d_t a temperatura medida na observação t . Um mapa de n_p pixels é obtido desta série, estando relacionados linearmente através da equação $d = P s + n$, onde P é a matriz de apontamento, n um ruído aleatório (normalmente *white noise* - ruído branco gaussiano) e s o sinal que chega ao detector ou o sinal da RCFM.

Tegmark [4] descreve que, no método COBE, a matriz que caracteriza o método é definida pela equação:

$$W = (P^T N^{-1} P)^{-1} P^T N^{-1}. \quad (2.1)$$

Então, uma estimativa do mapa é obtida através da equação:

$$m = (P^T N^{-1} P)^{-1} P^T N^{-1} d, \quad (2.2)$$

onde $N \equiv \langle n n^T \rangle$ é a matriz de covariância de ruído.

Substituindo o vetor de dados pela sua equação, tem-se que:

$$m = (P^T N^{-1} P)^{-1} P^T N^{-1} (P s + n) = s + v. \quad (2.3)$$

Esta equação mostra que a estimativa do mapa é igual ao mapa verdadeiro mais o ruído, dado por:

$$v = (P^T N^{-1} P)^{-1} P^T N^{-1} n, \quad (2.4)$$

com correlações dadas por:

$$Y \equiv \langle v v^T \rangle = (P^T N^{-1} P)^{-1}. \quad (2.5)$$

O processo para obtenção dos mapas pode ser convenientemente realizado em 3 etapas, que envolvem a solução das seguintes equações:

$$Y^{-1} = P^T N^{-1} P, \quad (2.6)$$

$$z = P^T N^{-1} d, \quad (2.7)$$

$$m = (Y^{-1})^{-1} z. \quad (2.8)$$

A Tabela 2.1 apresenta uma estimativa dos recursos computacionais requeridos para solução de cada passo deste algoritmo, como descrito por Borril [12].

TABELA 2.1 – Requisitos Computacionais para o Algoritmo de Produção de Mapas

Etapa	Disco rígido	Memória principal	Operações em ponto flutuante
$\Upsilon^{-1} = P^T N^{-1} P$	$4 n_d^2$	$16 n_d$	$2 n_p n_d^2$
$z = P^T N^{-1} d$	$4 n_d^2$	$16 n_d$	$2 n_d^2$
$m = (\Upsilon^{-1})^{-1} z$	$4 n_p^2$	$8 n_p^2$	$8 n_p^3 / 3$

A análise da enorme quantidade de informação fornecida por uma série temporal de dados de experimentos recentes para estudo da RCFM como os experimentos HACME, *Millimeter Anisotropy eXperiment IMaging Array* (MAXIMA) [13], *Balloon Observations Of Millimetric Extragalactic Radiation and Geophysics* (BOOMERanG) [12], *Microwave Anisotropy Probe* (MAP) [10] ou BEAST [7] tornou-se um grande desafio computacional. Por exemplo, para experimentos tais como BOOMERanG North América [12], com número de medidas da série temporal $n_d = 2 \cdot 10^6$ e mapas com número de pixels $n_p = 3 \cdot 10^4$, o algoritmo requer 16 TB para armazenamento de dados com 4 bytes de precisão, 7 GB de memória principal para cálculos com 8 bytes de precisão e envolve $2,4 \cdot 10^{17}$ operações em ponto flutuante. Para um processador de 600 MHz com desempenho nominal, o tempo necessário para obtenção do mapa seria aproximadamente de 12 anos. Devido a algumas características dos dados, como utilização de matrizes esparsas e simétricas, os cálculos podem ser simplificados, necessitando-se de 3,6 GB para armazenamento de dados e $7 \cdot 10^{13}$ operações em ponto flutuante, o que exigiria 32 horas de processamento.

Devido aos tempos excessivos para obtenção de mapas estimados para os próximos experimentos, é importante se tentar otimizar as operações com matrizes, típicas desse método, e buscar implementações mais eficientes, onde o emprego de programação paralela se apresenta como uma boa alternativa para a solução destes problemas.

2.2 – Modelo do Céu

Um modelo de céu [9] pode ser usado para ilustrar as operações realizadas durante o processo de produção de mapas. A Figura 2.5 mostra um modelo do céu com 14 pixels, que são observados em 3 varreduras circulares, que se cruzam somente nos pólos norte (N) e sul (S), totalizando 18 pontos de dados. Neste modelo, para número de pixels cruzando os pólos $N = 6$, ou seja, 2 pixels por varredura, o número de pontos de dados é dado por $n_d = N^2 / 2 = 18$ e o número de pixels é dado por $n_p = (N / 2) (N - 2) + 2 = 14$.

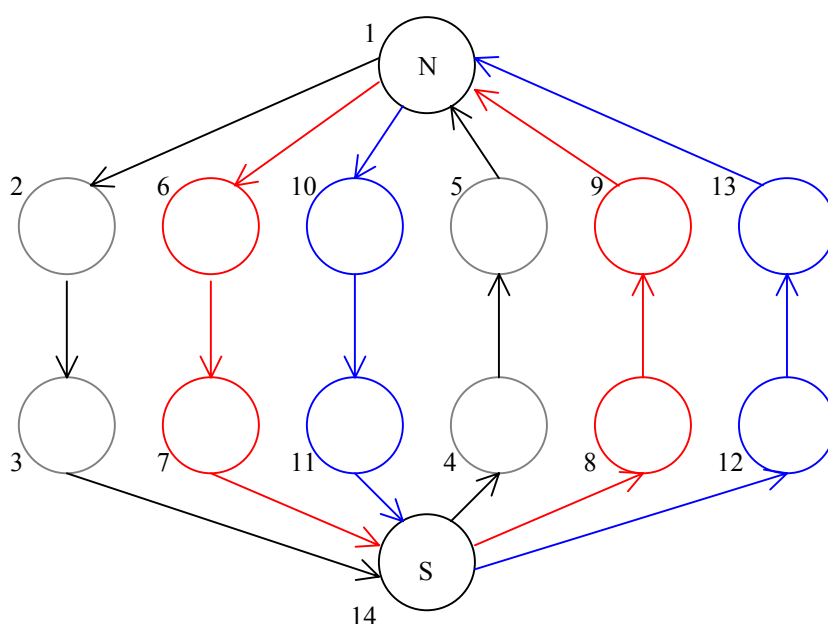


FIGURA 2.5 – Modelo do céu.

Cada pixel corresponde a uma dada posição do céu, que pode ser representado através de uma matriz P de $n_d \times n_p$ elementos, com uma linha para cada observação e uma coluna para cada pixel do mapa. Esta matriz de apontamento P tem todos seus elementos iguais a zero exceto um único elemento igual a um em cada linha e na coluna do pixel observado, conforme Figura 2.6. Os pólos são os pixels 1 e 14, a varredura 1 contém os pixels 1, 2, 3, 14, 4, 5, a varredura 2 contém os pixels 1, 6, 7, 14, 8, 9, enquanto a varredura 3 contém os pixels 1, 10, 11, 14, 12, 13.

Para uma série temporal de dados d , uma estimativa do mapa é obtida através da equação $m = (P^T N^{-1} P)^{-1} P^T N^{-1} d$, onde $N \equiv \langle nn^T \rangle$. Considerando $A = P^T N^{-1} P$ e $B = P^T N^{-1} d$, tem-se que o mapa é produzido através da solução da equação $m = A^{-1} B$.

Para se obter uma estimativa do mapa, um passo intermediário é se obter a matriz de correlação de ruído, representada pela matriz A ($n_p \times n_p$), e mostrada na Figura 2.8.

$$A = \frac{1}{25} \begin{pmatrix} 90 & -6 & -6 & -6 & -6 & -6 & -6 & -6 & -6 & -6 & -6 & -6 & -6 & -18 \\ -6 & 30 & -6 & -6 & -6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -6 \\ -6 & -6 & 30 & -6 & -6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -6 \\ -6 & -6 & -6 & 30 & -6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -6 \\ -6 & -6 & -6 & -6 & 30 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -6 \\ -6 & 0 & 0 & 0 & 0 & 30 & -6 & -6 & -6 & 0 & 0 & 0 & 0 & -6 \\ -6 & 0 & 0 & 0 & 0 & -6 & 30 & -6 & -6 & 0 & 0 & 0 & 0 & -6 \\ -6 & 0 & 0 & 0 & 0 & -6 & -6 & 30 & -6 & 0 & 0 & 0 & 0 & -6 \\ -6 & 0 & 0 & 0 & 0 & -6 & -6 & -6 & 30 & 0 & 0 & 0 & 0 & -6 \\ -6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 30 & -6 & -6 & -6 & -6 \\ -6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -6 & 30 & -6 & -6 & -6 \\ -6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -6 & -6 & 30 & -6 & -6 \\ -6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -6 & -6 & -6 & 30 & -6 \\ -18 & -6 & -6 & -6 & -6 & -6 & -6 & -6 & -6 & -6 & -6 & -6 & -6 & 90 \end{pmatrix}$$

FIGURA 2.8 – Matriz de correlação de ruído.

A estimativa do mapa é obtida solucionando-se a equação $m = A^{-1} B$, onde A^{-1} é a matriz de covariância de ruído inversa [9], representada na Figura 2.9.

$$A^{-1} = \begin{pmatrix} 0.24 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & 0.01 \\ -0.02 & 0.87 & 0.18 & 0.18 & 0.18 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.02 \\ -0.02 & 0.18 & 0.87 & 0.18 & 0.18 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.02 \\ -0.02 & 0.18 & 0.18 & 0.87 & 0.18 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.02 \\ -0.02 & 0.18 & 0.18 & 0.18 & 0.87 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.02 \\ -0.02 & -0.17 & -0.17 & -0.17 & -0.17 & 0.87 & 0.18 & 0.18 & 0.18 & -0.17 & -0.17 & -0.17 & -0.17 & -0.02 \\ -0.02 & -0.17 & -0.17 & -0.17 & -0.17 & 0.18 & 0.87 & 0.18 & 0.18 & -0.17 & -0.17 & -0.17 & -0.17 & -0.02 \\ -0.02 & -0.17 & -0.17 & -0.17 & -0.17 & 0.18 & 0.18 & 0.87 & 0.18 & -0.17 & -0.17 & -0.17 & -0.17 & -0.02 \\ -0.02 & -0.17 & -0.17 & -0.17 & -0.17 & 0.18 & 0.18 & 0.18 & 0.87 & -0.17 & -0.17 & -0.17 & -0.17 & -0.02 \\ -0.02 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & 0.87 & 0.18 & 0.18 & 0.18 & -0.02 \\ -0.02 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & 0.18 & 0.87 & 0.18 & 0.18 & -0.02 \\ -0.02 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & 0.18 & 0.18 & 0.87 & 0.18 & -0.02 \\ -0.02 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & -0.17 & 0.18 & 0.18 & 0.18 & 0.87 & -0.02 \\ 0.01 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & -0.02 & 0.24 \end{pmatrix}$$

FIGURA 2.9 – Matriz de covariância de ruído inversa.

Para grandes conjuntos de dados e mapas, a construção de A e A^{-1} é impraticável, contudo o cálculo de $A=P^T N^{-1} P$ pode ser feito em escala de tempo razoável dividindo-se o cálculo em três etapas:

- cálculo da matriz P que envolve n_d operações,
- cálculo da convolução de matrizes (nm^T) , feito em N blocos envolvendo $n_d \ln N$ operações quando se usa FFT,
- cálculo de P^T que envolve n_d operações.

As etapas deste método para obtenção de mapas serão analisadas com maiores detalhes adiante, no capítulo referente à otimização e investigação de uma implementação paralela baseada em paralelismo de dados.

CAPÍTULO 3

PROGRAMAÇÃO PARALELA

A demanda crescente por maior desempenho nas aplicações científicas, tal como no estudo da RCFM, tem incentivado o uso da computação paralela, na qual o processamento numérico é dividido por um certo número de processadores trabalhando em paralelo e trocando informações e dados entre si. Até bem recentemente, esta possibilidade esteve restrita a um certo número de máquinas concebidas explicitamente para processamento paralelo e de operação relativamente complexa. No entanto, com o advento de novas linguagens com suporte ao paralelismo e de bibliotecas de comunicação e troca de dados entre processadores, soluções mais acessíveis e de operação consideravelmente simplificada tornaram-se possíveis, através da utilização de arquiteturas paralelas de memória distribuída.

Uma arquitetura paralela de memória distribuída traz inúmeras vantagens na resolução de problemas onde há necessidade de grande capacidade computacional, exigência de muita memória e grande área para armazenamento. Dessa forma, por exemplo, um *cluster* de computadores interligados como nós de uma rede pode fornecer, além do menor custo, potencial computacional, capacidade de memória local e área de armazenamento adequada para aplicações utilizando programação paralela.

Neste trabalho, a utilização de uma plataforma paralela de memória distribuída composta de 16 nós baseados na arquitetura Intel IA-32 (máquina *cluster*), possibilitou o estudo do problema computacional e a investigação de uma solução empregando programação paralela em uma das etapas do processo de análise de dados coletados de experimentos desenvolvidos para estudo da RCFM.

Neste capítulo, serão descritos alguns conceitos sobre arquiteturas de alto desempenho, arquiteturas de máquinas paralelas, paradigmas e ambientes de programação paralela, algumas características das linguagens Fortran 90 e HPF, bem como alguns conceitos

sobre a biblioteca de troca de mensagens MPI, utilizadas na pesquisa de uma solução otimizada para produção de mapas da RCFM.

3.1 – Arquiteturas de Alto Desempenho

A busca por alto desempenho sempre foi uma preocupação tanto de fabricantes como de usuários, utilizando-se desde o paralelismo interno existente nos processadores atuais da arquitetura *Reduced Instruction Set Computers* (RISC), como o uso misto de arquiteturas *Complex Instruction Set Computers* (CISC) com micro-instruções RISC (caso da Intel) e ainda, o uso de arquiteturas paralelas; o que mostra a complexidade de formas existentes para se atender a necessidade atual de alto desempenho.

Em arquiteturas CISC, a tecnologia era pouco desenvolvida e se buscava um menor esforço de programação. Sendo assim, um conjunto de instruções maior significava maior desempenho.

Em arquiteturas RISC, observa-se a busca por alto desempenho em técnicas como o uso de *pipelining*, que é utilizado hoje em arquiteturas CISC como os processadores Pentium. O *pipelining* é uma técnica em que o hardware do computador executa mais de uma instrução simultaneamente sem a necessidade de se aguardar o término de uma instrução para executar a próxima, operação semelhante a uma linha de produção. A complexidade removida do conjunto de instruções é transferida para o compilador, influenciando o desempenho da máquina.

Em arquiteturas RISC, da mesma forma que nas máquinas CISC, uma instrução passa por quatro estágios: busca, decodificação, execução e escrita, sendo a passagem pelos estágios executada em paralelo. Ao completar um estágio, o resultado é encaminhado para o estágio seguinte, sendo iniciado o estágio atual com outra instrução. Cada instrução consome um ciclo de *clock* permitindo ao processador aceitar uma nova instrução a cada ciclo.

O objetivo inicial dos processadores RISC era uma instrução por ciclo. Atualmente duas ou mais instruções são executadas simultaneamente em *pipelines* separados, e máquinas que permitem executar múltiplas operações em vários *pipelines* simultaneamente são denominadas máquinas superescalares.

O desempenho de um processador RISC depende basicamente da forma como o código é escrito, permitindo, assim, explicitar suas características superescalares.

Quando os estágios do *pipeline* são divididos em sub-estágios capazes de executar operações ainda mais simples, a arquitetura é denominada *superpipelined*.

Quando o compilador funde duas ou mais instruções RISC em uma única palavra, sendo que todas as operações na palavra são executadas em paralelo, a arquitetura é denominada *Very Long Instruction Word* (VLIW). Este tipo de arquitetura requer compiladores sofisticados, pois a análise de dependência de dados, fluxo de instruções, execução fora de ordem e desvios é muito complexa.

Outra classe de arquiteturas compreende as denominadas máquinas vetoriais, que são caracterizadas por unidades funcionais que implementam operações sobre vetores. As máquinas vetoriais podem ser um monoprocessador, um multiprocessador ou um multicomputador, que utilizam processadores contendo registros e *pipelines* vetoriais e escalares, ou seja, processadores vetoriais. Tipicamente, não há *caches* e o acesso à memória é feito de forma rápida e uniforme.

3.2 – Arquiteturas de Máquinas Paralelas

Na busca por alto desempenho de uma aplicação científica, uma opção é a utilização de arquiteturas paralelas, que além do baixo custo comparado ao de um supercomputador seqüencial, permitem solucionar no menor tempo possível um determinado problema computacional. O problema é dividido entre um conjunto de processadores, que podem

executar diferentes tarefas de forma simultânea ou podem operar com subconjuntos dos dados originais, reduzindo o tempo de processamento.

Um conceito importante para o entendimento e utilização de arquiteturas paralelas é o conceito de granularidade, descrito a seguir.

Uma aplicação paralela consiste na distribuição de tarefas ou dados entre processadores distintos, de tal forma que sua execução seja mais rápida que a execução em um único processador. Para isto, é necessária uma comunicação entre os processadores para se garantir o progresso consistente da aplicação. A intensidade da comunicação entre as tarefas de uma aplicação paralela estabelece a granularidade desta aplicação. Aplicações que demandam intensa comunicação são ditas de fina granularidade. Aplicações cujas tarefas requerem pouca comunicação são ditas de grossa granularidade.

Uma aplicação paralela pode, também, ser classificada em alguns níveis de granularidade de acordo com o grau de paralelismo, ou seja, granularidade fina implica em paralelismo ao nível de instrução, granularidade fina/média implica em paralelismo ao nível de malhas, granularidade média implica em paralelismo ao nível de procedimentos e granularidade grossa implica em paralelismo ao nível de programas.

A granularidade estabelece requisitos para a arquitetura paralela na qual uma aplicação é executada. Dependendo da aplicação, uma certa arquitetura paralela pode ser mais, ou menos, conveniente. As aplicações podem demandar o uso de plataformas dedicadas, tais como supercomputadores massivamente paralelos, ou o uso de máquinas paralelas constituídas de nós fisicamente distribuídos e conectados por uma Internet de alta velocidade - caso da metacomputação. Cada nó pode ser uma máquina seqüencial ou paralela dos tipos expostos a seguir.

Na prática, as arquiteturas paralelas utilizadas atualmente são as do tipo *multiple-instruction, multiple-data* (MIMD) [14][15], que são arquiteturas caracterizadas pela execução simultânea de múltiplos fluxos de instruções. Essa capacidade deve-se ao fato

de que são construídas a partir de vários processadores operando de forma cooperativa ou concorrente, na execução de um ou vários aplicativos. Essa definição deixa margem para que várias topologias de máquinas paralelas e de redes de computadores sejam enquadradas como MIMD. A diferenciação entre as diversas topologias MIMD é feita pelo tipo de organização da memória principal, memória *cache* e rede de interconexão.

Basicamente, as arquiteturas MIMD podem ser divididas em duas categorias: MIMD com memória compartilhada e MIMD com memória distribuída [14].

As arquiteturas MIMD de memória compartilhada possuem mais de um processador em uma mesma máquina, todos eles compartilhando os recursos de memória principal e secundária existentes, sob controle do sistema operacional. Esta arquitetura é bem transparente ao usuário, ficando a cargo do sistema operacional a maior parte da complexidade. Um exemplo deste tipo de arquitetura é a máquina *polaris* do INPE-LAC (composta por 4 processadores, sistema operacional Linux) que foi utilizada nos estudos iniciais para otimização da produção de mapas da RCFM. A Figura 3.1 exemplifica este tipo de arquitetura.

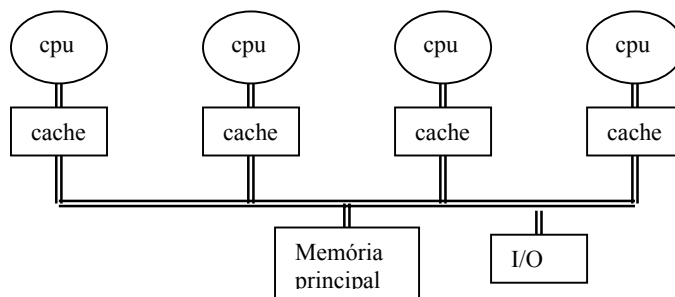


FIGURA 3.1 – Arquitetura paralela de memória compartilhada.

As arquiteturas do tipo MIMD com memória compartilhada são também denominadas *multiprocessadores* e podem, ainda, ser classificadas quanto ao espaço físico de memória em duas categorias: máquinas *Uniform Memory Access* (UMA) e *Non-uniform Memory Access* (NUMA).

Em uma arquitetura do tipo UMA, o conjunto de processadores vêem a mesma memória como uma área única, não dividida fisicamente. O tempo para acesso aos dados na memória é o mesmo para todos os processadores. A forma de interconexão mais comum é o barramento e a memória geralmente é implementada com um único módulo. Neste tipo de arquitetura, cada processador acessa a memória via *cache*, que são utilizadas para diminuir o tráfego no barramento. Como várias cópias de um mesmo dado podem ser manipuladas simultaneamente nas *caches* de vários processadores, é necessário que se garanta que os processadores sempre acessem a cópia mais recente. Essa garantia é chamada de coerência de *cache*, e máquinas UMA geralmente lidam com este problema diretamente em hardware. Um dos protocolos de coerência de *cache* mais populares é chamado de *snooping*, ou “bisbilhoteiro”. Neste caso, quando um dado compartilhado por várias *caches* é alterado por algum processador, todas as demais cópias são invalidadas e então atualizadas. Um caso particular desta arquitetura é o das máquinas denominadas *Symmetric Multi Processors* (SMP), nas quais os processadores executam cópias idênticas do sistema operacional.

Em uma arquitetura do tipo NUMA, a cada processador ou grupo de processadores é acrescentada uma memória local privada. Como a memória é dividida fisicamente, o tempo de acesso à memória não é uniforme. Cada processador tem acesso direto a toda a memória localizada em qualquer nó do sistema, acessando uma memória mais lenta através de uma interconexão desses nós. As máquinas NUMA também estão sujeitas aos problemas de coerência de *cache* e, conforme a solução implementada, existem algumas variações deste tipo de arquitetura [16]:

- *Non-Cache Coherent NUMA* (NCC-NUMA): nesse tipo de máquina não existe garantia de coerência de *cache* ou, simplesmente, não existe *cache*;
- *Cache Coherent NUMA* (CC-NUMA): nesse tipo de máquina a coerência de *cache* é garantida pelo hardware;

- *Software Coherent NUMA (SC-NUMA)*: nesse tipo de máquina a coerência de *cache* é garantida por software e essa implementação recebe o nome de *Distributed Shared Memory (DSM)*. O espaço de endereçamento único é conseguido através de uma abstração implementada em software que pode tomar duas formas. Na primeira, os mecanismos de gerência de memória do sistema operacional são modificados para suprir as faltas de página ou segmento a partir da rede de interconexão. Quando um processador precisa de um dado em um espaço de endereçamento alheio, o sistema operacional encarrega-se de encontrar e disponibilizar o mesmo para o processador. Na segunda forma, alterações são feitas nos compiladores e bibliotecas de funções para que reflita nas aplicações o modelo de memória utilizado. As aplicações são então modificadas para a incorporação de primitivas de coerência, para compartilhamento de dados e sincronização.

Outra variação das arquiteturas de memória compartilhada é a das máquinas *Cache-Only Memory Access (COMA)* [16]. Essas máquinas são multiprocessadores baseados em memórias *cache* de alta capacidade, em que a coerência é conseguida em hardware com a atualização simultânea em múltiplos nós dos dados alterados. Em alguns casos, os processadores contem somente memória *cache*. Esse tipo de arquitetura é bastante complexo e faz com que essas máquinas tenham um custo bastante elevado.

As arquiteturas MIMD de memória distribuída estão exemplificadas na Figura 3.2. Nestas arquiteturas, há pouco ou nenhum compartilhamento de recursos entre os processadores. Normalmente, cada nó é um computador independente, com memória principal e secundária próprias, sendo interconectados por uma rede. Nestes sistemas, o controle do paralelismo é realizado pela aplicação, que deve coordenar as tarefas e a coerência entre os diversos nós. Arquiteturas de memória distribuída apresentam vantagens comparadas às arquiteturas de memória compartilhada, pelo seu menor custo e por serem arquiteturas escaláveis, podendo seu conjunto de computadores ser facilmente expandido.

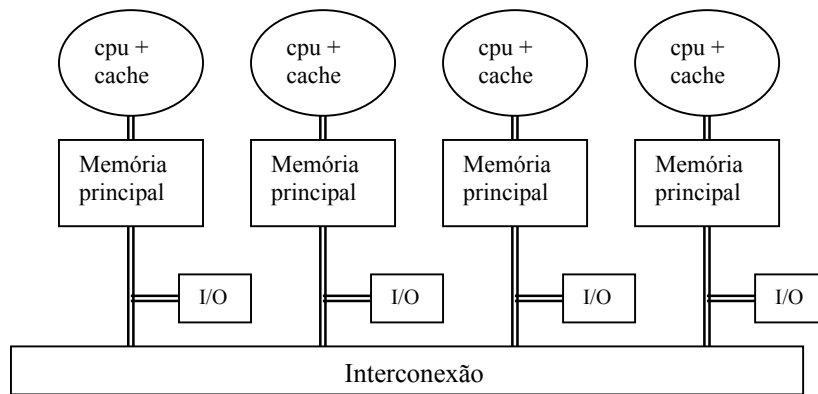


FIGURA 3.2 – Arquitetura paralela de memória distribuída.

As arquiteturas do tipo MIMD com memória distribuída, que também são denominadas *multicomputadores*, contêm espaços disjuntos de endereçamento de memória. Quando um problema é dividido entre os processadores e há dependência de dados entre eles, há a necessidade de comunicação entre as máquinas via rede. A identificação dos tempos de comunicação e a busca de sua minimização devem ser tratadas como parte da aplicação. Uma arquitetura deste tipo pode ser composta por nós que podem ser, por exemplo, máquinas SMP.

Quanto ao espaço físico de memória, as máquinas de memória distribuída são classificadas como *No Remote Memory Access* (NORMA) [16], permitindo somente acesso local.

Os multicomputadores podem ser divididos em duas categorias: *Massive Parallel Processors* (MPP) e *Cluster of Workstations* (COW) [16][17].

Máquinas MPP são multicomputadores compostos por um grande número de processadores, centenas ou milhares, fortemente acoplados através de uma rede de alta velocidade. Geralmente, são arquiteturas de custo elevado, pois utilizam processadores específicos e redes de interconexão proprietárias.

Já as máquinas COW, também chamadas de *Network of Workstations* (NOW) são construídas a partir de estações de trabalho completas, ligadas por redes de interconexão tradicionais.

Uma variação de mais baixo custo é a das arquiteturas baseadas em *cluster computing*, compostas por computadores comuns, interligados fisicamente por redes locais, em que os nós podem trabalhar juntos de forma integrada ou isolados, se necessário. Um exemplo deste tipo de arquitetura é a máquina *cluster* do INPE-DAS, que foi utilizada no estudo de um processo paralelo otimizado para a produção de mapas da RCFM.

Atualmente, também há a possibilidade de se executar aplicações paralelas em escala mundial com a utilização de computadores independentes conectados via Internet. Estes computadores são agrupados sob o nome de *Grid Computing*. Este tipo de plataforma é indicado para execução de aplicações paralelas que requerem pouca comunicação e tem sido ainda muito pouco utilizado. Diversas pesquisas têm sido desenvolvidas para disponibilizar este tipo de serviço, visando encontrar soluções efetivas e práticas para sua utilização.

3.3 – Paradigmas de Programação Paralela

Basicamente, em arquiteturas paralelas, o problema a ser resolvido é dividido entre os processadores disponíveis, devendo produzir a mesma solução que um único processador seqüencial. O problema pode ser decomposto de forma que processadores diferentes executem diferentes tarefas (*paralelismo de tarefas*) ou que processadores diferentes executem as mesmas instruções em dados diferentes (*paralelismo de dados*).

O paralelismo de tarefas é uma estratégia para se obter mais rapidamente resultados de tarefas grandes e complexas, através da divisão destas tarefas em tarefas menores que são distribuídas entre vários processadores e que são executadas simultaneamente. O paralelismo de dados consiste em dividir um conjunto de dados entre vários

processadores que executam um mesmo programa, cada um deles executando as instruções com uma parte dos dados originais.

Estes dois paradigmas de programação paralela podem ser aplicados tanto em multiprocessadores como em multicomputadores, podendo ser ainda aplicado um tipo misto, combinando paralelismo de dados e tarefas.

Duas classes de implementações de programação paralela [18] emergiram com maior aceitação pelos usuários: o uso de linguagens com bibliotecas de troca de mensagens ou *message passing*, aplicáveis ao paralelismo de dados, tarefas ou ambos; e o uso de linguagens com suporte ao paralelismo de dados ou *data parallel*.

Nas implementações do tipo *message passing*, o mesmo programa é executado em cada processador de um conjunto MIMD. O programa controla a movimentação dos dados entre os processadores através de chamadas a rotinas de comunicação, normalmente disponíveis em uma determinada biblioteca. O programador controla a distribuição dos dados e a comunicação entre processadores, sendo ainda o responsável pela organização dos processadores para que possam operar em conjunto. Os tempos de comunicação entre os processadores são adicionados ao tempo de execução do programa, passando a existir uma troca entre o custo da comunicação e a eficiência da execução em paralelo.

Message passing é ideal para máquinas com memória distribuída, em que cada processador tem sua própria memória. Sendo o espaço de endereçamento disjunto, a comunicação entre os nós é feita através da troca de mensagens via *send/receive*. Contudo, *message passing* possui algumas desvantagens. Como cada processador realiza uma tarefa isolada e executa códigos diferentes, o entendimento e a manutenção do programa tornam-se muito difíceis. Neste modelo, a troca de mensagens deve se manter mínima e pode ser uma tarefa exaustiva para o programador. É extremamente importante que as mensagens trocadas tenham destino e retorno, evitando que um processador fique aguardando uma mensagem que não foi enviada. Assim, o modelo

troca de mensagens fornece controle total ao programador, sendo de sua responsabilidade o eficiente intercâmbio entre os processadores.

Para o caso de espaço de endereçamento único, seja utilizando-se máquinas de memória compartilhada ou máquinas de memória distribuída em que as memórias fisicamente separadas são acessadas num espaço de endereçamento compartilhado logicamente, a troca de dados entre os nós é feita via *load/store*. Neste caso, há a implicação de se manter a coerência de *cache*. A mudança de variáveis pode causar problemas caso um processador tenha modificado variáveis na sua própria memória *cache* e ainda não na memória principal, e outro processador acesse estas mesmas variáveis que ainda não foram atualizadas.

Na comunicação com o uso de memória compartilhada [19] qualquer processador pode fazer referência a qualquer endereço proporcionando melhor programabilidade e menor *overhead* de comunicação, enquanto a comunicação por troca de mensagens implica em um hardware mais simplificado se comparado ao suporte para coerência de *cache* e na necessidade de se considerar os custos de comunicação.

Nas implementações do tipo *data parallel* tem-se comandos que suportam operações com conjuntos executadas em paralelo, em que todos os processadores envolvidos executam a mesma operação em um subconjunto dos dados originais. O compilador automaticamente provê a necessária comunicação entre os processadores.

A programação paralela de dados se encarrega de definir operações coletivas em matrizes e vetores, ou em seus subconjuntos, de forma a distribuí-los entre um determinado número de processadores. Se um determinado algoritmo puder ser expresso nestes termos, provavelmente a implementação paralela será eficiente.

Este paradigma é baseado em um conjunto de operações que forma a base para a implementação de algoritmos paralelos, descritas a seguir:

- Diretivas para distribuição de dados, que permitem ao programador ter o controle sobre a distribuição dos dados entre processadores e assim minimizar a comunicação entre eles, mantendo todos os processadores ocupados e realizando as operações em paralelo de forma a obter o maior desempenho possível;
- Operações com elementos de matrizes ou vetores, que podem ser realizadas em paralelo;
- Seções de matrizes ou vetores são identificadas por seus índices, permitindo que funções matemáticas possam ser aplicadas a subconjuntos destes;
- Operações com condicionais, que permitem que determinadas operações sejam aplicadas a um subconjunto de uma matriz ou vetor selecionado por uma máscara condicional ou uma expressão lógica ou aritmética;
- Operações de redução, que produzem um resultado derivado de uma combinação de elementos de uma matriz ou vetor como, por exemplo, uma busca pelo maior ou menor elemento de uma matriz ou a determinação do total de elementos nulos de uma matriz;
- Operações de deslocamento ou transposição de linhas e colunas de uma matriz, sendo que determinados deslocamentos podem ser executados em paralelo;
- Operações de varredura, que fornecem um resultado acumulativo de uma varredura dos elementos de um conjunto ou subconjunto, seguindo uma determinada lógica ou expressão;
- Comunicações generalizadas, quando se deseja combinar elementos em posições diferentes ou quando se deseja movimentar certos elementos para outras posições.

Este conjunto de funcionalidades facilita o paralelismo, permitindo assim, um maior desempenho da aplicação. Uma implementação do tipo *data parallel* tem a vantagem de ser mais simples de se programar e gera um código de mais fácil manutenção que implementações do tipo *message passing*, que por sua vez, permitem maior controle ao programador e maior portabilidade entre sistemas que utilizem bibliotecas uniformes.

A forma de implementação mais conhecida para o paradigma de paralelismo de dados em arquiteturas MIMD é a denominada *single-program, multiple-data* (SPMD) [20][21], em que os processadores executam as mesmas instruções em dados diferentes, identificados através de seu *rank* ou identificador. Por exemplo, na paralelização de uma malha, o processador identifica a faixa de iterações que lhe cabe através de seu *rank*, ou seja, *rank* 0 realiza a primeira faixa de iterações, *rank* 1, a segunda, e assim sucessivamente.

3.4 – Ambientes de Programação Paralela

Há vários ambientes de programação eficientes que dão suporte à programação paralela executáveis tanto em multiprocessadores como em multicomputadores. As linguagens e bibliotecas mais comuns são o *High Performance Fortran* (HPF), extensão do Fortran 90 visando o paralelismo de dados; o *Parallel Virtual Machine* (PVM) e o *Message Passing Interface* (MPI), bibliotecas de comunicação por troca de mensagens, aplicáveis em paralelismo de dados e tarefas. Outro padrão é o OpenMP, também uma extensão de linguagens (Fortran ou C/C++), aplicável em paralelismo de dados e tarefas, e executável, por enquanto, em multiprocessadores.

A seguir, são descritas algumas características importantes das linguagens de programação Fortran 90 e HPF, fundamentais no paradigma de programação paralela de dados, modelo adotado neste trabalho.

3.4.1 – Fortran 90/95

O Fortran 90 [22] é uma evolução do Fortran 77, desenvolvido com o objetivo de se obter a máxima eficiência de um código, onde novos comandos permitem explorar o paralelismo e estender um conjunto de operações com vetores. Algumas características do Fortran 77 foram substituídas ou até consideradas obsoletas, e novas características foram acrescentadas.

O Fortran 90 trouxe um novo formato de edição que permite a utilização de uma formatação livre, sem colunas reservadas, podendo as linhas conter até 132 colunas. Permite o uso de declarações múltiplas em uma mesma linha e a possibilidade de se acrescentar comentários na linha da declaração. Não é sensível a letras maiúsculas e minúsculas e permite a utilização de nomes longos com até 31 caracteres e o uso do caractere *underscore*.

Algumas características importantes do Fortran 90 são:

- A nova notação para vetores, que permite que operações aritméticas possam ser aplicadas diretamente a vetores e matrizes, ou a seções destes, sendo estas operações conceitualmente realizadas em paralelo;
- A inclusão de funções intrínsecas, como as operações de redução SUM (soma dos elementos de um vetor) ou MAXVAL (procura do maior elemento de um vetor);
- A introdução de armazenamento dinâmico, que permite a alocação de vetores temporários e uso de ponteiros, sob controle do usuário;
- A possibilidade de se definir tipos de dados, que facilita a criação de novas variáveis, fornecendo uma poderosa ferramenta orientada a objetos;
- A disponibilidade da declaração KIND, que permite maior portabilidade ao código, possibilitando a definição da precisão de variáveis de forma parametrizada;

- A recursão é permitida, o que facilita a solução de algoritmos complexos;
- A utilização de módulos, que permite a definição global de tipos de dados, variáveis, interfaces e subrotinas, proporcionando funcionalidade ao código;
- Novas estruturas para as malhas DO–ENDDO, que reduzem a necessidade de se usar rótulos. O comando EXIT passa a permitir uma saída mais limpa de uma malha e o comando CYCLE, que permite que se abandone uma iteração corrente de uma malha e se reinicie na próxima iteração;
- A inclusão do bloco de controle SELECT CASE, que facilita a elaboração de comparações de forma mais eficiente que o comando IF.

O Fortran 95 é uma revisão do Fortran 90, acrescentando novas características à linguagem, como por exemplo, construções FORALL, descritas na seção seguinte, funções puras e alguns novos procedimentos intrínsecos, desaparecendo com algumas características consideradas obsoletas.

Estas características tornam o Fortran 90 e 95 recomendáveis para o desenvolvimento de aplicações científicas e de engenharia.

3.4.2 – *High Performance Fortran - HPF*

O HPF [23] é o resultado da necessidade de se padronizar a linguagem Fortran para o paradigma de programação paralela de dados. O HPF é uma extensão do Fortran 90, desenvolvida para incorporar o paralelismo de dados, e que permite ao programador expressar o paralelismo de uma forma relativamente simples.

O paralelismo é definido através de diretivas, que são inseridas em um código escrito em Fortran 90. A utilização de diretivas HPF exige que, inicialmente, seja definida uma

grade conceitual de processadores e que os dados sejam distribuídos entre os processadores envolvidos, para que sejam executados os cálculos necessários. As diretivas HPF sugerem uma estratégia de implementação ao compilador, que poderá ignorá-las ou modificá-las. Assim, um programa em HPF pode ser compilado também de forma seqüencial, sendo as diretivas HPF tratadas como comentários em um compilador Fortran 90.

As diretivas HPF seguem o formato *!HPF\$<diretiva>* e são descritas a seguir:

- *PROCESSORS*: cria um arranjo abstrato ou uma grade de processadores, especificando o nome, número de dimensões e a extensão de cada dimensão;
- *DISTRIBUTE*: especifica um particionamento de dados em um arranjo de processadores, indicando a partição uniforme dos dados em blocos de elementos adjacentes (*BLOCK*) ou de forma cíclica (*CYCLIC*) entre os processadores;
- *ALIGN*: especifica mapeamento entre objetos, sendo mais eficientes as operações realizadas entre elementos de dados alinhados, pois os elementos são mapeados para um mesmo processador;
- *TEMPLATE*: declara um espaço abstrato de posições indexadas, que pode ser distribuído permitindo um alinhamento mais fácil e claro de vetores.

Por exemplo, a distribuição em blocos (*BLOCK*) de um vetor unidimensional de 16 elementos $X=\{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16\}$, entre 4 processadores (p_1, p_2, p_3, p_4), acarretará que os elementos do vetor X em cada processador sejam os seguintes: $x_1=\{1,2,3,4\}$, $x_2=\{5,6,7,8\}$, $x_3=\{9,10,11,12\}$, $x_4=\{13,14,15,16\}$.

Já a distribuição cíclica (*CYCLIC*) acarretará que os elementos em cada processador sejam os seguintes: $x_1=\{1,5,9,13\}$, $x_2=\{2,6,10,14\}$, $x_3=\{3,7,11,15\}$, $x_4=\{4,8,12,16\}$.

A distribuição em blocos (*BLOCK*) é útil principalmente quando são necessários cálculos entre elementos consecutivos do vetor original, já a forma cíclica (*CYCLIC*) distribui os dados seqüencialmente entre os processadores, podendo ser utilizada quando existir a necessidade de uma distribuição eqüitativa de carga computacional.

Ao optar pelo paradigma de programação paralela de dados, deve-se considerar que o HPF envolve uma troca entre paralelismo e comunicação. Ao se inserir diretivas HPF a um programa seqüencial, deve-se levar em conta que um aumento do número de processadores pode envolver um aumento da comunicação. Deve-se procurar seguir as recomendações:

- buscar um balanceamento da carga computacional;
- procurar manter a localidade dos dados;
- utilizar a notação de vetores do Fortran 90.

O Fortran 90 possui uma notação para conjuntos que explicita o paralelismo através da designação de conjuntos, mascaramento de elementos, seccionamento de subconjuntos e transposição de linhas e colunas. O HPF acrescenta a estas características a declaração *FORALL*, a diretiva *INDEPENDENT* e a atribuição *PURE*, que indicam operações potencialmente paralelas, descritas a seguir:

- *FORALL*: construtor para a distribuição paralela de dados, que permite uma atribuição flexível a seções de um conjunto sem forçar a designação individual dos elementos; e que garante o mesmo resultado caso o código seja executado em série ou em paralelo.
- *INDEPENDENT*: propriedade de uma malha interna (*DO-ENDDO*) ou de uma declaração *FORALL*, que assegura que as iterações podem ser feitas concorrentemente em uma malha interna, ou que as operações dos índices sejam isoladas para o *FORALL*.

A importância da diretiva *INDEPENDENT* pode ser mostrada através do exemplo a seguir, conforme descrito por Ewing [24]. A expressão do lado direito (RHS - *Right Hand Side*) é executada em paralelo e os resultados obtidos são atribuídos ao lado esquerdo (LHS - *Left Hand Side*). A execução do *FORALL* sem a diretiva *INDEPENDENT* provoca um sincronismo das expressões RHS1 e RHS2 antes da atribuição a LHS1 e LHS2. A Figura 3.3 mostra a avaliação das expressões com e sem o uso da diretiva *INDEPENDENT*, onde S1, S2, S3 e S4 denotam barreiras de sincronismo.

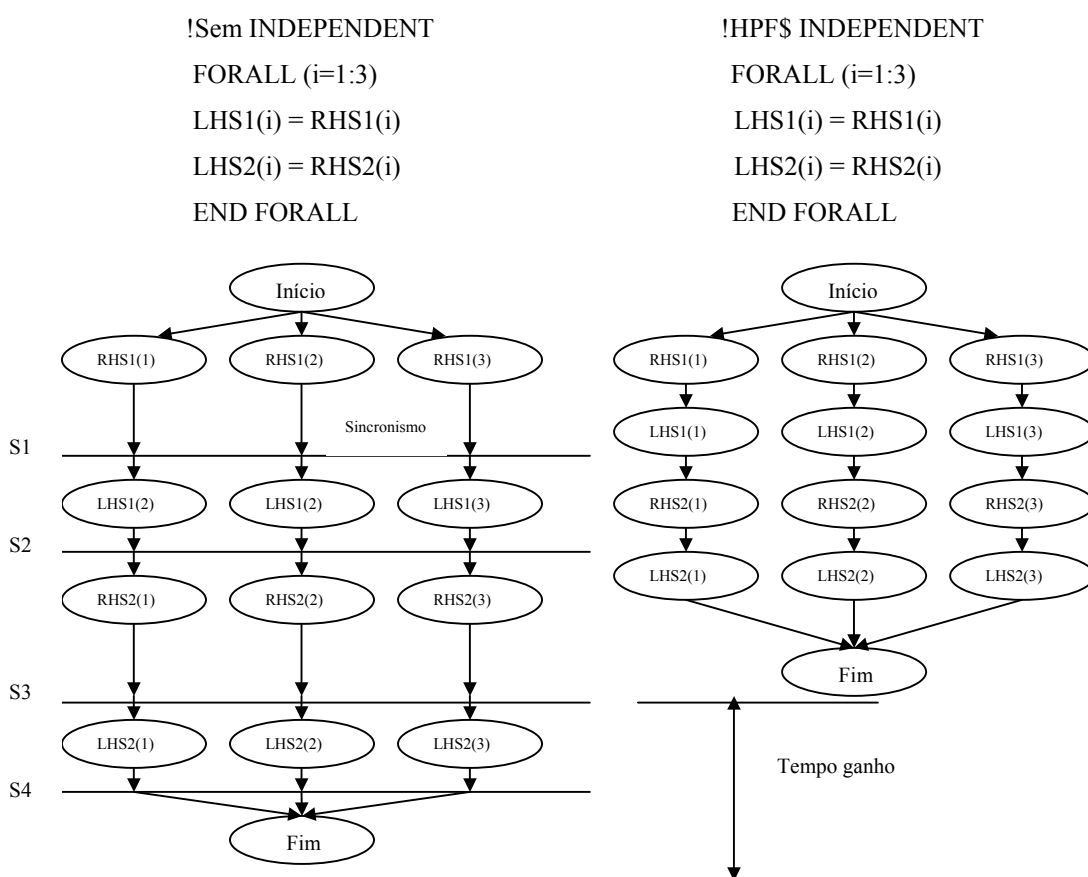


FIGURA 3.3 – Uso de *FORALL* e *INDEPENDENT*.
 FONTE: adaptada de Ewing [24].

- *PURE*: o atributo *PURE* aplica-se a funções e subrotinas, garantindo que um procedimento não altera seus dados de entrada, apenas retorna um valor no caso de funções ou modifica parâmetros *INTENT(OUT)* ou *INTENT(INOUT)* no caso de subrotinas. Permite identificar funções seguras para o uso da declaração *FORALL*, em que os índices podem ser repassados ao procedimento sem qualquer alteração, permitindo o paralelismo implícito do *FORALL*.

3.4.3 – *Message Passing Interface* - MPI

MPI [25][26][27] é uma biblioteca de comunicação por troca de mensagens desenvolvida para permitir a escrita de programas paralelos que incorporem o paralelismo de dados ou de tarefas. Particularmente, no primeiro caso, o MPI utiliza o modelo SPMD. A biblioteca MPI é comumente utilizada e constitui um padrão, o que facilita sua portabilidade, podendo ser usada em programas desenvolvidos em C, C++ ou Fortran.

Neste trabalho, essa biblioteca foi utilizada na subrotina de convolução de matrizes usando FFT, como uma alternativa para o cálculo de uma matriz transposta, buscando minimizar o tempo de comunicação excessivo da implementação HPF. As funções MPI descritas a seguir são as que foram utilizadas no programa, sendo apresentados somente alguns conceitos básicos para sua utilização.

Basicamente, este estudo utilizou as funções *MPI_Send* e *MPI_Recv*. Na primeira função, um processador (*source*) envia uma mensagem para um determinado processador (*dest*) e na segunda função, o processador destino recebe essa mensagem do processador fonte. Essas funções contêm a mensagem, o tipo e dimensão dos dados, e informações adicionais (*envelope*) necessárias para a realização da comunicação, como os identificadores (*ranks*) dos processadores fonte e destino, o tipo de comunicação (*communicator*) utilizado e um identificador (*tag*) da mensagem, obedecendo à sintaxe:

int MPI_Send (void message, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)*

int MPI_Recv (void message, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Status* status)*

As funções *MPI_Comm_rank* e *MPI_Comm_size* também foram utilizadas nesta implementação. A primeira retorna o *rank* do processador e a segunda o número de processadores envolvidos, obedecendo à sintaxe:

int MPI_Comm_rank (MPI_Comm comm., int rank)

int MPI_Comm_size (MPI_Comm comm, int size)

A utilização e os resultados do estudo da implementação mista utilizando HPF e MPI são descritos no capítulo 5.

CAPÍTULO 4

TÉCNICAS DE OTIMIZAÇÃO DE CÓDIGO

Muitas aplicações científicas são elaboradas visando à solução de um determinado problema, sem grande preocupação com o seu desempenho. A rara utilização de técnicas de otimização pode produzir aplicações de baixo rendimento. Muitas vezes estas aplicações são portadas de outras plataformas ou de aplicações mais antigas, podendo fazer uso de declarações já obsoletas, e acabam impedindo que todos os recursos disponibilizados por uma arquitetura sejam utilizados eficientemente.

Geralmente, o problema da otimização de desempenho de uma aplicação consiste em encontrar a melhor sintonia entre as estruturas utilizadas no código e o hardware, exigindo um diagnóstico minucioso sobre o desempenho da implementação original. A otimização se torna mais fácil quanto maior for o detalhamento das informações de desempenho disponíveis.

A necessidade de otimização de uma aplicação implica em uma análise detalhada do código fonte, levando em conta a estrutura do programa, principalmente as chamadas a subrotinas e a forma de utilização de malhas internas. O uso inadequado dessas malhas provoca consumo excessivo de tempo de CPU, contribuindo para a ineficiência da aplicação.

A identificação de trechos do programa onde existam gargalos de execução é obtida através da temporização das chamadas a subrotinas e das malhas internas. Essa tarefa, normalmente atribuída às ferramentas de *profiling*, realiza um levantamento do perfil dos tempos de execução das subrotinas envolvidas; porém, não fornecem, em geral, informações sobre os tempos de execução das malhas internas. A utilização dessas ferramentas, muitas vezes disponíveis como funções de bibliotecas nos compiladores das linguagens C e Fortran, depende de chamadas a rotinas internas do sistema – *system calls*, que trazem informações de tempo das chamadas a subrotinas. Para temporização

de malhas internas de um programa podem ser utilizadas rotinas que fazem a leitura do relógio do sistema, chamadas no início e no final de cada trecho ou malha a ser monitorado.

Identificados os trechos críticos de um programa, podem se utilizar técnicas de otimização para uma execução seqüencial, e posteriormente, se for o caso, se fazer uso da computação paralela. Os conceitos utilizados para a otimização de uma aplicação seqüencial podem ser estendidos a uma aplicação paralela. A aplicação deve ser estruturada de tal forma que a adição de processadores implique em uma redução proporcional do tempo total de execução.

4.1 – Otimização de Programas Seqüenciais

A otimização de uma aplicação seqüencial inicia-se com a instrumentação do programa para verificação das condições adversas de acesso à memória e identificação dos gargalos de tempo de execução, que podem dar uma orientação no caminho a seguir para se otimizar a aplicação.

Técnicas de otimização clássicas [14] visam, basicamente, obter o máximo desempenho do processador e otimizar o acesso à memória.

Uma das técnicas de obtenção do máximo de desempenho de um processador consiste em otimizar o uso do *pipelining*, técnica onde o hardware do computador executa mais de uma instrução simultaneamente sem necessidade de aguardar o término de uma instrução para executar a próxima. Para melhor desempenho, deve-se buscar manter os *pipelines* continuamente ocupados, explorando o paralelismo interno do processador.

Para otimização do acesso à memória, deve-se explorar a localidade espacial e temporal dos dados. Para isto, é visado o acesso a dados contíguos da memória ou *cache* e o acesso a dados acessados recentemente, ainda disponíveis na memória ou no *cache*.

Para isto, um compilador eficiente faz uso de técnicas de otimização de forma a maximizar a utilização simultânea dos *pipelines* do processador e a utilização eficiente dos registradores e a minimizar o número de erros de acesso às memórias *cache* (*cache misses*).

Técnicas conhecidas de otimização, tais como desenrolar malhas – *loop unrolling*, fusão de malhas, colapso de malhas, inversão de malhas, blocagem e remoção de condicionais internos, devem ser implementadas nos trechos em análise, sempre que possível.

O procedimento a ser adotado para a análise de desempenho de um trecho de um programa implica, inicialmente, na identificação de pontos onde os tempos de execução estejam comprometendo o desempenho do programa.

As ferramentas disponíveis para levantamento do perfil de tempos de execução ou *profiling*, como exemplo o utilitário *gprof*, permitem identificar as subrotinas consumidoras de tempo de CPU. Após esta verificação inicial, deve-se inspecionar o código das subrotinas mais críticas, procurando por situações onde possam existir interações adversas com a arquitetura, principalmente trechos que possam provocar *cache misses*.

A inspeção de um código deve iniciar com a pesquisa dos trechos que possam contribuir para a perda de desempenho, tais como:

- chamadas a subrotinas;
- referências indiretas ou ambíguas à memória;
- testes internos a malhas;
- conversão de tipos;
- testes de entrada de caracteres;
- uso de variáveis não necessárias no trecho ou malha.

O uso de subrotinas visa à obtenção de um código mais estruturado, porém implica em um gasto adicional, devido à interrupção do programa e ao contexto ser salvo e recuperado após execução da subrotina. As chamadas a subrotinas devem ser analisadas e se o tempo gasto for muito superior ao tempo de execução em si, elas devem ser evitadas. O uso de *inlining*, ou seja, a inserção do código da subrotina substituindo a chamada, pode reduzir consideravelmente as faltas de acesso à memória *cache*. As referências indiretas ou ambíguas à memória impedem que o compilador faça otimizações baseadas na localidade espacial dos dados. Os testes internos em malhas, tais como os condicionais, devem ser retirados da malha, sempre que possível, uma vez que a verificação do condicional implica em perda de desempenho. A conversão de tipos de dados como, por exemplo, referenciar uma variável inteira com um valor de dupla precisão também implica em perda de desempenho. Alguns desses itens também contribuem na restrição ao paralelismo e, portanto, devem ser evitados, tais como chamadas a subrotinas, referências indiretas à memória, testes internos a malhas e uso de ponteiros ambíguos.

Primeiramente, recomenda-se estudar as malhas internas, visando sempre o paralelismo. O padrão de referência à memória deve ser analisado com o objetivo de dar preferência ao acesso contíguo a posições vizinhas de memória.

Após a adaptação do programa com o uso das técnicas de otimização, deve-se inspecionar a validade das otimizações efetuadas. Deve-se verificar se os valores tratados no trecho não sofreram alteração e se as medidas de monitoração indicam otimização de desempenho.

Das várias técnicas de otimização de malhas internas duas serão exemplificadas a seguir, como o *loop unrolling* e a blocagem, que foram utilizadas neste trabalho.

- *Loop unrolling*

Esta técnica é a forma mais básica de otimização de malhas internas, realizada automaticamente pela maioria dos compiladores. Com o uso do *loop unrolling* mais instruções são executadas por iteração, causando uma redução do número de iterações de uma malha. O conceito pode ser entendido através do exemplo seguinte, uma malha em que é calculada uma soma de matrizes unidimensionais A e B de dimensão N.

```
DO i = 1, N
  A(i) = A(i) + B(i) * K
ENDDO
```

Com a expansão da malha em quatro linhas (*loop unrolling*), tem-se:

```
DO i = 1, N, 4
  A(i) = A(i) + B(i) * K
  A(i+1) = A(i+1) + B(i+1) * K
  A(i+2) = A(i+2) + B(i+2) * K
  A(i+3) = A(i+3) + B(i+3) * K
ENDDO
```

Neste exemplo, pode-se observar a exposição ao paralelismo, permitindo uma utilização mais eficiente da unidade de operações de ponto flutuante. Nesta situação, são lidas quatro linhas de elementos adjacentes, reduzindo o número de iterações da malha. Ainda, se o número de iterações N não for divisível por 4, serão necessárias uma, duas ou três iterações extras para adequação das operações realizadas na malha. Neste caso, deve-se incluir uma malha pré-condicionadora, como a seguir:

```
ii = IMOD (N , 4)
DO i = 1, ii
  A(i) = A(i) + B(i) * K
ENDDO
```

```

DO i = 1, N, 4
  A(i) = A(i) + B(i) * K
  A(i+1) = A(i+1) + B(i+1) * K
  A(i+2) = A(i+2) + B(i+2) * K
  A(i+3) = A(i+3) + B(i+3) * K
ENDDO

```

O acesso contíguo dos elementos de uma matriz pela leitura de toda uma linha, no caso da linguagem C, ou de uma coluna, no caso de Fortran, reduz de forma considerável o tempo de execução do trecho, uma vez que há um aumento dos acertos ao *cache* de dados. Nos casos de matrizes unidimensionais, a técnica de *loop unrolling* torna-se essencial como técnica de otimização.

- Blocagem

Algumas operações com matrizes não dão preferência a acessos a elementos adjacentes na memória *cache*, acessando os elementos com passos (*strides*) diferentes. Um caso clássico de utilização de blocagem é mostrado abaixo, em um trecho de programa em Fortran que calcula uma soma de vetores.

```

DO i = 1, N
  DO j = 1, N
    A(j, i) = A(j, i) + B(i, j)
  ENDDO
ENDDO

```

A malha interna definida pelo contador *j* faz a leitura dos *N* elementos adjacentes da matriz *A*: *A*(0, 0); *A*(1, 0); *A*(2, 0); ... ; *A*(*N*, 0). A leitura da matriz *B* é feita em linhas, implicando na leitura de elementos não adjacentes: *B*(0, 0); *B*(0, 1); *B*(0, 2); ... ; *B*(0, *N*). O próximo elemento adjacente de *B* será lido após *N* elementos. Neste exemplo, a matriz *A* tem um passo unitário, enquanto a matriz *B* tem um passo igual a *N*. Os elementos adjacentes da matriz *B* somente serão acessados após cada coluna ser lida, e por essa razão, podem não estar disponíveis na memória *cache* de dados.

A técnica de blocagem é indicada para este caso. O acesso à memória é realizado em pequenos blocos, ora dando preferência a uma matriz, ora dando preferência à outra, podendo melhorar a taxa de acertos ao *cache* de dados.

A blocagem pode ser implementada, por exemplo, para um bloco 2x2 com $N = 1000$. Neste trecho não há dependências entre os elementos das matrizes para iterações distintas da malha. A dimensão das matrizes é múltipla do *stride* (no exemplo, igual a 2), assim, não há necessidade de uma iteração extra. Se N não fosse divisível por 2, nem todas as iterações originais estariam cobertas, e uma iteração extra teria que ser adicionada.

A soma de vetores utilizando a blocagem com *stride* igual a 2 é apresentada a seguir.

```
DO i = 1, N, 2
  DO j = 1, N, 2
    A(j, i) = A(j, i) + B(i, j)
    A(j+1, i) = A(j+1, i) + B(i, j+1)
    A(j, i+1) = A(j, i+1) + B(i+1, j)
    A(j+1, i+1) = A(j+1, i+1) + B(i+1, j+1)
  ENDDO
ENDDO
```

As primeiras duas linhas dão preferência a matriz A, pois executam uma leitura de elementos adjacentes $A(j, i)$ e $A(j+1, i)$, enquanto as primeira e terceira linhas dão preferência a matriz B, lendo os elementos $B(i, j)$ e $B(i+1, j)$.

A técnica de blocagem na realidade consiste na aplicação da técnica de *loop unrolling* nas duas malhas. Esta técnica parte da idéia de se agrupar elementos adjacentes da matriz, fazendo com que duas ou mais colunas adjacentes sejam lidas simultaneamente, melhorando os acertos ao *cache* de dados.

4.2 – Monitoração de Programas Paralelos

A metodologia apresentada para programas seqüenciais pode ser facilmente estendida à programação paralela. Uma aplicação científica pode ter seu desempenho otimizado com a instrumentação seqüencial em uma máquina e, posteriormente, ser executada em várias máquinas em paralelo. Os trechos podem ser novamente monitorados e uma análise de desempenho pode ser realizada em cada máquina, com o objetivo de validar as alterações efetuadas.

Um paradigma de computação paralela deve ser adotado, como o modelo *message passing* (troca de mensagens) ou *data parallel* (programação paralela de dados). Ambos possuem implementações eficientes, sejam bibliotecas ou compiladores.

Basicamente, em máquinas paralelas, o problema a ser resolvido é dividido entre os processadores disponíveis, devendo produzir a mesma solução que um processador seqüencial. E para se obter uma implementação paralela eficiente, deve-se procurar minimizar a relação entre o tempo de comunicação e o tempo de processamento, bem como buscar um balanceamento da carga computacional entre os processadores envolvidos [28]. O tempo de comunicação corresponde à soma do tempo gasto para a inicialização da mensagem que será transmitida e o tempo efetivo para que a mensagem chegue ao destino especificado. Este tempo é crítico, podendo aumentar consideravelmente o tempo de execução total. O balanceamento da carga computacional é importante para que todas as máquinas sejam utilizadas com o mais próximo possível de sua capacidade de processamento nominal. O ideal é que o trabalho seja dividido igualmente entre os processadores envolvidos, que devem executar sua parte do trabalho ao mesmo tempo, minimizando o tempo de espera entre pontos de sincronização.

O desempenho de um programa paralelo pode ser quantificado através de duas medidas, o *speedup* e a eficiência [14]. *Speedup* [29] é a relação entre o melhor tempo de execução seqüencial e o tempo de execução paralela de um mesmo problema. É

expresso pela equação $S_p = T_i / T_p$, onde T_i é o tempo da execução seqüencial e T_p o tempo da execução paralela. A eficiência fornece a fração de tempo que os processadores estão sendo utilizados em processamento e é expressa pela equação $\eta_p = S_p / p$, onde p é o número de nós ou processadores. São medidas subjetivas, sendo que para uma máquina paralela com p processadores, o *speedup* ideal seria igual a p (*speedup* linear) e a eficiência ideal seria igual a 1 ou 100%.

Na aplicação em estudo, o programa foi inicialmente reestruturado utilizando a notação do Fortran 90 e foi instrumentado para investigação dos trechos críticos em tempo de processamento. Foi adotado o modelo de programação paralela de dados, onde o HPF tem sua melhor expressão. Através da inserção de diretivas HPF, foi realizada uma investigação visando uma correta distribuição dos dados entre as máquinas envolvidas.

4.3 – Instrumentação para Medidas de Tempo

Neste trabalho, o utilitário gráfico *pgprof* foi utilizado para levantamento do perfil de tempos de execução das subrotinas do programa. O utilitário fornece uma visão geral da execução de forma rápida e automática, informando o número de chamadas feitas a cada subrotina, seu tempo de execução e seu custo, ou seja, o tempo de execução gasto nesta subrotina acrescido dos tempos gastos nas subrotinas chamadas por ela.

O utilitário gráfico não fornece informações sobre os tempos de execução das malhas internas do programa principal ou das subrotinas, e para sua monitoração foi utilizada uma subrotina de temporização do sistema operacional.

A subrotina *timer*, apresentada na Figura 4.1, foi utilizada para fazer as medidas de tempo das malhas internas. A subrotina *timer* calcula a diferença entre um tempo medido e um tempo anteriormente calculado e passado como parâmetro. As medidas de tempo, por sua vez, são efetuadas pela subrotina *system_clock*.

```

SUBROUTINE timer(end_time,init_time)
  USE nrtype
  REAL(SP), INTENT(IN) :: init_time
  REAL(SP), INTENT(OUT) :: end_time
  INTEGER(I4B) :: finish,rate
  call system_clock(COUNT = finish, COUNT_RATE = rate)
  end_time = float(finish) / float(rate) - init_time
END SUBROUTINE timer

```

FIGURA 4.1 – Subrotina *timer*.

O programa foi convenientemente subdividido e em cada trecho ou malha a ser monitorado foram feitas chamadas a subrotina *timer* no início e no final do trecho para obtenção do tempo de execução do respectivo trecho, como representado na Figura 4.2.

```

...
call timer(t1, 0.0)

      trecho a ser monitorado

call timer(t2, t1)
...

```

FIGURA 4.2 – Instrumentação do programa.

A instrumentação contribui para o aumento do tempo de processamento, porém o tempo gasto com as chamadas a subrotina de instrumentação foi insignificante comparado aos tempos dos trechos monitorados, não influenciando as medidas realizadas dos tempos de execução.

O próximo capítulo descreve a otimização do programa para produção de mapas da RCFM, partindo-se da otimização de um programa seqüencial e posterior implementação paralela, adotando-se o modelo de paralelismo de dados.

CAPÍTULO 5

OTIMIZAÇÃO DA PRODUÇÃO DE MAPAS DA RCFM

Este capítulo descreve a otimização de um aplicativo para produção de mapas da RCFM, originalmente escrito em Fortran 77, e reestruturado com as declarações e notação do Fortran 90. Uma análise dos tempos de execução de alguns trechos do programa mostra os pontos onde há excesso de consumo de tempo de CPU. Esses trechos são otimizados e é feita uma comparação entre as duas versões sequenciais. Alguns conceitos sobre convolução de matrizes usando FFT são apresentados para uma melhor compreensão da aplicação, sendo este algoritmo utilizado nos trechos identificados como maiores consumidores de tempo de CPU.

São apresentadas duas implementações paralelas, uma baseada em paralelismo de dados e uma implementação mista, combinando paralelismo de dados e tarefas, desenvolvidas durante a otimização do processo.

Nesta aplicação, para se obter uma estimativa do mapa do céu são empregados o método dos gradientes conjugados [30][31] e um filtro otimizado (normalmente *white noise*).

Um mapa do céu é representado pela equação $m = P^T * W^T * W * d$, onde d é uma série temporal de dados, P uma matriz de apontamento e W um núcleo de um filtro *white noise*.

Geralmente, o vetor que contém a série temporal de dados é muito maior que a memória disponível, sendo o processo de obtenção de mapas realizado em seções. Diversas seções de d são lidas e convoluídas com o filtro *white noise*, o que equivale à multiplicação matricial $W^T * W * d$. A série temporal ponderada é multiplicada por P^T , o que corresponde à soma dos diversos valores medidos para cada pixel do mapa. A conversão da série temporal para pixels é feita para todos os elementos que não estejam

muito próximos das extremidades de cada seção de d . O processo é repetido em toda a extensão de d , utilizando o método *overlap-add* [30] para a convolução de matrizes.

A aplicação em uso divide a produção de mapas em duas etapas. Na primeira etapa, é calculada a matriz de covariância de ruído inversa para um dado padrão de observação e um filtro *white noise* (programa *doicvm*). Na segunda, seções da série temporal são convoluídas com o filtro *white noise* e multiplicadas pela matriz de apontamento, cuja finalidade é transformar a série temporal numa série espacial, sendo utilizada a matriz de covariância de ruído inversa obtida na etapa anterior e o método dos gradientes conjugados para se obter uma estimativa do mapa (programa *domap*).

O programa *doicvm* tem por objetivo calcular a inversa da matriz de covariância de ruído (Figura 2.9) para um dado padrão de observação, representado por uma matriz de apontamento (Figura 2.6), e um filtro pré-determinado (Figura 2.7). Inicialmente, é feita a leitura de alguns parâmetros selecionáveis pelo usuário, como a dimensão dos vetores de dados, o número de arquivos e setores e o número de pixels do mapa. O programa lê um vetor que contém um núcleo do filtro (*filter*), que é tratado e inserido na matriz de correlação de ruído utilizando rotinas de convolução de matrizes usando FFT. O cálculo é finalizado com a leitura em seções de um vetor que contém o padrão de observação (*pixel*), que são adicionadas à matriz produzindo a matriz de covariância de ruído inversa, que será aplicada posteriormente à série temporal. Devido à simetria da matriz de covariância de ruído, somente os dados referentes ao triângulo inferior da matriz são calculados e armazenados.

O programa *domap* usa o método dos gradientes conjugados e um filtro otimizado para obtenção de uma estimativa do mapa do céu. O programa lê os vetores do filtro, da matriz de apontamento e da matriz de covariância de ruído inversa obtida na etapa anterior. Os cálculos são realizados para cada seção da série temporal gerando uma estimativa do mapa do céu em pixels. Rotinas para armazenamento de matrizes esparsas e simétricas são utilizadas, armazenando somente os elementos diferentes de zero e fora da diagonal do triângulo inferior da matriz resultante.

Técnicas de otimização seqüencial foram aplicadas aos dois programas, sendo que no primeiro deles foram realizados o estudo e os testes de implementações paralelas que serão apresentados neste capítulo. Os métodos utilizados podem também, ser aplicados ao segundo programa, que utiliza as mesmas rotinas do primeiro, para uma solução otimizada da aplicação por completo.

5.1 – Otimização da Aplicação Seqüencial

A otimização da aplicação seqüencial foi realizada utilizando duas máquinas de teste. Inicialmente, os testes foram realizados utilizando o compilador *Microsoft Fortran Powerstation 4.0* na máquina de testes *slb* (mono-processador, sistema operacional Windows). O código foi reestruturado e portado para a máquina *polaris*, uma arquitetura paralela de memória compartilhada, sendo utilizados os compiladores Fortran *pgf77* e *pgf90* da *Portland Group*. Os arquivos de entrada utilizados para esta primeira análise foram *filter* (dimensão 251 pontos) e *pixel* (dimensão 2410 pontos). A dimensão dos vetores utilizados para obtenção da matriz de covariância foi de 512x512.

A análise do programa iniciou com a elaboração de um perfil de tempos de execução, com o objetivo de identificar os trechos críticos em tempos de processamento. Para esta análise, o programa foi dividido em trechos representados na Figura 5.1. No primeiro trecho é feita a leitura e tratamento do vetor contendo um núcleo do filtro *white noise*, com chamadas as subrotinas *LoadRawVector*, *RemoveMonopole* e *Normalizeto1*. No trecho 2 são feitas alocações e inicializações dos vetores que serão utilizados no cálculo da parte harmônica da matriz de covariância. No trecho 3, uma malha interna faz o cálculo da matriz de correlação de ruído, com chamadas a subrotina *convlv*. No trecho 4 são feitas alocações e inicializações dos vetores que serão utilizados no cálculo da matriz de covariância inversa de ruído. No trecho 5, uma malha interna faz a leitura em seções dos arquivos contendo o vetor *pixel*, que são somados à matriz resposta do trecho 3. No trecho 6, é processada a seção final e é feita uma chamada a subrotina *SaveRawVector* que armazena a matriz resultante.

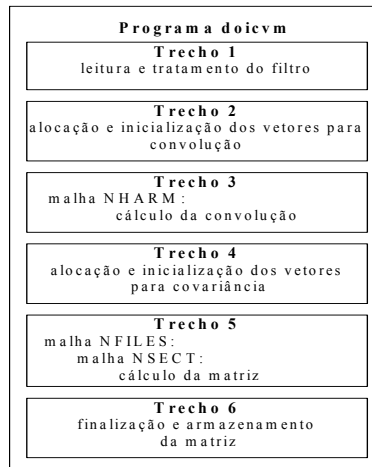


FIGURA 5.1 – Diagrama em blocos do programa *doicvm*.

Os tempos de execução de cada trecho medidos nas duas máquinas de teste são apresentados na Tabela 5.1, onde se identifica o trecho 3 como o de maior consumo de tempo de CPU.

TABELA 5.1 – Tempos de Execução Seqüencial das Malhas Internas

Trecho	t_{exec} (s) <i>slb</i>	t_{exec} (s) <i>polaris</i>
1	0,011	0,004
2	0,660	0,074
3	37,400	22,600
4	0,060	0,070
5	0,659	0,150
6	0,051	0,006
Total	43,500	23,000

Os trechos 1, 2, 4 e 6 são executados somente uma vez durante a execução do programa e são insignificantes quando comparados ao tempo total, não implicando em necessidade de otimização.

No trecho 1, as subrotinas para tratamento do filtro foram reestruturadas usando a notação para vetores do Fortran 90 e foram utilizadas operações de redução como *SUM*

e *MAXVAL*. A Figura 5.2 apresenta trechos do programa original e da versão otimizada usando Fortran 90.

Versão Fortran:	Versão Fortran 90:
...	...
DO i=1, n	
avg=avg +v(i)/n	avg=SUM(v)/n
END DO	
DO i=1, n	
v(i)=v(i)-avg	v=v-avg
END DO	
...	...
maxv=v(1)	
DO i=2, n	
IF (v(i).gt.maxv) then	
maxv=v(i)	maxv=MAXVAL(v)
END IF	
END DO	
DO i=1, n	
v(i)= v(i)/maxv	v=v/maxv
ENDDO	
...	...

FIGURA 5.2 – Uso de operações de redução.

O trecho 3 é o de maior consumo de tempo de CPU. Neste trecho, uma malha interna calcula a matriz de correlação de ruído utilizando um núcleo do filtro *white noise*. É feita uma chamada a subrotina de convolução de matrizes usando FFT para cálculo dos componentes das colunas ímpares da matriz. O processo é repetido para os componentes das colunas pares, resultando em uma matriz resposta que será aplicada à matriz de apontamento. O trecho 3 foi subdividido e foram obtidos os tempos de execução mostrados na Tabela 5.2, na máquina de testes *polaris*. Neste trecho, as subrotinas de convolução de matrizes foram identificadas como o principal problema a ser investigado neste trabalho. Alguns conceitos sobre convolução de matrizes serão apresentados na seção seguinte.

TABELA 5.2 – Tempos de Execução Sequencial do Trecho 3

Sub-trecho	Saída	$t_{exec}(s)$
Malha wtw	Vetor “white noise”	0,001
Malha colunas impares	Componente coluna impar	0,221
Subrotina convlv	Convolução	5,464
Malha harmresp(j,2*i-1)	Resposta coluna impar	0,003
Malha wtw	Vetor “white noise”	0,001
Malha colunas pares	Componente coluna par	0,217
Subrotina convlv	Convolução	5,446
Malha harmresp(j,2*i)	Resposta coluna par	0,003
Tempo total		22,940

As subrotinas de convolução foram extraídas de *Numerical Recipes in Fortran* [30]. Na subrotina *convlv* [30, p. 536] é feita a convolução de um conjunto de dados reais $data(1:n)$ com uma função de resposta $respns(1:m)$, usando FFT. O vetor $respns$ é rearranjado em um vetor de comprimento n e é feita uma chamada a subrotina *twofft* [30, p. 505], que calcula a transformada dos dois vetores $data$ e $respns$. Após a multiplicação dos elementos dos vetores transformados é feita uma chamada a subrotina *realft* [30, p. 507], que calcula a transformada inversa do vetor resultante. Estas duas subrotinas por sua vez fazem chamadas a subrotina *four1* [30, p. 501] que faz o cálculo da transformada propriamente dito.

O programa foi reestruturado usando as declarações e notação do Fortran 90, e foram feitas algumas modificações para utilização das subrotinas de FFT disponíveis em *Numerical Recipes in Fortran 90* [31]. A nova versão da subrotina *convlv* [31, p. 1253] faz uma chamada a subrotina *realft* [31, p. 1243] para cada vetor a ser convoluido. Na nova versão da subrotina *realft*, o vetor é rearranjado em um vetor de números complexos (onde cada elemento representa dois elementos do vetor real), é feita uma chamada a subrotina *four1* e novo rearranjo do vetor para números reais. Na subrotina *four1* [31, p. 1239] o vetor de entrada é rearranjado em uma matriz bidimensional, que tem suas dimensões recalculadas, e é feita uma chamada a subrotina *fourrow* [31, p. 1235] para cálculo da transformada de cada linha da matriz. A seguir é calculada a matriz transposta utilizando-se a função *transpose*, sendo feita nova chamada a subrotina *fourrow* para cálculo da transformada de cada coluna da matriz. Após esta

etapa, o vetor retorna à forma original. Nestas rotinas, o paralelismo é expresso pelas mudanças de forma do vetor.

A Tabela 5.3 apresenta uma comparação entre os tempos de execução dos subtrechos críticos, nas duas versões em Fortran 77 e Fortran 90.

TABELA 5.3 – Comparativo entre Versões Sequenciais Usando Fortran 77 e Fortran 90

Sub-trecho	$t_{exec}(s)$ F77	$t_{exec}(s)$ F90	Redução (%)
Malha colunas Impares	0,221	0,221	
Subrotina convlv	5,464	3,549	35
Malha colunas pares	0,217	0,216	
Subrotina convlv	5,446	3,628	33
Tempo total	22,940	15,440	32

Nesta tabela são apresentados os tempos de duas chamadas a subrotina de convolução feitas para uma iteração da malha. A redução média de 34% no tempo de processamento da subrotina de convolução (*convlv*) e 32% no tempo total do programa mostram a eficiência da reestruturação do programa usando as declarações e notação do Fortran 90.

No trecho 5, é calculada a matriz de covariância de ruído. A Tabela 5.4 apresenta uma comparação entre o programa original e o otimizado, mostrando os tempos de execução do trecho e o tempo total do programa para padrões de diferentes dimensões, obtidos na máquina de teste *polaris*.

TABELA 5.4 – Execução Seqüencial do Trecho 5

n_p	Fortran 77		Fortran 90		Redução
	$t_{exec}(s)$	$t_{total}(s)$	$t_{exec}(s)$	$t_{total}(s)$	%
2.410	0,12	25,36	0,09	15,61	38,4
9.640	0,31	25,25	0,31	15,76	37,5
38.560	1,18	26,33	1,18	16,62	36,9
154.240	4,65	29,83	4,67	20,01	32,9
616.960	18,87	43,57	18,87	34,68	20,4
2.467.840	75,36	100,09	75,71	92,68	7,4

Neste trecho, há vários testes internos a malhas, sendo necessário um estudo mais detalhado de sua estrutura para a obtenção de uma redução do tempo de execução do trecho. Com o aumento da dimensão do padrão de observação, a redução do tempo de execução total do programa é cada vez menor, mostrando que este algoritmo é bastante custoso para a CPU. Uma alternativa para otimização do trecho seria a busca de algoritmos mais eficientes.

5.2 – Programação Baseada em Paralelismo de Dados

Com a identificação da subrotina de convolução de matrizes usando FFT como trecho de maior consumo de tempo de CPU desta aplicação, foi dada prioridade ao estudo desta rotina e suas descendentes, cuja otimização envolveu o emprego de programação paralela.

Alguns conceitos sobre convolução de matrizes usando FFT serão apresentados para uma melhor compreensão da aplicação, sendo estes conceitos importantes para a implementação paralela usando o modelo de paralelismo de dados, adotado na busca de uma solução otimizada do problema.

O HPF foi utilizado na paralelização da subrotina de convolução de matrizes usando FFT, sendo os dados das matrizes convenientemente distribuídos entre os processadores envolvidos através da inserção de diretivas HPF no código escrito em Fortran 90.

Nesta seção também serão apresentados os resultados de testes realizados em uma máquina paralela de memória compartilhada (máquina *polaris*) e em uma máquina paralela de memória distribuída (máquina *cluster*).

5.2.1 – Convolução de Matrizes Usando FFT

A Figura 5.3 ilustra a técnica de convolução de matrizes usando FFT [21][32].

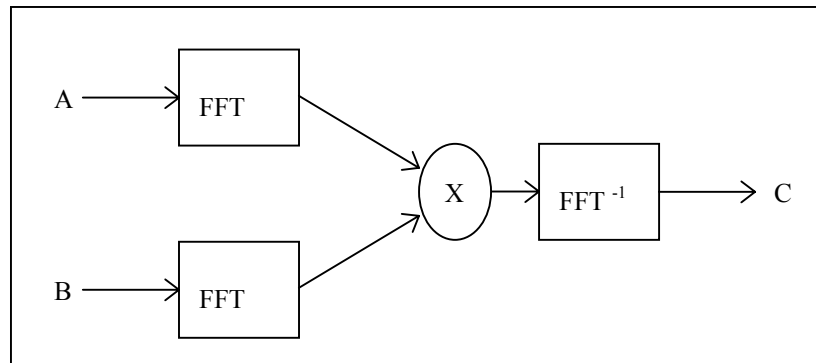


FIGURA 5.3 – Convolução de matrizes usando 2D-FFT.

A convolução de duas matrizes A e B de dimensão $M \times N$ envolve a aplicação de transformadas de Fourier bidimensionais (2D-FFT) nas duas matrizes, que por sua vez envolvem a aplicação de transformadas de Fourier unidimensionais (1D-FFT) nas linhas das matrizes e depois nas colunas, podendo ser descrita através dos seguintes passos:

- Cálculo da 2D-FFT da matriz A
- Cálculo da 2D-FFT da matriz B
- Cálculo da multiplicação $C = \text{FFT}(A) * \text{FFT}(B)$
- Cálculo da 2D-FFT inversa de C

A transformada 2D-FFT de uma matriz A ($M \times N$) é realizada através dos passos:

- Cálculo de N x 1D-FFT de dimensão M
- Cálculo da transposta da matriz. $\text{FFT}(A)$
- Cálculo de M x 1D-FFT de dimensão N

Em uma implementação HPF, é feita uma chamada a subrotina 2D-FFT para cada matriz A e B. As matrizes são distribuídas na dimensão N (por colunas) e é feita uma chamada a subrotina 1D-FFT para cada matriz. Com esta distribuição, cada processador realiza N/p transformadas de dimensão M (supondo que N seja divisível pelo número de processadores p). As matrizes transformadas são transpostas e é feita nova chamada a

subrotina 1D-FFT para cada matriz. Assim, cada processador realiza agora M/p transformadas de dimensão N. Após multiplicação das matrizes resultantes, é feita uma nova chamada a subrotina 2D-FFT para cálculo da transformada inversa.

A Figura 5.4 apresenta um exemplo da implementação HPF de um algoritmo para cálculo de uma transformada 2D-FFT [33] denominado algoritmo *transpose*. O cálculo da transformada de Fourier de uma matriz bidimensional é realizado fazendo uma chamada a subrotina *rowfft* que calcula a transformada de Fourier para cada linha da matriz. Após a transposição da matriz transformada é feita uma nova chamada a subrotina *rowfft*, que calcula a transformada de Fourier para cada coluna da matriz. No cálculo das transformadas não há dependência entre os elementos distribuídos, que podem ser processados em paralelo. A diretiva *PROCESSORS* indica a utilização de um arranjo com 8 processadores e a diretiva *DISTRIBUTE* indica que a matriz é distribuída em blocos de linhas entre os processadores. Esta distribuição permite que a execução da subrotina *rowfft* não cause qualquer comunicação entre processadores, porém a utilização da função *transpose* acarreta comunicação entre todos os processadores envolvidos.

```
!HPF$ processors pr(8)
      complex A(8, 8)
!HPF$ distribute A(BLOCK,*)
      do i = 1, 100
          call read(A)
          call rowfft(8, A)
          A = transpose(A)
          call rowfft(8, A)
          call write(A)
      end do
```

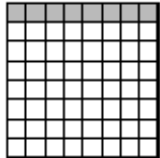
A : 

FIGURA 5.4 – Implementação HPF de 2D-FFT.
FONTE: [33].

5.2.2 – Execução em uma Máquina Paralela de Memória Compartilhada

O Apêndice A apresenta um trecho da subrotina *convlv* utilizada na implementação paralela baseada em paralelismo de dados.

Os testes foram realizados inicialmente na máquina de testes *polaris*, sendo o programa monitorado para obtenção dos valores estatísticos dos tempos de execução.

A Tabela 5.5 mostra os tempos medidos nos trechos críticos para a versão seqüencial em Fortran 90 e a versão paralela em HPF, rodando com 1, 2 e 4 processadores.

TABELA 5.5 – Comparação dos Tempos da Execução Sequencial e Paralela Usando HPF.

Sub-trecho	t _{exec} (s) F90	t _{exec} (s) HPF		
		1p	2p	4p
Loop colunas impares	0,221	0,210	0,110	0,055
Subrotina convlv	3,549	4,010	7,400	14,670
Loop colunas pares	0,216	0,210	0,110	0,053
Subrotina convlv	3,628	4,060	7,370	14,760
Tempo total	15,440	17,430	33,370	65,950

Na execução do programa paralelo utilizando HPF, nos trechos de cálculo dos componentes das colunas ímpares e pares da matriz (linhas 1 e 3 da tabela) percebe-se o ganho obtido com a distribuição dos dados entre 2 ou 4 processadores. Em contrapartida, no trecho da subrotina *convlv* houve um acréscimo no tempo de execução devido ao custo da comunicação entre processadores decorrente da utilização das funções *reshape* e *transpose*, funções implícitas do Fortran 90.

Para uma análise detalhada do problema, a subrotina *convlv* foi isolada e o tempo de execução foi dividido em dois componentes denominados t_{comp} e t_{comm} . Foram monitorados o tempo gasto nos cálculos das transformadas de Fourier e na multiplicação das matrizes (tempo de cálculo ou de computação - t_{comp}) e o tempo gasto na transposição das matrizes (tempo da função *transpose* ou de comunicação - t_{comm} ,

devido a esta operação ser a responsável pela comunicação gerada nesta implementação). Foram utilizados arquivos de dimensão 256x256, 512x512 e 1024x1024, sendo obtidos os tempos mostrados na Tabela 5.6, onde N Proc é o número de processadores.

TABELA 5.6 – Medidas dos Tempos da Rotina de Convolução.

DIMENSÃO	256x256		512x512		1024x1024	
	t _{comp1}	t _{comm1}	t _{comp2}	t _{comm2}	t _{comp3}	t _{comm3}
N Proc						
1	0,336	0,047	1,398	0,242	5,831	0,821
2	0,169	0,078	0,743	0,452	3,169	1,783
4	0,081	0,366	0,419	0,611	2,846	7,476

Na Figura 5.5 são mostrados somente os tempos de computação e comunicação para os vetores de dimensão 256x256 (t_{comp1} e t_{comm1}) e 512x512 (t_{comp2} e t_{comm2}), por permitirem melhor visualização.

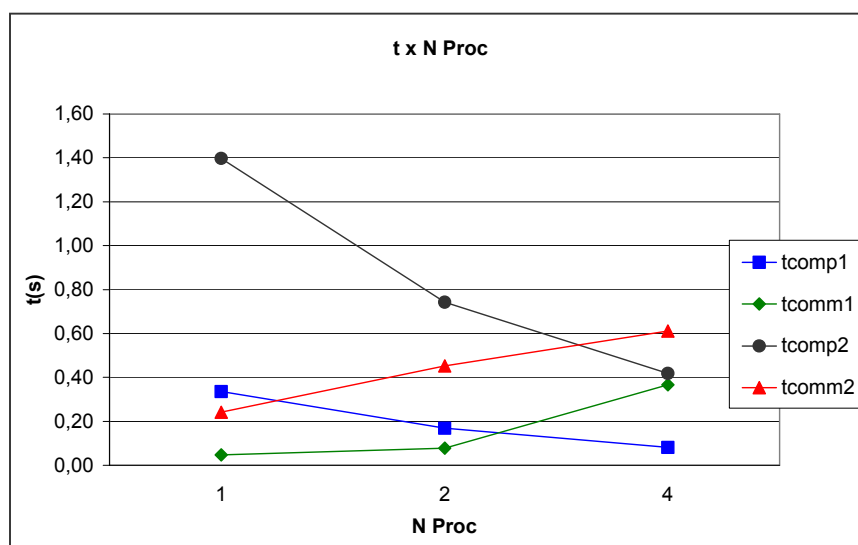


FIGURA 5.5 – Tempos de computação e comunicação em máquina de memória compartilhada.

Analisando as medidas de tempos de execução, pode-se verificar o ganho obtido no tempo de computação, por exemplo, com a distribuição dos dados entre 4

processadores. Para a dimensão de interesse desta aplicação (512x512), verifica-se que o tempo da execução paralela é 3,33 vezes menor que o da execução seqüencial, o que pode ser visto na coluna 4 da Tabela 5.6.

Entretanto, o tempo de comunicação medido durante a execução do programa com a utilização de 4 processadores foi maior que o tempo de computação, implicando na necessidade de se estudar outras técnicas mais eficientes para cálculo de 2D-FFT.

5.2.3 – Execução em uma Máquina Paralela de Memória Distribuída

A Tabela 5.7 apresenta os tempos de execução da subrotina *convlv* obtidos das medidas realizadas na máquina *cluster*, para vetores de dimensões 256x256, 512x512 e 1024x1024, utilizando o número de processadores disponíveis.

TABELA 5.7 – Tempo Total da Rotina de Convolução Usando HPF.

DIMENSÃO	256x256	512x512	1024x1024
N Proc	t_{conv1}	t_{conv2}	t_{conv3}
1	0,097	0,403	1,672
2	0,182	0,628	2,597
4	0,144	0,493	1,757
8	0,093	0,336	1,288
16	0,078	0,248	1,044

Na Figura 5.6 são mostrados os tempos de execução da subrotina *convlv*, apresentados na tabela anterior. Neste gráfico, pode-se verificar que a implementação paralela da rotina *convlv* só passa a ser vantajosa quando são utilizados 8 ou mais processadores, devido ao custo da comunicação gerada no cálculo da transposição de matrizes. Os testes realizados foram limitados ao número máximo de processadores da arquitetura utilizada, ou seja, 16 processadores.

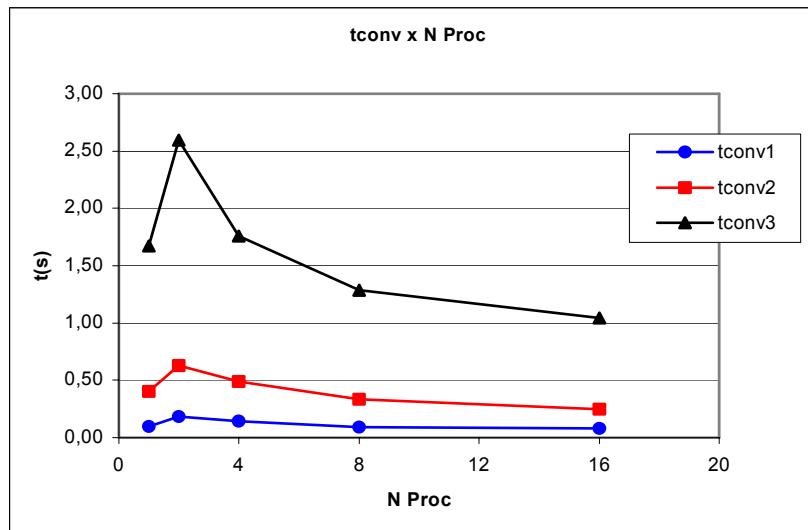


FIGURA 5.6 – Tempo da rotina de convolução de matrizes.

Para uma análise mais detalhada do problema, a Tabela 5.8 apresenta os tempos de execução divididos em dois componentes, o tempo de computação e tempo de comunicação, para visualização do tempo gasto com a comunicação entre os processadores envolvidos.

TABELA 5.8 – Tempos Parciais da Rotina de Convolução Usando HPF.

DIMENSÃO	256*256		512*512		1024*1024	
	t_{comp1}	t_{comm1}	t_{comp2}	t_{comm2}	t_{comp3}	t_{comm3}
N Proc						
1	0,089	0,008	0,363	0,040	1,497	0,175
2	0,043	0,139	0,176	0,452	0,745	1,852
4	0,019	0,125	0,095	0,398	0,381	1,376
8	0,011	0,082	0,042	0,294	0,184	1,104
16	0,005	0,073	0,022	0,226	0,091	0,953

Na Figura 5.7 são mostrados somente os tempos de computação e comunicação para os vetores de dimensão 512x512 (t_{comp2} e t_{comm2}) e 1024x1024 (t_{comp3} e t_{comm3}), por permitirem melhor visualização.

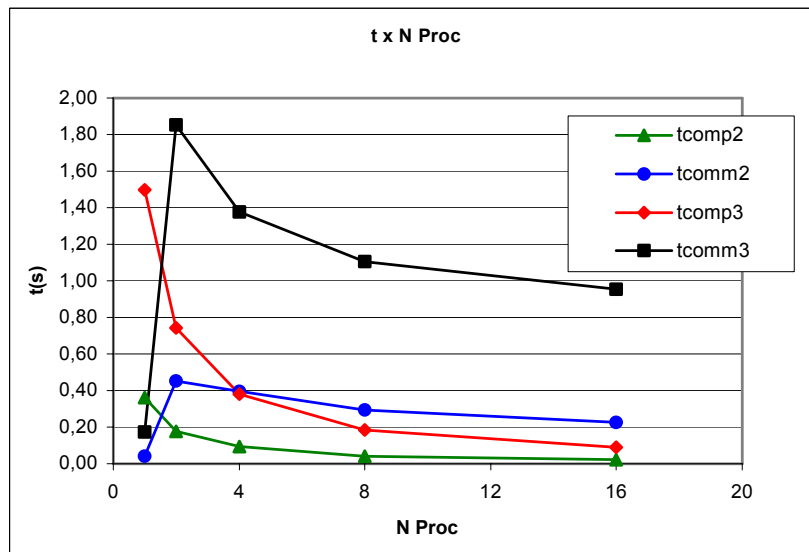


FIGURA 5.7 – Tempos de computação e comunicação em máquina de memória distribuída.

Analisando o gráfico, verifica-se o ganho obtido no tempo de computação com a distribuição dos dados entre os processadores disponíveis. Sendo as matrizes distribuídas em blocos de colunas entre os processadores, a comunicação gerada na paralelização é devida unicamente ao tempo gasto na transposição das matrizes, e apresenta um alto custo, sendo a paralelização vantajosa somente ao se utilizar 8 ou 16 processadores, máximo permitido pela arquitetura utilizada. Os resultados obtidos nos testes indicaram a necessidade de se estudar outras técnicas mais eficientes para cálculo da convolução de matrizes usando 2D-FFT.

A próxima seção descreve uma implementação mista combinando paralelismo de dados e tarefas, onde funções da biblioteca MPI são usadas para distribuição de tarefas em um programa paralelo usando HPF.

5.3 – Programação Baseada em Paralelismo de Dados e Tarefas

A implementação paralela do algoritmo para convolução de matrizes usando FFT baseada no paralelismo de dados apresentou resultados que motivaram a pesquisa de

métodos alternativos de paralelização mais eficientes em tempos de processamento. A implementação HPF apresentou um alto custo de comunicação, devido à utilização da função *transpose*, que acarreta a comunicação entre todos os processadores envolvidos.

A operação realizada pela função *transpose* pode ser feita através de troca de mensagens com a utilização da biblioteca MPI [33][34]. As funções *MPI_Send* e *MPI_Recv* podem ser utilizadas na transposição de matrizes, conforme algoritmo apresentado na Figura 5.8, denominado algoritmo *pipeline*.

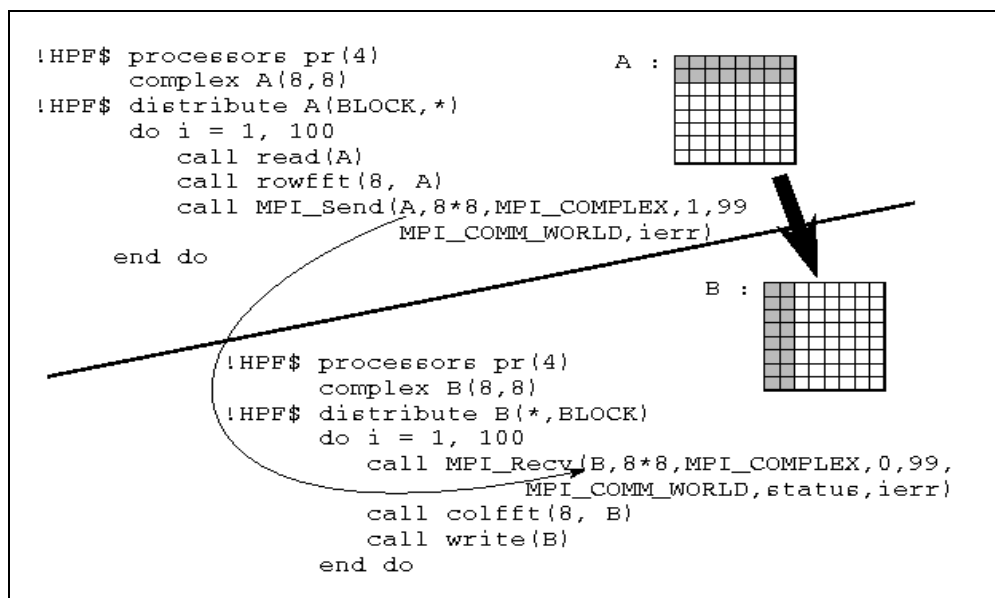


FIGURA 5.8 – Implementação HPF/MPI de 2D-FFT.
FONTE: [33].

A utilização da biblioteca MPI em um programa HPF permite a combinação de dois padrões de programação paralela, onde o paralelismo de tarefas é realizado em um ambiente baseado em paralelismo de dados. Neste exemplo de uma implementação mista HPF/MPI da 2D-FFT, uma tarefa é realizada com a distribuição da matriz por linhas entre um arranjo de processadores, é feita uma chamada a subrotina *rowfft* (que calcula a 1D-FFT para cada linha da matriz) e uma chamada a função *MPI_Send*, que envia o conteúdo da matriz em blocos de linhas. A segunda tarefa implementa a

transposição da matriz através da função *MPI_Recv*, que recebe o conteúdo da matriz em blocos de colunas, e é feita uma chamada a subrotina *colfft*, que calcula a 1D-FFT para cada coluna da matriz.

O Apêndice B apresenta um trecho da subrotina de convolução de matrizes utilizada na implementação baseada em paralelismo de dados e tarefas.

A Tabela 5.9 apresenta os tempos medidos na máquina *cluster*, para matrizes de dimensão 512x512. O tempo t_{hpf} é o tempo da subrotina implementada em HPF e o tempo $t_{\text{hpf/mpi}}$ é o tempo da subrotina implementada em HPF/MPI.

TABELA 5.9 – Comparativo entre os Tempos das Rotinas HPF e HPF/MPI

N Proc	t_{hpf}	$t_{\text{hpf/mpi}}$
1	0,339	-
2	0,367	0,438
4	0,245	0,376
6	0,237	0,338
8	0,199	0,328
10	0,181	0,309
12	0,144	0,301
14	0,124	0,300
16	0,119	0,298

Na Figura 5.9 são apresentados os tempos totais das rotinas HPF e HPF/MPI. A comparação entre os valores medidos mostra que a implementação HPF pura apresentou um menor tempo de execução, apesar do custo computacional devido à transposição de matrizes.

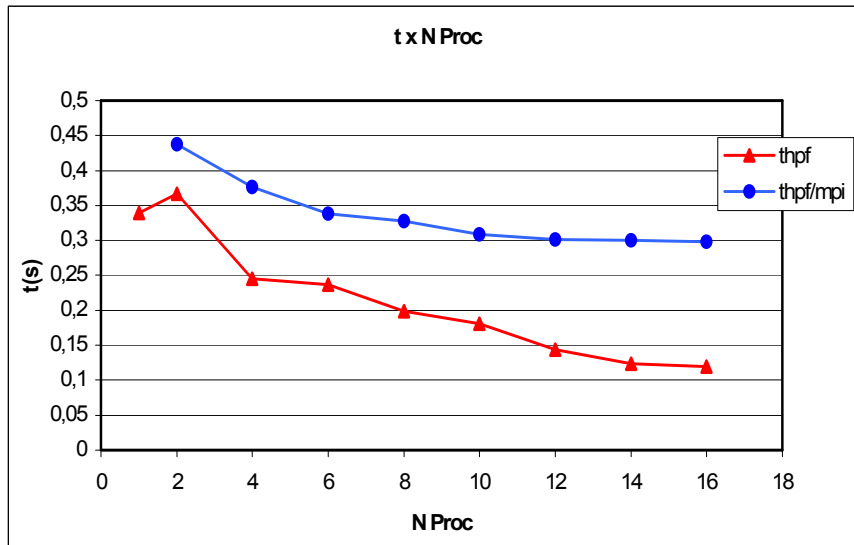


FIGURA 5.9 – Comparativo dos tempos de execução das implementações HPF e HPF/MPI.

Os resultados indicam um aumento de desempenho com a distribuição de tarefas e dados entre processadores, porém esta implementação apresentou resultados menos eficientes que a implementação utilizando somente distribuição de dados.

A aplicação exige um estudo mais detalhado de sua estrutura e os testes realizados indicam a necessidade de se explorar novas técnicas para a solução dos problemas encontrados na produção de mapas da RCFM.

CAPÍTULO 6

CONCLUSÕES

Este trabalho apresenta uma estratégia de otimização de desempenho para o software de produção de mapas da Radiação Cósmica de Fundo em Microondas (RCFM), cuja implementação é dividida em duas etapas, sendo a primeira delas tema deste estudo.

A otimização do software para cálculo da matriz de covariância inversa apresentou resultados que motivaram o estudo e a aplicação de programação paralela como uma solução para as dificuldades enfrentadas no processo de análise de dados da RCFM, dentre elas a manipulação de conjuntos de dados com um imenso número de observações e a incapacidade dos algoritmos utilizados atualmente em lidar com as técnicas de análise de dados em uma escala de tempo adequada.

A instrumentação inserida no programa permitiu a identificação de alguns trechos críticos em consumo de tempo de CPU e possíveis candidatos à aplicação de técnicas de otimização. A utilização do Fortran 90 implicou em uma reestruturação do código para adequação dos dados de entrada e técnicas clássicas de otimização foram empregadas nos trechos críticos. Os resultados apresentados no capítulo 5 mostraram a sua eficiência. A otimização do programa seqüencial para cálculo da matriz de covariância de ruído para mapas de 2.10^3 pixels apresentou uma redução de 32% no tempo de execução.

Em alguns trechos do programa, a existência de diversos condicionais internos a malhas impossibilitou a obtenção de melhor desempenho, indicando a necessidade de um estudo mais detalhado sobre a possibilidade de se realizar uma reestruturação do código.

A análise das medidas de tempo de execução das duas versões seqüenciais do programa forneceu subsídios para o estudo e a aplicação de programação paralela, visando à minimização do tempo de processamento nos trechos críticos. Rotinas de convolução de

matrizes usando FFT foram identificadas como maiores consumidoras de tempo de CPU e foram analisadas mais detalhadamente.

A implementação paralela baseada em paralelismo de dados apresentou um alto custo de comunicação, principalmente devido à transposição de matrizes, uma operação ineficiente quando se utiliza o HPF. Uma máquina paralela de memória compartilhada foi utilizada nos testes iniciais e possibilitou a indicação da forma mais eficiente de distribuição dos dados entre os processadores. Primeiro, os comandos HPF para distribuição dos vetores foram inseridos na rotina principal. Os tempos de execução da rotina de convolução de matrizes foram medidos usando um mapeamento descritivo, onde é passada ao compilador a informação de que os dados já estavam distribuídos na rotina chamadora. Com a utilização de um mapeamento prescritivo, em que os comandos HPF para distribuição dos vetores são inseridos na própria rotina de convolução de matrizes, foram obtidos os resultados apresentados no capítulo anterior, que são ligeiramente melhores que os do primeiro teste.

A implementação HPF mais eficiente foi testada em uma máquina paralela de memória distribuída, cujos resultados apresentados no capítulo 5 mostram a relação entre o ganho obtido com a paralelização e o custo da comunicação gerada, sendo a paralelização vantajosa somente ao se utilizar oito ou mais nós da máquina, limitados aos 16 nós da arquitetura utilizada. Os resultados indicaram a necessidade de se estudar métodos mais eficientes para redução do tempo de processamento do algoritmo de convolução de matrizes usando FFT.

O estudo de uma implementação paralela mista combinando paralelismo de dados e tarefas foi proposto com o objetivo de proporcionar uma comunicação mais eficiente entre os nós no cálculo da matriz transposta. Contudo, o uso da biblioteca MPI para melhorar o desempenho do HPF não apresentou os resultados esperados, sendo a implementação HPF pura a que apresentou menores tempos de execução. Uma proposta para extensão deste trabalho é o estudo de uma implementação MPI pura, onde

paralelismo de tarefas seria utilizado na busca de maior desempenho da implementação paralela.

O maior problema encontrado neste algoritmo clássico de produção de mapas, implementado na arquitetura de memória distribuída, é a alta comunicação entre nós, sendo indicado a realização de estudos mais avançados para se tentar otimizar os gargalos de comunicação.

A maior contribuição deste estudo foi demonstrar conceitos e a aplicabilidade da programação paralela na produção de mapas da RCFM numa arquitetura escalável, uma vez que os requisitos computacionais para a resolução do problema completo requerem uma arquitetura com grande número de processadores e memória.

Uma proposta para extensão deste estudo é trabalhar com uma maior granularidade, ou seja, minimizar os gargalos de comunicação entre os processadores envolvidos na aplicação. Uma estratégia seria dividir o conjunto de dados entre os nós, cada um deles operando com seu subconjunto dos dados, sendo alguns elementos sobrepostos. Isto pode ser feito inserindo comandos MPI no código em Fortran 90. Cada processador faria o cálculo da convolução de matrizes sobre seu subconjunto de dados, executando o algoritmo FFT de forma seqüencial. Esta técnica será utilizada na missão Planck Surveyor [35]. Esta solução poderia ser mais eficiente que a execução do algoritmo FFT de forma paralela, tendo a vantagem de aproveitar a localidade dos dados, reduzindo a comunicação entre processadores.

Outra possibilidade para melhoria do desempenho da aplicação é a utilização da biblioteca de domínio público *Fastest Fourier Transform in the West* (FFTW) [36], um conjunto de rotinas otimizadas para cálculo de FFT, desenvolvidas em linguagem C. Esta biblioteca também contém uma versão paralela para máquinas de memória distribuída utilizando MPI, e apresenta algumas definições para sua utilização em programas desenvolvidos em Fortran.

Novas técnicas para produção de mapas, tais como algoritmos genéticos e redes neurais, também têm sido exploradas pela comunidade científica para a solução destes problemas. Um algoritmo genético foi desenvolvido recentemente na Divisão de Astrofísica, porém não apresentou um desempenho competitivo com o método tradicional em relação ao tempo gasto na elaboração de mapas. Uma sugestão de trabalho futuro é a otimização deste algoritmo e o desenvolvimento de uma versão que possa ser utilizada em arquiteturas paralelas, visando torná-lo competitivo na análise de dados experimentais da RCFM.

REFERÊNCIAS BIBLIOGRÁFICAS

- 1 - Wright, E.L.; Hinshaw, G.; Bennett, C.L. Producing mega-pixels CMB maps from differential radiometer data. **The Astrophysical Journal Letters**. 458:L53-L55. Feb. 1996. Disponível em: <http://arxiv.org/PS_cache/astro-ph/pdf/9510/9510102.pdf>. Acesso em: Mar. 2000.
- 2 - Hu, W.; Dodelson, S. Cosmic Microwave Background anisotropies. **Annual Review of Astronomy and Astrophysics**. 40:171. 2002. Disponível em: <http://arxiv.org/PS_cache/astro-ph/pdf/0110/0110414.pdf>. Acesso em: Fev. 2002.
- 3 - Hu, W. CMB temperature and polarization anisotropy fundamentals. **Annals of Physics**. 303:203-225. 2003. Disponível em: <http://arxiv.org/PS_cache/astro-ph/pdf/0210/0210696.pdf>. Acesso em: Fev. 2003.
- 4 - Tegmark, M. How to make maps from CMB data without losing information. **The Astrophysical Journal Letters**. 480:L87-L90. May. 1997. Disponível em: <http://arxiv.org/PS_cache/astro-ph/pdf/9611/9611130.pdf>. Acesso em: Mar. 2000.
- 5 - Tegmark, M.; Oliveira-Costa, A.; Staren, J.; Meinhold, P.; Lubin, P.; Childers, J.; Figueiredo, N.; Gaier, T.; Lim, M.; Seiffert, M.; Villela, T.; Wuensche, C.A. Cosmic Microwave Background maps from the HACME experiment. **The Astrophysical Journal**. 541:535-541. Oct. 2000. Disponível em: <http://arxiv.org/PS_cache/astro-ph/pdf/9711/9711076.pdf>. Acesso em: Abr. 2001.
- 6 - Staren, J.; Meinhold, P.; Childers, J.; Lim, M.; Levy, A.; Lubin, P.; Seiffert, M.; Gaier, T.; Figueiredo, N.; Villela, T.; Wuensche, C.A.; Tegmark, M.; Oliveira-Costa, A. A spin-modulated telescope to make two-dimensional Cosmic Microwave Background maps. **The Astrophysical Journal**. 539:52-56. Aug. 2000. Disponível em: <http://arxiv.org/PS_cache/astro-ph/pdf/9912/9912212.pdf>. Acesso em: Mar. 2001.
- 7 - Meinhold, P.R.; Bersanelli, M.; Childers, J.; Figueiredo, N.; Gaier, T.C.; Halevi, D.; Lawrence, C.R.; Kangas, M.; Levy, A.; Lubin, P.M.; Malaspina, M.; Mandolesi, N.; Marvil, J.; Mejía, J.; Natoli, P.; O'Dwyer, I.; O'Neill, H.; Parendo, S.; Pina, A.; Seiffert, M.D.; Stebor, N.C.; Tello, C.; Villa, F.; Villela, T.; Wandelt, B.D.; Williams, B.; Wuensche, C.A. A map of the Cosmic Microwave Background from the BEAST experiment. **The Astrophysical Journal**, Mar. 2003. Disponível em: <http://arxiv.org/PS_cache/astro-ph/pdf/0302/0302034.pdf>. Acesso em: Mar. 2003.
- 8 - Hanany, S. A comparison of designs of off-axis gregorian telescopes for mm-wave large focal plane arrays. **The Astrophysical Journal**. Jun. 2002. Disponível em: <http://arxiv.org/PS_cache/astro-ph/pdf/0206/0206211.pdf>. Acesso em: Jun. 2002.

- 9 - Wright, E.L. Scanning and mapping strategies for CMB experiments. In: IAS CMB Data Analysis Workshop, 1996. **The Astrophysical Journal**. Nov. 1996. Disponível em: <http://arxiv.org/PS_cache/astro-ph/pdf/9612/9612006.pdf>. Acesso em: Mar. 2000.
- 10 - Tegmark, M. CMB mapping experiments: a designer's guide. **Physical Review D**. 56:4514-4529. Oct. 1997. Disponível em: <http://arxiv.org/PS_cache/astro-ph/pdf/9705/9705188.pdf>. Acesso em: Fev. 2000.
- 11 - Dupac, X.; Giard, M. How to make CMB maps from huge timelines with small computers. In: Proceedings of the "Mining the Sky", Garching, Germany, 2000. **The Astrophysical Journal**. Fev. 2001. Disponível em: <http://arxiv.org/PS_cache/astro-ph/pdf/0102/0102102.pdf>. Acesso em: Jun. 2001.
- 12 - Borrill, J. The challenge of data analysis for future CMB observations. In: Proceedings of "3K Cosmology Euroconference", Rome, Italy, 1998. **The Astrophysical Journal**. Mar. 1999. Disponível em: <http://arxiv.org/PS_cache/astro-ph/pdf/9903/9903204.pdf>. Acesso em: Fev. 2001.
- 13 - Stompor, R.; et al. Making maps of the Cosmic Microwave Background: the MAXIMA example. **Physical Review D**. 65. 022003. Jan. 2002. Disponível em: <http://arxiv.org/PS_cache/astro-ph/pdf/0106/0106451.pdf>. Acesso em: Abr. 2002.
- 14 - Dowd, K.; Severance, C.R. **High performance computing**. 2. ed. Sebastopol: O'Reilly & Associates. 1998. 446 p.
- 15 - Modi, J.J. **Parallel algorithms and matrix computation**. Oxford: Clarendon Press. 1988. 260 p.
- 16 - Pfister, G.F. **In search of clusters**. 2. ed. New Jersey: Prentice-Hall. 1998. 575 p.
- 17 - Buyya, R. **High performance cluster computing: architectures and systems**. New Jersey: Prentice-Hall. 1999. v. 1, 849 p.
- 18 - Hwang, K; Briggs, F.A. **Computer architecture and parallel processing**. New York: McGraw-Hill Book. 1984. 846 p.
- 19 - Skillicorn, D.B.; Talia, D. Models and languages for parallel computing. **ACM Computing Surveys**. v. 30, n. 2, p 123-169, Jun. 1998.
- 20 - Hennessy, J.L.; Patterson, D.A. **Computer architecture: a quantitative approach**. Maddison: Morgan Kauffman. 1995. 760 p.
- 21 - Foster, I. **Designing and building parallel programs**. Reading: Addison-Wesley. 1995. 381 p.
- 22 - Ellis, T.M.R.; Philips, I. R.; Lahey, T. M. **Fortran 90 programming**. Harlow: Addison-Wesley. 1994. 825 p.
- 23 - Koelbel, C.H.; Loveman, D.B.; Schreiber, R.S.; Steele Jr., G.L.; Zosel, M.E. **The High Performance Fortran handbook**. Cambridge: The MIT Press. 1997.

- 24 - Ewing, A.K.; Hare, R.J.; Richardson, H.; Simpson, A.D. **Writing data parallel programmes with High Performance Fortran**. Disponível em: <<http://www.epcc.ed.ac.uk/epcc-tec/courses.html>>. Acesso em: Out. 1999.
- 25 - Snir, M.; Otto, S.; Huss-Lederman, S.; Walker, D.; Dongarra, J. **MPI: the complete reference**. Cambridge: The MIT Press. 1996. 336 p.
- 26 - Snir, M.; Otto, S.; Huss-Lederman, S.; Walker, D.; Dongarra, J. **MPI - the complete reference, the MPI core**. 2. ed. Cambridge: The MIT Press. 2000. v. 1.
- 27 - Pacheco, Peter. **Parallel programming with MPI**. Maddison: Morgan Kauffman. 1997. 419 p.
- 28 - Culler, D.E.; Singh, J.P.; Gupta, A. **Parallel computer architecture – a hardware/software approach**. San Francisco: Morgan Kauffman. 1025 p. 1999.
- 29 - Bader, D.A.; Moret, B.M.E.; Sanders, P. **Algorithm engineering for parallel computation**. Disponível em: <<http://hpc.eece.unm.edu/papers/Dagstuhl2002.pdf>>. Acesso em: Set. 2002.
- 30 - Press, W.H.; Teukolsky, S.A.; Vetterling, W.T.; Flannery, B.P. **Numerical recipes in Fortran – the art of parallel scientific computing**. 2. ed. Cambridge: Cambridge University Press. 1992. 963 p.
- 31 - Press, W.H.; Teukolsky, S.A.; Vetterling, W.T.; Flannery, B.P. **Numerical recipes in Fortran 90 – the art of parallel scientific computing**. 2. ed. Cambridge: Cambridge University Press. 1996. 911 p.
- 32 - Van de Velde, E.F. **Concurrent scientific computing**. New York: Springer-Verlag. 1994. 328 p.
- 33 - Foster, I.; Kohr Jr., D.R.; Krishnaiyer, R.; Chouldhary, A. **Double standards: bringing task parallelism to HPF via the Message Passing Interface**. In: SC96 Technical Papers. Disponível em: <<http://www.supercomp.org/sc96/proceedings/SC96PROC/FOSTER2/INDEX.HTM>>. Acesso em: Mar. 2000.
- 34 - Gross, T.; O’Hallaron, D.; Subhlok, J. Task parallelism in a High Performance Fortran framework. **IEEE Parallel & Distributed Technology**. v. 2, n. 3, p 16-26. Fall. 1994.
- 35 - Natoli, P.; Gasperis, G.; Gheller, C.; Vittorio, N. A map-making algorithm for the Planck Surveyor. **Astronomy and Astrophysics**. 372:346-356. Jun. 2001. Disponível em: <http://arxiv.org/PS_cache/astro-ph/pdf/0101/0101252.pdf>. Acesso em: Nov. 2001.
- 36 - Frigo, M.; Johnson, S.G. **FFTW: an adaptive software architecture for the FFT**. Disponível em: <<http://www.fftw.org/fftw-paper-icassp.pdf>>. Acesso em: Out. 2002.

APÊNDICE A

Trecho da subrotina usando paralelismo de dados

Trecho da subrotina com diretivas HPF:

```
Subroutine convlv(d1,d2,d3,M,N)
...
!HPF$ PROCESSORS nproc(NUMBER_OF_PROCESSORS())
!HPF$ TEMPLATE :: temp1(M,N) ,temp2(N,M)
!HPF$ DISTRIBUTE(*,BLOCK) ONTO nproc :: temp1 , temp2
!HPF$ ALIGN WITH temp1 :: d1, d2, d3
!HPF$ ALIGN WITH temp2 :: dt1,dt2,dt3

sign=1

!HPF$ INDEPENDENT
do i = 1,N
    call fftrow(d1(:,i),M,sign)
    call fftrow(d2(:,i),M,sign)
enddo

dt1 = transpose(d1)
dt2 = transpose(d2)

!HPF$ INDEPENDENT
do i = 1,M
    call fftrow(dt1(:,i),N,sign)
    call fftrow(dt2(:,i),N,sign)
enddo

dt3 = dt1 * dt2

! calculo da inversa
sign=-1

!HPF$ INDEPENDENT
do i = 1,M
    call fftrow(dt3(:,i),N,sign)
enddo

d3 = transpose(dt3)

!HPF$ INDEPENDENT
do i = 1,N
    call fftrow(d3(:,i),M,sign)
enddo
...
```


APÊNDICE B

Trecho da subrotina usando paralelismo de dados e tarefas

Trecho da subrotina com diretivas HPF e funções MPI:

```
Subroutine convlv_mpi(d1,d2,d3,M,N)
...
include 'mpi.h'

!HPF$ PROCESSORS np(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(*,BLOCK) ONTO np :: d1
!HPF$ DISTRIBUTE(BLOCK,*) ONTO np :: dt1
!HPF$ ALIGN (,:) WITH d1(:,) :: d2, d3
!HPF$ ALIGN (,:) WITH dt1(:,) :: dt2,dt3

sign=1

call MPI_COMM_RANK (MPI_COMM_WORLD,myrank,ierr)

if(myrank.eq.0) then      !Task 0 – d1
  !HPF$ INDEPENDENT
  do i = 1,N
    call fftrow(d1(:,i),M,sign)
  enddo
  do i = 1,N
    call MPI_SEND(d1(:,i),M,MPI_COMPLEX,1,i,MPI_COMM_WORLD,ierr)
  enddo
else
  if(myrank.eq.1) then      !Task 1
    do i = 1,N
      call MPI_RECV(dt1(:,i),M,MPI_COMPLEX,0,i,MPI_COMM_WORLD,status,ierr)
    enddo
    !HPF$ INDEPENDENT
    do i = 1,M
      call fftrow(dt1(i,:),N,sign)
    enddo
  endif
endif

if(myrank.eq.0) then      !Task 0 – d2
  !HPF$ INDEPENDENT
  do i = 1,N
    call fftrow(d2(:,i),M,sign)
  enddo
  do i = 1,N
    call MPI_SEND(d2(:,i),M,MPI_COMPLEX,1,i,MPI_COMM_WORLD,ierr)
  enddo
else
  if(myrank.eq.1) then      !Task 1
    do i = 1,N
      call MPI_RECV(dt2(:,i),M,MPI_COMPLEX,0,i,MPI_COMM_WORLD,status,ierr)
    enddo
  endif
endif
```

```

!HPF$ INDEPENDENT
do i = 1,M
    call fftrow(dt2(i,:),N,sign)
enddo
endif

if(myrank.eq.1) dt3 = dt1 * dt2      !mult. FFTs

sign=-1

if(myrank.eq.1) then                ! calculo da FFT inversa
    !HPF$ INDEPENDENT
    do i = 1,M
        call fftrow(dt3(i,:),N,sign)
    enddo
    do i = 1,M
        call MPI_SEND(dt3(i,:),N,MPI_COMPLEX,0,i,MPI_COMM_WORLD,ierr)
    enddo
else
    if(myrank.eq.0) then
        do i = 1,M
            call MPI_RECV(d3(i,:),N,MPI_COMPLEX,1,i,MPI_COMM_WORLD,status,ierr)
        enddo
        !HPF$ INDEPENDENT
        do i = 1,N
            call fftrow(d3(:,i),M,sign)
        enddo
    endif
    ...

```