Paralelização do Cálculo de Estruturas de Bandas de Semicondutores usando o High Performance Fortran

Rodrigo Daniel Malara

Dissertação apresentada ao Instituto de Física de São Carlos, da Universidade de São Paulo, para obtenção do título de Mestre em Ciências: Física Aplicada opção - Computacional

Orientador: Prof. Dr. Guilherme Matos Sipahi

São Carlos – 2005

Malara, Rodrigo Daniel

"Paralelização do cálculo de estruturas de bandas de semicondutores usando o High Performance Fortran."
Rodrigo Daniel Malara – São Carlos, 2005

Dissertação (Mestrado) – Área de Física Aplicada, opção Computacional do Instituto de Física de São Carlos da Universidade de São Paulo
2005 - Páginas: 59

Orientador: Prof. Dr. Guilherme Matos Sipahi
Programação Paralela, Clusters, Fortran.
I. Título

UNIVERSIDADE
DE SÃO PAULO
Instituto de Física de São Carlos

Caixa Postal 369
13560-970 São Carlos, SP
Av. Trabalhador São-carlense, 400,
13560-970 São Carlos, SP

Fone/Fax 16 3373 9777
www.ifsc.usp.br
wladerez@ifsc.usp.br

MEMBROS DA COMISSÃO JULGADORA DA DISSERTAÇÃO DE MESTRADO DE RODRIGO DANIEL MALARA APRESENTADA AO INSTITUTO DE FÍSICA DE SÃO CARLOS, UNIVERSIDADE DE SÃO PAULO, EM 14/01/2005.

COMISSÃO JULGADORA:

Prof. Dr. Guilherme Matos Sipahi (Orientador e Presidente) – IFSC/USP

Prof. Dr. Marcio Lobo Netto – EP/USP

Dr. Marcon Soares Barbosa – IFSC/USP

Agradecimentos

Ao prof. Dr. Guilherme Matos Sipahi, pela confiança, amizade, paciência e pela grande dedicação na orientação deste trabalho.

Aos docentes que ministraram as disciplinas que cursei.

Aos amigos Marcel, Felipe e Leandro pelo apoio e ajuda com os problemas técnicos com o *cluster* utilizado durante a elaboração deste trabalho.

Aos amigos do grupo de pesquisa pelas horas de descontração e pela amizade.

À Wladerez e à Cristiane pelo carinho e presteza sempre que precisei de sua ajuda.

Aos meus pais Maria Bernadete e Nilson e à minha sogra Maria Aparecida pelo amor e incentivo.

A Deus por permitir que este trabalho fosse realizado.

À minha esposa Silvana pelo seu amor, compreensão e pelo apoio incondicional. Ao meu filho Matheus por abrir mão dos finais de semana em que fomos privados do nosso convívio. A minha motivação vem de vocês.

Resumo

O uso de sistemas multiprocessados para a resolução de problemas que demandam um grande poder computacional tem se tornado cada vez mais comum. Porém a conversão de programas seqüenciais para programas concorrentes ainda não é uma tarefa trivial. Dentre os fatores que tornam esta tarefa difícil, destacamos a inexistência de um paradigma único e consolidado para a construção de sistemas computacionais paralelos e a existência de várias plataformas de programação para o desenvolvimento de programas concorrentes.

Nos dias atuais ainda é impossível isentar o programador da especificação de como o problema será particionado entre os vários processadores. Para que o programa paralelo seja eficiente, o programador deve conhecer a fundo aspectos que norteiam a construção do hardware computacional paralelo, aspectos inerentes à arquitetura onde o software será executado e à plataforma de programação concorrente escolhida. Isto ainda não pode ser mudado.

O ganho que podemos obter é na implementação do software paralelo. Esta tarefa pode ser trabalhosa e demandar muito tempo para a depuração, pois as plataformas de programação não possibilitam que o programador abstraia dos elementos de hardware. Tem havido um grande esforço na criação de ferramentas que otimizem esta tarefa, permitindo que o programador se expresse mais fácil e sucintamente quanto à paralelização do programa.

O presente trabalho se baseia na avaliação dos aspectos ligados à implementação de software concorrente utilizando uma plataforma de portabilidade chamada *High Performance Fortran*, aplicado a um problema específico da física: o cálculo da estrutura de bandas de heteroestruturas semicondutoras.

O resultado da utilização desta plataforma foi positivo. Obtivemos um ganho de performance superior ao esperado e verificamos que o compilador pode ser ainda mais eficiente do que o próprio programador na paralelização de um programa. O custo inicial de desenvolvimento não foi muito alto, e pode ser diluído entre os futuros projetos que venham a utilizar deste conhecimento pois após a fase de aprendizado, a paralelização de programas se torna rápida e prática.

A plataforma de paralelização escolhida não permite a paralelização de todos os tipos de problemas, apenas daqueles que seguem o paradigma de paralelismo por dados, que representam uma parcela considerável dos problemas típicos da Física.

Abstract

The employment of multiprocessor systems to solve problems that demand a great computational power have become more and more usual. Besides, the conversion of sequential programs to concurrent ones isn't trivial yet. Among the factors that makes this task difficult, we highlight the nonexistence of a unique and consolidated paradigm for the parallel computer systems building and the existence of various programming platforms for concurrent programs development.

Nowadays it is still impossible to exempt the programmer of the specification about how the problem will be partitioned among the various processors. In order to have an efficient parallel program the programmer have to deeply know subjects that heads the parallel hardware systems building, the inherent architecture where the software will run and the chosen concurrent programming platform. This cannot be changed yet.

The gain is supposed to be on the parallel software implementation. This task can be very hard and consume so much time on debugging it, because the programming platforms do not allow the programmer to abstract from the hardware elements. It has been a great effort in the development of tools that optimize this task, allowing the programmer to work easily and briefly express himself concerning the software parallelization.

The present work is based on the evaluation of aspects linked to the concurrent software implementation using a portability platform called *High Performance Fortran*, applied to a physics specific problem: the calculus of semiconductor heterostructures' valence band structure.

The result of the use of this platform use was positive. We obtained a performance gain superior than we expected and we could assert that the compiler is able to be more effective than the programmer on the paralelization of a program. The initial development cost wasn't so high and it can be diluted between the next projects that would use the acquired knowledge, because after the learning phase, the programs parallelization task becomes quick and practical.

The chosen parallelization platform does not allow the parallelization of all kinds of problems, but just the ones that follow the data parallelism paradigm that represents a considerable parcel of tipical Physics problems.

Conteúdo

Lista de Figuras

Lista de Tabelas

Introdução

O uso de sistemas de computação concorrente tem se tornado cada vez mais comum tanto em instituições de ensino e pesquisa como em instituições privadas e grandes organizações. Dentre as vantagens, podemos citar ganhos em performance, escalabilidade e em confiabilidade, uma vez que a falha de uma unidade de processamento não inviabiliza o funcionamento das outras.

O uso deste recurso computacional agregado não costuma ser trivial e passível de ser feito por qualquer programador. Existem vários tipos de sistemas concorrentes diferentes tanto do ponto de vista da sua arquitetura computacional, quanto da sua efetiva utilização. Atualmente, é imprescindível a presença de mão-de-obra qualificada e especificamente treinada para a instalação, configuração e uso de sistemas paralelos.

No caso dos sistemas computacionais uniprocessados, como por exemplo os computadores pessoais, é possível que os programadores abstraiam dos detalhes de implementação da máquina e criem seus programas com base na ubíqua plataforma estabelecida pela "Arquitetura de Von Neumann"[1]. No caso dos sistema de computação paralelos, a ciência da computação tem avançado no mesmo sentido, mas ainda há um grande esforço a ser feito. A ausência de um padrão firmemente estabelecido para arquiteturas concorrentes permite a existência de diversos ambientes e ferramentas para auxiliar o desenvolvedor em utilizar todos os recursos que um sistema paralelo oferece.

Muitas destas ferramentas são de difícil aprendizado e a depuração dos programas criados com elas é trabalhosa. A comunicação entre os processadores fica totalmente exposta ao programador que, além de elaborar o algorítmo e o programa para sua aplicação, tem que verificar se o tráfego de dados está ocorrendo corretamente e se a seqüência de processamento está sendo respeitada sob a penalidade de produzir um programa ineficiente, que não funciona ou que não é portável para outras arquiteturas computacionais.

A existência de plataformas de programação que isente o programador deste tipo de preocupação é desejada para reduzir o tempo de aprendizado, programação e depuração, permitindo que o programador se dedique mais em resolver seu problema do que em lidar com as dificuldades inerentes ao processamento concorrente. A paralelização do

código ficaria a cargo do sistema de compilação, que deveria identificar os trechos paralelizáveis e providenciar que os dados sejam trocados e sincronizados entre processadores. A existência de tal sistema de compilação ainda viabilizaria a portabilidade do código fonte entre arquiteturas computacionais paralelas diferentes, sendo necessário que exista um compilador para a arquitetura destino e que a aplicação seja apropriadamente recompilada.

Atualmente não se encontra um sistema de compilação que possa realizar a paralelização de programas sozinho. Ainda que minimamente, o programador deve instruir como o trabalho precisa ser dividido entre os elementos processantes. O ideal é que este tipo de intervenção se mantenha em um patamar mínimo, evitando o envolvimento excessivo do programador no processo de paralelização do código.

Além da facilidade de implementação, um dos fatores chave no uso de sistemas computacionais concorrentes concerne à performance. O uso de tal sistema computacional não deve prejudicar a performance da aplicação devido ao uso de uma plataforma de programação onde o programador é o responsável por todas as operações inter-processadores, ou seja, a plataforma de portabilidade [2] não deve produzir um programa paralelo com uma performance menor do que o gerado "manualmente por um programador.

Este trabalho tem o intuito de pesquisar, aprender, usar e finalmente avaliar sistemas de compilação de programas paralelos baseados no padrão *High Performance Fortran* (HPF) no que tange à facilidade de aprendizado, implementação e finalmente sua performance.

1 O Problema

1.1 Introdução

A onipresença da eletrônica em nossa vida mostra a importância que a indústria eletrônica têm na sociedade moderna. Para suprir esta constante demanda por novos produtos existe uma busca pelo projeto e confecção de novos dispositivos eletrônicos capazes de executar novas tarefas ou de proporcionar uma melhoria de desempenho em relação aos dispositivos já existentes. Com o recente desenvolvimento da computação, cada vez mais busca-se na simulação das propriedades dos materiais estudados, elementos que possibilitem novos usos ou características necessárias ao funcionamento do dispositivo. Como exemplo desses usos podemos citar: a possível utilização de semicondutores magnéticos diluídos (SMD) em dispositivos spintrônicos, a utilização de nitretos semicondutores na confecção de dispositivos emissores de luz (LEDs), o uso desses mesmos nitretos em dispositivos eletrônicos funcionando a altas temperaturas, etc.

Todos os sistemas citados acima foram objeto de pesquisa recente do Laboratório de Física Computacional (LFC), coordenado pelo Prof. Guilherme Matos Sipahi e vinculado ao Grupo de Instrumentação e Informática (GII) do Instituto de Física de São Carlos. Este projeto insere-se na pesquisa do grupo supra-citado por desenvolver novos métodos computacionais para a simulação das propriedades eletrônicas e óticas de sistemas semicondutores de baixa dimensionalidade (heteroestruturas, pontos e fios quânticos).

O programa de simulação estudado foi implementado durante a elaboração da tese de doutoramento do Prof. Guilherme Matos Sipahi [3] e vem sendo aumentado, melhorado e modificado desde então. Ele utiliza a resolução autoconsistente das equações da Massa Efetiva e de Poisson, possibilitando o cálculo da estrutura de bandas e perfis de carga e potenciais desses sistemas, bem como a determinação de espectros teóricos de luminescência e absorção.

O método autoconsistente parte da resolução de um Hamiltoniano tentativo que

contém um termo de energia cinética e de strain (representado por uma matriz $\mathbf{k} \cdot \mathbf{p}$) e dos potenciais coulombiano, de troca-e-correlação e de descasamento de bandas da heteroestrutura. Pode-se também incluir termos que descrevam potenciais piezoelétricos e a quebra de estrutura de bandas por um campo magnético constante. Os autovalores obtidos são ordenados e a ocupação dos estados é obtida através deste ordenamento e do número de portadores presentes no sistema. Depois ds determinação dos estados ocupados é calculada a distribuição de cargas do sistema e a partir desta, usando a equação de Poisson, são calculados os novos potenciais coulombiano e de troca-e-correlação. Estes potenciais obtidos são comparados com os potenciais tentativos e, se forem iguais, o cálculo convergiu, e este é o estado de mínima energia do sistema. Se forem diferentes novos potenciais são estimados a partir dos potenciais originais e dos potenciais resultados, sendo estes os novos potenciais tentativos. O processo repete-se até a convergência dos potenciais originais e resultados.

Os estados finais, que descrevem a estrutura de bandas do sistema, os potenciais e as distribuições de carga são salvos para análise. A partir dos estados eletrônicos ocupados, levando em conta o alargamento causado pela temperatura, pode-se determinar espectros teóricos de luminescência e de absorção. Isto é feito através da convolução dos estados iniciais e finais da transição desejada, ponderada pela ocupação desses mesmos estados, incluindo nesta o alargamento devido à temperatura.

Do ponto de vista computacional, o programa recebe vários parâmetros através de um arquivo texto de entrada, executa um "loop" autoconsistente para a resolução da equação de massa efetiva multibandas no espaço recíproco e posteriormente calcula os respectivos espectros de luminescência e absorção. O processo envolve a diagonalização de várias matrizes densas de números complexos (que são executadas em paralelo por vários processadores) e várias convoluções dos estados das bandas de portadores, representadas por matrizes de duas dimensões.

De forma geral, podemos classificar os problemas que são endereçados pelo sistema em questão em dois tipos, de acordo com a demanda por processamento:

1. Problema que demanda intensivo processamento na fase de diagonalização das matrizes do Hamiltoniano. O custo computacional se concentra quase que totalmente neste trecho do programa.

2. Problema que demanda um processamento intensivo no trecho do programa que faz o cálculo da luminescência considerando transições diretas e indiretas.

1.2 O Programa

O programa existente foi primeiramente desenvolvido para ser executado seqüencialmente e possuía uma variante concorrente que utilizava o MPICH [4] como ferramenta para viabilizar a execução em paralelo. A linguagem utilizada para o desenvolvimento foi o FORTRAN 77 e o software foi posteriormente portado para o FORTRAN 90, sendo esta a linguagem utilizada atualmente.

A diagonalização é um procedimento matemático muito utilizado e que se encontra disponível em várias bibliotecas de cálculo numérico. Para esta implementação utilizamos a função `CHEEVX` da biblioteca LAPACK (Linear Algebra PACKage). O LAPACK faz uso de uma plataforma de cálculo matemático altamente otimizada chamada BLAS (*Basic Linear Algebra Subprograms*).

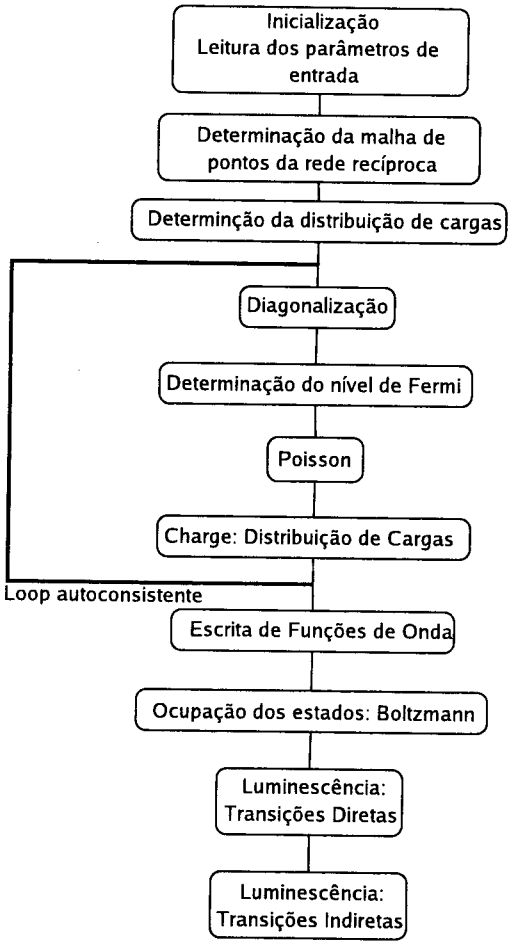

Figura 1.1: Diagrama de Blocos: Variante seqüencial

1.2.1 Oportunidade de Paralelização

Um diagrama de blocos simplificado da versão seqüencial pode ser visto na figura 1.1. Comparando este diagrama com o diagrama da versão concorrente (figura 1.2) é possível se verificar os locais onde as paralelizações foram realizadas:

- A diagonalização de matrizes para o cálculo dos Hamiltonianos
- O cálculo dos coeficientes da distribuição de cargas (equação de Poisson)
- O cálculo do espectro de luminescência para transições diretas
- O cálculo do espectro de luminescência para transições indiretas

Os trechos destacados em itálico nas tabelas 1.1 e 1.2, são responsáveis por mais de 90% do processamento dispendido para a resolução de um problema simples (na variante seqüencial). Além disso, a divisão do trabalho nos vários processadores foi um sucesso, pois o processamento pode ser dividido em grandes partes (granularidade grossa), proporcionando um grande ganho de desempenho pois a relação computação/comunicação se mantém relativamente alta, indicando que os processadores dispendem seu poder computacional efetivamente resolvendo o problema, ao invés de perderem ciclos de processamento enquanto informações são trocadas entre os processadores através da rede de comunicação.

Trecho	**Tempo (s)**	**Percentual (%)**
Inicialização	0,1	0,02
Determinação da Malha de Pontos na Rede Recíproca	0	0
Distribuição de cargas	16,27	2,55
Diagonalização	*599,54*	*93,82*
Nível de Fermi	0,24	0,04
Poisson	0,72	0,11
Distribuição de cargas	0,75	0,12
Escrever Funções de Onda	21,27	3,33
Total	**638,89**	**100**

Tabela 1.1: Custo computacional por trecho: Problema Tipo 1

1.2.2 Implementação em MPI

Na implementação da variante em MPI foi necessário utilizar rotinas de sincronização, comunicação em massa e de redução dos dados divididos entre os vários processadores.

Trecho	**Tempo (s)**	**Percentual (%)**
Inicialização	0,08	0
Determinação da Malha de Pontos na Rede Recíproca	0	0
Distribuição de cargas	5,64	0,05
Diagonalização	43,56	0,42
Nível de Fermi	1,33	0,01
Poisson	1,79	0,02
Distribuição de cargas	6,05	0,06
Escrever Funções de Onda	2,2	0,02
Ocupação dos estados: Boltzmann	0,38	0
Luminescência: TD	40,87	0,39
Luminescência: TI	*10267,49*	*99,02*
Total	**10639,39**	**100**

Tabela 1.2: Custo computacional por trecho: Problema Tipo 2

O desenvolvimento da variante concorrente usando o MPI, executado anteriormente a este trabalho, foi custoso pois vários meses foram investidos no aprendizado, na implementação e principalmente no diagnóstico dos problemas encontrados.

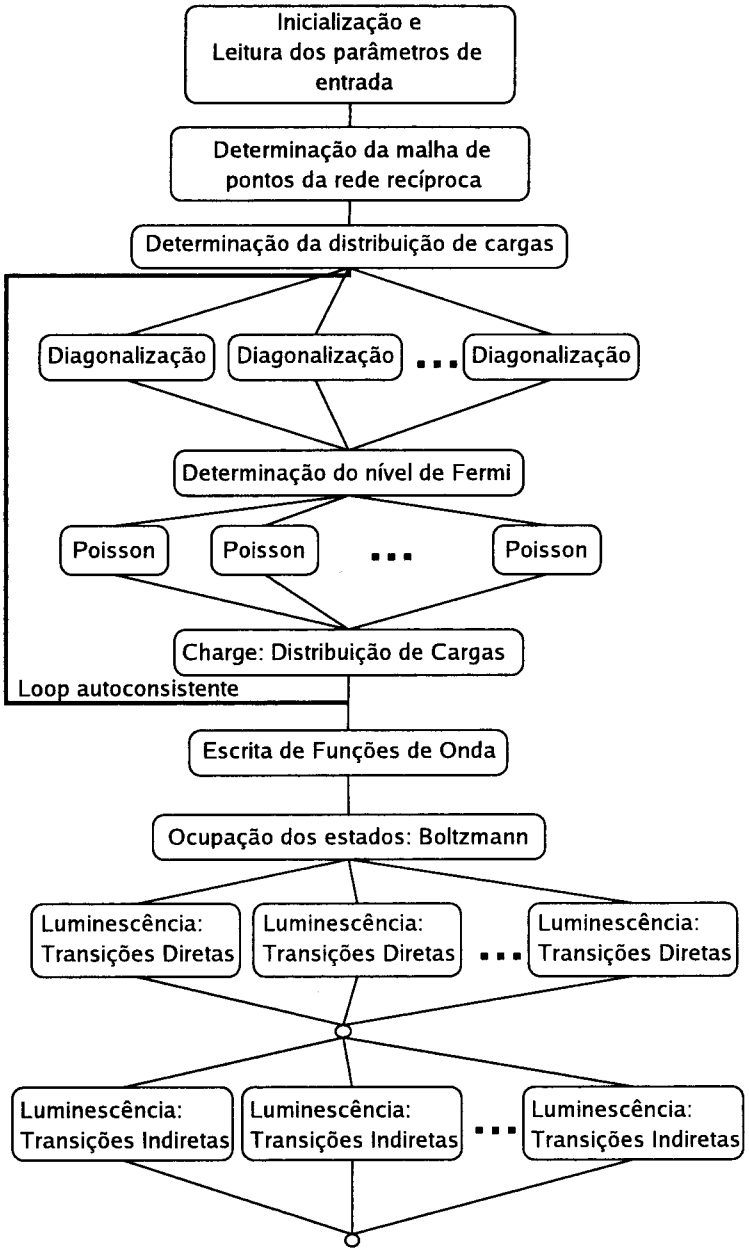

Figura 1.2: Diagrama de Blocos: Variante concorrente

2 Revisão Bibliográfica

2.1 Sistemas Paralelos

Um computador paralelo é um conjunto de elementos processantes (ou processadores) que se comunicam e cooperam para resolver grandes problemas mais rapidamente [5]. Atualmente existem várias arquiteturas diferentes para computadores paralelos. Essa diversidade gerou a necessidade de se criar taxonomias e classificações.

Dentre as taxonomias, temos a de Flynn [6] que define quatro categorias de computadores paralelos em termos dos principais fluxos (*streams*) [7].

1. SISD - *Single Instruction Single Data*: Uniprocessadores tradicionais. Sua constituição é similar à arquitetura proposta pelo matemático John Von Neumann em 1946 [8]. Atualmente, a busca por uma maior performance se dá através do paralelismo em nível de instrução aplicando-se técnicas superescalares e melhoria do pipeline [9]. Exemplos: microcomputadores padrão IBM/PC e PowerPC dentre outros.

2. SIMD - *Single Instruction Multiple Data*: Processadores matriciais e vetoriais (*matrix and vector processors*). Geralmente formados por vários processadores idênticos executando a mesma instrução concorrentemente sobre dados diferentes. Formam a plataforma natural para a resolução de problemas que envolvam o processamento de grandes estruturas de dados homogêneas, ou seja, do mesmo tipo de dados, também conhecidas como matrizes. Processadores matriciais são caracterizados por possuírem várias unidades de processamento que operam concorrentemente em vários elementos de dados, enquanto que processadores vetoriais possuem um único elemento de processamento que opera em vários elementos de dados simultâneamente [7]. Exemplos: Cray X-MP e Y-MP, IBM 3090 dentre outros.

3. MISD - *Multiple Instruction Single Data*: Embora seja natural extrapolar e se criar esta categoria de computadores paralelos, este tipo de organização não é facilmente

encontrada. Abstratamente, um sistema MISD seria um *pipeline* de unidades funcionais independentes que operam sobre um mesmo dado, sendo que a informação em processamento é passada de uma unidade para outra. Além da dificuldade de implantação deste tipo de sistema, há a ausência de uma plataforma ou paradigma de programação que permita se utilizar este tipo de arquitetura eficientemente. É possível se considerar arquiteturas sistólicas como representantes desta categoria.

4. MIMD - *Multiple Instruction Multiple Data*: É a categoria onde grande parte dos multiprocessadores atuais se encontram. Arquiteturas deste tipo se caracterizam por serem formadas por um conjunto de processadores que executam seqüências diferentes de instruções sobre conjuntos de dados distintos [9]. Exemplos: Silicon Graphics Power Challenge (até 64 processadores), IBM SP/2, NUMA-Q da Seqüent e clusters Beowulf.

2.2 Múltiplas Instruções e Múltiplos Dados

É o tipo mais comum de arquitetura multiprocessada. Devido ao grande avanço tecnológico e aos baixos valores de mercado dos microprocessadores, é uma tendência e quase uma regra, sua utilização ao invés do projeto de processadores especiais, como é o caso dos utilizados em arquiteturas vetoriais. Isto se deve à economia de escala, pois o mercado de microprocessadores movimenta um valor muito maior que o de supercomputadores, permitindo que as empresas façam pesados investimentos para satisfazer consumidores ávidos por maiores níveis de performance e também para se sobressair em meio à concorrência.

De maneira geral, existem três possíveis constituições para os sistemas paralelos que se utilizam de microprocessadores. Eles diferem principalmente quanto a arquitetura de comunicação existente:

2.2.1 Multiprocessadores de Memória Compartilhada Com Acesso Uniforme à Memória

São sistemas compostos por vários microprocessadores que se comunicam através de um Espaço de Endereçamento Compartilhado e são interconectados através de um barramento, uma chave *crossbar* [10] ou outro meio de comunicação de alto desempenho. Também são chamados de SMPs ou *Symmetric Multi Processors* (Multiprocessadores Simétricos).

A comunicação entre os processadores é alcançada através da carga e armazenamento (*loads and stores*) de dados no espaço de endereçamento compartilhado (memória principal) [10]. Isto ocorre de forma que qualquer dado que precise ser transferido de um processador para outro deve ser armazenado (*stored*) na memória compartilhada pelo processador 1 e carregado (*loaded*) a partir da mesma pelo processador 2.

Neste tipo de constituição, existem potenciais problemas de coerência da memória compartilhada, assim como das memórias *cache* de cada microprocessador, este último resolvido através de dispositivos de *hardware* especiais.

Cada processador do sistema experimenta um mesmo tempo de acesso e latência ao acessar qualquer posição da memória compartilhada. Este tipo de sistema possui um número máximo de processadores que é limitado não somente pelas características físicas do *hardware* de comunicação (ex: número de soquetes disponíveis), mas também por outras propriedades intrínsecas do sistema de gerenciamento e coerência da memória compartilhada e pela perda de performance ocasionada pela adição de mais um processador num sistema onde o duto de comunicação (ou barramento) possui uma capacidade limitada de transferência de dados.

2.2.2 Multiprocessadores de Memória Compartilhada Com Acesso Não-Uniforme à Memória

São chamados de NUMA ou *Non-Uniform Memory Access* (Acesso não-Uniforme à Memória). Sistemas que se encaixam nesta categoria trabalham de maneira similar a sistemas SMP. Cada nó do sistema possui, porém, sua própria memória local que pode ser acessada por todos outros processadores do sistema. O custo de acesso à memória local é menor que o custo de acesso a dados presentes na memória local de outros processadores, sendo o custo estimado em termos de tempo de acesso e latência. O acesso à memória não-local é feita através de um controlador de memória remota [10].

Dessa forma, o tráfego gerado no meio de comunicação é diminuído, permitindo que mais processadores sejam adicionados ao sistema, tornando o NUMA uma opção intermediária entre sistemas SMP e *clusters*. Em alguns casos a arquitetura é totalmente exposta ao programador, que é responsável por evitar acessos inúteis à memória não-local.

Sistemas de arquitetura NUMA herdam um problema existente nos sistemas de arquitetura SMP: a coerência da memória principal e da memória *cache*. Um sistema é

classificado como ncNUMA (*no-caching* NUMA) quando não apresenta coerência de *cache* e ccNUMA (*cache coherent* NUMA) em caso contrário [11].

Em um sistema ccNUMA quando um processador inicia um acesso à memória, se a posição requerida não está na *cache* deste processador, uma busca na memória é realizada. Se a linha requerida está na porção local da memória principal, ela é obtida por meio do barramento local. Se a posição está em uma memória remota, é feita uma requisição de busca à memória remota através do sistema de conexão que interliga os processadores. O dado é então entregue ao barramento local, que é então entregue à *cache* e ao processador requisitante. O método mais popular de construção de sistemas ccNUMA é baseado no conceito de diretório (*directory-based multiprocessor*) [12], que funciona como uma espécie de banco de dados que indica a localização das várias porções de memória, assim como o status das memórias *cache* de cada processador.

2.2.3 Multiprocessadores de Memória Distribuída

São sistemas compostos por vários computadores completos, capazes de operar de forma totalmente desacoplada que se comunicam através de operações de entrada e saída [10]. Os dispositivos normalmente utilizados são interfaces de rede de alto desempenho.

A princípio, não existe uma área de memória compartilhada. Cada computador possui a sua própria memória principal, constituindo assim, uma hierarquia de memória distribuída. Caso um processador necessite de uma informação que está na memória principal de outro processador, esta informação deve ser explicitamente transferida entre os processadores utilizando-se de um mecanismo de "passagem de mensagens".

Nesta categoria se encontram os *clusters*. Este tipo de constituição também é chamada de multicomputador, e geralmente são constituídos por microcomputadores comuns (chamados de nós) interligados através de uma rede de alto desempenho.

Mesmo utilizando um meio de comunicação relativamente eficiente, este tipo de organização é classificada como "Fracamente Acoplada", devido à distância entre processadores, largura da banda de comunicação e latência.

Caracteriza-se por largura de banda de comunicação a medida da quantidade de bytes que pode ser enviada do transmissor ao receptor em um determinado espaço de tempo. A latência de comunicação mede o tempo que uma transmissão de dados elementar leva desde a sua requisição até a seu término [10].

Assim como as outras organizações descritas acima, eles são constituídos para formarem um recurso computacional único com poder computacional que pode chegar à um patamar impossível de ser atingido por qualquer supercomputador individual. Atualmente os *clusters* são utilizados por sistemas gerenciadores de bancos de dados, por servidores de conteúdo para a Internet, em centros de pesquisa e desenvolvimento, dentre outros.

Este tipo de organização paralela surgiu como uma opção robusta e promissora, devido a suas características inerentes como alta disponibilidade, escalabilidade e boa relação custo/benefício. Por se tratarem de computadores independentes, os *clusters* conseguem oferecer uma alta disponibilidade. Caso ocorra algum problema em qualquer um dos seus nós, a operação do sistema não precisa ser interrompida. É possível que o sistema continue a operar mesmo com um prejuízo na performance. Eles também são escaláveis bastando-se adicionar mais nós ao agregado computacional caso um maior poder de processamento seja necessário. Isto não é sempre possível nos sistemas de memória compartilhada. Os equipamentos produzidos segundo este paradigma não podem receber novos processadores indefinidamente por restrições físicas e lógicas. O aumento do número de processadores pode ser limitado pela quantidade de soquetes nas placas do sistema ou devido à limitação de banda no barramento de comunicação entre a memória principal e os processadores.

Devido ao uso de partes produzidas em grande escala pela indústria de equipamentos eletrônicos, os *clusters* apresentam uma boa relação custo-benefício quando comparados com computadores de grande porte que oferecem uma performance comparável. A ausência de dispositivos de *hardware* específicos, que demandam grandes investimentos para sua concepção tornam os *clusters* uma opção financeiramente acessível.

Os *Beowulfs* são *clusters* construídos a partir de microcomputadores padrão IBM/PC de baixo custo (algumas vezes obsoletos) e integrados computacionalmente através de sistemas operacionais de código livre (*Open Source*), visando um manter o custo o mais baixo possível. A conexão entre os nós se dá através de redes de médio e/ou alto desempenho, como por exemplo, redes *Fast* e/ou *Gigabit Ethernet* [13].

Dependendo da relevância da performance do meio de comunicação é necessário implementar-se duas redes independentes em um *cluster*: uma para o tráfego de dados das aplicações e outra para o tráfego de informações de controle, permitindo o acesso remoto ao *cluster* sem grandes prejuízos à aplicação que estiver sendo executada, ou ainda aumentar o *throughput* de comunicação.

2.3 Medidas de Desempenho

Bons projetos de algorítmos paralelos visam otimizar não somente a velocidade de execução, mas também o consumo de memória, o tempo de implementação, custos envolvidos com a manutenção e a escalabilidade [14].

Ao se analisar a performance de determinada implementação deve-se definir primeiramente, quais aspectos serão importantes. Uma avaliação válida de performance pode ser feita ao analisar-se o nível de aderência de um determinado software aos requisitos especificados pelo usuário, ou então um software pode ser avaliado segundo seu tempo de execução ou eficiência em que usa os recursos de uma arquitetura paralela. Ainda pode-se utilizar um conjunto de métricas, visando obter um software de boa qualidade e estabelecer critérios para a comparação entre implementações.

Na maioria das vezes a paralelização de alguma solução tem por objetivo maior a diminuição do tempo de execução. É de se esperar que a utilização de uma quantidade N vezes maior de recursos computacionais reflita em uma diminuição do tempo de execução até no máximo da mesma ordem. Segundo Amdahl [15], todo software paralelo possui um trecho de código seqüencial que irá limitar o grau de paralelismo máximo alcançável pelo algorítmo. Se a componente seqüencial do algorítmo toma $\frac{1}{s}$ do tempo de execução do programa, então o *speedup* máximo que este programa pode atingir é s.

2.3.1 *Speedup* e Eficiência

O *speedup* é uma métrica largamente utilizada para se medir o ganho obtido pela implementação de um determinado algorítmo em uma sistema paralelo. O *speedup* em P processadores é a razão entre a performance obtida utilizando-se um único processador e a performance obtida utilizando-se P processadores 2.1. A performance geralmente é medida em termos do tempo de execução, o que nos leva à relação:

$$Sr_P = \frac{T_1}{T_P} \tag{2.1}$$

A grandeza acima também é chamada de *speedup* relativo, pois a comparação se dá entre a performance do algorítmo em P processadores e em um único processador. O *speedup* absoluto [14] é calculado através da razão entre o tempo de execução obtido usando o melhor algorítmo conhecido (*best known*) P_{bk1} usando um processador pelo tempo de execução em P processadores (equação 2.2).

$$Sa_P = \frac{T_{bk1}}{T_P} \tag{2.2}$$

Outra maneira equivalente de se expressar o mesmo conceito é através da eficiência. A eficiência mede a fração do tempo que o processador gasta executando um trabalho útil, caracterizando a efetividade com a qual o algorítmo utiliza os recursos computacionais. Define-se como eficiência relativa (Er) para P processadores a razão entre o *speedup* relativo e o número de processadores (equação 2.3). De maneira análoga, calcula-se a eficiência absoluta (Ea).

$$Er_P = \frac{Sr_P}{P}, Ea_P = \frac{Sa_P}{P} \tag{2.3}$$

2.3.2 Escalabilidade

É uma métrica que analisa o quanto o algorítmo é adaptável a mudanças no tamanho do problema, número de processadores, latência e largura de banda da rede de comunicação. Devido à convergência das arquiteturas paralelas [14], a análise de escalabilidade pode ser simplificada para considerar somente o tamanho do problema e o número de processadores sem grandes prejuízos.

2.3.2.1 Escalabilidade com um Problema de Tamanho Fixo

Consiste em analisar como a performance é afetada quando se mantém o problema com um tamanho fixo e se varia no número de processadores. Através desta análise é possível se avaliar quando é interessante a adição de mais processadores para a resolução de um problema, e até mesmo, identificar quando o aumento do número de processadores é inútil ou prejudicial à performance.

2.3.2.2 Escalabilidade com um Problema de Tamanho Variável

Verifica como a performance é afetada quando se varia o tamanho do problema. Um algorítmo com boa escalabilidade apresenta um aumento linear do custo computacional em relação a P para manter a eficiência constante [14]. Geralmente esta relação pode ser prejudicada pela replicação da computação a ser realizada ao invés de sua distribuição.

2.4 Paradigmas da Programação Concorrente

O desenvolvimento de softwares que explorem os recursos computacionais de um sistema paralelo ainda não é totalmente transparente. Ainda não é possível isentar o usuário da tarefa de decidir, levando-se em conta o sistema disponível, como a distribuição do trabalho será feita e/ou qual o mecanismo de comunicação é mais interessante para o problema em questão.

O aumento do nível de abstração é uma constante busca na área da computação. Isto visa permitir que o programador se concentre na resolução do problema ao invés de se concentrar em excessivos detalhes de implementação.

Na computação paralela não é diferente. Atualmente há um nível de abstração confortável para a programação de sistemas monoprocessados, muito favorecidos pela ubiqüidade da arquitetura de Von Neumann [1]. Para os sistemas paralelos, ainda existe um caminho a ser percorrido para que se atinja um nível de abstração comparável ao dos sistemas monoprocessados.

Devido à convergência das arquiteturas paralelas atuais [14] temos dois modelos de programação utilizados na implementação de softwares paralelos. Cada um possui suas próprias peculiaridades e são descritos a seguir:

2.4.1 Passagem de Mensagens

Este modelo de programação se baseia no desenvolvimento de programas que utilizem chamadas explícitas a procedimentos ou funções destinadas a efetuar o transporte de dados entre os processadores que estiverem cooperando na resolução de um problema específico.

Para a execução de um trabalho, é possível se utilizar programas diferentes executados nos diversos processadores, de forma que eles cooperem entre si (conhecido como MPMD - *Multiple Program Multiple Data* ou Múltiplos Programas Múltiplos Dados) ou utilizar somente um programa que dê origem a diferentes processos que são executados nos diferentes processadores sobre dados diferentes (conhecido como SPMD - *Single Program Multiple Data* ou Único Programa Múltiplos Dados).

Devido à exposição da arquitetura computacional, o usuário é o responsável pela paralelização dos seus programas e para isto dispõe de primitivas de comunicação que possuem um comportamento previsível.

Dessa forma, as implementações tendem a serem otimizadas para o problema e para o ambiente onde o usuário produziu o seu programa. Mesmo com a gradual consolidação das arquiteturas paralelas [14], existem muitos sistemas com características distintas e assim, um programa eficiente em um tipo de sistema pode ser ineficiente em outro sistema. Assim sendo durante a migração pode ser necessário adaptar e reescrever o programa, para que o mesmo possa tirar proveito de um ambiente computacional com características diferentes.

2.4.1.1 Message Passing Interface - MPI

MPI (acrônimo de *Message Passing Interface* ou Interface para a Passagem de Mensagens) [16] é um padrão para se implementar programas baseados no paradigma de programação de passagem de mensagens. Esta interface estabelece um padrão flexível, estável, eficiente, portável e simples para a programação baseada na passagem de mensagens entre processos. O MPI foi concebido para ser utilizado em sistemas distribuídos fracamente acoplados, também chamados de Redes de Estações de Trabalho (ou NOWs - *Network of Workstations*), mas funciona perfeitamente em outras arquiteturas.

Durante a definição do padrão MPI, foram levadas em consideração as vantagens de sistemas criados anteriormente, como o P4 [17], o PVM [18] e Express [19] .

Implementações do padrão MPI comerciais e de código aberto são mantidas por várias entidades. Dentre as implementações *Open Source* destacam-se o MPICH [20] e o LAM-MPI. Várias indústrias integradoras de computadores médio e grande porte possuem sua própria implementação do padrão MPI otimizadas para seus próprios sistemas.

O padrão MPI foi implementado como uma biblioteca para as linguagens C e FORTRAN 77. A versão 1.1 do padrão MPI possui 129 funções. O MPI não é um ambiente completo de desenvolvimento e não possui nenhum recurso adicional para a depuração dos programas, embora existam ferramentas disponíveis para isto. O MPI não utiliza o modelo de programação *multithreaded*, mas é um dos seus objetivos poder ser utilizado em ambientes *multithread*, permitindo a existência de vários fluxos de execução em um único processo [12].

2.4.1.2 Parallel Virtual Machine - PVM

O PVM ou *Parallel Virtual Machine* surgiu em 1989 no Oak Ridge National Laboratory criado pelos pesquisadores Vaidy Sunderam e Al Geist. Na ocasião da elaboração deste trabalho o PVM se encontra na versão 3.4.5.

O PVM é um sistema que permite utilizar um conjunto heterogêneo de computadores conectados através de uma rede de comunicação como Ethernet, FDDI, etc. para resolução de problemas em paralelo. O PVM é executado em cada máquina e pode ser configurado pelo usuário de forma que se apresente como um único recurso computacional para aplicações concorrentes.

O modelo computacional do PVM é simples e genérico, acomodando uma grande variedade de estruturas de programas. A interface de programação é direta e permite a criação de programas de maneira intuitiva. O usuário escreve sua aplicação como um conjunto de tarefas que cooperam entre si e as tarefas acessam os recursos do PVM através de chamadas a rotinas de uma biblioteca padrão, que permitem a comunicação e a sincronização entre si [18].

O uso de recursos computacionais heterogêneos é facilitado pelo PVM que cuida da transmissão, roteamento e conversão de dados entre arquiteturas incompatíveis de uma maneira uniforme do ponto de vista do usuário. Como o PVM é um sistema baseado em arquiteturas multicomputacionais fracamente acopladas, ele pode ser utilizado tanto em arquiteturas SMP quanto em *clusters* de computadores conectados por redes de comunicação de alto desempenho.

Para ser utilizado, o PVM deve ser instalado e executado em cada nó do sistema computacional, funcionando como uma camada entre os computadores e o programa escrito através de suas primitivas de comunicação e sincronização do PVM.

2.4.2 Paralelismo por Dados (*Data Parallelism*)

O modelo de paralelismo por dados visa aumentar o nível de abstração, tentando eximir o programador da tarefa de realizar a paralelização, tornando a programação mais confortável e semanticamente próxima ao paradigma de programação seqüencial.

Este tipo de programação se baseia na divisão dos dados a serem processados de forma que cada processador execute uma parte do trabalho. Cada processador deve processar seu próprio conjunto de dados visando diminuir ao máximo a comunicação com outros processadores e maximizar a localidade de referência para um bom uso da hierarquia de memórias existente nos processadores atuais. Esta premissa é definida pela regra *'The Owner Computes'* (O Proprietário Computa) que define que a unidade de processamento responsável por atualizar certo valor é aquela onde este valor esteja armazenado. Geralmente um único programa é executado por cada processador, e cada

processo atua sobre seus próprios dados (SPMD - *Single Program Multiple Data*).

Em linguagens que aderem a este modelo, existem construções implicitamente e explicitamente paralelas que formam um programa. Durante a compilação do programa o compilador detecta e valida instruções paralelas. Caso um processador precise de um dado armazenado na memória principal de outro processador, o compilador providencia para que as operações de comunicação necessárias sejam realizadas.

Como nem todos os problemas podem ser representados por um algorítmo paralelo por dados, linguagens de programação que aderem a este modelo são mais restritivas. No entanto estas prometem uma maior facilidade de implementação e portabilidade, pois ao se mudar de arquitetura, bastaria efetuar a recompilação do programa que o sistema de compilação se encarregaria da geração de código paralelo eficiente para o sistema alvo.

2.4.2.1 High Performance Fortran - HPF

O HPF é um padrão que define extensões para o Fortran 90 (versões 1.0 [21] e 1.1 [22]) e 95 (versão 2.0 [23]) com a finalidade de paralelizar programas Fortran, utilizando o paralelismo por dados. O seu uso envolve a disposição de diretivas que expressam a distribuição de dados entre múltiplos processadores e o relacionamento entre informações que orientam o compilador a explorar ao máximo a localidade de referência. Um programa escrito em HPF pode ser compilado por qualquer compilador Fortran como um programa seqüencial, sendo as diretivas interpretadas como comentários.

O padrão foi publicado inicialmente em maio de 1993 na versão 1.0. Subseqüentemente foi lançada a versão 1.1 em novembro de 1994 e posteriormente a 2.0 em janeiro de 1997. Na versão 2.0 ainda existem inúmeras extensões aprovadas que tem o intuito de torná-lo mais poderoso e adaptável a alguns sistemas paralelos e a implementações que queira explorar o paralelismo por tarefas.

Dentre as diretivas mais importantes encontram-se as seguintes:

- `DISTRIBUTE`: Utilizada para definir como determinado conjunto de dados será dividido entre os processadores. A divisão pode ser feita de três maneiras diferentes: em blocos, cíclica e em blocos cíclicos. Para uma melhor explicação, utilizaremos um exemplo simplificado.

```
      real a(1000), b(1000), c(1000)
!HPF$ PROCESSORS MEUSPROCS(10)
```

```
!HPF$ DISTRIBUTE a(BLOCK), b(CYCLIC), c(CYCLIC(10))
```

No trecho de código acima, as matrizes unidimensionais a, b e c são da mesma ordem. A diferença é a maneira pela qual seus elementos estão distribuídos entre os 10 processadores disponíveis. Os 100 primeiros elementos da matriz a serão colocados no primeiro processador, os elementos de 101 a 200 serão colocados no segundo processador e assim por diante. No caso da matriz b, o elemento número 1 será colocado no primeiro processador, o elemento número 2 no segundo processador, o elemento 11 no primeiro processador, o elemento 12 no segundo e assim por diante. Para a matriz c, os elementos de 1 a 10 serão colocados no primeiro processador, os elementos de 11 a 20 no segundo processador e assim sucessivamente, até que os elementos de 91 a 100 sejam colocados no décimo processador. O ciclo então se reinicia colocando os elementos de 101 a 110 no processador 1, de 111 a 120 no processador 2 até que todos os elementos estejam distribuídos.

- `ALIGN`: Esta diretiva é imprescindível para que o compilador consiga delinear os padrões de comunicação entre os processadores. Em poucas palavras, esta diretiva especifica a distribuição de uma estrutura de dados homogênea em relação à outra. Isto visa facilitar a manutenção do código fonte evitando erros.

  Tomemos como exemplo a multiplicação de um vetor por uma matriz, dado pela equação 2.4.

$$y = A \times x \tag{2.4}$$

O algoritmo se resume a executar a soma dos produtos entre elementos da matriz A e do vetor x, fixando a linha e variando a coluna de A e fixando a coluna e variando a linha de x. Assim sendo, o seguinte programa seqüencial executaria esta tarefa.

```
program matXvet
   real, dimension(100) :: y, x
   real, dimension(100,100) :: A
   integer :: i, k

   call random_number(x)
   call random_number(a)
```

```
      do i=1,100
         y(i) = 0
         do k = 1,100
            y(i) = y(i) + A(i,k) * x(k)
         end do
      end do

      write( *,* ) y
   end program matXvet
```

Um algorítmo paralelo possível permite que cada processador de um sistema paralelo opere sobre uma parte dos dados. Tomando o programa seqüencial como exemplo e estipulando um número de 10 processadores, cada processador seria responsável por calcular 10 resultados. Para que isto seja possível no HPF, basta distribuir os dados corretamente entre os processadores disponíveis.

Pode-se notar que para calcular um resultado $y(k)$, utiliza-se a *k-ésima* linha de A e todas as linhas de x. Visando manter a comunicação em um patamar mínimo é desejável que durante um cálculo de um determinado elemento de y não haja comunicação entre os processadores para a busca de dados.

Levando todas estas necessidades em consideração propomos distribuir a única dimensão de y em blocos e alinhar com a primeira dimensão de A, de acordo com a figura 2.1.

Visando evitar comunicações quando algum valor do vetor x for acessado, faremos sua replicação para todos os processadores. Assim sendo, cada processador possuirá uma cópia local do vetor x. Para que isto ocorra não é necessário utilizar nenhuma diretiva HPF, basta omitir a distribuição ou alinhamento do vetor x.

Assim, temos o seguinte programa paralelizado utilizando as diretivas do HPF[1].

```
program matXvet
   real, dimension(100) :: y, x
   real, dimension(100,100) :: A
   integer :: i, k
```

[1] A diretiva INDEPENDENT é abordada na página 24

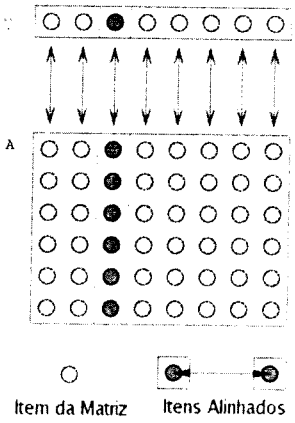

Figura 2.1: Exemplo: Multiplicação de um Vetor por uma Matriz

```
!HPF$ PROCESSORS nos(10)
!HPF$ DISTRIBUTE y(BLOCK) ONTO nos
!HPF$ ALIGN A(:,*) WITH y(:)

          call random_number(x)
          call random_number(a)

!HPF$ INDEPENDENT
          do i=1,100
             y(i) = 0
             do k = 1,100
                y(i) = y(i) + A(i,k) * x(k)
             end do
          end do

          write( *,* ) y
       end program matXvet
```

O símbolo : (dois-pontos) marca as dimensões que deverão ser alinhadas, de forma que se o *k-ésimo* elemento de y for mapeado no processador 3, a *k-ésima* linha de A também o será. O símbolo * (asterisco) marca as dimensões em que os elementos

deverão permanecer juntos, ou que não deverão ser distribuídos.

Vale observar que só é possível alinhar objetos com as mesmas dimensões e mesmos tipos de dados (*shape conformance*).

- `forall`: Esta estrutura permite que o programador expresse operações iterativas que devem ser aplicadas a uma matriz, sem utilizar a estrutura de repetição `do .. end do` que implica em executar cada operação seqüencialmente. Esta estrutura foi definida no padrão HPF versão 1.1 e posteriormente incorporada pelo padrão Fortran 95.

  O seguinte exemplo toma um vetor x de 100 posições e coloca seus valores na diagonal principal de uma matriz quadrada M de dimensão 100.

```
program atribMat
   real, dimension(100) :: x
   real, dimension(100,100) :: M
   integer :: i

   call random_number(x)

   M = 0
   forall ( i=1:100 )
      M(i,i) = x(i)
   end forall

   write( *,* ) M
end program atribMat
```

O `forall` possui uma grande limitação: não é possível fazer qualquer tipo de redução dentro do corpo da repetição. O compilador não permite que uma mesma variável seja atualizada várias vezes dentro do laço, pois isto poderia levar a um problema de concorrência na variável a ser atualizada. O seguinte exemplo não é válido e a variável total não seria atualizada, permanescendo com o valor zero:

```
program somaVet
   real, dimension(100) :: x
```

```
    real :: total
    integer :: i

    call random_number(x)

    total = 0
    forall ( i=1:100 )
       total = total + x(i)
    end forall

    write( *,* ) total
end program somaVet
```

- INDEPENDENT: Esta diretiva é utilizada em conjunto com laços de repetição do .. end do. Essa estrutura é complementar ao FORALL. Serve para assegurar ao compilador que a ordem de execução dos comandos no bloco de repetição não é importante e que não há nenhum tipo de dependência entre as iterações, ou seja, a próxima iteração não depende de valores produzidos na iteração atual. Isto permite que o compilador distribua execução das iterações entre os processadores disponíveis, de forma que cada processador itere sobre seus próprios dados.

  Como exemplo, continuemos utilizando o programa de multiplicação de um vetor por uma matriz da página 22. A presença da diretiva INDEPENDENT permite que o compilador restrinja cada processador a calcular somente os valores de y que estejam mapeados na memória local. Assim sendo, o primeiro processador irá executar calcular os valores dos 10 primeiros elementos de y, o segundo processador calcularia os y de 11 a 20 e assim por diante

  Caso o compilador detecte alguma inconsistência entre a distribuição e alinhamento dos dados com a diretiva INDEPENDENT, o laço de repetição não será paralelizado e uma mensagem de aviso será gerada.

  Associada à diretiva INDEPENDENT existe a diretiva acessória NEW. Esta diretiva informa ao compilador quais são as variáveis que não devem ser sincronizadas durante a execução do laço de repetição, ou seja, que possuirão valores diferentes para cada processador durante o laço. Geralmente tais variáveis são utilizadas como armazenamento temporário para comandos executados dentro do laço. Isto pode ocorrer

tanto para variáveis escalares quanto para matrizes de dados.

- `INHERIT`: Utilizada para assegurar que o mapeamento dos parâmetros de uma determinada subrotina não sejam remapeados no ato da chamada. Os argumentos da subrotina "herdam" o mapeamento atribuído no programa que faz a chamada.

  É essencial atribuir a devida importância a esta diretiva. Programas em Fortran são geralmente organizados em conjuntos de subrotinas e a passagem de argumentos complexos é feita por referência, ou seja, não há a cópia de dados no momento da invocação de uma subrotina.

  A execução de determinadas subrotinas pode envolver um grande custo computacional e é possível solicitar o remapeamento de argumentos para privilegiar a execução destas subrotinas. O remapeamento de dados tende a ser um processo custoso porém justificável em alguns casos.

  O remapeamento de argumentos é declarado através de diretivas HPF de distribuição e alinhamento de dados no preâmbulo da subrotina. Durante a execução da subrotina que solicita o remapeamento ocorrem os seguintes eventos:

  1. Executa-se o remapeamento dos argumentos.
  2. A subrotina é executada utilizando os dados remapeados.
  3. Os argumentos são mapeados de volta para o esquema de distribuição original (assim como era antes da invocação da subrotina).

  Como a redistribuição de dados não foi utilizada neste trabalho, não nos aprofundaremos nestes conceitos, nos restringindo somente a conhecer o processo utilizado para isto.

- Subrotinas "extrínsecas" (*extrinsic subroutines*): São um mecanismo utilizado para permitir a integração de subrotinas não-HPF a um programa HPF. Este tipo de construção pertence a extensões aprovadas do padrão HPF versão 2.0 [23]. Por convenção, programas HPF executam no modo `EXTRINSIC (HPF_GLOBAL)`, ou modo SPMD (Single Program Multiple Data). Abaixo seguem os tipos mais comuns e as situações onde são utilizados:

  1. `EXTRINSIC (HPF_GLOBAL)` É o modo padrão de execução de subrotinas. As subrotinas tem "conhecimento" sobre o formato e a distribuição dos objetos distribuídos, permitindo que o compilador gere as comunicações necessárias para a busca de dados que não estejam no espaço de endereçamento local.

2. `EXTRINSIC (HPF_LOCAL)` É utilizada para se restringir a ação da subrotina a somente os dados que estejam presentes no espaço de endereçamento local. O compilador não gera qualquer tipo de comunicação e até mesmo os índices das matrizes são adaptados para permitirem apenas o acesso local das variáveis. Subrotinas classificadas dessa forma não possuem qualquer visibilidade sobre o estado e a distribuição das matrizes.
3. `EXTRINSIC (HPF_SERIAL)` Faz com que a rotina execute somente em um processador levando em consideração os dados mapeados localmente e remotamente. O compilador providencia a consolidação de todas as partes distribuídas entre os processadores, invoca a subrotina classificada como `EXTRINSIC (HPF_SERIAL)` e após a sua execução, envia os dados modificados para todos os processadores que não participaram do processamento (todos menos o processador onde a subrotina foi executada).
4. `EXTRINSIC (F77_LOCAL)` É útil para se efetuar a chamada de subrotinas de bibliotecas que possuam rotinas compiladas no padrão Fortran 77 [24]. A rotina será executada em todos os processadores, mas apenas sobre os dados mapeados localmente.
5. `EXTRINSIC (F77_SERIAL)` Funciona da mesma maneira que o classificador `EXTRINSIC (HPF_SERIAL)`, porém utilizando o padrão de passagem de argumentos do Fortran 77 [24]. Este classificador não foi utilizado neste trabalho.

O padrão HPF foi implementado em vários produtos, mas muitos foram descontinuados. Neste trabalho utilizamos o *Cluster Development Kit* (CDK) do *Portland Group* e o ADAPTOR.

O CDK encontra-se na versão 5.2 (na data do desenvolvimento deste trabalho), mas a versão utilizada foi a 5.02 que era a versão disponível. Trata-se de um pacote comercial e possui o seu preço associado ao número de nós do sistema. Além do compilador HPF, o pacote também possui compiladores para o Fortran 77 e 90 e para a linguagem C, um depurador (*debugger*) e uma ferramenta para a análise de desempenho (*profiler*). Para utilizar o *profiler* basta o uso de opções em tempo de compilação e para usar o debugger além do uso das opções de compilação, o programa deve ser executado pelo aplicativo depurador.

O ADAPTOR (Automatic DAta Parallelism TranslaTOR) é um tradutor de código fonte FORTRAN que permite a paralelização de programas através do uso de diretivas de compilação e que suporta os padrões HPF 2.0 e OpenMP. Ele utiliza o MPI como meio

de troca de dados entre os processadores, mas o programador não precisa usar e não tem acesso direto às rotinas de comunicação.

Foi desenvolvido pelo SCAI - Fraunhofer-Institute for Algorithms and Scientific Computing ou Instituto Fraunhofer para algorítmos e Computação Científica e seu código-fonte é disponibilizado através da Internet. Na ocasião do desenvolvimento deste trabalho ele se encontrava na versão 10.1. A documentação sobre suas funcionalidades, suas fraquezas e problemas é sucinta e enumera as diferenças entre o que foi implementado e o padrão HPF 2.0 (e suas extensões aprovadas).

2.4.2.2 OpenMP

O OpenMP é um padrão utilizado para direcionar explicitamente a criação de código *multithreaded* em sistemas de memória compartilhada. Sua API engloba 3 componentes primários: diretivas para o compilador (dispostas na forma de comentários no código fonte), subrotinas e variáveis de ambiente.

O OpenMP já foi implementado para a maioria das plataformas UNIX e Windows e pode ser usado com as linguagens C, C++ e Fortran.

Segundo o Comitê de Revisão de Arquitetura ou Architecture Review Board (ARB) [25], o OpenMP não é apropriado para o uso em sistemas paralelos de memória distribuída e embora seja direcionado a arquiteturas de memória compartilhada, não é possível se garantir que ele fará o melhor uso da memória compartilhada, pois não existem diretivas ou outros mecanismos para garantir ou melhorar o uso da localidade de referência.

Os objetivos do OpenMP são padronização, facilidade de uso e portabilidade. A paralelização dos programas é feita pelo compilador através da análise de diretivas colocadas no código fonte na forma de comentários e do próprio corpo do programa.

3 *Metodologia*

3.1 Introdução

Neste capítulo descrevemos a evolução deste trabalho. Abordamos aspectos que dirigiram a escolha da plataforma de portabilidade, as dificuldades encontradas durante o processo de paralelização, o desenvolvimento de uma ferramenta para a validação dos resultados produzidos pelo programa, o uso de uma ferramenta de controle de revisões e, por fim, a tentativa de se utilizar um sistema de compilação HPF *Open Source*, o ADAPTOR.

3.2 Avaliações Iniciais

O trabalho teve início pela análise do problema físico com a finalidade de se adquirir os conhecimentos mínimos necessários para o entendimento das implementações já existentes: a implementação padrão em Fortran 95 e a implementação paralela que possui chamadas a funções da biblioteca MPI.

Efetuamos análises preliminares de performance para ambas as variantes, explorando a execução de cada uma delas (seqüencial e MPI) sobre problemas de tamanhos e complexidades diferentes. As avaliações contemplaram tanto o tempo de execução quanto o consumo de espaço de endereçamento principal e *swap*, com o intuito de se adquirir familiaridade com o ambiente computacional e com a implementação que já estava pronta.

3.3 Plataforma de Implementação

Em seguida, iniciamos o estudo da plataforma de portabilidade utilizada neste trabalho: o *High Performance Fortran.*

As razões que nortearam seu uso foram:

- A suposta facilidade de paralelização de um programa seqüencial
- A facilidade de manutenção das versões seqüencial e paralela,
- A portabilidade do código
- A disponibilidade da plataforma.

Nos parágrafos abaixo explicitamos estes motivos e comparamos a plataforma escolhida às outras opções disponíveis (OpenMP, PVM e MPI).

3.3.1 Facilidade de Paralelização

Conforme pode ser verificado na seção 2.4.2.1, o método empregado para a paralelização se baseia na disposição de diretivas de compilação na forma de comentários interpretáveis por compiladores HPF. Devido a esta simplicidade, esperávamos que o tempo de implementação e, conseqüentemente, o custo de desenvolvimento de uma variante paralela seria mais baixo. A plataforma de portabilidade OpenMP (veja a seção 2.4.2.2) também atende a este requisito, mas esta é indicada apenas para arquiteturas de memória compartilhada, enquanto que as plataformas PVM e MPI (seções 2.4.1.2 e 2.4.1.1) não atendem a este quesito, pois sua utilização depende da invocação de rotinas de comunicação específicas.

Devido a várias dificuldades encontradas durante a paralelização do trecho que efetua a diagonalização de matrizes, verificamos que esta expectativa não foi plenamente atendida. Dentre os motivos, podemos salientar a ausência de documentação sobre as mensagens de erro exibidas pelo compilador e a falta de clareza dessas mensagens. Isto nos levou a modificar o código intuitivamente tentando elaborar construções que o compilador fosse capaz de paralelizar. Além destas dificuldades, apesar de se tratar de um software proprietário, não conseguimos ser atendidos pelo departamento de suporte da empresa que distribui o software mesmo após várias tentativas.

3.3.2 Facilidade de Manutenção

A facilidade de manutenção é conseguida com a existência de somente um código fonte. Os códigos-fonte seqüencial e paralelo são exatamente os mesmos. A diferença é a disposição das diretivas HPF na forma de comentários em locais específicos do código fonte. Caso as diretivas sejam ignoradas, têm-se a implementação seqüencial. Caso as

diretivas sejam interpretadas pelo compilador HPF, têm-se a versão paralela do software. O OpenMP também atende a este requisito, mas o PVM e o MPI não.

3.3.3 Portabilidade

A portabilidade do código também é uma característica desejada durante a implementação de uma variante paralela de um software. É desejável que a portabilidade do software entre máquinas paralelas que pertençam a paradigmas diferentes seja conseguida com o mínimo esforço possível. Isto seria possível com o HPF bastando que existisse um compilador HPF para a plataforma destino.

Mesmo com a portabilidade garantida, a paralelização pode ser atingida de várias maneiras, de acordo com o que foi constatado neste trabalho. Utilizando o HPF fornecido pelo Portland Group, a paralelização é conseguida através do paradigma utilizado pelo MPI, que se resume a se criar um número de processos que interagem através das primitivas de comunicação ponto-a-ponto e em grupo. Estes processos podem estar dispostos em máquinas diferentes ou na mesma máquina. Assim, caso se escolha utilizar um *cluster* computacional, basta que os processos executem em nós diferentes e, caso se utilize uma única máquina multiprocessada de memória compartilhada, cada processo pode ser executado por um processador diferente.

3.3.4 Disponibilidade

Todas as plataformas avaliadas estão disponíveis para o uso geral. O HPF, o OpenMP e o MPI são padrões internacionais desenvolvidos por órgãos suportados por grandes empresas e pela comunidade científica. O PVM é uma ferramenta desenvolvida por um laboratório de pesquisa. Todas essas plataformas possuem implementações em código livre disponíveis.

No nosso caso, o uso do HPF se deu primeiramente através de um compilador proprietário desenvolvido pelo Portland Group. Este compilador estava apropriadamente instalado e configurado no *cluster beowulf* utilizado para este trabalho. Posteriormente o ADAPTOR também foi instalado.

Apesar da disponibilidade das ferramentas acima, compiladores HPF não são facilmente encontrados atualmente. Muitas implementações se tornaram obsoletas e não são mais distribuídas e nem utilizadas.

3.4 A Implementação em HPF

A paralelização do programa HPF envolveu paralelizar dois laços iterativos de repetição e duas subrotinas de caráter global. Também utilizamos os classificadores "extrínsecos" (seção 2.4.2.1, página 25) em uma subrotina e nas funções por ela invocadas.

3.4.1 Paralelizações Iniciais

Primeiramente paralelizamos a subrotina que constrói e diagonaliza um Hamiltoniano para cada ponto da malha que descreve a representação irredutível da primeira zona de *Brillouin*. Esta subrotina é a mais custosa para os problemas do tipo 1 (seção 1.1, página 4).

O número de matrizes a serem diagonalizadas depende diretamente do tamanho da malha do problema a ser simulado. Normalmente se diagonaliza de 400 a 2500 matrizes para cada iteração do ciclo autoconsistente, mas esse número pode chegar a até 8000 matrizes.

O processo executado por esta rotina consiste na construção da matriz hamiltoniana de cada ponto da malha e na invocação da subrotina `CHEEVX` do pacote LAPACK [26] para sua diagonalização. A paralelização consiste em atribuir um conjunto de pontos da malha diferentes para cada um dos processadores. Dessa forma, cada processador é responsável por construir e diagonalizar um número aproximadamente igual de hamiltonianos. A paralelização desta tarefa é altamente eficiente, pois há um nível muito reduzido de comunicações envolvidas e a tarefa demanda uma grande quantidade de processamento restrito à CPU [12].

A título de exemplo, supondo que existam 7936 matrizes a serem diagonalizadas em 12 processadores, teria-se que 11 processadores diagonalizariam 662 matrizes e o último processador seria responsável pela diagonalização de 654 matrizes. A distribuição dos índices do laço iterativo seria:

- Processador 1: Da primeira matriz até a matriz 662.
- Processador 2: Da matriz de número 663 à matriz 1325
- ...
- Processador 12: Diagonalizará 654 matrizes (da matriz 7282 à 7936)

3.4.2 Problemas com o LAPACK

Também tivemos uma grande dificuldade inicial para obter sucesso nas primeiras execuções. A subrotina `ILAENV` do pacote LAPACK que vem com o compilador do Portland Group precisou sofrer um *patch* pois a execução era terminada abruptamente sem um motivo aparente. Este *patch* é recomendado pelos próprios desenvolvedores devido a diferenças de arquitetura. Há um trecho de código que pode ser problemático para determinadas arquiteturas que apresentam algum tipo de não-conformidade como padrão de ponto flutuante definido pelo IEEE.

Além deste problema, o pacote LAPACK distribuído com os compiladores do Portland Group é incompleto e foi necessário incorporar as subrotinas restantes. Como este procedimento não é trivial criamos um procedimento, disponível na seção 4.3 (página 53) para a correção do problema com a subrotina `ILAENV` e para a inclusão das subrotinas restantes.

3.4.3 Outras Paralelizações

A implementação prosseguiu com as outras partes paralelizáveis do código. Descobrir quais eram os trechos em que a paralelização traria maiores benefícios não foi difícil pois isto já havia sido feito quando da criação da primeira variante concorrente escrita em MPI.

3.4.3.1 Paralelização da Resolução da Equação de Poisson

Para a paralelização deste trecho fomos levados a mudar o código fonte da subrotina que efetua os cálculos necessários para a resolução da equação de Poisson. Basicamente, retiramos o laço de repetição mais abrangente de dentro da subrotina levando-o para o programa principal e utilizamos a diretiva `INDEPENDENT` para tornar sua execução concorrente. Assim, utilizamos o mesmo tipo de solução empregada na paralelização da diagonalização de matrizes.

3.4.3.2 Paralelização de subrotinas: Luminescência

O cálculo da luminescência consiste em realizar a convolução entre 2 matrizes de vetores. A convolução se dá através do cálculo do produto de cada vetor do cubo A com cada vetor do cubo B um a um. Com o aumento do tamanho do problema a ser

processado o custo computacional cresce exponencialmente.

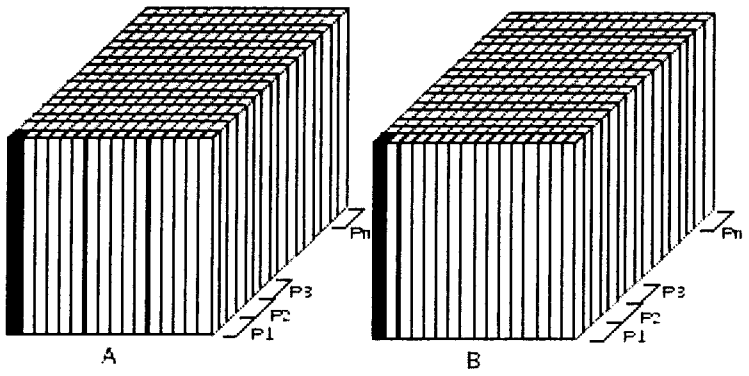

Figura 3.1: Matrizes envolvidas no cálculo da luminescência

A paralelização das subrotinas que efetuam o cálculo da luminescência ocorreu em nível de subrotina, ou seja, não foi necessário se fazer nenhuma modificação no código fonte do programa principal. Foi necessário apenas determinar a distribuição das matrizes utilizadas por estas subrotinas.

A distribuição dos dados está indicada na figura 3.1. O cubo é dividido pelo número de processadores, de forma que o primeiro fique com a primeira parte, o segundo com a segunda e assim por diante. Durante o cálculo toma-se um elemento do primeiro vetor do primeiro cubo e percorre-se os vetores do segundo cubo um a um realizando os cálculos.

Tivemos as mesmas dificuldades elucidadas na seção 3.3.1 (página 29). A principal dificuldade, no entanto, foi dada pela impossibilidade de se utilizar a diretiva `INHERIT`. De acordo com o padrão HPF versão 1.1 [22], deveria ser utilizada para viabilizar a herança das distribuições e alinhamentos dos parâmetros das funções, evitando redistribuições dos dados ao se entrar e sair de subrotinas (seção 2.4.2.1, página 25).

Ao invés do `INHERIT`, utilizou-se o `DISTRIBUTE *` que possui a mesma funcionalidade, caso os dados já estejam apropriadamente distribuídos no programa que invoca a subrotina. O problema é que sua utilização deixa o código menos legível, pois as distribuições e alinhamentos devem ser refeitos para cada subrotina. Além disso, ele gera um potencial problema de manutenção visto que se uma distribuição for modificada no módulo principal, ela deve ser modificada em toda subrotina que a herde para evitar o remapeamento.

Além da disposição de diretivas sobre a distribuição dos dados utilizamos a diretiva `INDEPENDENT` para a paralelização dos laços de repetição. Para que o compilador fosse

capaz de efetuar a paralelização, fizemos a reordenação de alguns laços de repetição. Isso também beneficiou o código seqüencial tornando-o mais eficiente, possibilitando um melhor aproveitamento da localidade de referência espacial e temporal, e conseqüentemente na melhoria na utilização dos elementos da hierarquia de memória do sistema.

3.4.4 A Asserção de Não-Paralelização: Fermi

Para evitar o *speedup* negativo da rotina que efetua a determinação do nível de Fermi do sistema foi necessário empregar o qualificador `EXTRINSIC(HPF_SERIAL)`. Este instrui o compilador a executar a subrotina somente no nó principal e a sincronizar os resultados em todos os processadores após a execução (seções 2.4.2.1 e 2.4.2.1).

Esta abordagem se mostrou mais eficiente do que não utilizar nenhum qualificador (o que implica no uso automático do `extrinsic(hpf_global)`. Devido às características do algorítmo, o compilador precisava efetuar a comunicação de pequenos blocos de dados com muita freqüência, levando a um *speedup* negativo.

Devido às características do algorítmo também não foi possível executá-lo utilizando o qualificador `extrinsic(hpf_local)` pois é mandatório o acesso a dados que estejam alocados localmente em outros nós de processamento.

3.5 ADAPTOR

A princípio, o ADAPTOR [27] se mostrou uma opção interessante para a paralelização de programas utilizando o padrão HPF. O ADAPTOR teve sucesso durante os testes com programas exemplo, mas não foi capaz de endereçar a compilação de um sistema maior como é o caso da aplicação paralelizada.

O ADAPTOR precisa ser utilizado em conjunto com um compilador FORTRAN, e para este trabalho, utilizamos o Intel Fortran Compiler (IFC) versão 8.0.

A instalação foi trabalhosa e após conseguirmos realizá-la com sucesso elaboramos um documento contendo instruções para tornar o processo passível de reprodução (seção 4.3, página 55). Não foi necessário nenhum tipo de adaptação nas bibliotecas utilizadas pois as bibliotecas do Math Kernel Libraries 7.0 da Intel funcionaram perfeitamente com o Intel Fortran Compiler 8.0, conforme o esperado.

Precisamos reescrever algumas partes do código pois a checagem de sintaxe e semântica do pré-compilador do ADAPTOR é mais rígida do que a checagem executada pelos outros

compiladores utilizados até então. As mudanças eram feitas sob demanda sem nenhum auxílio das mensagens do compilador, somente através de tentativas e erros, fazendo mudanças mínimas na declaração de variáveis, nas declarações do tipo de retorno de funções, removendo linhas de inclusão de módulos, dentre outras.

3.5.1 Mudanças Necessárias

Devido à experiência adquirida e à grande quantidade de informações produzidas pelo pré-compilador esperávamos que a paralelização de subrotinas no ADAPTOR fosse mais fácil do que a paralelização efetuada no HPF do Portland Group. No entanto, devido a existência de problemas na execução do programa que serão detalhados oportunamente, isso não ocorreu.

O ADAPTOR se mostrou mais aderente ao padrão HPF (mais especificamente à versão 2.0) principalmente nos pontos abaixo:

- No caso das diretivas que foram utilizadas para que as subrotinas herdassem o *layout* de dados estabelecido no programa principal, não foi necessário utilizar nenhuma construção inesperada para efetuar a herança das distribuições a alinhamentos dos parâmetros, bastando utilizar a diretiva `INHERIT`.

- Para os laços `INDEPENDENT`, a sua criação foi mais direta, pois não havia qualquer detecção de quais variáveis deveriam e quais não deveriam ser declaradas como NEW (seção 2.4.2.1, página 24).

  No caso do compilador do Portland Group, não era necessário se declarar todas as variáveis não sincronizáveis. No ADAPTOR, todas as variáveis não sincronizáveis devem ser declaradas para que o laço possa ser paralelizado. Caso as variáveis não sejam declaradas, o compilador emite uma mensagem dizendo que o laço não pode ser paralelizado. No caso do ADAPTOR, o pré-compilador se isenta de reconhecer quais variáveis devem ser sincronizadas e quais não devem. Isto deve ser feito pelo programador através do uso da diretiva acessória `NEW` associada à diretiva `INDEPENDENT`

  Esta característica do ADAPTOR facilitou o desenvolvimento, permitindo um controle mais fino sobre processo de desenvolvimento e evitando que o compilador decida algo não aprovado pelo programador.

- Mudança de nomes de variáveis que poderiam conflitar com nomes de subrotinas.

- Remoção de variações sobre os tipos INTEGER e REAL. O parâmetro KIND não é suportado pelo ADAPTOR (ex: INTEGER(1) substituído por INTEGER)

- Problemas com os tipos de dados derivados (*Derived Data Types*) e módulos. As funções que pertenciam aos módulos precisaram ser convertidas para subrotinas. Algumas outras funções foram convertidas por subrotinas devido a problemas desconhecidos durante a compilação. A conversão foi feita por palpite, sem indicações do compilador.

- Foi necessário remover comandos USE inúteis. O compilador também não indicou o problema.

- Em alguns casos, não é possível se utilizar : (dois pontos) nas diretivas ALIGN e DISTRIBUTE. Deve-se utilizar variáveis inteiras como índices (ex: I,J,...)

- Reordenação de *loops*. O ADAPTOR não identifica e não faz isso automaticamente [27]. Além de viabilizar a aplicação da diretiva `INDEPENDENT`, esta mudança também possibilitou a otimização do uso da hierarquia de memórias [8], aproveitando a localidade física e temporal dos dados sendo acessados e modificados.

Durante a compilação é possível solicitar que o ADAPTOR não apague os arquivos temporários que são criados durante as várias fases da tradução do código fonte. Também é possível manter o arquivo final (que será linkado com as bibliotecas de distribuição de vetores ou DALIB - *Distributed Array LIBrary*). Mesmo que o programa tenha sido escrito usando o padrão Fortran 90 ou 95, o ADAPTOR converte a maioria do código para construções do Fortran 77. Caso o programa use *Derived Data Types*, eles não serão modificados pelo ADAPTOR.

3.6 O Uso do CVS para o Controle das Variantes

Utilizamos o CVS (Concurrent Versions System) visando obter um controle mais fino sobre o processo de desenvolvimento e sobre o código fonte. A versão seqüencial do programa foi tomada como a versão tronco. As versões paralelas para o PGHPF e para o ADAPTOR foram introduzidas como variantes do código seqüencial através da criação de 2 linhas de desenvolvimento paralelas (ou *branches*).

Além disso, o uso do CVS permitiu que houvesse o controle da evolução do código fonte e também de como alguns problemas foram resolvidos. Isto ocorria de maneira que

toda mudança que introduzisse uma melhoria ou que reparasse um erro ou eliminasse um aviso de compilação fosse inserida no repositório. Durante a inserção do programa corrigido no repositório, colocavamos também o erro resolvido ou melhoria implementada na forma de comentário.

Dessa forma, estava implantada a rastreabilidade do código fonte no processo de desenvolvimento. Qualquer erro que fosse inserido poderia ser desfeito utilizando-se a versão armazenada no repositório.

3.7 Validação das Versões Intermediárias: *Sanity*

Durante o processo de desenvolvimento, sentimos a necessidade de comprovar se o código produzido ou modificado havia introduzido algum erro no resultado final produzido pelo software. De maneira bem simples, seria possível conduzir esta análise apenas especificando-se uma entrada conhecida e analisando-se o resultado produzido pelo programa em testes e uma saída produzida por uma versão estável do software.

Durante o desenvolvimento do programa que efetua a comparação dos resultados, tivemos em mente a natureza não-determinística dos valores situados depois da sexta casa decimal nos números de ponto flutuante. A comparação é feita através da diferença (d) entre um resultado produzido pelo programa sob análise (R_a) e um resultado equivalente produzido por um programa confiável (R_c). Este resultado deve ser menor que uma constante épsilon (ϵ) para ser considerado um resultado aceitável (equação 3.1).

$$d = R_c - R_a < \epsilon \tag{3.1}$$

O programa foi chamado de *Sanity* e é responsável por fazer comparações em cada um dos potenciais armazenados no arquivo de entrada e saída **entrada.het**. Caso sejam encontrados erros, o Sanity reporta o problema encontrado escrevendo diretamente na saída padrão de forma a permitir o processamento direto da saída por quaisquer utilitários de processamento de texto disponíveis nas diversas plataformas computacionais.

4 Resultados e Discussões

4.1 Plataforma de Desenvolvimento

A implementação deste trabalho se deu em um *cluster* Intel composto por 6 nós conectados por duas redes de comunicação distintas:

- Fast Ethernet: rede utilizada para o tráfego de dados de controle, como os gerados durante uma conexão remota. Isto visa evitar que as execuções de programas não sejam influenciadas pelo tráfego desses dados pelo sistema.
- Gigabit Ethernet: rede utilizada para o tráfego de dados diretamente relacionados com a execução dos programas.

Cada nó do *cluster* conta com dois processadores Pentium III de 1.13 GHz com 256MB de memória *cache*, dispostos em uma placa-mãe Intel e com um disco rígido SCSI de 17 GB. O primeiro nó do equipamento possui 1,0 GB de memória RAM ECC e os outros 5 nós possuem 3 GB de memória RAM ECC. O *cluster* roda o Fedora Core 1 com o kernel 2.4 SMP.

Utilizamos os compiladores Fortran 90 e HPF do *Cluster Development Kit* da Portland Group na versão 5.02. Neste mesmo sistema encontram-se instalados o Intel Fortran Compiler na versão 8.0 e o Math Kernel Libraries na versão 7.0, estes últimos usados nas tentativas de compilação e execução com o ADAPTOR.

4.2 Configuração do Software

Para as tomadas de tempo, utilizamos as 3 variantes disponíveis do programa:

1. Seqüencial
2. Concorrente MPI

3. Concorrente HPF

Todas as versões foram compiladas utilizando os compiladores da Portland para o Fortran 90 e para o High Performance Fortran. A versão MPI foi compilada com a biblioteca MPI que acompanha o *Cluster Development Kit*. As opções de otimização que são especificadas em tempo de compilação foram as mesmas para as 3 variantes, visando manter a maior consistência possível entre as medidas do tempo de execução das três versões.

Devido ao grande número de arquivos e subrotinas, o processo de compilação foi controlado pelo aplicativo GNU Make na versão 3.8.

4.3 Medidas de Performance

Para a execução das medidas consideramos os dois tipos de problemas (veja seção 1.1, página 4). O primeiro demanda um processamento intensivo no processo de diagonalização de matrizes enquanto que o segundo demanda um maior tempo de computação durante o cálculo dos espectros de luminescência. Embora não haja interesse nos resultados físicos (além do compromisso com a exatidão dos resultados), o arquivo de entrada foi configurado com as propriedades físicas de uma heteroestrutura baseada em GaAs.

A instrumentação do código foi feita com cautela. Tentamos manter o número de pontos de medida o menor possível para que a performance não fosse prejudicada. Tomamos a precaução de instrumentar trechos de programa relativamente custosos para que o impacto na medida local fosse mantido num patamar mínimo [28].

Além disso, instrumentamos somente o nó principal. Isto não interfere nos resultados obtidos, pois o programa explora o paralelismo por dados e existe uma sincronia das atividades realizadas em cada processador. Além disso, toda entrada e saída é realizada pelo nó principal, o que reforça ainda mais a necessidade de se concentrar as medidas neste elemento.

Efetuamos três medidas para cada problema analisado e os resultados são constituídos pela média das três execuções.

4.4 Diagonalização de Matrizes

Os dados de entrada para esta execução estabeleciam uma execução para 30 ondas planas e 10 pontos em cada direção da malha. Foram executadas 20 iterações no *loop* autoconsistente e não foi realizado o cálculo da luminescência. Na tabela 4.1 constam os tempos de execução e o *speedup* (equação 2.1) relativo para um número diferente de processadores. O tempo de execução seqüencial médio obtido foi de 12381,16 segundos. A eficiência pode ser facilmente calculada utilizando os dados abaixo com base na equação 2.3.

# Processadores	Tempo HPF(s)	Tempo MPI(s)	*Speedup* HPF	*Speedup* MPI
1	12619,46	12682,58	0,98	0,98
2	6696,94	6603,09	1,85	1,88
4	3576,43	3689,61	3,46	3,36
6	2629,41	2568,98	4,71	4,82
8	2122,78	2107,73	5,83	5,87
10	1803,65	1762,2	6,86	7,03
12	1529,02	1488,89	8,1	8,32

Tabela 4.1: Problema Tipo 1 - Speedup geral HPF e MPI

Assim, verificamos que o *speedup* atingido pela variante HPF se mantém próximo ao *speedup* da variante MPI, conforme a tabela 4.1 e a figura 4.1.

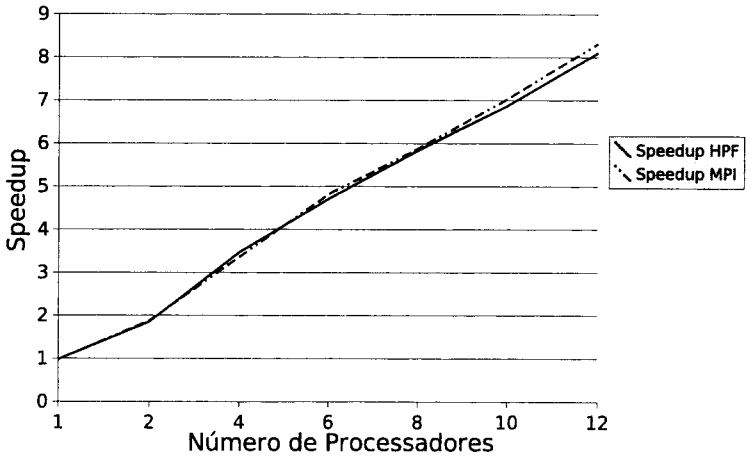

Figura 4.1: Problema Tipo 1 - Speedup geral HPF e MPI

Através da instrumentação apenas do trecho da diagonalização de matrizes, foi possível medirmos o tempo dispendido durante este trecho. Os tempos de execução produziram a tabela 4.2 e a figura 4.2 que mostram um *speedup* ainda mais agressivo para ambas as

variantes. Com esta medida é possível avaliar a efetividade da concorrência atingida por ambos os métodos de paralelização, sem levar em conta a componente seqüencial presente em vários trechos do programa. O tempo de execução seqüencial para a diagonalização foi em média 600,78 segundos.

# Processadores	Tempo HPF(s)	Tempo MPI(s)	*Speedup* HPF	*Speedup* MPI
1	621,87	616,83	0,96	0,97
2	323,47	323,58	1,85	1,85
4	159,71	165,66	3,75	3,62
6	114,89	102,39	5,22	5,86
8	85,22	83,8	7,04	7,15
10	70,67	66,04	8,48	9,08
12	56,58	57,27	10,6	10,47

Tabela 4.2: Problema Tipo 1 - Speedup diagonalizações HPF e MPI

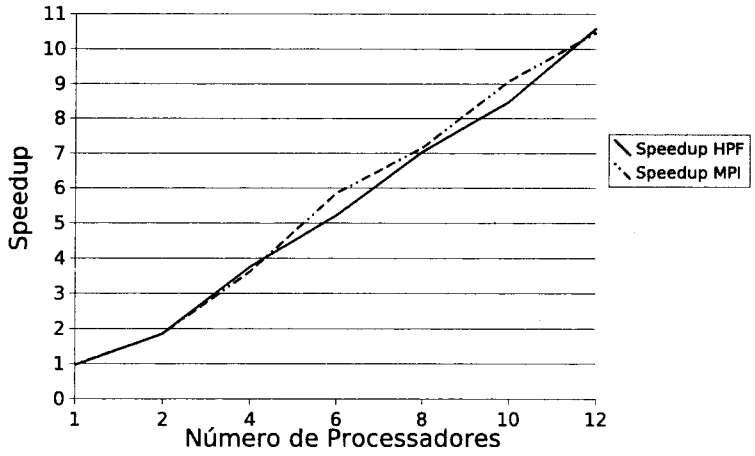

Figura 4.2: Problema Tipo 1 - Speedup diagonalizações HPF e MPI

Usando ainda dados obtidos com a instrumentação do programa, pudemos verificar que que há um forte impacto das componentes seqüenciais dentro do *loop* autoconsistente. As componentes seqüenciais são representadas por trechos de código que executam entrada e saída (I/O) e algumas funções de caráter estritamente seqüencial, como a função que calcula o nível de Fermi do sistema e a função que calcula a distribuição de cargas no sistema.

Visando manter a performance das funções seqüenciais em um patamar equivalente ao obtido na variante seqüencial, utilizamos extensões do padrão HPF versão 2.0 que permitiram a qualificação das subrotinas como sendo de execução seqüencial por apenas um processador (veja a seção 2.4.2.1).

4.5 Espectros de Luminescência

Para este tipo de execução consideramos dados de entrada fixados em 10 ondas planas e 10 pontos em cada direção da malha. Foi executada apenas uma iteração no *loop* autoconsistente pois o intuito não era medir a performance do processo de diagonalização de matrizes. Os tempos de execução e *speedup* relativo podem ser visualizados na tabela 4.3 para até 12 processadores.

O tempo de execução seqüencial médio obtido foi de 10387,24 segundos.

# Processadores	Tempo HPF(s)	Tempo MPI(s)	*Speedup* HPF	*Speedup* MPI
1	10638,79	11428,62	0,98	0,91
2	5305,16	5730,20	1,96	1,81
4	2671,19	2887,89	3,89	3,60
6	1801,02	2029,51	5,77	5,12
8	1462,14	1561,38	7,10	6,65
10	1127,47	1227,32	9,21	8,46
12	992,17	1012,92	10,47	10,25

Tabela 4.3: Problema Tipo 2 - Speedup geral HPF e MPI

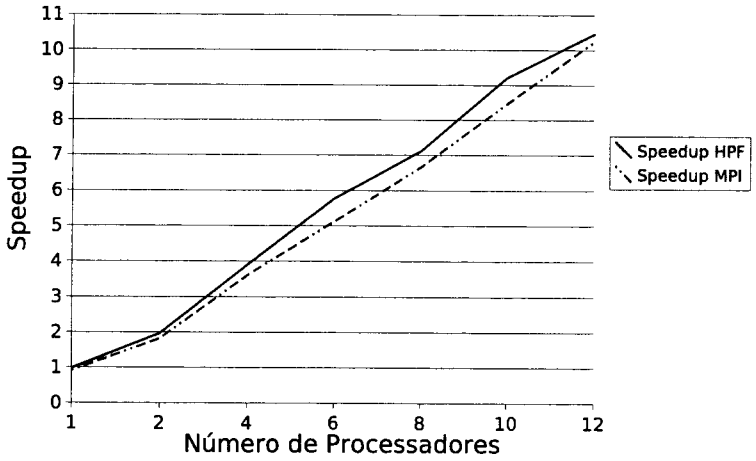

Figura 4.3: Problema Tipo 2 - Speedup geral HPF e MPI

Através da análise da figura 4.3 e da tabela 4.3 é possível verificar que o *speedup* atingido pela variante HPF foi superior ao *speedup* atingido pela variante MPI este resultado pode ser confirmado através da tabela 4.4 d da figura 4.4 que exibe apenas o speedup do trecho paralelizado. Este foi um resultado inesperado, porém justificável.

O algorítmo do programa concorrente escrito em MPI é conhecido pois a paralelização é visível através da chamada às primitivas de comunicação e sincronização. No entanto,

# Processadores	Tempo HPF(s)	Tempo MPI(s)	*Speedup* HPF	*Speedup* MPI
1	10528,77	11310,43	0,99	0,92
2	5217,53	5635,55	1,99	1,84
4	2618,50	2830,93	3,97	3,67
6	1750,97	1973,11	5,93	5,26
8	1412,25	1508,10	7,36	6,89
10	1075,28	1170,52	9,66	8,87
12	938,41	958,03	11,07	10,84

Tabela 4.4: Problema Tipo 2 - Speedup convoluções HPF e MPI

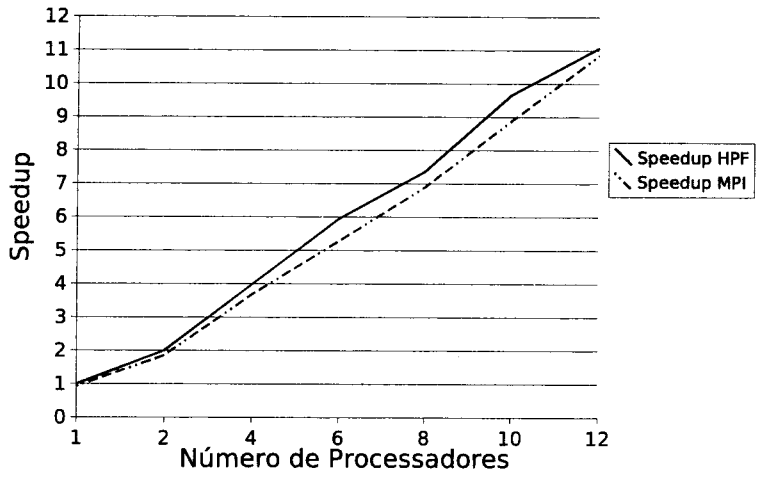

Figura 4.4: Problema Tipo 2 - Speedup convoluções HPF e MPI

o algorítmo do programa concorrente paralelizado pelo compilador HPF é desconhecido pois o compilador do Portland Group não armazena nenhuma versão intermediária que permita visualizar as chamadas a procedimentos de comunicação ou sincronização. Devido à localização da diretiva `INDEPENDENT` presumimos que há uma comunicação orientada à demanda Desta forma, a partir do momento que um bloco de dados não armazenado localmente é necessário há a comunicação ponto-a-ponto.

Em vista disto, tentamos nos assegurar quanto ao padrão de comunicação da variante HPF através do uso de um aplicativo que permitisse determinar o padrão de comunicação durante o cálculo do espectro de luminescência. Foram registrados todos os pacotes de dados trocados entre o nó principal e os outros nós do sistema.

O aplicativo utilizado foi o *ethereal* versão 0.10.3. Os gráficos de comunicação foram gerados pelo seu "front-end" gráfico projetado para o gerenciador de janelas Gnome.

Os perfis de comunicação gerais podem ser verificados nas figuras 4.5 (variante MPI) e 4.6 (variante HPF). Estes perfis sofreram uma magnificação no trecho de maior interesse

gerando as figuras internas etiquetadas com a palavra "Magnificação".

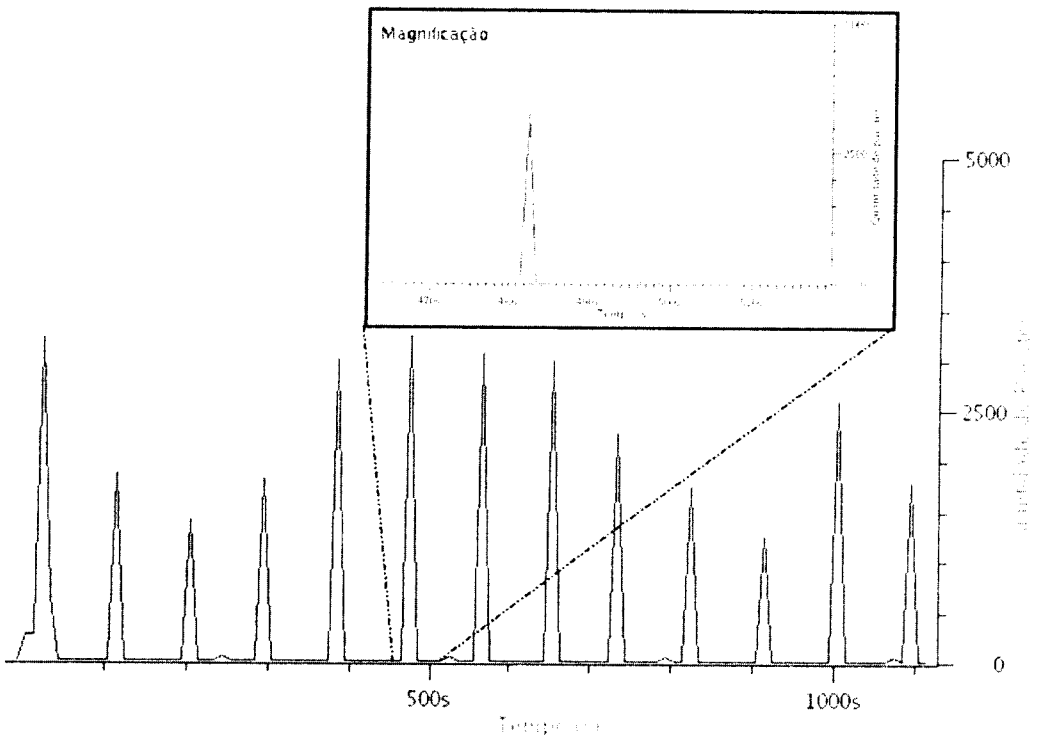

Figura 4.5: Perfil de comunicações - MPI

Na magnificação da figura 4.5 encontra-se o padrão de comunicações da variante MPI. Verificamos que as trocas de mensagens ocorrem um número fixo de vezes em momentos bem definidos. Neste caso, existem 13 picos: 1 para o cálculo de uma diagonalização e outros 12 que ocorrem durante o cálculo do espectro de luminescência, pois utilizamos 12 processadores para este cálculo.

No caso da execução da variante HPF utilizamos também 12 processadores. Através da análise da figura 4.6, é possível verificar que a comunicação não permanece isolada em determinados períodos, mas ao invés disto ela ocorre durante toda a execução do cálculo do espectro de luminescência, como pode ser visto na magnificação na figura 4.6. Em momentos diferentes a comunicação se dá entre processadores diferentes. Isto sugere que a cada fase do processamento, o processador principal necessita de dados que estejam em processadores diferentes, de acordo com as distribuições de dados feitas no programa principal. Isto permite que a infraestrutura de comunicação disponível seja ótimamente aproveitada. O uso de switches (ao invés de hubs) permite que haja a comunicação direta entre os nós, levando ao ganho acima verificado.

Dessa forma, verificamos que a plataforma de portabilidade e seu sistema de com-

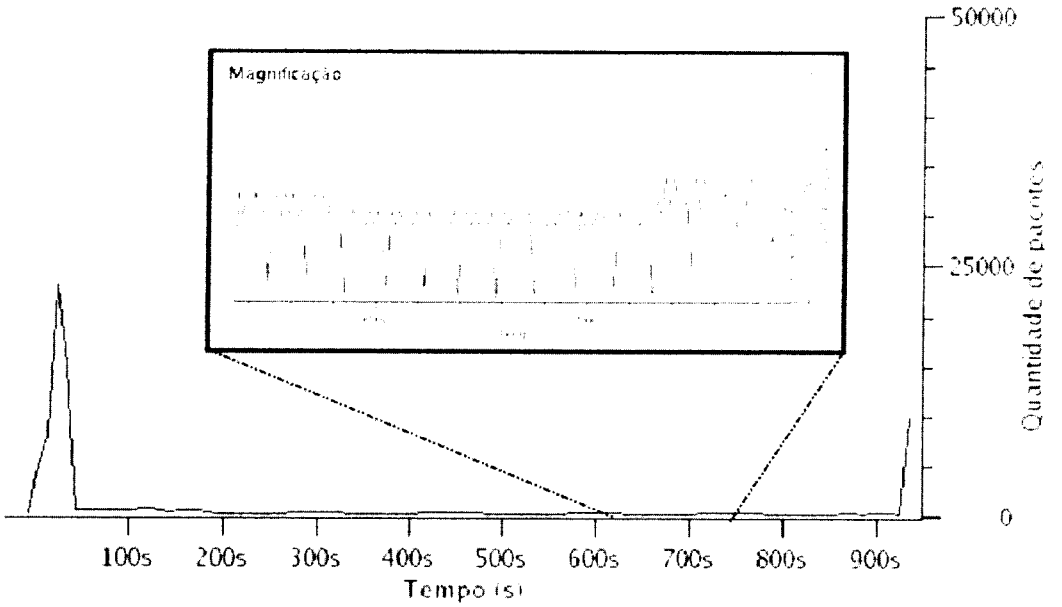

Figura 4.6: Perfil de comunicações - HPF

pilação podem ser ainda mais eficientes do que o próprio programador na paralelização de um programa dependendo do problema a ser paralelizado.

4.6 Problemas com o ADAPTOR

Não obtivemos êxito com a paralelização do programa de cálculo de estruturas de bandas utilizando o ADAPTOR pois a execução em paralelo do trecho de código que deveria executar a diagonalização de matrizes não funcionou. O programa era terminado abruptamente, emitindo uma mensagem de erro do sistema de comunicação que encapsula chamadas a rotinas do MPI conhecido como DALIB (Distributed Array LIBrary). Após várias semanas de tentativas, conseguimos remover este erro através de uma total reengenharia do método de diagonalização que utilizou o qualificador de funções `extrinsic(hpf_local)`. Assim, o sistema de comunicação do ADAPTOR conseguiu gerenciar a distribuição de dados.

Numa segunda fase, mesmo conseguindo executar as diagonalizações em paralelo, não conseguimos utilizar a rotina `CHEEVX`, que causava o término abrupto do programa devido a uma falha de segmentação. Após analisarmos o código fonte gerado nas fases intermediárias da pré-compilação, pudemos constatar que o ADAPTOR modificava a

chamada e sua respectiva interface, adicionando descritores das matrizes que eram passadas como parâmetros para o rotina `CHEEVX`. Fizemos várias tentativas utilizando os qualificadores de rotinas `extrinsic` e seus acessórios como os classificadores LAYOUT e PASS_BY (disponíveis somente no padrão HPF 2.0). O uso desta extensão realmente faz somente metade do trabalho, pois evita o envio de descritores de matrizes à funções do padrão Fortran 77, mas faz a duplicação na interface, não permitindo que o código fonte sequer compile. Talvez isto possa ser caracterizado como um defeito (ou bug) no ADAPTOR 10.1, pois o seu comportamento não está de acordo com o descrito no padrão HPF 2.0. Este problema foi reportado aos responsáveis pela manutenção e desenvolvimento da ferramenta.

Ainda tentando superar este problema, tentamos optar pela criação de um *wrapper* para a rotina `CHEEVX`. Esta rotina encapsuladora era compilada diretamente, não sendo submetida ao processo de pré-compilação. Também não obtivemos sucesso com esta tentativa.

Além dos problemas para a invocação de funções de bibliotecas que recebessem matrizes como parâmetros, o ADAPTOR também apresentou problemas quanto ao gerenciamento das estruturas de dados temporárias criadas para viabilizar a paralelização do código. Nos testes realizados ocorreram algumas falhas de segmentação devido a incorreta manipulação de matrizes alocadas dinâmicamente criadas pelo próprio sistema de compilação. Dessa forma, não foi possível avaliarmos o desempenho do programa utilizando o compilador HPF ADAPTOR.

Conclusões

4.1 O trabalho desenvolvido

4.1.1 Performance e Uso do High Performance Fortran

A performance obtida com a paralelização utilizando um sistema de compilação que suporta o padrão HPF superou nossas expectativas iniciais. Não esperávamos que uma ferramenta que provê uma solução genérica conseguiria endereçar tão bem a paralelização do nosso problema.

Sua estrutura inspira uma grande facilidade para paralelizar aplicações através de uma curva de aprendizado que rapidamente atinge a condição de regime, mas isto não reflete a realidade. As primeiras paralelizações foram difíceis de serem conseguidas. Devido ao contato com outros grupos de pequisa que trabalham com computação concorrente e às pesquisas feitas na Internet e em periódicos onde são publicados artigos na área, pudemos perceber que o HPF é uma plataforma pouco utilizada para a paralelização de aplicações.

Assim não pudemos contar com o auxílio de programadores mais experientes para a troca de informações. Também não encontramos listas ou fóruns de discussão, nem listas de perguntas mais freqüentes (FAQs). Conforme pode ser verificado na seção 3.3.1 (página 29), houveram ainda outros problemas encontrados durante o aprendizado. A documentação do sistema de compilação do Portland Group não é suficientemente clara quanto à aderência ao padrão HPF. Consta que o compilador suporta características da versão 1.1 [22], mas foi possível utilizar com êxito uma extensão aprovada do HPF 2.0 [23]).

4.1.2 O uso do ADAPTOR

Mesmo com a falta de êxito, o ADAPTOR mostrou-se uma ferramenta interessante para o desenvolvimento de programas paralelos. Os programas utilizados para os primeiros testes foram paralelizados com sucesso. O ADAPTOR se mostrou valioso para melhorar e depurar melhor o código pois seu pré-compilador faz uma análise mais estrita do programa

incluindo também uma análise interprocedural das subrotinas automaticamente. Isso permitiu melhorar o código em alguns pontos onde os outros compiladores não acusavam nenhum problema.

Por outro lado, não foi possível utilizar a biblioteca LAPACK, estritamente necessária para o problema em questão. Seu uso foi baseado em tentativas e erros além de exercitar a intuição do programador para mudar a maneira de se escrever algumas linhas de código sem nenhum auxílio das mensagens geradas pelo compilador.

Alguns dos problemas existentes para o pacote do Portland Group também estiveram presentes no uso do ADAPTOR: a falta de grupos de usuários, fóruns e listas de discussões. Também não existe nenhum tipo de suporte por parte dos desenvolvedores do software, que foram acionados sobre o problema com o LAPACK, mas sem nenhum retorno.

De forma geral, foi adquirido um extenso conhecimento sobre o padrão HPF e parte de suas extensões aprovadas. Ele decorreu das horas investidas na depuração do programa e demandou a constante consulta aos padrões publicados pelo HPF Fórum e a restrita documentação do ADAPTOR.

4.2 Análise da Usabilidade do HPF para o Processamento Científico

O uso de plataformas de portabilidade como o *High Performance Fortran* para a paralelização de programas "paralelos por dados" (*Data Parallel*) dispensa o uso de ferramentas que aplicam o paradigma de passagem de mensagens (*Message Passing*).

Devido a uma menor exposição de aspectos ligados a arquitetura do sistema, a paralelização de programas pode ser feita mais rapidamente. Apesar de um potencial problema ligado ao tempo necessário ao aprendizado e à adaptação inicial ao HPF, este trabalho apresenta algumas das dificuldades encontradas para o programador iniciante nesta tecnologia. Apesar das dificuldades, o programador iniciante se certificará que é possível obter-se um *speedup* comparável ao obtido por programas desenvolvidos utilizando alguma plataforma baseada em passagem de mensagens.

4.3 Perspectivas

O uso de plataformas de portabilidade [2] visa diminuir a complexidade da utilização de múltiplos recursos computacionais para a resolução concorrente de problemas. Seguindo a mesma linha de pesquisa deste trabalho, é possível avaliar outros mecanismos de paralelização de programas como o OpenMP, em conjunto com sistemas operacionais para *clusters* de computadores como o MOSIX e o OpenMOSIX. O OpenMOSIX é um sistema operacional SSI - *Single System Image* - que permite que o usuário utilize os recursos de vários computadores conectados entre si como se fosse somente um sistema. Como o OpenMOSIX é capaz de gerenciar e migrar apenas processos entre os nós do sistema, é necessário aplicar um *patch* (ainda em fase experimental) para permitir o gerenciamento de threads individualmente, o que viabilizaria a paralelização usando o OpenMP. O OpenMP possui uma quantidade maior de usuários e recursos para guiar o desenvolvimento. É possível também vislumbrar o uso do ScaLAPACK (versão concorrente do LAPACK) em clusters ou grids de computadores com o HPF, MPI ou OpenMP sobre o OpenMOSIX.

Referências

1 NEUMANN, J. V. *First draft of a report on the EDVAC.* Moore School, University of Pennsylvania: [s.n.], 1945.

2 CENTURION, A. M. Mestrado, *Análise de Desempenho de Algoritmos Paralelos Utilizando Plataformas de Portabilidade.* São Carlos, SP: [s.n.], Dezembro 1998.

3 SIPAHI, G. M. *Teoria do Confinamento de Buracos em Heteroestruturas semicondutoras do tipo δ – Doping.* Tese (Douturado) — Universidade de São Paulo, São Paulo, SP, 1997.

4 SNIR, M.; OTTO, S.; HUSS-LEDERMAN, S.; WALKER, D.; DONGARRA, J. *MPI: The Complete Reference.* Cambridge, Massachussets: The MIT Press, 1996.

5 ALMASI G.S. E GOTTLIEB, A. *Highly Parallel Computing.* 2ª. ed. Redwood City, CA: Benjamin/Cummings, 1989.

6 FLYNN, M. Some computer organizations and their effectiveness. *IEEE Transactions on Computing*, v. 21, n. 9, p. 948–960, Setembro 1972.

7 FLYNN M.J. E RUDD, K. Parallel architectures. *ACM Computing Surveys*, ACM Press, v. 28, n. 1, p. 67–70, March 1996. ISSN 0360-0300.

8 HWANG, K. *Advanced computer architecture: parallelism, scalability, programmability.* New York, NY: McGraw-Hill, 1993.

9 STALLINGS, W. *Arquitetura e Organização de Computadores: Projeto para o Desempenho.* 5ª. ed. Sao Paulo, SP: Prentice Hall, 2002.

10 CULLER, D. E. *Parallel Computer Architecture: A Hardware/Software Approach.* San Francisco, CA: Morgan Kaufmann, 1998.

11 TANENBAUM, A. S. *Structured Computer Organization.* 4ª. ed. Upper Saddle River, NJ, USA: Campus, 1999.

12 TANENBAUM, A. S. *Modern Operating Systems.* 2ª. ed. Upper Saddle River, NJ, USA: Prentice Hall, 2001.

13 TANENBAUM, A. S. *Redes de Computadores.* 4ª. ed. Rio de Janeiro, RJ: Campus, 2003.

14 FOSTER, I. *Designing and Building Parallel Programs.* 1ª. ed. Boston, MA, USA: Addison Wesley Publishing Company, 1995.

15 AMDAHL, G. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In: *Proceedings of the AFIPS Spring Joint Computer Conference, Atlantic City, New Jersey, USA.* [S.l.]: AFIPS Press, Reston, Virginia, USA, 1967. p. 483–485.

16 MPIFORUM. *MPI: A Message-Passing Interface Standard.* [S.l.], 1994, acessado em 15/10/2004. Disponível em: <http://citeseer.ist.psu.edu/forum94mpi.html>.

17 ARGONNE NATIONAL LABORATORY. *User's Guide to the p4 parallel programming system. Technical Report ANL-92/17.* USA, October 1992.

18 GEIST, A.; BEGUELIN, A.; DONGARRA, J.; JIANG, W.; MANCHEK, R.; SUNDERAM, V. S. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing (Scientific and Engineering Computation S.).* Cambridge, Massachussets: The MIT Press, November 1994.

19 PARASOFT CORPORATION. *Express Version 1.0. A communication environment for Parallel Computers.* USA, 1988.

20 ARGONNE NATIONAL LABORATORY. *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard.* USA, acessado em 26/09/2004. Disponível em: <http://www.mcs.anl.gov/mpi/mpich/papers/mpicharticle/paper.html>.

21 HPFFORUM. *High Performance Fortran Language Specification v1.0.* May 1993, acessado em 29/09/2004. Disponível em: <http://citeseer.lcs.mit.edu/article/hpff93high.html>.

22 HPFFORUM. *High Performance Fortran Language Specification v1.1.* Nov 1994, acessado em 10/10/2004. Disponível em: <ftp://softlib.rice.edu/pub/HPF/hpf-v11.ps.gz>.

23 HPFFORUM. *High Performance Fortran Language Specification v2.0.* Jan 1997, acessado em 28/09/2004. Disponível em: <http://www.crpc.rice.edu/HPFF/versions/hpf2/files/hpf-v20.ps.gz>.

24 American National Standards Institute. *American National Standard programming language FORTRAN: approved April 3, 1978, American National Standards Institute, Inc.: ANSI X3.9-1978. Revision of ANSI X3.9-1966.* 1978, acessado em 07/10/2004. World-Wide Web document. Disponível em: <http://www.fortran.com/fortran/F77_std/rjcnf0001.html>.

25 ARB, O. *OpenMP Fortran Application Program Interface.* [S.l.], November 2000.

26 ANDERSON, E.; BAI, Z.; BISCHOF, C.; BLACKFORD, L. S.; DEMMEL, J.; DONGARRA, J. J.; CROZ, J. D.; HAMMARLING, S.; GREENBAUM, A.; MCKENNEY, A.; SORENSEN, D. *LAPACK Users' guide.* third. [S.l.]: Society for Industrial and Applied Mathematics, 1999. ISBN 0-89871-447-8.

27 INSTITUTE FOR ALGORITHMS AND SCIENTIFIC COMPUTING (SCAI). *ADAPTOR HPF Language Reference Manual.* Augustin, Germany, 2004, March 17.

28 REED, D. A.; SHIELDS, K. A.; SCULLIN, W. H.; TAWERA, L. F.; ELFORD, C. L. Virtual reality and parallel systems performance analysis. *IEEE Computer*, v. 28, n. 11, p. 57–67, 1995.

29 WHALEY, R. C.; PETITET, A.; DONGARRA, J. J. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, v. 27, n. 1–2, p. 3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

30 INSTITUTE FOR ALGORITHMS AND SCIENTIFIC COMPUTING (SCAI). *ADAPTOR HPF Programmers Guide*. Augustin, Germany, 2004, March 5.

Anexos

Anexo A - Reconstrução da biblioteca LAPACK para uso com o CDK

Recompilando o liblapack.a para o HPF (High Performance Fortran) do Portland Group CDK 5.02

Este *how-to* foi feito para orientar a reconstrução de uma biblioteca LAPACK otimizada e capaz de ser utilizada no *Cluster Beowulf* do grupo de Instrumentação e Informática do Departamento de Física da USP de São Carlos.

Descrição do ambiente computacional

- *Hardware*: *Cluster* Intel de 12 processadores (Pentium III 1.13GHz SSE), com 6 nós SMP de 2 processadores cada
- Sistema Operacional: Fedora Core 1 (kernel 2.4 SMP)
- *Software*: Portland Group *Cluster Development Kit* 5.02

Para construir o liblapack.a que funciona com o compilador pghpf, siga os passos a seguir dentro de um diretório criado para este fim:

1. Faça o download do ATLAS [29] (sistema de compilação e otimização automática das bibliotecas de subrotinas básicas de álgebra linear - BLAS) pré-compilado para a processadores Pentium III SSE (testado com a versão 3.6.0)
2. Crie o diretório tmp

   ```
   # mkdir tmp
   ```

3. Entre no diretório tmp e descompacte o ATLAS .

```
# cd tmp
# tar zxvf ../atlas3.6.0_Linux_PIIISSE1.tar.gz
```

4. Copie o arquivo `liblapack.a` que vem com o compilador da Portland para o diretório atual. Supondo que o CDK 5.02 esteja no diretório `/usr/pgi/linux86/5.0/lib`, faça:

```
# cp /usr/pgi/linux86/5.0/lib/liblapack.a .
```

5. Descompacte o `liblapack.a` que vem com o ATLAS no diretório atual:

```
# ar x Linux_PIIISSE1/lib/liblapack.a
```

6. PASSO MAIS IMPORTANTE!: Remova o arquivo `ilaenv.o` do diretorio atual para evitar que o ilaenv da Portland seja sobrescrito.

```
# rm ilaenv.o
```

7. Copie o arquivo `bsilaenv` que acompanha este *how-to* [1] para o diretório atual e recompile-o utilizando o compilador HPF da Portland.

```
# pghpf -c bsilaenv.f
```

8. Reconstrua o `liblapack.a` com os objetos otimizados do ATLAS e com o `bsilaenv.o` (sem problema para maquinas *non-IEEE conformant*)

```
# ar r liblapack.a *.o
```

PRONTO! A biblioteca está pronta!

Autores: Rodrigo Malara (malara@sc.usp.br) e Guilherme Sipahi (sipahi@if.sc.usp.br)

[1] Disponível em http://lfc.ifsc.usp.br/ rodrigo/bsilaenv.f

Anexo B - Instalação do ADAPTOR usando o Intel Fortran Compiler 8.0

Instalação do ADAPTOR-HPF usando o Intel Fortran Compiler em um *cluster beowulf*

Este *how-to* foi feito para facilitar a tarefa de se configurar e compilar o ADAPTOR-HPF usando o Intel Fortran Compiler no *cluster beowulf* do grupo de Instrumentação e Informática do Departamento de Física da USP de São Carlos.

Descrição do ambiente computacional

- *Hardware*: *Cluster* Intel de 12 processadores (Pentium III 1.13GHz SSE), com 6 nós SMP de 2 processadores cada
- Sistema Operacional: Fedora Core 1 (kernel SMP)
- *Software*: Intel Fortran Compiler versão 8.0 instalado
- MPICH 1.2.5.2 compilado com o Intel C Compiler versão 8.0

Para instalar o ADAPTOR versão 10.1 na sua área pessoal, siga os passos a seguir dentro de um diretório criado para este fim:

1. Faça o download do código fonte do Adaptor do site do SCAI-Fraunhoffer-Institute for Algorithms and Scientific Computing [30].
2. crie o diretorio adaptor

```
# cd
# mkdir adaptor
```

3. entre no diretório adaptor e descompacte o ADAPTOR

```
# cd adaptor
# tar zxvf ../adaptor_10.1.tar.gz
```

4. Exporte as seguintes variáveis para o ambiente utilizando os comandos abaixo:

```
# export F77='/usr/bin/ifort'
# export F77_FLAGS='-unroll -tpp6 -align -pad'
# export F77_EXTEND='-extend_source'
# export F77_NOWARN='-w'
# export F77_MP='-auto'
# export MPI_HOME='/usr/local/built/mpich-1.2.5-intel'
# export MPI_LIBS='mpich'
```

5. Faça a configuração do ADAPTOR. Isto vai levar apenas alguns segundos...

```
# ./configure
```

6. PASSO MAIS IMPORTANTE: Edite o arquivo .adaptorrc e atribua o valor "-Vaxlib -static" a variável de ambiente LD_FLAGS (fica aproximadamente na linha 82)

```
LD_FLAGS=-Vaxlib -static
```

7. Agora já é possível proceder com a criação dos binários do ADAPTOR. Esta operação pode levar vários minutos:

```
# make
```

8. Caso a compilação tenha terminado com sucesso, o ADAPTOR já estará pronto para ser utilizado. Para usá-lo, atualize o seu arquivo .bashrc para inserir as variáveis de ambiente utilizadas pelo ADAPTOR (utilize seu editor de textos favorito):

```
export PHOME=$HOME/adaptor
export PATH=$PATH:$PHOME/bin
export MANPATH=$MANPATH/$PHOME/man
```

9. Aplique as alterações no seu shell atual:

```
# source ~/.bashrc
```

10. Faça um programa simples (test.f90) para testar se o ADAPTOR está funcionando:

```
program TEST
WRITE(6,"('ADAPTOR OK')")
end
```

11. Compile o programa:

```
# adaptor -hpf_1 -o test test.f90
```

12. Execute o programa

```
# ./test
```

PRONTO! O ADAPTOR já pode ser usado!

Autores: Rodrigo Malara (malara@sc.usp.br) e Guilherme Sipahi (sipahi@if.sc.usp.br)

Anexo C - Comparação da performance: Intel versus Portland Group

Metodologia

A comparação entre os compiladores Fortran 90 foi feita utilizando um problema do tipo 1 (seção 1.1, 4). Os compiladores e as bibliotecas LAPACK utilizadas foram:

- Intel Fortran Compiler (IFC) 8.0 e Math Kernel Libraries (MKL) 7.0
- Portland Group *Cluster Development Kit* (CDK) 5.02. Parte do LAPACK já veio incluído no pacote e parte precisou ser compilada a partir do código-fonte e adicionada ao LAPACK que veio com o CDK.

Para cada medida realizamos 3 execuções (devidamente instrumentadas) do programa seqüencial em duas arquiteturas distintas:

- Pentium III de 1.13GHz e 1 GB de memória principal (RAM) ECC padrão EDO 133MHz
- Opteron 2.0 GHz com 4 GB de memória principal (RAM) padrão DDR 400MHz.

Vale notar que a performance obtida não deve ser atribuída somente ao sistema de compilação, mas também às bibliotecas LAPACK utilizadas. É importante salientar que a MKL incorpora otimizações feitas pela própria Intel e o LAPACK que foi utilizado com o compilador Portland não passou por este mesmo processo. Além disso, é necessário se considerar que não foi possível utilizar o MKL com o compilador do Portland Group, pois o padrão de nomeação de rotinas após a compilação difere, não sendo possível gerar o arquivo binário.

Medidas

As medidas obtidas na plataforma Pentium III podem ser verificadas na tabela A.1.

Na tabela A.2 constam as medidas obtidas na plataforma Opteron.

Conclusão

Compilador	**Tempo (s)**	**Diferença (%)**
IFC	3015	0%
CDK	4002	32,73%

Tabela A.1: Comparação entre compiladores plataforma Pentium III

Compilador	**Tempo (s)**	**Diferença (%)**
IFC	1343	0%
CDK	1421	5,80%

Tabela A.2: Comparação entre compiladores plataforma Opteron

Para o tipo de problema por nós estudado, verificamos que o conjunto IFC/MKL se sobressai em relação ao compilador do Portland Group, principalmente em arquiteturas compostas por processadores de fabricação da própria Intel. No caso do processador Opteron isso ainda ocorre, mas com uma menor intensidade, pois o compilador da Intel foi desenvolvido para uma arquitetura de 64 bits supostamente diferente, apesar de ambos os processadores serem variantes da arquitetura x86.