UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE FÍSICA DE SÃO CARLOS
DEPARTAMENTO DE FÍSICA E INFORMÁTICA

PARALELIZAÇÃO DE UM PROGRAMA PARA CÁLCULO DE PROPRIEDADES FÍSICAS DE IMPUREZAS MAGNÉTICAS EM METAIS

Eloiza Helena Sonoda

Dissertação apresentada ao Instituto de Física de São Carlos, Universidade de São Paulo, para obtenção do título de Mestre em Física Aplicada.

Orientador: *Prof. Dr. Gonzalo Travieso*

SÃO CARLOS
2001

Aos meus pais.

Agradecimentos

Ao Prof. Dr. Gonzalo Travieso pela grande competência com que orientou esse trabalho, e principalmente pela valorosa dedicação, bom humor e amizade.

Ao Prof. Dr. Jan Slaets pelos questionamentos e sugestões valiosos.

Ao Prof. Dr. Valter Líbero pelas discussões esclarecedoras acerca do modelo de Anderson, sempre muito solícito e atencioso, e ao Prof. Dr. Luis Nunes de Oliveira por ter cedido o código do seu programa para usarmos como ponto de partida.

Ao Marcatinho, para o qual não existem palavras que agradeçam todo amor, carinho e compreensão. Seu apoio e companheirismo foram fundamentais para a conclusão deste trabalho.

Aos meus irmãos, Yuri e Kathia. Especialmente aos meus pais, Mari e Luiz, pelo imenso amor e oportunidades que me propiciaram, este trabalho é resultado do nosso esforço comum.

Aos colegas do grupo: Bruno, Zem, Paulino, Francisco, Raul, Patrícia, Cláudio, especialmente ao Andrré pela amizade sólida e à Thaty que se revelou uma ótima companhia.

Agradeço à FAPESP pelo suporte financeiro.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho se dedica à paralelização de um programa para cálculos de propriedades físicas de ligas magnéticas diluídas. O método do grupo de renormalização aplicado ao modelo de Anderson de duas impurezas se mostrou particularmente adequado ao processamento paralelo visto que grande parte dos cálculos pode ser executada simultaneamente, assim como variações nos conjuntos de dados requeridas pelo método. Para tal reescrevemos o programa seqüencial usado anteriormente pelo Grupo de Física Teórica do IFSC e implementamos três versões paralelas. Essas versões diferem entre si em relação à abordagem dada à paralelização. O uso de *clusters* de computadores se revelou uma opção conveniente pois verificamos que o limitante no desempenho é o tempo tomado pelos cálculos e não pela comunicação. Os resultados mostram uma grande redução no tempo total de execução, porém deficiências no *speedup* e escalabilidade devido a problemas de balanceamento de carga. Analisamos esses problemas e sugerimos alternativas para solucioná-los.

Abstract

This dissertation discuss the parallelization of a program that calculates physical properties of dilute magnetic alloys. The renormalization group method applied to Anderson's two impurities model showed to be specially suitable to parallel processing because a large amount of calculations as well as variations of data entries required by the method can be performed simultaneously. To achieve this we rewrote the sequential program previously used by the Theoretical Physics Group of the IFSC and wrote three parallel versions. These versions differ from each other by the parallelization approach. The use of computer clusters revealed to be an appropriate option because the calculation time is the limiting factor on performance instead of communication time. The results show a good reduction of execution time, but speedup and scalability lack due to load balancing problems. We analyze these problems and suggest possible solutions.

Capítulo 1

Introdução

Este trabalho se dedica à paralelização de um programa para cálculo de propriedades físicas de impurezas magnéticas em metais através do uso do método do grupo de renormalização [1, 2] para o modelo de Anderson [3] estendido a duas impurezas [4]. Foram observadas anomalias em certos metais com a presença de impurezas magnéticas, tal como um mínimo no gráfico de resistividade pela temperatura [5]. O problema desperta interesse pois a formação do estado fundamental nesse modelo depende fortemente da competição entre dois efeitos, a interação RKKY [6, 7, 8] e o efeito Kondo [1].

O grupo de Física Teórica do IFSC vem adquirindo extensa experiência nesse problema [9, 10, 11], tendo desenvolvido um programa para os cálculos, mas tem como fatores limitantes nas pesquisas o tempo tomado por sua execução e o uso excessivo de memória do computador quando certa precisão é requerida. Além disso, é necessário executar algumas vezes o programa variando parâmetros iniciais para eliminar oscilações típicas do método. Com todas essas exigências é fácil perceber o gargalo computacional.

O problema é de grande interesse para a área de programação

paralela, visto que apresenta características adequadas às técnicas de paralelização. Grande parte dos cálculos, assim como as diferentes execuções com variações nos parâmetros, podem ser realizados simultaneamente por computadores trabalhando cooperativamente a fim de melhorar seu desempenho, sendo este o objetivo deste trabalho.

No capítulo 2 descrevemos brevemente os aspectos do modelo empregado, assim como do método numérico utilizado. Não temos a pretensão de apresentar o formalismo completo, mas sim de ilustrar o problema físico em que trabalhamos.

A seguir, no capítulo 3, apresentamos características de processamento paralelo [12], o desenvolvimento de sistemas paralelos de baixo custo como os *clusters* de computadores, o modelo de programação paralela de passagem de mensagem utilizado no desenvolvimento do trabalho e também uma descrição da estrutura do *cluster* do tipo Beowulf [13] existente no laboratório.

Descrevemos detalhadamente as implementações dos programas no capítulo 4. Primeiramente discutimos uma versão seqüencial elaborada a partir da utilizada anteriormente, orientada a objetos e visando a paralelização. Esta nova versão alcançou uma redução de 70% do tempo de execução. Posteriormente apresentamos uma versão paralela baseada em variação de dados, a qual executa paralelamente os cálculos para os diversos conjuntos de parâmetros. Temos ainda a versão paralela por setores, que executa os cálculos de cada conjunto de parâmetros concorrentemente, e a versão paralela híbrida que integra as anteriores.

As análises dos desempenhos para alguns casos dessas versões são mostradas no capítulo 5. Verificamos que todas as versões paralelas apresentam um tempo de processamento sensivelmente menor

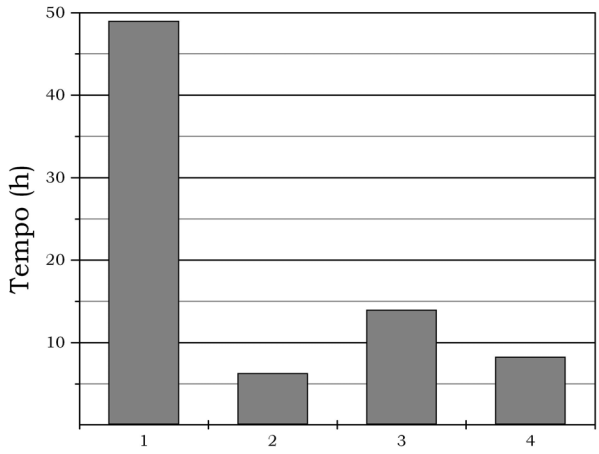

Figura 1.1: Histograma dos tempos de execução de oito conjuntos de dados para diferentes versões do programa utilizando 15 processadores, seguindo a numeração: 1) execuções seqüenciais; 2) paralelização baseada em variação de parâmetros; 3) execuções paralelas por setores; 4) paralelização híbrida.

que a nova versão seqüencial, como pode ser visto na figura 1.1 (ver secção 5.4 para mais detalhes), onde mostramos o tempo de execução em horas de oito conjuntos de dados de entrada utilizando 15 processadores segundo o esquema:

1. execuções seqüenciais;
2. paralelização baseada em variação de parâmetros;
3. execuções paralelas por setores;
4. paralelização híbrida (dois conjuntos de dados simultâneos);

As conclusões e sugestões para futuros trabalhos são apresentadas no capítulo 6.

Fechamos essa dissertação com três apêndices, todos de caráter técnico e explicativo. No apêndice A discutimos as classes

que foram escritas, descrevendo características importantes de seus membros e métodos. No apêndice B colocamos os três arquivos de entrada que foram usados nas execuções para análise do desempenho das diferentes implementações realizadas. Finalmente o apêndice C apresenta aspectos importantes do MPI e também os protótipos das rotinas que usamos na elaboração dos programas paralelos.

Capítulo 2

Modelo e Método Numérico

A aplicação considerada neste trabalho realiza cálculos para determinação de propriedades físicas de impurezas magnéticas em metais. Para descrever os momentos magnéticos utilizamos o modelo inicialmente proposto por Anderson para uma impureza em 1961 [3], e em 1964 estendido a duas impurezas por Anderson e Alexander [4], apresentado na secção 2.1.

O método numérico empregado, o grupo de renormalização [1, 2], secção 2.2, foi o primeiro tratamento capaz de descrever todo o espectro de temperaturas, permitindo aprofundar o estudo da formação do estado fundamental e também permite calcular propriedades dinâmicas mesmo em regiões de *crossover*. Este método é baseado na discretização logarítmica da banda de condução do metal hospedeiro à qual a impureza está acoplada.

2.1 Modelo de Anderson

Desde 1930 tem-se observado anomalias em certos metais com a presença de impurezas magnéticas. O primeiro efeito observado foi um mínimo no gráfico de resistividade pela temperatura. Esses sis-

temas são chamados de ligas magnéticas diluídas (metais magnéticos dissolvidos em metais não magnéticos em concentração menor que 1%) e neles ocorre a formação de momentos magnéticos localizados. Estudos indicaram que o mínimo na resistividade é devido ao espalhamento dos *spins* dos elétrons no sítio da impureza e dos elétrons de condução do metal hospedeiro [5, 14].

Proposto para descrever momentos magnéticos localizados, o modelo de Anderson, em sua versão original com uma impureza [3], se aplica a ligas magnéticas diluídas. Momentos magnéticos localizados ocorrem em metais não magnéticos contendo impurezas iônicas com camadas de valência interna incompletas (orbitais d e f). Exemplos de impurezas desse tipo são os metais de transição, os lantanídeos e os actinídeos.

Anderson supôs o metal hospedeiro não magnético representado por uma banda de elétrons de condução com estados de Bloch de energia $\varepsilon_{\vec{k}}$ e momentum $\vec{k}$. A impureza é um nível separado da banda de condução. O nível acrescentado pela impureza é caracterizado pela energia do orbital d (ou f) ε_d com degenerescência de *spin* σ. O orbital da impureza possui quatro configurações possíveis pelo princípio de exclusão de Pauli: vazio, ocupado com um elétron com *spin* para cima, ocupado com um elétron com *spin* para baixo ou duplamente ocupado. O acoplamento entre banda de condução e impureza é representado através do elemento de matriz não diagonal V (hibridização). A taxa de transição entre impureza e banda de condução é calculada pela regra de ouro de Fermi.

A repulsão Coulombiana U entre os elétrons de *spins* opostos que ocupam o mesmo orbital na impureza reduz a probabilidade da dupla ocupação do orbital da impureza e, se a energia ε_d for negativa, favorece a ocupação simples do orbital. Uma representação

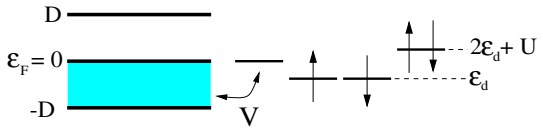

Figura 2.1: Representação esquemática do modelo de Anderson de uma impureza. O metal hospedeiro, não magnético, é representado por uma banda de condução de largura $2D$ semi-preenchida com estados de energia $\varepsilon_{\vec{k}}$. A impureza magnética é um nível localizado caracterizado pela energia ε_d. U é a repulsão Coulombiana entre elétrons ocupando o mesmo orbital na impureza. A interação do orbital localizado com a banda de condução ocorre via hibridização V. Fonte: [15].

esquemática do modelo é apresentada na figura 2.1.

O Hamiltoniano que descreve o modelo de Anderson de uma impureza é:

$$H = \sum_{\vec{k}\sigma} \varepsilon_{\vec{k}} c^{\dagger}_{\vec{k}\sigma} c_{\vec{k}\sigma} + \varepsilon_d \sum_{\sigma} c^{\dagger}_{d\sigma} c_{d\sigma} + V \sum_{\vec{k}\sigma} (c^{\dagger}_{\vec{k}\sigma} c_{d\sigma} + \text{h.c.}) + U c^{\dagger}_{d\uparrow} c_{d\uparrow} c^{\dagger}_{d\downarrow} c_{d\downarrow} \quad (2.1)$$

onde $c^{\dagger}_{\vec{k}\sigma}$ cria um elétron com momentum $\vec{k}$ e *spin* $\sigma = \pm 1/2$ na banda de condução enquanto $c^{\dagger}_{d\sigma}$ cria um elétron no orbital da impureza; h.c. é o hermitiano conjugado.

Como o caso de uma impureza isolada já foi suficientemente estudado, o interesse atual se volta para aglomerados de impurezas. O modelo de duas impurezas, que é o tratado no presente trabalho, é o mais simples que leva em consideração a interação entre elas. Essa interação não é direta e sim mediada pelos elétrons da banda de condução. O Hamiltoniano de Anderson de duas impurezas [4], extensão do Hamiltoniano de uma impureza, é escrito como:

$$\begin{aligned} H \;=\; & \sum_{\vec{k}\sigma} \varepsilon_{\vec{k}} c^{\dagger}_{\vec{k}\sigma} c_{\vec{k}\sigma} + \varepsilon_d \sum_{\sigma_i} c^{\dagger}_{d\sigma_i} c_{d\sigma_i} + V \sum_{\vec{k}\sigma_i} (e^{i\vec{k}\vec{R}_i} c^{\dagger}_{\vec{k}\sigma} c_{d\sigma_i} + \text{h.c.}) + \\ & + U \sum_{i} c^{\dagger}_{d\uparrow_i} c_{d\uparrow_i} c^{\dagger}_{d\downarrow_i} c_{d\downarrow_i} \end{aligned} \quad (2.2)$$

onde o índice i indica a impureza. Uma representação esquemática do modelo pode ser vista na figura 2.2, semelhante à figura 2.1.

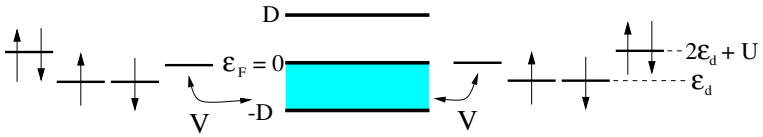

Figura 2.2: Representação esquemática do modelo de Anderson de duas impurezas. Os elétrons da banda de condução mediam a interação entre as impurezas, ou seja, elas não interagem diretamente. Fonte: [15].

O problema de duas impurezas tem simetria de inversão: o Hamiltoniano não muda quando se trocam as posições das impurezas. Como conseqüencia, os auto-estados, e em particular os estados da banda de condução, podem ser classificados segundo sua paridade. Estados pares não trocam de sinal sob inversão e ímpares trocam de sinal. Como dito anteriormente, os elétrons de condução mediam a interação entre as impurezas, e essa interação é denominada RKKY [6, 7, 8]. Seu efeito é ordenar os spins da impureza, enquanto que o efeito Kondo [1] tende a quebrar esse ordenamento blindando os spins. A formação do estado fundamental depende fortemente da competição entre esses dois efeitos.

2.2 Método do Grupo de Renormalização

O Hamiltoniano de Anderson foi analisado inicialmente na literatura por meio de cálculos perturbativos que mostravam divergências logarítmicas com o decréscimo da temperatura. O primeiro tratamento capaz de descrever o espectro completo de temperaturas foi o grupo de renormalização numérico. Esse método permitiu o entendimento dos processos físicos que ocorrem em temperaturas bem baixas e também a transição entre duas situações bem distintas, as regiões de *crossover*.

No método do grupo de renormalização, desenvolvido por Wilson

inicialmente para o modelo de Kondo [1] e depois aplicado ao modelo de Anderson por Krishna-murthy e co-autores [2], o contínuo de níveis de energia da banda de condução é substituído por um conjunto de níveis discretos, distribuídos de forma logarítmica.

Neste método numérico é diagonalizada iterativamente uma matriz real e simétrica equivalente ao Hamiltoniano quântico do modelo, que é diagonal por blocos quando escrita em uma base apropriada, e assim o processo se reduz à diagonalização de dezenas de matrizes menores.

Depois de aplicadas várias transformações do método, o Hamiltoniano iterativo satisfaz à seguinte relação de recursão:

$$H_{N+1} = \Lambda^{1/2} H_N + \xi_N H_{NI} \tag{2.3}$$

onde o índice $N+1$ é a iteração corrente, Λ é a constante de discretização, ξ_N é um coeficiente dependente de Λ e os elementos da matriz H_{NI} são calculados com os elementos de matriz invariante $< ||f^{\dagger}|| >$. Para detalhes das transformações do método e do procedimento iterativo ver [2], assim como a determinação do H_0 necessário para iniciar a relação de recursão.

Usamos nesse trabalho uma extensão do método tradicional pois existe um parâmetro z a mais na discretização, como definido em [16, 17]. Dessa forma a banda é dividida em intervalos de larguras proporcionais a Λ^{-n-z}, com $n = 1, 2, 3, ...$, $\Lambda > 1$ e $0 < z \leq 1$, como mostrado na figura 2.3. O espectro contínuo é reproduzido quando $\Lambda = 1$ e a discretização de Wilson quando $z = 1$.

As adaptações do Hamiltoniano de duas impurezas ao grupo de renormalização, a extensão da discretização logarítmica e uma análise mais detalhada do modelo podem ser encontradas em [15, 18].

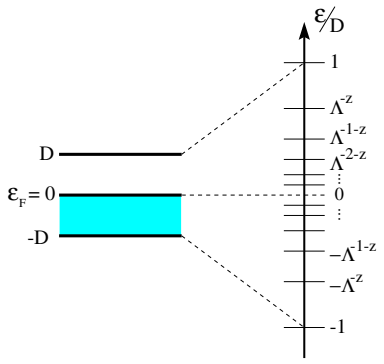

Figura 2.3: Representação esquemática da discretização da banda de condução. A banda de condução se caracteriza por duas energias, uma mínima ε_{min} e outra máxima ε_{max}. Admitindo que a banda esteja semi-cheia, ou seja, que o nível de Fermi esteja em $\varepsilon_F = (\varepsilon_{min} + \varepsilon_{max})/2$, costuma-se medir a energia em relação a ε_F, de forma que a banda passa a se estender desde $-D$ até D, onde a meia largura $D \equiv (\varepsilon_{max} - \varepsilon_{min})/2$. Fonte: [15].

2.3 Processo Iterativo

O número de estados da base em que o Hamiltoniano deve ser projetado torna-se em poucas iterações grande demais para ser manuseado. É assim necessário limitar o número de estados efetivos, o que se consegue descartando estados com energia acima de um limite conhecido como *truncamento ultravioleta*. Com isso, cada iteração cobre uma janela de energia que se estende de $\Lambda^{-(n+1)-z}$ a Λ^{-n-z}, e à medida que o número de iterações cresce a superposição dessas janelas vai dando acesso a todo o espectro de energias, como pode ser visto na figura 2.4.

O Hamiltoniano é escrito de forma tal que seja bloco diagonal, fazendo com que a diagonalização seja efetuada em várias matrizes menores ou blocos. Cada um desses blocos pode ser visualizado como um elemento de uma matriz tridimensional (fig. 2.5), e é rotulado pelo valor de carga (q), *spin* (s) e paridade (p), definindo *setores* qsp.

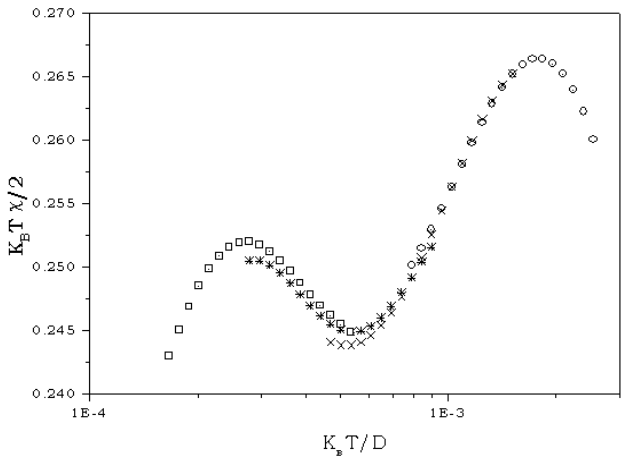

Figura 2.4: Gráfico de $K_BT\chi/2$ *vs* K_BT/D para duas impurezas evidenciando a janela de energias coberta na iteração. Apresentamos somente duas iterações, e portanto quatro fases representadas pelas quatro curvas distintas.

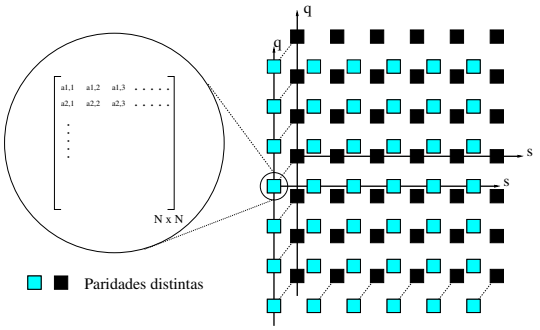

Figura 2.5: Representação esquemática da organização dos blocos em carga, *spin* e paridade. Cada bloco é denominado *setor* e rotulado como *qsp*.

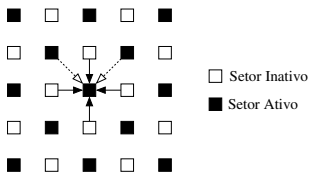

Figura 2.6: Setores ativos e inativos em uma fase (simplificado eliminando a paridade) evidenciando as comunicações entre setores.

Cada iteração é dividida em duas fases como descrito em [15], chamadas de *alta* e *baixa*, que se referem a uma variação no parâmetro z. Somente metade dos setores qsp estão efetivamente executando cálculos em cada fase, e serão chamados de setores *ativos*. Os setores ativos e inativos ficam dispostos como que em um tabuleiro de xadrez, como esquematizado na figura 2.6. Neste ponto podemos ressaltar um dos aspectos importantes para paralelização: somente os setores ativos na fase poderão executar em paralelo.

Podemos resumir cada fase da seguinte forma:

- parâmetros da execução, como z a ser utilizado e fase atual, são ajustados;
- setores vizinhos no norte, sul, leste e oeste (fig. 2.6 setas com linha cheia) são consultados para verificar a geração de novos estados;
- construção do Hamiltoniano usando valores do setor calculados na última fase em que este esteve ativo, tais como autovalores e elementos de matriz invariante (estes últimos são armazenados em estruturas que chamaremos de *páginas* seguindo [15]);
- diagonalização do Hamiltoniano, calculando os novos autovalores e autovetores do setor;

- cálculo das novas páginas usando seus autovalores e autovetores juntamente com valores já calculados por setores vizinhos no noroeste e nordeste (fig. 2.6 setas com linha tracejada);
- valores termodinâmicos de interesse são calculados e escritos em arquivo.

Existe uma iteração preliminar tratada à parte, chamada de iteração -1, referente somente às duas impurezas e que representa o H_0 citado na secção anterior. A partir dela todas seguem o algoritmo descrito acima. A paridade à qual os pais ou vizinhos de um setor pertencem depende da fase que está sendo executada.

2.4 Demandas Computacionais

O tempo tomado para os cálculos neste processo iterativo depende da precisão desejada e é um grande limitante no avanço das pesquisas. São necessárias algumas execuções com variação nos parâmetros de entrada para eliminar oscilações inerentes ao método, além de experimentação para determinar alguns parâmetros como energia de corte. Como já foram realizadas execuções que consumiram três semanas fica claro que este tempo pode se tornar proibitivo.

Além do fator tempo, outro limitante em execuções que requerem precisão alta é a memória dos computadores, visto que nesses casos as matrizes a serem manipuladas são bastante grandes.

Vê-se portanto que o problema necessita de processamento de alto desempenho, que é o tópico do próximo capítulo.

Capítulo 3

Processamento Paralelo

3.1 Características de Processamento de Alto Desempenho

Temos acompanhado um crescimento explosivo na performance e capacidade dos computadores, e isso se deve principalmente ao avanço da tecnologia VLSI (*Very Large-Scale Integration*), pois viabiliza um crescimento na taxa de ciclo de *clock* (tempo requerido para realizar a operação mais primitiva em um processador) e a integração de um número maior de componentes em um *chip*.

A arquitetura de computadores transforma o potencial bruto da tecnologia em desempenho e capacidade cada vez maiores dos sistemas computacionais. O *paralelismo* exerce um papel central nesse cenário. Um grande volume de recursos significa que mais operações podem ser executadas de uma vez, em paralelo. A arquitetura de computadores paralelos organiza esses recursos para que trabalhem bem juntos. Dados manipulados a taxas cada vez maiores devem ser guardados em algum lugar da máquina e por isso o armazenamento é também muito importante. A história de proces-

samento paralelo é profundamente entrelaçada com localidade de dados e comunicação.

Paralelismo é uma perspectiva fascinante pela qual entender arquitetura de computadores porque é aplicado em todos os níveis de projeto, interage com essencialmente todos os outros conceitos de arquitetura, e depende unicamente da tecnologia básica. Em particular, necessidades de localidade, largura de banda, latência e sincronismo surgem em vários níveis no desenvolvimento de sistemas paralelos.

A definição de Almasi e Gottlieb [19] representa bem os elementos de uma arquitetura paralela: *"um computador paralelo é um conjunto de elementos de processamento que se comunicam e cooperam para resolver rapidamente grandes problemas."*

Essa definição é extensa o bastante para incluir supercomputadores paralelos com centenas ou milhares de processadores, redes de computadores e *workstations* com múltiplos processadores. Computadores paralelos são interessantes porque oferecem a possibilidade de concentrar recursos em problemas computacionais (quer processadores, memória ou largura de banda de entrada/saída).

Arquitetura de computadores, tecnologia e aplicações evoluem juntas e têm uma forte interação. Arquitetura de computadores paralelos não é exceção e incorpora o aspecto adicional do número de processadores. Vamos considerar as características de performance individual de processadores. A figura 3.1 obtida de [20] ilustra o crescimento no desempenho de processadores ao longo do tempo para várias classes de computadores. As extensões (em linha tracejada) das linhas representam uma extrapolação simples delas. Devemos ser cuidadosos antes de fazermos conclusões quantitati-

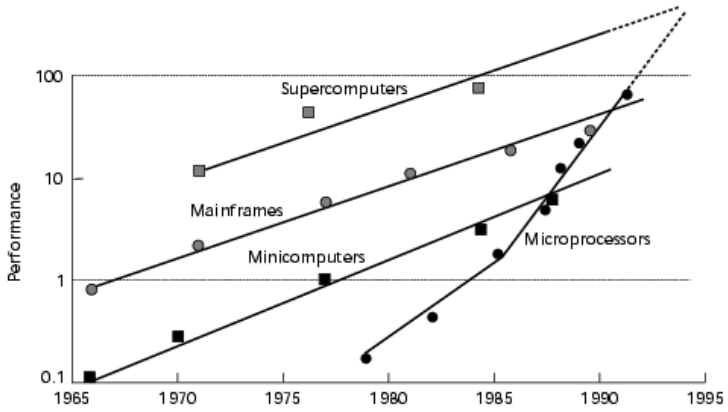

Figura 3.1: Tendências de desempenho de microprocessadores, minicomputadores, *mainframes* e supercomputadores ao longo do tempo. Fonte: [20].

vas de um conjunto de dados limitado, porém podemos fazer uma valiosa observação. A performance dos microprocessadores comuns está crescendo a uma taxa de 50% ao ano e superando as alternativas maiores e mais caras. As vantagens em usar processadores pequenos, baratos, de capacidade moderada e produzidos em massa, como os blocos de construção para sistemas de computadores é intuitivamente lógica. Uma máquina paralela com centenas deles pode satisfazer as necessidades que seriam atendidas por um processador individual em cerca de 10 anos, se mantida a taxa de crescimento atual.

As tendências sugerem que será cada vez mais difícil esperar que os processadores individualmente se tornem rápidos o suficiente para a aplicação, fazendo com que a arquitetura paralela se torne cada vez mais atraente. Sob várias perspectivas - econômica, tecnológica, arquitetônica e demanda de aplicações - vemos que a arquitetura paralela é crescentemente atrativa e central.

A grande demanda por desempenho coloca exigências cada vez maiores na arquitetura e faz com que arquiteturas paralelas se-

jam cada vez mais necessárias para atendê-las. Atualmente uma boa opção para obter desempenhos significativamente maiores são múltiplos processadores, e as aplicações mais exigentes são escritas como programas paralelos [12].

Desenvolvimentos mais recentes na velocidade dos processadores comerciais e na tecnologia de redes levaram à sua adoção na implementação de sistemas paralelos. Um caso extremo é representado pelo uso de *cluster* para processamento paralelo, como nos projetos NOW [21] e Beowulf [13], em que também a rede de interconexão utilizada para a comunicação entre os nós é comercial. Neste tipo de sistema vantagens como preço, facilidade de expansão e manutenção são claras. Discutiremos mais sobre esse tipo de sistema na próxima secção.

3.2 *Clusters* de Computadores

Os baixos preços dos computadores pessoais, que são fabricados em massa, e o avanço na velocidade das redes de comunicação fizeram com que *clusters* de computadores tenham emergido como boa opção de sistemas para alto desempenho a baixo custo.

Um *cluster* de computadores é um sistema que compreende um conjunto de computadores independentes e uma rede interconectando-os. A rede de um *cluster* é dedicada à integração dos nós e é separada do "mundo externo". Um *cluster* do tipo Beowulf [13] é um sistema cujos nós são computadores pessoais interligados por uma rede local de computadores ou uma rede do próprio sistema, e abrigando um sistema de código aberto do tipo UNIX como sistema operacional de cada nó.

Clusters do tipo Beowulf são viáveis porque não necessitam de

projetos de desenvolvimento longos e caros anteriores ao seu uso inicial. Mesmo os primeiros sistemas desse tipo apresentavam uma relação de custo-benefício vantajosa quando comparados com os supercomputadores contemporâneos. Ainda permitem uma flexibilidade de configuração pouco encontrada em outros sistemas de processamento paralelo maciço. Número de nós, capacidade de memória por nó, número de processadores por nó e topologia de interconexão são todos parâmetros da estrutura que podem ser especificados unicamente para um sistema particular sem custos adicionais. Além disso, a estrutura do sistema pode ser facilmente alterada ou aumentada à medida da necessidade. Também permitem uma rápida resposta aos avanços tecnológicos.

3.3 Convergência de Arquiteturas Paralelas

A rápida difusão de computadores no comércio, ciência e educação deve-se à primeira padronização de um modelo de máquina simples, o *computador de von Neumann*, que consiste de uma unidade central de processamento (CPU) conectada a uma unidade de armazenamento (memória). Este modelo simples permite aos programadores projetarem algoritmos para uma máquina abstrata de von Neumann e não para uma máquina específica. Um desenvolvimento semelhante era necessário para a difusão de arquiteturas paralelas.

Examinando a evolução das principais arquiteturas paralelas, notamos uma recente convergência para máquinas escaláveis através de uma máquina paralela genérica [12]. Esta máquina genérica, o *multicomputador*, deve ser simples para facilitar a compreensão e programação, e realista para assegurar que programas desenvolvidos para ela executem com razoável eficiência em computadores

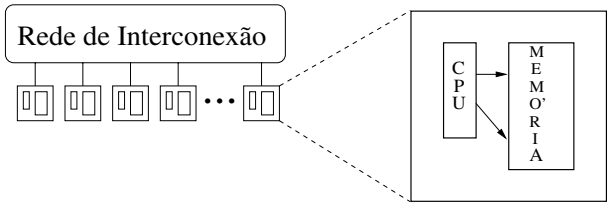

Figura 3.2: Organização de um multicomputador genérico.

paralelos reais, sejam de memória compartilhada ou distribuída.

Um multicomputador compreende um conjunto de computadores de von Neumann (ou nós) interligados por uma rede como ilustra a figura 3.2. Cada computador executa seu próprio programa, que pode acessar a memória local e enviar e receber mensagens pela rede. As mensagens são usadas para a comunicação com outros computadores ou equivalentemente ler e escrever em memórias remotas (localizadas em outro nó).

Um atributo do modelo de multicomputador é que acessos à memória local são menos custosos que à memória remota, isto é, leitura e escrita são menos custosos que envio e recepção. Portanto é desejável que acessos a dados locais sejam mais freqüentes que acessos a dados remotos.

3.4 Modelo de Programação Paralela

A programação paralela introduz fontes adicionais de complexidade em relação à programação seqüencial. Não apenas o número de instruções executadas aumenta como também é necessário gerenciar explicitamente a execução em centenas de computadores e coordenar milhões de comunicações entre eles.

O modelo de *passagem de mensagens* pode ser utilizado juntamente com a máquina genérica descrita na secção anterior para

satisfazer exigências da programação paralela, tendo sido empregado no desenvolvimento dos programas deste trabalho. Existem outros modelos que não serão discutidos aqui como tarefas e canais, paralelismo de dados e memória compartilhada, apresentados em [22].

Passagem de Mensagem

A passagem de mensagem é um modelo de programação paralela amplamente utilizado. Os programas de passagem de mensagens criam múltiplas tarefas, cada uma delas encapsulando dados locais. Cada tarefa é identificada por um nome próprio e interage com outras através do envio e recepção de mensagens para e de outras tarefas.

Na abordagem de passagem de mensagens, uma coleção de programas pode ser escrita em uma linguagem seqüencial padrão complementada com chamadas para uma biblioteca com funções para envio e recepção de mensagens.

O paradigma de passagem de mensagem é muito atrativo devido à fácil portabilidade de programas que o utilizam. Ele é compatível tanto com sistemas de memória compartilhada quanto com de memória distribuída, sendo que a passagem de mensagem não se tornará obsoleta com a renovação do sistema.

O MPI, *Message Passing Interface* [23], é um padrão de biblioteca de passagem de mensagens para sistemas paralelos. Foi desenvolvido para fornecer uma base comum de desenvolvimento de programas paralelos em plataformas distintas. Em virtude disso, ele incorpora aspectos de variados sistemas ao invés de adotar como padrão um sistema específico. Podemos utilizar a biblioteca MPI para especificar a comunicação entre conjuntos de processos

formando um programa concorrente.

De acordo com a intenção do comitê de padronização, o MPI Forum [24], o padrão especifica apenas uma interface de programação e sua concretização em algumas linguagens, correntemente C, Fortran 77 e C++. Detalhes de implementação são deixados por conta do implementador, de forma a flexibilizar o sistema e possibilitar implementações eficientes.

Discutimos no apêndice C detalhes técnicos do MPI e também as rotinas que foram utilizadas na confecção dos programas.

3.5 Estrutura do Nosso Sistema

O Grupo de Instrumentação e Informática do IFSC tem trabalhado com aplicações de processamento paralelo, inclusive para problemas de física tais como dinâmica molecular [25, 26]. Seguindo a tendência de montagem de sistemas paralelos de baixo custo conforme discutido na secção 3.2, foi montado em nosso laboratório um *cluster* de dezesseis máquinas do tipo computador pessoal mostrado na figura 3.3. Cada uma das máquinas apresenta a seguinte configuração: processadores K6-III 450 MHz com 256 MBytes de memória, sistema próprio, área de *swap* e espaço em disco. São interligados por um *hub* Ethernet e uma *switch* FastEthernet. A conexão Ethernet é usada para montar arquivos de sistema do usuário e para administração, e a FastEthernet para a comunicação das aplicações paralelas.

Para o funcionamento desse sistema foram utilizados *softwares* de distribuição livre, basicamente o sistema operacional Linux (kernel 2.2) [27], com as ferramentas de programação que são distribuídas com o mesmo [28] (compilador GCC, depurador GDB, make,

Figura 3.3: Foto do *cluster* existente no laboratório.

entre outros) e a biblioteca de passagem de mensagem padrão MPI desenvolvida no Argonne National Laboratory (MPICH) [29].

Este sistema foi utilizado para o desenvolvimento dos programas e obtenção dos dados para análise, discutidos nos capítulos a seguir.

Capítulo 4

Descrição das Implementações

Os programas foram escritos em linguagem C++ [30] seguindo os princípios de Programação Orientada a Objetos, e para a comunicação entre os processos usamos uma biblioteca MPI [23, 29].

Iniciamos o trabalho com um estudo do programa seqüencial utilizado pelo grupo de Física Teórica do IFSC, já citado no capítulo 1. Nesse estágio acreditamos ser necessária a reescrita do programa já visando a paralelização. A nova versão seqüencial é descrita na secção 4.1.

Implementamos então uma paralelização inicial baseada em variação de parâmetros pois, como discutido na secção 2.3, é necessário que o programa seja executado repetidas vezes. Essa implementação foi realizada sem a necessidade de grandes alterações no programa seqüencial e é apresentada na secção 4.2.

Para eliminar limitações desta paralelização inicial foi implementada outra, baseada na execução paralela dos cálculos dos setores, secção 4.3.

Como último passo integramos as versões paralelas anteriores conseguindo uma versão híbrida descrita na secção 4.4. Essa versão realiza paralelamente os diversos cálculos com variação nos pa-

râmetros de entrada, e esses cálculos são realizados com o uso da paralelização dos cálculos dos setores.

4.1 Programa Seqüencial

O programa seqüencial utilizado pelo grupo de Física Teórica do IFSC foi detalhadamente estudado para que pudéssemos encontrar os pontos de concorrência e analisar o custo das comunicações envolvidas. Porém, como os programadores não visavam a paralelização, os dados estavam todos concentrados em várias matrizes que abrangiam todo o espaço, o que a dificultaria. Além disso, as funções eram bastante entrelaçadas e pouco otimizadas. Então acreditamos ser necessária a reescrita desse código de forma que facilitasse a paralelização.

Foram então escritas várias classes para armazenar e tratar os dados. Os elementos da matriz da figura 2.5 são representados por objetos da classe *Setor*, que estão dispostos em uma matriz tridimensional pertencente à classe *Espaco* que gerencia as vizinhanças entre os setores qsp e suas ações. Outros membros do *Espaco* são objetos de várias classes que tratam dos parâmetros da execução (iteração, fase), do modelo (separação das impurezas, interação Coulombiana), da banda de condução, de valores termodinâmicos entre outras, e ainda é esta classe que cuida do cálculo e impressão dos resultados físicos de interesse utilizando dados de todos os setores. A classe *Setor* apresenta por sua vez membros de outras classes, como a que trata dos autovalores e autovetores, das páginas ativa e passiva e do espaço primitivo (armazena valores para indicar a geração de novos estados na fase). Várias classes foram construídas utilizando uma classe básica *Vetores*. Discriminamos

```
Leitura dos dados de entrada;
Construção de objetos;
Iteração - 1; // Ajuste dos valores iniciais para execução
Enquanto(não acabam as iterações)
{ Ajusta parâmetros para iniciar a iteração;
  Executa a fase alta;
  Ajusta parâmetros para iniciar fase baixa;
  Executa a fase baixa;
}
```

Figura 4.1: Algoritmo do programa principal da versão seqüencial, em pseudo-código.

todas as classes escritas no apêndice A, onde são melhor discutidas.

O algoritmo do programa principal em pseudo-código pode ser descrito como na figura 4.1.

O algoritmo do ajuste dos parâmetros para iniciar iteração ou fase atualiza por exemplo iteração e fase correntes, troca página ativa com passiva, quais são os setores ativos e passivos na fase, z utilizado, entre outros. Já o algoritmo de execução da fase é mais complexo e o apresentamos em pseudo-código na figura 4.2, onde a matriz tridimensional de setores apresenta como dimensões: carga variando de q_{lim} a $-q_{lim}$, *spin* de 0 a s_{lim}, e paridade 0 a 1.

A varredura do setores qsp é feita por um método da classe *Espaco* que realiza chamadas para métodos dos vários setores para a realização dos cálculos.

Com a reescrita do código seqüencial, além de adquirir maior conhecimento sobre o programa, conseguimos reduzir sensivelmente a cópia de dados e simplificar algumas funções. Com essas mudanças conseguimos uma redução no tempo de execução de aproximadamente 70% do original, como pode ser visto nas análises dos resultados, secção 5.1.

```
de(carga q variando de q_lim a -q_lim)
  de(spin s variando 0 a s_lim)
    de(paridade p variando de 0 a 1)
      Se(setor qsp é ativo)
      { Apaga as páginas ativa e passiva do setor qsp;
        Se(há geração de novos estados (consulta espaço primitivo))
        { Calcula o Hamiltoniano do setor qsp e diagonaliza;
          Calcula a página ativa e atualiza a passiva do setor qsp;
        }
      }
Apaga autovetores de todos os setores;
Calcula e imprime valores de interesse com dados de todos os setores;
```

Figura 4.2: Algoritmo da execução da fase (alta ou baixa) da versão seqüencial, em pseudo-código.

4.2 Paralelização Baseada em Variação de Dados

Como discutido anteriormente (secção 2.3), uma necessidade no estudo de propriedades físicas de metais utilizando o método numérico escolhido é rodar o programa várias vezes variando parâmetros de execução, tais como z e paridade da fase alta. As propriedades físicas calculadas apresentam oscilações dependentes do parâmetro Λ da discretização sendo essas reduzidas com a variação dos parâmetros, como descrito em [16, 17]. Dessa forma implementamos uma versão paralela inicial que realiza concorrentemente os diversos cálculos.

Nessa implementação o processo mestre faz a leitura dos dados de um arquivo e os envia aos processos escravos variando os parâmetros necessários, como esquematizado na figura 4.3. Os escravos executam essencialmente o mesmo código do programa seqüencial descrito na secção anterior, e quando terminam o trabalho ficam disponíveis e recebem mais dados do mestre para novos cálculos ou um sinal para finalização. Quando todos finalizam, a execução do programa acaba.

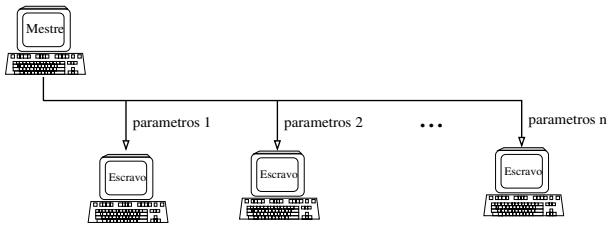

Figura 4.3: Representação esquemática da paralelização baseada em variação de dados. O processo mestre distribui os parâmetros para os cálculos aos escravos que realizam os cálculos.

```
Leitura dos dados de entrada;
Envia os dados comuns para todos os escravos;
Enquanto(há parâmetros para nova execução)
{ Espera um escravo;
  Envia os dados da execução;
}
de(escravo variando do primeiro ao número máximo)
{ Espera um escravo;
  Notifica finalização;
}
```

Figura 4.4: Algoritmo do mestre da paralelização baseada em variação de parâmetros, em pseudo-código.

O algoritmo do mestre em pseudo-código é apresentado na figura 4.4, e para os escravos na figura 4.5.

Os processos envolvidos em uma execução recebem uma numeração que chamamos de *rank* (ver apêndice C), e em uma execução com P processos os *ranks* variam de 0 a $P-1$. Usualmente, denominamos o processo com *rank* 0 mestre e os outros escravos. Note que um processador pode alojar ao mesmo tempo processos mestre e escravos, dependendo de quantos processos rodam simultaneamente por processador e de como são distribuídos.

Para essa distribuição de processos por processadores, fornecemos ao MPI uma lista com os números IP das máquinas envolvidas naquela execução e o número de processos que queremos disparar, e é feita uma distribuição no esquema *Round-Robin* dos processos

```
Recebe os dados comuns do mestre;
Notifica ao mestre que está livre;
Recebe mensagem do mestre;
Se(recebe dados da execução)
{ Construção de objetos;
  Faça
  { Iteração - 1; // Ajuste dos valores iniciais para execução
    Enquanto(não acabam as iterações)
    { Ajusta parâmetros para iniciar a iteração;
      Executa a fase alta;
      Ajusta parâmetros para iniciar fase baixa;
      Executa a fase baixa;
    }
    Notifica ao mestre que está livre;
    Recebe mensagem do mestre;
  }enquanto(recebe dados da execução)
}
```

Figura 4.5: Algoritmo do escravo da paralelização baseada em variação de parâmetros, em pseudo-código.

pelos processadores.

Nessa versão onde temos vários cálculos simultâneos, precisamos gerar saídas diferenciadas para cada um deles para que não haja confusão. Para isso, colocamos um sufixo nos nomes dos arquivos de saída de acordo justamente com os parâmetros variados que são z e paridade da fase alta, criando assim um nome único para cada conjunto de parâmetros.

Como vantagens dessa versão podemos citar que a variação dos parâmetros é feita automaticamente, o *speedup* alcançado é muito bom pois é quase linear se o número de conjuntos de parâmetros for múltiplo do número de processadores, e se esse número de conjuntos de parâmetros for bem maior que o número de computadores temos um bom balanceamento de cargas, ou seja, uma boa distribuição do trabalho pelos processadores, porque ao terminar os cálculos a máquina disponível já recebe outros parâmetros para novos cálculos. No entanto, apresenta algumas limitações como não es-

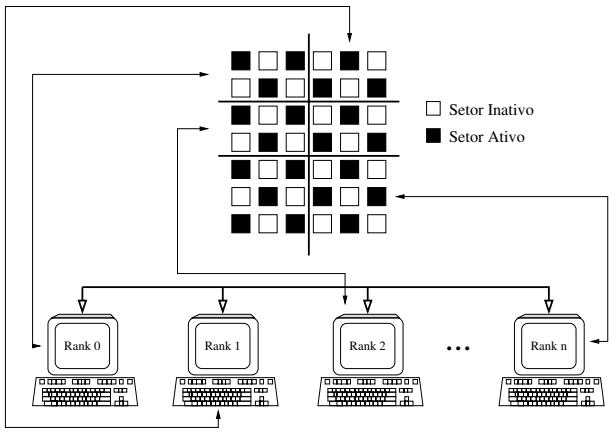

Figura 4.6: Representação esquemática da paralelização por setores com a distribuição dos setores *qsp* pelos processos.

calabilidade, ou seja, não se beneficia com o aumento do número de computadores disponíveis além do número de parâmetros variados (que atualmente tem sido 8). Outro problema marcante é que a precisão dos resultados continua limitada à memória disponível nos computadores utilizados tal como era na versão seqüencial.

4.3 Paralelização por Setores

Para minimizar as desvantagens da versão baseada em variação de parâmetros citadas na secção anterior, tornou-se importante paralelizar a execução de cada conjunto de parâmetros de entrada. Como dito anteriormente, os setores *qsp* podem executar simultaneamente várias partes do código e podemos distribuir esses setores entre processos para trabalharem paralelamente, como ilustrado na figura 4.6.

Há dois tipos de comunicações entre os setores: setores vizinhos na horizontal e vertical (norte, sul, leste e oeste) que *não estão* ativos nessa fase (fig. 2.6 setas com linha cheia) que denominaremos de *"pais"*, e setores vizinhos na diagonal no quadrante superior (no-

roeste e nordeste) que *estão* ativos na fase (fig. 2.6 setas com linha tracejada) que denominaremos *"vizinhos"*.

As comunicações com esses dois tipos de setores lidam com dados diferentes que são relevantes à execução em momentos distintos (ver secção 2.3): os pais são consultados no início da fase pois são utilizados na montagem das matrizes a serem diagonalizadas, e os vizinhos no cálculo das páginas, somente depois que já realizaram sua própria diagonalização.

Para executarem paralelamente, dividimos os setores somente nas dimensão q e s (ver figura 2.5) em blocos retangulares. A dimensão p não foi dividida pois teríamos um número de comunicações demasiadamente grande, além de um aumento indesejável de sincronismo entre os processos. Isso ocorreria porque os pais e os vizinhos não estão necessariamente na mesma paridade que o setor como aparenta a figura 2.6, que não considera essa dimensão por motivo de simplicidade. Para calcular o Hamiltoniano ou as páginas, setores nas duas paridades são consultados, e se esta dimensão fosse dividida, seria obrigatória a passagem de dados entre as paridades antes desses cálculos serem realizados para qualquer setor, aumentando o sincronismo. Da forma em que é feita a divisão dos setores, somente os que estão nas arestas dos blocos precisam esperar dados para seus cálculos.

Foi alocada uma camada a mais de setores nas arestas dos blocos de setores para armazenar os dados comunicados. Esses setores alocados com função única de armazenar dados, não de executar, são chamados de setores *sombras* (setores com hachuras nas figuras 4.7 e 4.8). Dessa forma um setor que está executando cálculos e precisa dos dados de um pai ou vizinho consulta o setor sombra correspondente.

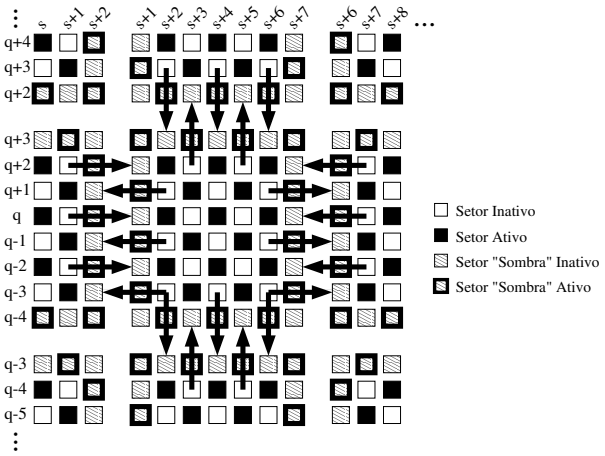

Figura 4.7: Representação esquemática da comunicação de pais entre os processos (sem considerar paridade). Os setores sombras são setores alocados somente para armazenar dados enviados pelos blocos de setores justapostos, não efetuam cálculos.

Os setores inativos em todo o perímetro do bloco enviam dados de pai para os setores sombras inativos equivalentes no bloco justaposto como mostrado na figura 4.7, onde evidenciamos essa comunicação em um só processo, e os cálculos da fase começam. A execução continua até que após a diagonalização do Hamiltoniano os setores ativos no perímetro enviam dados de vizinhos, menos os setores que estão no valor superior de carga, ver figura 4.8, (e por simetria os setores sombra ativos que estão na aresta inferior não recebem dados) visto que os setores consultados são somente os que estão acima do próprio setor. Ao receberem os vizinhos necessários, os setores continuam sua execução, realizando o cálculo das páginas e finalizam os cálculos da fase.

O algoritmo do programa principal para essa versão é o mesmo que para a versão seqüencial, figura 4.1. Nos ajustes dos parâmetros para executar a iteração ou fase agora é também realizado o envio de pais de todas as arestas do bloco de setores. O algoritmo que executa a fase deve cuidar das comunicações dos vizinhos, e

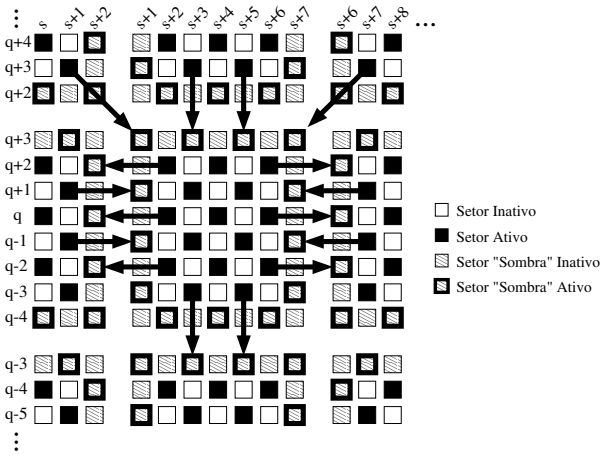

Figura 4.8: Representação esquemática da comunicação de vizinhos entre os processos (sem considerar paridade).

além disso, cada processo trabalha com um bloco de setores, com determinados valores de carga e *spin*, que denominaremos de q_{max}, q_{min}, s_{max} e s_{min}. O algoritmo para execução da fase é apresentado na figura 4.9.

Os setores que estão nas arestas executam os cálculos das páginas depois dos que estão internos ao bloco para que não haja sincronismo nesse estágio. Se realizassem os cálculos juntamente com os outros, deveriam esperar dados dos blocos vizinhos.

Pela descrição da figura 4.9, podemos perceber que, apesar de várias execuções serem feitas simultaneamente, os setores ficam sincronizados em alguns instantes. Além desse fato, alguns poucos setores trabalham por tempos muito maiores que outros, dificultando o balanceamento de carga e a localidade dos dados. A tabela 5.3 mostra essa discrepância no tempo de execução para setores distintos.

Cada processo calcula com qual bloco de setores vai trabalhar usando o número de seu *rank* como mostra a figura 4.6. Assim, se a divisão dos setores resultar em quatro colunas, o de *rank* 0 fica

```
de(carga q variando de q_max a q_min)
  de(spin s variando s_min a s_max)
    de(paridade p variando de 0 a 1)
      Se{setor qsp é ativo)
      { Apaga as páginas do setor qsp;
        Se(há geração de novos estados (consulta espaço primitivo))
        { Calcula o Hamiltoniano do setor qsp e diagonaliza;
          Se(setor qsp não está nas arestas)
            Calcula a página ativa e atualiza passiva do setor qsp;
        }
        Se(setor qsp nas arestas)
          Envia dados de vizinho para blocos correspondentes;
    }
Recebe dados de vizinho;
de(carga q variando de q_max a q_min)
  de(spin s variando s_min a s_max)
    de(paridade p variando de 0 a 1)
      Se(setor é ativo e está nas arestas)
        Calcula a página ativa e atualiza a passiva;
Apaga autovetores de todos os setores;
Calcula e imprime valores de interesse.
```

Figura 4.9: Algoritmo da execução da fase (alta ou baixa) da paralelização por setores, em pseudo-código.

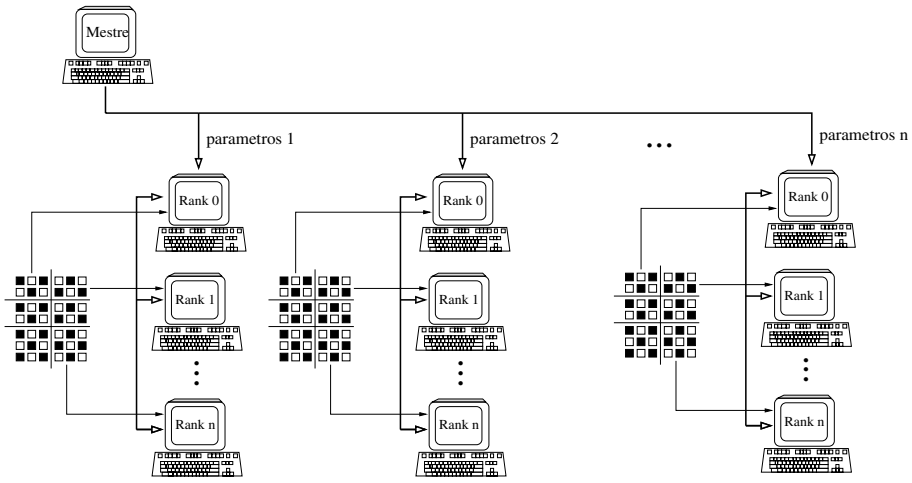

Figura 4.10: Representação esquemática da paralelização híbrida. O processo mestre distribui os parâmetros para os cálculos e estes são compartilhados nos sub-grupos de processos para que realizem os cálculos usando a paralelização por setor.

com o bloco na posição $(1,1)$, o de *rank* 1 com o $(1,2)$, *rank* 2 com o $(1,3)$, *rank* 3 com o $(1,4)$, *rank* 4 com o $(2,1)$, e assim por diante até o último bloco.

4.4 Versão Híbrida

As duas opções de paralelização descritas nas secções 4.2 e 4.3 foram integradas, resultando em uma versão paralela híbrida do nosso programa. Nessa versão os processos se dividem em grupos (para isso são criados novos comunicadores, ver apêndice C), e o processo mestre distribui os parâmetros da execução para os *"sub-mestres"* desses grupos, que os compartilham com seus companheiros para realizarem os cálculos usando a paralelização por setores, conforme esquema da figura 4.4.

Os processos se dividem pelos grupos usando o valor de seu *rank* e dentro desses grupos os processos recebem mais um valor

```
Recebe os dados comuns do mestre;
Se(for sub-mestre)
{ Notifica ao mestre que está livre;
  Compartilha com os companheiros a mensagem do mestre;
}
Senão // não é sub-mestre
  Recebe os dados do sub-mestre;
Se(receberam dados da execução)
{ Construção de objetos;
  Faça
  { Iteração - 1; // Ajuste dos valores iniciais para execução
    Enquanto(não acabam as iterações)
    { Ajusta parâmetros para iniciar a iteração;
      Executa a fase alta;
      Ajusta parâmetros para iniciar a fase;
      Executa a fase baixa;
    }
    Se(for sub-mestre)
    { Notifica ao mestre que está livre;
      Compartilha com os companheiros a mensagem do mestre;
    }
  }enquanto(recebem dados da execução)
}
```

Figura 4.11: Algoritmo para os escravos da paralelização híbrida, em pseudo-código.

de *rank* associado que varia de 0 ao número de processos no grupo. A exemplo da nomeação do mestre, os sub-mestres são os processos que têm *rank* 0 dentro do seu grupo.

O algoritmo do mestre é igual ao da paralelização baseada na variação de dados e descrito na figura 4.4, porém a comunicação é realizada apenas com os sub-mestres. O algoritmo usado pelos escravos muda, como mostrado na figura 4.11.

O algoritmo de execução da fase é o mesmo mostrado na figura 4.9 que descreve a paralelização por setores.

Nessa versão a precisão não fica limitada à memória da máquina como ocorre com a paralelização baseada em variação de parâmetros.

4.5 *Checkpointing*

Adicionamos a possibilidade de criação de arquivos de *checkpoint* que são escritos periodicamente e contêm todas as informações necessárias para reiniciar a execução a partir do ponto em que foi salvo. Estes servirão para minimizar perdas ao ocorrer um contratempo, tal como falta de energia elétrica, pane em um dos computadores ou problemas com as ventoinhas que ajudam na refrigeração dos processadores. Esses contratempos podem ser minimizados com o uso de *nobreaks* e lubrificação das ventoinhas, por exemplo, porém continuam freqüentes, e em execuções anteriores a esse trabalho faziam com que dias ou até semanas de execução fossem perdidos.

Depois de um número determinado de iterações escolhido pelo usuário, cada processo envia os dados relativos aos setores que lhe pertencem e que devem ser enviados ao processo mestre da execução. Esse processo que recebe os dados de todos os setores os escreve em arquivos, e para diferenciá-los utiliza como sufixo o rótulo qsp dos setores. Os dados salvos dos setores são os autovalores, as páginas ativa e passiva e alguns *flags* utilizados para sinalizar ações ocorridas ou não no decorrer dos cálculos. Ainda é criado um arquivo geral onde são salvos alguns parâmetros da execução, tal como iteração e fase correntes, dados relativos à banda de condução, entre outros.

A escrita dos arquivos não é feita diretamente no diretório "oficial", ou seja, no diretório que o programa procuraria para ler os dados salvos e reiniciar sua execução. Primeiramente, os arquivos são escritos em um diretório temporário até que todos sejam escritos. Então o diretório oficial antigo (com arquivos salvos de uma iteração anterior) é renomeado para um *backup* e então o diretório

temporário torna-se o oficial. Essa manobra evita que haja arquivos desatualizados em relação a outros no mesmo diretório se porventura algum problema ocorrer antes da escrita acabar. Os diretórios descritos acima também recebem um sufixo igual ao citado na secção 4.2, constituído do valor do parâmetro z utilizado e da paridade da fase alta, para poderem ser diferenciados de diretórios de outros conjuntos de dados.

Capítulo 5

Análise dos Resultados

Para analisarmos o desempenho do nosso programa utilizamos três conjuntos de dados que diferiam entre si no tempo de processamento, e os rotulamos por *pequeno, médio* e *grande*. Esses conjuntos estão listados no apêndice B.

Para as execuções paralelas usamos o *cluster* de computadores existente em nosso laboratório (descrito na secção 3.5). Os processos são distribuídos pelos processadores de acordo com um esquema *Round-Robin* usando uma lista fornecida ao MPI com os números IP dos computadores que participarão de determinada execução.

Apesar do *cluster* apresentar dezesseis computadores disponíveis, apenas quinze são absolutamente iguais e não utilizamos o de configuração diferente para que este não influenciasse nas execuções, seqüenciais ou paralelas. A diferença existente é a presença de uma placa de vídeo enquanto que todos os outros apresentam vídeo *onboard*. Conforme analisado em nosso laboratório, essa diferença na configuração pode alterar o desempenho dos processadores em até 20% [31].

Todos os tempos tabelados estão em *horas* e foram obtidos com a

função `gettimeofday`. Asseguramo-nos de que nenhuma outra aplicação competia com a nossa ao tomarmos as medidas de tempo. Devido ao fato de não utilizarmos um computador dedicado o sistema operacional e alguns *daemons* rodam concomitantemente com nossas execuções, e assim temos uma certa variação nos tempos de execução.

Não foram feitas medidas suficientes para realizarmos um levantamento estatísitico e apresentarmos medidas com desvios-padrão. Isso porque, como será mostrado nas secções seguintes, algumas execuções demoraram várias horas e não teríamos tempo hábil para repetí-las o número de vezes necessário para tal. Os tempos apresentados são considerados significativos pois repetimos algumas execuções por três vezes e verificamos uma variação de até 10% nas medidas.

O *speedup* em P processadores é como o definido em [12] para um problema fixo, no qual a performance da máquina é simplesmente o inverso do tempo tomado pela execução do problema.

$$Speedup(P \text{ processadores}) = \frac{\text{Tempo } (1 \text{ processador})}{\text{Tempo } (P \text{ processadores})}$$

onde o tempo em um processador é o tempo tomado pelo programa seqüencial (*speedup* absoluto).

5.1 Programa Seqüencial

Na tabela 5.1 apresentamos os tempos de execução do programa seqüencial usado anteriormente e da nova versão, descrita na secção 4.1, para os três conjuntos de dados e a porcentagem de redução alcançada com a reescrita do programa.

A sensível redução no tempo de processamento mostrada na tabela 5.1 ocorre igualmente para os três conjuntos de dados utiliza-

Tabela 5.1: Tempos de execução dos programas seqüenciais, versões original e nova, para os três conjuntos de dados

	Tempo (h)		
	Original	*Nova*	*% Redução*
Pequeno	0.70	0.24	66
Médio	20	6.7	67
Grande	65	21	68

dos. Esta redução é devida principalmente à diminuição de cópia de vetores elemento a elemento, que era bastante comum em vários trechos do programa anterior, além de otimizações em algumas funções.

5.2 Paralelização Baseada em Variação de Dados

Executamos o programa discutido na secção 4.2 usando oito conjuntos de dados de entrada e os quinze computadores idênticos do *cluster* do nosso laboratório. A tabela 5.2 apresenta o tempo de execução desta versão paralela e também a soma de oito execuções da versão seqüencial (os oito conjuntos de dados são diferenciados pelo sufixo que é utilizado para gerar as diferentes saídas).

Note que apesar de termos quinze computadores disponíveis, somente oito estão efetivamente realizando cálculos justamente pelo fato de usarmos oito conjuntos de dados de entrada. Dessa forma, utilizando apenas oito processadores conseguiríamos o mesmo *speedup*, que neste caso seria quase linear.

É fácil perceber que um *speedup* quase linear é alcançado quando o número de conjuntos de dados for múltiplo do número de com-

Tabela 5.2: Tempos de execução para oito conjuntos de dados da paralelização baseada em variação de dados e da versão seqüencial, para o problema médio.

Versão		*Tempo (h)*	
Seqüencial	1000AP	6.5	
	1000AI	6.8	
	0875AP	6.3	
	0875AI	6.5	
	0750AP	5.6	
	0750AI	6.2	
	0625AP	5.5	
	0625AI	5.2	
	TOTAL	49	
Variação de Dados		6.3	*Speedup* 7.8

putadores disponíveis, visto que os escravos executam essencialmente o programa seqüencial. Além disso, essa versão apresenta um bom balanceamento de carga quando o número de conjuntos de dados for bem maior que as máquinas disponíveis porque assim que um escravo termina seus cálculos recebe dados para reiniciar seu trabalho.

5.3 Paralelização por Setores

Para a versão que paraleliza os cálculos por setores descrita na secção 4.3, a distribuição destes pelos processos se mostrou muito importante e decisiva no desempenho do programa. Isso ocorre porque alguns poucos setores executam por tempos milhares de ve-

Tabela 5.3: Tempos de execução total de alguns setores da paralelização por setores, para o problema médio.

Setor (q,s,p)	*Tempo (s)*
0, 2, 1	1539
-1, 1, 0	1117
-2, 1, 0	452
3, 4, 0	1.77
-4, 5, 1	0.05

zes maiores que outros, como mostra a tabela 5.3, onde temos as somas dos tempos totais de execução dos cálculos dos setores listados. Diferentemente das outras tabelas, os tempos são mostrados em *segundos* para evidenciar a discrepância entre eles.

Fixamos o número de processadores em 15, adotamos uma distribuição por linhas e outra por uma granularidade bem fina (discutida a seguir) novamente para o problema médio e mostramos os resultados na tabela 5.4. Com os dados que estamos executando, o valor de q varia de -7 a $+7$, tendo portanto 15 linhas de setores.

Tabela 5.4: Tempos de execução da paralelização por setores para 15 processadores, com duas distribuições diferentes dos setores pelos processadores, para o problema médio.

Divisão	*Tempo (h)*
Por linhas	3.4
Fina	1.8

Depois de várias tentativas para alcançar uma boa distribuição

dos setores pelos processadores para obter um bom balanceamento de cargas, percebemos que uma com granularidade bem fina é em geral boa para os diversos números de processadores. Isso ocorre porque conseguimos um número grande de processos para serem distribuídos pelos processadores e assim os cálculos pesados apresentam uma melhor dispersão, o que ajuda no balanceamento de carga. Apesar de uma distribuição fina necessitar de muito mais comunicação e perder em localidade de dados, seu desempenho não fica prejudicado visto que o tempo gasto nas comunicações é pequeno frente ao tomado pelos cálculos.

A menor granularidade utilizada deixa apenas dois pares de setores adjacentes por processo, (q, s, p) e $(q, s+1, p)$, nas duas paridades, sendo essa granularidade a mais fina desejável pois como os setores ativos estão distribuídos como que em um tabuleiro de xadrez, sempre temos um par de setores executando por processo em cada fase. A partir desse ponto a distribuição utilizada pela versão paralela por setores será sempre a de granularidade bem fina discutida.

Para um conjunto fixo de parâmetros, variamos o número de processadores, e os resultados estão na tabela 5.5.

Não foi possível executar o programa para um número de processadores menor que três pois são muitos processos que devem rodar por processador nesses casos (mais de 20 processos por processador) chegando a uma limitação da versão usada do MPICH (versão 1.2.0).

A queda no *speedup* com quatro e oito processadores verificado na figura 5.1 para os três tamanhos de problema é explicado ao verificarmos a distribuição resultante dos setores pelos processadores, pois há uma concentração de setores bem trabalhosos em

Tabela 5.5: Tempos de execução e *speedup* alcançado com a paralelização por setores, variando o número de processadores e o tamanho do problema. P é o número de processadores.

	Pequeno		*Médio*		*Grande*	
P	*Tempo (h)*	*Speedup*	*Tempo (h)*	*Speedup*	*Tempo (h)*	*Speedup*
03	0.3	0.9	4.0	1.7	12.9	1.6
04	1.2	0.2	7.6	0.9	20.7	1.0
05	0.2	1.0	3.5	1.9	10.8	1.9
06	0.1	1.9	2.1	3.3	6.6	3.1
07	0.1	2.1	2.1	3.2	6.7	3.1
08	0.3	0.9	3.1	2.2	9.5	2.2
09	0.1	2.2	2.1	3.2	6.5	3.2
10	0.1	2.4	1.9	3.5	5.9	3.5
11	0.1	2.3	2.1	3.3	6.4	3.2
12	0.1	2.2	2.1	3.3	6.2	3.4
13	0.1	2.3	2.0	3.4	6.2	3.3
14	0.1	2.7	1.8	3.8	5.6	3.7
15	0.1	2.6	1.8	3.7	5.6	3.7

alguns processadores, prejudicando o balanceamento de carga.

Alteramos a distribuição dos processos pelos processadores para esses dois casos, de forma que o trabalho ficasse melhor distribuído. Os resultados da execução para o problema médio estão na tabela 5.6, que mostra que após as alterações o *speedup* retorna ao mesmo nível dos valores conseguidos para número de processadores próximos, visto na figura 5.2.

Mostramos que podemos melhorar levemente o desempenho do programa alterando a distribuição dos processos pelo processado-

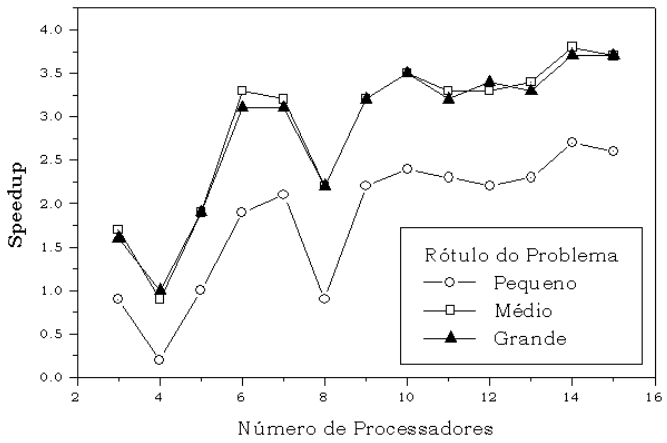

Figura 5.1: Gráfico de *speedup vs* número de processadores obtido da tabela 5.5 para os três problemas. A queda observada para quatro e oito processadores é justificada pela distribuição resultante dos setores pelos processadores.

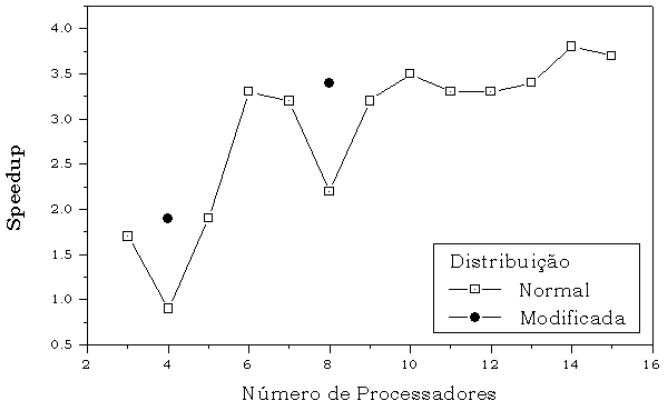

Figura 5.2: Gráfico de *speedup vs* número de processadores para o problema médio. Os pontos fora da curva correspondem aos tabelados em 5.6. Vemos que a queda foi eliminada com a distribuição dos processos modificada.

Tabela 5.6: Tempos de execução da paralelização por setores com distribuição modificada dos processos entre os processadores, para o problema médio.

Número de	*Médio*	
Processadores	*Tempo (s)*	*Speedup*
04	3.5	1.9
08	2.0	3.4

res (tab. 5.6), porém da figura 5.1 pode ser visto que conseguimos um *speedup* sub-linear, ou seja, o tempo de execução cai menos que linearmente com o aumento no número de processadores. Isso porque o tempo tomado para o cálculo de alguns setores limita o desempenho do programa. Pela tabela 5.5 podemos observar que o *speedup* não se alterou muito a partir de seis processadores. Isso se deve ao fato de que o *speedup* máximo que é possível de se atingir é limitado pelo tempo total tomado pelo processo que mais executa cálculos. O tempo de execução desse processo, assim como o valor do *speedup* máximo, sem considerar as perdas nas comunicações, é apresentado na tabela 5.7.

Para alcançarmos uma melhora sensível no *speedup* seria necessário dividir também a dimensão p (ver matriz da figura 2.5). Apesar de sincronizar bastante a ação dos setores como discutido na secção 4.3, com a granularidade fina que estamos usando os setores já estão sincronizados porque todos os setores do bloco estão nas arestas. Portanto não ocorreria tanta perda de desempenho em relação às execuções que realizamos e seria possível distribuir melhor o trabalho pelos processadores. Deixamos essa sugestão para trabalhos futuros.

Tabela 5.7: Tempos de execução total dos setores presentes no processo mais lento da paralelização por setores e o *speedup* máximo possível, para o problema médio.

Setor (q,s,p)	*Tempo (h)*	
0, 2, 0	0.42	
0, 2, 1	0.43	
0, 3, 0	0.19	
0, 3, 1	0.19	
TOTAL	1.2	*speedup máximo* 5.5

Da figura 5.1, vemos que o *speedup* alcançado pelos problemas médio e grande são bastante equivalentes, porém para o problema pequeno não. Essa diferença existe porque para o problema pequeno a comunicação envolvida no processo de paralelização é prejuducial ao desempenho pois o tempo gasto com ela é mais próximo ao gasto com os cálculos, que são menos custosos que para os outros dois casos. Então, a relação entre o tempo tomado para comunicações e o tempo total de execução limita o *speedup* para o problema pequeno.

5.4 Versão Híbrida

Usamos os oito conjuntos de dados de entrada citados na secção 5.2. A tabela 5.8 apresenta os tempos tomados para execuções da versão paralela por setores e para duas execuções da versão híbrida, descrita na secção 4.4. Estas últimas apresentam diferença na distribuição dos processos pelos processadores: rodamos dois conjuntos de dados simultaneamente e a execução *a* compartilha

os cálculos simultâneos pelos 15 processadores, enquanto que a *b* agrupa um número de máquinas (no caso sete) trabalhando com cada conjunto de dados. Novamente os oito conjuntos são diferenciados pelo sufixo utilizado para gerar as diferentes saídas.

Tabela 5.8: Tempos de execução para oito conjuntos de dados da paralelização por setores e da versão híbrida, para o problema médio. Execuções *a* e *b* utilizam a versão híbrida com a diferença: execução *a* compartilha os cálculos simultâneos pelos 15 processadores e execução *b* agrupa sete máquinas trabalhando com cada conjunto de dados.

Versão		*Tempo (h)*	*Speedup*
Setores	1000AP	1.8	
	1000AI	1.8	
	0875AP	1.7	
	0875AI	1.8	
	0750AP	1.6	
	0750AI	1.7	
	0625AP	1.6	
	0625AI	1.5	
	TOTAL	14	3.5
Híbrida	a	11	4.5
	b	8.3	5.9

Observamos que a versão híbrida tem um desempenho melhor que a versão paralela por setores executada repetidas vezes, que já apresenta diminuição no tempo em relação à seqüencial executada da mesma forma.

O agrupamento dos processos pelos processadores na execução *b* da paralelização híbrida mostrou-se bastante conveniente quando

comparada com a execução *a*. Isso se deve ao fato de o *speedup* da paralelização por setores não aumentar significativamente a partir de seis computadores, como pode ser visto na figura 5.5, e nesse caso usávamos sete computadores. Além disso, na execução *a* processos pesados dos dois conjuntos de dados que estavam sendo calculados poderiam cair no mesmo processador, aumentando com isso o tempo total de execução.

Com os resultados apresentados nas tabelas 5.2 e 5.8 construímos o histograma da figura 5.3 para melhor evidenciar as diferenças nas várias execuções de todas as versões. Suas barras são classificadas como a seguir:

1. execuções seqüenciais;
2. paralelização baseada em variação de parâmetros;
3. execuções paralelas por setores;
4. paralelização híbrida *b*;

Podemos perceber que todas as versões paralelas apresentam diminuição considerável em relação à versão seqüencial reescrita, barra 1. A versão paralela híbrida, barra 4, tem um desempenho pior que a paralelização inicial baseada em variação de parâmetros, barra 2, porém devemos lembrar que a versão híbrida poderá resolver problemas que ficariam impossibilitados pela outra paralelização, limitados pelos recursos exigidos pelos cálculos seqüenciais (principalmente memória).

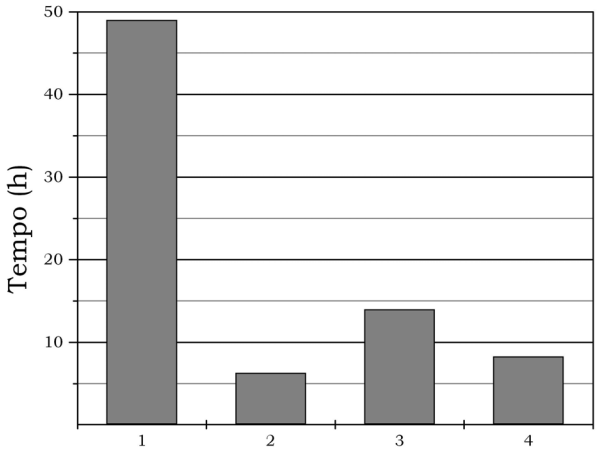

Figura 5.3: Histograma dos tempos de execução de diferentes versões do programa, seguindo a numeração: 1) oito execuções seqüenciais; 2) paralelização baseada em variação de parâmetros; 3) oito execuções paralelas por setores; 4) paralelização híbrida *b*.

Capítulo 6

Conclusões

O método do grupo de renormalização para o modelo de Anderson de duas impurezas é bastante adequado ao processamento paralelo com o uso de *clusters*. Entre os gargalos na pesquisa nessa área realizada pelo grupo de Física Teórica do IFSC podemos citar o tempo excessivo tomado pela execução do programa, as várias entradas de dados necessárias para as análises físicas e a precisão nos resultados obtidos que ficavam limitados à memória do computador. Com o uso de paralelismo conseguimos minimizar esses fatores.

Primeiramente reescrevemos o programa seqüencial orientando-o a objetos, eliminamos algumas cópias desnecessárias de vetores elemento a elemento e otimizamos algumas de suas funções. Com isso conseguimos uma diminuição considerável no tempo de processamento, mas a precisão ainda fica restrita às limitações impostas pelo tamanho da memória das máquinas.

A versão paralela baseada em variação de dados, descrita na secção 4.2, executa concorrentemente os cálculos para os diversos parâmetros de entrada em um esquema de mestre e escravos. Essa versão apresenta um *speedup* quase linear quando o número de processadores for um divisor do número de conjuntos de dados. O

balanceamento de carga é bom quando o número de processadores for bem menor que o número de conjuntos de dados, pois assim que uma máquina termina seus cálculos recebe novos parâmetros para reiniciar o procedimento. Suas limitações são falta de escalabilidade, visto que não se beneficia com o aumento dos computadores além da quantidade de parâmetros variados, e a precisão dos resultados continua condicionada à memória disponível como na versão seqüencial.

Paralelizamos então as execuções de cada conjunto de dados. Esta versão, discutida na secção 4.3, divide os setores por processos que trabalham paralelamente. Apresenta uma escalabilidade maior que a anterior, mas o balanceamento de carga é precário porque alguns setores trabalham por muito mais tempo que outros. Para melhorarmos o balanceamento de cargas adotamos uma granularidade bastante fina descrita no capítulo 5, mas o *speedup* fica limitado pelo tempo de execução do processo mais lento. Outra desvantagem é que somente um conjunto de dados é tratado por execução.

Finalmente implementamos uma versão paralela híbrida, que integra as duas versões anteriores e foi descrita na secção 4.4. Esta versão minimiza limitações das anteriores pois executa concorrentemente os diversos cálculos com variação de parâmetros, que passam a ser feitos como na paralelização por setores.

Foi adicionada a possibilidade de escrita de arquivos de *checkpointing* como descrito na secção 4.5 para minimizar perdas ao ocorrer um contratempo.

Com este trabalho adquirimos experiência na implementação de programas paralelos para *clusters*. O problema no qual trabalhamos se mostrou bastante adequado à programação paralela, pois é

possível distribuir os cálculos de diferentes setores e também as variações requeridas nos parâmetros de entrada entre processadores distintos, melhorando o desempenho final do programa.

O uso de um *cluster* de baixo custo como sistema paralelo para este problema é interessante porque verificamos que o principal limitante no desempenho foi o tempo tomado pelos cálculos e não o gasto em comunicações, um ponto fraco deste tipo de sistema que utiliza redes comerciais. Dessa maneira, um outro sistema com interconexão mais rápida não melhoraria o desempenho neste caso.

Como encaminhamento futuro desse trabalho podemos citar modificações nas divisões dos setores no processo de paralelização, ou seja, a dimensão p poderia ser explorada. Quando iniciamos as análises para a paralelização descartamos essa possibilidade pois acreditamos que o aumento na sincronização poderia ser prejudicial ao desempenho do programa (como discutido na secção 4.3). Porém com o uso da granularidade fina, já ocorre a sincronização dos processos, e a divisão dessa dimensão poderia melhorar o *speedup*. Além disso, seria interessante que dado o número de máquinas participantes da execução o arquivo fornecido ao MPI para a distribuição dos processos pelos processadores fosse gerado automaticamente de forma a atingir um melhor balanceamento de carga.

Ainda seria possível realizar a paralelização interna de alguns dos cálculos executados nos setores, tais como a montagem e a diagonalização do Hamiltoniano. Para isso seria possível utilizar a biblioteca *ScaLapack* [32], que apresenta métodos já paralelizados de manipulação de matrizes.

Referências Bibliográficas

[1] K. G. Wilson. The Renormalization Group: Critical Phenomena and the Kondo Problem. *Rev. of Mod. Physics*, 47(4):773, 1975.

[2] H. R. Krishna-murthy, J. W. Wilkins, and K. G. Wilson. Renormalization-Group Approach to the Anderson Model of Dilute Magnetic Alloys. *Phys. Rev. B*, 21(3):1003, 1980.

[3] P. W. Anderson. Localized Magnetic States in Metals. *Phys. Rev.*, 124(1):41, 1961.

[4] S. Alexander and P. W. Anderson. Interaction Between Localized States in Metals. *Phys. Rev.*, 133(6):1594, 1964.

[5] J. Kondo. Resistance Minimum in Dilute Magnetic Alloys. *Prog. of Theor. Phys.*, 32(1):37, 1964.

[6] M. A. Ruderman and C. Kittel. Indirect Exchange Coupling of Nuclear Magnetic Moments by Conduction Electrons. *Phys. Rev.*, 96(1):99, 1954.

[7] T. Kasuya. A Theory of Metallic Ferro- and Antiferromagnetism on Zener's Model. *Prog. of Theor. Phys.*, 16(1):45, 1956.

[8] K. Yosida. Anomalous Electrical Resistivity and Magnetoresistence Due to an s-d Interaction in Cu-Mn Alloys. *Phys. Rev.*, 107(2):396, 1957.

[9] L. N. Oliveira. The Numerical Renormalization Group and the Problem of Impurities in Metals. *Braz. J. of Phys.*, 22(3):155, 1992.

[10] W. C. Oliveira and L. N. Oliveira. Generalized Numerical Renormalization Group Method to Calculate the Thermodinamical Properties of Impurities in Metals. *Phys. Rev. B*, 49(17):11986, 1994.

[11] J. B. Silva, W. L. C. Lima, W. C. Oliveira, J. L. N. Mello, L. N. Oliveira, and J. W. Wilkins. Particle-Hole Asymmetry in the Two-Impurity Kondo Model. *Phys. Rev. Lett.*, 76(2):275, 1996.

[12] David E. Culler, Jaswinder Pal Sing, and Anoop Gupta. *Parallel Computer Architecture. A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[13] T. Sterling. Beowulf: a Parallel Workstation For Scientific Computation. In *Proc. of the 1995 Intl. Conf. on Parallel Processing*, 1995.

[14] S. C. Costa. Calor Específico do Modelo de Anderson de uma Impureza por Grupo de Renormalização Numérico. Tese de Mestrado, IFSC-USP, 1995.

[15] C. A. de Paula. *Densidade Espectral para o Modelo de Anderson de Duas Impurezas*. Tese de Doutorado, IFSC-USP, 1998.

[16] M. Yoshida, M. A. Whitaker, and L. N. Oliveira. Renormalization-Group Calculation of Excitation Properties for Impurity Models. *Phys. Rev. B*, 41(3):9403, 1990.

[17] S. C. Costa, C. A. Paula, V. L. Líbero, and L. N. Oliveira. Numerical Renormalization-Group Computation of Specific Heats. *Phys. Rev. B*, 55(1):30, 1997.

[18] W. L. C. Lima. *Assimetria Partícula-Buraco no Modelo de Kondo de Duas Impurezas*. Tese de Doutorado, IFSC-USP, 1997.

[19] G. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamim/Cummings, 2nd edition, 1994.

[20] J. L. Hennessy and N. Jouppi. Computer Technology and Architecture. An Evolving Interaction. *IEEE Computer*, 24(9):18, 1991.

[21] T. E. Anderson, D. E. Culler, and D. A. Patterson. A Case For NOW. *IEEE Micro*, 1(15):54, 1996.

[22] Ian Foster. *Designing and Building Parallel Programs*. Addison Wesley, 1995.

[23] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1995.

[24] MPI Forum. `http://www.mpi-forum.org`. visitado em 07/05/2001.

[25] G. Travieso. *Proposta e Implementação de um Sistema de Processamento Paralelo para Dinâmica Molecular*. Tese de Doutorado, IFSC-USP, 1993.

[26] G. Travieso. Comparing High-Level and Low-Level Implementations of a Molecular Dynamics Algorithm. In *Proc. of the Second Intl. Workshop on High-Level Programming Models and Supportive Environments*, 1997.

[27] The Linux Kernel. http://www.kernel.org. visitado em 07/05/2001.

[28] Free Software Foundation. The GNU Project, http://www.gnu.org. visitado em 07/05/2001.

[29] Willian Gropp and Edwing Lusk. *User's Guide for MPICH, A Portable Implementation of MPI.* Argonne National Laboratory, 1996.

[30] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 3rd edition, 1997.

[31] A. Muezerie, R. Nakashima, G. Travieso, and J. Slaets. Matrix Calculations with SIMD Floating Point Instructions on x86 Processors. In *Proc. of the SBAC*, 2001. a ser publicado.

[32] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLapack User's Guide*. SIAM, 1997.

Apêndice A

Discussões do Código Computacional

Como dito no capítulo 4, o programa foi escrito em C++ e várias classes foram implementadas para armazenar e tratar os dados. Uma classe representa um *tipo abstrato de dados* em C++, e especifica um novo tipo de dados, sua estrutura, as operações que podem ser efetuadas sobre o mesmo e controla o acesso aos diversos elementos do tipo.

Foram usadas *exceções* para o tratamento de erros. O tratamento de exceções corresponde a construções sintáticas que nos permitem lidar com as situações de erro fora do fluxo normal do programa, isto é, o tratamento dos erros não fica distribuído pelo programa misturado com o código de processamento normal.

A seguir vamos discutir os aspectos mais importantes das classes escritas no decorrer do nosso trabalho.

Classe *Vetor* É uma classe bem geral e pode ser reaproveitada para qualquer programa que utilize vetores de *doubles*. Contém como membros um vetor de *doubles* alocado dinamicamente e sua dimen-

são, e os métodos para acessar esses membros sem corrompê-los. Entre seus métodos temos a sobrecarga dos operadores = e [ ], leitura da dimensão do vetor, acesso aos seus elementos, realocação, liberação de memória, produto restrito e produto vetorial.

Classe *Execucao* A classe *Execucao* apresenta a iteração e fase correntes, número máximo de iterações para aquela execução e a paridade da fase alta. Seus parâmetros são atualizados antes de iniciar iteração ou fase e são consultados em diversas partes do programa. Para isso apresenta métodos para leitura da iteração, fase e paridade da fase alta, métodos que ajustam os parâmetros para iniciar iteração e fase, e métodos que sinalizam se está na última iteração e em qual fase.

Classe *Modelo* A classe *Modelo* trata dos parâmetros do modelo utilizado tais como interação RKKY, acoplamento banda-impureza, repulsão coulombiana, distância entre as impurezas, e também de parâmetros do método numérico como Λ e z. Esses parâmetros são lidos de um arquivo de entrada e usados várias vezes aos longo da execução. Vários métodos fornecem acesso a esses parâmetros e ainda há métodos para fornecer o valor de z na fase baixa e alta.

Classe *KsiNu* Coeficientes ξ e ν da banda de condução. Essa classe apresenta métodos para calcular os elementos dos vetores ξ e ν dado o valor de z e também para leitura desses elementos.

Classe *Cond* Trata de parâmetros da banda de condução. Apresenta métodos para calcular susceptibilidade de Pauli e a contribuição no calor específico para os dois canais da banda de condução.

Classe *Termo* A classe *Termo* trata de propriedades termodinâmicas. Armazena também os limites de energias para o truncamento ultravioleta e para a região dos cálculos termodinâmicos. Entre seus métodos apresenta o cálculo e a leitura das temperaturas inversas renormalizadas, retorno da energia de corte, da energia limite para o truncamento ultravioleta, da temperatura, degenerescência da impureza, número de temperaturas para as quais as propriedades termodinâmicas são calculadas em cada iteração e conversor de energia adimensional em dimensionada.

Classe *Primitivo* Trata da base primitiva que é usada para a geração de novos estados em um setor ativo na fase. Armazena o número de auto-estados dos pais dos setores e o relaciona com sua posição "geográfica" (norte, sul, leste ou oeste). Apresenta métodos para leitura do número de auto-estados de um determinado pai, leitura da dimensão do vetor que armazena os dados e realocação.

Classe *Estado* Essa classe cuida dos autovetores e autovalores. Seus métodos fornecem acesso às dimensões dos vetores, aos seus elementos, realocação e cópia.

Classe *Pagina* Para armazenarmos e tratarmos dos elementos de matriz invariante usamos as páginas. Um objeto dessa classe armazena as páginas ativa e passiva. Para essa classe diminuimos bastante a cópia desnecessária de dados trabalhando com o uso de ponteiros. Vale a pena ressaltar que aproximadamente metade do tempo de execução total é tomado pelos cálculos das páginas e portanto sua otimização é de grande importância para o desempenho final do programa. Também apresenta métodos para leitura das dimensões e dos elementos das duas páginas, realocação e có-

pia. Ainda tem um método para trocar a página ativa com a passiva (usado na preparação da iteração ou fase).

Classe *Setor* Um objeto da classe *Setor* representa um dos setores qsp (conforme discutido na secção 4.1) e tem todos os elementos necessários ao seu funcionamento. Além de seu rótulo qsp, apresenta os autovalores e autovetores (classe *Estado*), as páginas (classe *Pagina*), o estado primitivo (classe *Primitivo*), ponteiros para seus pais e vizinhos, além de *flags* para indicar a atividade do setor na fase. Apresenta vários métodos como para leitura de carga, spin, paridade, *flags*, método para atualizar os pais e/ou vizinhos que mudam em relação à fase, chamadas para os métodos de realocação e leitura dos autovalores, autovetores, páginas e espaço primitivo. Tem também métodos para construção do espaço primitivo, para construção e diagonalização do Hamiltoniano e também para o cálculo da página ativa e atualização da passiva. Dois métodos são usados quando o setor tem pai ou vizinho sombra, para alocação destes.

Classe *Espaco* É a estrutura que representa a matriz esquematizada na figura 2.5. Grande parte dos cálculos é gerenciada por essa classe pois é ela quem comanda a vizinhança entre os setores. A classe *Espaco* contém uma matriz tridimensional de objetos da classe *Setor* e portanto é a estrutura que é dividida na paralelização. Apresenta além da matriz de objetos do tipo *Setor*, os limites de carga e spin da matriz, objetos da classe *Execucao*, *Modelo*, *KsiNu*, *Termo* e *Cond*. A iteração -1 é um de seus métodos, assim como as preparações de iteração e fase (que fazem chamadas para métodos de seus membros), a execução da fase, que chama os métodos de *Setor* para cálculos com o Hamiltoniano e com as páginas, e determina se setor é ativo ou não na fase. Ainda apresenta o método que

calcula e imprime as propriedades físicas de interesse. Um de seus métodos trata das comunicações entre os setores e seus sombras equivalentes.

Apêndice B

Dados de Entrada

No capítulo 5 citamos três conjuntos de dados que foram utilizados para analisar as várias versões do nosso programa. Esses conjuntos diferiam entre si no tempo de processamento, e foram rotulados por *pequeno*, *médio* e *grande*. A seguir apresentamos os três arquivos de entrada, nas figuras B.1, B.2 e B.3, respectivamente. Para algumas execuções há ainda variação nos valores de z e na paridade da fase alta (alta_par).

```
1.118034e0      V
100.            U
1.0             kf
1.507           r
-0.0            \Delta_I/J^2
8.              Lambda
1.000           z
1.e-15          E_min
12.             E_lim_2
18.             E_lim_4
18.             E_lim_termo
0               alta_par
0               Cntrle_de_imprssao(enrgias):HAM=-1,ALTA=0,BAIXA=1,
                                            NADA=2,AL+BX=3
7               q_lim
7               ds_lim
0.35            bbar_2era0.23
9               ntemp
0.35            bbar_4
180             min_size
300             max_size
```

Figura B.1: Arquivo de entrada usado para o problema pequeno.

```
1.118034e0      V
100.            U
1.0             kf
1.507           r
-0.0            \Delta_I/J^2
8.              Lambda
1.000           z
1.e-15          E_min
17.             E_lim_2
30.             E_lim_4
30.             E_lim_termo
0               alta_par
0               Cntrle_de_imprssao(enrgias):HAM=-1,ALTA=0,BAIXA=1,
                                            NADA=2,AL+BX=3
7               q_lim
7               ds_lim
0.35            bbar_2era0.23
9               ntemp
0.35            bbar_4
180             min_size
300             max_size
```

Figura B.2: Arquivo de entrada usado para o problema médio.

```
1.118034e0      V
100.            U
1.0             kf
1.507           r
-0.0            \Delta_I/J^2
8.              Lambda
1.000           z
1.e-15          E_min
21.             E_lim_2
35.             E_lim_4
35.             E_lim_termo
0               alta_par
0               Cntrle_de_imprssao(enrgias):HAM=-1,ALTA=0,BAIXA=1,
                                            NADA=2,AL+BX=3
7               q_lim
7               ds_lim
0.35            bbar_2era0.23
9               ntemp
0.35            bbar_4
180             min_size
300             max_size
```

Figura B.3: Arquivo de entrada usado para o problema grande.

Apêndice C

MPI

O MPI, *Message Passing Interface*, é um padrão de biblioteca de passagem de mensagens para sistemas paralelos. Foi elaborado para fornecer uma base comum de desenvolvimento de programas paralelos em plataformas distintas.

Para os exemplos mostraremos a interface em C somente das rotinas que foram utilizadas no desenvolvimento do nosso programa. Os parâmetros já apresentados não serão discutidos nas rotinas seguintes. Para maiores detalhes da semântica das rotinas do MPI consulte [23].

C.1 Conceitos Básicos

Em um programa MPI, antes da chamada de qualquer das rotinas MPI este deve ser inicializado, e antes do término do programa ele deve ser finalizado. Para isso são definidas as rotinas, respectivamente:

```
int MPI_Init(int *argc, char **argv);
int MPI_Finalize();
```

os parâmetros `argc` e `argv` devem ser os tradicionais parâmetros recebidos por `main`.

As rotinas de MPI são construídas em torno dos conceitos de *processos*, *comunicadores*, *mensagens* e *tipos de dados*.

Processos O elemento básico de computação em MPI é chamado *processo*. Um processo é uma entidade de execução seqüencial, que pode ser comparada com um programa seqüencial em execução. A diferença consiste em que um processo MPI tem a possibilidade de cooperar com outros processos MPI para a execução de alguma tarefa. Um programa MPI consiste então na especificação do código a ser executado por um conjunto de processos que cooperam na solução de um problema

Comunicadores Diferentes processos são interligados em MPI através de uma entidade chamada *comunicador*. Sempre que precisar existir alguma interação entre processos distintos, eles devem estar associados a um mesmo comunicador.

Mensagens Quando processos precisam coordenar suas operações, faz-se necessária a troca de informações. Em MPI a troca de informações é realizada por meio de *mensagens*. Uma mensagem pode ser enviada entre dois processos por meio de um comunicador em que eles tomem parte.

Tipos de Dados Os dados enviados em mensagens têm tipos associados (inteiro, número de ponto flutuante, etc). Apesar de em um sistema constituído de máquinas iguais os tipos não serem importantes, eles são importantes quando nós distintos do sistema paralelo têm arquiteturas diferentes. Se necessária, uma conversão de representação será realizada *automaticamente* pelo MPI, utilizando informações de tipo fornecidas

pelo usuário.

C.2 Comunicações

Para uma melhor compreensão das interfaces de comunicação, dois termos são importantes: *rank* e *tag*.

rank Cada comunicador tem associado a ele um grupo de processos. Cada um desses processos tem uma numeração própria dentro do comunicador, denominada *rank*. Se um comunicador tem associado a ele P processos, esses processos serão numerados com *ranks* de 0 a $P-1$.

tag Como diversas mensagens podem estar circulando simultaneamente pelo sistema, inclusive pelo mesmo comunicador, além dos dados a serem transmitidos a mensagem deve conter informações de identificação. Essas informações, denominadas o *envelope* da mensagem são: processo remetente, processo destinatário, um inteiro denominado *tag* e o comunicador utilizado. O *tag* é utilizado para distinguir mensagens diferentes provenientes de um mesmo remetente.

Normalmente especificam-se valores do rementente e do *tag explicitamente*, o que resulta em que apenas mensagens com o valor de *tag* especificado e provenientes do remetente indicado serão recebidas. Se outras mensagens que não concordam com essa especificação estiverem disponíveis elas serão ignoradas. Esta regra pode ser abrandada pelo uso de especificações de *tag* ou remetente *genéricos*, utilizando as constantes `MPI_ANY_TAG` e `MPI_ANY_SOURCE`. Especificando o *tag* ou o remetente dessa forma, uma mensagem com *quaisquer tag* ou remetente, respectivamente, será aceita pa-

ra recepção. Se desejarmos posteriormente saber qual o *tag* ou o remetente da mensagem podemos utilizar informações do *status* da comunicação.

C.2.1 Comunicação Ponto-a-Ponto

O termo comunicação ponto-a-ponto é utilizado em MPI para indicar os tipos de comunicação que envolvem um par de processos. Essas comunicações são efetuadas por primitivas *send* (no remetente) e *receive* (no destinatário).

Para uma comunicação devemos especificar os dados a serem comunicados e o envelope.

A interface para transmissão é:

```
int MPI_Send(void *buffer, int cont, MPI_Datatype tipo, int dest,
             int tag, MPI_Comm com);
```

onde `buffer` indica o endereço do primeiro elemento a ser enviado, `cont` o número de elementos a enviar, `tipo` indica o tipo de dados como `MPI_DOUBLE` ou `MPI_INT`, `com` indica o comunicador a utilizar para a transmissão, `dest` o *rank* do processo destinatário em `com` e `tag` indica o *tag* associado a esta comunicação.

A interface para a recepção é similar, mas envolve um parâmetro adicional para a verificação do *status* da comunicação:

```
int MPI_Recv(void *buffer, int cont, MPI_Datatype tipo, int rem,
             int tag, MPI_Comm com, MPI_Status *status);
```

o parâmetro `rem` indica o *rank* do processo do qual a mensagem deve ser recebida. Na recepção `cont` indica o número máximo de elementos que podem ser recebidos no *buffer* especificado.

No programa utilizamos uma variante dessas rotinas, que é a comunicação *não-bloqueante*. Este tipo de comunicação deve ser utilizada quando desejamos iniciar uma comunicação mas prosseguir com o processamento enquanto ocorre a comunicação. Dessa forma é feita uma chamada para a rotina `MPI_Isend` e a execução pode ser continuada até que esses dados devam ser alterados, quando é feita uma chamada para `MPI_Wait` a qual bloqueia até que o *buffer* possa ser alterado. A interface dessas rotinas é:

```
int MPI_Isend(void *buffer, int cont, MPI_Datatype tipo, int dest,
              int tag, MPI_Comm com, MPI_Request *request);
int MPI_Wait(MPI_Request *request, MPI_Status *status):
```

onde `request` é um objeto opaco que identifica a operação de comunicação.

Outras variantes de comunicação ponto-a-ponto podem existir, tais como *bufferizada*, *síncrona* e *recepção pronta* [23].

Utilizamos ainda `MPI_Pack` e `MPI_Unpack` para empacotar dados em um só *buffer* antes de enviá-lo, e desempacotá-lo depois de o receber. O tempo gasto para iniciar uma comunicação é grande, e a vantagem de se fazer o envio de uma só mensagem ao invés de várias é que esse tempo transcorre uma única vez.

```
int MPI_Pack(void *buf1, int cont1, MPI_Datatype tipo, void *buf2,
             int cont2, int *posicao, MPI_Comm com);
int MPI_Unpack(void *buf1, int cont1, int *posicao, void *buf2,
               int cont2, MPI_Datatype tipo, MPI_Comm com);
```

onde `buf1` é o *buffer* de entrada, `cont1` o número de dados de entrada, `buf2` indica o *buffer* de saída, ou seja, o que será enviado na

comunicação, `cont2` indica o tamanho do `buf2` e `posicao` indica a posição corrente no `buf2` em *bytes*.

C.2.2 Comunicação Coletiva

Comunicações coletivas são aquelas que envolvem um grupo de processos, ao invés de apenas um par de processos. Em MPI, o grupo envolvido na comunicação é especificado através de um comunicador: todos os processos associados ao comunicador utilizado participarão da comunicação. As operações coletivas em MPI são as seguintes: barreira, *broadcast*, coleta, distribuição, redistribuição, redução e prefixo. Em todas as operações coletivas, todos os processos do comunicador utilizado para especificar o grupo devem realizar uma chamada para a rotina da operação.

No nosso programa utilizamos apenas algumas dessas possibilidades de comunicações coletivas, e as decreveremos a seguir, assim como a interface dessas operações.

Barreira Uma *barreira* é utilizada quando desejamos sincronizar a operação de todos os processos de um grupo. Os processos somente passam pela barreira quando todos os processos do grupo a tiverem atingido:

```
int MPI_Barrier(MPI_Comm com);
```

Broadcast Um *broadcast* é uma operação em que um dado presente em um dos processos é transmitido para todos os outros processos do grupo:

```
int MPI_Bcast(void *buffer, int cont, MPI_Datatype tipo,
              int raiz, MPI_Comm com);
```

Na operação de *broadcast*, o processo que inicialmente possui o dado a ser transmitido é denominado *raiz*.

Redução Operações de redução são aquelas onde cada um dos processos fornece um valor para um cálculo global, como uma somatória ou um produtório, ou a busca de um mínimo. Na variante de redução que utilizamos o valor calculado é recebido por todos os processos envolvidos:

```
int MPI_Allreduce(void *sbuffer, void *rbuffer, int cont,
                  MPI_Datatype tipo, MPI_Op op,
                  MPI_Comm com);
```

onde `sbuffer` indica o valor que será fornecido pelo processo para o total a ser calculado, `rbuffer` é o local onde o total será armazenado e `op` indica qual a operação que deve ser efetuada, como por exemplo `MPI_MIN` para encontrar o mínimo valor entre todos ou `MPI_SUM` para executar uma somatória, entre outros.

C.3 Comunicadores

Todos os processos de uma mesma aplicação MPI são automaticamente associados pelo sistema a um comunicador denominado `MPI_COMM_WORLD`. Para muitos problemas o uso deste comunicador pré definido é suficiente. Em alguns casos, porém, isso não ocorre, especialmente quando desejamos realizar comunicações coletivas usando apenas uma parte dos processos disponíveis para a aplicação ou quando desejamos implementar rotinas paralelas que serão úteis em diversos programas.

Na versão híbrida do nosso programa secção 4.4, tínhamos a necessidade de criar novos comunicadores porque os processos que

trabalhavam com cada conjunto de parâmetros realizam comunicações coletivas internas à execução. Para viabilizar essas comunicações usamos uma rotina que particiona um comunicador em outros novos. A sintaxe é:

```
int MPI_Comm_split(MPI_Comm com, int cor, int chave,
                   MPI_Comm *novo_com);
```

onde `com` é o comunicador que será particionado, `cor` é um valor que indica a qual novo comunicador o processo pertencerá, o valor da `chave` define como será a ordem em termos de *rank* no novo comunicador, e `novo_com` indica o novo comunicador.

Para encontrar o *rank* de um processo em um dado comunicador deve-se utilizar a rotina:

```
int MPI_Comm_rank(MPI_Comm com, int *rank);
```

onde `rank` devolverá o *rank* do processo no comunicador `com`.

Se desejarmos saber o número de processos em um comunicador usamos:

```
int MPI_Comm_size(MPI_Comm com, int *size);
```

C.4 Implementações

O padrão MPI especifica apenas a interface de programação, sem entrar em detalhes sobre a compilação e execução dos programas. Isto foi feito deliberadamente pois esses pontos são muito dependentes do tipo de sistema operacional e da arquitetura da máquina paralela.

Desde o lançamento da primeira versão de MPI em 1995 diversos fabricantes têm implementado MPI para suas máquinas, e também têm surgido implementações para redes de estações de trabalho. Entre as implementações para estações, as duas mais utilizadas são a MPICH e a LAM. As duas são praticamente equivalentes e seu uso é muito semelhante.

Descreveremos como realizar a compilação e a execução de programas em MPICH, que foi a implementação usada por nós. Lembramos que o sistema operacional usado foi LINUX.

C.4.1 Compilação

Para auxiliar na compilação de programas MPI, as implementações citadas têm programas que se encarregam de chamar o compilador do sistema passando para ele automaticamente a locação dos arquivos de inclusão e biblioteca necessários ao MPI.

A compilação de um programa em C++ chamado `main.cc` será:

```
prompt> mpiCC main.cc -o main
```

C.4.2 Execução

Ao executar um programa MPI, devemos especificar, além do código a executar, o número de processos que desejamos colocar em execução. Isto é especificado pelo programa `mpirun`.

Por exemplo, a execução do programa `main` com 15 processos:

```
prompt> mpirun -np 15 main
```

Para as nossas execuções, especificamos ainda um arquivo `maq`

contendo uma lista com os números IP dos processadores que participam da execução. A linha de comando então torna-se:

```
prompt> mpirun -np 15 -nolocal -machinefile maq main
```

a opção `nolocal` é utilizada para que a própria máquina que dispara o processo não participe da execução, e sim as listadas em `maq`. No entanto, ela pode fazer parte da lista de `maq`.