Universidade de São Paulo
Instituto de Física de São Carlos
Departamento de Física e Informática

Técnicas de orientação ao objetos para computação científica paralela

Francisco Aparecido Rodrigues

Dissertação apresentada ao Instituto de Física São Carlos- IFSC, USP, como parte dos requisitos necessários para obtenção do título de Mestre em "Física Aplicada - Opção Física Computacional"

Orientador: *Prof. Dr. Gonzalo Travieso*

São Carlos
2004

Agradecimentos

Ao Prof. Dr. Gonzalo, pela amizade, confiança, paciência e pela grande competência na orientaçao do trabalho, além do constante otimismo oferecido.

À minha mãe Arsênia por sempre me apoiar e confiar na minha capacidade, além de sempre oferecer otimismo e esparança incabáveis.

Aos meus irmão Fabrício, Gabriela, Fátima, Meire, Elisa, João Carlos e Alcindo pelo apoio e compreensão.

A todos os meus amigos pela solidariedade e lealdade, além dos momentos de diversão que estes me ofereceram durante todo esse período.

Agradeço ao Paulino, a Thaty, o André e a Elô, meus grandes amigos de laboratório, pelo companheirismo e pela ajuda sempre prestativa.

Agradeço a todas as pessoas que de uma forma ou de outra contribuíram para o meu trabalho.

Finalmente, agradeço a Deus pela oportunidade de trabalhar em a favor da ciência.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Neste trabalho apresentamos a metodologia de orientação ao objeto no desenvolvimentos de uma biblioteca de classes para facilitar o processo de programação numérica paralela. Na implementação dos métodos das classes utilizamos as rotinas do pacote ScaLAPACK, sendo que essas classes oferecem métodos para manipulações matriciais básicas e para a diagonalização de matrizes, onde essas matrizes podem ser reais e complexas, de simples e dupla precisão. Este trabalho apresenta detalhes de implementação e uma análise comparativa de desempenho, a fim de mostrarmos a eficiência e as facilidades de uso da orientação ao objeto no desenvolvimento de programas científicos paralelos.

Abstract

In this work we apply object oriented technics to the development of a class library to facility the process of program development for numerical parallel computations. In the implementation of the class methods, we used the routines of the ScaLAPACK package; the classes offer methods to basic matrix manipulations and to matrix diagonalization, where the matrices can be real and complex, with single or double precision. This work describes details of implementation and a comparative analysis of performance, in order to show the efficiency and the easy of use of object orientation in the develpment of parallel scientific programs.

Capítulo 1 Introdução

A necessidade por poder computacional favoreceu o desenvolvimento da computação paralela. Hoje, mais do nunca, sistemas paralelos estão se tornando cada vez mais populares. No entanto, as facilidades apresentadas no desenvolvimento dos softwares, não é a mesma oferecida pelo hardware, sendo o processo de programação paralela uma tarefa árdua e em muitos casos desencorajadora.

1.1 Motivação

Apesar de existirem muitas linguagens paralelas, compiladores paralelizantes e bibliotecas paralelas que podem ser usadas conjuntamente com linguagens seqüenciais, a construção de programas paralelos que ofereçam alto desempenho é complexa. Deste modo, muitos estudos têm sido feitos de forma a obter meios de programação paralela que ofereçam grau de dificuldade próximo ao oferecido pela programação seqüencial.

Uma das possibilidades para obter essa facilidade é o uso de compiladores paralelizantes, que tentam extrair o máximo de paralelismo de códigos seqüenciais. No entanto, o desempenho conseguido desta forma está muito longe do ideal.

Outra possibilidade são as linguagens paralelas. No entanto elas também apre-

sentam certos fatores que desestimulam seu uso. Um deles é o fato do programador ter que aprender uma nova linguagem. Outro, é que a extração do paralelismo pode ser difícil, já que a obtenção de um bom balanceamento de carga e granularidade grossa é um desafio para essas linguagens.

Finalmente, é possível criar-se programas paralelos aproveitando-se as linguagens seqüenciais e usando-se bibliotecas que permitam a comunicação entre os processos. Neste trabalho discutiremos diversas questões relacionadas a esse tipo de programação.

No entanto, a construção de softwares paralelos utilizando-se essas bibliotecas ainda é bastante complexa. Cabe ao programador especificar os meios de comunicação, a distribuição dos dados entre os processadores, os momentos de troca de mensagens, etc, ou seja, ele tem que desviar constantemente a atenção do processo de resolução do problema para se preocupar com detalhes do paralelismo que poderiam estar ocultos a ele. O MPI [2] é um padrão que define algumas dessas bibliotecas que oferecem rotinas para fazerem a comunicação entre os processadores e a elaboração de programas paralelos.

Como a computação paralela ainda está intimamente relacionada à computação científica, uma saída encontrada para contornar esse problema foi a construção de bibliotecas numéricas paralelas, como o ScaLAPACK [3], que é uma biblioteca paralela para problemas de álgebra linear que envolvam cálculos matriciais. Mesmo assim, sua interface e muitos detalhes de uso dificultam sua utilização.

1.2 Objetivos

O objetivo principal do nosso trabalho foi desenvolver uma biblioteca que ofereça um desempenho tão bom quanto ScaLAPACK, mas que possua uma interface muito mais amigável, a fim de facilitar o processo de desenvolvimento de progra-

mas científicos.

Nos preocupamos unicamente com os problemas de autovalores e autovetores aplicados em matrizes densas. Na implementação da biblioteca, utilizamos diversos recursos do ScaLAPACK bem como de suas bibliotecas internas, como o BLACS e PBLAS [4].

Com o uso de orientação ao objeto, detalhes de paralelismo, como a construção da grade de processos ou mesmo inicialização das rotinas de comunicação do ScaLAPACK ficaram ocultos ao programador. Além disso, usando *templates*, herança e sobrecarga de operadores, pudemos definir operações matriciais básicas fornecendo uso simplificado. Deste modo, para a utilização da biblioteca que desenvolvemos não é necessário conhecimento aprofundado de paralelismos e o código implementado com o uso de nossa biblioteca muitas vezes ficará parecido com um código puramente seqüencial.

Como o uso de orientação ao objeto em nosso trabalho mostrou-se bastante promissor, há a possibilidade de desenvolvimento futuros com a implementação de biblioteca análogas à que criamos para outros problemas científicos, como de álgebra linear, equações diferenciais, estatística, etc. Além disso, há a possibilidade de estender esta biblioteca para todas as rotinas do ScaLAPACK e para diversos tipos de matrizes.

1.3 Próximos capítulos

No restante da dissertação, teremos os seguintes capítulos:

Capítulo 2 : Apresentamos conceitos sobre programação paralela bem como sobre programação numérica.

Capítulo 3 : Introduzimos conceitos básicos de álgebra linear e de computação numérica utilizada para solução de alguns problemas algébricos.

Capítulo 4 : Discutimos em detalhes as características do ScaLAPACK e seu uso, descrevendo algumas rotinas e mostrando o método padrão de construção de programas que utilizam essa biblioteca.

Capítulo 5 : Os conceitos de orientação ao objeto são introduzidos e apresentados como ferramenta para a construção de softwares numéricos que tratam de problemas de álgebra linear.

Capítulo 6 : A implementação da biblioteca é discutida e apresentada algumas características básicas, bem como sua facilidade de uso.

Capítulo 7 : Mostramos uma aplicação da biblioteca bem como uma análise de desempenho em relação ao uso do ScaLAPACK tradicional, onde exibimos as vantagens e desvantagens na utilização de orientação ao objeto.

Capítulo 8 : Concluímos o trabalho apresentando as limitações de uso e as facilidades da biblioteca bem como discutimos as aplicações e trabalhos futuros.

CAPÍTULO

2
Problemas numéricos e paralelismo

Cientistas e engenheiros descrevem fenômenos físicos em termos de modelos matemáticos. Esses modelos geralmente são considerados teoricamente dentro de um espaço contínuo e no qual a obtenção de uma solução analítica pode ser complexa. Desta forma, faz-se uso de matemática discreta, usando métodos numéricos que podem ser transferidos para *softwares* e resolvidos com auxílio de computadores.

A simulação de modelos físicos provenientes de matemática discreta em computadores oferece grandes vantagens aos pesquisadores. Dentre elas podemos citar a redução do custo de pesquisa, bem como a redução de riscos, como no caso dos experimentos perigosos. Outro fator de importância são os experimentos que não estão disponíveis no mundo real, como os de construções de modelos em cromodinâmica quântica. Além disso, experimentos com fármacos, de engenharia aeronáutica, de semicondutores, etc, podem ser realizados várias vezes com um custo muito menor do que o realizado de forma real, além dos experimentos perigosos, como os de bombas nucleares.

Com a utilização de métodos numéricos, os modelos tornaram-se mais precisos e a necessidade de um poder computacional cada vez maior se fez necessária. Como os computadores seqüenciais não oferecem o desempenho exigido, os modelos seqüenciais precisaram ser transferidos para as máquinas de melhor

desempenho, ou seja, as máquinas paralelas. Essa transferência implicou no desenvolvimento de *softwares* paralelos e a adequação de bibliotecas seqüenciais a esse novo modelo.

Nos últimos, a padronização de bibliotecas numéricas vem se tornando um fato real, determinando a forma da declaração das subrotinas, o modo de armazenamento, os tipos de dados e a linguagem a ser utilizada. No entanto, a implementação não é padronizada, o que possibilita aos desenvolvedores criar versões da biblioteca para uma arquitetura específica, de forma a oferecer um certo nível de otimização, usando recursos específicos dessa arquitetura.

2.1 Bibliotecas e softwares numéricos

Dentre as bibliotecas e softwares numéricos seqüenciais, há uma grande quantidade disponível gratuitamente. Entre elas, podemos citar:

- **MTL** (*Matrix Template Library*) [5]: é uma biblioteca construída de forma similar à STL [6, 7] para operar sobre matrizes. A proposta é fornecer rotinas que facilitem ao programador o trabalho com matrizes esparsas, densas, de banda, simétricas, triangulares e algoritmos básicos que possam operar sobre elas.

- **IML++**(*Interative Methods Library*) [8]: é uma biblioteca de *templates* construída em C++ que usa modernos métodos iterativos na resolução de sistemas de equações lineares simétricos e não simétricos. As operações são efetuadas sobre matrizes densas, esparsas e matrizes distribuídas.

- **ARPACK++** [9]: o ARPACK é uma biblioteca para solução de problemas de autovalores. O ARPACK++ é sua versão que utiliza orientação ao objeto, fornecendo uma coleção de classes em C++ que permitam efetuar as

mesmas operações do ARPACK. O uso de *templates* e classes facilitam a resolução de problemas de autovalores fornecendo uma interface mais amigável do que a oferecida pelo ARPACK, que usa FORTRAN, facilitando a construção de programas.

- **SparseLib++** [10, 11]: é uma biblioteca de classes em C++ para computação de matrizes esparsas em diversas plataformas computacionais. O pacote SparseLib++ é constituído de diversas classes para armazenamento de matrizes esparsas e funções que permitem o tratamento dessas matrizes.

- **MV++** [12]: é um pequeno, mas eficiente, conjunto de classes para computação de vetores e matrizes simples. Ele foi escrito em C++ e sua criação objetivou a computação otimizada em sistemas RISC e arquiteturas que usam *pipeline* de instruções.

- **Template Numerical Toolkit (TNT)** [13]: é uma coleção de interfaces para construção de softwares numéricos usando C++. O TNT define as interfaces básicas para estruturas de dados, tal como *arrays* multidimensionais e matrizes esparsas. O objetivo principal deste pacote é fornecer componentes de software que ofereçam portabilidade e facilidade na manutenção do código do programa.

- **FTensor** [14]: é uma biblioteca de alta performance escrita em linguagem C++ para tratamento de tensores.

- **Scilab** [15]: ambiente gráfico que permite a resolução de problemas numéricos tais como problemas da álgebra linear, polinômios, integração numérica, tratamento de sinais, visualização gráfica, etc. Na seção 3.5 apresentamos maiores detalhes sobre o Scilab.

- **GOOSE** (*The GNU Object-Oriented Statistics Environment*) [16]: é biblioteca escrita em C++ dedicada a cálculos estatísticos. Esta biblioteca usa as vantagens oferecidas pelo C++ para fornecer uma interface clara e facilitada ao programador para lidar com problemas estatísticos.

Além dessas bibliotecas e softwares que citamos acima, há muitas outras que são apresentadas em [17]. Em nosso trabalho, utilizamos apenas bibliotecas que tratam de problemas de álgebra linear.

2.1.1 Bibliotecas para álgebra linear

Um desenvolvimento possível de uma aplicação numérica de álgebra linear pode ser resumido da seguinte forma:

1. Descrever em termos de álgebra linear o problema a ser resolvido.
2. Selecionar a biblioteca que melhor descreve o problema.
3. Traduzir o problema para uma linguagem que suporte a biblioteca e escolher as rotinas a utilizar.

No terceiro passo, podemos conseguir uma certa otimização escolhendo rotinas da biblioteca que estejam relacionadas à forma da matriz que é usada no cálculo, de forma a usar suas propriedades algébricas. Assim, para matrizes esparsas, convém utilizar as rotinas que tratam desse tipo de matriz.

A comunidade que trabalha no desenvolvimento de bibliotecas criou o termo **bibliotecas tradicionais** para as bibliotecas de álgebra linear desenvolvidas usando a metodologia *top-down* em linguagens imperativas, principalmente Fortran. Exemplos dessas bibliotecas são LINPACK (*Linear Algebra Package*) [18], EISPACK (*Eigensystem Package*) [19], LAPACK (*Linear Algebra Package*) [20], BLAS (*Basic Linear Algebra Subprograms*) [21, 22] e ScaLAPACK (*Scalable LAPACK*)

[3, 4]. Essas duas últimas possuem versões que utilizam orientação ao objeto, a LAPACK++ [23] e ScaLAPACK++ [24] (ainda não implementada).

EISPACK

EISPACK [19] é uma coleção de rotinas em Fortran que calculam autovalores e autovetores de nove classes de matrizes: complexa, complexa hermitiana, real, real simétrica, real simétrica densa, real simétrica tridiagonal, real especial tridiagonal, real generalizada e real generalizada simétrica.

Essa biblioteca foi inicialmente desenvolvida na linguagem Algol, na década de 60 e teve sua *release* em Fortran em 1972, tendo grande sucesso na comunidade científica.

LINPACK

LINPACK [18] é uma coleção de rotinas em Fortran que analisam e resolvem sistemas de equações lineares e problemas de mínimos quadrados. As matrizes que podem ser tratadas são dos tipos gerais, densas, simétricas indefinidas, simétricas positivas definidas, triangular e triangular quadrada.

A biblioteca LINPACK é organizada ao redor que quatro fatorizações matriciais: fatorização LU, fatorização de Cholesky, fatorização QR e decomposição de valores (todos esses métodos são apresentados em [25]).

Uma característica interessante que foi implementada ao LINPACK é a orientação de seus algoritmos à coluna, ou seja, ele orienta *arrays* como colunas. A escolha desse método é devida à forma que o Fortran faz o armazenamento de matrizes na memória. Como ele usa armazenamento por coluna, isto otimiza a execução do código LINPACK.

O LINPACK usa as rotinas de nível um do BLAS (que serão discutidas nas seções a seguir) e isso causa algumas influências no desempenho, pois para ma-

trizes pequenas, com um número de elementos da ordem de 25, o desempenho é prejudicado devido ao *overhead* que o BLAS introduz. No entanto, para matrizes grande, esse efeito é desprezível [26].

Com o desenvolvimento do LAPACK, o LINPACK teve seu uso reduzido.

BLAS

O BLAS é constituído de rotinas básica de algebra linear e oferece bom desempenho em diversas arquiteturas, sendo que ele possui versões otimizadas, onde suas bibliotecas são escritas em código de máquina. Ele é constituído de três níveis de operações:

- **Nível 1:** Operações vetor-vetor: $y = \alpha x + y$,
- **Nível 2:** Operações matriz-vetor: $y = \alpha Ax + \beta y$,
- **Nível 3:** Operações matriz-matriz: $C = \alpha AB + \beta C$.

Sendo A, B e C são matrizes, x e y vetores, e α, β escalares. Mais informações sobre o BLAS, bem como suas rotinas podem ser vistas em [22].

LAPACK

O LAPACK [20] é uma biblioteca de álgebra linear constituída de rotinas para resolução de sistemas de equações lineares, problemas de mínimos quadrados, de autovalores e autovetores. Essas rotinas podem operar sobre matrizes reais ou complexas, usando precisão simples ou dupla.

O objetivo inicial do LAPACK foi usar as rotinas do EISPACK e LINPACK em computadores vetoriais ou de memória compartilhada de forma eficiente. Isso se deu porque essas duas bibliotecas apresentam certo sucesso entre a comunidade científica, mas não ofereciam um bom desempenho nessas máquinas, uma vez que

elas não consideram a hierarquia de memória existente nelas, gastando mais tempo no envio de dados do que nas operações de ponto flutuante.

No LAPACK, há o uso de algoritmos de divisão em bloco (que discutiremos nos próximos capítulos) que são usados para efetuar operações com matrizes, de forma que a hierarquia de memória é aproveitada para otimizar os cálculos.

Com o desenvolvimento do LAPACK, houve um decréscimo no uso do LINPACK e EISPACK, uma vez que ele supera esses dois em algumas características, tais como velocidade, precisão, robustez e funcionalidade, o que é conseguido através do uso das rotinas de nível três do BLAS, além de possuir um maior número de rotinas para operações de álgebra linear.

ScaLAPACK

O LAPACK vinha apresentando um grande sucesso entre a comunidade científica. No entanto, ele não apresentava um bom desempenho em computadores de memória distribuída e essas máquinas vinham se tornando cada vez mais utilizadas. Surgiu então a necessidade de se criar uma biblioteca que fosse parecida com o LAPACK, mas que oferecesse bom desempenho em computadores de memória distribuída.

O desenvolvedores criaram então o ScaLAPACK [3], que é constituído de rotinas que resolvem os mesmos problemas que as rotinas do LAPACK, além de outras que fazem a comunicação e divisão das matrizes entre os processos.

Para tornar o ScaLAPACK parecido com o LAPACK, foi criada a biblioteca de comunicação BLACS (*Basic Linear Algebra Communication*) [3, 4] além do PBLAS (*Parallel BLAS*) [3, 4]. Além disso, o ScaLAPACK usa rotinas do LAPACK em algumas operações seqüências nos diversos processadores. O capítulo 4 descreverá o ScaLAPACK em detalhes.

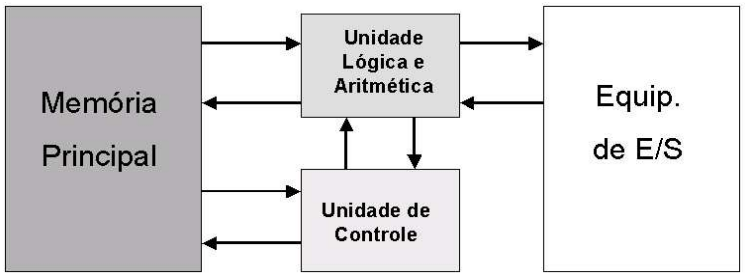

Figura 2.1: Arquitetura de von Neuman, proposta por ele na década de 50.

2.2 Computação Paralela

O modelo computacional de von Neuman foi fundamental no desenvolvimento dos computadores. No entanto, ele oferece certas limitações que prejudicam seu desempenho, como o gargalo que surge no acesso à memória. Isso ocorre porque há apenas um barramento de acesso, de forma que apenas um componente computacional pode ser acessado por vez (ver figura 2.1). Além disso, o modelo sugere que o processador execute as instruções de forma seqüencial, causando ociosidade em certos componentes, pois nem mesmo a tarefa de busca de uma instrução na memória e a execução de outra, que são tarefas independentes, podem ser executadas em paralelo.

Com o desenvolvimento de novas tecnologias, essas limitações foram superadas e melhor desempenho foi obtido. Dentre esses avanços, podemos citar o uso de múltiplos registradores no lugar de um único acumulador, uso de memória *cache*, *pipeline* de instruções e organização superescalar. Estes dois últimos fatores aumentaram o desempenho em um fator de dez em relação aos computadores puramente seqüenciais [27].

Outro fator que possibilitou o surgimento de computadores mais rápidos foi a diminuição no tamanho dos transistores e o conseqüente aumento da freqüência de clock. Com isso, o número de transistores nos processadores seguiu a famosa

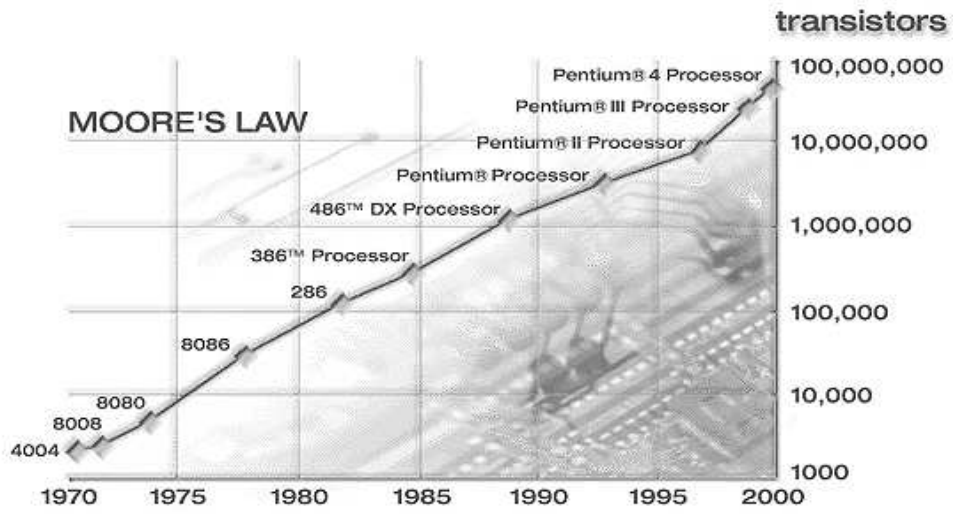

Figura 2.2: Gráfico que mostra o aumento do número de transistores nos processadores da Intel, obedecendo à lei de Moore. Fonte Intel [29].

lei de Moore [28], que diz que este número dobra a cada dezoito meses, conforme notamos na figura 2.2. No entanto, esse aumento é limitado, pois os circuitos eletrônicos não podem ser melhorados indefinidamente e a redução do tamanho dos transistores será limitada por barreiras físicas, quando o número de átomos no transistor for tão pequeno que efeitos quânticos passarem a ser importantes. Por exemplo, em um transistor de 0,07 μm, cujo lançamento foi anunciado pela Intel para 2005 [29], cada um dos três terminais que compõe um transistor terá apenas três átomos de cobre de espessura. Desta forma, o sinal elétrico para mudar o estado do transistor será composto de apenas algumas dezenas de elétrons. Com transistores deste tamanho, a Intel espera chegar a 15 Ghz de freqüência de clock até 2007 [29]. Transistores menores do que 0,07 μm pertencerão à computação quântica, ainda pouco desenvolvida. Além dessa limitação que tende a atrapalhar o avanço dos computadores seqüenciais. temos o fato de que a velocidade de acesso à memória não cresce no ritmo da lei de Moore.

Apesar de todo esse avanço na tecnologia dos computadores seqüenciais, eles não conseguem suprir a necessidade de poder computacional que certas áreas científicas exigem. Aplicações em física, química ou engenharia, como as simulações de fenômenos quânticos, construção de modelos climáticos ou mesmo no projeto de aeronaves, têm uma demanda por poder computacional que essas máquinas não oferecem [30].

Com isso, uma solução encontrada para satisfazer essa demanda são os computadores paralelos [31]. Segundo Almasi [32] "um computador paralelo é uma coleção de elementos de processamento que se comunicam e cooperam na resolução de grandes problemas de forma rápida". Desta forma, enquanto que nos computadores seqüenciais a velocidade do processador é o principal fator na determinação do desempenho, nos computadores paralelos este é apenas um dos fatores. Além disso, os computadores paralelos podem agregar recursos de computadores seqüenciais, como o caso dos *clusters*. Assim, ao invés de se tentar construir um computador com um único processador cujo período de processamento seja de 0,001 ns, pode-se usar uma máquina paralela com 1000 processadores, tendo eles um período de 1 ns cada, que é uma tarefa mais simples.

Neste contexto, Flynn definiu em 1972 uma classificação para os modelos computacionais baseada em dois conceitos, seqüência de instruções e de dados [27], que é dividida da seguinte forma:

- **SISD** - *Single instruction stream, single data stream*: Um único processador executa uma única seqüência de instruções, usando dados armazenados em uma única memória. Um computador de um único processador pertence à essa categoria.

- **SIMD** - *Single instruction stream, multiple data stream*: Uma única instrução de máquina controla a execução simultânea de um certo número de elementos de processamento em passos de execução. Cada elemento de

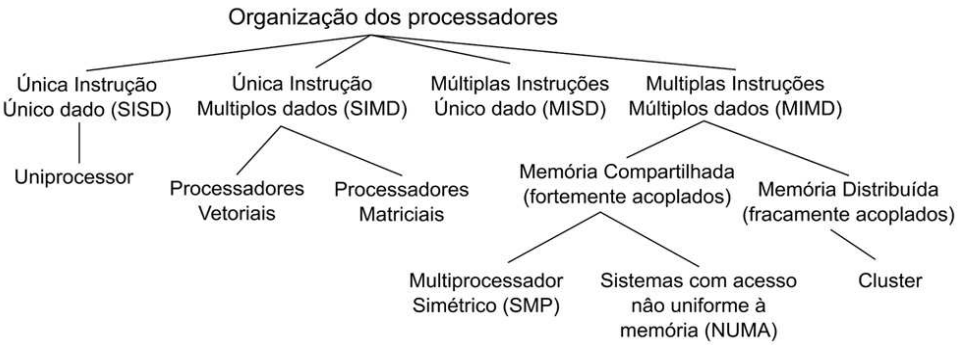

Figura 2.3: Taxonomia de organização de computadores paralelos. Fonte [33].

processamento possui uma memória de dados associada, de forma que cada processador executa as mesmas instruções sobre dados diferentes. Os computadores vetoriais pertencem a essa categoria.

- **MISD** - *Multiple instruction stream, single data stream*: Um conjunto de dados é transmitido para diferentes processadores e cada um deles executa diferentes operações sobre ele. Uma arquitetura desse tipo nunca foi implementada.

- **MIMD** - *Multiple instruction stream, multiple data stream*: Múltiplos dados são transmitidos para diferentes processadores que realizam diferentes operações sobre eles. Os *clusters* pertencem a essa categoria.

Na figura 2.3 é mostrada a taxonomia dos computadores levando-se em conta a organização sugerida por Flynn. Os modelos SIMD possuem várias unidades lógicas e aritméticas que podem executar um mesmo conjunto de operações sobre diversos dados em paralelo. Seu desenvolvimento foi impulsionado pela necessidade de aplicações científicas que requerem operações sobre matrizes e vetores com grande número de elementos [33].

Figura 2.4: Fotos do Earth Simulator, localizado no Japão.

Os primeiros computadores vetoriais foram fabricados em 1976 pela *Cray Research*, inicialmente com o Cray-1 e depois com o C90 e T90. Esse tipo de máquina dominou o mercado da computação científica durante décadas. No entanto, devido ao alto custo, e ao aumento no desempenho de computadores seqüenciais, que podem ser usados na construção de máquinas paralelas, seu uso tornou-se limitado. Hoje, são poucas a empresas que ainda produzem este tipo de máquina. O último computador paralelo que utiliza processamento vetorial que apresentou grande sucesso foi o *Earth Simulator* [34] (ver figura 2.4). Este computador, construído pela NEC e localizado no Japão, apresenta uma performance superior a 35 Tflop/s e é o que possui maior poder computacional atualmente [35]. Na tabela 2.2 é mostrada uma tabela com os computadores mais rápidos do mundo ao longo do tempo.

No caso dos modelos MIMD, temos os computadores de memória compartilhada (*multiprocessadores*) e de memória distribuída (*multicomputadores*), conforme observamos na figura 2.3. No primeiro casos temos, por exemplo, alguns tipos de *workstations* e no segundo os *clusters*.

Um multiprocessador é um sistema que possui vários processadores e apenas um espaço de endereçamento visível a eles, ou seja, todos os processadores têm acesso a todos os módulos de memória e aos dispositivos de entrada e saída. Um

Tabela 2.1: Computadores mais rápidos do mundo. Fonte [1].

Período	Supercomputador	Velocidade	Localização
1943-1944	Colossus		Bletchley Park, England
1945-1950	Manchester Mark I		University of Manchester, England
1950-1955	MIT Whirlwind		Massachusetts Institute of Technology, Cambridge, MA
1955-1960	IBM 7090	210 KFLOPS	U.S. Air Force BMEWS (RADC), Rome, NY
1960-1965	CDC 6600	10.24 MFLOPS	Lawrence Livermore Laboratory, California
1965-1970	CDC 7600	37.27 MFLOPS	Lawrence Livermore Laboratory, California
1970-1975	CDC Cyber 76		
1975-1980	Cray-1	160 MFLOPS	Los Alamos National Laboratory, New Mexico (1976)
1980-1985	Cray X-MP	500 MFLOPS	Los Alamos National Laboratory, New Mexico
1985-1990	Cray Y-MP	1.3 GFLOPS	Los Alamos National Laboratory, New Mexico
1990-1995	Fujitsu Numerical Wind Tunnel	236 GFLOPS	National Aerospace Lab
1995-2000	Intel ASCI Red	2150 GFLOPS	Sandia National Laboratories, New Mexico
2000-2002	IBM ASCI White, SP Power3 375 MHz	7226 GFLOPS	Lawrence Livermore Laboratory, California
2002-2003	Earth Simulator	35 TFLOPS	Yokohama Institute for Earth Sciences, Japan

fator encorajador no uso dessas máquinas é a facilidade de se escrever programas para elas, sendo que em alguns casos esses podem ser idênticos aos seqüenciais. No entanto, algumas limitações de desempenho desfavorecem seu uso [36].

Já nos multicomputadores, temos diversos nós que são compostos por uma memória local e uma ou mais unidades de processamento ligados por uma rede de interconexão de alta velocidade. O número de unidades de processamento pode variar, dependendo da máquina – os supercomputadores MPPs (*Massively Parallel Processors*) podem ter milhares de unidades de processamento. Essas máquinas são bastante caras, podendo custar alguns milhões de dólares e geralmente apenas alguns laboratórios de pesquisa no mundo possuem tais modelos.

Com o avanço no poder de processamento dos computadores e na velocidade das redes de conexão, como a Ethernet ou ATM, surgiram os *clusters*, que são compostos de algumas dezenas ou centenas de computadores pessoais, ou estações de trabalho, interligados por uma rede de alta velocidade. O fator que impulsiona o uso dessas máquinas é a relação custo/desempenho, que pode ser de três a dez vezes melhor do que um supercomputador tradicional [36].

Um *cluster* pode ser definido como um grupo de computadores completos interconectados, com um sistema operacional distribuído, trabalhando conjuntamente, podendo criar a ilusão de ser uma única máquina. Cada computador que compõe o *cluster* é chamado nó e é um computador completo, podendo operar

normalmente, independente do *cluster*. Esses nós são compostos de um ou mais processadores e podem ter a mesma configuração (*cluster* homogêneo) ou não (*cluster* heterogêneo).

Os *clusters* possuem certos benefícios devido ao seu tipo de organização [33].

- **Escalabilidade absoluta:** a configuração do *cluster* permite a construção de uma máquina com um número elevado de nós, podendo ser da ordem de milhares, ultrapassando a capacidade de computação das modernas máquinas individuais.

- **Escalabilidade incremental:** Novos recursos podem ser agregados ao *cluster* de forma facilitada, como mais nós ou uma rede de interconexão mais rápida, sem que os recursos antigos tenham que ser completamente desprezados.

- **Alta disponibilidade:** Como cada nó é um computador independente, uma falha em um deles não compromete a perda de serviço, pois o tratamento à falhas pode ser feito pelo software.

- **Melhor relação custo desempenho:** Devido ao avanço dos recursos para computadores pessoais e à redução de preços destes, é possível construir um *cluster* com poder de computação igual ou maior ao de uma máquina de grande porte, com custo muito menor.

Devido à variedade de configurações possíveis, os *clusters* podem ser divididos em dois tipos. Um deles é chamado COW (*Cluster of Workstations*) que são estações de trabalho ou computadores pessoais espalhados por um prédio ou uma sala e interligadas por uma rede de alta velocidade. Geralmente os computadores são heterogêneos e são utilizados em computação paralela apenas quando estão ociosos. O uso dessas estações apenas nesta circunstância pode implicar numa

migração de tarefas e esta acrescenta uma complexidade no desenvolvimentos do software [27].

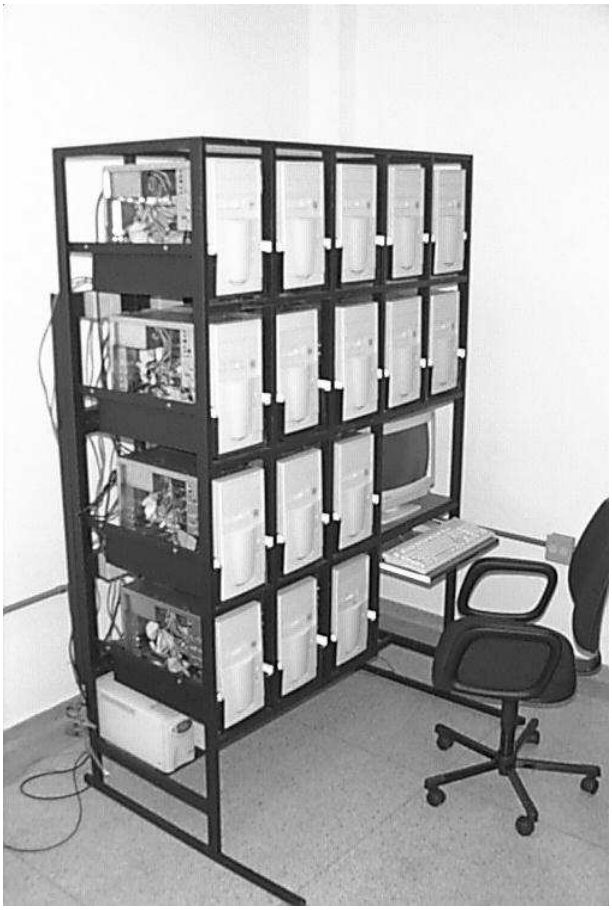

Figura 2.5: Cluster do Laboratório de Processamento Paralelo Aplicado (LPPA), utilizado no desenvolvimento do trabalho apresentado nesta dissertação.

Outro tipo comum de *clusters* são os dedicados, conhecidos como *Beowulf* [37], que é o nome do herói épico anglo-saxão de um poema escrito em inglês arcaico, de grande coragem, que derrotou um monstro chamado Grendel. Beowulf era um guerreiro de vivia na Finlândia e ao ouvir do monstro Grendel, que atacava o reino de Hrothgar, na Dinamarca, juntou quatorze guerreiros e juntos foram para

lá para tentar acabar com o monstro. Grendel era uma criatura bastante peluda, como um urso, maior que um homem e possuía grande força. Bewoulf o derrotou e depois voltou para a Finlândia, onde foi proclamando rei. Reinou por mais de cinqüenta anos e morreu depois de ajudar a derrotar um dragão naquele país.

Os *beowulfs* são constituídos, geralmente, de computadores homogêneos e são dedicados, ou seja, não possuem teclados, mouses ou monitores, sendo que apenas o nó servidor pode possuir tais recursos. O único acesso ao nó cliente é feito via conexão remota pelo nó servidor. Além disso, eles estão ligados por uma rede privada e apenas o servidor pode ter acesso à rede externa. Ao contrário dos COWs, que geralmente são usados à noite ou quando as pessoas não estão usando seus computadores, o *Beowulf* é dedicado apenas a computações paralelas e é otimizado para esse propósito.

Em nosso trabalho utilizamos um *Beowulf* de 16 nós dedicados, homogêneos, sendo cada nó um computador K6-III com 450 Mhz de freqüência de clock, 256 MB de memória SDRAM, com discos rígidos de 10 GB, e interconectados por uma *switch* de alta velocidade, Fast-Ethernet (100 Mb/s). Na figura 2.5 é mostrado o *cluster* que utilizamos.

2.2.1 Softwares paralelos

O desenvolvimento de softwares paralelos pode ser feito de duas formas básicas, utilizando uma linguagem paralela ou utilizando uma linguagem seqüencial conjuntamente com uma biblioteca paralela, que permita a troca de mensagens entre os processadores.

Dentre as linguagens paralelas, pode-se usar linguagens funcionais, como a Haskell [38] e Sisal [39], ou linguagens lógicas, como o Prolog [40]. Essas linguagens evitam a seqüencialização do código, especificando apenas as relações entre as operações e não a ordem em que devem ocorrer. No entanto, o desen-

volvimento de código usando essas linguagens pode ser desencorajador, pois o programador tem que aprender uma nova linguagem. Além disso, a extração do paralelismo pode não ser simples em certos casos, pois o balanceamento de carga e o uso de granularidade grossa podem ser difíceis de serem alcançados em certo problemas.

A outra possibilidade vem se mostrando mais viável, que é a utilização de uma linguagem seqüencial e bibliotecas paralelas. Neste caso, cabe ao programador apenas aprender a manusear a biblioteca e usar a linguagem seqüencial que preferir. Há diversas bibliotecas que oferecem ao programador recursos para fazer a comunicação entre os processos. Umas delas é o PVM (*Paralell Virtual Machine*) [41]. Outras são as que foram construídas segundo o padrão MPI (*Message Passing Interface*) [42], tal como o MPICH [43] e o LAM [44]. Com isso, pode-se desenvolver softwares paralelos eficientes e bibliotecas paralelas específicas para determinadas computações, como o ScaLAPACK, que é uma biblioteca de álgebra linear que utiliza bibliotecas do padrão MPI ou o PVM para fazer a comunicação (maiores detalhes no capítulo 4).

2.2.2 MPI - *Message Passing Interface*

MPI é um padrão de biblioteca de passagem de mensagem para sistemas paralelos. Seu desenvolvimento procurou fornecer uma interface comum a diversas plataformas, de forma que um programa possa rodar em uma máquina MPP ou mesmo em um COW. Assim, o padrão MPI, definido pelo comitê de padronização *MPI Forum* [42], especifica apenas a interface de programação e não a implementação, ficando esta por conta do implementador.

Os sistemas baseados na troca de mensagem devem ter no mínimo dois processos rodando independentemente e a comunicação entre eles é feita através de rotinas `send` e `receive`. Essa comunicação pode se dar de três formas diferentes:

- Troca de mensagem síncrona.
- Troca de mensagem com ajuda de um *buffer*.
- Troca de mensagem não-bloqueada.

Na troca de mensagem síncrona, quando o transmissor realiza uma operação de envio (`send`), ele fica bloqueado até que o receptor realize uma operação de recepção (`receive`). Assim, que esta é feita, ele continua a execução do programa. No caso da troca de mensagem com ajuda de um buffer, quando a mensagem é enviada pelo transmissor, ela é armazenada em um *buffer* até que o receptor a recolha. Desta forma, o transmissor não fica bloqueado, continuando a execução do seu código após o `send`. No entanto, se o *buffer* estiver cheio, uma mensagem de erro é enviada pelo compilador. Já o último caso, o de troca de mensagem não-bloqueada, o transmissor pode continuar a execução do código após a chamada `send`, não ficando bloqueado em hipótese alguma. Se o envio não puder ser feito no momento da chamada, o software informa ao sistema operacional que a chamada `send` pode ser efetuada mais tarde, quando houver tempo.

Além das rotinas de comunicação ponto a ponto, o MPI também fornece rotinas para comunicações coletivas, como *broadcast*, barreira, coleta, distribuição, redistribuição, redução e prefixo. Para maiores informações sobre as rotinas do MPI, ver [42].

O pacote MPI é baseado em quatro conceitos básicos:

1. **Processos**: é o elemento básico de computação em MPI. Um processo pode ser comparado a um programa seqüencial em execução, sendo que este pode se comunicar com outros processos através da troca de mensagens. Assim, em MPI temos diversos processos rodando em paralelo, cooperando para a execução de uma determinada tarefa.

2. **Comunicadores**: são usados para interligar diversos processos, criando um conjunto de comunicação. Os processos que se comunicam devem estar associados a um mesmo comunicador. O comunicador padrão usado pelo MPI é chamado `MPI_COMM_WORLD`, sendo que o usuário pode definir outros comunicadores. Os processos associados a um mesmo comunicador são identificados pelo *rank*, que é um inteiro que varia de 0 a $P-1$, sendo P o número de processos no comunicador.

3. **Mensagens**: As mensagens são os dados trocados entre os processos através das diretivas de comunicação, mais as informações de identificação, que especificam o envelope da mensagem. Essas informações definem os processos destinatário e remetente, um inteiro denominado TAG, que é utilizado para distinguir mensagens provenientes de um mesmo processo, e o comunicador utilizado.

4. **Tipos de Dados**: Os dados transmitidos na mensagem são de um determinado tipo (inteiro, ponto flutuante, caráter, etc.) e este tipo pode ser representado diferentemente em arquiteturas distintas. Desta forma, se um *cluster* for constituído de nós com arquiteturas distintas, quando os dados são transmitidos de um nó para outro, o MPI faz a conversão automática dos dados de forma que sejam tratados de forma correta.

O MPI pode ser usados nas linguagens FORTRAN 77 e C. Já o MPI-2, além da linguagens mencionadas, pode ser usado em Fortran90 e C++.

No nosso trabalho, utilizamos a primeira versão do MPI, denominada MPI-1. Esta versão foi definida em 1995 e aperfeiçoada em 1997, com a definição do MPI-2 [45], que permite novas operações, como suporte para processos dinâmicos, que trata da criação e gerência de processos como faz o PVM (*Parallel Virtual Machine* [41]), comunicação coletiva não bloqueada, entrada/saída esca-

lável, processamento em tempo real, etc. Ver [45] para maiores detalhes.

Existem diversas implementações para o MPI, pois o padrão especifica apenas a interface, conforme discutimos anteriormente. Desde o lançamento da primeira versão de MPI, em 1995, surgiram diversas implementações para redes de estações de trabalho [2], sendo que as mais utilizadas atualmente são a MPICH [43] e a LAM [44]. Elas são praticamente equivalentes e o uso é bastante semelhante. Em nosso trabalho, usamos a implementação MPICH.

Na utilização do MPI é fornecido ao programador um conjunto de rotinas que permite realizar a comunicação e sincronizar as operações entre os processos. No entanto, o uso do MPI para tarefas complexas pode ser árduo e o programador tem que se preocupar com detalhes do MPI, como o instante que deve ocorrer comunicações entre os processos, quais os tipos do dados a serem enviados, se a comunicação deve ser bloqueante ou não, entre outros. Esses detalhes atrapalham no desenvolvimento do código, pois o programador tem que desviar a atenção do desenvolvimento do código para detalhes de comunicação entre os processos.

De modo a facilitar muitas dessas tarefas complexas, foram construídas algumas bibliotecas numéricas paralelas, que usam MPI na sua implementação, ficando seus detalhes escondidos do programador. Uma dessas bibliotecas é o pacote ScaLAPACK, que foi rapidamente discutido nessa seção e será apresentado em detalhes no capítulo 4.

CAPÍTULO
3
Álgebra Linear Numérica

Após a Segunda Guerra Mundial, de 1945 em diante, o uso de computadores na execução de cálculos matemáticos se tornou comum. A partir daí, muitos estudos foram feitos na obtenção de algoritmos numéricos para álgebra linear que oferecessem um resultado preciso e um tempo de execução mínimo. Na sua construção, eles usam propriedades matemáticas das matrizes de forma a facilitar o desenvolvimento do código e conseguir softwares otimizados.

Neste capítulo, vamos introduzir conceitos gerais sobre matrizes tal como as operações fundamentais, propriedades e problemas mais gerais, como os sistemas lineares. Além disso, vamos mostrar como as propriedades das matrizes podem ser exploradas para a obtenção de algoritmos eficientes. Finalmente examinaremos o uso das bibliotecas LAPACK e BLAS e do software Scilab.

3.1 Conceitos fundamentais

Álgebra linear numérica está estritamente relacionada a operações sobre matrizes, que podem ser divididas em dois grupos. O primeiro consiste de operações básicas, tal como soma, multiplicação, transposição, etc. Já o segundo, trata de sistemas de equações lineares, problemas de autovalores e problemas de mínimos

quadrados, que são mais complexos do que as operações do primeiro grupo. Nas próximas seções vamos apresentar as notações e as operações algébricas que usam matrizes.

Matrizes

Os espaço vetorial dos números reais é representado por $\mathbf{R}$. Nós denotamos o espaço de dimensão $\mathbf{R}^{m\times n}$ como sendo o das matrizes com m linhas e n colunas. Assim:

$$A \in \mathbf{R}^{m\times n} \Longleftrightarrow A = (a_{ij}) = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix}.$$

Os elementos a_{ij} são ditos elementos da matriz e podem ser reais ou complexos, sendo que neste último caso o espaço é representado por $C^{m\times n}$. Quando $m = n$, a matriz é chamada matriz quadrada de ordem n. Nos casos especiais em que $m = 1$ ou $n = 1$, a matriz é chamada matriz linha ou matriz coluna, respectivamente, sendo análoga a um vetor. No entanto, para o caso dos vetores, temos a seguinte notação:

$$x \in \mathbf{R}^n \Longleftrightarrow x = (x_1, \dots, x_n).$$

Neste caso x_i, $i = 1, ..., n$ representam os elementos do vetor. Neste trabalho, vamos representar matrizes por letras maiúsculas ($A, B,$ etc), vetores por minúsculas ($x, y, z,$ etc) e as quantidades escalares por letras gregas ($\alpha, \beta,$ etc).

Algumas operações básicas sobre as matrizes podem ser vistas na tabela da figura 3.1

Definidas as operações sobre matrizes, vamos apresentar as propriedades gerais de matrizes reais e complexas, que podem ser observadas a seguir:

- *Adição comutativa*: $A + B = B + A$.

Tabela 3.1: Operações matriciais e vetoriais básicas.

Operação	**Notação**	**Definição**
Norma do vetor	$\|\|x\|\|_p$	$\alpha \leftarrow (\sum_i \|x_i\|^p)^{1/p}$
	$\|\|x\|\|_\infty$	$\alpha \leftarrow max_i \|x_i\|$
Norma da Matriz	$\|\|A\|\|_1$	$\alpha \leftarrow max_j \sum_i \|a_{ij}\|$
	$\|\|A\|\|_F$	$\alpha \leftarrow (\sum_{ij} \|a_{ij}\|^2)^{1/2}$
Vetor transposto	x^T	
Matriz transposta	A^T	
Matriz inversa	A^{-1}	
Produto escalar	$\alpha \leftarrow x^T y$	$\alpha \leftarrow \sum_i x_i y_i$
Multiplicação por escalar	$y \leftarrow \alpha x$	$y_i \leftarrow \alpha x_i$
Adição de vetores	$z \leftarrow x + y$	$z_i \leftarrow x_i + y_i$
Multiplicação matriz-vetor	$y \leftarrow Ax$	$y_i \leftarrow \sum_j a_{ij} x_j$
Multiplicação por escalar	$C \leftarrow \alpha A$	$c_{ij} \leftarrow \alpha a_{ij}$
Adição de matrizes	$C \leftarrow A + B$	$c_{ij} \leftarrow a_{ij} + b_{ij}$
Multiplicação matriz-matriz	$C \leftarrow AB$	$c_{ij} \leftarrow \sum_k a_{ik} b_{kj}$

- *Adição associativa*: $A + (B + C) = (A + B) + C$.
- *Lei distributiva*: $\alpha(A + B) = \alpha A + \alpha B$.
- *Multiplicação não comutativa*: geralmente $AB \neq BA$.
- *Multiplicação associativa*: $\alpha A(BC) = (\alpha AB)C = (AB)\alpha C$.
- *Transposição*: $(A + B)^T = A^T + B^T$, $(AB)^T = B^T A^T$.
- *Conjugação*: $(\alpha A + B)^* = \alpha^* A^* + B^*$.

Estas operações são usadas principalmente na construção de bibliotecas básicas de álgebra linear, como no caso do BLAS [21].

3.2 Tipos de matrizes

Além das operações vistas na seção anterior, pode-se usar certas propriedades das matrizes na construção de softwares numéricos para álgebra linear. As matrizes podem ser classificadas de acordo com a distribuição de seus elementos ou mesmo quanto a certas operações que apresentam simetrias, como no casos das matrizes simétricas. Esta classificação é adotada pelos pacotes LAPACK e ScaLAPACK na construção de suas rotinas, e é apresentada a seguir.

Matrizes gerais

Uma matriz geral é aquela que não apresenta uma estrutura especial de distribuição de seus elementos, bem como qualquer simetria. Desta forma, nenhum ganho na construção de softwares pode ser obtido usando uma distribuição de dados especial, sendo que o armazenamento deve ser feito no formato padrão, usando um *array* bi-dimensional segundo a linguagem de programação.

Matrizes de banda gerais

Uma matriz de banda geral tem seus elementos não nulos organizados uniformemente ao redor da diagonal. Na figura 3.1 temos dois exemplos desse tipo de matriz. Como podemos ver, os elementos podem ser determinados por: $a_{ij} = 0$ se $(i - j) > ml$ ou se $(j - i) > mu$, onde ml e mu são as larguras de banda inferior e superior. A soma $ml + mu + 1$ é a largura de banda total da matriz.

No caso em que $ml = mu = 1$, temos uma matriz tridiagonal.

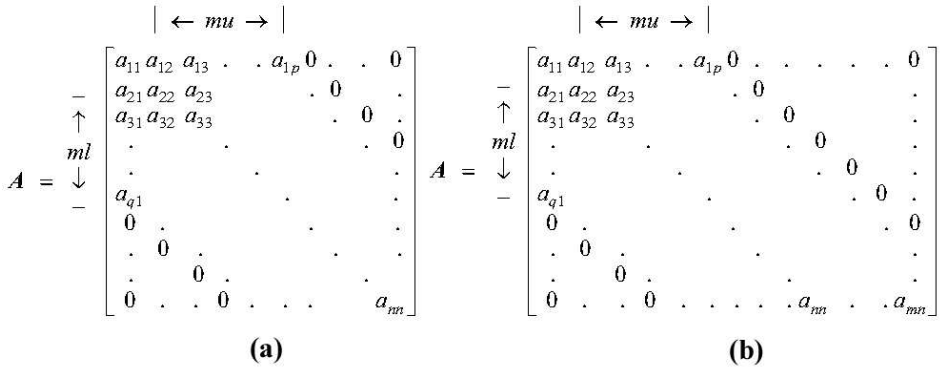

Figura 3.1: Matrizes de banda gerais. Em (a) temos uma matriz quadrada e em (b) uma matriz retangular.

Matrizes tridiagonais

Numa matriz tridiagonal, seus únicos elementos não nulos estão na diagonal principal e nas duas diagonais vizinhas a ela. Assim, os elementos podem ser definidos por $a_{ij} = 0$ se $|i - j| > 1$. Na figura 3.2 temos um exemplo deste tipo de matriz.

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & . & . & . & 0 \\ a_{21} & a_{22} & a_{23} & 0 & & & . \\ 0 & a_{32} & a_{33} & a_{34} & 0 & & . \\ . & 0 & a_{43} & a_{44} & . & . & . \\ . & & 0 & . & . & . & \\ . & & & . & . & . & . \\ & & & & . & . & . \\ 0 & . & . & . & & . & a_{nn} \end{bmatrix}$$

Figura 3.2: Matriz tridiagonal. Os elementos não nulos estão na diagonal principal e nas duas diagonais vizinhas a ela.

Matrizes triangulares

Há dois tipos de matrizes triangulares, a triangular superior e a triangular inferior. Estas matrizes devem ter o mesmo número de linhas e colunas. Considerando uma matriz U triangular superior, seus elementos são definidos por $u_{ij} = 0$ se $i > j$.

$$\boldsymbol{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & & \vdots \\ 0 & 0 & u_{33} & & \vdots \\ \vdots & & & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & u_{nn} \end{bmatrix} \qquad \boldsymbol{L} = \begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & & \vdots \\ l_{31} & l_{32} & l_{33} & & \vdots \\ \vdots & & & \ddots & \vdots \\ l_{n1} & \cdots & \cdots & \cdots & l_{nn} \end{bmatrix}$$

Figura 3.3: Exemplos de matrizes triangulares. Do lado esquerdo da figura temos uma matriz triangular superior e do lado esquerdo uma inferior.

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{21} & 0 & \cdots & \cdots & 0 \\ a_{21} & a_{22} & a_{32} & 0 & & \vdots \\ 0 & a_{32} & a_{33} & a_{43} & 0 & \vdots \\ \vdots & 0 & a_{43} & a_{44} & \ddots & \vdots \\ \vdots & & 0 & \ddots & \ddots & \\ \vdots & & & \ddots & \ddots & \\ 0 & \cdots & \cdots & \cdots & \cdots & a_{nn} \end{bmatrix}$$

Figura 3.4: Uma matriz simétrica tridiagonal tem elementos não nulos ao redor da diagonal principal e é igual à sua transposta.

Já no caso de uma triangular inferior L, temos que seus elementos são definidos por $l_{ij} = 0$ se $i < j$. Na figura 3.3 temos esses dois casos. Quando os elementos da diagonal são iguais a um, a matriz é chamada triangular unitária.

Matrizes simétricas

Uma matriz A é dita simétrica se ela é igual à sua transposta, $A = A^T$. Para isto, devemos notar que ela deve ser quadrada e os elementos a_{ij} deve ser iguais a a_{ji}.

Matrizes simétricas tridiagonais

Um caso especial de matrizes tridiagonais são aquelas são também simétricas. Assim, os elementos são dados por $a_{ij} = 0$ se $|i-j| > 1$ e $a_{ij} = a_{ji}$ se $|i-j| = 1$. Na figura 3.4 temos esse tipo de matriz.

Matrizes hermitianas

As matrizes hermitianas obedecem à propriedade de ser igual à transposta da sua conjugada. Assim, para uma matriz A ser hermitiana, ela deve obedecer a relação: $A = (A^T)^* = A^\dagger$.

Matrizes simétricas ou hermitianas de banda positivas definidas

Uma matriz simétrica ou hermitiana obedece às relações apresentadas anteriormente. No caso de elas serem de banda positivas definidas, o valor definido por $x^T Ax$ (ou $x^\dagger Ax$ no caso das hermitianas) é positivo para todo vetor x não nulo.

Matrizes simétricas ou hermitianas tridiagonais positivas definidas

Uma matriz simétrica (ou hermitiana) tridiagonal A é positiva definida se o valor $x^T Ax$ (ou $x^\dagger Ax$ no caso complexo) é positivo para todo vetor não nulo x.

Matrizes ortogonais

Se a transposta de uma matriz A é igual à sua inversa A^{-1} ela é dita ortogonal ($A^T = A^{-1}$). Desta forma, $A^T A = AA^T = A^{-1}A = AA^{-1} = I$, onde I é a matriz identidade, que tem os elementos da diagonal principal iguais a um e os outros elementos iguais a zero ($a_{ij} = 0$ se $i \neq j$ e $a_{ij} = 1$ se $i = j$).

Matrizes unitárias

Se a transposta da conjugada de uma matriz complexa A é igual à sua inversa, ela é dita unitária ($A^\dagger = (A^T)^* = A^{-1}$).

Matrizes trapezoidais

Uma matriz trapezoidal é essencialmente uma matriz retangular. Nos casos especiais em que $a_{ij} = 0$ se $i > j$ e $a_{ij} = 0$ se $i < j$, ela é dita trapezoidal superior ou trapezoidal inferior, respectivamente.

Matrizes esparsas

Matrizes esparsas são aquelas em que a maioria dos seus elementos são zero e os elementos não nulos não obedecem a nenhuma distribuição especial dentro da matriz.

Outros tipos de matrizes

Além desses tipos de matrizes, podemos ainda ter um classificação por blocos de acordo com a configuração destes. No entanto, como esse tipo de distribuição não é usada pela nossa biblioteca, vamos assumir essas matrizes como sendo gerais.

3.3 Operações algébricas sobre matrizes

Nesta seção vamos discutir algumas operações mais complexas que podem ser executadas sobre matrizes. Essas operações são todas suportadas pelo pacote ScaLAPACK.

3.3.1 Sistema de equações lineares

Um sistema de equações lineares é constituído por um número finito de equações lineares que pode ser expresso por

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1j}x_j + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2j}x_j + \ldots + a_{2n}n_n = b_2 \\ \quad \vdots \qquad \ddots \qquad \quad \vdots \qquad \quad \ddots \qquad \quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mj}x_j + \ldots a_{mn}x_n = b_m \end{cases} \tag{3.1}$$

Nestes sistemas, $a_{11}, a_{12}, \ldots, a_{mn}, b_1, b_2, \ldots, b_m$ são valores constantes. As incógnitas do problema são dadas por $x_1, x_2, \ldots, x_n$. Este sistemas pode ser convenientemente escrito com o uso de matrizes e vetores,

$$Ax = b \Rightarrow \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1j} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{1j} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mj} & \ldots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

Sistemas lineares possuem diversos métodos de resolução, sendo os mais comuns o que usa fatorização LU, método de eliminação de Gauss-compacto, método de Cholesky, eliminação de Gauss com pivotamento parcial, etc. Estes métodos podem ser vistos em [25, 46].

3.3.2 Autovalores e autovetores

Autovalores e autovetores estão presentes em muitos problemas matemáticos, físicos, de engenharia, química, etc. Na física por exemplo, os autovalores e autovetores aparecem em problemas de mecânica quântica, onde a aplicação de operadores a matrizes que descrevem o problema resultam nos auto-estados (autovetores) e energias (autovalores) do sistema [47].

Dada uma matriz A, $n \times n$, um vetor x é chamado *autovetor* de A se o produto Ax é um múltiplo de x e x tem ao menos um elemento não nulo, isto é:

$$Ax = \lambda x$$

para algum escalar λ. Este valor escalar é chamado *autovalor* de A, e x é dito autovetor de A correspondente a λ.

Métodos numéricos para encontrar os autovalores e autovetores de matrizes podem ser vistos em [25, 46]. Em [46] são apresentadas rotinas computacionais do pacote LAPACK que permitem efetuar essas operações numéricas.

3.3.3 Problemas de mínimos quadrados

Dado um sistema de equações lineares $Ax = b$ de m equações em n variáveis com $n \leq m$, devemos achar um vetor x que minimiza

$$||Ax - b||_2$$

Os métodos de obtenção da solução podem ser vistos em [25].

3.4 Construção de softwares numéricos

Como vimos na seção 2.1.1, a construção de softwares numéricos pode seguir três passos básicos. Outro fator importante que deve ser considerado, no caso de bibliotecas paralelas, é quanto à forma de distribuição da matriz. Esse tópico será abordado no capítulo 4 da dissertação.

3.5 BLAS, LAPACK e Scilab

Conforme já discutimos anteriormente, o BLAS é uma biblioteca que possui rotinas para operações básicas de álgebra linear. Uma lista completa das rotinas pode

ser vista em [21]. O seu uso é bastante simples, sendo necessário ao programador apenas instalar a biblioteca, escolher as rotinas a utilizar de acordo com os tipos de dados e o problema em questão, e definir os parâmetros de entrada da rotina.

Na figura 3.5 é mostrado um programa em linguagem C++ que utiliza a biblioteca BLAS (versão em Fortran). A operação realizada pelo programa é dada por: $y = \alpha Ax + \beta y$. Assim, é definida a matriz A e os vetores x e y e é chamada a função `dgemv`, do nível 2 do BLAS, que executa a operação desejada armazenando o resultado no vetor y. Note que a chamada é feita usando o caráter _ após o nome da função (`dgemv_`), pois a biblioteca, no exemplo considerado, é escrita em linguagem FORTRAN 77, sendo usada por um programa escrito em linguagem C++.

Para se fazer a compilação do programa, usamos o seguinte comando: `g++ ex_blas.cc -o ex_blas blas_LINUX.a`, sendo `ex_blas.cc` o nome do programa e `blas_LINUX.a` o arquivo que contém a biblioteca BLAS, que pode ser obtido em [22], onde há disponibilidade para diversas arquiteturas computacionais.

No caso do LAPACK, o processo é análogo, sendo necessário ao programador instalar o pacote LAPACK (disponível em [48]), definir as rotinas a serem usadas de acordo com os tipos de dados e das matrizes, e definir os parâmetros de entrada das rotinas do LAPACK.

Outra maneira de tratar-se problemas numéricos é utilizando o software científico Scilab [15]. Este é um ambiente de programação bastante amigável desenvolvido pelo instituto INRIA, localizado na França. Ele vem apresentando sucesso entre a comunidade científica devido aos seguintes fatores:

1. A sintaxe usada na construção de programas é bastante simples,

2. o Scilab possui centenas de funções matemáticas pré-definidas que facilitam o desenvolvimento do código pelo programador,

```
#include<iostream.h>
#include<math.h>
extern "C"{
    void dgemv_(char *TRANS,int *M,int *N,double *ALPHA,
                double *A,int *LDA,double *X,int *INCX,
                double *BETA,double *Y,int *INCY);}
int main ( )
{
    char TRANS = 'N';
    int N = 10, LDA = N, INCX = 1, INCY = 1, lin = 0;
    double *A, *X, *Y, BETA = 0.5, ALPHA = 0.2;
    A = new double [N*N];
    for(int j=1; j < = N; j++)
        for(int i = 1; i <= N; i++)
            {  A[lin] = i+j ;
                lin++;}
    X = new double [N];
    Y = new double[N];
    lin = 0;
    for (int i=1;i <= N;i++)
        {  X[lin] = i/10;
           Y[lin] = 2*i/10;
           lin++; }
    dgemv_(&TRANS,&N,&N,&ALPHA,A,&LDA,X,&INCX,&BETA,Y,&INCY);
    for (int i=0; i<N;i++)
        cout<<"Y ["<<i<<" ] = "<<Y[i]<<endl;
    return 0;
}
```

Figura 3.5: Programa em linguagem C++ que utiliza a biblioteca BLAS.

3. oferece uma interface gráfica e instruções que permitem a manipulação de gráficos em duas ou três dimensões,

4. possui uma linguagem de alto nível que facilita o tratamento de matrizes, sendo tal sintaxe análoga à usada pelo Fortran90,

5. permite a manipulação de estruturas de dados, polinômios, números racionais, números complexos, matrizes esparsas, operações básicas de manipulação algébrica, rotinas para tratamento de sinais e imagens, etc, sendo que todas elas possuem uma interface de uso simplificada,

6. o Scilab pode ser intefaceado com outras linguagens, como C e Fortran, e pode gerar códigos em Fortran,

7. é um software de domínio público.

Na figura 3.6 temos um exemplo de um programa do Scilab, que realiza a mesma operação que o programa da figura 3.5, $y = \alpha Ax + \beta y$.

```
1  M = 10;
2  BETA = 0.5;
3  ALPHA = 0.2;
4  for i=1:M
5      for j=1:M
6          A(i,j) = i+j;
7      end
8  end
9  x(1:M) = (1:M)/10; y(1:M) = 2*(1:M)/10;
10 y = ALPHA*A*x + BETA*y;
```

Figura 3.6: Exemplo de um programa do Scilab.

Apesar de todas essa vantagens, o Scilab possui alguns aspectos negativos que prejudicam seu uso. Um deles, talvez o principal, é quanto ao seu desempenho. Como ele utiliza técnicas sofisticadas de interpretação ele não pode oferecer um bom desempenho em relação a alguns compiladores clássicos, podendo ser essa perda de até dez vezes no tempo de execução. No entanto, em certos problemas essa limitação pode ser contraposta pela facilidade de desenvolvimento do código.

Muitos estudos de técnicas de otimização do Scilab estão sendo implementadas. Outra alternativa, é a versão paralela do Scilab que está sendo desenvolvida pelo *Ouragan Project*, o Scilab$_{//}$ [49].

CAPÍTULO 4
ScaLAPACK

O ScaLAPACK [3] é uma biblioteca de álgebra linear desenvolvida para computadores de memória distribuída. Seu desenvolvimento foi baseado no LAPACK, que conforme discutido no capítulo 2, é uma biblioteca de álgebra linear usada em computadores seqüenciais, vetoriais ou de memória compartilhada.

O ScaLAPACK usa passagem de mensagens na comunicação entre os processos e as implementações existentes permitem o uso da biblioteca PVM [41] ou das bibliotecas baseadas no padrão MPI [42], tais como MPICH [43] e LAM [44]. Essas implementações foram construídas em linguagem FORTRAN77 e espera-se o desenvolvimento em C++, tal como anunciado em [24]. No entanto, esse fato não restringe o ScaLAPACK a ser usado apenas em programas na linguagem FORTRAN77.

Neste capítulo vamos discutir detalhadamente muitos aspectos dos ScaLAPACK, apresentando nas próximas seções sua organização e as suas bibliotecas auxiliares. Discutiremos conceitos como a distribuição de dados e os aspectos de organização e convenções sobre suas rotinas.

4.1 Estrutura e funcionalidade

O ScaLAPACK possui rotinas que podem ser usadas na construção de programas para a resolução de equações lineares, problemas de mínimos quadrados, de autovalores e de valores singulares. Para efetuar essas operações, ele usa algumas bibliotecas auxiliares que permitem a troca de mensagem entre os processos bem como a execução de operações locais. Na figura 4.1 são mostradas as diversas bibliotecas usadas pelo ScaLAPACK no seu funcionamento.

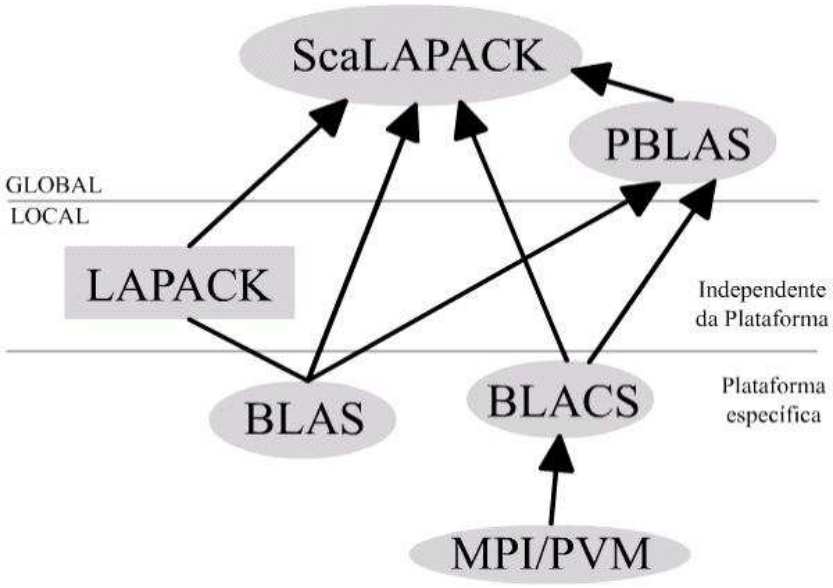

Figura 4.1: Organização hierarquica entre as bibliotecas que compõem o pacote ScaLAPACK usadas nas computações locais e na comunicação.

4.1.1 PBLAS

No desenvolvimento do LAPACK, foi utilizada a biblioteca BLAS, conforme discutimos no capítulo 2. Como a intenção dos implementadores era tornar o ScaLAPACK parecido com o LAPACK, foi criada uma versão paralela do BLAS,

chamada PBLAS (*Parallel BLAS*), que é constituída de rotinas de álgebra linear básicas, tais como as oferecidas pelo BLAS, com a diferença que suas operações são realizadas em paralelo com auxílio da biblioteca BLACS, que faz a passagem de mensagem, e do BLAS, usado nas computações locais.

As rotinas do PBLAS podem ser utilizadas independente das rotinas do ScaLAPACK em computações que exijam operações básicas de álgebra linear. Elas são divididas em três níveis, de acordo com as operações:

- **Nível 1:** Operações vetor-vetor: $y = \alpha x + y$
- **Nível 2:** Operações matriz-vetor: $y = \alpha Ax + \beta y$
- **Nível 3:** Operações matriz-matriz: $C = \alpha AB + \beta C$

Esse níveis são os mesmos que os níveis do BLAS, conforme discutidos na seção 3.5. A lista de rotinas do PBLAS pode ser vista em [50].

4.1.2 BLACS

O BLACS (*Basic Linear Algebra Communication Subprogram*) [51, 52] é uma biblioteca de comunicação especialmente desenvolvida para álgebra linear. Em seu modelo computacional, o BLACS considera que os processos estão distribuídos de acordo com uma **grade** de duas dimensões (como uma matriz), permitindo que os processos troquem mensagens entre si de forma síncrona. Além dessas comunicações ponto a ponto entre os processos, o BLACS permite operações coletivas, como *broadcast* e redução global. Suas rotinas utilizam bibliotecas de passagem de mensagem tais como o PVM ou o MPICH nas suas implementações.

Em um mesmo programa, é permitido criar-se diversas grades de processos, onde estes se comunicam via um contexto associado a cada grade. A formação da grade é feita pela chamada das rotinas do BLACS que discutiremos na seção 4.2.

4.2 Distribuição dos dados

O ScaLAPACK tem um modo particular de fazer a distribuição de matrizes e vetores entre os processadores que é especificada pelo programador na chamada das rotinas.

Essas rotinas de distribuição são específicas para três tipos básicos de matrizes:

1. ***In-core Dense matrices***: Matrizes densas que podem ser alocadas totalmente na memória das máquinas,

2. ***In-core Narrow matrices***: Matrizes de banda e tridiagonais que podem se alocadas totalmente na memória das máquinas,

3. ***Out-of-core matrices***: Matrizes que não podem ser alocadas totalmente na memória, necessitando que parte seja armazenada em disco.

O armazenamento local das matrizes é feito com o uso de *arrays* convencionais. O modo como as matrizes estão armazenadas bem como o tipo de armazenamento para diferentes matrizes é passado às rotinas via um *array* de inteiros chamado *array descriptor*, que será discutido na seção 4.2.3.

4.2.1 Conceitos básicos

Uma necessidade inicial do ScaLAPACK é que os dados globais (matrizes e vetores) sejam distribuídos entre os processadores. Assim, os dados globais são mapeados na memória local assumindo uma distribuição específica, sendo que o dado local é referenciado como um *array* local. Este método maximiza a razão entre operações de ponto flutuante e referências à memória capacitando o reuso de dados tanto quanto possível enquanto ele é armazenado nos níveis mais altos de hierarquia de memória (caches de alta velocidade, registradores vetoriais, etc).

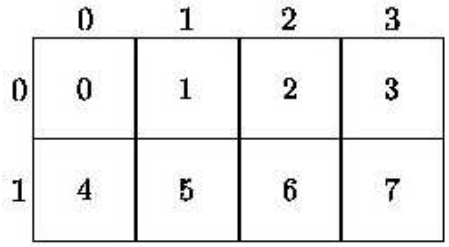

Figura 4.2: Exemplo de uma grade montada com 8 processadores.

Essa especificação da distribuição é muito importante no uso do ScaLAPACK, pois algumas rotinas necessitam de um tipo específico de distribuição, tais como as de resolução de sistemas lineares densos, que exigem uma distribuição bloco-cíclica de uma ou duas dimensões, ou as rotinas de resolução de sistemas lineares de banda e tridiagonais, que assumem que todo o dado global foi distribuído usando uma distribuição bloco-cíclica de uma dimensão.

4.2.2 Grade de processadores

O ScaLAPACK usa um tipo especial de distribuição dos processadores, assumindo que P processadores, que serão usados na execução do programa, estão distribuídos em uma grade $P_r \times P_c$, sendo que P_r é o número de linhas na grade e P_c é o número de colunas. Por exemplo, se temos oito nós em um *cluster*, podemos montar uma grade com $P_r = 2$ e $P_c = 4$, conforme a figura 4.2.

A inicialização da grade de processadores utiliza as rotinas do BLACS. Por exemplo, a função `BLACS_GRIDINIT` assume uma ordenação dos processos como na figura 4.2. Outra maneira de fazer esse mapeamento é usando a função `BLACS_GRIDMAP`, que é uma forma mais geral, que permite ao programador definir como os processos serão mapeados na grade [51].

Na comunicação entre os processos o ScaLAPACK usa o conceito de contextos (*contexts*), onde cada grade de processadores está incluída em um ou mais

contextos. Na verdade, estes contextos têm a mesma função que os comunicadores em MPI, que associam um grupo de processos e estes só podem se comunicar usando o contexto definido.

Em uma aplicação, pode-se ter mais de um contexto, sendo que um processo pode fazer parte de diversos desses. Assim todos os processos da grade podem se comunicar por meio do contexto que foi associado a ela ou por outro que foi associado a um grupo de processos na grade, como no caso de termos um contexto para comunicação entre os processos coluna na grade.

A criação dos contextos pode ser feita usando as rotinas do BLACS, tais como `BLACS_GRIDINIT` e `BLACS_GRIDMAP`, sendo que ao término do uso, devemos liberar os recursos utilizados usando as rotinas `BLACS_GRIDEXIT` ou `BLACS_EXIT`.

4.2.3 *Array descriptors*

O ScaLAPACK necessita de um mapeamento entre o *array* global e a forma como ele é mapeado na memória local dos processadores. Para isso, ele usa *descritores* (*array descriptors*), que são vetores de inteiros que armazenam informações sobre o mapeamento da matriz global na grade e sobre o tipo de dados da matriz, bem como outras informações.

Esses descritores são fornecidos para matrizes densas, matrizes de banda tridiagonais e matrizes *out-of-core*.

Na documentação das rotinas, as entradas no descritor são denotadas com o uso de _ após o nome global da variável. Por exemplo, se `M` é o número de linhas de uma matriz global `A`, então o parâmetro que define esse valor no descritor será denotado por `M_A`.

O tamanho deste descritor é denotado pelo parâmetro `DLEN_`, cujo valor é dado de acordo com o tipo da matriz:

- Matrizes Densas: `DLEN_` = 9,

- Matrizes de banda estreita e tridiagonais: `DLEN_` = 7,
- Matrizes *out-of-core*: `DLEN_` = 11.

O tipo da matriz a que o descritor é associado é definido pela sua entrada `DTYPE_`, que pode ter os seguintes valores:

- `DTYPE_` = 1 : matrizes densas,
- `DTYPE_` = 501: matrizes de banda estreita e tridiagonal,
- `DTYPE_` = 502: matrizes de banda estreita e tridiagonal de lado direito (*right-hand-side*),
- `DTYPE_` = 601: matrizes *out-of-core*.

A inicialização dos descritores é feita usando a rotina `DESCINIT`, que fica localizada no diretório `TOOLS` do código fonte do ScaLAPACK.

Com estes conceitos introduzidos, vamos a seguir discutir as matrizes densas *in-core* e as matrizes de banda estreita e tridiagonais *in-core* . As matrizes do tipo *out-of-core* não serão discutida neste trabalho, pois estivemos interessados apenas em fazer uma implementação inicial da biblioteca e o nosso objetivo principal foi mostrar as facilidades que a orientação ao objeto oferece. Detalhes sobre as *out-of-core* podem ser vistos em [3].

4.2.4 Matrizes densas *in-core*

Na distribuição das matrizes na grade de processos, o ScaLAPACK assume uma forma de distribuição cíclica em uma ou duas dimensões, sendo possível a obtenção de escalabilidade e balanceamento de carga [3].

Nas próximas considerações, vamos numerar os processos de 0 a $(P-1)$ e as linhas ou colunas da matriz de 1 a N.

Inicialmente, vamos considerar uma distribuição bastante simples, que é a distribuição por colunas em uma dimensão, conforme a figura 4.3(a). A coluna K é armazenada no processo $\lfloor K/N_B \rfloor$ onde $N_B = \lceil N/P \rceil$ é o número máximo de colunas por processo. Em alguns casos este método não oferece um bom balancemento de carga, pois quando cada processo termina sua computação sobre o bloco-coluna que recebeu, ele fica ocioso. No caso de fazer essa distribuição por linha teremos o mesmo problema.

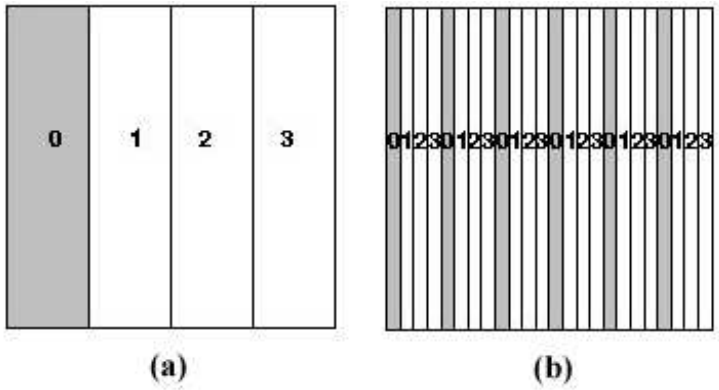

Figura 4.3: (a) Distribuição usando blocos coluna da matriz e (b) distribuição cíclica de cada coluna da matriz

Um segundo método seria fazer uma distribuição por colunas da matriz de forma cíclica, ou seja, cada coluna é associada a um processo. Podemos ver essa distribuição na figura 4.3(b). Assim, a coluna K vai estar no processador $(K - 1) \bmod P$. Neste caso, como são armazenadas simples colunas ao invés de blocos, não se pode usar os níveis 2 e 3 do BLAS, havendo perda de desempenho.

Uma maneira intermediária entre os dois métodos anteriores é usar uma distribuição bloco cíclica com blocos de tamanho N_B, conforme a figura 4.4(a). Neste caso, a coluna K é armazenada no processo $\lfloor (K-1)/N_B \rfloor \bmod (P)$. No caso de $N_B > 1$, temos um balanceamento de carga pior do que no caso anterior, mas podemos usar os níveis 2 e 3 do BLAS para computações locais.

Uma outra forma de distribuição é usando blocos bidimensionais da matriz global, de forma cíclica. Cada bloco tem tamanho $M_B \times N_B$, sendo estes valores determinados pelo usuário. Na figura 4.4(b) é mostrado um exemplo com $N = 16$, $P = 4$, e $M_B = N_B = 2$.

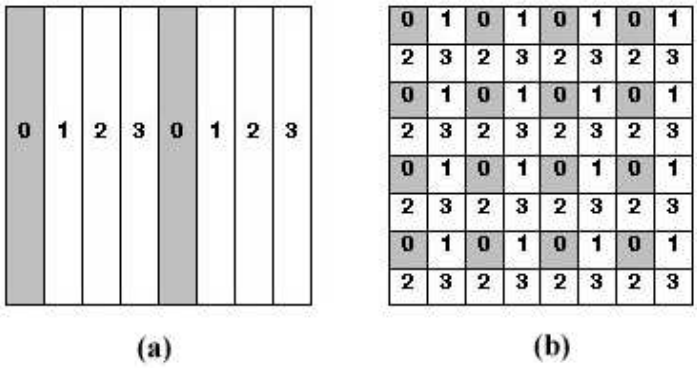

Figura 4.4: (a) Distribuição cíclica usando blocos coluna com N=16, P=4 e N_B=2. Cada bloco coluna da figura contém duas colunas da matriz global. (b) Distribuição cíclica usando blocos bidimensionais. Cada bloco da figura contém duas linhas e duas colunas consecutivas da matriz global.

Esta distribuição permite o uso dos níveis 2 e 3 do BLAS, oferece bom balanceamento de carga e assegura propriedades de escalabilidade. No caso das matrizes densas, vamos assumir que o mapeamento usado no ScaLAPACK será o de blocos bidimensionais distribuídos ciclicamente.

Armazenamento local e mapeamento bloco cíclico

O ScaLAPACK tem algumas regras que associam os índices da matriz global aos índices locais localizados nos processos. Assim, se temos, por exemplo, uma *array* de comprimento N armazenados em P processos, este é dividido em blocos de tamanho N_B, sendo que caso N não seja divisível por N_B, o último bloco

do *array* conterá $N \bmod N_B$ elementos. Desta forma, se o processo 0 recebe o primeiro bloco, o J-ésimo bloco está localizado no processo $J \bmod P$.

Se temos um *array* global indexado por I, seu mapeamento pode ser definido pela equação 4.1.

$$I = (lP + p)N_B + x, \tag{4.1}$$

sendo que I é o índice global no *array*, l é a coordenada do bloco local no qual estas entradas residem, p é a coordenada do processo proprietário daquele bloco e x é a coordenada dentro do bloco onde a entrada do *array* global I foi baseada. Na figura 4.5 temos um exemplo onde estes parâmetros são mostrados.

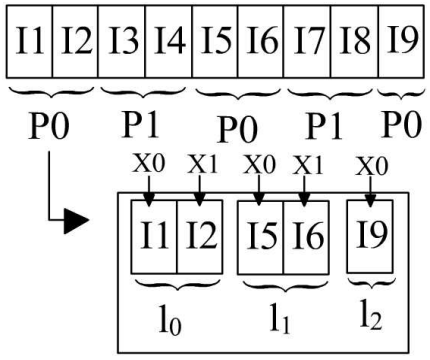

Figura 4.5: Exemplo de um mapeamento de um vetor em uma grade 1D.

Assim, temos

$$p = \left\lfloor \frac{I-1}{N_B} \right\rfloor \bmod P, \tag{4.2}$$

$$l = \left\lfloor \frac{I-1}{PN_B} \right\rfloor \tag{4.3}$$

$$x = [(I-1) \bmod N_B] + 1. \tag{4.4}$$

Com estas equações podemos determinar informações locais, sendo possível o acesso a posições locais da matriz distribuída através dos índices globais. Desta forma, podemos determinar o índice local com $lN_B + x$ e as coordenadas do

processo P correspondente à entrada global indexada por I, bem como fazermos a operação inversa.

Por exemplo, se $N = 16$, $P = 2$ e $N_B = 8$, temos o esquema da figura 4.6. Já na figura 4.7 temos o mesmo número de processos e um vetor do mesmo tamanho, mas com $N_B = 1$.

I	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
p	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
l	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
$l * NB + x$	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8

Figura 4.6: Mapeamento unidimensional para N=16, P=2 e N_B=8. Fonte [3].

I	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
p	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
l	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
x	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$l * NB + x$	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8

Figura 4.7: Mapeamento unidimensional para N=16, P=2 e N_B=1. Fonte [3].

Há uma maneira mais geral de fazermos esta distribuição, onde esta não precisa iniciar no processo 0, mas pode começar a ser feita num processo arbitrário, que pode ser denotado por SRC. As equações para determinarmos os parâmetros são:

$$p = (SRC + \left\lfloor \tfrac{I-1}{N_B} \right\rfloor) \bmod P, \tag{4.5}$$

$$l = \left\lfloor \tfrac{I-1}{PN_B} \right\rfloor, \tag{4.6}$$

$$x = [(I-1) \bmod N_B] + 1. \tag{4.7}$$

Na figura 4.8, temos um exemplo que usa $P = 2$, $N = 16$, $SRC = 1$ e $N_B = 3$.

I	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
p	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0
l	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2
x	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1
$l*NB+x$	1	2	3	1	2	3	4	5	6	4	5	6	7	8	9	7

Figura 4.8: Mapeamento unidimensional com N=16, P=2, N_B=3 e SRC=1. Fonte [3].

Para o caso bidimensional, as equações que definem o mapeamento são um pouco mais complexas, sendo necessários dois índices para determinar os elementos das matrizes.

Deste modo, uma matriz de dimensão $M \times N$, particionada em blocos de tamanho $M_B \times N_B$, tem o primeiro bloco associado ao processo de coordenadas $(RSRC, CSRC)$ na grade, onde $RSRC$ define a coordenada-linha da grade e $CSRC$ define a coordenada coluna da grade, onde é iniciada a distribuição dos blocos.

As associações entre as coordenadas globais e locais da matriz é dada pelo seguinte conjunto de equações:

$$(l, m) = \left(\left\lfloor \frac{I-1}{P_r M_B} \right\rfloor, \left\lfloor \frac{J-1}{P_c N_B} \right\rfloor\right) \tag{4.8}$$

$$(p_r, p_c) = \left((RSRC + \left\lfloor \frac{I-1}{M_B} \right\rfloor) \bmod P_r, (CSRC + \left\lfloor \frac{(J-1)}{(N_B)} \right\rfloor) \bmod P_c\right) \tag{4.9}$$

$$(x, y) = ([(I-1) \bmod M_B] + 1, [(J-1) \bmod N_B] + 1). \tag{4.10}$$

Nestas equações, (l, m) definem as coordenadas dos blocos locais, (p_r, p_c) são as coordenadas do processo na grade de processos, e (x, y) são as coordenadas dentro do bloco onde a entrada da matriz global (I, J) foi baseada.

Desta forma, a matriz global é quebrada em blocos de tamanho $M_B \times N_B$ e é distribuída na grade de processos de forma cíclica. Na figura 4.9, temos um exemplo para $M = N = 5$, $M_B = N_B = 2$ e uma grade $P_r = P_c = 2$. Vemos que a distribuição iniciou no processo zero e que os blocos da matriz não tem

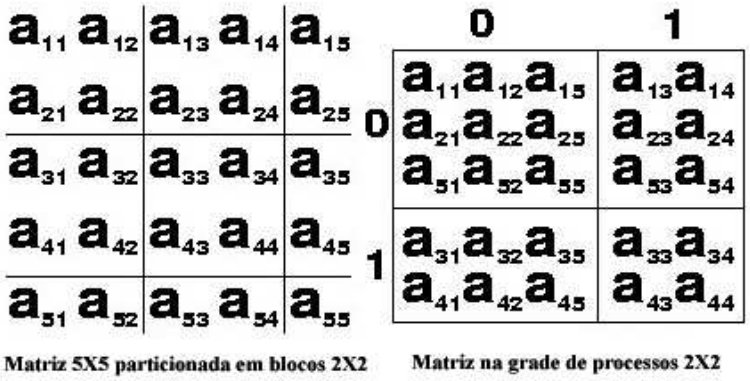

Figura 4.9: Mapeamento de uma matriz 5×5 numa grade 2×2.

tamanhos iguais, sendo que alguns tem tamanho menor.

Para determinarmos o número de linhas e colunas que cada processo recebe, recorremos ao cálculo de LOC_r e LOC_C, que revelam estes valores locais.

$$LOC_r \simeq \frac{\frac{M+M_B-1}{M_B} + P_r - 1}{P_r} M_B \tag{4.11}$$

$$LOC_c \simeq \frac{\frac{N+N_B-1}{N_B} + P_c - 1}{P_c} N_B. \tag{4.12}$$

Estas quantidades pode ser automaticamente calculadas pela rotina `NUMROC` do ScaLAPACK.

Array descriptors para matrizes densas *in-core*

O descritor é usado para as rotinas de resolução de sistemas lineares e problemas de autovalores. Na tabela 4.1 temos os parâmetros do descritor.

Os valores de entrada no descritor são definidos pelo usuário e são armazenados usando-se a rotina `DESCINIT` do ScaLAPACK.

Para um melhor entendimento dessa distribuição, vamos mostrar um exemplo de como é feita a distribuição de uma matriz entre os processos em uma grade.

Tabela 4.1: Variáveis do descritor de*array* para matrizes densas in-core.

Array descritor para matrizes densas in-core			
DESC_	Nome simóblico	Escopo	Definição
1	DTYPE_A	global	Descritor de tipo: DTYPE_A = 1, para matrizes densas.
2	CTXT_A	global	Contexto de BLACS que indica a grade de processos sobre o qual a matriz A é distribuída.
3	M_A	global	Número de linhas no *array* global A.
4	N_A	global	Número de colunas no *array* global A.
5	M_B_A	global	Número de linhas no bloco a ser distribuído.
6	N_B_A	global	Número de colunas no bloco a ser distribuído.
7	RSRC_A	global	Processo linha inicial na grade onde é iniciada a distribuição.
8	CSRC_A	global	Processo coluna inicial na grade onde é iniciada a distribuição.
9	LLD_A	local	Dimensão principal do *array* local: $LLD\_A \geq MAX(1, LOC_r(M\_A))$.

Exemplo: distribuição de uma matriz densa *in-core*

Vamos considerar que temos 6 processadores que compõem uma grade 2×3, previamente criada com o uso das rotinas do BLACS, conforme discutimos em 4.2.2. A matriz utilizada é quadrada de tamanho 9×9 e os blocos usados na distribuição têm dimensão 2×2, conforme mostrado na figura 4.10.

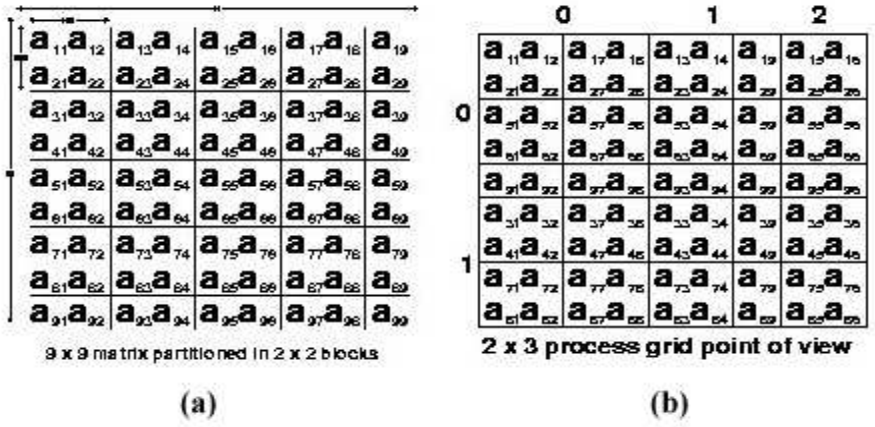

Figura 4.10: (a) Matriz 9×9 particionada em blocos 2×2 e (b) matriz distruída na grade de processos 2×3

O número de linhas e colunas que cada processo possui é dado pelos valores de LOC_r e LOC_c, calculados pela equações 4.11 e 4.12. Os valores calculados estão na tabela 4.2.

O valor de $LLD\_$ (*leading dimension of the local array*) é utilizado para o processo saber o número de linhas que ele deve somar à posição atual para ir para a próxima coluna da matriz, pois ela é armazenada localmente na forma de um vetor unidimensional. No caso de utilizarmos a linguagem C ou C++, é o número de colunas, uma vez que o armazenamento nestes casos é feito por linhas e não por colunas, como na linguagem FORTRAN. Logo, no caso da tabela 4.2, $LLD\_$ é dado pelo número de linhas em cada bloco local, uma vez que estamos

considerando o caso da linguagem FORTRAN.

Tabela 4.2: Valores de $LLD\_$, LOC_r e LOC_c em cada processo.

Coordenada do processo	$LLD\_$	$LOC_r(M\_)$	$LOC_c(N\_)$
(0,0)	5	5	4
(0,1)	5	5	3
(0,2)	5	5	2
(1,0)	4	4	4
(1,1)	4	4	3
(1,2)	4	4	2

Algumas convenções sobre armazenamentos

O ScaLAPACK usa algumas convenções de forma a facilitar o armazenamento dos dados e diminuir o acesso a eles tanto quanto possível, aproveitando propriedades matemáticas de matrizes e vetores. Deste modo, para matrizes densas não-simétricas, simétricas ou hermitianas, temos diferentes formas de acesso.

Para matrizes simétricas ou hermitianas, apenas a parte triangular superior ou triangular inferior da matriz é acessada. Na chamada das rotinas do ScaLAPACK, o parâmetro `UPLO` deve ter seu valor definido em `UPLO='U'` para acessar apenas a parte superior (*upper*) ou `UPLO='L'` para usar apenas a parte inferior (*lower*) da matriz. Caso a matriz seja triangular unitária, ou seja, possua os elementos da diagonal iguais a um, definimos o parâmetro `DIAG='U'`, e esses elementos também não serão acessados.

Caso tenhamos uma matriz hermitiana, que tem a propriedade de ser igual à sua transposta conjugada ($A = A^{\dagger}$), os elementos da diagonal são todos reais, sendo que sua parte imaginária é assumida ser zero e não é necessário qualquer

acesso a ela.

4.2.5 Matrizes de banda estreita e tridiagonais *in-core*

As matrizes de banda estreita (*narrow band*) e tridiagonais usam o tipo de distribuição bloco-coluna ou bloco-linha, sendo que, especificamente, a *narrow band* e a tridiagonal usam a bloco-coluna, e a matriz densa de lado direito (*right-hand-side*), bloco-linha.

No caso das matrizes de banda estreita, algoritimos de divisão e conquista são utilizados, pois oferecem maior escopo na exploração do paralelismo. Além disso, elas são particionadas em blocos (como as triangulares), e o paralelismo fica limitado pelo número de blocos, pois estes são processados de forma independente. Logo, é necessária uma escolha do número de blocos ao menos igual ao paralelismo desejado [3].

Distribuição bloco-coluna e bloco-linha

Conforme já discutimos, o ScaLAPACK assume uma distribuição bloco-coluna para as rotinas que operam sobre matrizes densas e triangulares, onde são usados algoritimos de divisao e conquista.

Na figura 4.11 são mostradas as distribuições por bloco-linha e bloco-coluna. Assumimos que os processos são numerados de 0 a $P-1$ e a matriz, com linhas de 1 a M e colunas de 1 a N.

Na figura 4.11(a), temos as distribuição bloco-coluna em uma dimensão. Cada processo recebe ao menos um bloco coluna da matriz. Se N é divisível por N_B, cada bloco possui N_B colunas da matriz global. Caso contrário, os $\lfloor N/N_B \rfloor$ primeiros blocos contêm N_B colunas da matriz global e o último possui $N \bmod N_B$.

No caso da distruibuição bloco-linha, a situação é a transposta da que acabamos de descrever. Na figura 4.11(b) é mostrado um exemplo desse tipo de

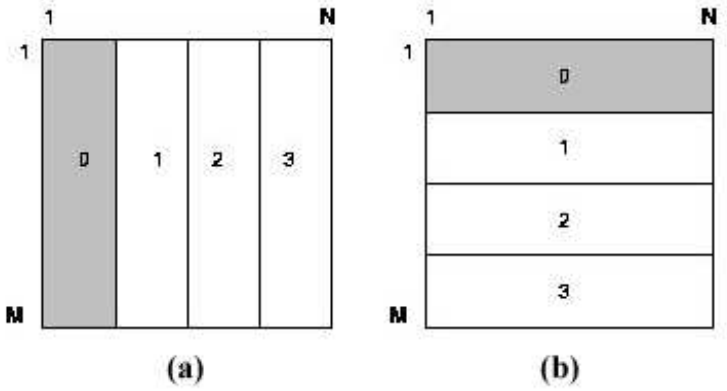

Figura 4.11: (a) Distribuição de uma matriz em uma grade do tipo bloco-coluna e (b) distribuição em uma grade do tipo bloco-linha.

distribuição.

Mapeamento do bloco

Se tivermos uma matriz A, $M \times N$, para ser distribuída em uma grade $1 \times P$, o processo 0 recebe o primeiro bloco da matriz e o p-ésimo processo tem coordenada $(0, p)$ na grade. O mapeamento de uma coluna J da matriz global é feito por:

$$J = pN_B + x \tag{4.13}$$

onde p é o processo que contém essa coluna, N_B é o tamanho do bloco e x é a coordenada da coluna dentro do bloco. Assim, temos:

$$p = \left\lfloor \frac{J-1}{N_B} \right\rfloor, \tag{4.14}$$

$$x = [(J-1) \bmod N_B] + 1. \tag{4.15}$$

Na tabela 4.3 temos um exemplo para $P = 2$, $N = 16$ e $N_B = 8$.

No entanto, não é necessário que a distribuição inicie no processo 0. Podemos definir o início da distribuição por SRC, que é o processo inicial na grade de uma dimensão. Assim, ficamos com as seguinte equações que definem a distribuição:

Tabela 4.3: Mapeamento bloco-coluna: $P = 2$ e $N_B = 8$.

J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
p	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
x	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8

$$p = SRC + \left\lfloor \frac{J-1}{N_B} \right\rfloor, \tag{4.16}$$

$$x = [(J-1) \bmod N_B] + 1. \tag{4.17}$$

Na tabela 4.4 temos um exemplo para $N = 16$, $P = 2$. $N_B = 8$ e $SRC = 1$.

Tabela 4.4: Mapeamento bloco-coluna: $P = 2$, $SRC = 1$ e $N_B = 8$.

J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
p	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
x	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8

Podemos ainda determinar o número de linhas ou colunas que cada processo recebe usando as equações 4.11 e 4.12. Uma maneira prática de definirmos o número de colunas que cada processo recebe é usando os algoritmos mostrados na figura 4.12. O algoritmo para a distribuição bloco-linha é análogo.

Armazenamento local para matrizes de banda estreita

Para discutir os conceitos sobre o armazenamento local das matrizes de banda estreita, vamos considerar uma matriz A, 7×7, dada a seguir. Esta matriz tem largura de banda $BW = 2$, com banda superior $BWU = 2$ e inferior $BWL = 2$.

Se assumirmos que a matriz A é não simétrica, o usuário pode escolher realizar um pivotamento (permutação entre as linhas da matriz) ou não durante a

```
1  NB = N/P;
2  K = int(N/NB);
3  se (mod(N,NB) == 0) entao
4        processos (0,0) ... (0,K) recebem
5            locc = NB;
6  caso contrário
7        processos (0,0) ... (0,K-1) recebem
8            locc = NB
9        processo (0,K) recebe
10            locc = N - K*NB
11 end.
```

Figura 4.12: Algoritmo de distribuição bloco-coluna.

fatorização LU. Para isso, ele pode usar as rotinas `PxGBTRF` ou `PxDBTRF`, respectivamente, que realizam uma fatorização LU. Em qualquer um desses casos, uma distribuição bloco-coluna é assumida, sendo que, se o pivotamento for selecionado, é necessário um espaço adicional para preenchimento, como veremos a seguir.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} & a_{67} \\ 0 & 0 & 0 & 02 & a_{75} & a_{76} & a_{77} \end{pmatrix}$$

No caso de escolhermos que nenhum pivotamento é selecionado, vamos exemplificar distribuindo a matriz A em uma grade 1×3 com blocos $N_A = 3$. Na tabela

4.5 temos este esquema. Nesta, o valor $*$ significa que nenhum valor definido é necessário neste local. Neste caso, o LLD (*leading dimension of the local array*) deve ser ao menos igual a $BWL + BWU + 1$.

Tabela 4.5: Mapeamento da matriz A (não simétrica) sem pivotamento.

Processos						
0			**1**			**2**
$*$	$*$	a_{13}	a_{24}	a_{35}	a_{46}	a_{57}
	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	a_{67}
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{77}
a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	a_{76}	$*$
a_{31}	a_{42}	a_{53}	a_{64}	a_{75}	$*$	$*$

No entanto, se selecionarmos um pivotamento parcial e distribuirmos a mesma matriz em uma grade 1×3 com $N_B = 3$, temos a situação da tabela 4.6. A quantidade necessária para o armazenamento adicional, representada por F na tabela 4.6, é igual à soma das quantidades BWL e BWU. O LLD neste caso deve ser ao menos igual a $2(BWL + BWU) + 1$.

Um outro caso importante é quando a matriz A é simétrica positiva com largura de banda definida, neste caso, $BW = 2$. Vamos assumir que esta matriz é distribuída sobre uma grade 1×3 com $N_B = 3$. Como a matriz é simétrica e triangular, vamos usar apenas a parte triangular inferior,que é definida pela variável `UPLO='L'` na chamada das rotinas. A distribuição é mostrada na tabela 4.7.

Para realizar a fatorização, podemos chamar a rotina `PxPBTRF` com $BW = 2$, por exemplo. Outra forma de fazermos esta distribuição é usar apenas a parte triangular superior da matriz, ou seja, `UPLO='U'`. Neste caso temos uma distribuição como na tabela 4.8. Em ambos os casos, o LLD deve ser ao menos igual a $BW + 1$.

Tabela 4.6: Mapeamento da matriz A (não simétrica) com pivotamento.

Processos						
0			**1**			**2**
F	F	F	F	F	F	F
F	F	F	F	F	F	F
F	F	F	F	F	F	F
F	F	F	F	F	F	F
	*	a_{13}	a_{24}	a_{35}	a_{46}	a_{57}
	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	a_{67}
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{77}
a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	a_{76}	*
a_{31}	a_{42}	a_{53}	a_{64}	a_{75}	*	*

Modos de armazenamento de matrizes tridiagonais

Para descrevermos esses modos, vamos usar uma matriz 7×7 que pode ser representada pelos vetores (DL, D, DU), que representam as diagonais inferior, principal e superior, respectivamente.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 & 0 & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} & 0 & 0 \\ 0 & 0 & 0 & a_{54} & a_{55} & a_{56} & 0 \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} & a_{67} \\ 0 & 0 & 0 & 0 & 0 & a_{76} & a_{77} \end{pmatrix}$$

Vamos assumir que essa matriz é simétrica e que nenhum pivotamento é exigido e dividir a matriz em uma grade 1×3 usando blocos 3×3. Assim temos o

Tabela 4.7: Mapeamento da matriz A simétrica positiva. `UPLO='L'`.

Processos						
0			**1**			**2**
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{77}
a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	a_{76}	*
a_{31}	a_{42}	a_{53}	a_{64}	a_{75}	*	*

Tabela 4.8: Mapeamento da matriz A simétrica positiva. `UPLO='U'`.

Processos						
0			**1**			**2**
*	*	a_{31}	a_{42}	a_{53}	a_{64}	a_{75}
	a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	a_{76}
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{77}

esquema da tabela 4.9, onde podemos notar que os elementos nulos da matriz não são armazenados.

Outra maneira de efetuarmos esse armazenamento é armazenando apenas a parte superior ou inferior da matriz, definindo o valor da variável `UPLO` em U ou L respectivamente. Nas tabelas 4.10 e 4.11 são apresentadas essas distribuições.

Array descriptor para matrizes de banda estreita e tridiagonais

O descritor para as matrizes de banda estreita e tridiagonais tem comprimento igual a sete e o parâmetro `DTYPE_` deve ser igual a 501. Na tabela 4.12 temos os valores que devem ser definidas no descritor.

Quando a matriz A não for simétrica e a operação que for realizada não exigir pivotamento, o valor de `LLD_A` deve ser ao menos igual a $BWL+BWU+1$, onde

Tabela 4.9: Mapeamento de uma matriz não-simétrica tridiagonal.

	Processos						
	0			1			2
DL		a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	a_{76}
D	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{77}
DU	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	a_{67}	

Tabela 4.10: Mapeamento de uma matriz não-simétrica tridiagonal usando apenas a parte inferior (`UPLO = 'L'`).

	Processos						
	0			1			2
D	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{77}
E	a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	a_{76}	

BWL e BWU são as larguras de banda inferior e superior, respectivamente. No entanto, se for exigido pivotamento pela função que vai operar sobre a matriz, o valor de `LLD_A` deve ser ao menos igual a $2(BWL + BWU) + 1$. No caso da matriz ser simétrica, `LLD_A` = $BW + 1$, e caso elas seja tridiagonal, esse valor não precisa ser definido.

Array descriptor para matriz de lado direto

No caso das matrizes de lado direito (*right-hand-side*), o descritor deve ter tamanho sete e a variável `DTYPE_` = 502. Na tabela 4.13 são apresentadas os valores do descritor.

Maiores detalhes sobre a distribuição das matrizes no ScaLAPACK podem ser vistos em [3].

Tabela 4.11: Mapeamento de uma matriz não-simétrica tridiagonal usando apenas a parte superior (`UPLO = 'U'`).

	Processos						
	0			1			2
D	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{77}
E	a_{12}	a_{23}	a_{34}	a_{54}	a_{56}	a_{67}	

4.3 Lista geral de argumentos

No código fonte das rotinas do ScaLAPACK podem ser vistas características de uso, bem como uma lista de argumentos com as rotinas que devem ser passadas à função no seu manuseio, sendo descritos em ordem de entrada.

Esse argumentos geralmente aparecem na seguinte ordem:

1. Opções de computação da matriz: como no caso de usar apenas a parte superior ou inferior de uma matriz simétrica, definida pela variável `UPLO`.

2. Aspectos dimensionais: definem os valores iniciais na matriz e sua distribuição, como a partir de qual linha e coluna a matriz deve ser operada, ou mesmo o descritor com os argumentos já definidos.

3. Matriz ou vetor de entrada: são passados à função e podem ter seus valores alterados durante a computação.

4. Matrizes, vetores ou escalares que guardam os resultados calculados: como no caso dos vetores que armazenam os autovalores de uma matriz.

5. *Work arrays*: são vetores usados nas computações oferecendo espaço de troca e armazenamento.

Tabela 4.12: Variáveis do *array* descritor para matrizes de banda estreita e tridiagonais.

Array descritor para matrizes de banda estreita e tridiagonais			
DESC_	Nome simóblico	Escopo	Definição
1	DTYPE_A	global	Descritor de tipo: DTYPE_A = 501, para uma grade de processos $1 \times P_c$ para matrizes de banda e tridiagonais usando distribuição bloco-coluna.
2	CTXT_A	global	Contexto de BLACS que indica a grade de processos sobre o qual a matriz A é distribuída.
3	N_A	global	Número de colunas no *array* global A.
4	N_B_A	global	Número de colunas no bloco a ser distribído.
5	CSRC_A	global	Processo coluna inicial na grade onde é iniciada a distribuição.
6	LLD_A	local	*Leading dimension* para o vetor local. Para matrizes tridiagonais, esta entrada é ignorada.
7			Ignorada,reservada

Tabela 4.13: Variáveis do *array* descritor para matrizes de lado direito.

Array descritor para matrizes de lado direito			
DESC_	Nome simóblico	Escopo	Definição
1	DTYPE_B	global	Descritor de tipo: DTYPE_B = 502, para uma grade de processos $P_r \times 1$ para usando distribuição bloco-linha
2	CTXT_B	global	Contexto de BLACS que indica a grade de processos sobre o qual a matriz A é distribuída
3	M_B	global	Número de linhas no *array* global B
4	MB_B	global	Número de linhas no bloco a ser distribído
5	CSRC_B	global	Processo coluna inicial na grade onde é iniciada a distribuição
6	LLD_B	local	*Leading dimension* para o vetor local. $LLD \geq MAX(1, LOC_r(M\_B))$. Para matrizes tridiagonais essa entrada é ignorada.
7			Reservada

6. INFO: variável de controle de erro.

Assim, durante o uso do ScaLAPACK é importante a verificação desses valores de acordo com o problema que se deseja resolver. Uma lista completa desses valores pode ser vista em [3], sendo que no código fonte das rotinas escolhidas para uso também são apresentadas descrições sobre os argumentos de entrada das funções.

4.4 Conteúdo do ScaLAPACK

Conforme discutimos anteriormente, as rotinas do ScaLAPACK podem ser usadas na resolução de sistemas lineares, problemas de autovalores e mínimos quadrados. No apêndice B é mostrada uma lista com as rotinas de ScaLAPACK, classificadas de acordo com os tipos de matrizes sobre as quais elas operam. Essas rotinas são classificadas de acordo com sua funcionalidade:

- ***Driver routines***: cada uma dessas rotinas resolvem um problema completamente, sendo esses: *equações lineares, problemas de mínimos quadrados, problemas de autovalores, problemas de valor singular, problemas de autovalores simétricos e problemas de autovalores generalizados definidos simétricos.*

- ***Computational routines***: essas rotinas realizam uma computação distinta e seu uso principal é na obtenção de soluções em problemas que não podem ser solucionados pelas *driver routines*. As rotinas podem resolver os seguintes problemas: *equações lineares, fatorização ortogonal e problemas de mínimos quadrados (fatorização QR, fatorização LQ, fatorização QR com pivotamento, fatorização ortogonal completa e outras fatorizações), fatorizações ortogonais generalizadas, problemas de autovalores simétri-*

cos, problemas de autovalores não simétricos, decomposição de valor singular e problemas de autovalores generalizados definidos simétricos.

- ***Auxiliary routines***: rotinas auxiliares usadas para realizar subtarefas de algoritmos de bloco-particionamento, computações de mais baixo nível, como cálculo da norma de uma matriz, ou mesmo computações básicas que são extensões do PBLAS.

Essas rotinas são ainda divididas de acordo com o tipo de dados que operam, sendo eles **real**, **real de dupla precisão**, **complexo** e **complexo de dupla precisão**. Esses tipos são usados na nomenclatura das rotinas, conforme veremos a seguir.

4.4.1 Nomenclatura da rotinas

A nomenclatura das rotinas do ScaLAPACK segue uma regra básica que é parecida com a usada pelo LAPACK. As *driver routines* e *computational routines* possuem a seguinte regra:

PXYYZZZ

- **P**: usado para diferenciar as rotinas do ScaLAPACK das rotinas do LAPACK.
- **X**: indica o tipo de dado usado na computação, quer pode ser:
  - **S**: Real (*float*),
  - **D**: Dupla precisão (*double*),
  - **C**: Complexo de precisão simples (*complex*),
  - **Z**: Complex de dupla precisão (*double complex*).

- **YY**: indica o tipo da matriz que será operada pela função. As matrizes podem ser:
  - DB: De banda geral diagonalmente dominante,
  - GT: Tridiagonal geral,
  - DT: Tridiagonal geral sem pivotamento,
  - GB: De banda geral,
  - GE: Geral,
  - GG: Geral para problemas generalizados,
  - HE: Hermitiana,
  - OR: Ortogonal,
  - PB: Simétrica ou hermitiana, positiva definida de banda geral,
  - PO: Simétrica ou hermitiana positiva,
  - PT: Simétrica ou hermitiana positiva definida tridiagonal,
  - ST: Simétrica tridiagonal,
  - SY: Simétrica,
  - TR: Triangular,
  - TZ: Trapezoidal,
  - UN: Unitária,
  - HS: Matriz de Hesenberg.
- **ZZZ**: indica a computação que a rotina realiza. Por exemplo `SV` indica as rotinas que resolvem sistemas de equações lineares. Detalhes sobre essas rotinas podem ser vistos em [3] ou no apêndice.

Algumas das matrizes discutidas acima foram discutidas na seção 3.2 ou em [25, 46]. Vamos considerar alguns exemplos da nomenclatura:

- PDSYEVX : essa rotina opera sobre dados do tipo *double* (D) em matrizes simétricas (SY) e resolve um problema de autovalor (EVX),
- PSGESVX: essa rotina opera sobre dados de simples precisão (S), em matrizes gerais (GE) e resolve sistemas de equações lineares (SVX),
- PZGEBRD: essa rotina opera sobre dados do tipo complexo de dupla precisão (Z), em matriz gerais (GE) e realiza uma redução bidiagonal (BRD).

Conforme observamos, a nomenclatura das rotinas do ScaLAPACK é bastante complexa e esse é uma característica que dificulta seu uso. Nos próximos capítulos de nosso trabalho, vamos mostrar como o uso de orientação ao objeto pode ajudar a solucionar esse problema.

CAPÍTULO

5
Álgebra linear com orientação ao objeto

O nível de abstração entre as linguagens de programação desde os primórdios da computação sofreu profundas modificações. Inicialmente os programas eram escritos em linguagem de máquina usando notação binária, sendo os dados e as instruções colocados conjuntamente num mesmo programa. Os acessos à memória eram feitos diretamente pelo programador especificando a posição de armazenamento como uma instrução binária.

Um grande avanço ocorreu com o surgimento das linguagens montadoras, que introduziram instruções de máquina como `load, move, add`, etc. No entanto, o desenvolvimento de programas nesta época ainda era bastante complexo, dependendo da operação a ser executada.

Durante a década de 50 surgiram linguagens inovadoras como a FORTRAN e LISP, que introduziram recursos que facilitaram imensamente o desenvolvimento de códigos. Uma operação matemática como $z = x^2+y^2$, que em linguagem montadora poderia exigir várias linhas de instrução tornou-se possível ser realizada com a execução de um único comando, como `Z = X**2 + Y**2`, na linguagem FORTRAN.

A partir das décadas de 60 e 70, surgiram linguagens de programação estruturadas, como Pascal e C, que facilitaram o desenvolvimento de códigos permitindo dividir programas em procedimentos e funções que pudessem ser reaproveitados em diversas partes dos programas. Com isso, algumas funções foram agrupadas em módulos, surgindo computação modular, que usa o princípio de divisão e conquista no desenvolvimento do código.

No entanto, mesmo com esses avanços, no desenvolvimento de programas era necessário pensar-se em termos da estrutura do *software* ao invés do problema real, devido ao insuficiente nível de abstração. Para suprir essas necessidades, na década de 60 foram desenvolvidos métodos de programação orientada ao objeto, que começaram com a linguagem Simula e alcançaram difusão na década de 80 com o Smaltalk. Neste capítulo, introduziremos de forma quantitativa alguns princípios de orientação ao objeto aplicando-os a problemas de álgebra linear.

5.1 Aspectos de orientação ao objeto

A metodologia da orientação ao objeto é baseada no princípio de decomposição e classificação dos problemas. Aristóteles já usava esse princípio quando fala em “classes de peixes e classes de pássaros”. Do mesmo modo, a orientação ao objeto usa abstrações que são chamadas *classes*, sendo um membro específico de uma determinada abstração, um *objeto*. Esse objeto pode ser pensado como um objeto do mundo real, sendo que ele possui métodos que podem ser aplicados sobre ele, definindo sua interface. Por exemplo, podemos ter uma classe de tipos de lâmpadas e uma lâmpada incandescente pode ser considerada como um objeto dessa classe. As operações como ligar, desligar e queimar são os métodos que são chamados via envio de mensagens aos objetos.

As classes são compostas por membros públicos, privados ou protegidos [53].

Os membros privados são visíveis apenas ao implementador da classe e seu uso fica restrito à implementação dos métodos, que podem operar sobre esses membros. Deste modo, o implementador só disponibiliza ao usuários os membros públicos, que constituem a interface da classe. Essa proteção é importante, pois proíbe ao usuário acessar os membros privados da classe, ocorrendo encapsulação dos membros havendo uma redução de *bugs* no desenvolvimento de *softwares*.

Além disso, a orientação ao objeto permite ao desenvolvedor da classe alterar aspectos de sua implementação sem alterar sua interface. Essa característica permite a criação de bibliotecas padronizadas com interface definida como a STL [6] e a MTL [5]. Assim, as modificações na implementação, como no caso de optimizações de código na biblioteca, não afetam a interface de uso.

Outro aspecto importante com relação à orientação ao objeto, é que é possível fazer-se composição de classes, onde membros de uma determinada classe podem ser constituídos de uma classe completa. Essa relação de "tem-um" é importante, pois objetos do mundo real obedecem a esse princípio, como no caso de um carro que possui um motor. O motor pode ser constituído de uma classe completa que pode ser introduzida na classe carro.

Objetos do mundo real podem ser classificados e reclassificados de inúmeras formas. Um exemplo é uma classe para representar figuras geométricas. Derivada dessa classe temos os círculos, os retângulos, triângulos, etc. No entanto, a classe retângulo pode ser decomposta em retângulos e quadrados (que é uma forma especial de retângulo), e a classe triângulo pode ser decomposta em triângulos isósceles, eqüiláteros e escalenos. Essa hierarquia de classes compartilha de funções semelhantes como rotação, translação, etc. Desta idéia surgiu o conceito de herança.

Na figura 5.1 é usada a representação UML (*Unified Modeling Language*) [54], que define a forma como os métodos e atributos da classe são declarados

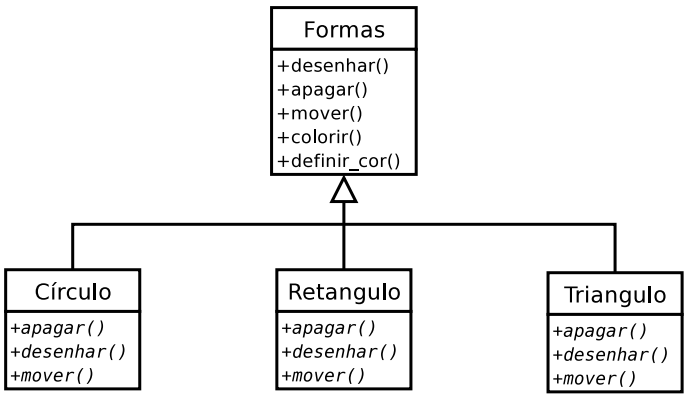

Figura 5.1: Exemplo de uma hierarquia de classes considerando figuras geométricas. Note que a classe base Forma é abstrata e as demai são construídas via herança.

nos diagramas. Nessa representação, o primeiro sub-retângulo contém o nome da classe, o segundo as características, chamadas atributos da classe, e o terceiro contém as operações, que são os métodos da classe. Neste exemplo, a classe `Formas` é uma classe abstrata, pois alguns de seus métodos são virtuais, e as demais são derivadas dela.

O conceito de herança está associado ao reaproveitamento de código e oferece um alto nível de abstração. As classes derivadas da classe base "herdam" sua interface. Assim, conforme notamos na figura 5.1, alguns métodos são comuns a todas a figuras, como o caso de `mover()`, `desenhar()` e `apagar()`, sendo que apenas as implementações desses métodos são diferentes.

Deste modo, se uma classe B herda uma classe A, então todos os métodos de A também são de B. Além disso, cada objeto de B também é objeto de A. Como as classes de uma hierarquia possuem métodos comuns, uma mesma mensagem deve resultar na chamada de métodos diferentes, dependendo do objeto que realizou a chamada, sendo a decisão tomada em tempo de execução. Assim, no caso da figura 5.1, se um objeto da classe círculo chamar o método `apagar()`, ele usará a

implementação associada à classe `Circulo` e não a da classe `Formas`, já que ela é abstrata e não possui implementação para esse método. O mesmo ocorre com as outras formas. No entanto, a chamada ao método `definir_cor()` resultará na chamada ao método implementado na classe `Formas`, já que as classe derivadas não possuem implementação para ele, conforme notamos na figura.

Conforme notamos, o uso de herança é uma maneira de reaproveitarmos código. Apenas os métodos que necessitam ser adaptados às características específicas de uma classe mais especializada ou aqueles específicos à uma dada classe necessitam ser implementados. Os demais podem ser herdados da classe base sem modificações. Além da herança simples, também podemos ter herança múltipla, onde uma dada classe pode herdar métodos de mais de uma classe.

Outro fato importante quanto à essa metodologia é que cada classe conhece apenas os detalhes de sua própria implementação. Das outras classes, apenas suas interfaces são conhecidas.

Na próxima seção vamos aplicar os conceitos de orientação ao objeto no desenvolvimento de uma hierarquia de classes de matrizes. Maiores detalhes sobre orientação ao objeto pode ser vistos em [7, 53, 55, 56, 57]. Em [58] são discutidos as limitações que a orientação ao objeto ainda sofre e quais serão os desenvolvimentos futuros dessa metodologia.

5.2 Orientação ao objeto e álgebra linear

As bibliotecas tradicionais de álgebra linear, discutidas no capítulo 3, apresentam dois pontos fracos fundamentais: interface complexa e dificuldades de implementação devido ao pouco reaproveitamento de código que as linguagens não orientadas ao objeto oferecem. Esses problemas surgem devido à metodologia usada na construção dessas bibliotecas, pois é utilizada a metodologia *top-down*,

ou programação estruturada, onde a decomposição das soluções é feita a nível das subrotinas, sendo o modelo uma composição de chamadas a essas sub-rotinas. Esse nível de abstração não permite ao programador pensar ao nível do problema, mas ao nível das computações, o que dificulta o desenvolvimento de código. No restante desse capítulo vamos mostrar como a orientação ao objeto pode superar esses pontos fracos facilitando o desenvolvimento das bibliotecas.

5.2.1 Representação de matrizes

Usando o conceito de orientação ao objeto, matrizes podem ser pensadas como uma abstração de problemas de álgebra linear e terem uma classe associada a elas. Uma matriz específica seria então um objeto da classe e as operações sobre essa matriz, como inversão e transposição, os métodos dessa classe.

Como os diversos tipo de matrizes possuem propriedades comuns, pode-se construir uma hierarquia de classes usando herança, onde as matrizes específicas, como densas e de banda, herdam essas propriedades comuns de uma matriz geral, possuindo ainda propriedades específicas a cada uma.

Operações como acesso a um elemento ou inclusão de um elemento na matriz são operações comuns a todas as matrizes, o que implica que esses métodos podem pertencer a uma classe de matriz geral. Além disso, alguns membros como o número de linhas e colunas também são comuns a todas as matrizes. Podemos então ter uma classe geral definida como abstrata e as matrizes de um tipo específico que herdam seus métodos. Como exemplo, podemos considerar a figura 5.2, onde temos uma matriz geral e três tipos de matrizes: quadrada, retangular e um vetor, que é um tipo específico de matriz com apenas uma linha ou uma coluna, onde é usada a representação UML.

A interface da classe possui alguns métodos comuns, sendo a implementação de cada um adaptada ao tipo da matriz, de forma que aspectos específicos das

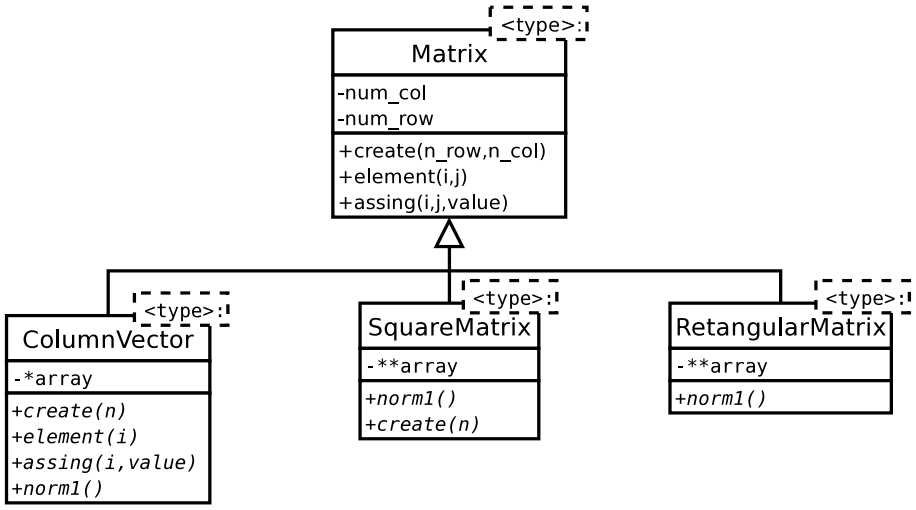

Figura 5.2: Diagrama UML de classes considerando uma hierarquia de matrizes.

matrizes, como no caso das matrizes quadradas que possuem mesmo número de linhas e colunas, podem ser utilizados. Além disso, alguns métodos podem ser sobrecarregados, como no caso do método `create(n_row,n_col)` e `create(n)` na classe `SquareMatrix`. Neste último caso, como a matriz é quadrada, não é necessário especificar o número de linhas e colunas, pois são iguais, sendo necessário apenas um deles. Em linguagem C++ essa metodologia é chamada sobrecarga de funções. Algumas operações como soma ou mesmo multiplicação de matrizes também podem ser sobrecarregadas na linguagem C++, onde elas adquirem funções específicas podendo ser adaptadas às operações matriciais.

Além da classe base `Matrix`, as classes `ColumnVector`, `SquareMatrix` e `RectangularMatrix` também podem ser classes genéricas, sendo usadas de acordo com o tipo de dados armazenados na matriz, definida por `<type>` na figura 5.2. Esse valor pode ser *double*, *float* ou *complex*, por exemplo, sendo o tipo especificado na declaração do objeto da classe. Assim, essas classes podem representar matrizes de diversos tipos, sendo que algumas operações devem ser ajustadas para

alguns deles, como por exemplo na inclusão de elementos no caso das matrizes complexas, que possuem parte real e imaginária. No entanto, esses detalhes são restritos à implementação das classes.

5.3 Hierarquia de classes de matrizes

A construção da hierarquia de classes para matrizes pode variar dependendo de quais características das matrizes devem ser levadas em conta. Uma das possibilidades é considerar a forma de armazenamento dos valores na matriz e algumas propriedades matriciais, como simetria e ortogonalidade.

Deste modo, vamos considerar um exemplo ilustrativo que considera a hierarquia de matrizes para programas seqüenciais, em princípio. Na figura 5.3 temos uma hierarquia de classe que considera a forma de armazenamento dos dados na matriz. Deste modo, temos uma classe base e as classe derivadas dessa, que possuem métodos implementados de acordo com a especialidade da classe.

Na classe base, temos os atributos `numRows` e `numColumns` que representam o número de linha e colunas na matriz, respectivamente. Esses valores são gerais para todas as matrizes e, portanto, podem ser definidos na classe base. Além disso, os métodos `element(i,j)` e `assign(i,j,elem)` são definidos considerando o acesso a um elemento e a inclusão de um valor na matriz como sendo geral, ou seja, em uma matriz densa. Portanto, na classe derivada `DenseMatrix` não é necessária uma nova implementação desses métodos.

Na classe base `DenseMatrix` são definidos o método `create`, reservando espaço para armazenamento, e o método `eigen`, que calcula os autovalores e autovetores da matriz. Esse último método é sobrecarregado e, de acordo com os parâmetros envolvidos na sua chamada, ele calcula apenas os autovalores ou também os autovetores. Na linguagem C++ esse recurso é obtido com sobrecarga

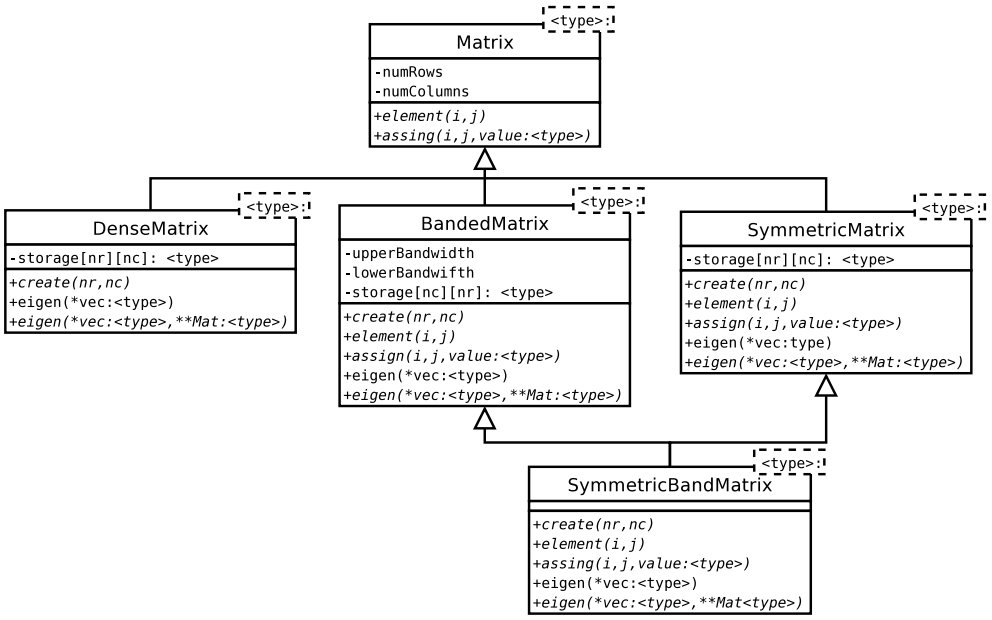

Figura 5.3: Hierarquia de classes considerando a forma de armazenamento na matriz

de funções, sendo que os operadores matemáticos, como soma e multiplicação, também podem ser sobrecarregados.

Nas classes derivadas `BandedMatrix` e `SymmetricMatrix`, são definidos métodos específicos de acordo com o armazenamento dos elementos na matriz. Assim, no caso da `BandedMatrix`, não é necessário armazenar todos os elementos da matriz, mas apenas os que estão localizados nas bandas superior e inferior. Desta forma, operações de acesso ou inclusão de elementos na matriz possuem a mesma interface que a classe base, mas implementação diferente. O mesmo ocorre com a classe `SymmetricMatrix`, que pode armazenar apenas a parte superior ou inferior da matriz, pois elas são iguais.

No caso da `SymmetricBandMatrix`, que é construída por herança das classes `BandedMatrix` e `SymmetricMatrix`, poderá usar a forma de armazenamento de uma delas. A implementação de seus métodos deverá considerar esse armazena-

mento sendo que alguns deles poderão ser herdados sem nova implementação.

Conforme notamos, as classes derivadas foram construídas como classes genéricas, independendo do tipo dos elementos que são armazenados na matriz. Assim, para cada uma delas é necessária a implementação de acordo com o tipo, usando novamente herança. Um exemplo é mostrado na figura 5.4, onde a partir da classe `BandedMatrix` são construídas por herança as classes para as matrizes de banda de acordo com os tipos de dados dos elementos armazenados na matriz.

A mesma construção deve ser feita para as demais classes da figura 5.3.

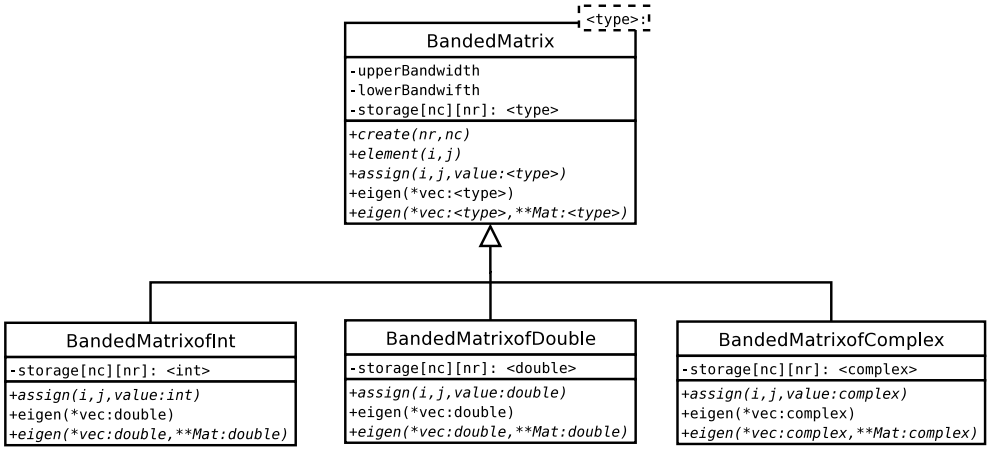

Figura 5.4: Diagrama de classes considerando o tipo de dado dos elementos armazenados na matriz.

No próximo capítulo vamos discutir a biblioteca que desenvolvemos e voltaremos a abordar os conceitos de orientação ao objeto.

Capítulo 6
Implementação da biblioteca POOLALi

A biblioteca que desenvolvemos, chamada *Parallel Object Oriented Linear Algebra Library* (POOLALi), foi construída usando os conceitos discutidos nos capítulos anteriores.

Inicialmente fizemos uma classificação das rotinas da biblioteca ScaLAPACK de acordo com o tipo das matrizes. No apêndice B é mostrada essa classificação, onde notamos que algumas rotinas não estão disponíveis para todos os tipos de matrizes, como no caso das rotinas de resolução de autovalores, que estão disponíveis apenas para as matrizes dos tipos simétrica e hermitiana. Deste modo, classificamos as matrizes de acordo com as operações mais gerais.

Como o objetivo de nosso trabalho era analisar aspectos de orientação ao objeto que facilitassem a construção de bibliotecas paralelas e não a construção de uma biblioteca completa, nos concentramos apenas nas matrizes densas e principalmente nas rotinas de resolução de autovalores e autovalores, uma vez que essa é uma operação bastante comum em problemas de física e engenharia, que muitas vezes necessitam da diagonalização de matrizes com elevado número de elementos, sendo que, em muitos casos, essa operação não é possível de ser feita

em computadores seqüenciais, conforme discutimos em capítulos anteriores.

Na implementação da POOLALi, utilizamos rotinas do ScaLAPACK para efetuar as operações de álgebra linear. Na utilização do ScaLAPACK são necessários quatro passos básicos:

- **Passo 1:** Iniciar a grade de processos.
- **Passo 2:** Distribuir a matriz na grade de processos.
- **Passo 3:** Chamar a rotina do ScaLAPACK.
- **Passo 4:** Liberar a grade de processos.

Assim, quando o programador necessita construir um *software* que utiliza as rotinas do ScaLAPACK, ele precisa especificar cada um desses passos no código do programa, tornando o processo de programação complexo, sendo que o programador necessita se preocupar com esses detalhes que oferecem um baixo nível de abstração.

No desenvolvimento da biblioteca POOLALi, procuramos ocultar esses detalhes de programação e oferecer um alto nível de abstração aos usuários.

6.1 Construção da grade de processos

No desenvolvimento da POOLALi nos preocupamos com os passo necessários no uso da ScaLAPACK, de forma a facilitar o processo de programação. Inicialmente, desenvolvemos uma classe chamada `Parallel`, que ao ser declarado um objeto desta classe, é inicializado o processo MPI através de um construtor e quando é terminada a execução, o destrutor da classe libera os processos. A interface dessa classe é constituída dos seguintes métodos

- `rank()`: retorna o *rank* associado ao processo,

- `nprocs()`: retorna o número de processos na grade.

Na figura 6.1 mostrada a interface completa da classe `Parallel`.

```
1 class Parallel
2 {
3  public:
4    Parallel(int &argc, char **&argv);
5    int rank();
6    int nprocs();
7    ~Parallel();
8 }
```

Figura 6.1: Métodos públicos da classe Parallel.

Deste modo, com a declaração de um objeto dessa classe é realizado parte do primeiro passo e ao final do programa, o quarto passo é executado automaticamente.

Para a iniciar a grade de processos, é necessária a chamada das funções do BLACS, conforme discutimos no capítulo 4. Na construção de nossa biblioteca criamos uma classe que executa essa operação automaticamente quando um objeto é inicializado, através do construtor da classe. Essa classe é chamada `Grid` e na declaração de um objeto associado a ela é necessária a passagem do objeto do tipo `Parallel`, que deve ser declarado anteriormente. Além disso, a grade pode ser construída de duas formas, podendo ser construída da forma mais quadrada possível de acordo com o número de processos, ou através da definição do número de linhas e colunas na grade, que é feita na declaração do objeto.

Os métodos oferecidos pela classe `Grid` são os seguintes:

- `nprow()`: retorna o número de linhas na grade de processos,
- `npcol()`: retorna o número de colunas na grade de processos,

```
1  class Grid
2  {
3   public:
4     Grid( Parallel &p , const int nr = 0, const int nc = 0 );
5     int nprow();
6     int npcol();
7     Context context();
8     int myrow();
9     int mycol();
10 }
```

Figura 6.2: Métodos públicos da classe Grid.

- `context()`: retorna o contexto associado à grade,
- `myrow()`: retorna a coordenada linha do processo na grade,
- `mycol()`: retorna a coordenada coluna do processo na grade.

Na figura 6.2 é mostrada a interface da classe `Grid`.

Assim, com a declaração dos objetos das classes `Parallel` e `Grid` os passos um e quatro são executados. Na figura 6.3 é mostrada a inicialização da grade de processos utilizando o ScaLAPACK de modo tradicional, usando linguagem C++, e na figura 6.4 usando a biblioteca POOLALi. Vemos que o uso da POOLALi simplifica a inicialização.

Comparando as duas implementações, vemos que usando a biblioteca POOLALi, os aspectos relativos às rotinas do BLACS ficam ocultos ao programador, uma vez que as rotinas são chamadas na implementação dos métodos. Quando é declarado um objeto do tipo `Grid`, as rotinas da figura 6.3 são chamadas automaticamente, além das variáveis serem inicializadas. Assim, vemos que a orientação ao objeto facilitou radicalmente a implementação da inicialização da grade de

```
1  int main(int argc, char *argv[])
2  {
3    int rank, nprocs, zero = 0, context, less_one = -1;
4    int nprow, npcol, myrow, mycol;
5    char erre = 'R';
6    MPI_Init(&argc,&argv);
7    blacs_pinfo__(&rank,&nprocs);
8    blacs_get__(&less_one, &zero, &context);
9    nprow = (int) sqrt(nprocs);
10   while (( nprow!=1 ) && ((nprocs \% nprow) != 0))
11     nprow--;
12   npcol = (int) (nprocs/nprow);
13   blacs_gridinit__(&context, &erre, &nprow, &npcol);
14   blacs_gridinfo__(&context, &nprow, &npcol, &myrow, &mycol);
15   blacs_exit__(&zero);
16   return 0;
17 }
```

Figura 6.3: Inicialização da grade de processos usando o ScaLAPACK.

processos.

6.2 Construção das hierarquia de classes

No desenvolvimento da hierarquia de matrizes utilizamos os mesmos métodos descritos no capítulo 5. Nos preocupamos apenas com as matrizes densas dos tipos simétrica e hermitiana. Para os demais tipos de matrizes, o método de construção das classes é análogo.

As classes para as matrizes simétricas e hermitianas foram adaptadas de acordo com o tipo de dados dos elementos armazenados na matriz. Assim, obtivemos

```
1 #include "grid.h"
2
3 int main(int argc, char *argv[])
4 {
5   Parallel p(argc, argv);
6   Grid g(p);
7   return 0;
8 }
```

Figura 6.4: Inicialização da grade de processos usando a biblioteca POOLALi.

uma hierarquia de classes conforme mostrada na figura 6.5, onde `elem_type` representa o tipo de dado armazenado na matriz, podendo ser float, double, complex ou double complex, e `precision_type` representa o tipo de dado dos autovalores encontrados, podendo ser float ou double.

A seguir vamos descrever essas classes.

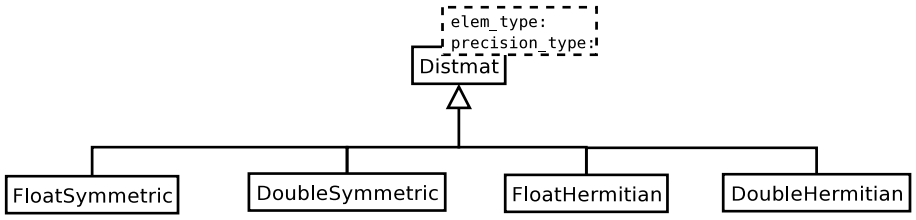

Figura 6.5: Diagrama UML da hierarquia de classes da biblioteca POOLALi.

6.2.1 Distmat

A classe `Distmat` é uma classe abstrata genérica, ou seja, os tipos dos elementos que ela armazena não são definidos em sua implementação, e ela possui algumas

funções virtuais. Deste modo, nenhum objeto pode ser declarado como pertencente a essa classe.

Como ela é uma classe geral, seu atributos possuem informações sobre o número de linhas e colunas da matriz global ou mesmo das matrizes locais distribuídas em cada processador.

No entanto, o usuário da classe não precisa ter conhecimentos desses detalhes de implementação, sendo necessário conhecer apenas sua interface. Os métodos oferecidos por essa classe são:

- `rows()`: retorna o número de linhas global,
- `cols()`: retorna o número de colunas global,
- `block_rows()`: retorna o número de linhas no bloco de distribuição,
- `block_cols()`: retorna o número de colunas no bloco de distribuição,
- `local_rows()`: retorna o número de linhas na matriz local,
- `local_cols()`: retorna o número de colunas na matriz local.

Além disso, sobrecarregamos alguns operadores que permitem efetuar as seguintes operações:

- **Acesso por índice global**: permite o acesso a partes locais da matriz especificando apenas seus índices globais. Essa operação é feita usando o operador `()`. Por exemplo, se temos um objeto chamado `Matriz`, essa operação é efetuada através do comando `Matriz(i,j)`, onde `i` e `j` são índices globais.
- **Acesso por índice local**: permite o acesso a partes da matriz especificando apenas seus índices locais.

Dentre essas operações, o acesso por índice global é fundamental para facilitar a construção de programas paralelos, pois na implementação de programas usando os métodos convencionais, o programador deve fazer um mapeamento entre os índices globais e locais das matrizes, o que, em muitos casos, pode ser uma a tarefa bastante complexa, principalmente quando os valores dos elementos dependem dos valores dos índices globais.

Assim, a construção da matriz usando esse operador de acesso global é feita de modo parecido ao realizado usando programação seqüencial, sendo a que a diferença básica está no fato que uma atribuição em linguagem C++ é feita usando `Matriz[i][j] = valor`, e na biblioteca POOLALi, `Matriz(i,j) = valor`. A implementação desse método define a qual processo é associado o elemento através dos índices globais.

O acesso local é feito usando `Matriz.local(i,j)`, que acessa a porção local a matriz em cada processador.

Na figura 6.6 apresentamos a interface completa da classe `Distmat` sem as funções virtuais, que serão mostradas na implementação das classes derivadas.

Os métodos que acabamos de discutir são herdados para as matrizes que descreveremos a seguir usando herança simples.

6.2.2 Classes derivadas

As classes `FloatSymmetric`,`DoubleSymmetric`, `FloatHermitian` e `DoubleHermitian` são usadas para matrizes do tipo simétrica e hermitiana cujos elementos são do tipo `float` ou `double`. Elas são construídas por herança da classe `Distmat`, herdando todos os seus métodos.

Além dos métodos herdados, essas classes possuem o método:

- `eigen`: calcula os autovalores e autovetores da matriz,

```
1  template<typename elem_type, typename precision_type>
2  class DistMat
3  {
4   public:
5     DistMat ( const int M, const int N, const Grid &g,
6               const int MB = 1, const int NB = 1 );
7     DistMat ( const DistMat<elem_type,precision_type> &A );
8     int rows();
9     int cols();
10    int block_rows();
11    int block_cols();
12    int local_rows();
13    int local_cols();
14    elem_type &operator()( const int I, const int J );
15    elem_type &local( const int i, const int j );
16    virtual ~DistMat();
17 }
```

Figura 6.6: Métodos públicos não virtuais da classe Distmat.

e o operador:

- $<<$: cada processo imprime sua matriz local em uma *stream*.

O método `eigen` é construído por sobrecarga de funções e, de acordo com as entradas na chamada do método, ele realiza uma determinada operação:

- `eigen(type *eigenval)`: calcula todos os autovalores da matriz e armazenados no vetor `eigenval`,
- `eigen(type *eigenval, int il, int iu)`: calcula os autovalores desde o *il-ésimo* até o *iu-ésimo*,

- `eigen(type *eigenval, type vl, type vu)`: calcula os autovalores com valores no intervalo $[\text{vl}, \text{vu}]$,
- `eigen(type *eigenval, ClassMat EigenVec)`: calcula todos os autovalores e autovetores da matriz,
- `eigen(type *eigenval, ClassMat EigenVec, int il, int iu)`: calcula os autovalores desde o *il-ésimo* até o *iu-ésimo* e os respectivos autovetores da matriz,
- `eigen(type *eigenval, ClassMat EigenVec, type vl, type vu)`: calcula os autovalores no intervalo $[\text{vl}, \text{vu}]$ e os respectivos autovetores da matriz.

Nestes métodos, `type` representa o tipo de dados dos elementos na matrizes, podendo ser `float` ou `double` e `ClassMat` é a classe associada à matriz, podendo ser `FloatSymmetric`,`DoubleSymmetric`, `FloatHermitian` ou `DoubleHermitian`. Nas figuras 6.7, 6.8, 6.9 e 6.10 são mostradas as interfaces das classes derivadas.

Assim, vemos que os métodos possuem no máximo quatro entradas, enquanto que na chamada das rotinas do ScaLAPACK que calculam os autovalores e autovetores, como a `pdsyevx`, necessitam de vinte nove entradas, sendo que muitas delas não são usadas no cálculo, dependendo da operação. O uso de orientação ao objeto neste caso facilita o trabalho de programação.

A implentação desses métodos utiliza rotinas do ScaLAPACK e para cada uma das diferentes classes que citamos acima, são usadas diferentes rotinas, pois as rotinas do ScaLAPACK estão organizadas de acordo com o tipo de matriz e com os tipos de dados que elas podem armazenar.

Outra facilidade que os métodos dessas classes oferecem é quanto à nomenclatura, que é mais intuitiva do que os nomes das rotinas oferecidas pelo ScaLA-

```
1  class FloatSymmetric : public DistMat<float,float>
2  {
3   public:
4     FloatSymmetric (const FloatSymmetric &A):DistMat<float,float>(A);
5     FloatSymmetric( const int M, const int N, Grid &g, const int MB = 1,
6                     const int NB = 1 ):DistMat<float,float>(M,N,g,MB,NB);
7     FloatSymmetric &operator=( FloatSymmetric A );
8     friend FloatSymmetric operator+( FloatSymmetric A,FloatSymmetric B);
9     friend FloatSymmetric operator-( FloatSymmetric A,FloatSymmetric B);
10    friend FloatSymmetric operator*( FloatSymmetric A,FloatSymmetric B);
11    void eigen( float *eigenval );
12    void eigen( float *eigenval, int il, int iu );
13    void eigen( float *eigenval, float vl, float vu );
14    void eigen( float *eigenval, DistMat<float,float> &eigenvec );
15    void eigen( float *eigenval, DistMat<float,float> &eigenvec,
16                int il, int iu );
17    void eigen( float *eigenval, DistMat<float,float> &eigenvec,
18                float vl, float vu );
19    ~FloatSymmetric(){};
20 }
```

Figura 6.7: Métodos públicos da classe FloatSymmetric.

PACK, conforme observamos na seção 4.4.1.

No desenvolvimento das classes, também sobrecarregamos algumas operações comuns entre matrizes, pois as matrizes estão distribuídas localmente em cada processador e as operações de soma e multiplicação não são triviais, sendo necessária a utilização das rotinas do ScaLAPACK e do PBLAS para executá-las, por exemplo. Nas figuras 6.7, 6.8, 6.9 e 6.10 são mostradas a interface desses operadores.

A operações implementadas são:

```
1  class DoubleSymmetric : public DistMat<double,double>
2  {
3   public:
4     DoubleSymmetric (const DoubleSymmetric &A):DistMat<double,double>(A);
5     DoubleSymmetric( const int M, const int N, Grid &g, const int MB = 1,
6                      const int NB = 1 ):DistMat<double,double>(M,N,g,MB,NB);
7     DoubleSymmetric operator=( DoubleSymmetric A );
8     friend DoubleSymmetric operator+( DoubleSymmetric A,DoubleSymmetric B);
9     friend DoubleSymmetric operator-( DoubleSymmetric A,DoubleSymmetric B);
10    friend DoubleSymmetric operator*( DoubleSymmetric A,DoubleSymmetric B);
11    void eigen( double *eigenval );
12    void eigen( double *eigenval, int il, int iu );
13    void eigen( double *eigenval, double vl, double vu );
14    void eigen( double *eigenval, DistMat<double,double> &eigenvec );
15    void eigen( double *eigenval, DistMat<double,double> &eigenvec,
16                int il, int iu );
17    void eigen( double *eigenval, DistMat<double,double> &eigenvec,
18                double vl, double vu );
19    ~DoubleSymmetric(){};
20 }
```

Figura 6.8: Métodos públicos da classe DoubleSymmetric.

- **Atribuição**: essa operação faz atribuição direta entre todos os elementos de duas matrizes. Se uma matriz possui uma dada distribuição de blocos, quando atribuída, ela poderá possuir uma nova distribuição. Assim, se `A` e `B` são matrizes de uma das classes que citamos acima, e `A` tem distribuição em blocos $m_a \times n_a$, e `B` em $m_b \times n_b$, então a atribuição `A = B` resulta que a matriz `A` possuirá seus elementos iguais ao de `B`, e com distribuição dos blocos $n_b \times n_b$. Para efetuar essa operação usando as rotinas de redistribuição do ScaLAPACK, é necessário escolher qual rotina utilizar, de acordo com o

```
class FloatHermitian : public DistMat<fcomplex,float>
{
 public:
   FloatHermitian (const FloatHermitian &A):DistMat<fcomplex,float>(A);
   FloatHermitian( const int M, const int N, Grid &g, const int MB = 1,
                   const int NB = 1 ):DistMat<fcomplex,float>(M,N,g,MB,NB);
   FloatHermitian operator=( FloatHermitian A );
   friend FloatHermitian operator+( FloatHermitian A,FloatHermitian B);
   friend FloatHermitian operator-( FloatHermitian A,FloatHermitian B);
   friend FloatHermitian operator*( FloatHermitian A,FloatHermitian B);
   void eigen( float *eigenval );
   void eigen( float *eigenval, int il, int iu );
   void eigen( float *eigenval, float vl, float vu );
   void eigen( float *eigenval, DistMat<fcomplex,float> &eigenvec );
   void eigen( float *eigenval, DistMat<fcomplex,float> &eigenvec,
               int il, int iu );
   void eigen( float *eigenval, DistMat<fcomplex,float> &eigenvec,
               float vl, float vu );
   ~FloatHermitian(){};
}
```

Figura 6.9: Métodos públicos da classe FloatHermitian.

tipo da matriz e os dados que ela armazena, e especificar as suas entradas, que são onze no total.

- **Adição**: A adição de matrizes é uma das operações fundamentais. Para realizá-la quando as matrizes estão distribuídas, essa tarefa pode não ser trivial. Deste modo, se `A`, `B` e `C` são matrizes das classes que discutimos acima, então a operação de adição `C = A + B` atribui a `C` a soma dos elementos das matrizes `A` e `B`, sendo que pode haver uma redistribuição dos

```
class DoubleHermitian : public DistMat<dcomplex,double>
{
 public:
   DoubleHermitian (const DoubleHermitian &A):DistMat<dcomplex,double>(A);
   DoubleHermitian( const int M, const int N, Grid &g, const int MB = 1,
                    const int NB = 1 ):DistMat<dcomplex,double>(M,N,g,MB,NB);
   DoubleHermitian operator=( DoubleHermitian A );
   friend DoubleHermitian operator+( DoubleHermitian A,DoubleHermitian B);
   friend DoubleHermitian operator-( DoubleHermitian A,DoubleHermitian B);
   friend DoubleHermitian operator*( DoubleHermitian A,DoubleHermitian B);
   void eigen( double *eigenval );
   void eigen( double *eigenval, int il, int iu );
   void eigen( double *eigenval, double vl, double vu );
   void eigen( double *eigenval, DistMat<dcomplex,double> &eigenvec );
   void eigen( double *eigenval, DistMat<dcomplex,double> &eigenvec,
               int il, int iu );
   void eigen( double *eigenval, DistMat<dcomplex,double> &eigenvec,
               double vl, double vu );
   ~DoubleHermitian(){};
}
```

Figura 6.10: Métodos públicos da classe DoubleHermitian.

dados, uma vez que a atribuição usa o operador de atribuição que descrevemos acima.

- **Subtração**: essa operação é análoga à descrita para a adição, sendo que neste caso, a operação `C = A - B` resulta que a matriz `C` possuirá os elementos da operação `A - B`.

- **Multiplicação**: Neste caso, a operação `C = A*B` armazena em `C` os elementos da multiplicação das matrizes `A` e `C`. É claro que neste caso, o número

de colunas de `A` deve ser igual ao número de linhas de `B`. A implementação desse método utliliza as rotinas do PBLAS, sendo que elas necessitam de dezoitos elementos como entrada.

Esses métodos favorecem o processo de programação, pois a utilização das rotinas do PBLAS e do ScaLAPACK podem ser complexas, devido à nomenclatura, ao alto número de rotinas (para diferentes tipos de matrizes e diferentes tipos de dados há rotinas diferentes) e à interface complexa, já que é necessário um alto número parâmatros na chamada das rotinas.

Portanto o uso de orientação ao objeto ajuda na simplificão da interface de uso das rotinas do ScaLAPACK e ajuda no desenvolvimento de programas.

6.3 Aspectos da implementação

Há alguns detalhes de uso do ScaLAPACK que tornam seu uso complexo, como no caso na definição do espaço de trabalho utilizado na diagonalização das matrizes. Uma forma de definir esse espaço é chamando a rotina de diagonalização, como a `pdsyevx` atribuindo às variáveis `lwork` e `liwork` o valor -1. Com isso, o ScaLAPACK calcula automaticamente o espaço necessário sendo necessária nova chamada da rotina para efetuar a diagonalização da matriz. Essa dupla chamada da função é feita automaticamente na implementação da biblioteca POOLALi, não sendo necessário, ao usuário da classe, se preocupar com esses detalhes.

Além disso, algumas variáveis necessárias nas chamadas das rotinas de diagonalização são inicializadas internamente na implementação da classe, sendo que na chamada do método de diagonalização são necessárias no máximo quatro entradas, ao invés das vinte e nove necessárias na chamada das rotinas do ScaLAPACK.

No próximo capítulo vamos mostrar um exemplo real da utilização da bibli-

oteca POOLALi e comparar essa aplicação com uma implementação do mesmo problema usando as rotinas do ScaLAPACK sem qualquer orientação ao objeto, onde vamos analisar o nível de *overhead* que a orientação ao objeto introduz e o desempenho que ela oferece.

CAPÍTULO

7

Análise de desempenho da biblioteca POOLALi

Conforme discutimos no capítulo anterior, a bibliteca POOLALi utiliza as rotinas do ScaLAPACK em sua implementação para realizar as operações de álgebra linear. Deste modo, o uso das classes que compõem a biblioteca introduz um *overhead* em relação ao uso do ScaLAPACK da maneira tradicional. Por isso, é importante analisarmos a influência desse *overhead* no tempo de processamento.

Neste capítulo, vamos discutir um problema de física do estado sólido que foi implementado usando o ScaLAPACK da maneira tradicional e usando o a biblioteca POOLALi. Vamos comparar as duas implementações e avaliar o desempenho da biblioteca que desenvolvemos.

7.1 O problema de física do estado sólido

Em física quântica, são comuns os problemas que tratam de matrizes com alto número de elementos. Uma das áreas importantes em que essa necessidade é comum é a física do estado sólido [59].

O problema que consideramos trata das estruturas semicondutoras baseadas

no GaAs e nitretos semicondutores. Uma das aplicações desse estudo é em computação quântica, na confecção de pontos quânticos. Detalhes dessa teoria e os métodos utilizados podem ser vistos em [60].

Esse problema é um exemplo de aplicação que não é possível de ser resolvida em computadores convencionais, pois para se diagonalizar as matrizes envolvidas no cálculo do ponto quântico, é necessária uma quantidade de memória RAM superior a 4 Gbytes, o que não está disponível no computadores convencionais. Deste modo, a paralelização torna-se fundamental não apenas do ponto de vista do desempenho.

O problema em questão, do ponto de vista computacional, pode ser resumido na construção da matriz hamiltoniana que compõe o problema e a sua diagonalização. A matriz hamiltoniana tem a seguinte forma:

$< jm_jk|H|j'm'_jk > \delta_{KK'} \Rightarrow$

$$\begin{pmatrix} \dots & \vdots & \vdots & \vdots & \vdots & \vdots & \dots \\ \dots & H(K_0) & H(K_1) & H(K_2) & H(K_3) & H(K_4) & \dots \\ \dots & H(K_1)* & H(K_0) & H(K_1) & H(K_2) & H(K_3) & \dots \\ \dots & H(K_2)* & H(K_1)* & H(K_0) & H(K_1) & H(K_2) & \dots \\ \dots & H(K_3)* & H(K_2)* & H(K_1)* & H(K_0) & H(K_1) & \dots \\ \dots & H(K_4)* & H(K_3)* & H(K_2)* & H(K_1)* & H(K_0) & \dots \\ \dots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} \tag{7.1}$$

Cada $H(K_i)$ na matriz corresponde a uma matriz 8×8, que é a soma dos termos de energia cinética, culombiana e do potêncial de troca e correlação.

$$H(K_i) = H_{k\cdot p}(K_i) + V_C(K_i) + V_{XC}(K_i) \tag{7.2}$$

Para a energia cinética, essa matriz 8×8, o hamiltoniano de Kane, é dada por:

$H_{p\cdot K}(K) \Rightarrow$

$$\begin{pmatrix}
E_g + A(k+K)^2 & 0 & iP_+ & \sqrt{\frac{2}{3}}P_z & i\sqrt{\frac{1}{3}}P_- & 0 & i\sqrt{\frac{1}{3}}P_z & \sqrt{\frac{2}{3}}P_- \\
0 & E_g + A(k+K)^2 & 0 & -\sqrt{\frac{1}{3}}P_+ & i\sqrt{\frac{2}{3}}P_z & -P_- & i\sqrt{\frac{2}{3}}P_+ & -\sqrt{\frac{1}{3}}P_z \\
-iP_- & 0 & Q & S & R & 0 & \frac{i}{\sqrt{2}}S & -i\sqrt{2}R \\
\sqrt{\frac{2}{3}}P_z & -\sqrt{\frac{1}{3}}P_- & S^* & T & 0 & R & \frac{-i}{\sqrt{2}}(Q-T) & i\sqrt{\frac{3}{2}}S \\
-i\sqrt{\frac{1}{3}}P_+ & -i\sqrt{\frac{2}{3}}P_z & R^* & 0 & T & -S & -i\sqrt{\frac{3}{2}}S^* & \sqrt{\frac{-i}{\sqrt{2}}}(Q-T) \\
0 & -P_+ & 0 & R^* & -S^* & Q & -i\sqrt{2}R^* & \frac{-i}{\sqrt{2}}S^* \\
-i\sqrt{\frac{1}{3}}P_z & -i\sqrt{\frac{2}{3}}P_- & \frac{-i}{\sqrt{2}}S^* & \frac{i}{\sqrt{2}}(Q-T) & i\sqrt{\frac{3}{2}}S & i\sqrt{2}R & \frac{1}{2}(Q+T)-\Delta & 0 \\
\sqrt{\frac{2}{3}}P_+ & -\sqrt{\frac{1}{3}}P_z & i\sqrt{2}R^* & -i\sqrt{\frac{3}{2}}S^* & \frac{i}{\sqrt{2}}(Q-T) & \frac{i}{\sqrt{2}}S & 0 & \frac{1}{2}(Q+T)-\Delta
\end{pmatrix} \tag{7.3}$$

A energia de *gap* é dada por:

$$E_g = E_c - E_v - \frac{\Delta}{3}, \tag{7.4}$$

sendo E_c e E_v as energias de fundo de banda de condução e do topo da banda de valência, respectivamente. Δ é a energia de sepração da banda de buracos *split-off*. Os demais termos da matriz são dados por:

$$P_\pm = P[(k_x + K_x) \pm i(k_x + K_y)], \tag{7.5}$$

$$P_z = P(k_z + K_z), \tag{7.6}$$

$$Q = -\frac{\hbar^2}{2m_0}(\gamma_1 + \gamma_2)[(k_x + K_x)^2 + (k_x + K_y)^2] + (_1{-}2_2)(k_z + K_z)^2, \tag{7.7}$$

$$T = -\frac{\hbar^2}{2m_0}(\gamma_1 - \gamma_2)[(k_x + K_x)^2 + (k_y + K_y)^2] + (\gamma_1 + 2_2)(k_z + K_z)^2, \tag{7.8}$$

$$S = i\frac{\hbar^2}{2m_0}\sqrt{3}[(k_x + K_x) - i(k_y + K_y)](k_z + K_z), \tag{7.9}$$

$$R = -\frac{\hbar^2}{2m_0}\sqrt{3}\gamma_2[(k_x + K_x)^2 - (k_y + K_y)^2] - 2i\gamma_3(k_x + K_x)(k_y + K_y). \tag{7.10}$$

A matriz do potencial é dada por:

$$V_C(K) \Rightarrow \begin{pmatrix} V_K & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & V_K & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & V_K & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & V_K & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & V_K & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & V_K & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & V_K & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & V_K \end{pmatrix} \tag{7.11}$$

O cálculo de V_K pode ser visto em [60].

Já a matriz que define o potencial de troca e correlação é dada por:
$V_{XC}(K) \Rightarrow$

$$\begin{pmatrix} E_g + A(k+K)^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & E_g + A(k+K)^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & Q & S & R & 0 & 0 & 0 \\ 0 & 0 & S^* & T & 0 & R & 0 & 0 \\ 0 & 0 & R^* & 0 & T & -S & 0 & 0 \\ 0 & 0 & 0 & R^* & -S^* & Q & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2}(Q+T) - \Delta & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2}(Q+T) - \Delta \end{pmatrix} \tag{7.12}$$

Com esses valores definidos, implementamos o programa que diagonaliza essa matriz.

7.2 Implementação

Para comparar o desempenho fornecido pela POOLALi com o uso padrão do ScaLAPACK implementamos duas versões de um programa que diagonaliza a matriz hamiltoniana descrita anteriormente. A primeira versão usa o procedimento padrão de construção de *softwares* usando o ScaLAPACK. Já a segunda utiliza a biblioteca POOLALi e seus recursos de orientação ao objeto.

Como estávamos interessados apenas na análise de desempenho da biblioteca que criamos e não nos resultados obtidos para o hamiltoniano, descrito anteriormente, implementamos um programa que constrói essa matriz hamiltonia considerando sua parte imaginária nula, sendo a matriz simétrica neste caso.

O código da inicialização da grade de processos usando as rotinas do BLACS é mostrado na figura 6.3, e usando a biblioteca POOLALi na 6.4. Conforme discutimos no capítulo anterior, a biblioteca POOLALi ofereceu uma grande simplificação do código na construção da grade de processadores.

Parte do código da construção da matriz usando a biblioteca POOLALi é mostrado na figura 7.1 e na figura 7.2 é mostrada a construção de modo tradicional. A diferença entre o código usando a POLALi e uma versão seqüencial está apenas na instrução `Matrix(i,j) = Hamilton(i,j,n)`, que deve ser substituída por `Matrix[i][j] = Hamilton(i,j,n)`. Assim, vemos que a biblioteca POOLALi possibilita a construção de matrizes de forma bastante parecida com a forma seqüencial, o que facilita o desenvolvimento do código.

```
1  #include "dsymmetric.h"
2  ...
3  DoubleSymmetric Matrix(N,N,g,8,8);
4  for( int i = 0; i < N; i++)
5   {...
6     for( int j=0; j< N; j++)
7              {...
8              Matrix(i,j) = Hamilton(i,j,n);
9              ...}
10  ...}
11 ...
```

Figura 7.1: Construção da matriz usando a biblioteca POOLALi.

```
#include "dsymmetric.h"
...
DoubleSymmetric Matrix(N,N,g,8,8);
for( int i = 0; i < N; i++)
 {...
   for( int j=0; j< N; j++)
      {...
         int Pr, Pc, l, m, x, y,i,j;
         Pr = (int)(floor( I/mb ));
         Pc = (int)(floor( J/nb ));
         if ( ( myrow == Pr ) && ( mycol == Pc ) )
         {
             l = (int)(floor( I/(nprow*mb) ) );
             m = (int)(floor( J/(npcol*nb) ) );
             x = I % mb;
             y = J % nb;
             i = l*mb + x;
             j = m*nb + y;
             Matrix[i][j] = Hamilton(i,j,n);
          }
               }
 ...}
...
```

Figura 7.2: Construção de modo tradicional sem o uso da biblioteca POOLALi.

A diagonalização da matriz usando a biblioteca POOLALi é bastante simples, mostrada na figura 7.3.

```
1 ...
2 DoubleSymmetric Vec(N,N,g,8,8);
3 double *eigenval;
4 eigenval = new (double)[N];
5 Matrix.eigen(eigenval, Vec);
6 ...
```

Figura 7.3: Diagonalização da matriz usando a biblioteca POOLALi.

No `array` `eigenval` ficam armazenados os autovalores da matriz e no objeto `Vec` os autovetores da matriz. Na figura 7.4 é mostrada essa mesma operação usando as rotinas do ScaLAPACK.

Conforme notamos na figura 7.4, a rotina `pdsyevx` necessita de um elevado número de entradas para efetuar a diagonalização da matriz. Além disso, para cálcular o espaço de necessário para a diagonalização, a rotina essa função deve ser chamada primeiramento com os valores das entras `lwork` e `liwork` definidas com valor menos um. Com isso, a função calcula automaticamente o espaço necessário e armazena esse valor nas variáveis `tmpwork` e `tmpiwork`.

Comparando os códigos das figuras 7.3 e 7.4, notamos que a biblioteca POOLALi oferece uma grande simplificação na construção do código, além de tornar o código mais amigável, já que o nome do método `eigen` é mais intuitivo do que o nome da função `pdsyevx`.

A orientação ao objeto ofereceu um alto nível de simplificação no código e oferece uma interface mais amigável do que a construção de *softwares* usando as rotinas do ScaLAPACK. Em nossa implementação, consideramos apenas as rotinas de diagonalização de matrizes. No entanto, esse mesmo resultado pode ser

```
... // declaração das variáveis de entrada na função
  static int lwork = -1, liwork = -1;
  double *work, tmpwork;
  int *iwork, tmpiwork;
  ifail = new int[n];
  iclustr = new int[2*nprow*npcol];
  gap = new double[nprow*npcol];
  pdsyevx_(&jobz, &range, &uplo, &n, ml, &ia, &ja, desca, &vl, &vu,
           &il, &iu, &abstol,&n_aut, &n_vec, eigenval, &orfac, z, &iz,
           &jz, descz, &tmpwork, &lwork, &tmpiwork, &liwork, ifail,
           iclustr, gap, &info);
  lwork = int(2*tmpwork);
  liwork =tmpiwork;
  work = new double[lwork];
  iwork = new int[liwork];
  pdsyevx_(&jobz, &range, &uplo, &n, ml, &ia, &ja, desca, &vl, &vu,
           &il, &iu, &abstol,&n_aut, &n_vec, eigenval,&orfac,z, &iz,
           &jz, descz, work, &lwork, iwork, &liwork, ifail,
           iclustr, gap, &info);
...
```

Figura 7.4: Diagonalização da matriz usando as rotinas do ScaLAPACK.

obtido para as demais operações permitidas pelo ScaLAPACK.

7.3 Análise de desempenho

A orientação ao objeto oferece um nível de `overhead` maior do que o uso das rotinas do ScaLAPACK. Para verificarmos o desempenho da biblioteca POOLALi, executamos o programa descrito anteriormente em um cluster de computadores, que foi descrito em 2.2, variando o tamanho a matriz e o número de processado-

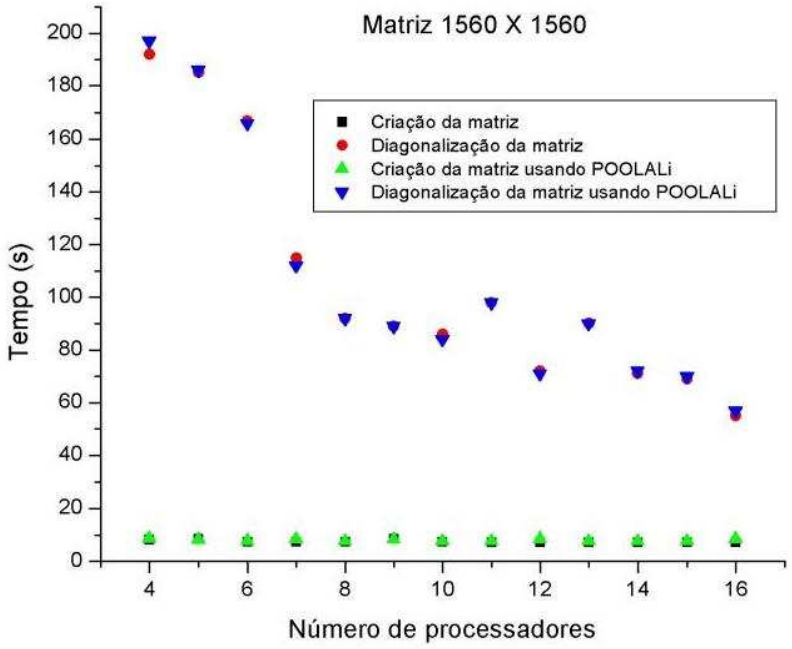

Figura 7.5: Tempos medidos para uma matriz 1560×1560.

res. Fizemos as mesmas medidas para a versão do programa usando as rotinas do ScaLAPACK e obtivemos os gráficos mostrados nas figuras 7.5, 7.6, 7.7 e 7.8. Conforme notamos, há pouca diferença entre os tempos de cálculo usando o ScaLAPACK tradicional ou usando a biblioteca POOLALi, o que mostra que o *overhead* que a programação orientada ao objeto adiciona não influencia no desempenho do programa.

Um fator importante observado nos gráficos é que há variações no tempo de processamento de acordo com a configuração da grade. Como variamos o número de processos de 4 a 16, usamos as configurações da grade conforme é mostrado na figura 7.9.

Conforme notamos nos gráficos das figuras 7.5, 7.6, 7.7 e 7.8, em alguns casos o aumento no número de processadores implica em tempo maior de processamento. Isso se deve ao fato de que as grades quadradas apresentam melhor

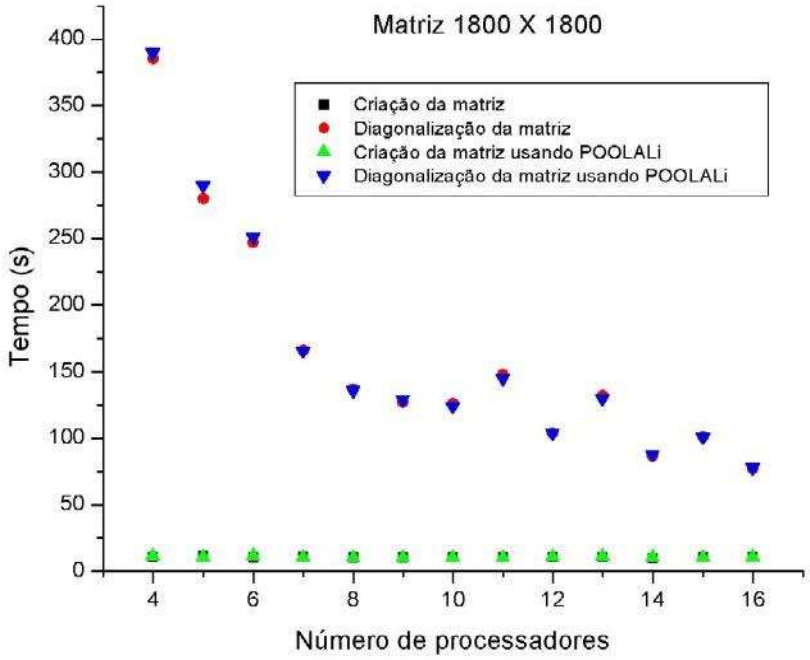

Figura 7.6: Tempos medidos para uma matriz 1800×1800.

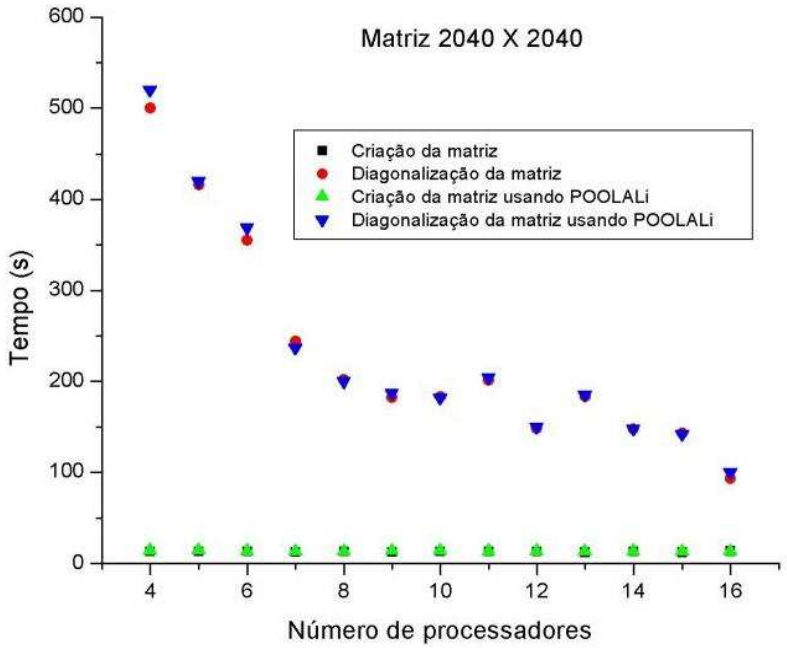

Figura 7.7: Tempos medidos para uma matriz 2040×2040.

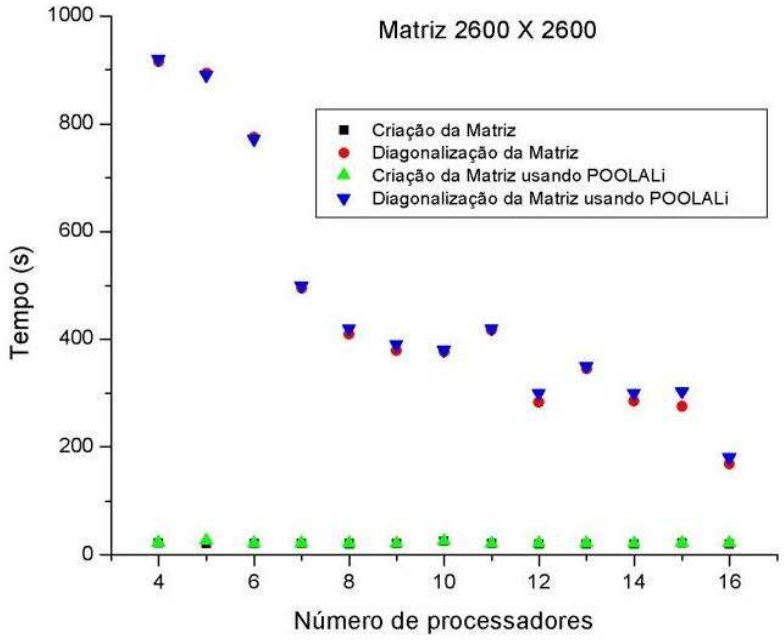

Figura 7.8: Tempos medidos para uma matriz 2600×2600.

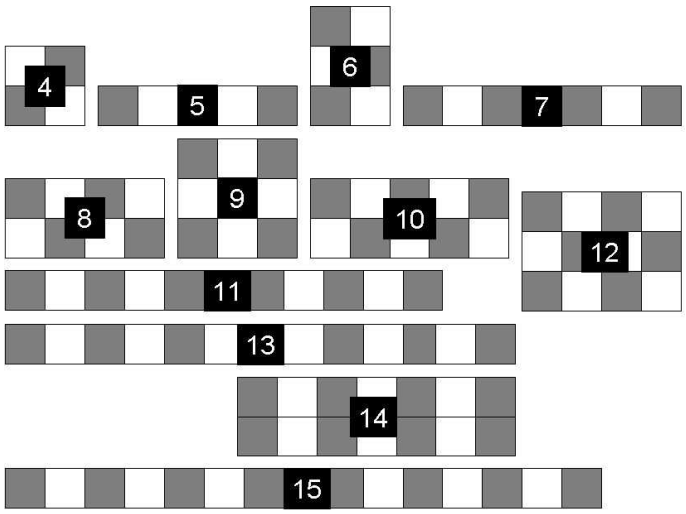

Figura 7.9: Configuração da grade de acordo com o número de processadores

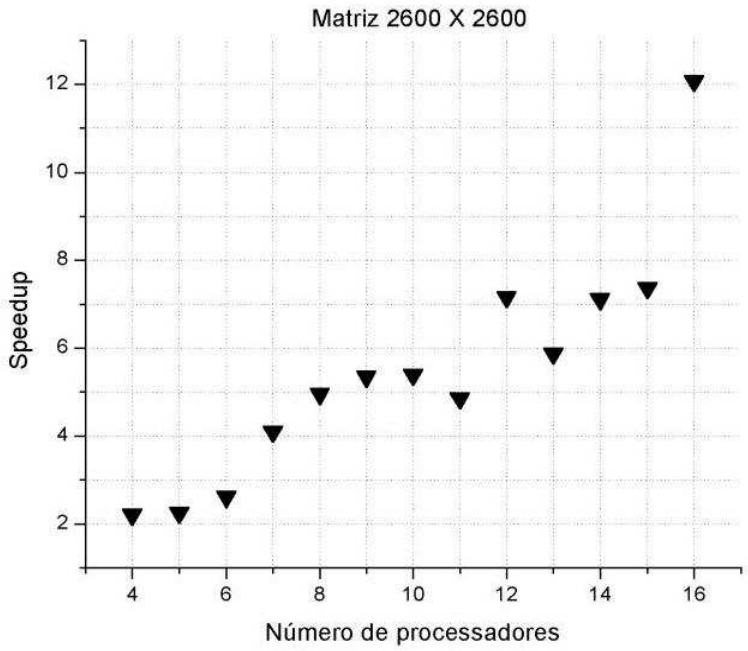

Figura 7.10: Curva de *speed-up* para uma matriz 2600×2600.

desempenho do que as retangulares, que por sua vez apresentam um desempenho melhor do que as grades em uma dimensão. Essa variação de desempenho é devido a tempo de comunicação entre processadores vizinhos.

Também observamos nos gráficos que as matrizes diagonalizadas com maior número de elementos, obtém melhor desempenho quando é aumentado o número de processadores, pois há um maior número de computações locais. Para verificarmos o ganho oferecido pela POOLALi, construímos a curva de *speed-up*, que é mostrada na figura 7.10. Conforme é observado, para 16 processadores, o ganho em relação à execução seqüencial é aproximadamente 12.

Analisando os resultados obtidos, vemos que a utilização da biblioteca POOLALi favoreceu o desenvolvimento de programas de álgebra linear paralelos e ela ofereceu desempenho análogo ao oferecido pelas rotinas do ScaLAPACK usado de modo tradicional. Portanto os recursos oferecidos pelas técnicas de orientação

ao objeto podem ser largamente usados na construção de bibliotecas paralelas que utilizem em sua implementação bibliotecas já existentes que não ofereçam uma interface amigável, como fizemos com o ScaLAPACK.

Capítulo 8
Conclusões

O uso de orientação ao objeto é uma forma de diminuir a distância intelectual entre o problema de álgebra linear e sua implementação paralela. Essa metodologia oferece um alto nível de abstração em álgebra linear e fornece recursos de encapsulação dos detalhes de implementação.

Neste trabalho, observamos que a construção de bibliotecas usando orientação ao objeto pode oferecer um nível de abstração maior do que qualquer outra metodologia. As bibliotecas tradicionais de álgebra linear oferecem recursos que facilitam o processo de programação. No entanto, a diferença entre a descrição matemática do problema e sua implementação é bastante grande. Conforme foi discutido, as rotinas que constituem o pacote ScaLAPACK oferecem bom desempenho na resolução de problemas de álgebra linear usando paralelismo, mas as suas interfaces, detalhes de construção da grade de processos e a nomenclatura das rotinas são os grande obstáculos que desestimulam seu uso.

A biblioteca POOLALi, que utiliza as rotinas do ScaLAPACK e suas bibliotecas auxiliares em sua implementação, é uma solução para essas limitações do ScaLAPACK, já que a interface das classes é bastante simplificada e a nomenclatura dos métodos é bastante intuitiva. Além disso, ela não oferece perda de desempenho nas chamadas das rotinas do ScaLAPACK, conforme notamos na

análise de desempenho discutida no capítulo 7.

8.1 Trabalhos futuros

O próximo passo imediato desse trabalho é a extensão da biblioteca POOLALi para todos os métodos de resolução oferecidos pelo ScaLAPACK, como a resolução de sistemas lineares, problemas de mínimos quadrados, fatorização LU, fatorização ortogonal, inversão matricial, etc, utilizando na implementação dos novos métodos as rotinas descritas no apêndice B. Essa implementação é análoga à que desenvolvemos neste trabalho, sendo necessária a criação de novas classes de matrizes, como as matrizes de banda, tridiagonais, ortogonais, triangulares, unitárias, esparsas e matrizes gerais. Além disso, há a possibilidade de desenvolver classes para as matrizes `out-of-core`.

Uma outra possibilidade é utilizar outras bibliotecas paralelas para implementar métodos que realizem operações não oferecidas pelo ScaLAPACK, como as bibliotecas utilizadas na resolução de equações diferenciais, como a [61], ou mesmo biblioteca usadas em aplicações específicas, como as de física das partículas.

Referências Bibliográficas

[1] Wikipedia, the free encyclopedia, `http://www.wikipedia.org`.

[2] Implementações do MPI, `http://www.mpi.nd.edu/MPI2`.

[3] L.S. Blackford, J. J. Dongarra, and R. C. Whaley. *ScaLAPACK Users' Guide*. Siam - Society for Industrial and Applied Mathematics, 1996.

[4] ScaLAPACK, `http://www.netlib.org/SCALAPACK/index.html`.

[5] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE*, pages 59–70, 1998.

[6] Alex Stepanov and Meng Lee. *The Standard Template Library*. Hewlett Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, February 1995.

[7] Frank B. Brokken. *C++ Annotations*. University of Groningen, 2001. Disponível no endereço: `http://www.icce.rug.nl/docs/cplusplus`.

[8] IML++, `http://math.nist.gov/iml++`.

[9] ARPACK++, `http://www.caam.rice.edu/software/ARPACK/arpack++.html`.

[10] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A sparse matrix library in c++ for high performance architectures. In *Second Object Oriented Numerics Conference*, pages 214–218, 1994.

[11] *SparseLib++ Sparse Matrix Class Library, User's Guide*. `http://math.nist.gov/sparselib++` .

[12] Mv++, `http://math.nist.gov/mv++` .

[13] The Template Numerical Toolkit (TNT), `http://gams.nist.gov/tnt`.

[14] FTensor, `http://www.oonumerics.org/FTensor`.

[15] Scilab home page, `http://www.scilab.org`.

[16] Goose: The GNU Object-Oriented Statistics Environment, `http://www.gnu.org/software/goose/goose.html`.

[17] The Objetct-Oriented Numerics Page, `http://www.oonumerics.org/oon`.

[18] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM Press, 1979.

[19] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. Matrix eigensystem routines: Eispack guide. In *Lecture Notes in Computer Science*, volume vol. 5. Springer-Verlag, 2^a edition, 1976.

[20] E. Anderson, Z. Bai, C. Demmel, J. Dongarra, and J. Du Croz. *LAPACK Users' Guide*. SIAM Press, 2^a edition, 1995.

[21] BLAS Technical Forum. *Document for the Basic Linear Algebra Subprograms Standard*, August 2001.

[22] BLAS home page, `http://www.netlib.org/blas`.

[23] Jack J. Dongarra, Roldan Pozo, and David W. Walter. LAPACK++: A design overview of object-oriented extensions for high performance linear algebra. In *Proceedings of the Object Oriented Numerics Conference*. IEEE Computer Society Press, 1993.

[24] Jack J. Dongarra, Roldan Pozo, and David W. Walker. An object-oriented design for high performance linear algebra on distributed memory architectures. In *Proceedings of the Object Oriented Numerics Conference*. IEEE Computer Society Press, 1993.

[25] William H. Press, Saul A. Teukolsky, Willian T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2^a edition, 1992.

[26] Harold S. Stone. *High-Performance Computer Architecture*. Addison-Wesley Publishing Company Inc, 1993.

[27] Andrew S. Tanembaum. *Organização Estruturada de Computadores*. Livros Técnicos e Científicos, 3^a edition, 1999.

[28] Gordon E. Moore. Camming more components onto integrated circuits. *Eletronics*, 38, 1965.

[29] Intel, `http://www.intel.com/research/silicon`.

[30] Iam Foster. *Designing and Building Parallel Programs*. Addison Wesley, 1995.

[31] D. E. Culler, J.P. Singh, and A. Grupta. *Parallel Computer and Architecture: A Hardware/Software Approach*. Morgan-Kaufman, 1999.

[32] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings Pub. Company, Redwood City, CA, 1989.

[33] Willian Stallings. *Arquiteturas e organização de computadores*. Prentice Hall, 5^a edition, 2002.

[34] The earth simulator center, `http://www.es.jamstec.go.jp`.

[35] Top500 supercomputer sites, `http://www.top500.org`.

[36] Jacek Radajewski and Douglas Eadline. *Beowulf HOWTO*, November 1998.

[37] Beowulf, `http://www.beowulf.org`.

[38] Haskell, A Purely Functional Language, `http://www.haskell.org`.

[39] Sisal - A High Performance, Portable, Parallel Programing Language, `http://www,llnl.gov/sisal`.

[40] K. A M. Ali. Execution of prolog on a multi-sequential machine. *Jornal of Parallel Programing*, 1998. Proc. of Implementation of Functional Language.

[41] PVM - Parallel Virtual Machine,`http://www.netlib.org/pvm3`.

[42] MPI Forum. *The Message Passing Interface Standard*, 1995. `http://www.mpi-forum.org`.

[43] Mpich - a portable implementation of mpi, `http://www-unix.mcs.anl.gov/mpi/mpich`.

[44] LAM/MPI Parallel Computing, `http://www.lam-mpi.org`.

[45] MPI Forum. *MPI-2: Extension to the Message-Passing Interface*, 1997.

[46] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3^a edition, 1996.

[47] Claude Cohen-Tannoudji, Bernard Diu, and Franck Laloë. *Quantum Mechanics*. A Wiley-Interscience Publication, John Wiley Sons, 2^a edition, 1977.

[48] LAPACK, `http://www.netlib.org/lapack`.

[49] E. Caron, S. Chaumette, S. Contassot-Vivier, F. Desprez, E. Fleury, and C. Gomez. Scilab to Scilab$_{//}$. The Ouragan Project, September 2002.

[50] PBLAS, `http://www.netlib.org/pblas`.

[51] L. Dongarra and R. Van De Geijn. Two dimensional basic linear algebra communication subprograms. In *Computer Science Dept. Technical Report*, pages 91–138, 1991.

[52] L. Dongarra and R. C. Walker. User's guide to the blacs v1.1. In *Computer Science Dept. Technical Report*, pages 95–281, 1995.

[53] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, 2nd edition, 1991.

[54] Royal Software Corporation. *Unified Modeling Language: Notation Guide*, version 1.1 edition, 1997. http://www.rational.com/uml/1.1.

[55] Peter Müller. *Introduction to Object-Oriented Programming Using C++*. Globewide Network Academy (GNA), November 1996.

[56] Bruce Eckel. *Thinking in C++*, volume 1. Mind View Inc, $2^n d$ edition, 1999.

[57] Bruce Eckel. *Thinking in C++*, volume 2. Mind View Inc, 2^nd edition, 1999.

[58] Brian Hayes. The Post-OOP Paradigm. *American Scientist*, 91(2):106–110, march-april 2003.

[59] Robert Eisberg and Robert Resnick. *Física Quântica*. Campus, 1979.

[60] Guilherme Matos Sipahi. *Teoria do confinamento de buracos em heteroestruturas semicondutoras do tipo delta-Dopping*. PhD thesis, Universidade de São Paulo, Instituto de Física, São Paulo, 1997.

[61] Nag parallel library,`http://www.nag.co.uk`.

[62] Willian Group and Ewing Lusk. *Installation Guide to MPICH, a Portable Implementation of MPI*. Argone National Laboratory, Mathematics and Computer Science Division, University of Chicago.

[63] BLACS, `http://www.netlib.org/blacs`.

[64] R. Clint Whaley. Installing and testing the BLACS v. 1.1. Technical report, U. S. Department of Energy, May 1997.

[65] L. S. Blackford, J. Dongarra, and J. Demmel. Instalation Guide for ScaLAPACK. Technical report, National Science Fundation, August 2001.

APÊNDICE

A
Instalação do ScaLAPACK

A instalação do ScaLAPACK pode ser dividida em diversas fases, uma vez que necessita de diversas bibliotecas auxiliares na sua execução. Vamos discutir cada passo da instalção.

A.1 MPICH

Como usamos a biblioteca MPICH, vamos descrever seu processo de instalação de forma simplificada, sendo que maiores detalhes podems ser encontrados em [62].

O código fonte do MPICH pode ser obtido em [43]. Inicialmente é nessário a descompactação do arquivo fonte do MPICH:

```
tar xvfz mpich.tar.gz.
```

A seguir é necessária a configuração do MPCHI, que é dada pelo comando

```
./configure -prefix = /usr/local/mpi
```

sendo que há várias opções de configuração, tais como a escolha da arquitetura que será usada ou mesmo o tipo de sistema de arquivos usado. Essas opções podem ser vistas em [62].

Após a configuração é necessária a compilação do MPICH, que é feita usando o comando

```
make >& make.log.
```

Para verificação da instalação pode-se executar alguns programas de teste, localizados no diretório `mpich/examples/basic`.

A.2 SCALAPACK

O arquivo de instalação do SCALAPACK pode ser encontrado em [4]. A sua descompactação pode ser feita com o comando

```
tar xvfz scalapack.tgz.
```

No entanto, antes da configuração do ScaLAPACK é necessária a instalação do BLACS, cujo código fonte está disponível em [63]. Sua descompactação é feita pelo comando

```
tar xvfz mpiblacs.tgz
```

e sua compilação é feita após a configuração do arquivo `Bmake.inc`, onde os caminhos da biblioteca MPICH e do ScaLAPACK devem ser especificados. Com o comando

```
make mpi
```

é feita sua compilação e construção de sua biblioteca. Detalhes de configuração podem ser vistos em [64].

Outro pacote necessário para instalação do ScaLAPACK é o BLAS, que possui seu código fonte em linguagens C e Fortran77, ambos disponíveis em [22]. A sua instalção é feita descompactando o arquivo fonte,

```
tar xvfz blas.tgz
```

e a compilação da biblioteca, no caso da versão escrita em Fortran77, pode ser feita usando o comando

```
f77 -O -c *.f.
```

Com os arquivos objeto, a biblioteca é construída por

```
ar cr blas_LINUX.a *.o.
```

Com esses processo concluídos, a instalação do ScaLAPACK pode ser finalizada, configurado o arquivo `SLamake.inc` e fazendo sua compilação pelo comando

```
make.
```

Detalhes da instalação da instalação podem ser vistos em [65].

A verificação do sucesso da instalação pode ser feita executando-se programas exemplos disponíveis em [4].

APÊNDICE

B
Rotinas do ScaLAPACK

A seguir são descritas as rotinas do ScaLAPACK associadas a cada tipo de matrix.

B.1 DB - De banda geral diagonalmente dominante

B.1.1 Driver routines

Equações lineares: Resolve um sistema de equações lineares do tipo $AX = B$ (sem pivotamento). Há rotinas para *Simple Drive (SV)* e *Expert Drive (SVX)*

- PSDBSV, PSDBSVX, PCDBSV, PCDBSVX, PDDBSV, PDDBSVX,PZDBSV, PZDBSVX.

B.1.2 Computational routines

Equações lineares

- PSDBTRF, PCDBTRF, PDDBTRF, PZDBTRF: Calcula a fatorização LU de uma matriz de banda geral sem pivotamento.

- PSDBTRS, PCDBTRS, PDDBTRS, PZDBTRS: Resolve um sistema de equações lineares $AX = B, A^T X = B$ ou $A^H X = B$, usando a fatorização LU computada por PxDBTRF.

- PSDBCON, PCDBCON, PDDBCON, PZDBCON: Estima o recíproco do número condicional $k(A) = ||A||.||A^{-1}||$. Necessita da norma da matriz original A e a fatorização retornada por PxDBTRF.

- PSDBRFS, PCDBRFS, PDDBRFS, PZDBRFS: Calcula os limites de erro no cálculo da solução (retornada por PxDBTRS) e refina a solução para reduzir o erro. Necessita que as matrizes originais A e B, a fatorização retornada por PxDBTRF e a solução X retornada por PxDBTRS.

- PSDBEQU, PCDBEQU, PDDBEQU, PZDBEQU: Calcula os fatores de escala para equilibrar A. Essas rotinas não atuam realmente na escala da matriz. Rotinas auxiliares como a PxLAQDB podem ser usadas pra esse propósito.

- PSDBTRSV, PDDBTRSV, PCDBTRSV, PZDBTRSV: Resolve ums sistema triangular de equações lineares. A rotina PxDBTRF deve ser chamada primeiro.

B.2 GT - Tridiagonal geral

B.2.1 Driver routines

Equações lineares:

- PSGTSV, PSGTSVX,PCGTSV, PCGTSVX, PDGTSV, PDGTSVX, PZGTSV, PZGTSVX

B.2.2 Computational routines

Equações lineares

- PSGTTRF, PCGTTRF, PDGTTRF, PZGTTRF: Calcula a fatorização.
- PSGTTRS, PCGTTRS, PDGTTRS, PZGTTRS: Usa a fatorização para resolver o sistema $AX = B$ por substituição. Necessita da fatorização retornada por PxGTTRF.
- PSGTCON, PCGTCON, PDGTCON, PZGTCON: Estima o recíproco do número condicional $k(A) = ||A||.||A^{-1}||$. Necessita da norma da matriz original A e a fatorização retornada por PxGTTRF.
- PSGTRFS, PCGTRFS, PDGTRFS, PZGTRFS: Calcula os limites de erro na computação da solução (retornada por PxGTTRS) e refina a solução para reduzir o erro. Necessita das matrizes originais A e B, da fatorização retornada por PxGTTRF e a solução X retornada por PxGTTRS.

B.3 DT - Tridiagonal geral sem pivotamento

B.3.1 Computational routines

Equações lineares

- PSDTTRF, PCDTTRF, PDDTTRF, PZDTTRF: Calcula uma fatorização LU de uma matriz tridiagonal geral sem pivotamento.
- PSDTTRS, PCDTTRS, PDDTTRS, PZDTTRS: Resolve um sistema de equações lineares $AX = B, A^T X = B$ ou $A^H X = B$, usando fatorização LU calculada por PxDTTRF.

- PSDTCON, PCDTCON, PDDTCON, PZDTCON: Estima o recíproco do número condicional $k(A) = ||A||.||A^1||$. Necessita da norma da matriz origial A e a fatorização calculada por PxDTTRF.

- PSDTRFS, PCDTRFS, PDDTRFS, PZDTRFS: Calcula os limites de erro na computação da solução (retornada por PxDTTRS) e refina a solução a fim de reduzir o erro. Necessita das matrizes originais A e B. a fatorização retornada por PxDTTRF e a solução X retornada por PxDTTRS.

- PSDTTRSV, PDDTTRSV, PCDTTRSV, PZDTTRSV: Resolve um sistema triangular tridiagonal de equações lineares. As rotinas PxDTTRF devem ser chamadas primeiro.

- PDDTSV: Resolve um sistema de equações lineares geral tridiagonal $AX = B$ sem pivotamento.

B.4 GB - De banda geral

B.4.1 Driver routines

Equações lineares:

- PSGBSV, PSGBSVX, PCGBSV, PCGBSVX, PDGBSV, PDGBSVX, PZGBSV, PZGBSVX

B.4.2 Computational routines

Equações lineares

- PSGBTRF, PCGBTRF, PDGBTRF, PZGBTRF: Calcula a fatorização LU de uma matriz de banda usando pivotamento parcial com troca de linhas. Pode trabalhar em uma matriz produzida por PxGBEQU e PxLAQGB.

- PSGBTRS, PCGBTRS, PDGBTRS, PZGBTRS: Resolve um sistema de equações lineares de banda ($AX = B$, $A^TX = B$ ou $A^HX = B$, usando fatorização LU calculada por PxGBTRF.

- PSGBCON, PCGBCON, PDGBCON, PZGBCON: Estima o recíproco do número condicional $k = ||A||.||A^1||$. Necessita da norma da matriz original A e a fatorização retornada por PxGBTRF.

- PSGBRFS, PCGBRFS, PDGBRFS, PZGBRFS: Calcula os limites de erro na computação da solução (retornada por PxGBTRS) e refina a solução de forma a reduzir o erro. Necessita das matrizes originais A e B e fatorização retornada por PxGBTRF e a solução X retornada por PxGBTRS.

- PSGBEQU, PCGBEQU, PDGBEQU, PZGBEQU: Calcula o fator de escala para equilibrar a a matriz A.

B.5 GE - General

B.5.1 Driver routines

Equações lineares: Simple Drive (SV), Expert Drive (SVX)

Resolve um sistema de equações lineares $AX = B$

- PSGESV, PSGESVX, PCGESV, PCGESVX, PDGESV, PDGESVX, PZGESV, PZGESVX

Linear Least Square Problems

- PSGELS, PCGELS, PDGELS, PZGELS: Resolve um sistema linear usando QR ou LQ fatorização.

- PSGELSX, PCGELSX, PDGELSX, PZGELSX: Usa fatorização ortogonal completa.

- PSGESVD, PCGESVD, PDGESVD, PZGESVD: Singular Value Decomposition (SVD): Calcula a decomposição de valor singular de uma matriz geral.

B.5.2 Computational routines

Equações lineares

- PSGETRF, PCGETRF, PDGETRF, PZGETRF: Calcula a decomposição LU de uma matriz geral usando pivotamento parcial com troca de linhas. Pode trabalhar com matrizes produzidas por PxGEEQU e PxLAQGE.

- PSGETRS, PCGETRS, PDGETRS, PZGETRS: Resolve um sistema de equações lineares $AX = B, A^T X = B$ ou $A^H X = B$, usando fatorização LU calculada por PxGETRF.

- PSGECON, PCGECON, PDGECON, PZGECON: Estima o recíproco do número condicional $k(A) = ||A||.||A^1||$. Necessita da norma da matriz original A e sua fatorização, retornada por PxGETRF.

- PSGERFS, PCGERFS, PDGERFS, PZGERFS: Calcula os limites de erro na computação da solução (retornada por PxGETRS) e refina a solução a fim de reduzí-lo. Necessita das matrize A e B e a fatorização retornada por PxGETRF e a solução X retornada por PxGETRS.

- PSGETRI, PCGETRI, PDGETRI, PZGETRI: Calcula o inverso de uma matriz geral usando a fatorização LU calculada por PxGETRF.

- PSGEEQU, PCGEEQU, PDGEEQU, PZGEEQU: Calcula o fator de escala para equilibrar uma matriz A.

Fatorização Ortogonal

- PSGEQPF, PCGEQPF, PDGEQPF, PZGEQPF: Fatorização QR com pivotamento.
- PSGEQRF, PCGEQRF, PDGEQRF, PZGEQRF: Fatorização QR sem pivotamento.
- PSGELQF, PCGELQF, PDGELQF, PZGELQF: Fatorização LQ sem pivotamento.
- PSGEQLF, PCGEQLF, PDGEQLF, PZGEQLF: Fatorização QL sem pivotamento.
- PSGERQF, PCGERQF, PDGERQF, PZGERQF: Fatorização RQ sem pivotamento.

Autoproblemas não simétricos

- PSGEHRD, PCGEHRD, PDGEHRD, PZGEHRD: Reduz uma matriz geral a forma superior de Hessember por uma transformação ortogonal (ou unitária) de similaridade.

Decomposição do valor singular

- PSGEBRD, PCGEBRD, PDGEBRD, PZGEBRD: Redução bidiagonal.

B.6 GG - Geral para problemas generalizados

B.6.1 Driver routines

Problemas lineares de mínimos quadrados (LSE)

- PSGGLSE, PCGGLSE, PDGGLSE, PZGGLSE: Resolve LSE usando fatorizaçao RQ.
- PSGGGLM, PCGGGLM, PDGGGLM, PZGGGLM: Resolve problemas lineares de mínimos quadrados generalizados usando fartorização QR.

B.6.2 Computational routines

Generalized Orthogonal Factorizations

- PSGGQRF, PDGGQRF, PCGGQRF, PZGGQRF: Calcula a fatorização QR.
- PSGGRQF, PDGGRQF, PCGGRQF, PZGGRQF: Calcula a fatorização RQ.

B.7 HE - Hermitiana

B.7.1 Driver routines

Autovalores padrões e autoproblemas de valor singular.

- PCHEEV, PZHEEV, PCHEEVX, PZHEEVX: Autoproblemas simétricos (*Symmetric Eigenproblem* (SEP)).
- PCHEGVX, PZHEGVX: Autoproblemas generalizados simétricos de finidos (*Generalized Symmetric Definite Eigenproblem* (GSEP)).

B.7.2 Computational routines

Problemas de autovalores simétricos generalizados definidos

- PCHEGST, PZHEGST: Reduz um autoproblema generalizado à sua forma padrão.

- PCHETRD, PZHETRD: Reduz uma matriz hermitina a uma hermitiana tridiagonal por uma transformação de similaridade.

B.8 OR - Ortogonal

B.8.1 Computational routines

Fatorização Ortogonal

- PSORGQR, PDORGQR, PSORMQR, PDORMQR: Fatorização QR.
- PSORGLQ, PDORGLQ, PSORMLQ, PDORMLQ: Fatorização LQ.
- PSORGQL, PDORGQL, PSORMQL, PDORMQL: Fatorização QL.
- PSORGRQ, PDORGRQ, PSORMRQ, PDORMRQ: Fatorização RQ.
- PSORGRZ, PDORGRZ, PSORMRZ, PDORMRZ: Fatorização RZ.

Autoproblemas simétricos

- PSORGTR, PDORGTR, PSORMTR, PDORMTR: Gera uma matriz após a redução dada por PxSYTRD.

Autoproblemas não simétricos

- PSORGHR, PDORGHR, PSORMHR, PDORMHR: Redução de Hesenberg.

Decomposição do valor singular

- PSORGBR, PDORGBR, PSORMBR, PDORMBR: Gera uma matriz após uma redução bidiagonal.

B.9 PB - Simétrica ou hermitiana, positiva definida de banda geral

B.9.1 Driver routines

Equações lineares:

- PSPBSV, PSPBSVX, PCPBSV, PCPBSVX, PDPBSV, PDPBSVX, PZPBSV, PZPBSVX

B.9.2 Computational routines

Equações lineares

- PSPBTRF, PCPBTRF, PDPBTRF, PZPBTRF: Calcula a fatorização de Cholesky.
- PSPBTRS, PCPBTRS, PDPBTRS, PZPBTRS: Resolve um sistema simétrico definido de banda.
- PSPBCON, PCPBCON, PDPBCON, PZPBCON: Estima o recíproco do número condicional $k(A) = ||A||.||A^{-1}||$. Necessita da norma da matriz original A e a fatorização retornada por PxPBTRF.
- PSPBRFS, PCPBRFS, PDPBRFS, PZPBRFS: Calcula os limites de erro no cálculo da solução (retornada por PxPBTRS) e refina a solução para reduzir o erro. Necessita que as matrizes originais A e B, a fatorização retornada por PxPBTRF e a solução X retornada por PxPBTRS.
- PSPBEQU, PCPBEQU, PDPBEQU, PZPBEQU: Calcula o fator de escala para equilibrar a matriz A.

- PSPBTRSV, PDPBTRSV, PCPBTRSV, PZPBTRSV: Resolve os sistema de equações lineares.

B.10 PO - Simétrica ou hermitiana positiva

B.10.1 Driver routines

Equações lineares:

- PSPOSV, PSPOSVX, PCPOSV, PCPOSVX, PDPOSV, PDPOSVX, PZPOSV, PZPOSVX

B.10.2 Computational routines

Equações lineares

- PSPOTRF, PCPOTRF, PDPOTRF, PZPOTRF: Calcula a fatorização de Cholesky.

- PSPOTRS, PCPOTRS, PDPOTRS, PZPOTRS: Resolve o sistema de equações usando a fatorização Cholesky calculada por PxPOTRF.

- PSPOCON, PCPOCON, PDPOCON, PZPOCON: Estima o recíproco do número condicional $k(A) = ||A||.||A^1||$. Necessita da norma da matriz origial A e a fatorização calculada por PxPOTRF.

- PSPORFS, PCPORFS, PDPORFS, PZPORFS: Calcula os limites de erro no cálculo da solução (retornada por PxPOTRS) e refina a solução para reduzir o erro. Necessita que as matrizes originais A e B, a fatorização retornada por PxPOTRF e a solução X retornada por PxPOTRS.

- PSPOTRI, PCPOTRI, PDPOTRI, PZPOTRI: Calcula o inverso da matriz usando a fatorização Cholesky calculada por PxPOTRF.
- PSPOEQU, PCPOEQU, PDPOEQU, PZPOEQU: Calcula o fator de escala para equilibrar a matriz A.

B.11 PT - Simétrica ou hermitiana positiva definida tridiagonal

B.11.1 Driver routines

Equações lineares:

- PSPTSV, PSPTSVX, PCPTSV, PCPTSVX, PDPTSV, PDPTSVX, PZPTSV, PZPTSVX.

B.11.2 Computational routines

Equações lineares

- PSPTTRF, PCPTTRF, PDPTTRF, PZPTTRF: Calcula a fatorização de Cholesky.
- PSPTTRS, PCPTTRS, PDPTTRS, PZPTTRS: Resolve um sistema usando fatorização de Cholesky calculada por PxPTTRF.
- PSPTCON, PCPTCON, PDPTCON, PZPTCON: Estima o recíproco do número condicional $k(A) = ||A||.||A^1||$. Necessita da norma da matriz original A e sua fatorização, retornada por PxPTTRF.
- PSPTRFS, PCPTRFS, PDPTRFS, PZPTRFS: Calcula os limites de erro na computação da solução (retornada por PxPTTRS) e refina a solução a fim

de reduzí-lo. Necessita das matrize A e B e a fatorização retornada por PxPTTRF e a solução X retornada por PxPTTRS.

- PSPTTRSV, PDPTTRSV, PCPTTRSV, PZPTTRSV: Resolve um sistema de equações lineares. A rotinas PxPTTRF deve ser chamada primeiro.

B.12 ST - Simétrica tridiagonal

B.12.1 Computational routines

Autoproblemas simétricos

- PSSTEBZ, PDSTEBZ, PCSTEBZ, PZSTEBZ: Autovalores via bissecção.
- PSSTEIN, PCSTEIN, PDSTEIN, PZSTEIN: Autovetores via iteração inversa.

B.13 SY - Simétrica

B.13.1 Driver routines

Autoproblemas simétricos

- PSSYEV, PDSYEV, PSSYEVX, PDSYEVX

Autoproblemas simétricos generalizados

- PSSYGVX, PDSYGVX

B.13.2 Computational routines

Autoproblemas simétricos

- PSSYTRD, PDSYTRD: Reduz uma matriz a sua forma tridiagonal por uma transformação de similaridade.

Autoproblemas simétricos definidos generalizados

- PSSYGST, PDSYGST

B.14 TR - Triangular

B.14.1 Computational routines

Equações lineares

- PSTRTRS, PCTRTRS, PDTRTRS, PZTRTRS: Usa fatorização para resolver $AX = B, A^T X = B$ ou $A^H X = B$.

- PSTRCON, PCTRCON, PDTRCON, PZTRCON: Estima o recíproco do número condicional $k(A) = ||A||.||A^{-1}||$. Necessita da norma da matriz original A e a fatorização retornada por PxTRTRF.

- PSTRRFS, PCTRRFS, PDTRRFS, PZTRRFS: Calcula os limites de erro no cálculo da solução (retornada por PxTRTRS) e refina a solução para reduzir o erro. Necessita que as matrizes originais A e B, a fatorização retornada por PxTRTRF e a solução X retornada por PxTRTRS.

- PSTRTRI, PCTRTRI, PDTRTRI, PZTRTRI: Usa fatorização ortogonal para calcular a inversa.

B.15 TZ - Trapezoidal

B.15.1 Computational routines

Fatorização ortogonal

PSTZRQF, PCTZRQF, PDTZRQF, PZTZRQF, PSTZRZF, PCTZRZF, PDTZRZF, PZTZRZF.

B.16 UN - Unitária

B.16.1 Computational routines

Fatorização ortogonal

- PCUNGQR, PZUNGQR, PCUNMQR, PZUNMQR: Fatorização QR.
- PCUNGLQ, PZUNGLQ, PCUNMLQ, PZUNMLQ: Fatorização LQ.
- PCUNGRQ, PZUNGRQ, PCUNMRQ, PZUNMRQ: Fatorização RQ.
- PCUNGRZ, PZUNGRZ, PCUNMRZ, PZUNMRZ: Fatorização RZ.

Autoproblemas simétricos

- PCUNGTR, PZUNGTR, PCUNMTR, PZUNMTR.

Autoproblemas não simétricos

- PCUNGHR, PZUNGHR, PCUNMHR, PZUNMHR.
- PCUNGBR, PZUNGBR, PCUNMBR, PZUNMBR.

B.17 HS - Matriz de Hesenberg

B.17.1 Computational routines

Autoproblemas não simétricos

- PSHSEQR, PCHSEQR, PDHSEQR, PZHSEQR: Fatorização Schur.
- PSHSEIN, PCHSEIN, PDHSEIN, PZHSEIN: Autovalores por iteração inversa.

B.17.2 LA - Rotinas auxiliares

- PSLAHQR, PDLAHQR, PCLAHQR, PZLAHQR: Calcula a fatorização Schur.