2

Algoritmos Matriciais em Processamento de Alto Desempenho

Nicolas Maillard[1]
(Universidade Federal do Rio Grande do Sul, nmaillard@inf.ufrgs.br)

Resumo:

Este mini-curso apresenta vários algoritmos para o cálculo matricial de alto-desempenho. Este tipo de operação é muito encontrado em cálculo científico e é a base de numerosos programas na área do PAD. Benchmarks clássicos, tais como o Linpack, do TOP500, também implementam cálculos matriciais. Com essa família de algoritmos, este mini-curso ilustra o ganho em desempenho obtido pela melhoria algorítmica, tipicamente através do aproveitamento da localidade nos acessos na memória (possivelmente distribuída).

A partir de algoritmos clássicos, tais como a fatoração LU ou ainda métodos iterativos, apresenta-se como chegar a algoritmos eficientes, bem como as bibliotecas e alguns códigos de cálculo onde os mesmos se encontram implementados para o uso comum. O leitor terá assim um melhor conhecimento das técnicas usadas nas BLAS, no *benchmark* Linpack ou ainda na biblioteca ARPACK.

[1]Prof. no Instituto de Informática da UFRGS

2.1. Introdução

Para desenvolver um programa de alto desempenho, diversos requisitos devem ser cumpridos, levando em conta os vários níveis de atuação do programador:

- otimização do hardware (processador/rede);
- adaptação do sistema operacional;
- uso de middlewares específicos (*e.g.* compiladores apropriados, bibliotecas para a programação paralela...);
- programação otimizada (*e.g.* uso de tipos de dados vetoriais);
- algoritmos com eficiência comprovada.

Neste mini-curso, serão focadas as duas últimas categorias mencionadas acima. Apresentar-se-á alguns algoritmos eficientes para o cálculo científico, uma das áreas que mais recorre ao Processamento de Alto Desempenho (PAD). Serão exemplificados os algoritmos matriciais, uma vez que os mesmos se encontram freqüentemente em tais cálculos, seja diretamente pela modelagem usada, seja após uma fase de resolução de equações diferenciais. O objetivo é que o leitor acompanhe o refinamento dos principais algoritmos até sua versão mais eficiente, tal como se encontram implementados em bibliotecas usadas em produção na área.

Considera-se neste texto um sistema de equações lineares definido pelos coeficientes $a_{i,j}, b_i, i = 1 \ldots N, j = 1 \ldots N$. Procura-se a solução $x_i, i = 1 \ldots N$ tal que

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,N}x_N & = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,N}x_N & = b_2 \\ \vdots & = \vdots \\ a_{N,1}x_1 + a_{N,2}x_2 + \ldots + a_{N,N}x_N & = b_N. \end{cases}$$

Usando uma notação matricial, escreve-se $Ax = b$. O sistema possui uma solução única se e somente se $\det A \neq 0$. Para calcular o vetor x, tem-se duas categorias de algoritmos: os chamados *diretos* efetuam transformações sobre os coeficientes da matriz A até fatorá-la numa forma (*e.g.* diagonal) que torne a resolução trivial. Os algoritmos *iterativos* usam a matriz apenas como operador para construir iterativamente uma seqüência de vetores que converge para a solução x.

Este texto se apresenta em duas seções. A primeira é dedicada aos métodos diretos. Começa com a eliminação de Gauss (fatoração LU) e ilustra técnicas de reordenação de laços para otimizar os acessos à memória. Nesta seção, a formulação dos algoritmos em blocos também é explicada, terminando com sua implementação

nas bibliotecas BLAS e LAPACK. Por fim, o exemplo do *benchmark* Linpack é citado como ilustração de uma implementação distribuída.

A segunda seção apresetna alguns métodos iterativos: o gradiente conjugado e o GMRES, para a resolução de sistemas de equações; e o cálculo de auto-valores pelo método de Arnoldi/Lanczos com *restart*. É dado um exemplo de aplicação com a resolução da equação de Schroedinger num ambiente distribuído.

2.2. Métodos diretos

Os métodos diretos usam transformações algébricas para simplificar a matriz e permitir a resolução simples do sistema. A vantagem desta família de algoritmos é sua exatidão matemática (quando não se leva em conta a precisão finita da representação dos números reais), e o fato de se obter uma forma fatorada da matriz que pode servir para a obtenção de mais de uma solução, por exemplo quando as condições nos limites (b_i) mudam sem que as equações mudem. A principal desvantagem é o fato das transformações destruírem a estrutura da matriz inicial: mais especificamente, os valores nulos da matriz, que geralmente não precisam ser armazenados em memória, podem se tornar não zero à medida que o algoritmo progride, necessitando assim um aumento no uso de memória. Assim, para grandes matrizes esparsas, preferir-se-á o uso dos métodos iterativos.

2.2.1. Eliminação de Gauss

Se $a_{11} \neq 0$, pode-se eliminar a variável x_1 nas equações $2 - N$, graças a seu valor dado pela equação 1. Basta calcular $l_{i1} = \frac{a_{i1}}{a_{11}}, \forall i = 2, \ldots, N$ e depois alterar os coeficientes a_{ij} assim:

$$a_{ij}^{(1)} \leftarrow a_{ij} - l_{i1} * a_{1j}.$$

Assim se obtém um novo sistema equivalente ao inicial:

$$\begin{cases} a_{1,1}^{(1)}x_1 + a_{1,2}^{(1)}x_2 + \ldots + a_{1,N}^{(1)}x_N & = b_1^{(1)} \\ a_{2,2}^{(1)}x_2 + \ldots + a_{2,N}^{(1)}x_N & = b_2^{(1)} \\ \vdots & = \vdots \\ a_{N,2}^{(1)}x_2 + \ldots + a_{N,N}^{(1)}x_N & = b_N^{(1)}, \end{cases}$$

onde $b_1^{(1)} = b_1$ e $b_i^{(1)} = b_i - l_{i1}b_1, \forall i \geq 2$, e $a_{1j}^{(1)} = a_{1j}, \forall j$.

No caso que $a_{11} = 0$, sempre se pode trocar a linha 1 com uma outra linha $i \geq 2$ cujo primeiro coeficiente não seja zero. Existe obrigatoriamente pelo menos uma tal linha, pois caso contrário o determinante da matriz seria zero.

Após essa primeira etapa de substituição, o novo sistema $A^{(1)}x = b^{(1)}$ tem sua primeira coluna, menos seu coeficiente $a_{1,1}^{(1)}$, zerada. Pode-se repetir o procedimento de substituição dos elementos na sub-matriz formada pelas linhas e colunas $2, 3, \ldots, N$. Dessa forma, vai-se eliminar a variável x_2 nas linhas $3, 4, \ldots N$. Iterativamente, a cada etapa $k = 2, \ldots, N-1$, são calculados os valores: $l_{ik} = \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}, \forall i = k, \ldots, N$. Depois, os coeficientes $a_{ij}^{(k-1)}$ e $b_i^{(k-1)}$ são alterados assim:

$$a_{ij}^{(k)} \leftarrow a_{ij}^{(k-1)} - l_{ik} \times a_{ij}^{(k-1)}, \tag{2.1}$$

$$b_i^{(k)} \leftarrow b_i^{(k-1)} - l_{ik} \times b_k^{(k-1)}. \tag{2.2}$$

Na etapa $k = N - 1$, obtém-se o sistema seguinte:

$$\begin{cases} a_{1,1}^{(N-1)}x_1 + a_{1,2}^{(N-1)}x_2 + \ldots + a_{1,N}^{(N-1)}x_N & = b_1^{(N-1)} \\ a_{2,2}^{(N-1)}x_2 + \ldots + a_{2,N}^{(N-1)}x_N & = b_2^{(N-1)} \\ \vdots & = \vdots \\ a_{N,N}^{(N-1)}x_N & = b_N^{(N-1)}. \end{cases}$$

Esta última forma permite uma resolução direta do sistema para calcular o vetor x. Notando agora $u_{ij} = a_{i,j}^{(N-1)}$, tem-se diretamente

$$x_N = \frac{b_N^{(N-1)}}{u_{NN}}; \forall i = N-1, \ldots, 1, x_i = \frac{(b_i^{(N-1)} - \sum_{j=i+1}^{N} u_{ij}x_j)}{u_{ii}}.$$

Mostra-se facilmente que as fórmulas acima se resumem na notação matricial seguinte:

Teorema 1 (Fatorização LU) *Dadas as matrizes $L = (l_{ij})_{i,j=1,\ldots,N}$ e $U = (u_{ij})_{i,j=1,\ldots,N}$ definidas a partir dos coeficientes acima calculados (sendo que $l_{ij} = 0 \ \forall i < j$, $l_{ii} = 1$ e $u_{ij} = 0 \ \forall i > j$), a matriz A se fatora como:*

$$PA = LU.$$

P é uma matriz de permutação que integra as permutações de linhas para procurar um coeficiente não zero chamado pivô.

As linhas e colunas de P são constituídas de vetores unidade.

Freqüentemente, é interessante calcular uma vez só a fatoração LU da matriz A, para depois calcular a solução x.

Complexidade. O custo da fatoração LU é o seguinte. Precisa-se, na etapa j, de:

- $N - j$ divisões pelo pivô;
- $2(N - j)^2$ multiplicações e somas.

Logo, o total de operações para a fatoração é $\sum_{j=1}^{N-1} j + 2j^2 = \frac{2N^3}{3} + \frac{N^2}{2} + \mathcal{O}(N)$.

Estabilidade numérica e escolha do pivô. Seja A a matriz seguinte (exemplo devido a Forsythe):

$$\begin{cases} 1,00 \cdot 10^{-4} x_1 + 1,00 x_2 &= 1,00 \\ 1,00 x_1 + 1,00 x_2 &= 2,00 \end{cases}.$$

A solução exata é $x_1 = 1/0,9999 = 1,00010001\ldots$, $x_2 = 0,9998/0,9999 = 0,99989998\ldots$. Supondo que executaremos a fatoração de Gauss com uma representação em vírgula flutuante usando 3 casas decimais, no caso que se escolha $a_{11} = 1,00 \cdot 10^{-4}$ como pivô, obtém-se a solução $x_2 = 1,00$ (solução certa), mas $x_1 = (b_1 - a_{12}x_2)/a_{11} = 0$. O resultado numérico é, portanto, errado. Já no caso que se escolha a_{a1} como pivô, o mesmo valerá $1,00$ e obter-se-á $x_2 = 1,00$ e $x_1 = 1,00$. Ou seja, o resultado será numericamente correto. Olhando com detalhe, se percebe que o problema de perda de precisão acontece cada vez que o coeficiente l_{ik} é grande, ou seja quando o coeficiente a_{ik} é pequeno. Logo, deve-se escolher por pivô o coeficiente da linha não zero que seja de maior valor absoluto. O livro [HIG 96] trata deste problema em detalhe.

Pivô parcial e pivô total. Além de trocar as linhas abaixo da linha corrente para se usar o maior pivô possível, pode-se também trocar as colunas à direita da coluna da fatoração corrente. Maximiza-se assim o valor do pivô. O inconveniente é que neste caso se deve armazenar as trocas de colunas para efetuá-las também sobre as coordenadas do vetor solução. Quando se usa apenas a troca de linhas, fala-se de algoritmo com pivô parcial; quando se usa também a troca de colunas, além da troca de linhas, se fala de algoritmo com pivô total.

2.2.2. Outras decomposições

Quando a matriz A é simétrica e definida positiva, o mesmo algoritmo de Gauss pode ser simplificado (não é mais preciso buscar o maior pivô) e permite obter a decomposição de Choleski de A: $A = LDL^t = (LD^{(1/2)}) \cdot (LD^{(1/2)})^t$, onde D é uma matriz diagonal com coeficientes positivos.

Outra fatoração de uso frequente é a decomposição QR de uma matriz A, Q sendo ortogonal (*i.e.* $QQ^t = Q^tQ = I_N$) e R triangular superior. Essa fatoração é de interesse para calcular bases de vetores ortogonais, bem como possibilitar a expressão do operador linear representado pela matriz nesta base. A técnica usada consiste também na aplicação iterativa de matrizes de transformações sobre A (no caso, aplicando por exemplo rotadores de Givens). O custo algorítmico é também em $\mathcal{O}(N^3)$.

2.2.3. Algoritmos em blocos

Um dos componentes chave para se obter um bom desempenho é a organização dos acessos à memória, para usar melhor o deslocamento dos dados de um nível da hierarquia de memória para o outro. Em um ambiente distribuído, essa consideração é ainda mais crítica, uma vez que um acesso não local à memória pode implicar em comunicações via rede ou em mecanismos de sincronização.

2.2.3.1. Produto matricial e acessos à memória

O exemplo mais simples, e fundamental, é o algoritmo para multiplicar duas matrizes A e B, de tamanho $N \times N$. A matriz resultado será denotada C. O cálculo é trivial: $\forall i, j = 1 \ldots N, c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$. Para efetuá-lo, basta então os três laços clássicos do algoritmo 1.

Algoritmo 1 Produto Matricial, algoritmo `ijk` trivial.

1: Entradas: 2 matrizes A e B de tamanho $N \times N$.
2: Saída: 1 matriz C de tamanho $N \times N$.
3:
4: $c_{ij} \leftarrow 0$
5: **Para** $i = 1, 2, \ldots, N$ **Faça**
6: **Para** $j = 1, 2, \ldots, N$ **Faça**
7: $c_{ij} \leftarrow 0$
8: **Para** $k = 1, 2, \ldots, N$ **Faça**
9: $c_{ij} \leftarrow c_{ij} + a_{ik} b_{kj}$
10: **Fim Para**
11: **Fim Para**
12: **Fim Para**

No entanto, esses laços aninhados são especialmente mal projetados no que diz respeito ao acesso à memória. Para se ter uma noção melhor disso, vamos supor que a memória seja composta de dois níveis, sendo um de acesso rápido e de

capacidade tal que $\mathcal{C}$ elementos de matriz possam caber nela, e o outro de capacidade bem maior, mas de acesso muito mais lenta (observa-se que esta hipotése serve tanto para modelar um sistema de Cache, como uma máquina distribuída onde se distingue o acesso à memória local do acesso a dados remotos através de transmissão pela rede. Neste estudo, não se consideram mecanismos avançados tais como o *prefetch* que alguns processadores fornecem, ou ainda a possibilidade de mascarar comunicações com cálculos). Supõe-se que um acesso a um dado d na memória lenta, que não está armazenado na memória rápida (*miss*), provoca a atualização da mesma com d, bem como com os δ elementos próximos de d na memória lenta (mecanismo de paginação). Para simplificar, supõe-se que δ divide N. Geralmente, δ é pequeno o suficiente para que se possa atualizar δ coeficientes, sem que saiam da memória rápida todos os outros valores nela armazenada (no máximo $\delta = \mathcal{C}/3$). Por fim, vamos supor que, na memória lenta, os coeficientes de uma matriz são armazenados num espaço contínuo, ordenando os valores por colunas ("column major", ou seja segundo a implementação em Fortran. A linguagem C e a maioria das outras linguagens de programação usam a implementação contrária, segundo as linhas).

A instrução 1 do algoritmo `ijk`, ao ser executada pela primeira vez para uma dupla i, j dada, vai causar 3 *misses*, um para cada acesso a um dos coeficientes da matriz. Entretanto, nas $\delta - 1$ iterações seguintes de k, o coeficiente a_{ik} já se encontrará na memória rápida, porém devido à ordenação por colunas, os b_{kj} e c_{ij} não estarão: ter-se-á então 2 *misses* por cada uma dessas $\delta - 1$ iterações. Na iteração número $k = \delta + 1$, o coeficiente a_{ik} não estará mais, pois a capacidade da memória rápida será ultrapassada: ter-se-á novamente 3 *misses*, mas as $\delta - 1$ iterações seguintes provocarão novamente apenas 2 *misses*. Continuará assim até as N iterações terem sido feitas. Logo, o número de *misses* será $2N + 3\frac{N}{\delta}$ para cada valor de i, j.

Quando se termina o laço k, incrementa-se j, o que provoca nas novas N iterações de k um *miss* de c_{ij} e um b_{kj}. O último a_{ik} acessado foi a_{iN} e no início do laço k se quer a_{i0} que também não estará na memória rápida: volta-se a ter 3 *misses* no ínicio de cada laço k. Logo, conclui-se que os três laços aninhados i, j e k provocarão $N^3(2 + \frac{3}{\delta})$ *misses*.

Na verdade, pode-se executar os três laços em qualquer ordem: a ordem `ijk` apresentada acima é apenas uma das seis possibilidades. Assim como vai ser visto, não é a melhor, apesar de ser a mais natural devido à apresentação matemática da fórmula do produto matricial. Primeiramente, repara-se que para usar o mecanismo de atualização eficientemente, deve-se ordenar os laços de forma tal que os índices das linhas variem mais rapidamente. Na linha 9, as duas variáveis que aparecem como índices de linhas são unicamente i e k. Logo, as ordens `ikj` e `kij`, que vão fazer com que j varie mais freqüentemente, são naturalmente menos eficientes e causarão o maior número de *misses* (pode-se verificar que este chega a fazer três

misses a cada iteração, ou seja $3N^3$ no total).

Sobram então as ordens `jik`, `jki` e `kji`. No primeiro caso, a análise é igual à do caso `ijk` detalhado acima para um dado valor i, j. Porém, ganha-se alguns *misses*, uma vez que quando um laço k acaba, agora é o índice i que aumenta. Logo, o coeficiente c_{ij}, durante δ iterações de i, estará presente na memória rápida, pois terá sido trazido junto ao da iteração anterior. Assim, em comparação com o `ijk`, poupar-se-ão $N(N - N/\delta)$ *misses* (*i.e.* $N - N/\delta$ por valor de j) com a ordem jik, com um total de *misses* igual a $N^3(1 + \frac{3}{\delta}) - N^2(1 - \frac{1}{\delta})$.

No caso `jki`, o resultado é melhor: uma iteração sobre i provoca, na primeira vez ($i = 1$), um *miss* para cada um dos três acessos a_{ik}, b_{kj} e c_{ij}. No entanto, nas $\delta - 1$ iterações seguintes, o coeficiente b_{kj} obviamente ficou na memória rápida, e os a_{ik} e c_{ij} já se encontram nela graças ao mecanismo de atualização. Assim, ter-se-á apenas 2 *misses* a cada δ iterações em i, ou seja $\frac{2N}{\delta}$ para um dado valor de j, k. Quando k é incrementado, o a_{ik} não se encontra mais; o c_{ij} que acabou de ser acessado era o c_{Nj}, logo o c_{1j} não estará também na memória rápida. No entanto, ter-se-á o b_{kj} na memória, até que δ valores adjacentes da coluna k de B tenham sido acessadas. Logo, por valor de j, teremos $N - \frac{N}{\delta}$ *misses* a menos do que no caso em que cada iteração em k provoca $\frac{2N}{\delta}$. Afinal, por valor de j, ter-se-á então $N\frac{2N}{\delta} - N(1 - \frac{1}{\delta})$. Quando o j aumenta, obtém-se um *miss* no b_{kj} e os dois já explicados nos dois outros coeficientes. Assim, no total se obtém $\frac{2N^3}{\delta} - N^2(1 - \frac{1}{\delta})$ *misses* neste algoritmo.

O caso `kji` se estuda da mesma forma. Ele não é tão bom quanto o anterior, uma vez que uma iteração de j provoca automaticamente um *miss* no coeficiente b_{kj}, enquanto o `jki` o poupava freqüentemente ($N - N/\delta$ vezes).

Deste estudo, conclui-se que a ordem `jki` é a mais favorável para a implementação do produto matricial. Nota-se que entre a forma "imediata" `ijk` que provoca $N^3(2 + \frac{3}{\delta})$ *misses* e a versão `jki` que provoca $\frac{2N^3}{\delta} - N^2(1 - \frac{1}{\delta})$, a razão de *misses* (*i.e.* de transferência entre os dois tipos de memória) é igual a

$$\frac{N^3(2 + \frac{3}{\delta})}{\frac{2N^3}{\delta} - N^2(1 - \frac{1}{\delta})} = \frac{N(2\delta + 3)}{2N - \delta + 1} \underset{N\to\infty}{=} \delta.$$

No entanto, pode-se melhorar ainda o uso dos acessos à memória através da formulação do produto matricial em blocos. O algoritmo é igual ao algoritmo 1, porém onde se usa um coeficiente (*e.g.* a_{ik}) passa-se a usar um bloco de matriz de tamanho $b \times b$ (*e.g.* $a_{i=1,\ldots,b;j=1,\ldots,b}$).

Quando se acessam os coeficientes por linha ou coluna na memória, e independentemente da ordem de acesso, é preciso o total de $\mathcal{O}(N^3)$ acessos à memória lenta para ler os coeficientes e armazenar a matriz C resultante. Agora, supondo que o tamanho b dos blocos seja calculado de uma forma tal que se pelo menos 3 blocos

caibam na memória rápida, o cálculo do bloco $C_{ij}^{(k)} = A_{ik} \times B_{kj}$ necessita de apenas $2b^2$ acessos à memória distante. Precisa-se de N/b tais blocos para calcular a soma C_{ij} (que pode ser acumulada) na memória rápida. Desta forma, precisa-se de $\frac{N}{b} \times 2b^2 = 2Nb$ acessos à memória lenta para ter um dos $(N)^2$ blocos da matriz C. Afinal, serão então $2Nb \times \frac{N^2}{b^2} = \frac{2N^3}{b}$ acessos a serem realizados por essa versão em blocos. Em relação à versão sem blocos, ganha-se um fator b em número de acessos à memória.

Nota-se que o mesmo raciocínio tem toda validade num ambiente distribuído: os acessos à memória serão mensagens trocadas pela rede. Por esta razão, as implementações de operações matriciais em bibliotecas de PAD distribuído usam algoritmos em blocos. Em termos de computação paralela, dir-se-ia que a granularidade é maior através do uso de blocos.

2.2.3.2. Formulação em blocos

Uma vez que os algoritmos em blocos são mais eficientes em termos de acessos à memória (distribuída ou não), é preciso investigar a reformulação dos algoritmos clássicos em termos de produtos matriz - vetor ou, ainda mais eficiente, de produtos matriz - matriz (através da decomposição das matrizes em blocos). A formulação com produtos matriz - vetor é, via de regra, imediata. Assim, por exemplo, a transformação central do algoritmo de fatoração LU (equação 2.1) não é nada mais que:

$$\mathcal{A}_i^{(k)} \leftarrow \mathcal{A}_i^{(k-1)} - L_k \times \mathcal{A}_i^{(k-1)},$$

onde $\mathcal{A}_i^{(k)}$ é um vetor.

A maioria das operações matriciais pode ser reescrita para acessar os dados de uma forma mais compacta. Será visto o exemplo do algoritmo LU, tal como implementado no *benchmark* Linpack, mas a mesma técnica vale para a fatoração de Choleski. Dada a matriz A da seção anterior, sua fatoração LU após n_b fases pode ser escrita da seguinte maneira:

$$PA = \begin{pmatrix} L_{11} & & \\ L_{21} & I & \\ L_{31} & 0 & I \end{pmatrix} \times \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & A'_{22} & A'_{23} \\ & A'_{32} & A'_{33} \end{pmatrix},$$

onde L_{11} e U_{11} são matrizes (ou blocos) de tamanho $n_b \times n_b$ e P é a matriz de permutações devida ao pivoteamento. A' é o resto da matriz A fatorada, sendo A'_{22} seu bloco de tamanho $n_b \times n_b$ constituído das n_b primeiras linhas e colunas, A'_{33} seu bloco de tamanho $(N - n_b) \times (N - n_b)$ constituído das últimas linhas e colunas, e os blocos A'_{23} e A'_{32} de tamanho $n_b \times (N - n_b)$ e $(N - n_b) \times n$ respectivamente, constituídos com os elementos sobrando.

Para calcular uma nova fase da fatoração, calcula-se a próxima coluna de blocos de L e a próxima linha de blocos de U tais que:

$$\begin{pmatrix} I & \\ & P_2 \end{pmatrix} PA = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & I \end{pmatrix} \times \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & A'_{33} \end{pmatrix}.$$

(P_2 é uma matriz de permutação de tamanho $(N - n_b) \times (N - n_b)$.) Ao comparar os dois membros desta equação e identificar os blocos, percebe-se que a primeira etapa é a fatoração da primeira coluna de blocos da sub-matriz corrente, através do cálculo:

$$P_2 \begin{pmatrix} A'_{22} \\ A'_{32} \end{pmatrix} = \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} U_{22}. \tag{2.3}$$

Uma vez que foi feita essa fatoração LU da coluna $(A'_{22}, A'_{32})^T$, pode-se aplicar a matriz de permutação P_2 para efetuar o pivoteamento das linhas no resto da matriz A já parcialmente fatorada:

$$\begin{pmatrix} A'_{23} \\ A'_{33} \end{pmatrix} \leftarrow P_2 \times \begin{pmatrix} A'_{23} \\ A'_{33} \end{pmatrix}, \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \leftarrow P_2 \times \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix}. \tag{2.4}$$

Assim como no caso do algoritmo de fatoração de Gauss, pode-se então calcular o pivô (que será agora um bloco):

$$U_{23} \leftarrow L_{22}^{-1} A'_{23}. \tag{2.5}$$

Por fim, com o bloco pivô U_{23}, pode-se calcular o bloco sobrando A'_{33}:

$$A'_{33} \leftarrow A'_{33} - L_{32} U_{23}. \tag{2.6}$$

Isso finaliza a fase de fatoração, que pode continuar recursivamente com o bloco A'_{33}.

O cálculo (2.3) é uma fatoração LU sobre uma matriz $N \times n_b$. Para implementá-lo, pode-se usar uma chamada recursiva ao mesmo algoritmo por blocos, ou ainda usar uma implementação clássica reescrita para usar produtos matriz-vetor (muitas bibliotecas que implementam esses algoritmos foram historicamente escritas em Fortran-77, linguagem que não permitia a recursividade).

O pivoteamento de (2.4) é apenas uma troca de coeficientes na memória. A implementação é direta.

O cálculo do bloco pivô de (2.5) consiste na resolução de um sistema triangular de tamanho $n_b \times n_b$. Sua implementação em termos de produtos matriz-vetor é direta, uma vez que cada componente do vetor solução é obtido através da combinação linear dos componentes previamente calculados com uma linha da matriz.

Para finalizar, a atualização do resto da matriz em 2.6 consiste em $N - n_b$ produtos de matrizes de tamanho $n_b \times (N - n_b)$.

Afinal, todas as operações de uma fase da fatoração podem ser implementadas através de operações matriz-matriz ou matriz-vetor.

2.2.4. Exemplo de bibliotecas: BLAS, LAPACK

Uma das bibliotecas mais bem conceituadas para o cálculo matricial é a BLAS [TEM 2000] ("Basic Linear Algebra Subroutines"). Distinguem-se as BLAS de nível 1, 2 e 3. No nível 1, são fornecidas todas as operações tipo "vetor-vetor": produto escalar, soma de vetores, multiplicação de um vetor por um escalar, etc. No nível 2, são tratadas as operações que implicam uma matriz e um vetor (produto matriz-vetor, tipicamente), cuja complexidade geralmente é $\mathcal{O}(N^2)$. Por fim, as BLAS de nível 3 disponibilizam rotinas para as operações entre matrizes, *e.g.* produto de duas matrizes, fatoração LU, etc. Essas operações, via de regra, têm o custo $\mathcal{O}(N^3)$, daí o nome. Essas operações de nível 3, além de ter as otimizações acima mencionadas, em geral atuam em cima de $\mathcal{O}(N^2)$ dados em entrada, o que garante um volume de cálculos maior que o número de acessos à memória.

As BLAS são vendidas pelos fabricantes de processadores, assim como compiladores, por serem otimizadas em nível de linguagem de máquina. Existe uma implementação em código livre, em Fortran-77 [NET 2004]. Pode-se compilar essa versão, inclusive com as otimizações usuais de seu compilador preferido, porém o desempenho obtido é sigificativamente menor do que o de BLAS comerciais. A figura 2.1, página 12, apresenta alguns experimentos feitos com um Pentium III 733 MHz. O desempenho máximo esperado seria de 733 MFlops/s, supondo que uma operação em vírgula flutuante fosse executada por ciclo de relógio.

Foram feitas 5 medições por valor de N, e os gráficos incluem o desempenho máximo, mínimo e médio obtido para cada valor de N, com a versão compilada com o PGI. As opções de otimização máximas para a arquitetura alvo foram usadas:

```
-O2 -tp p6 -Minfo -Munroll=n:16 -mp \
    -Mvect=assoc,cachesize:262144 -Ml3f
```

Desta forma, obteve-se no máximo cerca de 300 MFlops/s, ou seja a metade do esperado. Já com a versão da Intel, chegou-se a 630 MFlops/s. Nota-se que em ambos casos, uma vez que o cache fica cheio com a matriz (quando N chega a cerca de 200), o desempenho se estabiliza. (Uma matriz de 183×183 *double* usa 253 Kbytes de memória, o cache L1 sendo no caso de 256 Kbytes.) Repara-se também a influência das linhas de cache no desempenho medido: a medida que uma linha se enche de dados, o desempenho (número de operações por segundo) cresce, até começar uma nova linha. Por isso, encontra-se o fenômeno de "dentes de serra" em ambos gráficos.

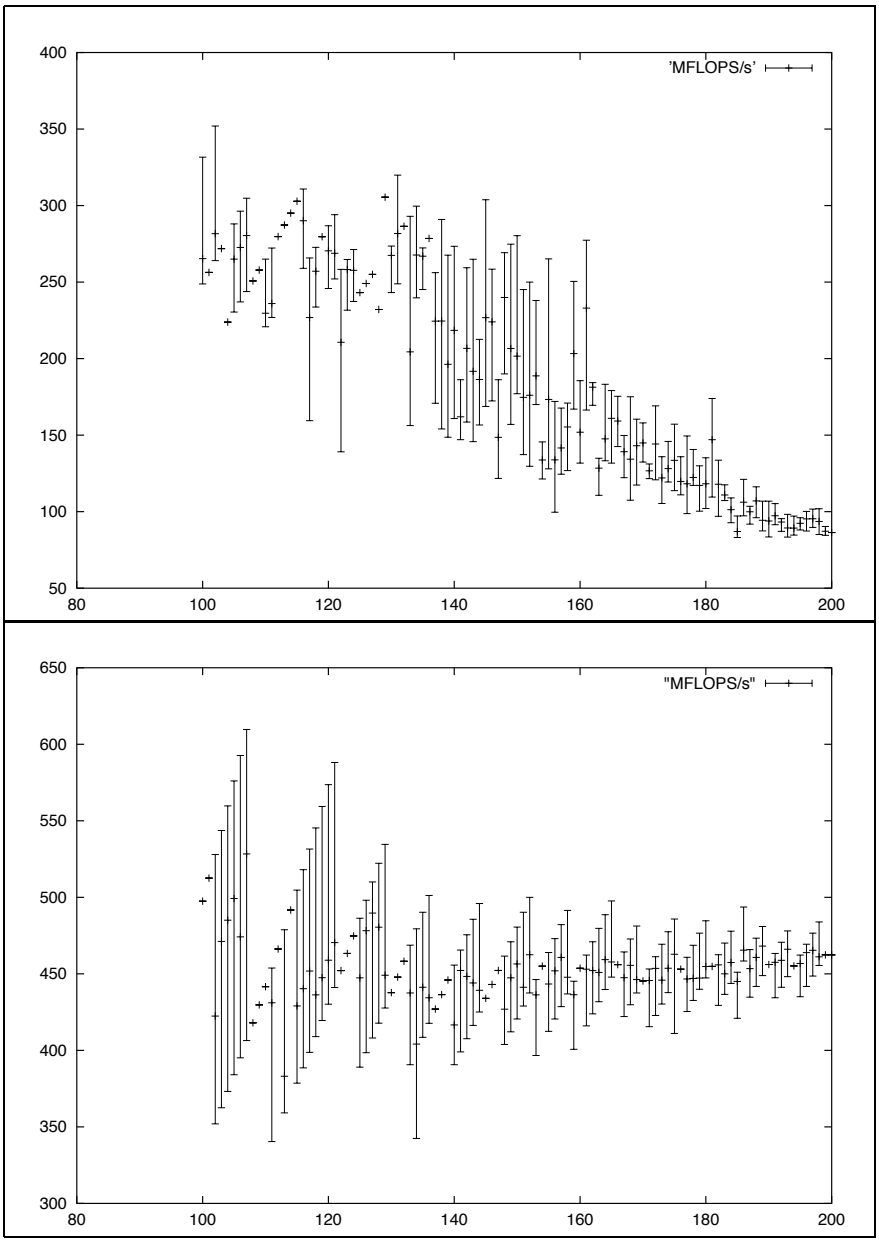

Figura 2.1: MFlops/s medidos com a rotina dgemm (produto matricial) das BLAS, *vs.* tamanho da matriz ($N = 100 \ldots 200$). No gráfico superior, a biblioteca foi compilada com o PGI. No gráfico inferior, foi usada a MKL, versão BLAS da INTEL. Nota-se que as as escalas são diferentes em cada figura.

2.2.5. Exemplo de aplicação: o *benchmark* Linpack do TOP500

A biblioteca Linpack [DEM 89] foi desenvolvida nos anos 70 para resolver sistemas de equações lineares, para vários tipos de matrizes. Ela foi integrada com a biblioteca Eispack, passando a se chamar LAPACK (Linear Algebra Package) no fim dos anos 80, e oferecendo mais algoritmos, tais como o cálculo de auto-vetores (vide seção 2.3.3.), com implementações eficientes para máquinas vetoriais. LAPACK foi projetado em cima das BLAS para garantir a eficiência, através da reformulação de seus algoritmos em versões com blocos, sendo as operações em nível de bloco implementadas nas BLAS. Para máquinas com memória distribuída, existe a versão ScaLAPACK [CHO 92]: ela foi implementada em cima de uma versão paralela das BLAS (P-BLAS), junto com uma camada específica de comunicações dedicadas ao cálculo matricial (BLACS), em cima de MPI.

A partir da biblioteca seqüencial Linpack original foi desenvolvido um teste, chamado Linpack-100, para medir o desempenho de processadores vetoriais. O teste, em 1979, era a fatoração LU de uma matriz 100×100. Com os algoritmos ótimos aqui apresentados, podia se esperar medir quase o desempenho máximo de um processador, na época, com este tamanho de matriz; e foi feita uma comparação do desempenho de 23 processadores no primeiro guia da biblioteca Linpack [DON 79]. Com o crescimento da capacidade das memórias, rapidamente se passou ao Linpack-1000, durante os anos 80, o qual era a fatoração de uma matriz de tamanho 1000.

A evolução seguinte foi paralelizar o teste Linpack para seu uso em máquinas com memória distribuída [NET 2001]. Essa versão passou a se chamar Highly-Parallel Linpack (HPL) e visa o teste da escalabilidade de uma máquina paralela. Para tanto, o único critério imposto pelo *benchmark* é o algoritmo de fatoração LU com complexidade $\frac{2N^3}{3} + 2N^2$. No entanto, pode-se alterar todos os outros parâmetros do teste: tamanho da matriz, algoritmos de comunicações globais, topologia de rede, versão dedicada a máquinas com memória compartilhada, etc. Essa margem deixada ao usuário ajudou este *benchmark* a se impor como a referência na comparação dos supercomputadores, através do TOP500, desde 1993 (vide http://www.top500.org).

Uma implementação padrão é proposta para o HPL, baseada no MPI, e que prevê muitos ajustes deixados ao programador; mas o uso dessa versão não é obrigatória. O algoritmo implementado para a fatoração LU é o pivoteamento de Gauss, com pivô parcial, da matriz particionada em blocos. Além de prover um desempenho muito bom devido ao uso de blocos, os mesmos permitem uma distribuição lógica simples dos cálculos, através de um mapeamento cíclico por blocos da matriz numa grade virtual de processadores. (A implementação em MPI de uma tal topologia é simples.)

O objetivo deste mini-curso não é uma discussão profunda do TOP500. O relatório técnico [RIC 2001] detalha a experimentação que foi feita para classificar este cluster no TOP500 em junho de 2001, e mais detalhes podem se encontrar nele. A figura 2.2 ilustra a escalabilidade do HPL num cluster de 225 nós Pentium-III [MAI 2001].

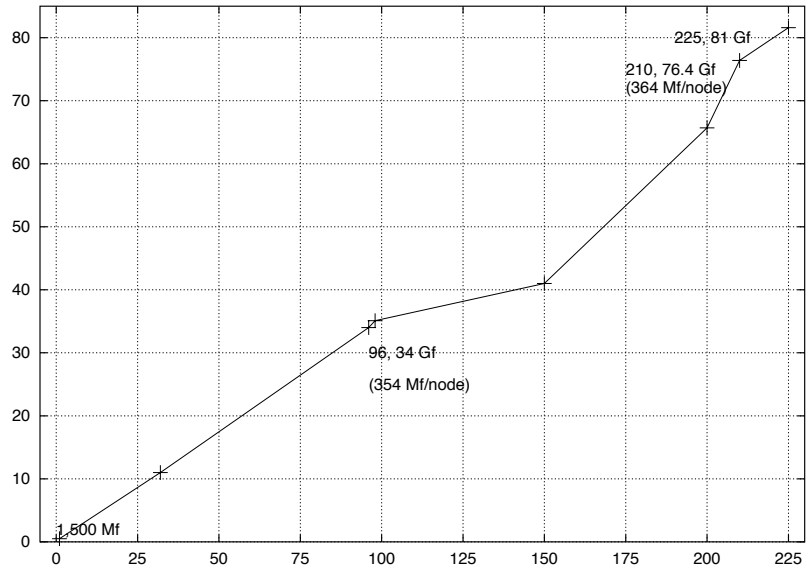

Figura 2.2: Desempenho do *benchmark* Highly-Parallel Linpack (GFlop/s) vs. número de nós no cluster de PIII I-cluster (2001).

2.3. Métodos iterativos

Métodos diretos são muito eficientes e permitem, quando formulados com blocos, atingir um desempenho alto, inclusive em máquinas distribuídas. No entanto, eles necessitam armazenar todos os coeficientes das matrizes; e quando os mesmos são zeros, há um desperdício de espaço memória. Pior, as operações de fatoração (tal como o pivoteamento de Gauss) tendem a "encher" as matrizes, ou seja a tornar coeficientes diferentes de zero. No caso que se usam matrizes esparsas, eventualmente bem maiores do que se fossem cheias, é usual utilizar algoritmos iterativos. A idéia básica é ter métodos que usam apenas produtos matriz - vetor, nos quais a estrutura esparsa da matriz pode ser usada (junto com técnicas de codifica-

ção de tais matrizes) para agilizar o cálculo do produto e limitar a memória usada a ser $\mathcal{O}(N)$.

No caso da resolução de um sistema de equações, serão apresentados os dois algoritmos mais usados, o Gradiente Conjugado e o GMRES. A seção 2.3.3. vai tratar de um outro problema importante com matrizes: o cálculo de auto-valores, que também pode ser feito com algoritmos iterativos.

2.3.1. Gradiente Conjugado

O Gradiente Conjugado (GC) é um dos algoritmos diretos mais antigos usados para resolver um sistema $Ax = b$: ele existe há 50 anos [GOL 89]. Ele cria iterativamente uma sequência de vetores $x^{(i)}$ cujo limite é x quando converge. A convergência é garantida para matrizes simétricas definidas positivas, *i.e.* quando $\forall y, y^T Ay \geq 0$, e que a igualdade vale se e somente se $y = 0$. É um método de "gradiente", no sentido físico que, a cada iteração, calcula um vetor que dá a direção que minimiza a energia do operador $J(y) = \frac{1}{2}y^T Ay - b^T y + c$, c sendo um valor escalar constante. A norma do vetor também é calculada de forma tal que permita "descer" nesta direção até o ponto de energia mínima ao longo dela. As iterações fazem assim convergir, a partir do ponto (vetor) inicial, para o vetor x, que minimize globalmente o operador J. Isso é equivalente a resolver o sistema inicial.

O GC é sintetizado no algoritmo 2. A constante ϵ é o erro numérico que se admite sobre a solução.

Algoritmo 2 Algoritmo do Gradiente Conjugado.

Entradas: o vetor inicial $x^{(0)}$; a matriz A e o vetor b.
Saída: o vetor x solução do sistema $Ax = b$.

$i \leftarrow 0$
$d^{(i)} \leftarrow b - Ax^{(i)}$
$r^{(i)} \leftarrow b - Ax^{(i)}$
Enquanto $r^{(i)T} r^{(i)} > \epsilon$ **Faça**
    $\alpha \leftarrow \frac{r^{(i)T} r^{(i)}}{d^{(i)T} Ad^{(i)}}$
    $x^{(i+1)} \leftarrow x^{(i)} + \alpha d^{(i)}$
    $r^{(i+1)} \leftarrow r^{(i)} - \alpha Ad^{(i)}$
    $\beta \leftarrow \frac{r^{(i+1)T} r^{(i+1)}}{r^{(i)T} r^{(i)}}$
    $d^{(i+1)} \leftarrow r^{(i+1)} + \beta d^{(i)}$
    $i \leftarrow i + 1$
Fim Enquanto
$x \leftarrow x^{(i)}$

Algumas propriedades interessantes do algoritmo são as seguintes: $r^{(i)}$ e $r^{(i+1)}$ são ortogonais. Cada $d^{(i)}$ pertence ao espaço vetorial gerado por $\{d^{(0)}, Ad^{(0)}, \ldots, A^{i-1}d^{(0)}\}$. Este espaço, notado aqui $\mathcal{K}(A)$, se chama espaço de Krylov associado a A e $d^{(0)}$. O fato de os $r^{(i)}$ serem ortogonais assegura que no máximo em N iterações do laço (2) haverá convergência, pois o vetor $r^{(i)}$ chegará a zero. Na prática, a convergência sempre acontece antes deste limite, que pode ser grande, quando se trata de matrizes grandes.

Este algoritmo é especialmente bem projetado para o PAD. De fato, ele precisa efetuar a cada iteração:

- 3 cópias de vetores (linhas 9, 10 e 12). É uma operação tipo BLAS-1;
- 2 produtos escalares ($r^{(i)T}r^{(i)}$, linha 11 e $d^{(i)T}Ad^{(i)}$, linha 8). São operações tipo BLAS-1;
- 1 produto matriz-vetor (linha 8, $Ad^{(i)}$), que é uma operação BLAS-2.

Além de serem implementadas eficientemente pelas BLAS, essas operações são fáceis de paralelizar. As operações BLAS-1 são trivialmente paralelas. O produto matriz-vetor também é, se o vetor pode ser replicado em todos os processadores que calculam (basta então distribuir a matriz por blocos de linhas). No caso que se queira otimizar o espaço e distribuir também o vetor iterado, é preciso de comunicações para transmitir os "pedaços" de vetor em cada processador onde há linhas da matriz.

Por fim, o último interesse do GC é sua economia de espaço em memória. Basta armazenar, de uma iteração para a seguinte, apenas os valores dos vetores $x^{(i)}, r^{(i)}$ e $d^{(i)}$ (além de alguns resultados escalares). É a grande diferença com o algoritmo GMRES apresentado na próxima seção.

2.3.2. GMRES

O segundo algoritmo iterativo muito usado para a resolução de sistemas de equações lineares é o GMRES (Generalized Minimal Residual) [SAA 86]. Ele pode ser visto como uma forma melhorada do GC, uma vez que ele também busca iterativamente um vetor resíduo no espaço de Krylov construído a partir de um vetor inicial $x^{(0)}$, porém fazendo com que eles, além de serem ortogonais, passem a formar uma base ortogonal do espaço explicitamente construída. Desta forma, é necessário armazenar todos os vetores da base, e não mais apenas um por iteração.

A construção da base é feita por ortogonalização dos vetores $r^{(i)}$ ao computá-los, pelo procedimento de Graham-Schmidt. Esta adaptação no caso do espaço de Krylov é conhecida como procedimento de Arnoldi. Ele consiste na fatoração

$Q^{(i)}R^{(i)}$ da matriz $N \times i$ constituída dos vetores $r^{(i)}$. A seguir, a matriz $H^{(i)} = Q^{(i)T}AQ^{(i)}$ é formada (de tamanho $i \times i$), que representa a projeção do operador A no espaço de Krylov. A última etapa consiste em procurar o vetor $y^{(i)}$ deste espaço que minimize o resíduo $\|\beta e_1 - H^{(i)}y\|$, onde e_1 é primeiro vetor da base canônica, e β a norma do resíduo inicial $b - Ax^{(0)}$. Isso se faz pela resolução de um problema de mínimo quadrático, realizada, por exemplo, através de uma fatoração QR de tipo BLAS-3.

A convergência é garantida para matrizes não simétricas, o que torna o GMRES mais genérico do que o GC. No entanto, deve-se armazenar todos os i vetores da base ortogonal $Q^{(i)}$ à medida que i aumenta. Isso representa um espaço de memória $\mathcal{O}(Ni)$, o que proíbe o uso de muitas iterações. Para contornar esta dificuldade, usa-se o algoritmo GMRES com *restart*: decide-se, no início, que serão feitas I iterações (I sendo escolhido conforme o espaço de memória disponível), e se calcula assim uma aproximação $x^{(I)}$ da solução. A partir desta primeira estimativa, repete-se o procedimento, sendo agora $x^{(I)}$ o valor inicial do processo iterativo. Continua-se este algoritmo até a precisão procurada ser atingida.

Assim como no caso do GC, as principais operações do GMRES são produtos matriz - vetor ou cálculos de normas. Ele inclui operações matriciais tipo BLAS-3, mas elas se fazem sobre as matrizes projetadas no espaço de Krylov (*e.g.* a fatoração QR de uma matriz $N \times I$, I sendo pequeno comparado a N). Essas matrizes são densas e pequenas, o que torna o custo aceitável. Afinal, a complexidade se concentra nos produtos matriz - vetor.

Existem várias implementações distribuídas do GMRES, como por exemplo uma gratuita no CERFACS, na França, em
http://www.cerfacs.fr/algor/Softs/GMRES/, que usa Fortran-77 e MPI. Essas implementações usam uma técnica chamada *template*: elas fornecem a estrutura de controle das iterações do algoritmo e todos os cálculos intermediários. Elas deixam apenas ao programador a carga de implementar uma função de produto matriz-vetor, vista como uma caixa preta, que pega como entrada um vetor e retorna um outro. Isso possibilita a otimização, pelo usuário, do produto, levando em conta a estrutura da matriz que só ele conhece. A implementação distribuída se resume então à programação de um produto matriz - vetor distribuído, assim como no caso do GC.

2.3.3. Cálculo de auto-valores

Os algoritmos usados nas outra seções têm todos por propósito resolver um sistema de equações lineares. Um outro problema de alta relevância é o cálculo de auto-valores e vetores de uma matriz A.

2.3.3.1. Auto-valores e auto-vetores de uma matriz

Trata-se de escalares $\lambda_i, i = 1 \ldots, n$ e de vetores $v_i, i = 1, \ldots, n$ tais que $Av_i = \lambda_i$. Fisicamente, os auto-vetores representam as direções no espaço vetorial nas quais o operador linear representado pela matriz mais age; o auto-valor associado a um auto-vetor representa a intensidade da ação do operador nessa direção. O exemplo clássico é o caso de um espaço de duas dimensões definido por dois vetores ortonormais. A ação de um operador linear neste espaço (*i.e.* de uma matriz 2×2) transformará o círculo unidade em uma elipse. Os dois eixos da elipse são as direções dos dois auto-vetores do operador, e a distância entre a origem do referencial e a intersecção elipse/eixos fornece o valor dos auto-valores associados (vide Figura 2.3).

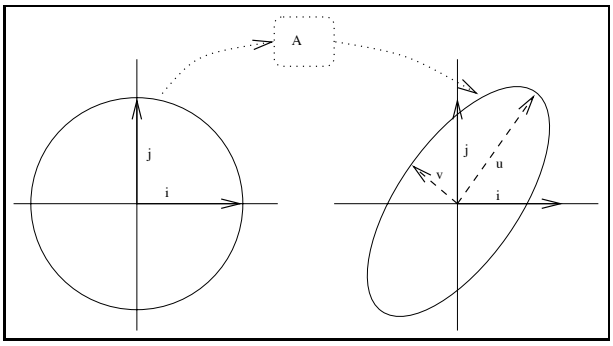

Figura 2.3: Ação de um operador linear A sobre o círculo unidade. Os eixos da elipse resultante dão as direções dos auto-vetores u e v. Os módulos dos mesmos dão os auto-valores.

Uma vez que se conhece uma decomposição LU, por exemplo, da matriz A, os auto-valores podem ser deduzidos automaticamente devido à estrutura triangular das matrizes L e U. Pode-se mostrar simplesmente que os auto-valores são os coeficientes diagonais da matriz L. No caso (freqüente) que a matriz de que se procurem os auto-valores seja simétrica e definida positiva, pode-se usar a fatoração QR para obter nos coeficientes diagonais de R os auto-valores, e na matriz Q uma base ortogonal de auto-vetores associados. O livro [PAR 80] apresenta exaustivamente todas as soluções no caso de tais matrizes.

Uma vez que o auto-valor de maior módulo é associado ao auto-vetor que dá a direção de maior ação da matriz A, um algoritmo imediato para calcular a dupla (auto-vetor, auto-valor) é iterar a ação do operador sobre um vetor inicial aleatório u_0. Cada iteração vai "torcer" o espaço na direção do auto-vetor, até chegar a um espaço reduzido a uma reta que será a direção do auto-vetor procurado. Este algoritmo se chama "algoritmo da potência", pois a aplicação matemática do operador

consiste na multiplicação pela matriz A, o que acaba fazendo com que se calcule $A^k u_0$ à iteração k. Essa seqüência converge para o auto-vetor de maior módulo. O algoritmo da potência é assim o mais simples algoritmo iterativo para calcular o auto-valor de maior módulo.

2.3.3.2. Algoritmos iterativos

Essa noção de usar as potências sucessivas da matriz A é a base da construção do espaço de Krylov, já encontrado nos algoritmos GC e GMRES. A mesma técnica se aplica para o cálculo de auto-valores, segundo um processo extremamente poderoso. Assim como no algoritmo GMRES, se projeta a matriz A no espaço de Krylov. A idéia é que este espaço, gerado por vetores que "tendem" a ter a orientação dos auto-vetores, vai ser uma boa aproximação onde procurá-los. Logo, vai-se calcular os auto-valores e auto-vetores da matriz $Q^T AQ$ projetada em $\mathcal{K}(A)$ (de tamanho $i \times i$), que serão boas estimativas dos auto-valores e vetores da matriz toda. Da mesma forma como no GMRES, se usará um "restart" para começar uma nova fase iterativa com essas estimativas como chute inicial do vetor solução.

Os algoritmos iterativos para o cálculo de auto-valores podem ser encontrados em [SAA 92, SOR 96]. Neste mini-curso, apresenta-se apenas uma versão do algoritmo de Arnoldi no caso que a matriz A é simétrica. As fórmulas de projeções se simplificam neste contexto, para chegar ao algoritmo 3, de Lanczos.

Algoritmo 3 Algoritmo de Lanczos

1: Entradas: um vetor q_1 normalizado ; a matriz A.
2: Saída: um vetor q_1.
3:
4: $\beta_1 \leftarrow 1, q_0 \leftarrow 0$.
5: **Iterações**
6: **Para** $j = 1, 2, \ldots, m$ **Faça**
7: $\quad r_j \leftarrow Aq_j - \beta_j q_{j-1}$
8: $\quad \alpha_j \leftarrow r_j^* q_j$
9: $\quad r_j \leftarrow r_j - \alpha_j q_j$
10: $\quad \beta_{j+1} \leftarrow \|r_j\|_2$
11: $\quad q_{j+1} \leftarrow r_j / \beta_{j+1}$
12: **Fim Para**
13: TMP $\leftarrow A \times Q$ {Q é a matriz formada pelos q_j.}
14: TMP $\leftarrow Q^T \times$ TMP
15: Retorna os auto-valores de TMP e seus auto-vetores.

No algoritmo 3 é apresentada apenas uma fase, a partir da qual seria feito

um restart do algoritmo.

Assim como no GMRES, a distribuição do algoritmo pode ser feita através de uma estrutura *template*, na qual apenas o produto matriz-vetor da linha 7 tem que ser distribuído.

2.3.3.3. Aplicação

A biblioteca Arpack. A principal biblioteca que implementa essas técnicas iterativas de cálculo de auto-valores e vetores se chama ARPACK [LEH 97], cujo nome deriva de ARnoldi PACKage (P-ARPACK para a versão distribuída). Ela está sendo usada para resolver problemas com até dezenas de milhões de variáveis. P-ARPACK foi desenvolvida usando o MPI, mas provê o uso das BLACS como opção alternativa. Através do mecanismo de *template*, o usuário tem que fornecer seu produto matriz-vetor distribuído. O cálculo da norma para o controle da convergência já vem paralelizado. O resto das operações é duplicado em todos os processadores ($e.g.$ as fatorações QR são feitas seqüencialmente).

Aplicação na mecânica quântica. P-ARPACK foi usada em [MAI 2001] numa aplicação em mecânica quântica. A principal equação de conservação da energia, nesta teoria, é uma equação de auto-valores (equação de Schroedinger) num espaço de funções: os operadores são operadores diferenciais, cuja discretização (no caso, por diferenças finitas) faz aparecer uma matriz.

A discretização e a resolução da equação de Schroedinger foram implementadas em Fortran e C, sendo o P-ARPACK usado para a resolução do problema matricial. As execuções foram feitas num Cray-T3D em 1999, e no cluster do INRIA em 2001. A figura 2.4, página 21, mostra o comportamento (tempo vs. número de processadores) da resolução do problema de auto-valores no cluster. Também consta na figura uma representação do auto-vetor calculado: no contexto, trata-se de uma função, que representa a probabilidade de presença de uma partícula, localizada num poço de potencial quântico em forma de T. Esta forma se percebe no plano horizontal do gráfico. Na interseção do T, a probabilidade de encontrar a partícula é bem maior. Se fosse um caso clássico de mecânica de Newton, a probabilidade seria exatamente zero fora do poço, e um em todos os pontos dentro do T.

2.4. Conclusão: cálculo matricial em PAD

Apresentou-se neste texto uma introdução à otimização de alguns algoritmos clássicos de cálculo matricial. Chegou-se à implementação nas principais bi-

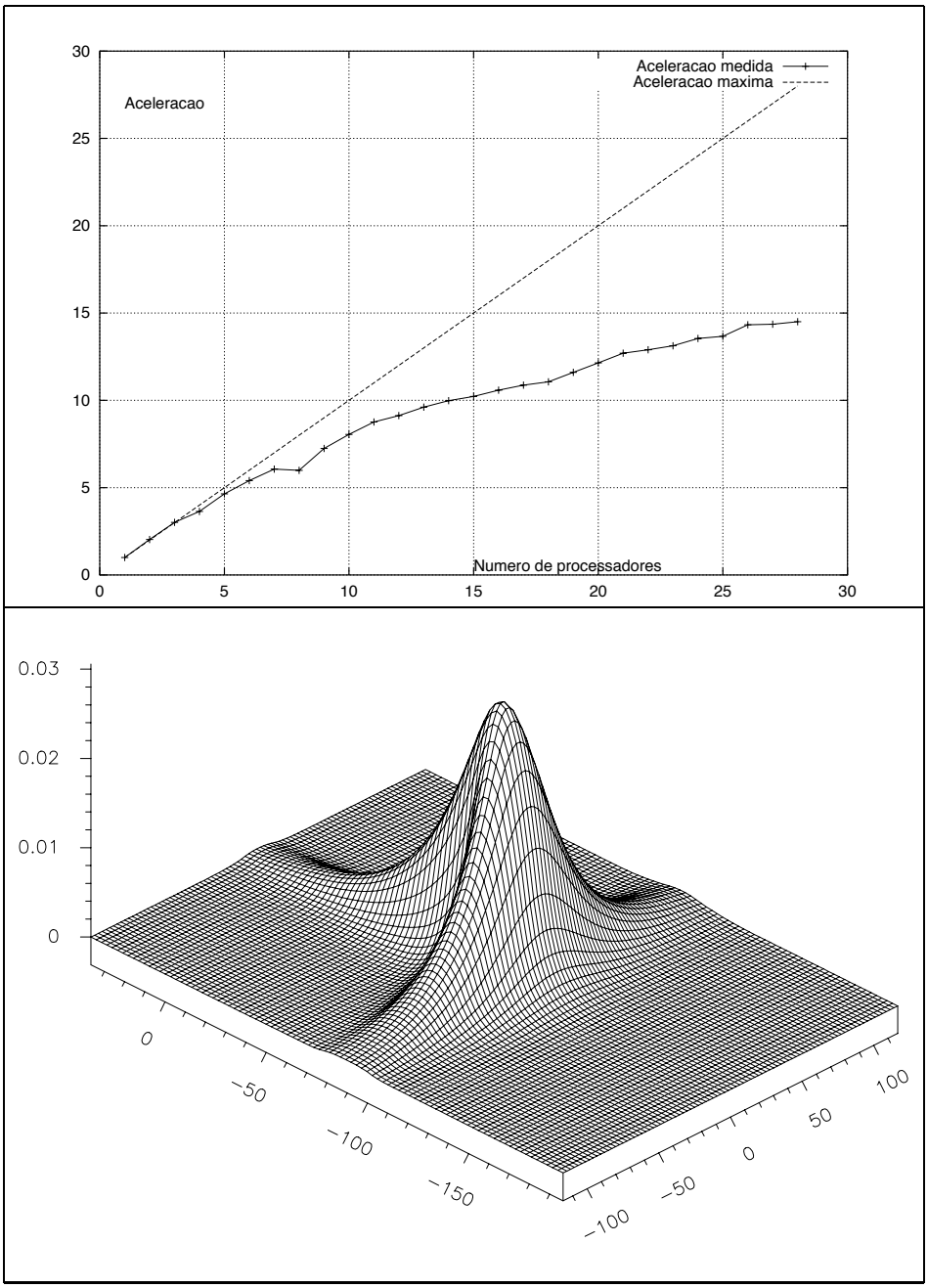

Figura 2.4: Aceleração do cálculo do menor auto-valor da equação de Schroedinger com o P-ARPACK e solução do cálculo.

bliotecas encontradas na área, cujo desempenho, seja seqüencial, seja em versões distribuídas, é reconhecido e foi ilustrado. É importante salientar que o domínio dessas técnicas é um requisito para se trabalhar em PAD com tais algoritmos. O trabalho que tem sido feito, por exemplo no Gradiente Conjugado durante 50 anos, levou a implementações, inclusive paralelas, robustas e altamente otimizadas, que dificilmente são obtidas quando se programa "à mão" o algoritmo nativo.

Existem muitas outras famílias de algoritmos numéricos que se beneficiaram do mesmo de tipo de otimizações. Pode-se citar, entre outros:

- a transformada de Fourier,
- a teoria dos grafos,
- a resolução de equações diferenciais, parciais ou não,
- a interpolação de dados,
- etc.

Este mini-curso tratou apenas de cálculos matriciais pelo fato deles, em geral, aparecerem como última etapa das demais famílias de algoritmos.

Duas conclusões podem ser tiradas deste estudo. A primeira é que dificilmente será obtido um código de alto desempenho sem o cuidadoso estudo preliminar do algoritmo mais bem adaptado ao problema tratado. Não adianta programar com o MPI um algoritmo de pivoteamento de Gauss ijk quando existem versões em blocos muito mais eficientes. A segunda é que este mesmo cuidado de otimização, às vezes focado numa versão distribuída do algoritmo, acaba se confundindo com a programação seqüêncial. Voltando ao exemplo do produto matricial, a noção de localidade espacial (ou de granularidade) acaba sendo igualmente crucial, seja para otimizar os acessos à memória, seja para otimizar a distribuição dos dados. Ou seja: "pensando em paralelo", se obtem um algoritmo eficiente para o caso seqüencial.

Por último, cabe destacar ainda que estes algoritmos clássicos são todos altamente síncronos. Eles são muito eficientes em máquinas paralelas, agregadas ou com memória compartilhada. A tendência atual de cada vez mais distribuição, por exemplo com Grades computacionais, dificilmente é compatível com tais algoritmos, que são, no entanto, os mais usados em PAD. Tipicamente, executa-se em Grades programas trivialmente paralelos. Atualmente, um grande desafio de pesquisa em paralelismo é estudar formas de adaptar esses algoritmos a ambientes heterogêneos e dinâmicos.

Para aprofundar esses algoritmos, a bibliografia fornece mais algumas pistas. Uma boa referência para os métodos numéricos em geral (e não somente matriciais)

é o livro "Numerical Recipes"[2] [PRE 92] que explicita os algoritmos númericos, com um enfoque muito aplicado (por isso o título "Receitas numéricas"), em várias linguagens de programação usual (Fortran, C, C++). Para aprofundar a teoria matricial, o livro fundamental é [GOL 89a]. O livro [TEM 2000], de Jack Dongarra, contem uma boa introdução, orientada a aplicações, aos métodos numéricos, inclusive sobre métodos diretos para matrizes. No caso específico dos métodos iterativos, o livro [BAR 94] é uma boa referência.

Agradecimentos. O autor agradece aos Profs. Dr. Patrícia Augustin Jaques, Maurício Pilla e Ricardo Dorneles, bem como ao Rafael Ennes, pelas correções e sugestões. O autor se responsabiliza por qualquer mesóclise estranha que ainda aparece no texto.

2.5. Bibliografia

[BAR 94] BARRETT, R. et al. **Templates for the solution of linear systems**: building blocks for iterative methods, 2nd edition. Philadelphia, PA: SIAM, 1994.

[CHO 92] CHOI, J. et al. Scalapack: a scalable linear algebra library for distributed memory concurrent computers. In: FOURTH SYMPOSIUM ON THE FRONTIERS OF MASSIVELY PARALLEL COMPUTATION, 1992. **Proceedings...** IEEE, 1992.

[DEM 89] DEMMEL, J. Lapack: a portable linear algebra library for supercomputers. In: IEEE CONTROL SYSTEMS SOCIETY WORKSHOP ON COMPUTER-AIDED CONTROL SYSTEM DESIGN, 1989., 1989. **Proceedings...** IEEE, 1989.

[DON 79] DONGARRA, J. et al. **Linpack user's guide**. [S.l.]: SIAM, 1979.

[NET 2001] DONGARRA, J.; LUSZCZEK, P.; PETITET, A. **The linpack benchmark**: past, present, and future. Available at http://www.netlib.org (dez. 2004).

[GOL 89] GOLUB, G. H.; O'LEARY DIANNE, P. Some history of the conjugate gradient and lanczos methods. **SIAM Review**, v.31, n.1, p.50–102, March 1989.

[2] que pode se obter em pdf, gratuitamente, na URL: http://www.library.cornell.edu/nr/ .

[GOL 89a] GOLUB, G. H.; VAN LOAN, C. F. **Matrix computations**. [S.l.]: John Hopkins University Press, Baltimore, 1989.

[HIG 96] HIGHAM, N. J. **Accuracy and stability of numerical algorithms**. [S.l.]: SIAM Press, 1996.

[LEH 97] LEHOUCQ, R. B.; SORENSEN, D. C.; YANG, C. **Arpack user's guide**: solution of large scale eigenvalue problems with implicitly restarted arnoldi methods. [S.l.: s.n.], 1997.

[MAI 2001] MAILLARD, N. **Calcul haute-performance et mécanique quantique**: analyse des ordonnancements en temps et en mémoire. 2001. Tese (Doutorado em Ciência da Computação) — INPG.

[NET 2004] NETLIB repository. Available at http://www.netlib.org (dez. 2004).

[PAR 80] PARLETT, B. N. **The symmetric eigenvalue problem**. [S.l.]: Prentice-Hall, Inc., 1980.

[PRE 92] PRESS WILLIAM H., e. a. **Numerical recipes in c** : the art of scientific computing. [S.l.]: Cambridge University Press, 1992.

[RIC 2001] RICHARD, B. et al. **I-cluster**: reaching top-500 performance using mainstream hardware. [S.l.]: HP labs, 2001. (HPL-2001-206 200110831).

[SAA 86] SAAD, Y.; SCHULTZ, M. Gmres, a generalized minimal residual algorithm for solving nonsymmetric linear systems. **SIAM Journal on Scientific and Statistical Computing**, v.7, p.856–869, 1986.

[SAA 92] SAAD, Y. **Numerical methods for large eigenvalue problems**. [S.l.]: John Wiley and Sons, New York, 1992.

[SOR 96] SORENSEN, D. C. **Implicitly restarted arnoldi/lanczos methods for large scale eigenvalue calculations**. [S.l.]: Rice University, 1996. (TR-96-40).

[TEM 2000] DONGARRA, J. et al. (Eds.). **Templates and numerical linear algebra**. [S.l.]: Morgan Kaufmann, 2000. p.575–620.