I-621-O

Universidade Federal do Rio Grande do Sul

Instituto de Matemática

Curso de Pós-Graduação em Matemática

Algoritmos Paralelos Numéricos para a Resolução de Sistemas de Equações Lineares

Maria Paula Gonçalves Fachin

Dissertação apresentada como
requisito parcial para a obtenção do Grau de
Mestre em Matemática

Dezembro de 1989

Apresentamos as características de algoritmos para obter a solução de sistemas de equações lineares, utilizando computadores com arquitetura paralela e explorando os recursos de paralelismo neles embutidos. São descritos os modelos de computadores paralelos e os procedimentos para a criação de algoritmos paralelos.

Palavras-chave: Sistemas de equações lineares, algoritmos paralelos.

The characteristics of algorithms to solve systems of linear equations, using parallel computers, are presented. The abstract models of these computers are described, as well as the methods to develop parallel algorithms.

Keywords: Systems of linear equations, Parallel algorithms.

UFRGS INSTITUTO DE INFORMÁTICA

Agradecimentos.

Aos Professores Julio Ruiz Claeyssen e Teresa Tsukazan de Ruiz, pela constante orientação e estímulo durante este estudo.

Sumário

Introdução		v.
Capítulo I		
1.1	Classificações de computadores	1.
	Computador SISD	3.
	"Pipelining"	3.
	Computador SIMD	6.
	"Vector processors"	8.
	"Array computers"	9.
	Computador MISD	9.
	Computador MIMD	10.
1.2	Medidas de eficiência	12.
	Organização dos dados	18.
Capítulo II		
2.1	Princípio de "recursive doubling"	26.
2.2	Princípio da formulação vetorial	31.
2.3	Princípio da iteração vetorial	42.
Capítulo III		
	Sistemas lineares triangulares densos	47.
	Sistemas lineares tridiagonais e tridiagonais por blocos	67.

Considerações finais 83.

Referências bibliográficas 85.

Introdução

Constantemente, no mundo real, surge a necessidade de resolvermos equações diferenciais parciais. Como a maior parte destas equações não possui solução analítica exata, precisamos resolvê-las numericamente através de aproximações. Na discretização feita para resolvermos estas equações, surgem sistemas de equações lineares. Estes são, em geral, de grande porte (maiores do que 1000 x 1000), se desejarmos obter uma solução próxima da exata (ou dentro de certa exatidão admissível).

Ao tentarmos resolver, portanto, uma equação diferencial parcial numericamente, devemos não só nos preocuparmos com a solução propriamente dita, mas também com o tempo necessário para obtê-la. A resolução de tais sistemas de equações lineares demanda um tempo considerável, e muitas vezes torna-se impraticável obter a solução. De alguma maneira, devemos reduzir o tempo de execução do algoritmo utilizado.

Pode-se refinar o algoritmo, procurando otimizá-lo ao máximo, mas a execução do mesmo em um computador seqüencial ainda assim não permitiria obtermos a solução desejada em um tempo razoável. A efetiva redução de tempo necessária é alcançada mediante o uso de algoritmos paralelos e de computadores com arquitetura paralela, os quais permitem que mais de uma operação seja executada em um mesmo período de tempo.

Apresentaremos no capítulo I os conceitos de computadores paralelos existentes, oferecendo uma descrição dos modelos abstratos, segundo a classificação de Flynn, e as medidas de eficiência utilizadas para medir o desempenho de algoritmos paralelos. Serão discutidos também os problemas que podem influir no desempenho destes algoritmos.

No capítulo II, serão discutidos os métodos de desenvolvimento de algoritmos paralelos, bem como algumas de suas aplicações.

No capítulo III, apresentaremos os algoritmos para a resolução de sistemas de equações lineares, para sistemas triangulares densos e tridiagonais.

UFRGS
SISTEMA DE BIBLIOTECAS
BIBLIOTECA SETORIAL DE MATEMÁTICA

CAPITULO I

Neste capítulo, descreveremos tipos de computadores que são utilizados na execução de algoritmos de processamento paralelo para a resolução de sistemas lineares. Consideraremos, também, para melhor compreensão dos métodos numéricos a serem vistos, quais as vantagens e as possíveis dificuldades que podem acontecer quando de sua utilização.

Para que um algoritmo seja útil, devemos ter disponível um processador que possa ler, entender e executar as instruções que compõem este algoritmo. Um computador é um processador de propósito geral que pode executar diferentes tipos de algoritmos.

Os computadores podem ser do tipo seqüencial ou paralelo, segundo os passos de um algoritmo sejam executados seqüencialmente, um de cada vez, ou por um certo número de processadores que permite executar porções de um algoritmo de maneira simultânea.

Existem algumas características que nos permitem fazer uma distinção entre os computadores e optar por um tipo ou outro. São elas: a velocidade de execução de operações, a capacidade de executar o algoritmo de forma confiável, a capacidade de armazenamento de dados e o custo de executar as operações.

1.1 Classificações de Computadores

Existem várias classificações possíveis para as diferentes arquiteturas de computadores (Flynn 1966, Feng 1972, Händler 1977). Usaremos aqui a estabelecida por Flynn, a qual é suficiente aos nossos propósitos. Esta classificação é feita com base nos seguintes parâmetros:

- fluxo ("stream") de instruções
- fluxo de operandos.

Os fluxos de instruções ou de operandos podem ser simples: (SI-"single instruction stream" ou SD-" single data stream") ou múltiplos: (MI-"multiple instruction stream" ou MD-"multiple data stream"). Fazendo a combinação destes quatro atributos e utilizando os símbolos acima, obtemos os seguintes tipos de computadores de acordo com Flynn:

-SISD: (instrução única, dado único). Uma única instrução opera sobre um único dado.

-SIMD: (instrução única, dados múltiplos). Todos os processadores interpretam uma única instrução e executam-na sobre diferentes dados.

-MISD (múltiplas instruções, dado único). É um tipo semelhante ao SISD, pois diferentes instruções operam sobre o mesmo dado.

-MIMD (múltiplas instruções, múltiplos dados). Todos os processadores trabalham com instruções diferentes e em diferentes dados. O número de processadores em geral é pequeno pela complexidade resultante deste tipo de computador.

Nos deteremos aqui no estudo de computadores das classes SIMD e MIMD, que são os que operam sobre diversos dados ao mesmo tempo.

Vejamos com mais detalhes como é a organização de cada um destes tipos de computador, ressaltando que o que será mais utilizado aqui serão os da classe SIMD, pois:

- Os computadores de classe SISD são seqüenciais;
- Não existem implementações de computadores MISD;

- Os da classe MIMD necessitam de um estudo muito mais aprofundado, sendo a implementação de algoritmos bem mais complexa se desejarmos utilizá-lo de maneira eficiente (mantendo os processadores "ocupados" a maior parte do tempo).

COMPUTADOR SISD

Os computadores SISD apresentam uma unidade de controle (responsável pela decodificação da instrução), uma unidade processadora (responsável pelo processamento da instrução) e um módulo de memória. A grande maioria dos computadores SISD utiliza o recurso de "pipelining". As instruções são executadas pela unidade processadora de forma seqüencial, mas as fases de decodificação e execução das instruções podem ser sobrepostas no tempo.

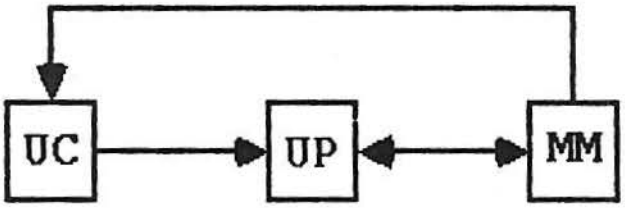


Figura 1 - Modelo de computador SISD

PIPELINING

O recurso de "pipelining" é utilizado para acelerar o processamento das instruções, e é uma maneira econômica de se obter paralelismo temporal (execução concorrente de diferentes tarefas ao longo do tempo). O "pipelining" é inspirado em uma linha de montagem em uma indústria: um produto (a instrução) tem suas diferentes peças (sub-instruções) montadas (executadas) em diferentes estágios.

Assim que um estágio termina de operar sobre sua peça, ela é transferida para o estágio seguinte. O estágio livre está então apto a repetir seus procedimentos sobre uma nova peça. Ao longo do tempo, este processo permite que inúmeros produtos sejam produzidos.

Podemos ter pipelines de instruções, de operações aritméticas e ainda outras subdivisões. Usualmente, uma instrução a ser executada pode ser dividida nos seguintes passos:

a. "Fetch" (busca) da instrução

b. Decodificação da instrução

c. "Fetch" do operando

d. Execução lógica e/ou aritmética

e. Armazenamento do resultado

A própria execução aritmética/lógica pode ser decomposta em outras operações elementares. Por exemplo, podemos ter:

a. admissão dos operandos à ALU

b. controle do sinal

c. comparação de expoentes

d. "shift" para alinhar os operandos

e. adição

f. contagem do número de zeros antes do primeiro dígito significativo

g. "shift" para normalizar o resultado

Num computador não-pipeline todos os passos precisam ser completados antes que a próxima instrução ou operando possa ser avaliada.

Num computador "pipeline", instruções sucessivas (ou, no caso da instrução ser a mesma: operandos sucessivos) são executadas de forma sobreposta.

O seguinte esquema ilustra como ocorre o pipeline para instruções (I_n).

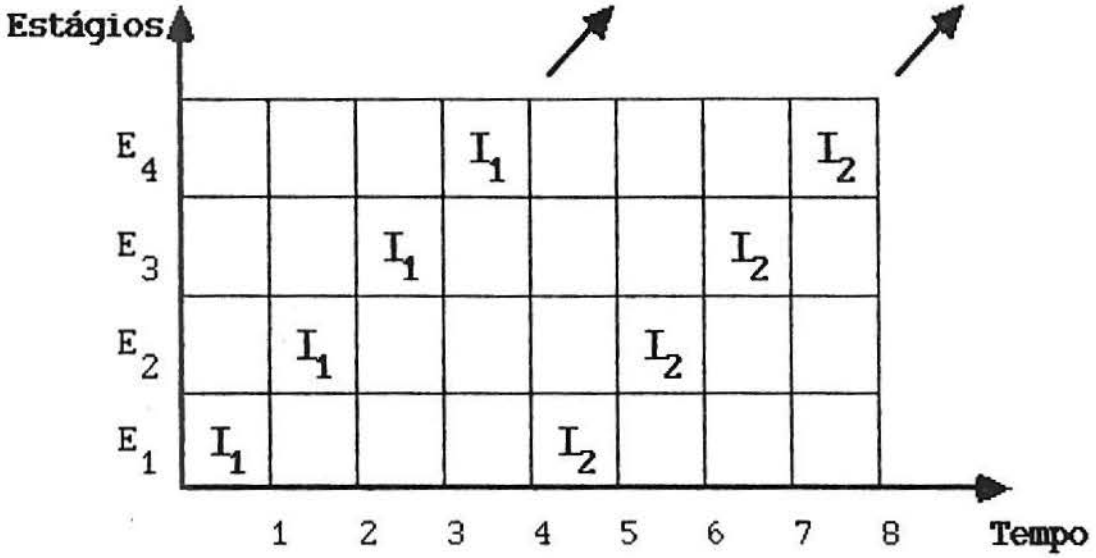


Figura 3 - Execução das instruções em computador não-"pipeline"

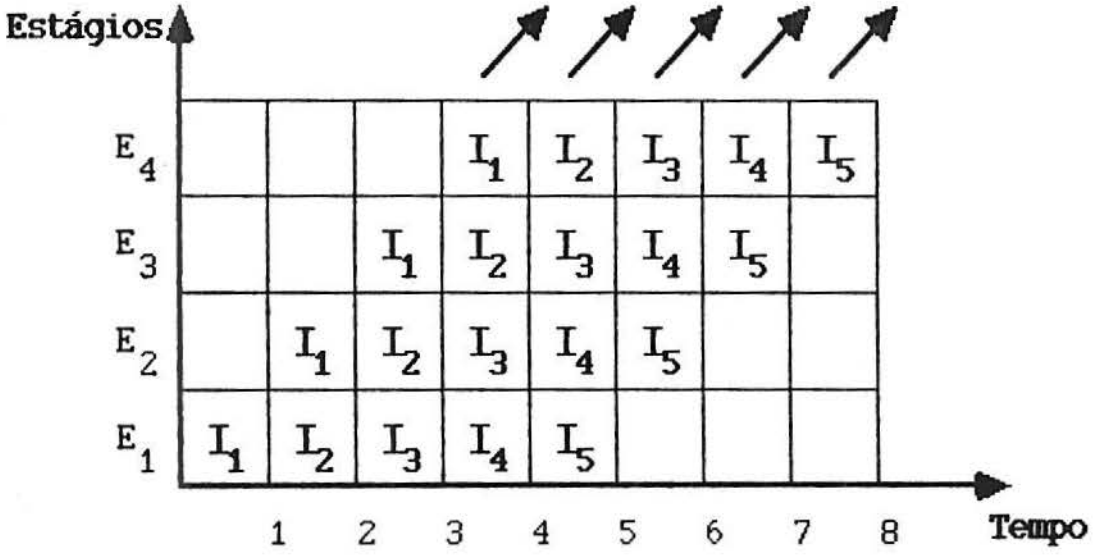


Figura 4 - Execução das instruções em computador "pipeline"

No computador não-pipeline são necessários 4 ciclos de processador (no caso do exemplo citado) para completar uma instrução. Num computador pipeline, uma vez que o pipeline esteja preenchido, teremos um resultado em cada ciclo.

Devido a esta sobreposição de instruções e execuções aritméticas, podemos ver que as máquinas pipeline são melhor aproveitadas se forem utilizadas para executar a mesma operação repetidas vezes. Daí, são computadores adequados para o processamento vetorial, onde seguidamente devemos realizar a mesma operação em diversas componentes.

COMPUTADOR SIMD

São computadores que executam a mesma instrução sobre diversos dados ao mesmo tempo. Podem ser incluídos nesta classe os "vector processors" (CDC Cyber-205, Cray Y-MP/216) e os "array processors" (Bell Laboratories PEPE, Goodyear Aerospace STARAN).

Em geral, a estrutura de um computador SIMD é dada pelo seguinte esquema, no qual temos um certo número de unidades processadoras, todas ligadas a uma única unidade de controle, o que fará com que uma única instrução seja interpretada a cada tempo e daí irradiada a estas unidades que executarão a mesma instrução sobre diferentes dados.

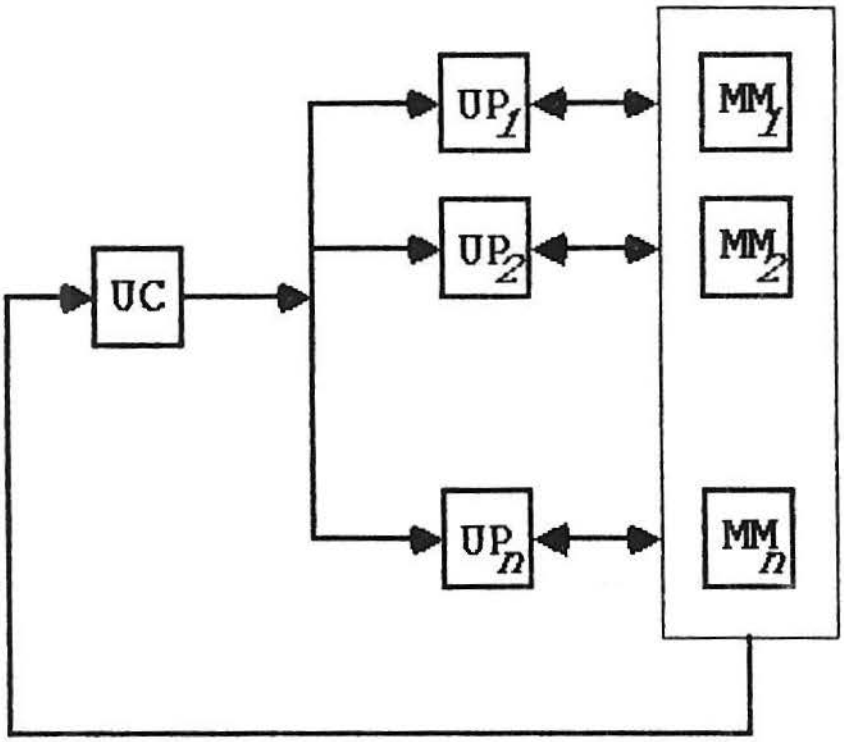


Figura 3 - Modelo de computador SIMD

É variável, entre os autores, a inclusão de computadores "pipeline" nesta classe. Eles não possuem diversos processadores, mas apenas um mais elaborado que é o que chamamos de pipeline. Realmente, se considerarmos que a mesma operação deve ser executada sobre diversos dados ao mesmo tempo, os pipeline escalares não poderiam estar aqui.

No entanto, podemos pensar neles como computadores paralelos, se considerarmos que mais de uma operação (ou sub-operação) é executada simultaneamente. Além disso, algoritmos paralelos são igualmente adequados para todas as máquinas desta classe. Ou seja, podemos estudar algoritmos para computadores SIMD em geral, e aplicar os resultados em array computers, pipeline computers ou outro desta classe, já que são todos computadores voltados ao processamento vetorial (processamento que ocorre sobre um conjunto de dados).

UFRGS
SISTEMA DE BIBLIOTECAS
BIBLIOTECA SETORIAL DE MATEMÁTICA

Para obter uma maior eficiência, talvez seja necessário levarmos em conta as características particulares de cada máquina e fazer algumas modificações no programa, mas o estudo genérico é de grande ajuda para sabermos como efetuar estas modificações.

"Vector processors"

Os "vector processors" são computadores que dispõem de uma ou mais unidades chamadas "vector facilities". Uma "vector facility" é uma ALU cujos operandos são vetores (conjuntos de dados, sejam eles valores numéricos ou lógicos). Os registradores da ALU (chamados de "vector registrers") normalmente armazenam um máximo de 256 elementos. Uma "vector facility" oferece operações aritméticas, tais como adição entre dois vetores, multiplicação entre dois vetores, adição, multiplicação e divisão entre um vetor e um valor escalar, soma dos elementos de um vetor e outras já embutidas.

Os computadores dotados de "vector facilities" apresentam também uma ALU escalar, a qual é responsável por efetuar as operações aritméticas e lógicas entre dados escalares. Exemplos destes computadores são o IBM-3090/VF, DEC 9000/440VP, Cray Y-MP/216 e CDC Cyber-205.

"Array computers"

"Array computers", também chamados de "Array processors" são exemplos típicos de computadores da classe SIMD. Eles possuem unidades lógico-aritméticas múltiplas que operam de forma sincronizada. Estas unidades, chamadas de Elementos de Processamento (" Processing Elements "-PE) são constituídas de uma ALU com registradores e uma memória local. Todas as unidades executam a mesma função ao mesmo tempo e não possuem capacidade de decodificar instruções, necessitando de uma unidade de controle que se encarregue das tarefas de "fetch" e decodificação da instrução. Tal unidade de controle é responsável por informar aos PEs qual instrução devem executar.

Os "array processors" são equipamentos que devem ser ligados a um "front end" (usualmente, um computador do tipo SISD), uma vez que eles não dispõem de recursos de I/O e, normalmente, apresentam uma capacidade reduzida de memória. Cabe ao "front end", portanto, enviar à unidade de controle do "array processor" as instruções que devem ser executadas neste e receber os resultados gerados no "array processor", os quais são, então, armazenados em algum dispositivo do "front end".

COMPUTADOR MISD

Os computadores MISD são compostos de múltiplas unidades processadoras, organizadas como em um linha de montagem ("pipe"), cada qual executando diferentes instruções sobre o mesmo conjunto de dados ou seus derivados. Os resultados gerados por um processador passam a ser os dados sobre os quais o próximo processador do "pipe" irá executar.

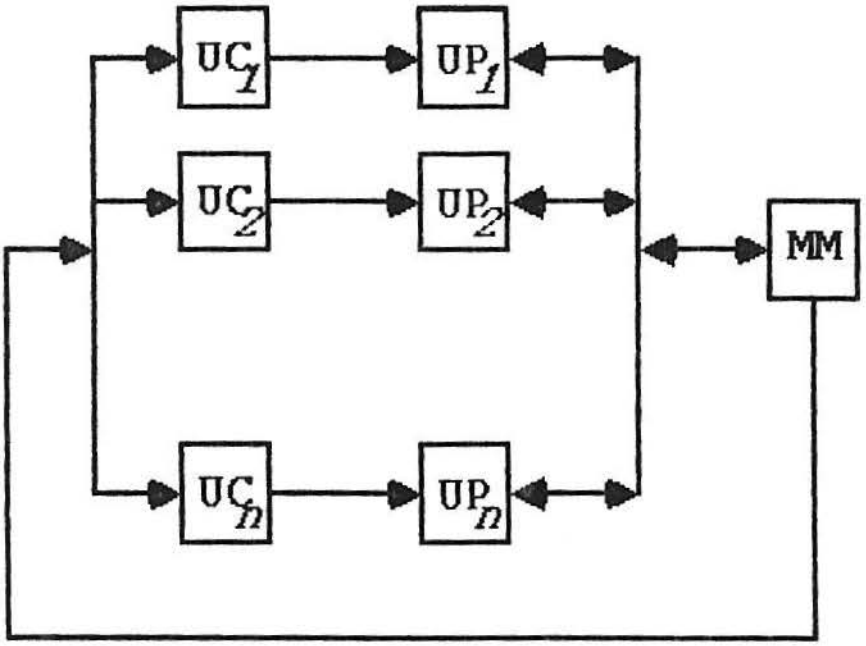


Figura 4 - Modelo de computador MISD

Esta estrutura tem recebido muito pouca atenção e tem sido classificada como impraticável por alguns. Alguns autores argumentam que os computadores MISD são equivalentes aos SISD. Não existem computadores MISD no mercado.

Computadores MIMD

São máquinas que possuem processadores independentes, cada um com sua própria unidade de controle. Podem executar diferentes operações sobre diferentes dados ao mesmo tempo. Os processadores dividem o acesso a módulos de memória comuns, a canais de I/O e outros periféricos. Pela complexidade de sua organização, os computadores desta classe possuem até agora, poucos processadores em relação aos da classe SIMD: 4 e 6 usualmente. Exceção neste caso é a "Connection Machine", com 65536 processadores.

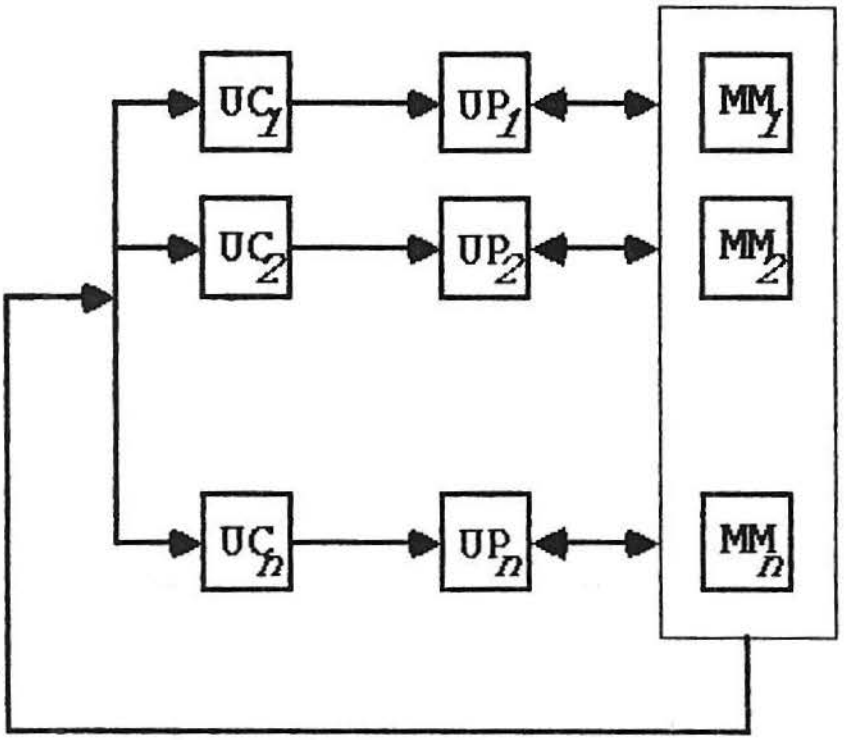


Figura 5 - Modelo de computador MIMD

O desenvolvimento de algoritmos para implementação neste tipo de máquina também é bastante complexo, exigindo grande cuidado na manipulação de dados e no uso dos processadores para não deixá-los ociosos por muito tempo.

UFRGS
SISTEMA DE BIBLIOTECAS
BIBLIOTECA SETORIAL DE MATEMÁTICA

1.2 Medidas de Eficiência

Quando pensamos em computadores paralelos que possuem P processadores, temos a idéia de que os programas a serem executados neles rodarão P vezes mais rápido do que se fossem executados em computadores seqüenciais que possuem apenas um processador. No entanto, vemos que isso não acontece e a velocidade é realmente bem menor.

Entre outros motivos,isto pode ocorrer por problemas de comunicação e transferência de dados entre memória e processadores- o que ocasiona que os processadores fiquem parados esperando informações- e também porque, em geral, não podemos implementar diretamente um programa seqüencial em computadores com mais de um processador.

Por vezes, a adaptação pode ocorrer diretamente, mas na maior parte dos casos isto não ocorre, podendo, além disso, serem algoritmos que não exploram eficientemente a concorrência natural existente no problema.

Devemos, então, procurar medir a eficiência com a qual um problema é resolvido de alguma maneira. Pelo comentário feito acima, vemos que podemos esperar, no máximo, que um programa rode P vezes mais rápido num computador com P processadores do que num que possua um único processador.

Uma medida comumente usada é a razão de "speed-up" (aceleração) definida da seguinte forma:

$$\text{Razão de "speed-up"} = \frac{\text{Tempo de cálculo num computador serial}}{\text{Tempo de cálculo num computador paralelo}} = \frac{T1}{Tp}$$

Para obtermos uma comparação mais justa entre os algoritmos para um determinado problema, utilizamos os melhores algoritmos para cada caso. Ou seja, são aqueles algoritmos que levam o menor tempo para serem executados. Por esta expressão dada para a "speed-up", pode-se verificar o mesmo que já foi dito acima: ela será sempre menor ou, no máximo, igual a P (considerando um computador com P processadores).

$$S = \frac{T1}{Tp} \leq \frac{T1}{T1/P} = P \qquad (\text{pois } Tp \geq T1/P)$$

Esta expressão nos diz quão mais rápido um problema pode ser resolvido num computador paralelo do que num computador seqüencial. Como uma speed-up igual a P é raramente alcançada, temos que um problema que tenha uma speed-up da ordem de kP, onde k é próximo de 1, mas estritamente menor do que 1 pode ser considerado um problema adequado à resolução em computadores paralelos.

Se a "speed-up" de um determinado problema não for proporcional a P e crescer como alguma função menor, o problema não é plenamente adequado para a resolução em paralelo. Temos, então, que quanto mais próxima de função f(P) = P estiver a expressão da speed-up, melhor ela será. Ou seja, as melhores speed-up serão aquelas cuja função for linear em P. Elas são alcançadas em problemas que possuem uma natureza iterativa, tais como sistemas de equações lineares e outros vetoriais-matriciais.

Razões da forma $kP/\log_2 P$ não são tão boas, mas ainda assim , os problemas que tem estas expressões como suas "speed-up", são problemas adequados para computadores paralelos. Problemas que tenham uma speed-up da ordem de $k\log_2 P$ são problemas não muito adequados à

resolução em paralelo, pois quando dobramos o número de processadores, o ganho na velocidade é muito pequeno, sendo então mais adequados para computadores seqüenciais ou com pequeno paralelismo [Stone 1973a]. Para termos uma medida de quão bem este computador paralelo está sendo utilizado temos a seguinte razão chamada de razão de eficiência.

Razão de Eficiência : $E = \frac{S}{P}$

Como $S \leq P$, temos que $E \leq 1$ e em geral, como S é estritamente menor do que P ,teremos que E será um número próximo de 1, mas estritamente menor do que 1. Este índice nos diz se a máquina está sendo bem ocupada ou não. Quanto mais próximo de 1, mais bem utilizados estão sendo os processadores, mais "ocupados" eles estão.

Em geral, é difícil sabermos qual é este tempo de execução num computador paralelo. Para podermos ter esta medida, utilizamos, então, a noção de unidades de tempo ("time unit steps") e de forma idealizada supomos que exatamente uma operação aritmética pode ser executada em cada unidade de tempo.

Podemos, ainda definir a seguinte medida que serve para compararmos dois algoritmos paralelos para um dado problema:

$F_p = \frac{S}{C_p}$, onde $C_p = P\,T_p$ mede o "custo" do algoritmo

Temos que $F_p = S/\,P\,T_p = E/\,T_p < 1/\,T_p < 1$. Portanto, F_p é uma medida tanto de aceleração (speed-up) como de eficiência e daí, um algoritmo paralelo pode ser considerado mais eficiente se maximizar F_p.

Vejamos num exemplo simples como estes índices se comportam. Suponhamos que queremos efetuar uma soma de dezesseis números. Num computador seqüencial, que possui um único processador precisamos fazer quinze adições. Utilizando o tempo para efetuarmos uma adição como unidade de tempo, temos que $T_1 = 15$. Se tivermos 2 processadores, podemos particionar a soma inicial em duas somas,cada uma a ser executada em um processador ao mesmo tempo :

$$A_1 = a_1 + a_2 + \ldots + a_8 \qquad A_2 = a_9 + a_{10} + \ldots + a_{16}$$

e em seguida efetuar

$$A_1 + A_2.$$

No 1º estágio necessitamos de 7 unidades de tempo e no 2º mais uma, dando um total de 8 unidades de tempo para efetuarmos esta soma utilizando 2 processadores. Temos, portanto,

$$T_2 = 8$$

$$S = T_1 / T_2 = 15/8 = 1{,}875$$

$$E = S / 2 = 15 / 16 = 0{,}9375$$

Se tivermos 3 processadores podemos partir a soma em 3 partes:

$$A_1 = a_1 + a_2 + \ldots + a_5 \qquad A_2 = a_6 + a_7 + \ldots + a_{10} \qquad A_3 = a_{11} + a_{12} + \ldots + a_{15}$$

E em seguida efetuar:

$$B_1 = A_1 + A_2 \qquad B_2 = A_3 + a_{16}$$

$$C_1 = B_1 + B_2 = A$$

No 1º passo, são necessárias 4 unidades de tempo, no 2º e no 3º , uma unidade de tempo em cada um. São necessárias, no total, 6 unidades de tempo e temos, portanto, os seguintes índices:

$T_3 = 6$

$S = T_1 / T_3 = 15/6 = 2,5$

$E = S / 3 = 15 / 18 \cong 0,8333$

Para o caso de termos 4 e 8 processadores, podemos repartir a soma em 4 e 8 sub-somas respectivamente, ficando uma a cargo de cada processador. A soma é, então, efetuada segundo os seguintes esquemas:

Para P = 4:

$A_1 = a_1 + a_2 + a_3 + a_4$ $\quad A_2 = a_5 + a_6 + a_7 + a_8$

$A_3 = a_9 + a_{10} + a_{11} + a_{12}$ $\quad A_4 = a_{13} + a_{14} + a_{15} + a_{16}$

E em seguida:

$B_1 = A_1 + A_2$ $\quad B_2 = A_3 + A_4$

$C_1 = B_1 + B_2 = A$

São, portanto, necessárias um total de $T_4 = 3 + 1 + 1 = 5$ unidades de tempo e daí, os índices são os seguintes:

$T_4 = 5$

$S = T_1 / T_4 = 15/5 = 3,0$

$E = S / 4 = 15 / 20 = 0,75$

Para P = 8, as operações são as seguintes:

$A_1 = a_1 + a_2$ $A_2 = a_3 + a_4 ...$ $A_7 = a_{13} + a_{14}$ $A_8 = a_{15} + a_{16}$

$B_1 = A_1 + A_2$ $B_2 = A_3 + A_4$ $B_3 = A_5 + A_6$ $B_4 = A_7 + A_8$

$C_1 = B_1 + B_2$ $C_2 = B_3 + B_4$

$D_1 = C_1 + C_2 = A$

Temos aqui, um total de T8 = 1 + 1 + 1 + 1 = 4 unidades de tempo necessárias para o cálculo de A. Os índices para P = 8 são:

$T_8 = 4$

$S = 15 / 4 = 3,75$

$E = 15 / 32 = 0,4688 \cong 0,47$

Fazendo um quadro comparativo destes valores para podermos examinar melhor a relação entre os índices calculados, temos:

Nº de Proc.	T_p	S	E	C	Fp T_1 = SE
1	15	1	1	15	1
2	8	1,88	0,94	16	1,76
3	6	2,5	0,83	18	2,08
4	5	3	0,75	20	2,25
8	4	3,75	0,47	32	1,76

Ao aumentarmos o número de processadores, verificamos, como seria de se esperar, que S (speed-up) aumenta enquanto que E diminui. F.T1, contudo, atinge um máximo quando P = 4 o que indica que a escolha ótima para o número de processadores nesta aplicação é P = 4. As diferenças, aqui, se devem também ao fato de que os algoritmos não são exatamente os mesmos para os diferentes P vistos. Para uma maior discussão a respeito dos índices de "speed-up" e de eficiência, ver Eager et al. [1989].

Para a análise de algoritmos paralelos é importante que, além destas medidas de eficiência, estudemos, também, a estabilidade do problema e que seja feita uma análise de erros. No entanto, o estudo destes aspectos ainda não é muito explorado, existindo poucos trabalhos a respeito.

Como vimos acima, geralmente os limites mínimos de tempo estabelecidos para a execução de um programa, não são obedecidos. Isto se deve, entre outros motivos, a problemas de comunicação entre os processadores. Este é um dos problemas que temos que nos defrontar quando lidamos com a computação paralela. Existem outros que veremos a seguir.

Organização dos dados

A organização dos dados na memória do computador é um fator muito importante para um bom desempenho na execução de um programa. Como num computador paralelo o acesso de cada processador a certas porções de memória é bastante difícil , dependendo da máquina que

estamos utilizando, devemos ter em mente o que desejamos fazer para organizarmos os dados de maneira eficiente. O problema aqui acontece porque as memórias são compartilhadas entre os processadores (ou unidades lógico-aritméticas) e a capacidade de comunicação entre os processadores é limitada por fatores físicos, tais como o número de caminhos existentes para que exista uma comunicação de n x n processadores, a dissipação de calor e o grau de integração dos circuitos. O acesso aos dados que estão sob "responsabilidade" (na memória local) dos outros processadores não é direto, sendo possível que estes dados percorram um longo caminho até chegarem ao processador desejado. Isto pode ocasionar um grande tempo de espera durante o qual os processadores ficam parados, prejudicando a performance do algoritmo. Por vezes, é necessário mesmo que os dados sejam reorganizados ao longo da execução do programa.

Vejamos, através de um exemplo simples que é o de armazenamento dos elementos de uma matriz, como podem ocorrer problemas em algo que antes não acontecia quando lidávamos com computadores seqüenciais.

Consideremos um computador que possui 3 processadores, cada um possuindo uma certa porção de memória própria. Suponhamos que uma matriz 3x3 é armazenada segundo a sua ordem "natural" de acordo com o esquema seguinte:

proces.	P1	P2	P3
	a_{11}	a_{12}	a_{13}
	a_{21}	a_{22}	a_{23}
	a_{31}	a_{32}	a_{33}

Desta maneira, temos que cada processador terá acesso a uma

das colunas da matriz. O processador P1 terá acesso aos elementos da 1ª coluna: a_{11}, a_{21} e a_{31}, o processador P2 aos elementos da 2ª coluna e P3 aos elementos da 3ª coluna. Sendo assim, quando da execução de um programa, podemos ter acesso simultâneo (cada elemento a cargo de um processador) aos elementos de uma linha, aos elementos das diagonais, mas não aos elementos de uma mesma coluna, pois cada coluna da matriz está restrita a um processador e os outros processadores não tem acesso àquela a qual não estão designados (pelo menos não imediatamente). Armazenar estes elementos na forma da matriz transposta, não soluciona o problema, pois daí não teremos acesso simultâneo aos elementos das linhas da matriz original. Pode-se observar, no entanto, que o seguinte esquema de armazenamento permite processamento paralelo entre as linhas, colunas e diagonais da matriz A.

P1	P2	P3
a_{11}	a_{12}	a_{13}
a_{23}	a_{21}	a_{22}
a_{32}	a_{33}	a_{31}

Por exemplo, o acesso à 1ª coluna é possível, já que a_{11} está disponível em P1, a_{21} disponível em P2 e a_{31} está disponível em P3.

Da mesma forma, podemos acessar os elementos da 2ª coluna simultaneamente já que a12 está disponível em P2, a_{22} em P3 e a_{32} em P1. Os elementos das linhas tem claramente acesso simultâneo e os da diagonal estão disponíveis, respectivamente, aos processadores P1, P3 e

P2.

Este processo de armazenamento pode ser generalizado, de certa forma, para matrizes com dimensões maiores, de modo que possa haver acesso simultâneo a linhas, colunas e diagonal. Tentando repetir este esquema para uma matriz de dimensão 4, em uma máquina com 4 processadores verificamos que temos acesso a linhas e colunas mas não a diagonais.

P1	P2	P3	P4
a11	a12	a13	a14
a24	a21	a22	a23
a33	a34	a31	a32
a42	a43	a44	a41

Pode-se mostrar que não há maneira de armazenarmos os elementos de uma matriz nxn, quando n for par e o número de unidades de memória (neste caso, as colunas) for igual a n, sem que haja conflito no acesso a linhas, colunas ou diagonais.

Se, no entanto, houver um número maior de unidades de armazenamento do que a dimensão da matriz, é possível efetuarmos processamento paralelo em qualquer uma destas linhas citadas. Seja m, o número de unidades de memória disponíveis. Se m for igual a 5, podemos armazenar a matriz 4x4 anterior da seguinte forma:

P1	P2	P3	P4	P5
a_{11}	a_{12}	a_{13}	a_{14}	
a_{24}		a_{21}	a_{22}	a_{23}
a_{32}	a_{33}	a_{34}		a_{31}
	a_{41}	a_{42}	a_{43}	a_{44}

O esquema acima obedece ao esquema geral que é o seguinte:

a. Fixa-se a_{11} em sua posição "natural";

b. Elementos que estão na mesma coluna da matriz original e em linhas sucessivas são armazenados duas unidades à direita de seu predecessor (por ex., como a_{11} está com P1, a_{21} ficará com P3, a_{31} com P5, a_{11} com P2- ver último item);

c. Elementos que estão na mesma linha original, continuam na mesma linha, colocados sempre uma unidade à direita do anterior (como normalmente: a_{22} segue a_{21} , a_{23} segue a_{22} e assim por diante);

d. Uma vez esgotadas as unidades de memória, o processo é recomeçado com a porção de memória referente ao 1º processador (se a_{31} está em P5, duas unidades à direita começando em P1, será P2; como a_{22} está em P4, a_{32} fica em P1).

Formalmente a nova posição (i',j') é dada pela seguinte transformação:

$i' = i$

$j' = j + 2(i - 1) \mod 5$, onde (i,j) é a posição inicial e 5 é o número de unidades de memória.

Este tipo de armazenamento é conhecido como "skewed storage". Kuck [1974] descreve esta maneira de armazenar elementos de uma matriz de modo a permitir que haja acesso a linhas, colunas e diagonal.

Um exemplo simples do problema de armazenamento de uma matriz é quando precisamos calcular o produto de duas matrizes A e B. Para podermos efetuar este cálculo necessitamos calcular o produto

vetorial entre linhas de A e colunas de B. Se estivermos utilizando um computador cuja organização é semelhante ao exemplo acima, o tipo de armazenamento dado acima resolve o problema. Se não for necessário o cálculo de BA, talvez a simples transposição da matriz B seja suficiente. Em todo o caso, o importante aqui, é notarmos que algo que não nos preocupava quando tratávamos de processamento sequencial agora pode atingir uma importância considerável.

Outros aspectos que devem ser levados em consideração quando estamos tratando de construir ou implementar algoritmos paralelos são os seguintes:

a. Nem sempre algoritmos seqüenciais eficientes se tornam algoritmos paralelos eficientes. Às vezes o inverso acontece e temos algoritmos que não se mostravam muito bons em seu formato seqüencial se transformarem em algoritmos realmente eficientes quando tratados em processamento paralelo. Um exemplo deste caso são os algoritmos de ordenamento [Traub 1973].

b. Muitas vezes algoritmos que à primeira vista parecem ser inerentemente seqüenciais podem, na verdade, estar escondendo um grande paralelismo que aparece após algumas pequenas manipulações no algoritmo original. Vejamos mais claramente o que ocorre neste caso através de um exemplo:

Consideremos o problema de recorrência :

$x_1 = a_1$

$x_i = a_i x_{i-1}$, $i \geq 2$

Queremos calcular x_i para i = 2,3,...,n

A primeira idéia que temos é calcular estes produtos seqüencialmente:

$x_2 = a_2 x_1 = a_2 a_1$

$x_3 = a_3 x_2 = a_3 a_2 a_1$

$x_4 = a_4 x_3 = a_4 a_3 a_2 a_1$

...

$x_n = a_n x_{n-1} = a_n \ldots a_2 a_1$

Observando a 3ª coluna vemos que cada x_i é dado pelo produto $x_i = \pi^{1}_{j=i} a_i$. Ao invés de calcularmos estes produtos um após o outro como indicado, podemos calcular os produtos de dois em dois como mostra o esquema abaixo para x_8.

1º passo: calcula-se os produtos $a_8 a_7$ $a_6 a_5$ $a_4 a_3$ $a_2 a_1$

2º passo: utilizando os produtos calculados anteriormente, calculamos $a_8 a_7 a_6 a_5$ e $a_4 a_3 a_2 a_1$

3º passo: utilizando os produtos do passo anterior, calculamos $a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1$

Este tipo de transformação é um modo comum de obtermos algoritmos paralelos a partir de algoritmos seqüenciais. Veremos mais adiante outros exemplos desta técnica.

c. Programas paralelos podem exibir comportamentos numéricos diferentes de suas versões seqüenciais. Exemplos típicos desses programas são os de cálculo de zeros de funções.

Por exemplo, o algoritmo de Newton (em sua versão seqüencial),

$$x_{i+1} = x_i - f(x_i) / f'(x_i)$$

tem sua versão paralela expressa por

$$x_{i+1} = x_i - f(x_i) / f'(x_j), \quad j \leq i,$$

onde j, i e i+1 indicam iterações. O paralelismo é obtido através de dois processos independentes: um (A) avalia a derivada da função em um ponto x_j, e o outro (B) calcula o novo valor de x (x_{i+1}), o qual é transmitido ao processo A.

A : { df:= f'(x) }

B : { x := x + f(x) / df }

Como o processo A é mais rápido do que B (pois o número de operações efetuadas em B é maior do que em A), o resultado é que B utiliza valores da derivada calculados sobre pontos x gerados em iterações antigas (daí $j \leq i$). Isto causa erros de arredondamento que influem no resultado final, e que não ocorrem no algoritmo seqüencial. A existência destes erros não implica em se dizer que o algoritmo paralelo oferece um resultado de qualidade inferior, em relação ao algoritmo seqüencial; muitas vezes, o resultado fornecido pelo primeiro é melhor, principalmente se a função f apresenta oscilações bruscas. [Bojanczyk 1984, da Cunha 1989].

CAPITULO II

PRINCIPIOS BASICOS NA CONSTRUÇÃO DE ALGORITMOS PARALELOS

A tendência inicial na construção de algoritmos paralelos é a de ajustar os algoritmos seqüenciais conhecidos de modo que possa ser executado de modo eficiente em um computador paralelo. Muitas vezes, os algoritmos tem embutidos nos seus passos o paralelismo, podendo ser desdobrados e executados de maneira independente ou reformulados de maneira tal que apresentem um paralelismo iterativo.

2.1 Princípio de "Recursive Doubling"

Uma maneira de obtermos algoritmos paralelos é a mesma idéia do exemplo dado anteriormente, para mostrarmos que algoritmos aparentemente seqüenciais tem, na realidade, um paralelismo embutido. Esta técnica é conhecida como "recursive doubling" (duplicação recursiva) e consiste em dividirmos o problema inicial em duas partes independentes, de modo que cada uma possua a mesma complexidade da outra. Além disso, estas duas partes devem possuir o mesmo tipo de operações de forma que possam ser executadas simultaneamente em um computador do tipo SIMD.

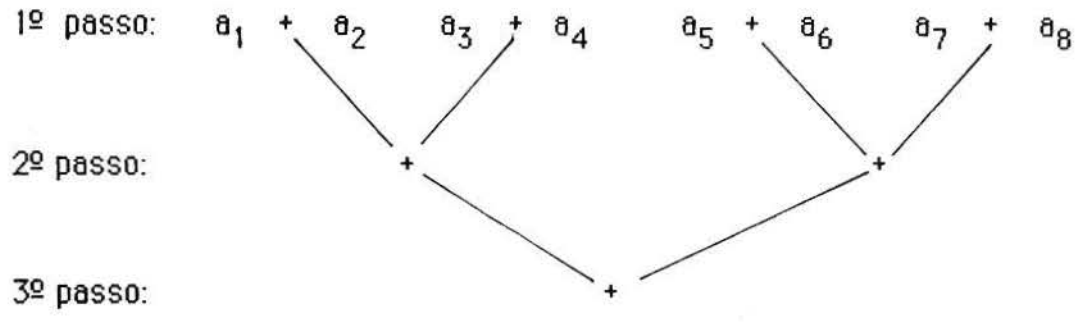
Estas duas partes são, por sua vez, "quebradas" em problemas menores que em seguida são também partidos e assim recursivamente, até que não possamos mais dividir o problema por limitações no número de processadores, porque o problema em si não pode mais ser particionado ou por algum outro motivo. Este processo é utilizado sobretudo quando temos uma seqüência de operações associativas que podem, como o nome, diz ser associadas livremente e, com isto serem "repartidas" entre os processadores.

Exemplos de operações deste tipo são a adição, multiplicação (inclusive de matrizes) e a obtenção dos valores máximo e mínimo de um conjunto de valores entre outras. Como este é um princípio geral para a construção de algoritmos, vamos repeti-lo aqui de forma mais esquemática.

Considerando a operação associativa como sendo uma adição entre 8 números, vejamos como ficam os passos a serem executados. (Se o número de parcelas não for potência de 2, devemos efetuar certas adaptações imediatas ao algoritmo-ver abaixo).

Em um computador seqüencial uma soma de 8 números envolve 7 adições e portanto 7 passos (7 unidades de tempo se considerarmos que cada adição leva uma unidade de tempo para ser executada) já que as operações são feitas uma após a outra. Em um computador paralelo este tempo pode ser bastante reduzido (ignorando os tempos de comunicação, se necessário).

a. Seja a soma $\sum_{0}^{8} a_i$. Utilizando 4 processadores podemos fazer:

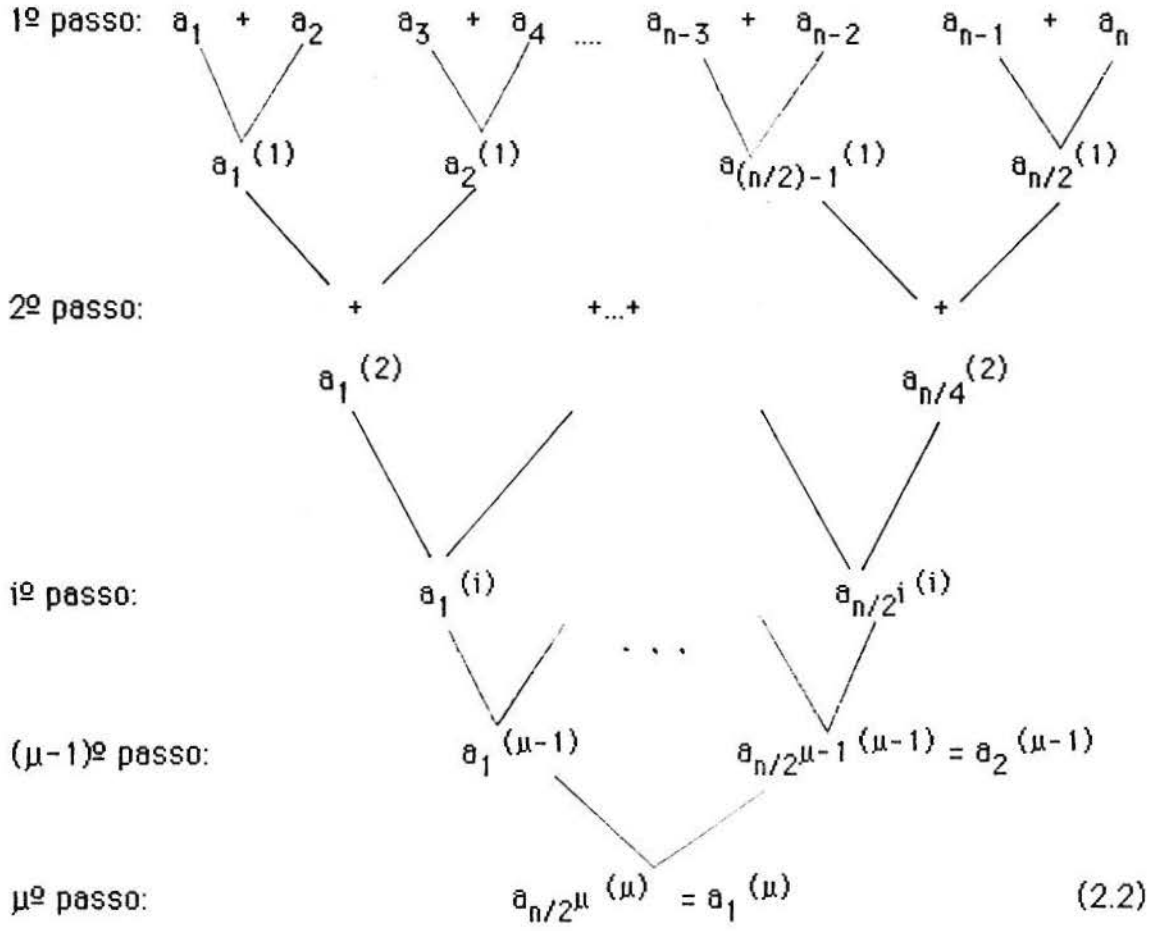


(2.1)

Aqui, obviamente o número de operações (7) foi o mesmo obtido para processamento em um computador seqüencial. No entanto, o número total de operações não é tão importante, pois podemos executar várias delas simultaneamente.

O número de operações por passo é relevante, mas se considerarmos que o mesmo grupo de operações é executado a cada passo tanto num tipo de processamento quanto no outro, basta considerarmos o número de passos num caso e outro para termos de comparação. O mais importante neste caso é o número de passos necessários ao cálculo da expressão, pois podemos considerar cada passo como sendo executado em uma unidade de tempo. Neste caso, o número de passos foi 3, ou seja, $T_4 = 3$. Temos, portanto que $S_4 = T_1 / T_4 = 7 / 3$ e $E_4 = S_4 / 4 = 7 / 12$

Vejamos como fica, em geral, para o número de parcelas igual a n. Para facilitar a escrita, consideremos n como uma potência de 2, por exemplo, $n = 2^{\mu}$. Utilizando n /2 processadores podemos calcular.



(2.2)

Foram necessários μ passos, pois no último passo deveríamos

ter somente a_1 e pela fórmula do termo geral temos que este índice (1) deve ser igual a $n / 2^k$, onde k é o passo que estamos considerando. Devemos ter, então, $1 = n / 2^k$, ou seja, $n = 2^k$ e portanto, $k = \mu = \log_2 n$.

No caso em que n não é potência de 2 devemos fazer algumas adaptações diretas no algoritmo. Não podemos mais considerar que sempre teremos um número par de parcelas em cada passo e por isto, para termos um algoritmo semelhante ao anterior, onde particionamos a expressão o máximo possível, devemos ter $\lceil n/2 \rceil$ processadores (a expressão $\lceil x \rceil$ significa que estamos tomando o menor inteiro maior ou igual a x). Além disso, o número de passos calculado da maneira anterior pode não dar inteiro e por este motivo o número de passos será dado aqui por $\lceil \log_2 n \rceil$. Pela análise feita acima, como utilizamos a partição máxima do problema, notamos que o menor número de passos para resolver um problema deste tipo será dado por $\lceil \log_2 n \rceil$. Qualquer outro tipo de implementação necessitará de um número de passos $T \geq \lceil \log_2 n \rceil$.

Outra solução para o caso em que o número de elementos não é uma potência de 2 consiste em se agregar tantos zeros quantos faltarem, até que o número de elementos seja uma potência de 2.

b. Um exemplo mais interessante de aplicação do mesmo princípio é quando temos certos problemas de recorrência. Aparentemente nestes problemas não encontramos operações associativas, mas após alguma modificação na formulação do problema ele pode ser tratado como o exemplo anterior.

Sejam dados a_i , b_i para i = 1,...n

$x_0 = 1$

$x_1 = b_1$

$x_i = b_1 x_{i-1} + a_i x_{i-2}$

Queremos calcular x_i para i = 2,...n.

Com esta representação do problema não podemos aplicar a técnica de "recursive doubling" diretamente, pois não há uma operação associativa claramente mostrada aí. No entanto, podemos escrever o problema acima na forma vetorial-matricial, sendo possível, então, usar este método.

As operações acima se transformam em:

$$X_i = A_i X_{i-1} = \begin{bmatrix} b_i & a_i \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ x_{i-2} \end{bmatrix} \qquad (2.3)$$

Colocado o problema no formato acima podemos calcular X_n em $\log_2 n$ passos, onde em cada passo calculamos o produto de matrizes 2 x 2 (pois $X_n = A_n X_{n-1} = A_n A_{n-1} X_{n-2} = \ldots = A_n A_{n-1} \ldots A_2 X_1$). A speed-up neste caso é da ordem de $n / \log_2 n$ o que torna o problema adequado para resolução em paralelo. Para mais detalhes deste algoritmo ver no final deste capítulo.

Além desta técnica temos a "cyclic odd-even reduction" que permite que passemos de um algoritmo serial para paralelo. Esta técnica, apesar de diferente da técnica anterior ("recursive doubling"), possui uma "speed-up" de mesma ordem e parece ser aplicada aos mesmo tipo de problema da anterior [Stone 1973b]. Heller [1978] descreve este método para a solução de um sistema de equações lineares triangular.

2.2 Princípio da formulação vetorial

Iniciamos esta seção falando de uma técnica que é bastante direta a partir do algoritmo seqüencial existente e por ser um método bastante utilizado. No entanto, talvez devêssemos ter citado antes uma maneira muito mais imediata de termos algoritmos paralelos. Esta maneira não modifica o algoritmo em nada, ela apenas tenta ver no algoritmo que possuímos as operações vetoriais existentes nele. Ou seja, concentra-se em verificar operações que podem ser executadas simultaneamente para diversos dados. Vejamos como no algoritmo de Gauss, as operações são inerentemente vetoriais.

Consideremos o algoritmo de Gauss para resolução de um sistema linear triangular $Ax = b$. Usualmente partimos da matriz aumentada $[A : b]$ e através de multiplicações, divisões e subtrações procuramos chegar na matriz $[I : x]$, onde x será a solução do problema. Isto pode ser feito, pois quando executamos operações elementares em todo o sistema ele não se altera.

Seja o sistema Ax = b:

$$\begin{bmatrix} a_{11} & 0 & 0 & \dots & 0 & 0 \\ a_{21} & a_{22} & 0 & \dots & 0 & 0 \\ a_{31} & a_{32} & a_{33} & \dots & 0 & 0 \\ & & & & & \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn-1} & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \\ b_n \end{bmatrix}$$

A primeira operação que devemos fazer é dividir a primeira linha por a_{11} (no caso, dividir b_1 por a_{11}). Em seguida, devemos eliminar a_{21} . Para isto, devemos dividir a 1ª linha por a_{11} , multiplicar por a_{21} e subtrair este resultado da 2ª linha. Ou seja, $l_2' = l_2 - a_{21}\, l_1 / a_{11}$.

Aqui, como já sabemos que a_{21}' deve ser 0 e que a_{22} não se modifica, pois a_{12} é igual a 0, as operações acima devem ser realizadas apenas para o elemento b_2 do vetor b: $b_2' = b_2 - (a_{21} / a_{11})* b_1$. Da mesma maneira, para eliminarmos a_{31} devemos fazer

$$l_3' = l_3 - a_{31}\, l_1 / a_{11}.$$

Pelo mesmo motivo de antes, efetuamos estas operações apenas sobre b_3 já que os outros elementos são nulos ou não se modificam. Temos então: $b_3' = b_3 - (a_{31} / a_{11})* b_1$.

Em geral devemos ter : $l_i' = l_i - a_{i1}\, l_1 / a_{11}$ e estes cálculos sendo efetuados apenas para b_i:

$$b_i' = b_i - (a_{i1} / a_{11})* b_1$$

As operações acima talvez sejam mais claramente visualizadas através da representação matricial deste algoritmo.

O que foi descrito equivale a multiplicarmos a matriz [A : b] inicial pela matriz :

$$A^{(1)} = \begin{bmatrix} 1 / a_{11} & 0 & 0 & \dots & 0 & 0 \\ -a_{21} / a_{11} & 1 & 0 & \dots & 0 & 0 \\ -a_{31} / a_{11} & 0 & 1 & \dots & 0 & 0 \\ & & \dots & & & \\ -a_{n1} / a_{11} & 0 & 0 & \dots & 0 & 1 \end{bmatrix}$$

obtendo-se , $A^{(1)} b = \begin{bmatrix} b_1/a_{11} \\ b_2 - a_{21}b_1/a_{11} \\ b_3 - a_{31}b_1/a_{11} \\ \dots \\ b_n - a_{n1}b_1/a_{11} \end{bmatrix}$

E em geral, para eliminarmos os elementos abaixo do elemento a_{ii} multiplicamos por:

$$A^{(i)} = \begin{bmatrix} 1 & 0 & \dots & 0 & 0 & 0 \dots & 0 & 0 \\ 0 & 1 & & & & & & \\ & & 1 & & & & & \\ & & & 1 / a_{ii} & 0 & 0 \dots & 0 & 0 \\ & & & -a_{i+1,i} / a_{ii} & 1 & 0 \dots & 0 & 0 \\ & & & -a_{i+2,i} / a_{ii} & 0 & 1 \dots & 0 & 0 \\ & & & & \dots & & & \\ & & & -a_{ni} / a_{ii} & 0 & 0 \dots & 0 & 1 \end{bmatrix}$$

Chamando cada matriz acima de $A^{(i)}$ temos que a solução x será dada por:

$$x = A^{(n)} A^{(n-1)} \ldots A^{(2)} A^{(1)} b ,$$

pois as operações indicadas podem ser efetuadas apenas sobre os elementos do vetor b.

O que deve ser calculado, então, são os vetores:

$$b^{(i)} = A^{(i)} b^{(i-1)} = \begin{bmatrix} b_1^{(i-1)} \\ b_2^{(i-1)} \\ \ldots \\ b_i^{(i-1)} / a_{ii} \\ b_{i+1}^{(i-1)} - a_{i+1,i} b_i^{(i-1)} / a_{ii} \\ . \\ . \\ b_n^{(i)} - a_{n,i} b_i^{(i-1)} / a_{ii} \end{bmatrix}$$

onde $b^{(0)} = b$ e $b^{(n)} = x$ é a solução do sistema.

Se formos executar este algoritmo seqüencialmente, como o quociente $b_i^{(i-1)} / a_{ii}$ é utilizado em vários cálculos , devemos ter este resultado calculado antes e armazenado. Para a primeira coluna são necessárias, portanto, 1 divisão + (1 multiplicação + 1 subtração) *(n -1) [nº de linhas sobre as quais devem ser efetuadas estas operações], ou seja,

para a 1ª coluna : 1 + 2 * (n-1) operações

para a 2ª coluna : 1 + 2 * (n-2) "

...

para a iª coluna : 1 + 2 * (n-i) "

No total são necessárias, portanto

$$n + 2 * (1+ 2 +...+ n-1) = n + 2*n(n-1)/2 = n + n^2 - n = n^2$$

operações aritméticas [n divisões, n(n-1)/2 multiplicações e n(n-1)/2 subtrações].

Pela representação matricial deste algoritmo pode-se ver claramente que é um algoritmo inerentemente vetorial. Para um sistema de dimensão n , se possuírmos n processadores disponíveis, poderemos efetuar os cálculos indicados no produto $A^{(1)}$ b em 3 passos segundo o esquema apresentado por Schendel [1984]. Além desta maneira existe outra bastante semelhante, que proporemos aqui, mas que pode ser executada utilizando apenas n-1 processadores.

Utilizando o primeiro esquema em que supomos a existência de n processadores, não é necessário calcularmos o quociente b_1/a_{11} primeiramente, pois de qualquer maneira usaremos um passo para este cálculo. Como na matriz A inicial já temos os elementos $a_{11}, a_{21}, \ldots, a_{n1}$ como um vetor natural, dividimos todos os elementos por a_{11} (com exceção do 1º que devemos substituir por 1 e daí fazer a divisão), multiplicamos o resultado por b_1 e subtraímos este último resultado do vetor $(0, b_2, ..., b_n)^t$. Como segundo esquema o qual apresentamos a seguir, calculamos num primeiro passo, como no processo seqüencial, o quociente b_1/a_{11} (que é uma operação escalar). Em seguida, calculamos os produtos

$a_{i1}\ b_1/a_{11}$ com exceção de i =1 (o processador 1 deve ser alocado para os cálculos envolvendo a_{21} e os processadores seguintes ficam responsáveis pelos elementos a_{31}, a_{41},..., a_{n1}). Feito isto, subtraímos o resultado componente a componente do vetor $(b_2,...,b_n)$. No final atribui-se à primeira posição do vetor solução o valor b_1/a_{11}. De qualquer maneira são necessários 3 passos para estes cálculos. No primeiro caso, atribuímos 1 ao 1º elemento de $a^{(1)}$ e 0 ao 1º elemento de b (3 passos e duas operações de atribuição - n processadores). No 2º caso calculamos o escalar b_1/a_{11}, operando sobre n-1 elementos (3 passos e n-1 processadores). A utilização de um algoritmo ou outro irá depender da máquina que se utilizar. Se o uso de um processador a menos é importante ou não, se uma operação escalar é mais rápida do que uma vetorial, se o processo de "shift" é fácil de ser realizado, se o bloqueamento de processadores é algo trivial e se operações de atribuição são rápidas. Os algoritmos são dados por:

Algoritmo 1:

a. atribui-se 1 à primeira posição do vetor $(a_{11},a_{12},...,a_{1n})^t$ obtendo:

$$a^{(1)} = (1,a_{12},...,a_{1n})^t$$

b.(1º passo): divide-se vetor $a^{(1)}$ por a_{11} obtendo-se:

$$a^{(1)'} = (1/a_{11},a_{12}/a_{11},...,a_{1n}/a_{11})^t$$

c.(2º passo): multiplica-se o vetor acima por b_1

$$a^{(1)''} = (b_1/a_{11},b_1a_{12}/a_{11},...,b_1a_{1n}/a_{11})^t$$

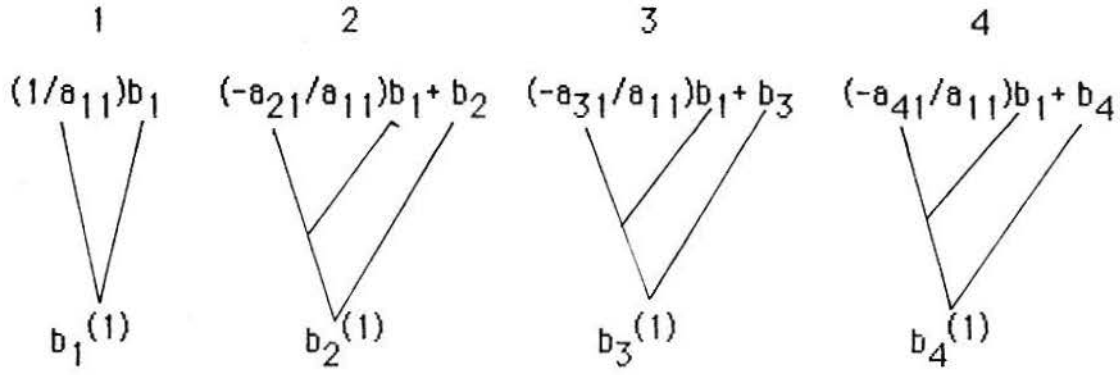
d.atribui-se 0 à 1º posição do vetor b : $(0,b_2,...,b_n)$ ou bloqueia-se processador 1 (verificar qual a melhor forma para o computador que se está utilizando)

e.(3º passo): subtrai-se do vetor $(0,b_2,...,b_n)$ o vetor $a^{(1)"}$ obtendo-se:

$$(b_1/a_{11}, b_2 - b_1a_{12}/a_{11},...,b_n - b_1a_{1n}/a_{11})^t$$

Para este algoritmo são necessários, portanto, n processadores e 3 passos (se não contarmos as operações de atribuição). Esquematicamente, o algoritmo tem a seguinte forma (considerando n=4, para fins de simplificação):

1.Cálculo de $A^{(1)}b$



Foram necessários 3 passos ou 3 unidades de tempo:

-uma divisão (por a_{11})

-uma multiplicação (por b_1)

-uma adição

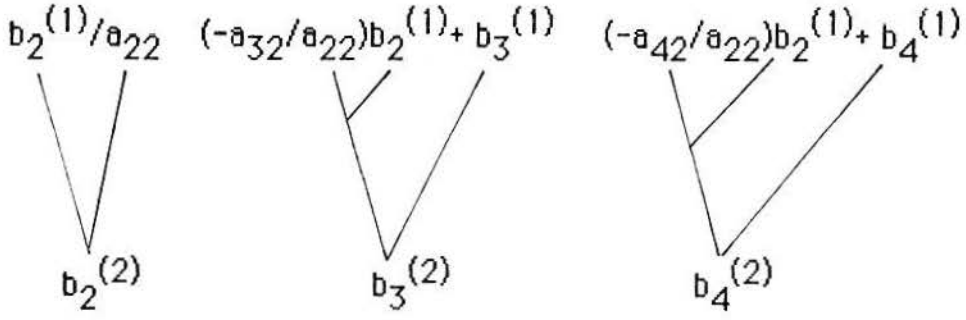
utilizando n processadores.

Num 2º estágio, a matriz que deve ser utilizada para eliminação dos elementos da 2ª coluna de A é a seguinte:

$$A^{(2)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ & 1 / a_{22} & 0 & 0 \\ & - a_{3,2} / a_{22} & 1 & 0 \\ & - a_{4,2} / a_{22} \quad 0 & 0 & 1 \end{bmatrix}$$

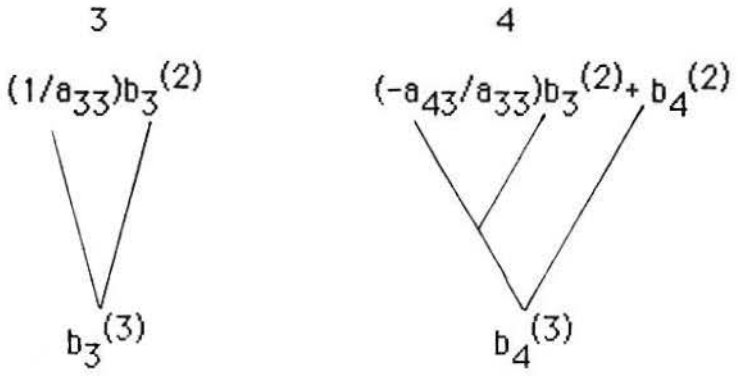
para obter $b^{(2)} = A^{(2)} b^{(1)} = \begin{bmatrix} b_1 / a_{11} = b_1^{(1)} \\ b_2^{(1)} / a_{22} \\ (- a_{3,2} / a_{22}) b_2^{(1)} + b_3^{(1)} \\ (- a_{4,2} / a_{22}) b_2^{(1)} + b_4^{(1)} \end{bmatrix}$

Processador 2 3 4

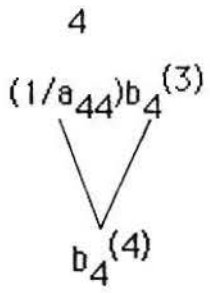


Novamente foram necessários 3 passos e como $b_1^{(2)} = b_1^{(1)}$, não precisamos calcular este valor, precisando de um processador a menos do que antes. Ou seja, utilizamos 3 processadores. Os passos seguintes também não alterarão este valor, logo já temos que $x_1 = b_1 / a_{11}$. Da mesma forma o valor de $b_2^{(2)}$ não será alterado e temos que $x_2 = b_2^{(2)}$. Logo, no próximo passo serão necessários apenas 2 processadores.

Para o cálculo de $A^{(3)}\, b^{(2)}$ o esquema é semelhante aos anteriores:



E finalmente para o cálculo de $A^{(4)}b_3$ temos:



Em cada estágio, com exceção do último, foram necessárias 3 unidades de tempo. No último apenas 2 são necessárias, logo, o algoritmo todo pode ser executado em $3(4-1) + 2 = 9 + 2 = 11$ unidades de tempo. Para n geral precisamos de $3(n-1) + 2 = 3n - 1$ unidades de tempo.

Algoritmo 2

a.(1º passo) Calcula-se o escalar b_1/a_{11}

b.(2º passo) Alocando o processador 1 para os cálculos envolvendo a_{21} e os seguintes para os elementos $a_{31}, a_{41}, \ldots, a_{n1}$ efetuamos:

$$a^{(1)} = (b_1/a_{11})\,(a_{21}, a_{31}, \ldots, a_{n1})$$

UFRGS
SISTEMA DE BIBLIOTECAS
BIBLIOTECA SETORIAL DE MATEMÁTICA

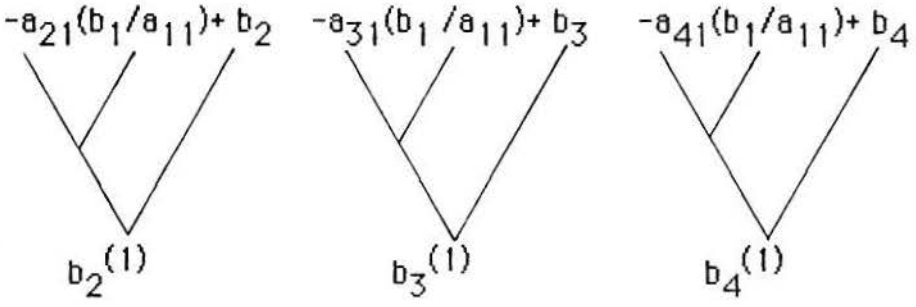
c.(3º passo) Subtrai-se o vetor $a^{(1)}$ do vetor b

d. Atribui-se b_1/a_{11} à variável x_1

O esquema para este algoritmo é semelhante ao anterior. A diferença é que calculamos b_1/a_{11} em um 1º passo e os processadores são alocados diferente.

1º passo: processador escalar: b_1/a_{11} $(= x_1)$

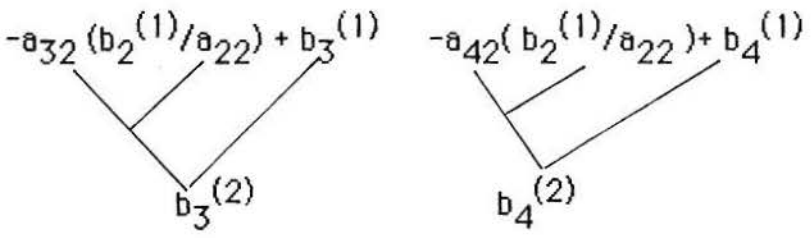
Processador 1 2 3



2º estágio: cálculo de $A^{(2)}b^{(1)}$

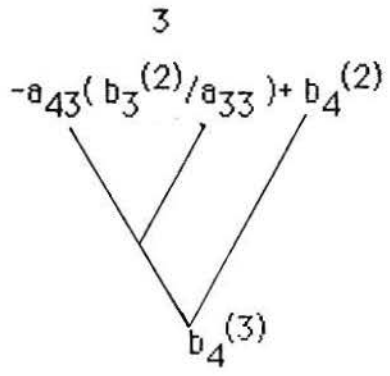
1º passo:calcula $b_2^{(1)}/a_{22}$ $(= x_2)$

Processador 2 3



-Cálculo de $A^{(3)}b^{(2)}$

1º passo: $b_3^{(2)}/a_{33}$



No processamento paralelo do 1º algoritmo em que possuímos n processadores, podemos executar este algoritmo em $3(n-1) + 2 = 3n - 1$ unidades de tempo, pois cada passo (com exceção do último, que precisa de apenas 2) requer 3 unidades de tempo e temos n-1 destes passos. Logo o número de unidades de tempo foi reduzido de n^2 no caso seqüencial para $3n - 1$ no caso paralelo. No 2º algoritmo são necessários n-1 processadores e o mesmo número de passos do 1º algoritmo (3 nos primeiros estágios e 2 no último), no entanto um destes passos é uma operação escalar e são necessárias menor número de operações de atribuição.

Se não possuímos n processadores e temos disponível apenas um número k de processadores que é menor do que n, então cada passo executado num computador com n processadores passará a ser $\lceil n/k \rceil$ passos num computador com k processadores (ou: a cada n passos de um algoritmo executado em um computador serial correspondem $\lceil n/k \rceil$ passos em um computador com k processadores).

No caso deste algoritmo, temos que o número de passos é O(n) e sua speed-up, portanto, é $S_n = n^2 / 3n = n / 3$ e a eficiência $E_n = (n / 3) / n = 1 / 3$. Com k processadores, o número de passos passa a ser $\lceil n^2/k \rceil$ (pois são n^2 passos seqüenciais) , a speed-up é $O(n^2/(n^2/k)) = O(k)$ e a eficiência $E_k = O(k/k) = O(1)$.

2.3 Princípio de Iteração Vetorial (2º Princípio)*

Um outro método de obtermos algoritmos paralelos é o chamado método de iteração vetorial. Este consiste em transformarmos um algoritmo seqüencial direto em um algoritmo paralelo iterativo. Consideremos como exemplo um método para decompor uma matriz tridiagonal A.

Seja $A = \begin{bmatrix} d1 & f1 & & & & \\ e2 & d2 & f2 & & 0 & \\ & e3 & d3 & f3 & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \ddots & \ddots & \\ & & & en-1 & dn-1 & fn-1 \\ & & & & en & dn \end{bmatrix}$

Queremos que A = LU, onde

$L = \begin{bmatrix} 1 & & & \\ m_2 & 1 & 0 & \\ & \ddots & \ddots & \\ 0 & & \ddots & \\ & & m_n & 1 \end{bmatrix}$ $U = \begin{bmatrix} u_1 & f_1 & & \\ & \ddots & \ddots & 0 \\ & & \ddots & \\ 0 & & u_{n-1} & f_{n-1} \\ & & & u_n \end{bmatrix}$

UFRGS
SISTEMA DE BIBLIOTECAS
BIBLIOTECA SETORIAL DE MATEMÁTICA

Efetuando o produto e comparando os coeficientes, vemos que as seguintes relações devem ser satisfeitas:

$$u_1 = d_1 \qquad e_i = m_i u_{i-1}$$

$$d_i = m_i f_{i-1} + u_i \qquad i = 2, \ldots, n$$

Logo, devemos ter: $u_i = d_i - m_i f_{i-1} = d_i - e_i f_{i-1}/u_{i-1}$ (2.4)

(já que $m_i = e_i/u_{i-1}$) (2.5)

Podemos, portanto, calcular os u_i a partir da relação (2.4) acima. Feito isto, os m_i podem ser calculados em paralelo pela relação (2.5) dada anteriormente:

$$m_i = e_i / u_{i-1} \qquad \text{para } i = 2, \ldots, n$$

O cálculo dos u_i é um procedimento seqüencial já que precisamos de u_{i-1} para o cálculo de u_i. Para podermos fazer este cálculo em paralelo transformamos a expressão (2.4) em um procedimento iterativo:

$$u_i^{(0)} = d_i$$

$$u_i^{(j)} = d_i - e_i f_{i-1} / u_{i-1}^{(j-1)} \qquad i = 1, 2, \ldots, n$$

Desta maneira, tendo calculado os $u_i^{(j-1)}$, temos disponíveis todos os elementos para calcular os $u_i^{(j)}$ e podemos, portanto, fazê-lo em

paralelo, calculando todos os u_i ao mesmo tempo, se tivermos n processadores disponíveis . Isto valerá a pena ser feito se o computador que possuímos for capaz de executar uma operação vetorial sobre um vetor de n componentes bem mais rapidamente do que n operações escalares. Além disso, o número de iterações necessárias deve ser bastante menor do que n (pois no processo serial precisávamos de n-1 passos para calcular os u_i).

Uma outra maneira de calcularmos os u_i é através da aplicação da técnica de "recursive doubling". A expressão que tínhamos para u_i era a seguinte: $u_1 = d_1$

$$u_i = d_i - e_i f_{i-1}/u_{i-1} \qquad i=2,\ldots,n$$

Definimos agora

$$q_0 = 1$$

$$q_1 = d_1$$

$$q_i = u_i\, q_{i-1}$$

E daí $$u_i = q_i / q_{i-1} \qquad (2.6)$$

Voltando à expressão (2.4) que tínhamos anteriormente para u_i e substituindo (2.6) calculado para i-1 em (2.4), obtemos:

$$u_i = q_i / q_{i-1} = d_i - e_i f_{i-1}\, q_{i-2} / q_{i-1}$$

e portanto: $$q_i = d_i\, q_{i-1} - e_i f_{i-1}\, q_{i-2} \qquad i = 2, \ldots, n$$

Esta recorrência pode ser expressa, como antes (2.3), na forma vetorial-matricial da seguinte maneira:

$$Q_i = \begin{bmatrix} q_i \\ q_{i-1} \end{bmatrix} = \begin{bmatrix} d_i & -e_i f_{i-1} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} q_{i-1} \\ q_{i-2} \end{bmatrix} = D_i Q_{i-1} \qquad \text{para } i = 2,\ldots,n$$

Temos, então,

$$Q_i = D_i D_{i-1} \ldots D_2 Q_1 \qquad \text{para } i = 2,\ldots,n$$

Novamente temos que Q_n pode ser calculado pela técnica de "recursive doubling" em $\log_2 n$ passos. Para conseguirmos todos os q_i simultaneamente, devemos utilizar n processadores e agrupar os elementos a serem calculados de modo diferente ao que fizemos anteriormente ((2.1) e (2.2)). Obtendo os q_i a partir do cálculo acima, os u_i são calculados a partir da expressão $u_i = q_i / q_{i-1}$ em um único passo se tivermos n processadores, em contraste com o modo seqüencial onde são necessárias n-1 divisões ou O(n) passos. Logo, a speed-up deste algoritmo é $O(\log_2 n / n)$ que é típica da técnica "recursive doubling". Stone [1973b] detalha como chegamos nas expressões para "recursive doubling", assim como analisando as propriedades dos q_i, desenvolve uma maneira mais otimizada para o seu cálculo, ressaltando que o mesmo algoritmo pode ser utilizado para resolver problemas de recorrência de ordem maior. Mostraremos o modo de agrupar as matrizes a serem calculadas através de um exemplo, onde n = 8 (Lambiotte e Voigt [1975]).

Consideremos um computador vetorial com 8 posições ou um "array processor" com 8 processadores. Para o caso de um computador "pipeline" ver [Lambiotte e Voigt 1975]. Sejam D_1, D_2,...,D_8 as matrizes envolvidas no cálculo do produto. Para fins de representação, consideremos o vetor formado por estas matrizes. O seguinte esquema mostra como serão feitos os cálculos. As posições em branco que aparecem nos vetores indicam que a operação correspondente àquela posição já está concluída, bastando repetir o resultado anterior.

$$\begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ D_4 \\ D_5 \\ D_6 \\ D_7 \\ D_8 \end{bmatrix} \begin{bmatrix} \\ D_1 \\ D_2 \\ D_3 \\ D_4 \\ D_5 \\ D_6 \\ D_7 \end{bmatrix} = \begin{bmatrix} D_1 \\ D_{21} \\ D_{32} \\ D_{43} \\ D_{54} \\ D_{65} \\ D_{76} \\ D_{87} \end{bmatrix} \begin{bmatrix} D_1 \\ D_{21} \\ D_{32} \\ D_{43} \\ D_{54} \\ D_{65} \\ D_{76} \\ D_{87} \end{bmatrix} \begin{bmatrix} \\ \\ D_1 \\ D_{21} \\ D_{32} \\ D_{43} \\ D_{54} \\ D_{65} \end{bmatrix} = \begin{bmatrix} D_1 \\ D_{21} \\ D_{321} \\ D_{4321} \\ D_{5432} \\ D_{6543} \\ D_{7654} \\ D_{8765} \end{bmatrix} \begin{bmatrix} D_1 \\ D_{21} \\ D_{321} \\ D_{4321} \\ D_{5432} \\ D_{6543} \\ D_{7654} \\ D_{8765} \end{bmatrix} = \begin{bmatrix} \\ \\ \\ \\ D_1 \\ D_{21} \\ D_{321} \\ D_{4321} \end{bmatrix} = \begin{bmatrix} D_1 \\ D_{21} \\ D_{321} \\ D_{4321} \\ D_{54321} \\ D_{65...21} \\ D_{76...21} \\ D_{87...21} \end{bmatrix}$$

Podemos ver pelo esquema acima que obtivemos todos os produtos Q_i $i=1,...,8$ em $\log_2 8 = 3$ passos, onde em cada passo calculamos $n=8$ produtos de matrizes. Observa-se também, que de uma iteração para outra o número de resultados obtidos é duplicado. Além disso, em cada iteração os resultados que obtivemos na iteração anterior são multiplicados pelo mesmo vetor deslocado duas unidades abaixo. Este mesmo esquema pode ser utilizado para um n qualquer.

Para maiores informações sobre problemas de recorrência e sua resolução ver Schendel [1984] e Kogge [1974].

CAPITULO III

ALGORITMOS PARALELOS PARA RESOLUÇÃO DE SISTEMAS LINEARES

No capítulo anterior tratamos de algumas maneiras para convertermos algoritmos seqüenciais em algoritmos paralelos. Vimos como o algoritmo de eliminação de Gauss é inerentemente paralelo e é, por isso, diretamente adaptável ao processamento em paralelo. Constatamos, também, que o número de passos é da ordem de n em contraste com o algoritmo seqüencial que tem $O(n^2)$ operações. Neste capítulo veremos como a partir da mesma idéia (eliminação de Gauss) podemos desenvolver um outro algoritmo o qual foi apresentado por Sameh e Brent [1977].

Sistemas Lineares Triangulares Densos

Suponhamos que temos um sistema linear $\mathbf{Lx} = \mathbf{b}$, onde L é uma matriz triangular inferior de ordem n.

Como já vimos antes, se definimos M_i como as matrizes abaixo,

$(i = 1,2,\ldots,n)$

$$M_i = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & 1/l_{ii} & & \\ & & -l_{i+1,i}/l_{ii} & 1 & \\ & & \vdots & & \\ & & -l_{ni}/l_{ii} & & 1 \end{bmatrix} \qquad (3.1)$$

temos que a solução x será dada por :

$$\mathbf{x} = \mathbf{M}_n \mathbf{M}_{n-1} \ldots \mathbf{M}_1 \mathbf{b}$$

Aqui, ao invés de calcularmos diretamente a solução como antes, simplesmente "traduzindo" de uma linguagem seqüencial para uma vetorial, utilizaremos o princípio de "recursive doubling" para o cálculo do produto acima. Ou seja, poderemos calcular o produto das matrizes M_i acima em $\lceil \log_2 n \rceil$ passos, onde em cada passo calculamos o produto de matrizes duas a duas. Calculamos no 1º passo, por exemplo, as matrizes: $M_{n-1}M_{n-2}$, $M_{n-3}M_{n-4}$, ..., M_3M_2 e o vetor M_1b. Em um último passo, calculamos o produto de M_n pelo produto das matrizes restantes, o que equivale a uma multiplicação de $1/l_{nn}$ por um vetor nx1, como veremos adiante.

Temos o seguinte esquema para o cálculo destas matrizes:

1º passo: calcula-se os produtos:

$$M_{n-1}M_{n-2} \quad M_{n-3}M_{n-4} \quad M_{n-5}M_{n-6} \quad \ldots \quad M_7M_6 \quad M_5M_4 \quad M_3M_2 \quad M_1b$$

$$M_{n/2-1}^{(1)} \quad M_{n/2-2}^{(1)} \quad M_{n/2-3}^{(1)} \quad \ldots \quad M_3^{(1)} \quad M_2^{(1)} \quad M_1^{(1)} \quad b^{(1)}$$

$$M_{(n-2)/2}^{(1)} \quad M_{(n-4)/2}^{(1)} \quad M_{(n-6)/2}^{(1)} \quad M_3^{(1)} \quad M_2^{(1)} \quad M_1^{(1)} \quad b^{(1)}$$

O último índice é dado por $n/2 - 1$, pois temos n elementos reunidos de 2 em 2. Como temos n/2 pares e o par M_1b não é uma matriz $M_i^{(1)}$, temos que tirar este par dos n/2 pares, ficando, portanto, com $n/2 - 1$ novos elementos. O mesmo raciocínio pode ser utilizado para encontrarmos os próximos índices.

2º passo: $M_{n/4-1}^{(2)}$ $M_{n/4-2}^{(2)}$ ... $M_1^{(2)}$ $b^{(2)}$

3º passo: $M_{n/8-1}^{(3)}$ . . . $b^{(3)}$

:

.

iº passo: $M_{n/2^i-1}^{(i)}$ . . . $M_1^{(i)}$ $b^{(i)}$

:

.

$(\mu-1)$º passo: $M_{n/2^{\mu-1}-1}^{(\mu-1)} = M_1^{(\mu-1)}$ $b^{(\mu-1)}$

μº passo: $b^{(\mu)}$

(onde $2^{\mu} = n$ e daí, $\mu = \log_2 n$)

Num passo seguinte precisamos calcular $M_n\, b^{(\mu)}$, o que equivale a multiplicarmos o último elemento do vetor $b^{(\mu)}$ por $1/l_{nn}$. (Já que $M_n = \text{diag}(1, 1, \ldots, 1/l_{nn})$)

Observando os passos acima e para melhor descrevermos o algoritmo, definimos:

$M_i^{(0)} = M_i$, para $i = 1, 2, \ldots, n-1$

$b^{(0)} = b$

$s_j = 2^j$, para $j = 0, 1, \ldots, \mu = \log_2 n$

$M_i^{(j+1)} = M_{2i+1}^{(j)} M_{2i}^{(j)}$, para $j = 0, 1, \ldots, \mu-2$ (pois $j+1 \leq \mu-1$)

$i = 1, 2, \ldots, \frac{n}{s_{j+1}} - 1$ $\quad (i \leq n/2^{j+1}-1)$

$b^{(j+1)} = M_1^{(j)} b^{(j)}$, para $j = 0, 1, \ldots, \mu-1$

Como as matrizes M_i envolvidas nos cálculos são matrizes não densas, podemos nos valer das propriedades destas matrizes para obtermos os produtos sem termos que calcular da forma geral. Para isto, vejamos de que tipo são estas matrizes e se todas elas possuem formatos semelhantes.

A matriz $M_1^{(0)}$, por exemplo, tem o formato abaixo:

$$M_1^{(0)} = \begin{bmatrix} & 1/l_{11} & \\ -l_{21}/l_{11} & 1 \; 0 & 0 \\ \vdots & 1 \; 0 & \\ \cdot & \cdot & \\ -l_{i1}/l_{11} & 1 \; 0 & \\ \vdots & & \\ -l_{n1}/l_{11} & & 1 \end{bmatrix}$$

E assim, todas as matrizes $M_i^{(0)}$ como são as próprias matrizes M_i, elas tem o formato dado por (3.1). As matrizes $M_i^{(j)}$ são o produto de duas matrizes calculadas anteriormente :

$M_i^{(j)} = M_{2i+1}^{(j-1)} M_{2i}^{(j-1)}$ e daí, podemos mostrar por indução que todas as matrizes $M_i^{(j)}$ tem o seguinte formato:

$$M_i^{(j)} = \begin{bmatrix} I_i^{(j)} & 0 & 0 \\ 0 & L_i^{(j)} & 0 \\ 0 & S_i^{(j)} & I'_i^{(j)} \end{bmatrix} \qquad (3.2)$$

onde $I_i^{(j)}$ e $I'_i^{(j)}$ são matrizes identidade de ordem

$q_i^{(j)} = is_j - 1$ e

$r_i^{(j)} = n + 1 - (i + 1)s_j$, respectivamente e

$L_i^{(j)}$ é uma matriz triangular inferior de ordem $s_j = 2^j$ (conseqüentemente, $S_i^{(j)}$ tem ordem $r_i^{(j)} \times s_j$).

Podemos ver claramente que as matrizes $M_i^{(0)}$ tem este formato. Para estas matrizes, $j = 0$ e portanto $s = 2^0 = 1$. Logo, $L_i^{(j)}$ tem ordem 1 e é dada por $L_i^{(j)} = 1/ l_{ii}$; $S_i^{(j)}$ é formada pelos elementos que estão abaixo de $1/l_{ii}$ e portanto, $S_i^{(j)} = (-l_{i+1,i} /l_{ii} \ -l_{i+2,i}/l_{ii} \cdots \ -l_{ni}/l_{ii})^t$. As matrizes identidade tem ordem $i - 1$ e $n +1- (i + 1) = n - i$ (ou, $n - (i-1) -1 = n - i$), respectivamente.

Para facilitarmos a demonstração, que é feita por indução, particionamos a matriz $S_i^{(j)}$ da seguinte forma:

$$S_i^{(j)} = \begin{bmatrix} U_i^{(j)} \\ V_i^{(j)} \end{bmatrix} \qquad i=1,2,\ldots,n/s_j - 2 \qquad (3.3)$$

onde $U_i^{(j)}$ é uma matriz quadrada de ordem s_j.

Como a forma dada para $M_i^{(j)}$ é válida para $j=0$, provemos que é válida para $2 \leq j+1 \leq \mu-1$, supondo que vale para j.

Temos que $M_i^{(j+1)}$satisfaz a seguinte relação:

$$M_i^{(j+1)} = M_{2i+1}^{(j)} M_{2i}^{(j)}$$

Daí, esquematicamente, temos:

$$M_i^{(j+1)} = \begin{bmatrix} I_{2i+1}^{(j)} & 0 & 0 \\ 0 & L_{2i+1}^{(j)} & 0 \\ 0 & S_{2i+1}^{(j)} & I'_{2i+1}^{(j)} \end{bmatrix} \begin{bmatrix} I_{2i}^{(j)} & 0 & 0 \\ 0 & L_{2i}^{(j)} & 0 \\ 0 & S_{2i}^{(j)} & I'_{2i}^{(j)} \end{bmatrix}$$

$$= \begin{bmatrix} I_{2i}^{(j)} & 0 & 0 & 0 \\ 0 & I_{sj} & 0 & 0 \\ 0 & 0 & L_{2i+1}^{(j)} & 0 \\ 0 & 0 & S_{2i+1}^{(j)} & I'_{2i+1}^{(j)} \end{bmatrix} \begin{bmatrix} I_{2i}^{(j)} & 0 & 0 & 0 \\ 0 & L_{2i}^{(j)} & 0 & 0 \\ 0 & U_{2i}^{(j)} & I_{sj} & 0 \\ 0 & V_{2i}^{(j)} & 0 & I'_{2i+1}^{(j)} \end{bmatrix}$$

Podemos particionar assim, havendo correspondência entre as partições , pois:

- $I_{2i+1}^{(j)}$ tem dimensão $(2i+1)s_j - 1 = 2is_j - 1 + s_j = \dim I_{2i}^{(j)} + s_j$
- $\dim I'_{2i}^{(j)} = n+1 - (2i+1)s_j$
- $\dim I'_{2i+1}^{(j)} = n+1 - (2i+2)s_j = n+1 - (2i+1)s_j - s_j = \dim I'_{2i}^{(j)} - s_j$

ou seja: $-\dim I'_{2i}^{(j)} = \dim I'_{2i+1}^{(j)} + s_j$

A multiplicação destas matrizes pode ser efetuada como se os blocos que a formam fossem elementos, pois as dimensões dos "blocos-elementos" a serem multiplicados são compatíveis. Por exemplo, $\dim I_{sj} = 2^j = \dim L_{2i}^{(j)}$ e $\dim L_{2i+1}^{(j)} = s_j = 2^j = \dim U_{2i}^{(j)}$ e podemos, portanto multiplicar I_{sj} por $L_{2i}^{(j)}$ e $L_{2i+1}^{(j)}$ por $U_{2i}^{(j)}$. Da mesma maneira, as matrizes $S_{2i+1}^{(j)}$, $V_{2i}^{(j)}$ tem dimensão $n+1-(2i+2)\,s_j \times s_j$ e $I'_{2i+1}^{(j)}$ é uma matriz quadrada de dimensão $n+1-(2i+2)\,s_j$.

Permitindo que as multiplicações $S_{2i+1}^{(j)}\ U_{2i}^{(j)}$, $l'_{2i+1}{}^{(j)}\ V_{2i}^{(j)}$, $S_{2i+1}^{(j)}\ I_{sj}$ sejam efetuadas (lembrando que para multiplicarmos duas matrizes A e B, devemos ter nº de colunas de A = nº de linhas de B e obtemos uma matriz C que satisfaz: nº de linhas de C= nº de linhas de A e nº de colunas de C = nº de colunas de B)

Visto isto, obtemos (reescrevemos as matrizes $M_{2i}^{(j)}$ e $M_{2i+1}^{(j)}$ para maior clareza na obtenção do produto):

$$M_i^{(j+1)} = \begin{bmatrix} I_{2i}^{(j)} & 0 & 0 & 0 \\ 0 & I_{sj} & 0 & 0 \\ 0 & 0 & L_{2i+1}^{(j)} & 0 \\ 0 & 0 & S_{2i+1}^{(j)} & l'_{2i+1}{}^{(j)} \end{bmatrix} \begin{bmatrix} I_{2i}^{(j)} & 0 & 0 & 0 \\ 0 & L_{2i}^{(j)} & 0 & 0 \\ 0 & U_{2i}^{(j)} & I_{sj} & 0 \\ 0 & V_{2i}^{(j)} & 0 & l'_{2i+1}{}^{(j)} \end{bmatrix}$$

$$= \left[\begin{array}{c|c|c|c|l} I_{2i}^{(j)} & 0 & 0 & 0 & 2i2^{j}-1 \\ \hline 0 & L_{2i}^{(j)} & 0 & 0 & 2^{j} \\ 0 & L_{2i+1}^{(j)} U_{2i}^{(j)} & L_{2i+1}^{(j)} & 0 & 2^{j} \\ \hline 0 & S_{2i+1}^{(j)} U_{2i}^{(j)} + V_{2i}^{(j)} & S_{2i+1}^{(j)} & l'_{2i+1}{}^{(j)} & n-22^{j}-2i2^{j}+1 \end{array}\right]$$

$$2i2^{j}-1 = 2is_j-1 \qquad 2^{j} = s_j \qquad 2^{j} \qquad n-2.2^{j}-2i2^{j}+1 \qquad n+1-(i+1)2^{j+1}$$

$$is_{j+1}-1 \qquad n+1-(i+1)2^{j+1} = n+1-(i+1)s_{j+1}$$

$$= \begin{bmatrix} I_i^{(j+1)} & 0 & 0 \\ 0 & L_i^{(j+1)} & 0 \\ 0 & S_i^{(j+1)} & I'_i{}^{(j+1)} \end{bmatrix}$$

onde

$\cdot I_i^{(j+1)} = I_{2i}^{(j)}$ é uma matriz identidade de dim = $2i2^j - 1 = i2^{j+1} - 1$

$$\cdot L_i^{(j+1)} = \begin{bmatrix} L_{2i}^{(j)} & 0 \\ L_{2i+1}^{(j)} U_{2i}^{(j)} & L_{2i+1}^{(j)} \end{bmatrix} \qquad (3.4)$$

é novamente, uma matriz triangular, pois $L_{2i}^{(j)}$ e $L_{2i+1}^{(j)}$ são matrizes triangulares e tem dimensão:

$$\dim L_i^{(j+1)} = \dim L_{2i}^{(j)} + \dim L_{2i+1}^{(j)} = s_j + s_j = 2s_j = s_{j+1}$$

$$\cdot S_i^{(j+1)} = \begin{bmatrix} S_{2i+1}^{(j)} U_{2i}^{(j)} + V_{2i}^{(j)} & S_{2i+1}^{(j)} \end{bmatrix} \qquad (3.5)$$

$$\dim S_i^{(j+1)} = (s_j + s_j) \times (n+1 - (2i+1+1)s_j) = 2s_j \times (n+1 - (i+1)2s_j)$$

$$= s_{j+1} \times (n+1 - (i+1)s_{j+1}$$

$$\cdot I'_i{}^{(j+1)} = I'_{2i+1}{}^{(j)}$$

é uma matriz identidade de dimensão $n+1 - (i+1)s_{j+1}$

UFRGS
SISTEMA DE BIBLIOTECAS
BIBLIOTECA SETORIAL DE MATEMÁTICA

(dimensão de $l'_{2i+1}{}^{(j)} = n+1 - (2i+2)s_j = n+1-(i+1)2s_j =$

$= n+1 - (i+1)22^j = n+1 - (i+1)2^{j+1} =$

$= n+1 - (i+1)s_j$)

Com isto, provamos que as matrizes $M_i^{(j)}$ tem todas a forma (3.2) citada com as respectivas dimensões sendo satisfeitas. As expressões dadas são válidas para : $j = 0,1 , \dots , \mu-2$

$i = 1,2 , \dots , (n/ s_{j+1}) - 1$

Se particionamos o vetor b e posteriormente os vetores $b^{(j)}$ com dimensões compatíveis as das matrizes $M_1^{(j)}$ podemos efetuar as multiplicações por blocos, como fizemos antes.

Ou seja, particionamos os vetores $b^{(j)}$ em vetores de comprimento s_j-1, s_j e $n - (s_j - 1) - s_j = n + 1 - 2s_j$, obtendo:

$$b^{(j)} = \begin{bmatrix} g_1^{(j)} & 2^{j-1} \\ g_2^{(j)} & 2^j \\ g_2^{(j)} & n+1-2s_j \end{bmatrix} \qquad (3.6)$$

e como as $M_1^{(j)}$ tem o seguinte formato:

$$\begin{bmatrix} I_{s_j-1} & 0 & 0 \\ 0 & L_1^{(j)} & 0 \\ 0 & S_1^{(j)} & I_{n+1-2s_j} \end{bmatrix}$$

temos que a expressão para $b^{(j+1)}$ é a seguinte:

$$b^{(j+1)} = M_1^{(j)} b^{(j)} = \begin{bmatrix} g_1^{(j)} \\ L_1^{(j)} g_2^{(j)} \\ S_1^{(j)} g_2^{(j)} + g_3^{(j)} \end{bmatrix} \qquad (3.7)$$

para $j = 0, 1, \ldots, \mu-1$

Seguindo as operações indicadas anteriormente para a resolução do sistema linear **Lx** = **b** , onde **L** é uma matriz triangular inferior de dimensão n, obtemos o seguinte algoritmo, onde calculamos as matrizes $M_i^{(j)}$ e no final o vetor $b^{(\mu+1)} = M_n\, b^{(\mu)}$ que é a solução do sistema.

Algoritmo para resolução de Lx = b, com L triangular inferior

[1]- Seja $L_i^{(0)} = \begin{bmatrix} 1/l_{ii} \end{bmatrix}$

$$S_i^{(0)} = \begin{bmatrix} -l_{i+1,i} / l_{ii} \\ -l_{i+2,i} / l_{ii} \\ \vdots \\ \cdot \\ -l_{n,i} / l_{ii} \end{bmatrix}$$

$$b^{(0)} = b$$

[2]- Repetir os seguintes procedimentos para $j = 0,1, \ldots, \mu-2$ ($\mu = \log_2 n$)

- $s_j = 2^j$
- Particionar $S_i^{(j)}$ como em (3.3)
  $b^{(j)}$ como em (3.6)

-Calcular $L_i^{(j+1)}$ como em (3.4)
$S_i^{(j+1)}$ como em (3.5) (para $i=1,2,\ldots,n/s_{j+1} - 1$)
$b^{(j+1)}$ como em (3.7)

[3] - Particionar $b^{(\mu-1)}$ como em (3.6) ($s_{\mu-1} = 2^{\mu-1}$)

- Calcular $b^{(\mu)}$ através de (3.7)

[4] - A solução **x** é dada por uma multiplicação de M_n por $b^{(\mu)}$,ou seja,

$$\mathbf{x} = \begin{bmatrix} b_1^{(\mu)} \\ b_2^{(\mu)} \\ \vdots \\ b_{n-2}^{(\mu)} \\ b_{n-1}^{(\mu)} \\ b_n^{(\mu)} / l_{nn} \end{bmatrix}$$

(pois $M_n = \begin{bmatrix} 1 & 0 & & \\ 0 & 1 & & \\ & & 1 & 0 \\ & & 0 & 1/l_{nn} \end{bmatrix}$)

Tendo obtido o algoritmo, vejamos quantos processadores e quantas etapas são necessários para executá-lo. Primeiramente, para formar cada matriz M_i necessitamos efetuar divisões e subtrações. Se possuirmos o número de processadores que precisarmos para operar sobre todos os elementos simultaneamente, estas operações podem ser executadas em dois passos. Para que isto seja possível, precisamos de

-n processadores para a matriz M_1 (pois devemos operar sobre os n elementos da 1ª coluna)

-n-1 processadores para a matriz M_2 (pois são n-2 elementos não nulos na 2ª coluna)

:

-n-i+1 processadores para a matriz M_i (já que os elementos não-nulos da coluna i são os abaixo da posição (i,i) incluindo este)

Se tivermos, portanto, $\sum_{i=1}^{n} (n-i+1) = \sum_{i=1}^{n} n+1 - \sum_{i=1}^{n} i = n(n+1) - n(n+1)/2 =$

$= n(n+1)/2$ processadores, poderemos obter todas as matrizes M_i ao mesmo tempo.

Para resolver o sistema **Lx** = **b** precisamos, ainda, calcular as matrizes $M_i^{(j)}$. Em cada uma destas matrizes, aparecem, como vimos antes, os seguintes produtos: $L_{2i+1}^{(j)} U_{2i}^{(j)}$, $S_{2i+1}^{(j)} U_{2i}^{(j)}$ e nos vetores $b^{(j)}$, os produtos $L_1^{(j)} g_2^{(j)}$ e $S_1^{(j)} g_2^{(j)}$. As matrizes $L_{2i+1}^{(j)}$ e $U_{2i}^{(j)}$ são matrizes quadradas de dimensão $s = 2^j$ e as matrizes $S_{2i+1}^{(j)}$ e $S_1^{(j)}$ tem dimensões $(n+1) - (2i+2)s \times s$ e $n+1-2s \times s$ respectivamente.

UFRGS
BIBLIOTECA SETORIAL DE MATEMÁTICA

Logo, para calcular

$L_{2i+1}^{(j)} U_{2i}^{(j)}$ devemos calcular s^2 produtos internos de comprimentos 1,2,..., s

para $S_{2i+1}^{(j)} U_{2i}^{(j)}$ devemos calcular $s \cdot (n+1-(2i+2)s)$ produtos internos de comprimento $\underline{s}$ e, finalmente, para o cálculo dos vetores

$L_1^{(j)} g_2^{(j)}$ são necessários s produtos internos de comprimento $\underline{s}$ e para

$S_1^{(j)} g_2^{(j)}$ são necessários $n+1-(2i+2)s$ produtos internos de comprimento s.

Em todos os produtos acima, vimos ser necessário o cálculo de produtos internos de vetores de comprimento s. Possuindo um computador com no mínimo s processadores, podemos calcular cada um destes produtos internos em $1 + \log_2 s = 1 + \log_2 2^j = 1 + j$ passos , sendo que um passo é necessário para o produto de elementos correspondentes e $\log_2 s$ passos são necessários para a adição destes produtos por "recursive doubling" .

Para as somas $S_{2i+1}^{(j)} U_{2i}^{(j)} + V_{2i}^{(j)}$ e $S_1^{(j)} g_2^{(j)} + g_3^{(j)}$ mais um passo é suficiente (desde que possuamos o número de processadores necesssários para efetuar esta soma, como será visto em seguida). Portanto, para cada etapa (j+1) são necessários um total de $1+j+1 = 2+ j$ passos, sendo que no final ainda devemos calcular o produto $M_n b^{(\mu)}$ (para o qual mais um passo é suficiente).

Logo, o tempo total necessário para calcularmos a solução do sistema será dado por:

$$t = 2 + 1 + \sum_{j=0}^{\mu-1} (2 + j) = 3 + 2\mu + \sum_{j=0}^{\mu-1} j = 3 + 2\mu + \mu(\mu-1)/2 =$$

$$= 3 + \mu(\mu+3)/2$$

onde os dois primeiros passos são relativos ao cálculo das M_i e o 3º se refere ao cálculo do produto final $M_n b^{(\mu)}$.

Para que o número de passos seja realmente este, devemos ser capazes de executar todas as operações necessárias em cada passo simultaneamente, precisando, para isto, saber quantos processadores são necessários.

Comecemos a análise, verificando o que ocorre no produto $M_i^{(j+1)} = M_{2i+1}^{(j)} M_{2i}^{(j)}$.

Para este produto, devemos calcular os produtos $L_{2i+1}^{(j)} U_{2i}^{(j)}$ e $S_{2i+1}^{(j)} U_{2i}^{(j)}$, o que pode ser feito simultaneamente.

Para o cálculo de $L_{2i+1}^{(j)} U_{2i}^{(j)}$ devemos efetuar produtos internos entre pares de vetores. Para formar cada coluna, $\underline{s}$ destes produtos internos são necessários já que o número de linhas de $L_{2i+1}^{(j)}$ é igual a s e como $L_{2i+1}^{(j)}$ é triangular inferior, o tamanho destes vetores será 1, 2,...,s.

Logo, para que o cálculo de todos os elementos de uma coluna possa ser feito simultaneamente, devemos ter um total de

$$\sum_{k=1}^{s} k = s(s+1)/2 \quad \text{processadores para cada coluna}$$

e portanto um total de $s^2(s+1)/2$ processadores para o produto todo (por haver s colunas).

Calcular $S_{2i+1}^{(j)}U_{2i}^{(j)}$ envolve $sr_{2i+1}^{(j)}$ produtos internos entre vetores de comprimento s. Como para cada produto interno necessitamos de s processadores, devemos ter um total de $s^2r_{2i+1}^{(j)} = s^2r_i^{(j+1)}$ processadores para todo o produto.

Portanto, para calcularmos os dois produtos entre matrizes simultaneamente, devemos ter um total de

$$\pi' = s^2(s+1)/2 + s^2r_i^{(j+1)} = s^2[(s+1)/2 + (n+1) - (i+1)2s] =$$
$$= \tfrac{1}{2}s^2[\, s+1 + 2n + 2 - 4is - 4s] = \tfrac{1}{2}s^2[\, 2n + 3 - 3s - 4is] =$$
$$= \tfrac{1}{2}s^2(2n+3) - \tfrac{1}{2}s^3(4i+3) \quad \text{processadores.}$$

Obtidos estes produtos, precisamos ainda calcular a soma $S_{2i+1}^{(j)}U_{2i}^{(j)} + V_{2i}^{(j)}$ que pode ser feita em um único passo se possuírmos $sr_{2i+1}^{(j)} = sr_i^{(j+1)}$ processadores ou seja,

$$\pi'' = s\,[n+1 - 2s\,(i+1)\,] = s(n+1) - 2s^2(i+1) \quad \text{processadores.}$$

Logo, para obtermos $M_i^{(j+1)}$ deveremos ter um número de processadores que comporte as duas etapas acima, ou seja, deveremos ter

$$\pi = MAX [\pi', \pi''] = \pi' .$$

Para isto, lembremos que $s = 2^j$ que j varia de 0 a $\mu-2$ e que i varia de 1 a $n/2s - 1$. Daí, temos que

$$\pi' = \tfrac{1}{2}(2^j)^2(2n + 3) - \tfrac{1}{2}(2^j)^3 (4i+3) =$$
$$= \tfrac{1}{2}2^{2j}(2n+3) - \tfrac{1}{2}2^{3j}(4i+3) =$$
$$= 2^{2j-1}(2n+3) - 2^{3j-1}(4i+3)$$

e

$$\pi'' = 2^j(n+1) - 22^{2j}(i+1) =$$
$$= 2^j(n+1) - 2^{2j+1}(i+1)$$

Para termos um estágio completo falta calcularmos $b^{(j+1)}$. Como $b^{(j+1)} = M_1^{(j)}b^{(j)}$ e pelo que foi visto anteriormente para simplificarmos este cálculo devemos calcular os seguintes subprodutos: $L_1^{(j)}g_2^{(j)}$ e $S_1^{(j)}g_2^{(j)}$. $L_1^{(j)}$ tem dimensão s e $S_1^{(j)}$ tem dimensão $n+1-2s$ x s ,logo devemos calcular produtos internos de vetores de ordem 1,2,...,s no 1º caso e vetores de ordem s no 2º caso, sendo que no 2º, $n+1-2s$ produtos internos devem ser calculados.

Precisamos, portanto, de um total de

$$\pi_0 = \Sigma_1^{\,s} k + s(n+1-2s) = s(s+1)/2 + sn + s - 2s^2 =$$
$$= s^2/2 + s/2 + sn + s - 2s^2 = s(1/2 + n + 1) - (3/2)s^2 =$$
$$= s(n + 3/2) - (3/2)s^2 =$$
$$= \tfrac{1}{2}s(2n + 3) - 3/2\ s^2$$ processadores (pois para a soma $S_1^{(j)}g_2^{(j)} + g_3^{(j)}$ precisamos apenas de $n + 1 - 2s$ processadores que é menor que o número de processadores dado acima).

Temos, portanto, completo o número de processadores para cada estágio j+1 que é a soma do número de processadores necessários para o cálculo de $M_i^{(j+1)}$ com o número de processadores necessários para o cálculo de $b^{(j+1)}$. Ou seja, para cada estágio j+1 são necessários um total de

$$\pi^{(j+1)} = \sum_{k=0}^{n/(2s)-1} \pi_k^{(j+1)} = \sum_{k=0}^{n/(2s)-1} \tfrac{1}{2}(2n+3)s^2 - \tfrac{1}{2}(4k+3)s^3 =$$

$$= \tfrac{1}{2}(2n+3)s^2 n/2s - \tfrac{1}{2}4s^3 \sum k - \tfrac{1}{2}3s^3 n/2s =$$

$$= \tfrac{1}{2}n^2 s + 3/4\, sn - \tfrac{1}{4}4s^3 n/2s\,(n/2s - 1) - \tfrac{1}{4}3s^2 n =$$

$$= s\,(\tfrac{1}{2}n^2 + 3/4\, n) - \tfrac{1}{2}s^2 n\,(n/2s - 1) - 3/4\, s^2 n =$$

$$= \ldots.$$

$$= 3/2\, s^3 - \tfrac{1}{4}\,(5n+12)s^2 + \tfrac{1}{4}\,(n^2 + 7n + 6)s$$

Como o número de processadores deve ser suficiente para todos os estágios, devemos ter que o algoritmo todo necessita de

$$\pi = \max_{0 \le j \le \mu-2} [\, \pi^{(j+1)}\, , n(n+1)/2\,] \quad \text{processadores.}$$

Para $n \geq 16$, este máximo é atingido para $s = n/8$ ($j = \log_2 s = \log_2 n/8 = \log_2 n - \log_2 8 = \mu - 3$) e é dado por $\pi^{(j+1)}$ calculado neste valor:

$$\pi = \underline{3}\,(n/8)^3 - \tfrac{1}{4}(5n+12)(n/8)^2 + \tfrac{1}{4}(n^2 + 7n + 6)n/8 =$$
$$2$$

$$= \underline{3}\, n^3/512 - \tfrac{1}{4}(5n+12)n^2/64 + 1/32\,(n^3 + 7n^2 + 6n)$$
$$2$$

$$= (n^3/32)\,(3/2.16 - 5/4.2 + 1) - n^2(3/64 - 7/32) + 3n/16 =$$

$$\pi = (n^3/32)(15/32) - n^2(-11/64) + 3n/16 =$$
$$= 15n^3/1024 + 11n^2/64 + 3n/16 =$$
$$= n/64\,(\,15n^2/16 + 11n + 12)$$

Ou seja, $\pi \cong n^3/68 + o(n^2)$ processadores são necessários (pois 15/16 = 1/ 1.0667 e daí, 15 / 16 . 64 = 1 / 1,0667 . 64 = = 1 / 68,2667 ≅ 1 / 68).

Temos, portanto, que o algoritmo dado soluciona o sistema triangular Lx = b , onde L tem ordem n, em $\tau = 3 + \mu(\mu+3)/2 = 3 + \log_2 n\,(\log_2 n + 3)/2$

$$\tau = \tfrac{1}{2}(\log_2 n)^2 + (3/2)\log_2 n + 3 \qquad \text{passos}$$

e necessita de um número de processadores da ordem de:

$$\pi = (15/1024)\, n^3 + O(n^2).$$

Análise de erro

Para que um algoritmo seja eficiente devemos saber como ele se comporta em relação aos erros, se a solução que o algoritmo dá é razoável ou se erros de arredondamento afetam de tal forma os resultados que a resposta obtida está muito distante de real solução do problema.

Sameh e Brent [1977] fazem uma análise cuidadosa do erro cometido ao calcularmos a solução de Lx = b, onde L é triangular inferior e o algoritmo é o dado acima.

Como utilizamos em todo algoritmo o cálculo de produtos internos, vejamos qual o erro cometido neste tipo de operação.

Sejam $a = (a_1, a_2, \ldots, a_n)$ e $b = (b_1, b_2, \ldots, b_n)$ dois vetores de ordem n. Podemos calcular o produto $\mathbf{ab}^t$ em $(1 + \log_2 n)$ passos e teremos que o cálculo em ponto flutuante satisfará:

$|fl(ab^t) - ab^t| \leq (1+\log_2 n)\, \varepsilon\, |a|\,|b^t|$, onde ε é a unidade de arredondamento de máquina.

Esta afirmação pode ser demonstrada facilmente, considerando os cálculos envolvidos no produto interno de dois vetores que nada mais são do que multiplicações e adições. Como já vimos anteriormente, para obtermos o produto interno entre dois vetores, executamos num primeiro passo as n multiplicações entre componentes correspondentes : $a_i b_i$ para $i=1,\ldots,n$. Obtidos estes resultados, efetuamos a soma $a_1b_1 + a_2b_2 + \ldots + a_nb_n$ por "recursion doubling" , somando estes elementos (e as somas posteriores) de dois em dois, o que pode ser feito em $\log_2 n$ passos. Sameh e Brent [1977] demonstram esta afimação e a utilizam para estabelecer limites nas "matrizes erro" $|E_i^{(j+1)}|$ e no "vetor erro" $|e^{(j+1)}|$, os quais são definidos por:

$$fl(M_i^{(j+1)}) = fl(M_{2i+1}^{(j+1)})\, fl(M_{2i}^{(j)}) + E_i^{(j+1)} \qquad i=1,2,\ldots,(n/2^{j+1})-1$$

$$fl(b^{(j+1)}) = fl(M_1^{(j)})\, fl(b^{(j)}) + e^{(j+1)} \qquad j=0,1,\ldots,\mu-1$$

onde $|M|$ indica a matriz cujos elementos são os módulos dos elementos da matriz original. Utilizando estes limites dados e supondo que $\|L\|_\infty \leq 1$ obtemos os seguintes limites para as normas de $E_i^{(j+1)}$ e $e^{(j+1)}$:

$$\| E_i^{(j+1)} \| \leq (j+2)(1+2j)\varepsilon \, \| L^{-1} \|^2$$

$$\| e^{(j+1)} \| \leq (j+2)(1+2j)\varepsilon \, \| L^{-1} \| \, \| x \|.$$

Após algumas considerações sobre o formato de $E_i^{(j+1)}$, com o que ocorre no produto de matrizes $fl(M_i^{(j)})$ e no cálculo de $fl(b^{(j)})$ obtemos uma relação satisfeita pela solução calculada x, a qual juntamente com resultados anteriores permite concluir que existe uma matriz perturbação δL tal que

$$(L + \delta L)\tilde{x} = b$$

onde

$$\| \delta L \| \leq \alpha_1 \varepsilon \| L^{-1} \|^2 \qquad (**)$$

$$\text{com } \alpha_1 = \tfrac{1}{4} n^2 \log_2 n + O(n \log_2 n)$$

Se retirarmos a hipótese $\| L \| \leq 1$ (**) fica:

$\| \delta L \| \leq \alpha_1 \varepsilon \kappa^2(L) \| L \|$ onde $\kappa(L)$ é o condicionamento da matriz L ($\kappa(L) = \| L \| \| L^{-1} \|$).

Obtemos que o erro relativo na solução x ($fl(x) = \tilde{x}$) é limitado por:

$$\| \tilde{x} - x \| / \| x \| \leq \alpha_1 \varepsilon \kappa^3(L) \| L \|$$

Os limites acima não são muito favoráveis já que em suas expressões aparecem os valores κ^2 e κ^3 que podem ser bastante grandes. Além disso, o erro da solução do sistema calculada seqüencialmente satisfaz a relação

$$\| \delta L \| \leq n\varepsilon \| L \| \qquad \text{(Wilkinson [1963])}$$

que pode ser bem menor do que a expressão dada para o algoritmo paralelo já que não envolve o número de condicionamento da matriz L e $\| L \|$ aparece com potência 1. Isto poderia nos levar a acreditar que a solução paralela é inferior à solução seqüencial, no entanto, experimentos realizados por Sameh e Brent[1977] mostram que não é sempre este o caso, sendo que em muitas ocasiões os erros cometidos possuíam a mesma magnitude em ambos os casos.

Pode-se encontar exemplos em que o resultado paralelo é menos exato que o seqüencial, no entanto os outros casos em que isto não ocorre nos mostra que os limites estabelecidos podem ser bastante pessimistas. A análise formal de erros é importante para termos uma idéia do que podemos obter, mas devemos sempre que possível testar os resultados obtidos com esta análise através da experimentação. Além disso, se através desta análise (que por vezes é bastante difícil de se fazer) constatarmos, como foi este o caso, que o erro cometido pode ser bastante grande, devemos utilizar a precisão-dupla para calcularmos e armazenarmos os produtos internos necessários. Este procedimento pode reduzir bastante os erros que poderiam ser cometidos e tornar estável o algoritmo.

Sistemas lineares tridiagonais e tridiagonais por blocos

Sistemas de equações lineares tridiagonais ou tridiagonais por blocos aparecem seguidamente nas aplicações e por terem banda pequena não é adequado resolvê-los com os métodos existentes para matrizes banda, necessitando de algoritmos especiais para eles.

Os sistemas tridiagonais aparecem, por exemplo, na solução numérica de equações diferenciais parciais. Em particular, quando discretizamos uma equação diferencial parcial elíptica através de diferenças finitas, verificamos que o tipo de matriz que surge, é justamente a que estamos considerando aqui.

UFRGS
ESCOLA DE ENGENHARIA
BIBLIOTECA SETORIAL DE MATEMÁTICA

Consideremos, portanto, a equação de Laplace :

$$\Delta U = u_{xx} + u_{yy} = 0$$

com condições de fronteira: $u(x,y) = g(x,y) \ \forall (x,y) \in$ fronteira de G

Estamos interessados aqui, em encontrar uma função **u** que satisfaça as equações acima e que pertença a C^2. G é a região de definição da função e será tomada aqui como um quadrante unitário em $\mathbf{R}^2$. Tomamos este quadrante e formamos uma rede dividindo-no por linhas paralelas aos eixos com igual distância entre elas (tantas quantas necessárias para obter as aproximações desejadas). Aplica-se sobre este reticulado um método de diferenças finitas, o que permitirá transformar o problema contínuo inicial em um problema discreto passível de ser resolvido numericamente. Este método dará portanto uma aproximação dos valores da função **u** nos nodos deste reticulado.

Transformamos a equação $u_{xx} + u_{yy} = 0$, utilizando as seguintes aproximações para cada uma das derivadas parciais acima (Ver Birkhoff e Fried [1979] para maiores detalhes) :

$$h^2 \ u_{xx}(x,y) \cong u(x-h, y) - 2u(x,y) + u(x+h, y)$$

$$h^2 \ u_{yy}(x,y) \cong u(x, y-h) - 2u(x,y) + u(x, y+h) ,$$

onde h é o comprimento de cada divisão do reticulado.

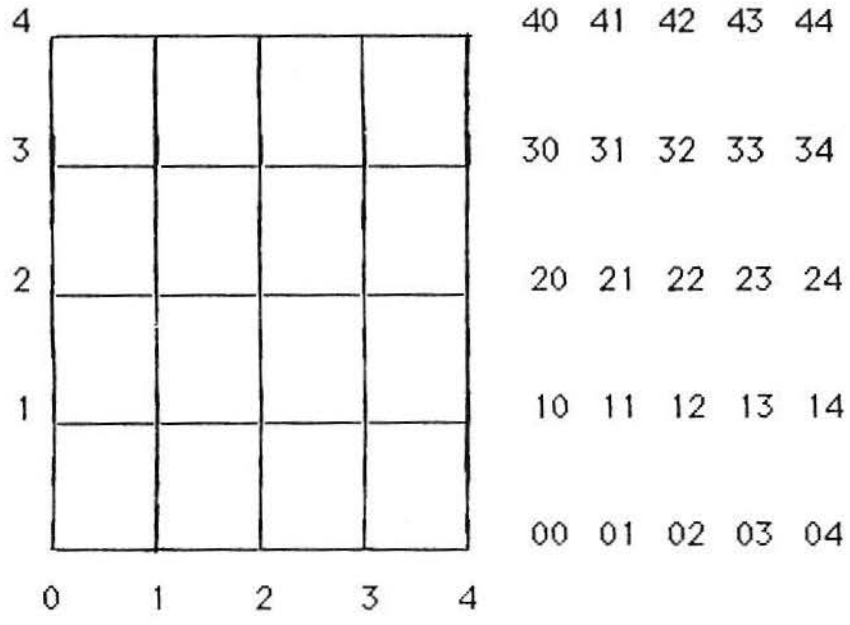
Substituindo estas expressões na equação diferencial, obtemos uma discretização do problema, cuja solução é uma aproximação para a solução u do problema contínuo.

Temos, então, que a equação diferencial inicial é transformada na seguinte expressão (u foi substituído por v que é a sua aproximação):

$$4v(x,y) - v(x-h,y) - v(x+h,y) - v(x,y-h) - v(x,y+h) = 0$$

Além disso, como queremos que v seja uma aproximação para u e u = g na fronteira de G, devemos ter v = g na fronteira. Com isto, temos os valores de v nos nodos de fronteira, faltando calcular nos nodos interiores. Pela expressão dada para v , verificamos que o valor em cada nodo interior depende do valor dos quatro nodos que o circundam. Numerando os nodos de acordo com as linhas e colunas nas quais eles se encontram, obtemos para cada nodo uma expressão e o conjunto destas expressões pode ser escrito na forma de um sistema de equações lineares com incógnita v : **Av = b**

Vejamos como exemplo, um reticulado dividido por 3 linhas interiores (verticais e horizontais) o que nos dá 9 nodos interiores para os quais precisamos calcular v e numerados da seguinte maneira (ver desenho) :



Em termos de índices, a expressão dada para v fica:

$$4\,v_{ij} - v_{i-1,j} - v_{i+1,j} - v_{i,j+1} - v_{i,j-1} = 0$$

Daí, temos:

$$4\,v_{11} - v_{01} - v_{21} - v_{12} - v_{10} = 0$$

$$4\,v_{12} - v_{02} - v_{22} - v_{13} - v_{11} = 0$$

$$4\,v_{13} - v_{03} - v_{23} - v_{14} - v_{12} = 0$$

$$4\,v_{21} - v_{11} - v_{31} - v_{22} - v_{20} = 0$$

$$4\,v_{22} - v_{12} - v_{32} - v_{23} - v_{21} = 0$$

$$\cdots \qquad \cdots$$

$$4\,v_{33} - v_{23} - v_{43} - v_{34} - v_{32} = 0$$

como $v_{ij} = g_{ij}$ ($= g(x_i,y_i)$) para i=0 ou j=0 temos que as equações acima nos dão o seguinte sistema:

$$Av = b$$

onde b é formado por somas dos elementos v_{ij} para i=0 ou j=0

v é formado pelas incógnitas v_{ij} para $i,j \neq 0$:

$$v = (v_{11} , v_{12} , v_{13} , v_{21} , \ldots , v_{32} , v_{33})^t$$

e A é a matriz abaixo:

$$A = \begin{bmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{bmatrix}$$

De modo geral, se formamos um reticulado com n linhas verticais e n linhas horizontais, teremos n^2 nodos interiores. Se escrevemos as n^2 expressões para estes nodos, obtemos um sistema do mesmo tipo Av = b , onde v é dado pelo vetor :

$$v = (v_{11}\ v_{12} \ldots v_{1n}\ v_{21} \ldots v_{2n} \ldots v_{n1} \ldots v_{nn})^t$$

e A tem o formato geral abaixo:

$$A = \begin{bmatrix} X & -I & 0 & & \ldots & 0 \\ -I & X & -I & 0 & \ldots & 0 \\ 0 & -I & X & -I & \ldots & 0 \\ & & . & . & & 0 \\ 0 & & -I & X & -I & \\ 0 & & 0 & -I & X & \end{bmatrix}$$

onde X é dado por:

$$X = \begin{bmatrix} 4 & -1 & 0 & & & 0 \\ -1 & 4 & -1 & 0 & & 0 \\ 0 & -1 & 4 & -1 & 0 & 0 \\ . & & . & . & . & \\ . & & . & . & . & \\ . & & & . & . & . \\ 0 & 0 & 0 & -1 & 4 & -1 \\ 0 & 0 & & 0 & -1 & 4 \end{bmatrix}$$

Observamos, com isto, que a matriz A obtida é uma matriz tridiagonal por blocos e os blocos X que a formam são também matrizes tridiagonais. Portanto, se tivermos algoritmos especialmente desenvolvidos para matrizes deste tipo (tridiagonais), poderemos solucionar a equação de Laplace, obtendo uma aproximação para a solução através do método descrito acima.

Existem diversos métodos desenvolvidos para resolver sistemas Ax = b, onde A é uma matriz tridiagonal. Stone [1973] apresenta um método que utiliza a técnica de "recursive doubling" o qual já foi discutido em parte no capítulo 2 e que apresenta um número de passos da ordem de $\log_2 n$ se utilizarmos n processadores (n é a dimensão do sistema). Para maiores detalhes ver a referência citada. Stone [1975] compara os algoritmos: "cyclic odd-even reduction" (redução cíclica par-ímpar), "recursive doubling" e algoritmo de Buneman fazendo a contagem das operações aritméticas envolvidas em cada um.

Todos os algoritmos podem ser resolvidos em um número de passos da ordem de $\log_2 n$, por isso aqui é importante contar o número de operações em cada passo para que esta comparação possa ser feita. Wang [1981] apresenta um outro método chamado de método da partição. Ele apresenta um número de operações vetoriais um pouco maior do que os métodos citados acima, no entanto, o número de operações escalares é bem menor do que o utilizado no "recursive doubling" e bastante semelhante ao de redução cíclica. Por este motivo, a contagem do número de operações não permite que tiremos conclusões definitivas e para podermos comparar os dois algoritmos é necessário que façamos uma análise do problema de organização e manipulação de dados em especial quando sistemas de grandes dimensões estão sendo considerados. Wang descreve também, ocasiões em que o seu método é preferível aos demais. Além destes, temos Traub [1973] que apresenta uma solução iterativa para o sistema tridiagonal baseado no algoritmo de decomposição LR e de Gauss seqüencial desenvolvido especialmente para sistemas deste tipo. Este método será apresentado a seguir. Descreveremos o algoritmo de Gauss em forma matricial, utilizando a decomposição da matriz A em A = LR, onde L é triangular inferior e R triangular superior e a partir desta decomposição calcular a solução x. Partimos do algoritmo seqüencial existente e o transformamos para processamento em paralelo através do método de iteração vetorial (cap. II). Veremos inicialmente como são os algoritmos de Gauss e de decomposição LR seqüenciais na forma matricial. Em seguida, veremos como são transformados para tratamento em paralelo.

Algoritmo para decomposição LR seqüencial

Partindo de A, matriz tridiagonal, desejamos decompô-la em A = LR, onde L é matriz triangular inferior e R é triangular superior. Podemos considerar A com diagonal unitária (se assim não for podemos pré-multiplicá-la por $D = diag(a_{11}^{-1}\ a_{22}^{-1} \ldots\ a_{nn}^{-1})$).

Temos que A é imediatamente decomposta em $A = A_L + I + A_R$ e, querendo que A = LR temos $A = A_L + I + A_R = LR$

Sendo $A = \begin{bmatrix} 1 & a_{12} & \ldots & & \\ a_{21} & 1 & a_{23} & \ldots & \\ 0 & a_{32} & 1 & \ldots & \\ & & & & \\ 0 & 0 & \ldots & a_{n-1n-2} & 1 & a_{n-1n} \\ & & & & a_{nn-1} & 1 \end{bmatrix}$

Temos que

$$A_L = \begin{bmatrix} 0 & \ldots & & \\ a_{21} & 0 & \ldots & \\ \ldots & a_{32} & 0 & \ldots \\ 0 & 0 & \ldots & 0 \\ & & a_{nn-1} & 0 \end{bmatrix}$$

$$A_R = \begin{bmatrix} 0 & a_{12} & \ldots & \\ 0 & 0 & a_{22} & \ldots \\ \ldots & 0 & 0 & \ldots \\ 0 & 0 & \ldots & a_{n-1n} \\ & & 0 & 0 \end{bmatrix}$$

Como L deve ser triangular inferior e R triangular superior, podemos ter (a fim de facilitar os cálculos):

$$L = A_L + D$$

$$R = I + J$$

Onde : D é matriz diagonal,

A_L e I são como acima e

J tem a mesma estrutura de A_R, ou seja, tem termos apenas na superdiagonal.

Teremos então

$$A = LR = (A_L + D)(I + J) = A_L + A_L J + D + DJ$$

Como nas matrizes acima temos que $A_L J$ e D são diagonais, DJ tem a mesma estrutura de J, e lembrando que $A = A_L + I + A_R$, devemos ter:

(1) $A_L J + D = I$

(2) $DJ = A_R$

De (1) temos :

(1') $D = I - A_L J$

Substituindo em (2) obtemos

(2') $(I - A_L J)J = A_R$

Daí, para resolver o sistema Au = v temos

$$Au = LRu = (A_L + D)(I+J)u = v$$

Chamando $(I + J)u = f$ o sistema acima fica

$$(3) \quad Au = (A_L + D)f = v$$

Substituindo D por sua expressão em (1') transformamos (3) em

$$(3') \quad Au = (A_L + I - A_L J)\, f = v \quad \text{ou}$$

$$(4) \quad Au = (\, I + A_L(I-J))\, f = v$$

Para facilitar a escrita de índices reescrevamos A, L e R como abaixo:

$$A = \begin{bmatrix} 1 & s_1 & & & 0 \\ t_2 & 1 & s_2 & & \\ 0 & t_3 & 1 & s_3 & \\ & \ddots & \ddots & \ddots & \\ & & t_{m-1} & 1 & s_{m-1} \\ 0 & & & t_m & 1 \end{bmatrix}$$

$$L = \begin{bmatrix} d_1 & 0 & & & \\ t_2 & d_2 & 0 & & \\ 0 & t_3 & d_3 & 0 & \\ & \ddots & \ddots & \ddots & \\ & 0 & t_{m-1} & d_{m-1} & 0 \\ & & 0 & t_m & d_m \end{bmatrix}$$

Já que L tem os elementos da subdiagonal iguais ao de A

$$R = \begin{bmatrix} 1 & e_1 & 0 & & \\ 0 & 1 & e_2 & 0 & \\ 0 & 0 & 1 & e_3 & 0 \\ & & & 1 & e_{m-1} \\ & & & & 1 \end{bmatrix}$$

A decomposição LR descrita antes nos diz que :

(1') $I - A_L J = D$

(2') $(I - A_L J)\, J = A_R$

Escrevendo estas matrizes e efetuando os cálculos indicados nas expressões acima, obtemos em termos de componentes as seguintes relações

de (1'):

(5) $d_1 = 1$

(6) $1 - t_k e_{k-1} = d_k \qquad k=2,...,m$

de (2'):

(7) $e_1 = s_1$

(8) $(1 - t_k e_{k-1})\, e_k = s_k$ ou $d_k\, e_k = s_k$ e daí:

(9) $e_k = s_k / d_k \quad k = 2,...,m-1$

onde $d_k = 1 - t_k e_{k-1} \quad k = 2,...,m$

Algoritmo de Gauss seqüencial

Tendo obtido a decomposição LR podemos utilizá-la para calcular a solução u do sistema Au = v.

Tínhamos a seguinte equação:

(2') $(I - A_L J)J = A_R$

Para obtermos o valor de u devemos calcular a solução dos sistemas:

(4) $(I + A_L(I-J)) f = v$

(5) $(I + J) u = f$

Obtidas as componentes de J através da expressão (2'), calculamos em seguida as componentes de f através de (4) e daí estamos aptos a obter os valores de u.

De (7) e (9) tínhamos que as componentes de J eram dadas por:

(7) $e_1 = s_1$

(9) $e_k = s_k / (1 - t_k e_{k-1})$ $\quad k = 2,...,m-1$

A partir da expressão (4) obtemos os seguintes valores para as componentes de f:

(10) $f_1 = v_1$

(11) $f_k = (v_k - t_k f_{k-1}) / (1 - t_k e_{k-1})$ $\quad k = 2,\ldots,m$

(pois $I - A_L J = \text{diag}\,(1,\ 1-t_2e_1,\ldots\ 1-t_ke_{k-1},\ldots,1-t_me_{m-1})$ e daí,

$$(I - A_L J + A_L)\, f = \begin{bmatrix} 1 & 0 & & & & \\ t_2 & 1-t_2e_1 & 0 & & & \\ 0 & t_3 & 1-t_3e_2 & 0 & & \\ & \ddots & \ddots & & & \\ & & t_k & 1-t_ke_{k-1} & & \\ & & & \ddots & \ddots & \\ & & & & t_m & 1-t_me_{m-1} \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_k \\ \vdots \\ f_m \end{bmatrix}$$

$f_1 = v_1$; $t_2 f_1 + 1-t_2e_1 f_2 = v_2$; e em geral $t_k f_{k-1} + 1-t_ke_{k-1} f_k = v_k$:

$1-t_ke_{k-1} f_k = (v_k - t_k f_{k-1}) / (\ 1-t_ke_{k-1})$)

E finalmente de (5) obtemos as componentes u_k:

(12) $u_m = f_m$

(13) $u_k = f_k - e_k u_{k+1}$ $\quad k = m-1, m-2, \ldots, 1$

Pelas expressões obtidas para e_k , f_k e u_k, verifica-se claramente que este é um algoritmo inerentemente seqüencial, pois a obtenção de um valor depende dos k-1 anteriores. Resta saber como adaptar este algoritmo ao caso paralelo.

Algoritmo LR paralelo

O algoritmo seqüencial existente é convertido em paralelo através do método de iteração vetorial visto em (2.3). A técnica consiste em transformarmos as expressões que possuímos em expressões iterativas, permitindo que uma iteração seja calculada em paralelo. Partimos da expressão (2') e supondo $J^{(0)}$ conhecido, modificamos (2') para:

$$(I - A_L J^{(i-1)})J^{(i)} = A_R \quad i=1,2,\ldots,M$$

onde M indica o número da iteração a partir da qual não há mudanças à exatidão desejada.

Tendo obtido $J^{(M)}$, calculamos a matriz D a partir de (1'), modificando-na para:

$$(14) \quad D = I - A_L J^{(M)} .$$

Escrevendo os elementos não-nulos de $J^{(i)}$ como $e_k^{(i)}$, obtemos as seguintes expressões para J em termos de componentes:

Suponhamos $e_k^{(0)}$ dados para k=2,...,m-1 , temos por adaptação direta às expressões (7),(8),(9) encontradas anteriormente:

$$e_1^{(i)} = s_1 \qquad i=0,1,...,M$$

$$e_k^{(i)} = s_k / (1 - t_k e_{k-1}^{(i-1)}) \qquad i=1,2,...,M$$

$$k=2,3,...,m-1$$

e $d_1 = 1$

$$d_k = 1 - t_k e_{k-1}^{(M)} \qquad k=2,3,...,m$$

Algoritmo de Gauss paralelo

Sejam dados $J^{(0)}$, $f^{(0)}$ e $u^{(0)}$. Adaptando as expressões já vistas à forma iterativa, temos como no algoritmo LR:

$$(I - A_L J^{(i-1)})J^{(i)} = A_R \qquad i=1,2,...,M$$

Obtido $J^{(M)}$ estamos aptos a obter os $f^{(i)}$ através de:

$$(I - A_L J^{(M)})f^{(i)} = v - A_L f^{(i-1)} \qquad i=1,2,...,N$$

Finalmente, obtido $f^{(N)}$, podemos calcular $u^{(i)}$ utilizando a expressão:

$$u^{(i)} = f^{(N)} - J^{(M)}u^{(i-1)} \qquad i=1,2,...,P$$

onde N e P são os números de iterações necessárias para obtermos as exatidões de f e u desejadas.

Este algoritmo pode ser calculado em paralelo, pois uma vez obtidas as componentes de $J^{(i-1)}$, as componentes de $J^{(i)}$ podem ser calculadas simultaneamente.

Traub demonstra que estabelecidas certas condições para a norma de A_R e A_L ($\| A_L \| + \| A_R \| < 1$), o algoritmo de Gauss está bem definido e estabelece limites para as normas de $J^{(i)} - J$, $f^{(i)} - f$ e $u^{(i)} - u$ os quais dependem apenas dos erros iniciais, da condição estabelecida acima e da exatidão desejada. Ele mostra que podemos escolher os valores de P, M e N de modo que a cada iteração tenhamos uma redução na norma dos erros cometidos.

Considerações finais

Muitos algoritmos mais poderiam ter cido citados e uma análise mais pormenorizada de alguns itens ter sido feita. No entanto, estes detalhes podem ser encontrados nas referências citadas. Por exemplo, algoritmos pra matrizes triangulares em geral são encontrados em [2], [20], [21], [10] e [19], sendo que estes dois últimos fazem uma revisão bibliográfica sobre a resolução de sistemas lineares em computadores paralelos. Algoritmos para sistemas tridiagonais são encontrados em [17], [21], [23], [24], [25] e [26] e algoritmos para resolução de sistemas do tipo banda podem ser vistos em [4], [18] e [21].

Não foi muito abordado também o problema de estabilidade dos algoritmos por haver pouca literatura a respeito, assim como o problema de comunicação entre os processadores e transferência de dados que por serem assuntos mais complexos e demorados talvez tenha mais sentido estudá-los de maneira prática com a real implementação dos algoritmos. Por não ter sido possível esta implementação por falta de equipamento também não consideramos problemas de programação. Alguma discussão sobre este assunto em computadores da classe MIMD, pode ser encontrada em Karp [1987] . Problemas de armazenamento de matrizes, assim como o algoritmo de Gauss para estas diversas formas de acesso aos elementos da matriz podem ser vistos com mais detalhes no artigo de Dongarra et al [1984]. Heller [1978] e mais recentemente Ortega e Voigt [1985] fazem uma ampla revisão de literatura a respeito dos assuntos tratados, desde uma descrição breve dos tipos de computadores existentes, passando pelos diversos métodos para resolução de sistemas lineares de acordo com os diferentes tipos de matrizes existentes até a solução de equações diferenciais parciais que acreditamos ser a maior aplicação para o estudo da resolução de sistemas lineares.

Este estudo foi apenas o começo de um campo que ainda é muito vasto. Esperamos com este trabalho incentivar mais pessoas ao estudo de um assunto que ainda está no início em nosso país e que demonstra desde já ser de grande importância para a resolução de problemas práticos reais. Muito ainda deve ser feito, e só se conseguirá ter uma idéia precisa quando tivermos acesso a computadores paralelos e neles implementemos os algoritmos estudados, de modo que, na prática, verifiquemos o que ocorre com eles, podendo compará-los com o processamento seqüencial.

Referências bibliográficas

1. BOJANCZYK, A., "Optimal asynchronous Newton method for the solution of nonlinear equations", Journal of the ACM, 31(4:792:803), Outubro, 1984.

2. BORODIN, A., MUNRO, I., The computational complexity of algebraic and numeric problems, New York, Elsevier, 1975.

3. BRENT, R.P., "The parallel evaluation of arithmetic expressions in logarithmic time". In: TRAUB, J.F. (Ed.).Complexity of sequential and parallel numerical algorithms, pp.83-102, New York, Academic Press,1973.

4. CHEN, S.C., KUCK, D.J., SAMEH, A.H., "Practical parallel band triangular system solvers", ACM Transactions on Mathematical Software, V.4, No.3, pp.270-77, Setembro, 1978.

5. da CUNHA, R.D., "Algoritmos paralelos numéricos para o cálculo de zeros de funções não-lineares", CPGCC/UFRGS, Porto Alegre, Novembro, 1989.

6. DONGARRA, J.J., GUSTAVSON, F.G., KARP, A., "Implementing linear algebra algorithms for dense matrices on a vector pipeline machine", SIAM Review, V.26, No.1, pp.91-112, Janeiro, 1984.

7. EAGER, D.L., ZAHORJAN, J., LAZOWSKA, E.D., "Speedup versus efficiency in parallel systems", IEEE Transactions on Computers, 38(3:308-423), Março, 1989.

8. FRIED, I. Numerical solution of differential equations. New York, Academic Press, 1979.

9. GOLUB, G.H., VAN LOAN, C.F., Matrix computations. Baltimore, The Johns Hopkins University Press, 1983.

10. HELLER, D., "A survey of parallel algorithms in numerical linear algebra", SIAM review, V. 20, No. 4, pp.740-77, Outubro, 1978.

11. HOUSEHOLDER, A.S.,The theory of matrices in numerical analysis. New York, Dover, 1975.

12. HWANG, K., BRIGGS, F.A., Computer architecture and parallel processing, New York, McGraw-Hill, 1984.

13. JAMIESON, L.H, GANNON,D., DOUGLASS, R.J. (Eds)., The characteristics of parallel algorithms, Cambridge,The MIT Press , 1987.

14. KARP, A.H., "Programming for parallelism", IEEE Computer, pp.43-57, Maio, 1987.

15. KOGGE, P.M., "Parallel solution of recurrence problems", IBM Journal of Research and Development, 18(2:138-148), Março, 1974.

16. KUCK, D., MURAOKA, Y., "Bounds on the parallel evaluation of arithmetic expressions using associativity and commutativity", Acta Informatica, V.3, pp.203-16, Springer-Verlag, 1974.

17. LAMBIOTTE JR., J.J., VOIGT, R.G, "The solution of tridiagonal linear systems on the CDC STAR-100 computer", ACM Transactions on Mathematical Software, V.1, No.4, pp.308-29, Dezembro, 1975.

18. LAWRIE, D.H, SAMEH, A.H, "The computation and communication complexity of a parallel banded system solver", ACM Transactions on Mathematical Software, V.10, No. 2, pp.185-195, Junho, 1984.

19. ORTEGA, J.M., VOIGT, R.G., "Solution of partial differential equations on vector and parallel computers", SIAM Review, V.27, No.2, pp.149-239, Junho, 1985.

20. SAMEH, A.H., BRENT, R.P., "Solving triangular systems on a parallel computer", SIAM Journal on Numerical Analysis, V.14, No.6, pp.1101-13, Dezembro, 1977.

21. SCHENDEL, U., CONOLLY, B.W., Introduction to numerical methods for parallel computers, Chichester, Ellis Horwood Ltd., 1984.

22. STONE, H.S., "Problems of parallel computation". In:TRAUB, J.F. (Ed.)., Complexity of sequential and parallel numerical algorithms, pp.1-15, New York, Academic Press,, 1973.

23. STONE, H.S., "An efficient parallel algorithm for the solution of a

tridiagonal linear system of equations", Journal of the ACM, V.20, No.1, pp.27-38, Janeiro, 1973.

24. STONE, H.S., "Parallel tridiagonal equation solvers", ACM Transactions on Mathematical Software, V.1, No.4, pp.289-307, Dezembro, 1975.

25. TRAUB, J.F., "Iterative solution of tridiagonal systems on parallel or vector computers". In: TRAUB, J.F. (Ed.).Complexity of sequential and parallel numerical algorithms, pp.49-82, New York,Academic Press, 1973.

26. WANG, H.H., "A parallel method for tridiagonal equations", ACM Transactions on Mathematical Software, V.7, No.2, pp.170-83, Junho, 1981.

27. WILKINSON, J.H., Rounding errors in algebraic processes, Englewood Cliffs, Prentice-Hall, 1963.