Novos Algoritmos SIMD para Multiplicação de Matrizes no Hipercubo

Carlos Alberto Alonso Sanches

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO

PARA OBTENÇÃO DO GRAU DE MESTRE
EM
MATEMÁTICA APLICADA

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Siang Wun Song**

Durante a elaboração deste trabalho, o autor recebeu apoio financeiro do CNPq e FAPESP (processo 91/1646-0).

–São Paulo, Agosto de 1992–

aos
meus pais

Agradecimentos

Essas poucas palavras, apesar de estarem no início, são as últimas a serem escritas, e talvez as mais difíceis, pelo desejo de expressar-me adequadamente sem esquecimentos.

Entretanto, pela falta de familiaridade com as palavras e pelas limitações pessoais, injustiças e erros tornam-se inevitáveis.

À banca examinadora, que teve a incumbência de compreender, corrigir e julgar este trabalho.

Ao meu orientador, Prof. Dr. Siang Wun Song, que além do esforço por compreender e orientar este trabalho, mostrou-se disponível, reto e cordial.

Aos meus amigos e principalmente aos meus pais que, sem se preocuparem com a compreensão deste trabalho, sempre me compreenderam e me apoiaram.

Resumo

A principal contribuição deste trabalho é a elaboração de vários algoritmos originais para multiplicação de matrizes $n \times n$ no hipercubo de p processadores, sendo que dois destes superam, em termos de complexidade de tempo, os melhores algoritmos conhecidos, devidos a Dekel, Nassimi e Sahni. Eles apresentaram algoritmos de $O(n^\lambda/p^{\frac{\lambda-1}{2}})$, com $2 \leq \lambda < 3$ e $1 \leq p \leq n^2$, e $O(\log \frac{p}{n^2} + \frac{n^3}{p})$, com $n^2 \leq p \leq n^3$. O algoritmo MMM_1 apresentado neste trabalho é $O(\frac{n^2}{p^{2/3}} \log p + \frac{n^\lambda}{p^{\lambda/3}})$, com $1 \leq p \leq n^3$. Demonstra-se que MMM_1 é melhor para $1 \leq p \leq n^3/\log^3 n$.

Através do estudo de alguns outros trabalhos dos mesmos autores, pôde-se observar que o hipercubo é encarado por eles de uma maneira interessante, a qual chamamos de **visualização matricial**. A principal vantagem dessa visualização é sugerir uma idéia geométrica do hipercubo, e ao mesmo tempo aproveitar a presença de vários sub-hipercubos nesta estrutura.

Com a utilização das **operações básicas de comunicação**, ganhou-se clareza, simplicidade e concisão nos algoritmos, que são descritos segundo uma nova formalização introduzida neste trabalho.

Nós também apresentamos outros algoritmos originais para multiplicação de matrizes no hipercubo segundo o paradigma da "divisão-e-conquista" que gastam espaço extra constante, e para multiplicação de matriz por vetor.

Abstract

The main contribution of this work is the elaboration of several original algorithms for $n \times n$ matrix multiplication on the hypercube of p processors, two of which outperform, in terms of time complexity, the best known algorithms by Dekel, Nassimi and Sahni. These authors presented algorithms of $O(n^{\lambda}/p^{\frac{\lambda-1}{2}})$, with $2 \leq \lambda < 3$ and $1 \leq p \leq n^2$, and $O(\log \frac{p}{n^2} + \frac{n^3}{p})$, with $n^2 \leq p \leq n^3$. The MMM_1 algorithm presented in this work is $O(\frac{n^2}{p^{2/3}} \log p + \frac{n^{\lambda}}{p^{\lambda/3}})$, with $1 \leq p \leq n^3$. It is shown that MMM_1 is better for $1 \leq p \leq n^3/\log^3 n$.

Through the study of some other works by the same authors, we could observe that the hypercube is viewed by them in an interesting manner, which we name as **matricial visualization**. The main advantage of this visualization is to suggest a geometric idea of the hypercube and at the same time make use of various sub-hypercubes in the structure.

By utilizing **basic communication operations**, we gain clarity, simplicity and concision in the algorithms, that are described through a novel formalization introduced in this work.

We also present some other original algorithms for matrix multiplication on the hypercube, according to the paradigm of "divide-and-conquer" that use constant extra space, as well as for vector-matrix multiplication.

Sumário

1 Considerações preliminares **1**
1.1 Notação para complexidades . . . 2
1.2 Modelos . . . 3
1.3 Operações básicas de comunicação . . . 5
1.4 Hierarquia das operações básicas . . . 6

2 O hipercubo e suas visualizações **7**
2.1 Definição . . . 7
2.2 Algumas propriedades topológicas . . . 7
2.3 Imersões . . . 8
2.4 Comparações . . . 10
2.5 Visualizações do hipercubo e concepção de algoritmos . . . 10
2.5.1 Notação para algoritmos SIMD . . . 10
2.5.2 Visualização linear . . . 11
2.5.3 Visualização matricial . . . 15

3 As operações básicas no hipercubo **23**
3.1 Alguns algoritmos ótimos . . . 24
3.1.1 Difusão e Acumulação Singular (DS e AS) . . . 24
3.1.2 Difusão e Acumulação Multi-nó (DM e AM) . . . 25
3.1.3 Dispersão e Recolhimento (Dp e Rc) . . . 28
3.1.4 Troca Completa (TC) . . . 28
3.2 Resultados e Comparações . . . 29
3.3 Alguns algoritmos SIMD . . . 29
3.3.1 Difusão Singular . . . 30
3.3.2 Acumulação Singular . . . 30
3.3.3 Difusão Multi-nó . . . 31
3.3.4 Complexidades . . . 32

4 Multiplicação de matrizes **33**
4.1 O problema e sua complexidade seqüencial . . . 34
4.2 Algoritmos paralelos básicos no hipercubo . . . 34
4.2.1 Classe N: com $p = n$ processadores . . . 34
4.2.2 Classe NN: com $p = n^2$ processadores . . . 35

4.2.3 Classe NNN: com $p = n^3$ processadores . . . 38
4.3 Os algoritmos de Dekel, Nassimi e Sahni . . . 40
4.3.1 Classe NNM: com $n^2 \le p \le n^3$ processadores . . . 40
4.3.2 Classe MM: com $1 \le p \le n^2$ processadores . . . 42
4.4 Situação atual . . . 44
4.5 Uma visão global . . . 44
4.5.1 Arranjo bi-dimensional . . . 44
4.5.2 Arranjo tri-dimensional . . . 45
4.6 As novas classes de algoritmos . . . 45
4.6.1 Classe MMM: com $1 \le p \le n^3$ processadores . . . 45
4.6.2 Classe NM: com $n \le p \le n^2$ processadores . . . 47
4.6.3 Classe M: com $1 \le p \le n$ processadores . . . 50
4.6.4 Algoritmo NNM_3: um outro algoritmo de NNM . . . 51
4.7 Resultados . . . 52
4.8 Comparações . . . 53
4.8.1 NM_1 versus NM_2 versus NM_3 . . . 53
4.8.2 MM_1 versus M_1 . . . 55
4.8.3 MMM_1 versus NM_3 . . . 55
4.8.4 MMM_1 versus MM_1 . . . 56
4.8.5 NNM_3 versus NNM_1 . . . 57
4.9 Melhores algoritmos . . . 59
4.10 Exemplos . . . 59
4.11 Algoritmos tipo "divisão-e-conquista" . . . 60
4.11.1 Algoritmo NN_{DC}: para $p = n^2$ processadores . . . 61
4.11.2 Algoritmo NNN_{DC}: com $p = n^3$ processadores . . . 62
4.11.3 Algoritmo NNM_{DC}: com $n^2 \le p \le n^3$ processadores . . . 64
4.11.4 Algoritmo MM_{DC}: com $1 \le p \le n^2$ processadores . . . 65
4.11.5 Algoritmo MMM_{DC}: com $1 \le p \le n^3$ processadores . . . 66
4.11.6 Comparações . . . 68

5 Multiplicação de matriz por vetor **69**
5.1 O problema . . . 69
5.2 Algoritmos paralelos conhecidos . . . 69
5.2.1 Algoritmo N: com $p = n$ processadores . . . 69
5.2.2 Algoritmo NN: com $p = n^2$ processadores . . . 70
5.3 Os novos algoritmos . . . 71
5.3.1 Algoritmo M: com $1 \le p \le n$ processadores . . . 71
5.3.2 Algoritmo MM: com $1 \le p \le n^2$ prodessadores . . . 71
5.4 Resultados . . . 72
5.5 Comparação . . . 72
5.6 Melhores algoritmos . . . 72

6 Comentários e perspectivas **75**

A Lista de símbolos **77**

Capítulo 1

Considerações preliminares

Ao iniciarmos esta dissertação, o objetivo que nos propusemos foi desenvolver algoritmos para máquinas paralelas com arquitetura do tipo **hipercubo** de dimensão d, também conhecido pelo nome de **d-cubo binário**.

O principal fator de motivação é o fato de o hipercubo ser uma rede de interconexão de processadores que se tem utilizado em muitas arquiteturas paralelas. Do ponto de vista de construção, a estrutura do hipercubo tem a vantagem de ser facilmente extensível ("escalabilidade"). O primeiro protótipo foi chamado *Cosmic Cube* [Sei85], composto por $2^6 = 64$ processadores *Intel 8088*, fabricado pelo *Caltech*. A empresa *Thinking Machines Corporation* produziu a *Connection Machine* [Hil85], com $2^{16} = 65536$ processadores agrupados em 4096 módulos ligados em um 12-cubo, enquanto a *Intel* fabricou o *iPSC/2*, com $2^7 = 128$ processadores *80386*, e uma série de outros hipercubos. Uma discussão sobre arquiteturas paralelas e suas perspectivas futuras pode ser encontrada em [Bel92].

Apesar da disponibilidade desses computadores paralelos, a sua efetiva utilização depende do projeto e desenvolvimento de algoritmos paralelos que, em geral, não são fáceis de conceber. Alguns trabalhos [DNS81, NaS81, NaS82] nos chamaram a atenção por introduzirem um modo interessante de visualizar o hipercubo. Esta técnica, que chamamos de **visualização matricial** ao longo de todo este texto, permite aproveitar uma série de propriedades da definição recursiva do hipercubo, constituindo-se em um instrumento muito útil para conceber algoritmos nesta topologia.

Estudando esta técnica, chegamos à conclusão de que pode ser melhor explorada com relação aos algoritmos para multiplicação de matrizes, e obtivemos novos algoritmos superiores aos já publicados por Dekel, Nassimi e Sahni [DNS81]. Eles apresentaram algoritmos de $O(n^\lambda/p^{\frac{\lambda-1}{2}})$, com $2 \leq \lambda < 3$ e $1 \leq p \leq n^2$, e $O(\log \frac{p}{n^2} + \frac{n^3}{p})$, com $n^2 \leq p \leq n^3$. O algoritmo MMM_1 apresentado neste trabalho é $O(\frac{n^2}{p^{2/3}} \log p + \frac{n^\lambda}{p^{\lambda/3}})$, com $1 \leq p \leq n^3$. Demonstra-se que MMM_1 é melhor algoritmo para $1 \leq p \leq n^3/\log^3 n$. Por esse motivo, o problema da multiplicação de matrizes tornou-se o tema dominante do nosso trabalho.

Neste capítulo inicial, depois de definirmos a notação das complexidades, descreveremos de uma maneira sucinta alguns dos modelos de computação paralela mais comumente utilizados na literatura. Esses modelos procuram, dentro do possível, refletir as máquinas reais.

Num sistema de computação paralela de memória distribuída, os processadores se comunicam através de troca de mensagens, conforme o algoritmo que implementam para resolver um determinado problema. Entretanto, comparando-se diversos algoritmos clássicos, pode-se observar que freqüentemente se repetem alguns tipos de comunicação entre os processadores, e que por isso recebem um cuidado especial. Esses tipos de comunicação serão chamados **operações básicas**, e suas descrições também estão neste capítulo.

No capítulo 2, definimos o grafo do hipercubo, e relacionamos algumas de suas principais propriedades, bem como possibilidades de imersões de outras topologias conhecidas. A partir disso, explicamos em que consiste a técnica das visualizações. Procuramos enumerar de um modo claro as propriedades que nos serão úteis adiante, e reescrevemos alguns algoritmos [NaS81, NaS82], aproveitando também as operações básicas.

O objetivo do terceiro capítulo é estabelecer hipóteses com relação às medidas de complexidade das operações básicas no hipercubo, para assim podermos descrever alguns de seus algoritmos, entre eles os do tipo SIMD que serão utilizados.

O capítulo 4 é o principal de todo o texto. Nele tratamos do problema da multiplicação de matrizes no hipercubo, descrevendo inicialmente os algoritmos já conhecidos e em seguida explicando como é possível a elaboração de vários originais. A partir das complexidades desses algoritmos, estabelecemos um critério de comparação para então concluirmos quais são os melhores. Dois dos novos algoritmos superam os conhecidos até agora. Por fim, apresentamos uma série de algoritmos originais, de espaço extra constante, para multiplicação de matrizes que se baseiam no método de "divisão-e-conquista".

O problema da multiplicação de matriz por vetor, pela sua semelhança com o anterior, pode ser resolvido empregando-se idéias análogas. A descrição de novos algoritmos para esse problema está no capítulo 5.

No último capítulo, tecemos alguns comentários sobre o trabalho realizado, onde procuramos explicar todo o seu desenvolvimento, e relatar alguns pontos que ainda necessitam de um maior estudo.

O apêndice A contém uma lista de símbolos usados, seus significados e o número da página de sua primeira ocorrência.

1.1 Notação para complexidades

Faremos aqui uma breve descrição das notações $O(.)$, $\Omega(.)$, $\Theta(.)$ e $o(.)$, que utilizaremos com freqüência nos cálculos das complexidades dos algoritmos.

Seja uma função $f : \mathrm{N} \rightarrow \mathrm{N}$. Definimos os seguintes conjuntos de funções:

$$\begin{aligned} O(f) &= \{g : \mathrm{N} \rightarrow \mathrm{N} \mid \exists\, n_0 \geq 0,\ \exists\, c > 0,\ \text{tal que } \forall\, n \geq n_0,\ g(n) \leq cf(n)\} \\ \Omega(f) &= \{g : \mathrm{N} \rightarrow \mathrm{N} \mid \exists\, n_0 \geq 0,\ \exists\, c > 0,\ \text{tal que } \forall\, n \geq n_0,\ g(n) \geq cf(n)\} \\ \Theta(f) &= O(f) \cap \Omega(f) \end{aligned}$$

Como é usual, escreveremos $g = O(f)$, $g = \Omega(f)$ e $g = \Theta(f)$ ao invés de $g \in O(f)$, $g \in \Omega(f)$ e $g \in \Theta(f)$, respectivamente.

Além disso, seguindo a definição de Wilf [Wil86], dadas duas funções $f(n)$ e $g(n)$, dizemos que $f(n) = o(g(n))$ se $\lim_{n\to+\infty} f(n)/g(n)$ existe e é igual a 0. Portanto, se temos dois algoritmos que gastam tempo $f(n)$ e $g(n)$, onde n é o tamanho da entrada, e $f(n) = o(g(n))$, sabemos que para todos os valores suficientemente grandes de n, o desempenho do primeiro algoritmo será garantidamente superior ao do segundo.

Um caso particular que aparece com freqüência no nosso trabalho é o seguinte:

$$\log^a n = o(n^b), \text{ para } a \geq 1 \text{ e } b > 0.$$

Isso pode ser provado com a aplicação da regra de L'Hospital, como mostramos abaixo. Fazemos a seguinte troca de variável: $n^b = m$, ou seja, $n = m^{1/b}$. c é uma constante.

$$\begin{aligned}
\lim_{n\to+\infty} \frac{\log^a n}{n^b} &= c \lim_{m\to+\infty} \frac{\ln^a m^{1/b}}{m} \\
&= \frac{c}{b^a} \lim_{m\to+\infty} \frac{\ln^a m}{m} \\
&= \frac{ca}{b^a} \lim_{m\to+\infty} \frac{\ln^{a-1} m(1/m)}{1} \\
&= \vdots \\
&= \frac{ca(a-1)\dots(a-\lfloor a \rfloor+1)}{b^a} \lim_{m\to+\infty} \frac{\ln^{a-\lfloor a \rfloor} m}{m} \\
&= \frac{ca(a-1)\dots(a-\lfloor a \rfloor+1)}{b^a} \lim_{m\to+\infty} \left(\frac{\ln m}{m}\right)^{a-\lfloor a \rfloor} \frac{1}{m^{\lfloor a \rfloor+1-a}} \\
&= 0.
\end{aligned}$$

1.2 Modelos

A computação paralela caracteriza-se pela presença de vários processadores trabalhando simultaneamente para resolver um mesmo problema. Quando esses processadores estão sincronizados e subordinados a um único controle, todos executando simultaneamente uma mesma instrução, apesar de possuírem dados diferentes, dizemos que neste caso o controle é centralizado. Na literatura utiliza-se a nomenclatura SIMD, que significa *Single Instruction Stream, Multiple Data Stream*. Por outro lado, quando não há esse controle central, e os processadores trabalham de modo independente, com sua própria unidade de controle, emprega-se o nome de MIMD, isto é, *Multiple Instruction Stream, Multiple Data Stream*.

Esta centralização pode ou não ocorrer com relação à memória. No caso em que há uma única memória, à qual todos os processadores têm acesso, dizemos que a memória é compartilhada (*shared*). Para que isso seja possível, é preciso estabelecer critérios para resolver conflitos de escrita ou leitura entre os processadores. O oposto ocorre quando cada processador tem sua própria memória, a qual chamamos então de memória distribuída.

Os processadores estão ligados entre si segundo uma topologia, a qual chamaremos de "rede de interconexão". Essa rede pode ser representada por um grafo, onde os vértices

são os processadores e as arestas são os canais de intercomunicação. Dessa forma, os processadores comunicam-se entre si através do envio de mensagens através desses canais.

Existem modelos com respeito ao modo como é feita a comunicação nesses canais segundo vários aspectos:

Capacidade de comunicação: Podem ser definidos três tipos:

- **1-port:** Durante uma comunicação, cada processador só pode utilizar um único canal por vez. Também é chamado de *processor-bound*, *whispering*, etc.
- **d-port:** Considerando que cada processador tenha até d canais, ele pode utilizá-los todos simultaneamente. Também é chamado de *link-bound*, *shouting*, etc.
- **k-port:** Cada processador pode utilizar somente k canais simultaneamente, $1 < k < d$. Também é chamado de *DMA-bound*.

Canais: Dados dois processadores p_i e p_j interconectados entre si, e supondo que o canal de comunicação que os une seja bi-direcional, este mesmo canal pode ser de dois tipos:

- **Half-duplex:** Somente uma mensagem por vez pode atravessar o canal, de p_i para p_j ou de p_j para p_i.
- **Full-duplex:** Duas mensagens podem atravessar simultaneamente o canal, uma em cada sentido.

Tempo de comunicação: O tempo T necessário para que uma mensagem chegue de um processador a outro através de um canal de comunicação que os interliga é modelado de diferentes modos na literatura. Fraigniaud e Lazard [FrL91] descrevem-nos da seguinte forma:

- **Modelo linear:** Através de experimentos, verifica-se que o tempo de comunicação depende do tamanho da mensagem enviada. Portanto, considera-se $T = \beta + L\tau$, onde β é o custo de *start-up* ("inicialização") e τ é o tempo de transmissão de uma mensagem de tamanho unitário. L é o tamanho da mensagem.
- **Modelo constante:** É feita a suposição de que o comprimento das mensagens é tão pequeno a ponto de considerar o tempo T como uma expressão constante. Além disso, também se supõe neste modelo que as mensagens podem ser quebradas e associadas[1] sem qualquer gasto extra de tempo.
- **Modelo segundo Bertsekas et al.:** Em [BOS91] e [BeT89], um outro modelo é sugerido, diferindo do modelo constante por não permitir quebra ou associação de mensagens. Os autores comentam que esse modelo pode ser comparado com o linear assumindo $\beta = 0$ e $\tau = 1$, e considerando as mensagens com tamanho unitário.

Há ainda outros modelos menos importantes para tempo de comunicação relacionados de uma maneira não exaustiva em [FrL91].

[1] Por associação de mensagens entende-se justaposição, ou seja, se obtém uma nova mensagem cujo tamanho é a soma dos tamanhos iniciais.

1.3 Operações básicas de comunicação

Supondo que a máquina possua memória distribuída, é necessária a troca de informações entre os processadores, e surge então o problema de comunicação entre eles. Apesar do acesso de um processador a outro depender obviamente da topologia de interconexão, algumas operações aparecem freqüentemente no estudo de algoritmos. Infelizmente não há uma nomenclatura padronizada para essas operações, e portanto teremos que adotar uma, baseada em [BeT89], em detrimento das demais.

Difusão Singular (DS): Envio de uma mesma mensagem de um dado vértice a todos os demais vértices da rede. Outros nomes: *broadcasting, single node broadcast, one to all*, etc.

Difusão Multi-nó (DM): Cada vértice envia uma mesma mensagem para todos os outros vértices. Corresponde a fazer simultaneamente uma difusão singular a partir de cada vértice. Outros nomes: *gossipping, multinode broadcast, all to all, total exchange*, etc.

Dispersão (Dp): Um determinado vértice envia para cada vértice da rede uma mensagem distinta. Outros nomes: *scattering, single node scatter, personalized one to all, distributing*, etc.

Troca Completa (TC): Cada vértice envia mensagens distintas para cada um dos outros vértices da rede. Corresponde a fazer simultaneamente uma dispersão em cada vértice da rede. Outros nomes: *multi-scattering, personalized all to all, complete exchange, data transposition, multiscatter-gather*, etc.

Dentre essas quatro operações básicas de comunicação, as três primeiras possuem operações duais:

Acumulação Singular (AS): Um determinado vértice da rede recebe uma mensagem de cada um dos outros vértices, mas em cada vértice intermediário do percurso, as mensagens recebidas são "combinadas", dando origem a uma única do mesmo tipo e tamanho. O tempo de "combinação" não é maior que o da transmissão entre dois vértices. Corresponde ao inverso da difusão singular. Outros nomes: *single node accumulation, convergecasting*, etc.

Acumulação Multi-nó (AM): Cada vértice envia uma mensagem para cada um dos outros vértices da rede, e as mensagens destinadas a um mesmo vértice são "combinadas" durante o percurso. Corresponde a fazer simultaneamente uma acumulação singular em cada vértice. Outro nome: *multinode accumulation.*

Recolhimento (Rc): Um determinado vértice recebe de cada vértice da rede uma mensagem distinta. Pode-se observar que a seqüência de transmissões no recolhimento é o inverso da seqüência da dispersão. Outros nomes: *gathering, single node gather*, etc.

Podemos ter uma melhor visão comparativa das operações básicas através do quadro abaixo, que descreve os envios iniciais e os recebimentos finais. p é o número total de vértices, $x \Rightarrow y$ significa que x vértices distintos enviam cada um y mensagens distintas, e $x \Leftarrow y$ significa que x vértices distintos recebem cada um y mensagens distintas.

OPERAÇÕES	ENVIOS	RECEBIMENTOS
DS	$1 \Rightarrow 1$	$(p-1) \Leftarrow 1$
DM	$p \Rightarrow 1$	$p \Leftarrow (p-1)$
Dp	$1 \Rightarrow (p-1)$	$(p-1) \Leftarrow 1$
TC	$p \Rightarrow (p-1)$	$p \Leftarrow (p-1)$
AS	$(p-1) \Rightarrow 1$	$1 \Leftarrow 1$
AM	$p \Rightarrow (p-1)$	$p \Leftarrow 1$
Rc	$(p-1) \Rightarrow 1$	$1 \Leftarrow (p-1)$

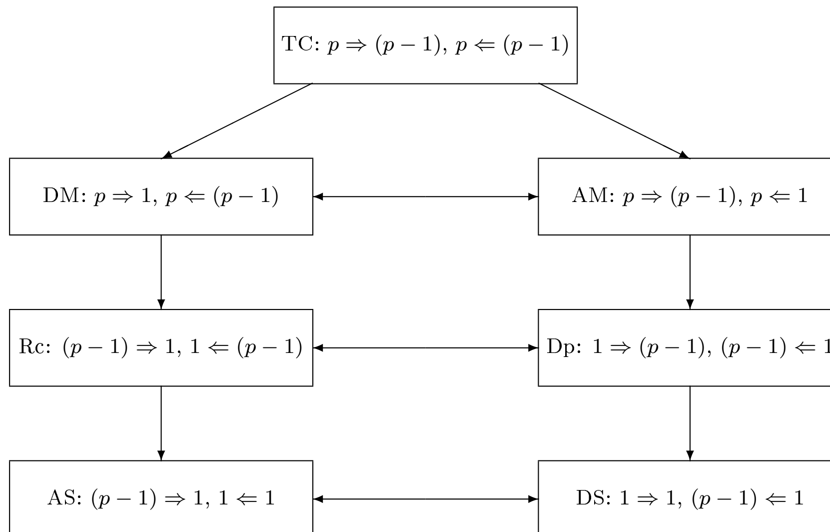
1.4 Hierarquia das operações básicas

Em [BeT89] as operações básicas estão dispostas segundo uma hierarquia de complexidade. É importante observar que tal hierarquia independe da topologia considerada.

Ela baseia-se nos seguintes fatos:

- DS e AS são casos particulares de Dp e Rc, respectivamente;
- Dp é AM onde um único vértice envia mensagens;
- Rc é DM considerando apenas o recebimento de mensagens em um vértice;
- DM e AM são casos particulares de TC.

Na figura abaixo, cada quadro corresponde a uma operacão básica. Sendo A e B dois quadros, $A \rightarrow B$ significa que um algoritmo que resolve A também pode resolver B, e o tempo ótimo para resolver A não é menor do que o tempo ótimo para resolver B. Se $A \leftrightarrow B$, então as operações A e B são consideradas duais.



Capítulo 2

O hipercubo e suas visualizações

O objetivo deste capítulo é descrever rapidamente algumas características da topologia do hipercubo, e apresentar em seguida o que chamamos de suas "visualizações", que nos parecem ser bons instrumentos para a concepção de algoritmos nesta rede de interconexão.

2.1 Definição

Um hipercubo de dimensão d possui $p = 2^d$ vértices. Cada vértice é identificado por um endereço de d *bits*. Cada *bit* do endereço corresponde a uma dimensão do hipercubo. Vamos usar a própria representação binária do endereço $e = e_{d-1} \dots e_1 e_0$, onde cada e_i é 0 ou 1, para denotar o vértice e. Este vértice é adjacente a todos os vértices cujos endereços diferem de e em exatamente um único *bit*, isto é, o vértice e é adjacente aos vértices em

$$\{e \oplus 2^j \mid 0 \leq j < d\}$$

onde $\oplus$ representa o operador **ou-exclusivo**.

Cada vértice de um hipercubo de p vértices é adjacente a $d = \log p$ outros vértices. Temos portanto um total de $d\ 2^{d-1}$ arestas ou ligações.

Chamaremos de $H(d)$ o hipercubo de dimensão d com $p = 2^d$ vértices.

2.2 Algumas propriedades topológicas

Saad e Schultz [SaS88] fazem uma apologia do hipercubo do ponto de vista topológico, enumerando uma série de vantagens desse grafo como rede de interconexão de processadores. Recolhemos abaixo os principais resultados apresentados, mas não repetiremos as demonstrações, por fugir do escopo deste trabalho.

- O hipercubo possui uma estrutura recursiva, isto é, $H(0)$ é um único vértice, e $H(d)$ é formado por dois $H(d-1)$, chamados H_1 e H_2, mais 2^{d-1} ligações: cada par de vértices, um de H_1 e outro de H_2, que tenham o mesmo endereço (de $d-1$ *bits*) são ligados entre si. Além disso, todos os vértices recebem um novo endereço: cada vértice de H_1 que tinha endereço $e = e_{d-2} \dots e_1 e_0$ passa a ter endereço $0e_{d-2} \dots e_1 e_0$, enquanto o mesmo vértice e de H_2 recebe como novo endereço $1e_{d-2} \dots e_1 e_0$.

- Há d modos diferentes de se dividir um hipercubo $H(d)$ em dois $H(d-1)$. O inverso do que está descrito no item anterior é um desses d modos, referente ao *bit* mais significativo dos endereços. O mesmo pode ser feito com qualquer um dos outros $d-1$ *bits*.

- Há $d!\ 2^d$ diferentes modos de numerar os 2^d vértices de um $H(d)$.

- Dados dois vértices v_1 e v_2 adjacentes de um hipercubo, temos que os vértices adjacentes a v_1 e os vértices adjacentes a v_2 são conectados um a um.

- Em qualquer hipercubo não há ciclos de comprimento ímpar.

- Sendo o diâmetro de um grafo a maior distância entre quaisquer dois de seus vértices, seu valor para o $H(d)$ é d.

- O seguinte teorema é apresentado: Um grafo $G = (V, E)$ é um $H(d)$ se e somente se:

  - $|V| = 2^d$;
  - Cada vértice de G tem grau d;
  - G é conexo;
  - Para quaisquer dois vértices v_1 e v_2 adjacentes de G, os vértices adjacentes a v_1 e os vértices adjacentes a v_2 estão ligados um a um.

  Esse teorema, entre outras coisas, permite o reconhecimento de um hipercubo através da verificação de somente quatro condições.

- A mínima distância entre dois vértices com endereços v_1 e v_2 de um hipercubo é igual ao número de *bits* em que diferem, isto é, é igual à distância de Hamming entre eles: $Ham(v_1, v_2)$.

- Sejam v_1 e v_2 dois vértices de $H(d)$ tais que $Ham(v_1, v_2) < d$. Portanto, há $Ham(v_1, v_2)$ caminhos disjuntos (que não possuem vértice em comum) de comprimento $Ham(v_1, v_2)$ entre v_1 e v_2, e $d-$ $Ham(v_1, v_2)$ caminhos também disjuntos de comprimento $Ham(v_1, v_2)+$ 2.

  Essa propriedade permite uma eficiente paralelização da transmissão de mensagens entre dois vértices do hipercubo.

2.3 Imersões

Devido às diversas topologias existentes, um conceito importante é o de imersão de um grafo em outro (*embedding*). A imersão nada mais é do que um mapeamento um a um entre os vértices do grafo a ser imerso e os vértices do grafo hospedeiro, que satisfaz a propriedade de que dois vértices adjacentes no grafo imerso estão relacionados a outros dois vértices do grafo hospedeiro que possuam algum caminho entre si.

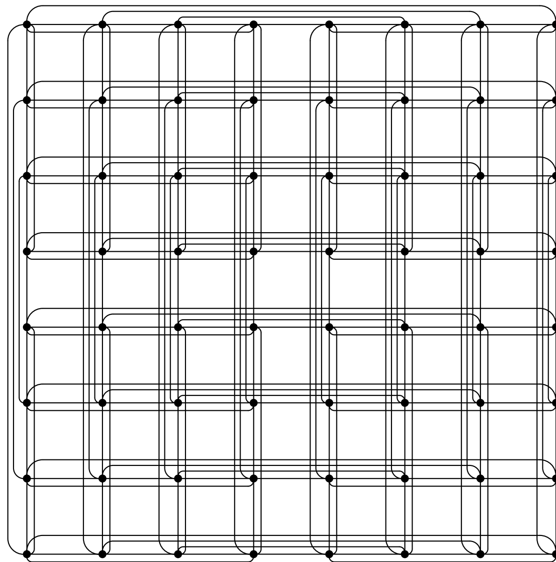
Duas razões relatadas em [SaS88] mostram a importância das imersões:

- Supondo que haja um algoritmo desenvolvido em uma determinada arquitetura, e a topologia dessa arquitetura possa ser imersa em uma outra, um algoritmo na segunda arquitetura pode ser obtido sem grande esforço.

- Um programa pode ter uma estrutura que é própria para ser resolvida sob uma determinada topologia. Se se conhece a imersão dessa topologia em uma outra, fica mais simples a resolução do mesmo problema na segunda topologia.

Uma imersão pode ser considerada boa se cada par de vértices adjacentes do grafo a ser imerso é levado a um par de vértices vizinhos no grafo hospedeiro. O hipercubo permite imersões deste tipo para anéis e grades, conforme descrito em [SaS88]. Em [Son91] é apresentado um método recursivo de imersão: de uma grade m-dimensional de r vértices em cada direção em um $H(d)$, onde $2^d = r^m$.

Na figura abaixo, através da aplicação do método descrito em [Son91], os vértices de um $H(6)$ estão dispostos de forma a permitir a visualização da grade toroidal 8×8 imersa nesta estrutura:



Monien e Sudborough [MoS88] relatam uma série de resultados conhecidos sobre imersões em algumas das topologias mais comuns. Utilizando duas medidas de qualidade (*dilation* e *expansion*), mostram como é possível imergir no hipercubo grafos como árvores binárias, grades, *X-trees*, embaralhadores-perfeitos (*shuffle-exchange*), De Brujn, entre outros.

2.4 Comparações

Na tabela abaixo estão alguns parâmetros importantes para uma rede de interconexão de processadores, e considerando algumas topologias conhecidas: anel[1], árvore binária balanceada, grade d-dimensional toroidal simétrica e hipercubo ($H(d)$). Conectividade é o número mínimo de vértices que precisam ser retirados para que o grafo se torne desconexo. Supõe-se que todas as topologias tenham p vértices.

TOPOLOGIAS	ANEL	ÁRVORE	GRADE	HIPERCUBO
Diâmetro	$p/2$	$2\log_2 p$	$d(p^{1/d} - 1)/2$	$\log_2 p$
Conectividade	2	1	$2d$	$\log_2 p$
Imersões	—	—	anel, árvore	anel, árvore, grade

2.5 Visualizações do hipercubo e concepção de algoritmos

Ao se estudar os trabalhos descritos em [NaS81], [NaS82] e [DNS81], pode-se observar que o hipercubo é encarado de uma maneira interessante. Diremos que os autores "visualizam" o hipercubo segundo diferentes ângulos, e pelo fato de esta topologia ter uma estrutura essencialmente recursiva, aproveitam a presença de vários sub-hipercubos nessas visualizações para a concepção de diversos algoritmos.

Esses trabalhos entretanto não dão uma perspectiva *top-down* dos algoritmos descrevendo-os a partir das propriedades das visualizações, e portanto sua compreensão é mais árdua. Chamaremos tais visualizações de **linear** e **matricial** para facilitar futuras referências, e daremos aqui informalmente uma descrição de suas principais características. Também mostraremos alguns exemplos de algoritmos SIMD, vários apresentados nesses mesmos trabalhos, que ficam mais simples de serem entendidos quando se conhecem essas visualizações. Antes apresentaremos uma notação para a especificação desses algoritmos SIMD. Apesar de concentrarmos nossa atenção em algoritmos SIMD, as visualizações também podem auxiliar a elaboração de algoritmos MIMD.

2.5.1 Notação para algoritmos SIMD

Seguiremos uma notação baseada em [NaS81] e [NaS82]:

- $R(i)$, $0 \leq i < p$, é um registrador R no vértice i de $H(d)$. Quando o registrador R do vértice i estiver vazio, $R(i) =$ null. Se R for um vetor, $R[j](i)$ é a j-ésima posição do vetor R do vértice i.
- i_b é o b-ésimo *bit* da representação binária de i.
- $i_{b_2:b_1}$ corresponde aos *bits* $b_2, \ldots, b_1$ da representação binária de i.
- Seja $i_{d-1} \ldots i_0$ a representação binária de i, para $i \in [0, p-1]$. $i^{(b)}$ é o número cuja representação binária é $i_{d-1} \ldots i_{b+1}\bar{i_b}i_{b-1} \ldots i_0$, onde $\bar{i_b}$ é o complemento de i_b, $0 \leq b < d$. No hipercubo, o vértice i é conectado com $i^{(b)}$, $0 \leq b < d$.

[1] Anel é grade toroidal 1-dimensional.

Um comando de atribuição envolvendo dois registradores de vértices de um hipercubo atribui o valor contido no registrador do primeiro ao registrador do outro. Consideremos os seguintes tipos de atribuições em alguns dos algoritmos que serão descritos:

$:=$ Atribuição que é utilizada somente entre registradores de um mesmo vértice. Não necessita de nenhuma comunicação entre vértices.

$\leftarrow$ Atribuição que requer uma comunicação entre dois vértices vizinhos. Por exemplo, $R(i) \leftarrow R(i^{(b)})$ é válido no hipercubo.

$\leftrightarrow$ Troca que requer uma comunicação entre dois vértices vizinhos. Por exemplo, $R(i) \leftrightarrow R(i^{(b)})$ no hipercubo.

Para arquiteturas SIMD, o algoritmo é o mesmo em todos os processadores, e cada comando pode ou não ser executado por um determinado processador. Em um comando, a habilitação de um vértice é feita através de uma condição colocada entre parênteses.

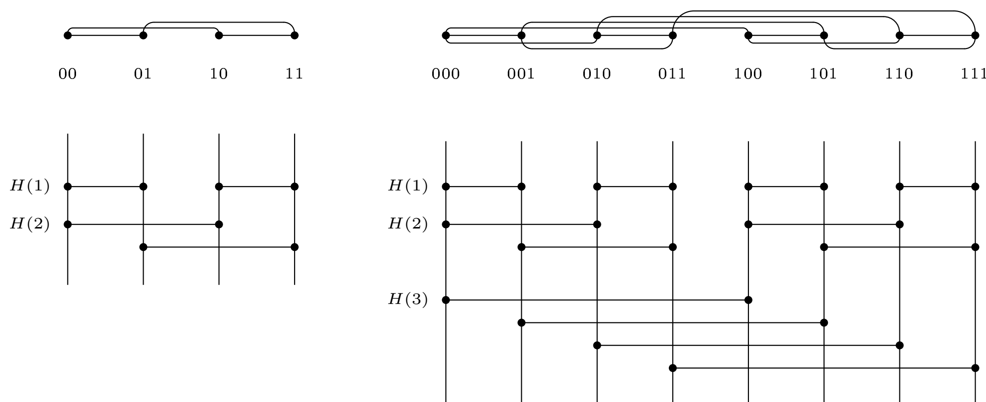
Exemplo:

$R(i) \leftarrow R(i^{(b)}) + 1, \ (i_4 = 1)$: somente é executado para vértices com endereço cuja representação binária contém 1 no *bit* 4.

2.5.2 Visualização linear

Esta visualização é muito simples, e baseia-se na definição recursiva do hipercubo, onde seus vértices estão dispostos de uma forma linear e seqüencial. Veremos mais adiante que é um caso particular da visualização matricial.

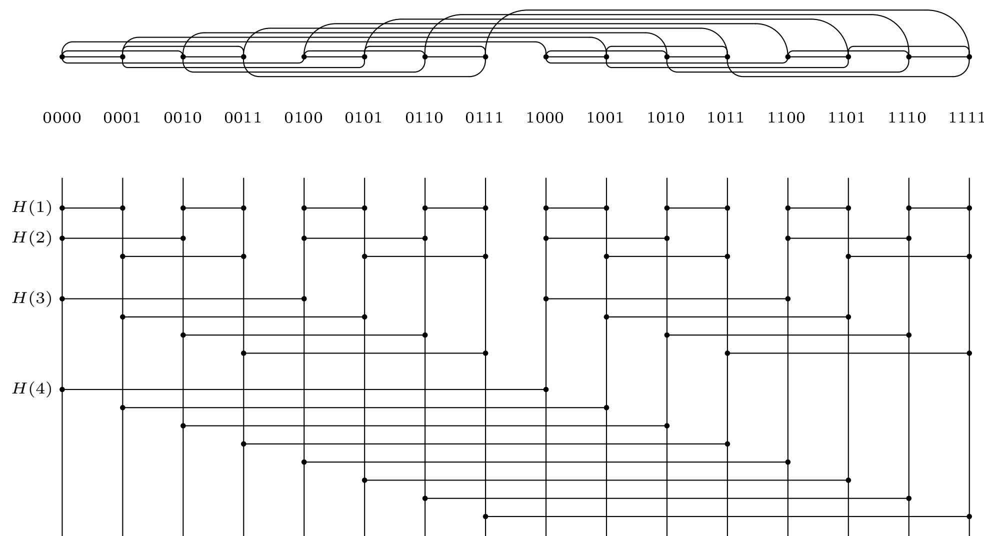
Nas figuras abaixo, estão ilustrados o $H(2)$ e o $H(3)$ sob a visualização linear:



Sob essa visualização, a figura de $H(d)$ é composta pelas figuras de dois $H(d-1)$, uma à esquerda e outra à direita. Em seguida, as arestas entre os vértices desses dois sub-hipercubos são ligações entre o i-ésimo vértice do $H(d-1)$ da esquerda e o i-ésimo vértice do $H(d-1)$ da direita, onde $0 \leq i < 2^{d-1}$. Isso pode ser melhor observado nos

diagramas que estão abaixo das figuras dos hipercubos, onde estão indicadas as ligações que são acrescentadas quando se deseja obter um hipercubo de dimensão maior.

Portanto, conforme a figura anterior, a partir de dois $H(3)$ pode-se chegar à representação linear de $H(4)$:



A numeração dos vértices é dada pelas suas posições na seqüência, começando com 0 e terminando em $2^d - 1$.

Daremos agora alguns exemplos de algoritmos para o $H(d)$ que se tornam mais simples de serem compreendidos a partir da visualização linear.

Inversão: Pode-se observar que na seqüência ordenada das representações binárias dos valores de 0 a $2^d - 1$, as posições simétricas (isto é, a primeira e a última, a segunda e a penúltima, etc.) têm exatamente os d *bits* trocados. Com essa observação, é fácil a elaboração de um algoritmo para inversão de uma seqüência de $2^d = p$ elementos presentes nos registradores R dos vértices do hipercubo, que coloca no primeiro vértice a informação que estava no último, no segundo a informação que estava no penúltimo, e assim por diante. Tal algoritmo gasta tempo $O(d) = O(\log p)$:

for $b := 0$ to $d - 1$ do
    $R(i^{(b)}) \leftrightarrow R(i)$

Nesta notação, conforme já foi comentado, i corresponde ao endereço de cada vértice do hipercubo. Todos os vértices do hipercubo ($0 \leq i < p$) executam esse mesmo algoritmo.

É interessante notar que não é necessário seguir a ordem crescente dos *bits* dentro do "for".

Intercalação (ou merge) bitônica: Esse algoritmo [Bat68] supõe que as duas seqüências a serem intercaladas estejam uma em cada $H(d-1)$, um valor por processador, armazenado no registrador R. A primeira está em ordem crescente e a segunda em ordem decrescente, com relação à ordem dos processadores. É utilizado também um registrador auxiliar S.

for $b := d-1$ downto 0 do
  $S(i) \leftarrow R(i^{(b)})$
  $R(i) := \min(S(i),\ R(i)),\ (i_b = 0)$
  $R(i) := \max(S(i),\ R(i)),\ (i_b = 1)$

Abaixo temos um exemplo para o $H(3)$, onde estão indicados os valores do registrador R:

Vértices	000	001	010	011	100	101	110	111
No início	2	3	3	9	6	4	2	1
$b = 2$	2	3	2	1	6	4	3	9
$b = 1$	2	1	2	3	3	4	6	9
$b = 0$	1	2	2	3	3	4	6	9

A partir dessa intercalação, fica muito simples a elaboração de um algoritmo de ordenação tipo "divisão-e-conquista" composto de três passos:

- ordenações recursivas em paralelo em cada $H(d-1)$;
- inversão dos valores no $H(d-1)$ da direita;
- intercalação bitônica.

Como a complexidade da intercalação e da inversão é $O(d) = O(\log p)$, esta ordenação custa $O(\log^2 p)$.

Rank: Seja $Rank(i)$ o número de vértices precedentes a i que tenham informação no registrador S, isto é, $Rank(i) = |\ \{j \mid j < i \text{ e } S(j) \neq \text{null}\}\ |,\ 0 \leq i < p$.

Exemplo de como deve ser calculado o *Rank* para $H(3)$:

Vértices	000	001	010	011	100	101	110	111
S	–	–	–	a	–	b	c	d
Rank	–	–	–	0	–	1	2	3

A partir da visualização linear de $H(d)$, a idéia é a seguinte:

- O algoritmo tem d passos, cada passo aglutinando dois hipercubos menores em um outro maior;
- Em cada passo, se um determinado vértice i estiver no sub-hipercubo da esquerda, ele não terá seu *Rank* modificado. Caso contrário, isto é, se estiver na direita, soma-se ao seu *Rank* o total de vértices com informação no registrador S que estão no sub-hipercubo da esquerda.

Nesse algoritmo [NaS81, NaS82], serão utilizados também os registradores R, V e A, supostos disponíveis em cada vértice.

Para $0 \leq k < d$, $H(k)$ é um sub-hipercubo de $H(d)$, e R^k e V^k são os valores dos registradores R e V no passo k. Seus significados são os seguintes:

- $R^k(i)$: $Rank(i)$ em $H(k)$;
- $V^k(i)$: número total de vértices com informação no $H(k)$ onde está o vértice i.

Fórmulas para o cálculo:

$$\begin{array}{rcl} R^0(i) & = & 0,\ 0 \leq i < p. \\ V^0(i) & = & 0,\ \text{se } S(i) = \text{null},\ 0 \leq i < p \\ & & 1,\ \text{se } S(i) \neq \text{null},\ 0 \leq i < p. \\ R^k(i) & = & R^{k-1}(i),\ \text{se } i_{k-1} = 0 \\ & & R^{k-1}(i) + V^{k-1}(i^{(k-1)}),\ \text{se } i_{k-1} = 1. \\ V^k(i) & = & V^{k-1}(i) + V^{k-1}(i^{(k-1)}). \end{array}$$

Algoritmo:

$R(i) := 0$
$V(i) := 1,\ (S(i) \neq \text{null})$
$V(i) := 0,\ (S(i) = \text{null})$
for $b := 0$ to $d-1$ do
    $A(i) \leftarrow V(i^{(b)})$
    $R(i) := R(i) + A(i),\ (i_b = 1)$
    $V(i) := V(i) + A(i)$

Em cada passo do "for", são calculados $R^{b+1}(i)$ e $V^{b+1}(i)$.

Concentração: Uma vez calculado $Rank(i)$ e armazenado em $R(i)$, pode-se concentrar as informações presentes no hipercubo, mantendo-se a ordem inicial. Em outras palavras, pode-se colocar a informação de $S(i)$ no vértice de endereço $R(i)$, deixando-se os demais vazios.

Para o mesmo exemplo anterior, indicamos os valores iniciais e finais dos registradores S:

Vértices	000	001	010	011	100	101	110	111
S	–	–	–	a	–	b	c	d
R	–	–	–	0	–	1	2	3
S	a	b	c	d	–	–	–	–

Algoritmo:

for $b := 0$ to $d-1$ do
    $(S(i^{(b)}), R(i^{(b)})) \leftarrow (S(i), R(i)),\quad (S(i) \neq \text{null and } R(i)_b \neq i_b)$

Em cada passo, as informações são enviadas a vértices cujos endereços possuem um *bit* a mais igual ao endereço desejado. A demonstração de que não há colisões nos passos intermediários é simples, e está em [NaS82].

Distribuição: Esse algoritmo é o contrário da concentração, isto é, espalha as informações segundo os valores que estão armazenados em R. Supõe que as informações estejam nos primeiros processadores no registrador S, além de que R possua valores apropriados.

Exemplo para o $H(3)$:

Vértices	000	001	010	011	100	101	110	111
S	a	b	c	–	–	–	–	–
R	1	5	6	–	–	–	–	–
S	–	a	–	–	–	b	c	–

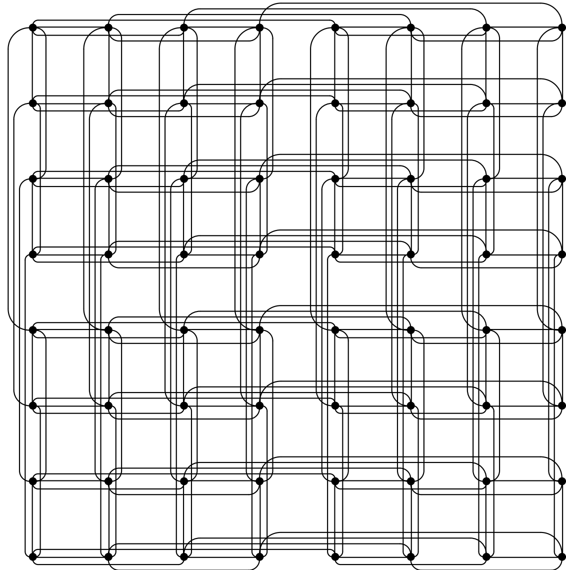
O algoritmo é o mesmo da concentração, invertendo-se os passos:

for $b := d - 1$ downto 0
$(S(i^{(b)}), R(i^{(b)})) \leftarrow (S(i), R(i)), \quad (S(i) \neq \text{null and } R(i)_b \neq i_b)$

2.5.3 Visualização matricial

Esta visualização, como havíamos dito, é uma generalização multi-dimensional da visualização linear. Na visualização matricial de duas dimensões, os vértices de $H(d)$ são considerados na forma de uma matriz com $L = 2^m$ linhas e $C = 2^n$ colunas. Portanto, $p = 2^d = 2^{m+n} = LC$.

Como exemplo, $H(6)$ é ilustrado abaixo, onde $m = n = 3$.



Os endereços dos vértices para esse exemplo estão no quadro abaixo:

000 000	000 001	000 010	000 011	000 100	000 101	000 110	000 111
001 000	001 001	001 010	001 011	001 100	001 101	001 110	001 111
010 000	010 001	010 010	010 011	010 100	010 101	010 110	010 111
011 000	011 001	011 010	011 011	011 100	011 101	011 110	011 111
100 000	100 001	100 010	100 011	100 100	100 101	100 110	100 111
101 000	101 001	101 010	101 011	101 100	101 101	101 110	101 111
110 000	110 001	110 010	110 011	110 100	110 101	110 110	110 111
111 000	111 001	111 010	111 011	111 100	111 101	111 110	111 111

Esse endereçamento é feito dentro de cada linha da esquerda para a direita, começando-se pela linha superior e terminando-se na inferior. Dessa forma, considerando o hipercubo como uma matriz $P = (p_{i,j})$, $0 \leq i < L$ e $0 \leq j < C$, o vértice correspondente à posição $p_{0,0}$ da matriz tem endereço 0, $p_{0,C-1}$ é o $C-1$, $p_{1,0}$ é o C, e assim por diante. O endereço de cada vértice tem uma representação binária com $d = n + m$ *bits*.

Com esta disposição, é fácil observar que dentro de cada linha os m *bits* mais significativos permanecem constantes, enquanto que os n menos significativos crescem de 0 a $C - 1$. Pode-se então considerar cada linha como um $H(n)$ disposto linearmente.

Por outro lado, o mesmo ocorre em cada coluna, onde os n *bits* menos significativos permanecem constantes e os m mais significativos crescem de 0 a $L - 1$. Portanto, cada coluna também pode ser considerada como um $H(m)$ disposto linearmente.

Além disso, quando se visualiza um $H(d)$ na forma matricial com $L = 2^m$ linhas e $C = 2^n$ colunas, pode-se considerar que a matriz P é formada por sub-matrizes $P_{i,j}$ de ordem $2^{m-m'} \times 2^{n-n'}$, onde $0 \leq m' \leq m$ e $0 \leq n' \leq n$, para $0 \leq i < 2^{m'}$ e $0 \leq j < 2^{n'}$:

$P_{0,0}$ … $P_{0,2^{n'}-1}$

⋮ ⋮

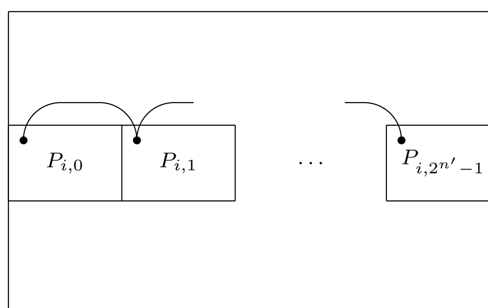
$P_{2^{m'}-1,0}$ … $P_{2^{m'}-1,2^{n'}-1}$

Para enunciarmos outras propriedades, será necessário estabelecer uma nomenclatura. Dadas k matrizes $M_0, M_1, \ldots, M_{k-1}$ de mesma ordem, chamaremos de **elementos correspondentes** cada grupo de k elementos, um de cada matriz, que possuam os mesmos

índices. Também chamaremos de sub-matrizes de P de uma mesma linha as sub-matrizes $P_{i,0}, P_{i,1}, \ldots, P_{i,2^{n'}-1}$, e de maneira análoga, $P_{0,j}, P_{1,j}, \ldots, P_{2^{m'}-1,j}$ serão chamadas sub-matrizes de P de uma mesma coluna.

Dessa forma, podemos descrever mais algumas propriedades importantes da visualização matricial:

- Cada uma das sub-matrizes $P_{i,j}$ corresponde a um $H(n + m - n' - m')$ também visualizado matricialmente.
- Cada grupo de elementos correspondentes das sub-matrizes de P de uma mesma linha forma um $H(n')$. A figura abaixo procura ilustrar esta propriedade, onde os arcos ligam vértices de um desses grupos:



- De modo análogo à propriedade anterior, cada grupo de elementos correspondentes das sub-matrizes de P de uma mesma coluna forma um $H(m')$.

Essa visualização matricial do hipercubo pode ser estendida a matrizes com dimensões maiores. Por exemplo, podemos considerar um arranjo cúbico com 2^m elementos na primeira dimensão, 2^n na segunda e 2^r na terceira. Com isso, $p = 2^m 2^n 2^r$, e $p_{i,j,k}$ é o processador com endereço $i + j2^m + k2^m 2^n$. Do mesmo modo que na visualização matricial de duas dimensões, cada grupo de processadores num mesmo eixo ou direção forma um sub-hipercubo, cada sub-matriz cúbica corresponde a um sub-hipercubo, e os elementos correspondentes das sub-matrizes também formam sub-hipercubos. Além disso, cada face é uma visualização matricial de duas dimensões.

Em seguida descreveremos alguns dos algoritmos que podem ser melhor compreendidos considerando a visualização matricial de duas dimensões de $H(d)$. Outros algoritmos que aproveitam a visualização matricial de três dimensões estão tratados de um modo especial no capítulo 4.

Permutação: Este algoritmo, publicado em [NaS82], considera o hipercubo sob a visualização $L = 2^m \times C = 2^n$, e permuta os C números de 0 a $C - 1$ (cada um representado por n *bits*), inicialmente armazenados nos vértices correspondentes à linha 0. No final, o número i termina em $p_{0,i}$, $0 \leq i < C$. Supõe-se que $n > m$, sem perda de generalidade.

A idéia baseia-se no *MSD-radix sort* [Knu73]: os números vão sendo agrupados conforme seus dígitos, começando-se pelos mais significativos.

A partir da representação binária de cada número, cada grupo de m *bits*, começando da esquerda para a direita, corresponderá a um dígito. Portanto, se os números possuem n dígitos binários, sua nova representação possuirá $\lceil n/m \rceil$ dígitos, dentro de 2^m dígitos diferentes (exatamente o número de linhas da matriz).

Como no *radix sort*, haverá uma fase para cada dígito dos números, ou seja, $\lceil n/m \rceil$ fases, começando nos dígitos mais significativos. Inicialmente, os C elementos são considerados como pertencentes a um único grupo formado pelas C colunas. Durante o andamento do algoritmo, cada grupo corresponderá aos números que tiveram os mesmos dígitos nas fases anteriores. Dessa forma, no final, cada grupo será unitário, e corresponderá a uma única coluna.

Descrição de cada fase i, $1 \leq i \leq \lceil n/m \rceil$:

- Deslocamento de cada número dentro de sua coluna até atingir a linha correspondente ao seu i-ésimo dígito.

  Utiliza-se um algoritmo de DS em cada coluna, pois elas correspondem a sub-hipercubos, eliminando-se depois o conteúdo dos vértices que não são o desejado.

- Dentro de cada linha de cada grupo (que é também um hipercubo disposto linearmente) é feito um *Rank* e uma concentração, aplicando-se os algoritmos já mostrados.

  Entretanto, como se trata de uma permutação, na fase i todas as linhas de um mesmo grupo terão 2^{n-im} elementos, pois cada uma delas corresponde a um dígito.

  Portanto, antes de se fazer a concentração, é somado um fator ao *Rank*. Esse fator é o número da linha multiplicado por 2^{n-im}. O resultado disso é que cada coluna terá um único número. Aqueles números que foram concentrados, além de estarem na mesma linha (na linha correspondente ao i-ésimo dígito que possuem em comum), ficam em colunas vizinhas, ao mesmo tempo que grupos com números menores ficam em colunas mais à esquerda. Os conjuntos de colunas vizinhas que possuem elementos numa única linha formam os novos grupos para a próxima fase.

Depois da última fase, aplica-se novamente o algoritmo de DS em cada coluna, para se ter os números na linha 0.

As figuras abaixo ilustram um exemplo no $H(5)$ quando $n = 3$ e $m = 2$. Dessa forma, o algoritmo tem $\lceil 3/2 \rceil = 2$ fases. Para facilitar o entendimento, os grupos correntes em cada momento estão separados por colunas duplas.

No início:

5	2	1	4	0	7	6	3

Fase 1:

		1		0			
	2						3
5			4				
					7	6	

1	0						
		2	3				
				5	4		
						7	6

Fase 2:

	0	2			4		6
1			3	5		7	

0		2		4		6	
	1		3		5		7

No final:

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

Será visto no próximo capítulo que existem algoritmos para DS (e portanto para AS também) no $H(d)$ que gastam tempo $O(d)$. Dessa forma, cada fase deste algoritmo gasta tempo $O(\log C + \log L) = O(n+m) = O(n)$. Portanto, para se permutar $C = 2^n$ elementos, a complexidade de tempo é $O(kn) = O(k \log C)$, onde $k = \lceil n/m \rceil$, e o número de vértices utilizados é $p = 2^{n+m} = 2^{n/k} 2^n = C^{1+1/k}$.

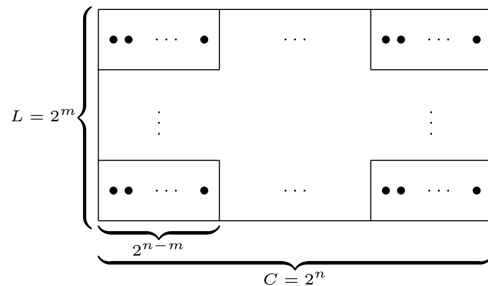
Ordenação: Este algoritmo de ordenação também apresentado em [NaS82] baseia-se no seguinte algoritmo descrito em [Pre78]:

- Dividir uma seqüência de 2^n elementos $a_1, a_2, \ldots, a_{2^n}$ em k sub-seqüências denotadas por $s_1, s_2, \ldots, s_k$.
- Ordenar as k sub-seqüências recursivamente.
- Fazer a intercalação das sub-seqüências ordenadas da seguinte forma: para cada elemento a_i na sub-seqüência s_l, calcular $c_{i,q}$ para todo q, que é o número de elementos na sub-seqüência s_q que:
  - não são maiores do que a_i, se $q < l$;
  - estão à esquerda de a_i, se $q = l$;
  - são menores do que a_i, se $q > l$.

  Em seguida, calcular $R_i = \sum_{q=1}^{k} c_{i,q}$, que é a posição do elemento a_i na ordenação final, e rotear cada elemento a_i segundo R_i.

Em [NaS82] o $H(d)$ é novamente considerado sob a visualização matricial com $L = 2^m$ linhas e $C = 2^n$ colunas, e supõe-se que os elementos estejam nos vértices da primeira linha, um elemento por processador. A seqüência de 2^n elementos é dividida em 2^m sub-seqüências de tamanho 2^{n-m} cada, que são ordenadas recursivamente e em paralelo. Para isto, supõe-se outra vez que $n > m$, sem perda de generalidade. Descreveremos sucintamente como é feita a intercalação das sub-seqüências aproveitando as propriedades da visualização matricial.

$H(d)$ é encarado como uma matriz quadrada de ordem 2^m, onde seus elementos são sub-matrizes de ordem $1 \times 2^{n-m}$. Dessa forma, segundo a descrição anterior das propriedades da visualização matricial, $P = (P_{i,j})$, $0 \leq i,j < 2^m$, onde $n' = m' = m$, conforme suas definições na página 16. Portanto cada sub-matriz $P_{i,j}$ corresponde a um $H(n-m)$. O esquema abaixo mostra a visualização de P:



Segundo as mesmas propriedades já descritas, cada grupo de elementos correspondentes das sub-matrizes $P_{i,0}, P_{i,1}, \ldots, P_{i,2^m-1}$ forma um $H(m)$ que chamaremos de hipercubo horizontal. O mesmo ocorre com cada grupo de elementos correspondentes de $P_{0,j}, P_{1,j}, \ldots, P_{2^m-1,j}$, que forma um $H(m)$ chamado vertical. Há portanto 2^m $H(m)$ horizontais e 2^m $H(m)$ verticais.

Para facilitar a compreensão do algoritmo, a descrição dos passos estará acompanhada de um exemplo para o $H(6)$, com $n = 4$ e $m = 2$. No esquema abaixo, as sub-matrizes correspondem a 4 processadores. A situação inicial é a seguinte:

2 5 8 9	1 3 7 9	0 2 5 6	4 4 5 8

Lembrando que a sub-seqüência j está inicialmente armazenada na sub-matriz $P_{0,j}$, um elemento por processador, os passos da intercalação são os seguintes:

- A partir de cada sub-matriz $P_{0,j}$, são feitas DS através dos $H(m)$ verticais, de modo que cada sub-matriz $P_{i,j}$ contenha os valores de $P_{0,j}$, isto é, a sub-seqüência j. Custo: $O(m)$.

2 5 8 9	1 3 7 9	0 2 5 6	4 4 5 8
2 5 8 9	1 3 7 9	0 2 5 6	4 4 5 8
2 5 8 9	1 3 7 9	0 2 5 6	4 4 5 8
2 5 8 9	1 3 7 9	0 2 5 6	4 4 5 8

- A partir de cada sub-matriz $P_{i,i}$, são feitas DS através dos $H(m)$ horizontais, de modo que cada sub-matriz $P_{i,j}$ contenha também os valores de $P_{i,i}$, isto é, a sub-seqüência i. Custo: $O(m)$.

2 5 8 9 2 5 8 9	1 3 7 9 2 5 8 9	0 2 5 6 2 5 8 9	4 4 5 8 2 5 8 9
2 5 8 9 1 3 7 9	1 3 7 9 1 3 7 9	0 2 5 6 1 3 7 9	4 4 5 8 1 3 7 9
2 5 8 9 0 2 5 6	1 3 7 9 0 2 5 6	0 2 5 6 0 2 5 6	4 4 5 8 0 2 5 6
2 5 8 9 4 4 5 8	1 3 7 9 4 4 5 8	0 2 5 6 4 4 5 8	4 4 5 8 4 4 5 8

- No final dos dois passos anteriores, cada sub-matriz $P_{i,j}$ possui as sub-seqüências i e j, um elemento de cada por processador. Dessa forma, através de uma intercalação bitônica estática, para cada elemento r da sub-seqüência i pode-se calcular o valor de $c_{r,j}$. Essa intercalação aproveita o fato das sub-seqüências estarem em um $H(n-m)$, e é uma variação da descrita na visualização linear. Custo: $O(n-m)$.

- A somatória dos valores de c para cada elemento de cada sub-seqüência pode ser feita com AS nos $H(m)$ verticais, seguida de uma nova DS, para que os valores estejam em todas as linhas. Custo: $O(m)$.

- A partir de cada sub-matriz $P_{i,i}$, são feitas distribuições dos elementos da sub-seqüência i segundo os valores de c nos $H(n)$ que correspondem a cada linha da visualização matricial completa. Cada elemento fica então no processador de sua coluna definitiva. Custo: $O(n)$.

⊔ ⊔ 2 ⊔	⊔ ⊔ ⊔ 5	⊔ ⊔ ⊔ ⊔	8 ⊔ 9 ⊔
⊔ 1 ⊔ ⊔	3 ⊔ ⊔ ⊔	⊔ ⊔ ⊔ 7	⊔ ⊔ ⊔ 9
0 ⊔ ⊔ 2	⊔ ⊔ ⊔ ⊔	5 ⊔ 6 ⊔	⊔ ⊔ ⊔ ⊔
⊔ ⊔ ⊔ ⊔	⊔ 4 4 ⊔	⊔ 5 ⊔ ⊔	⊔ 8 ⊔ ⊔

- Por fim, todos os elementos são espalhados através de DS nas colunas da visualização matricial. Custo: $O(m)$.

0 1 2 2	3 4 4 5	5 5 6 7	8 8 9 9
0 1 2 2	3 4 4 5	5 5 6 7	8 8 9 9
0 1 2 2	3 4 4 5	5 5 6 7	8 8 9 9
0 1 2 2	3 4 4 5	5 5 6 7	8 8 9 9

A complexidade desta intercalação é de $O(m) + O(n-m) + O(n) = O(n)$, uma vez que consideramos $n > m$. Com isso, temos que a ordenação de $C = 2^n$ elementos num $H(n+m)$ gasta tempo $T(2^n) = T(2^{n-m}) + O(n)$, pois as ordenações recursivas são feitas em paralelo. Como $T(1) = O(1)$, temos que $T(2^n) = O(kn) = O(k \log C)$, onde $k = \lceil n/m \rceil$, usando $p = 2^{n+m} = 2^{n/k}2^n = C^{1+1/k}$ processadores.

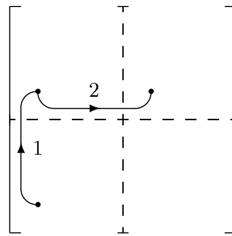
Transposição de matriz: Vejamos mais um exemplo que aproveita algumas propriedades da visualização matricial. Tal algoritmo é sugerido em [McV87] e [BeT89].

Seja M uma matriz quadrada $n \times n$. Suponhamos que cada um de seus elementos esteja armazenado no correspondente vértice da visualização matricial $n \times n$ de $H(d)$. Para que isso ocorra, temos que $n^2 = 2^d$. Utilizando as ligações extras pertencentes ao hipercubo, é possível obter a transposta de M em tempo $O(\log n)$ através de um algoritmo de "divisão-e-conquista":

- Seja
$$M = \begin{bmatrix} M_{0,0} & M_{0,1} \\ M_{1,0} & M_{1,1} \end{bmatrix},$$
onde $M_{i,j}$ são sub-matrizes $\frac{n}{2} \times \frac{n}{2}$.

  As matrizes $M_{i,j}$, consideradas dentro da visualização matricial, correspondem a sub-hipercubos $H(d-2)$.

- Inicialmente, $M_{0,1}$ e $M_{1,0}$ têm os seus elementos correspondentes trocados entre si. Utilizando-se as ligações do hipercubo, como pode ser visto abaixo, gasta-se nesta fase tempo constante.



- Em seguida, o algoritmo é aplicado recursivamente em paralelo em cada uma das sub-matrizes $M_{i,j}$.

O tempo gasto para transpor M é $T(n) = T(n/2) + O(1)$, de onde segue que $T(n) = O(\log n)$.

Capítulo 3

As operações básicas no hipercubo

Neste capítulo trataremos de algoritmos para resolver as operações básicas de comunicação no hipercubo. Como determinados aspectos da computação paralela, tais como tempo e capacidade de comunicação, diferentes tipos de canais, etc., são modelados de diversos modos, conforme comentamos no capítulo inicial, existem também diversos algoritmos, baseando-se cada um em suposições diferentes.

Será necessário estabelecer nossas suposições, e a partir disso definir algoritmos e calcular seus custos, para assim podermos utilizá-los adiante.

Com relação ao tempo de comunicação, o modelo constante faz restrições que julgamos muito fortes, ao passo que o modelo linear calcula o tempo em função dos parâmetros β, τ e L. Como ao longo de todo o trabalho somente será necessária a ordem do tempo gasto em algumas destas operações básicas, consideraremos apenas a dependência linear do tempo de comunicação com relação ao tamanho das mensagens. Pelo mesmo motivo, o fato do canal ser *half-duplex* ou *full-duplex* não nos importará, pois isso corresponde apenas a um fator 2 no tempo. Por outro lado, a opção entre d-port e 1-port é significativa. Para máquinas com arquitetura de hipercubo $H(d)$, isso corresponde a um fator de $d = \log p$ na ordem do tempo de comunicação. Apresentaremos algoritmos ótimos que consideram a hipótese de d-port, e que podem ser simulados em uma máquina 1-port com o fator extra indicado acima. Também descreveremos alguns outros algoritmos SIMD que exigem apenas a hipótese de 1-port. Esses algoritmos serão utilizados no capítulo seguinte.

Além disso, é importante relacionar outras hipóteses que consideramos:

- Utilizamos um modelo síncrono, com a restrição de que as mensagens são entregues em um único ciclo.

- O tempo de comunicação é o mesmo entre qualquer par de processadores interligados.

- A transmissão de mensagens entre os processadores é livre de erros.

- O tempo de "combinação" de mensagens, considerado nas operações AS e AM, não é superior ao tempo de comunicação entre dois processadores, e a mensagem resultante tem o mesmo tamanho das mensagens iniciais.

Nas descrições dos algoritmos, consideramos também que cada mensagem enviada por um processador e destinada a outro tem tamanho unitário.

A partir dessas hipóteses, podemos descrever os algoritmos ótimos apresentados em [BeT89] e [BOS91] para resolver DM, AM, Rc, Dp e TC. Eles evitam quebra ou associação[1] de mensagens (cujo tempo é desprezado nos outros modelos), consideram β e τ como constantes e a mensagem com tamanho unitário. Tais algoritmos foram inicialmente apresentados no primeiro trabalho citado, e são ótimos para o hipercubo quanto ao tempo de comunicação segundo o modelo por eles utilizado. Posteriormente sofreram algumas modificações descritas na segunda referência citada acima, com o objetivo de baixar o número de transmissões.

Saad e Schultz [SaS89] também apresentam algoritmos para algumas dessas operações básicas, considerando o modelo linear. Portanto, neste trabalho o tempo gasto em cada algoritmo está em função de vários parâmetros: β, τ, p e o tamanho da mensagem.

3.1 Alguns algoritmos ótimos

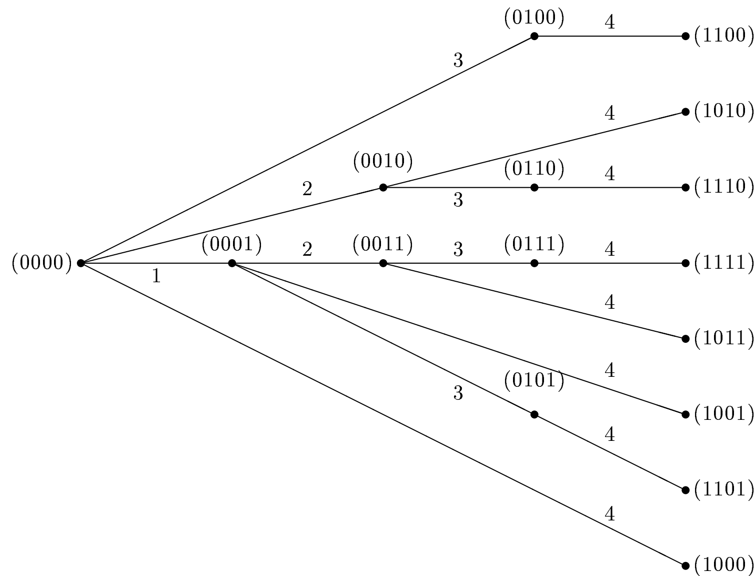
Os algoritmos aqui apresentados estão em [BeT89]. Na verdade, fizemos uma pequena modificação no algoritmo referente a DS e AS, para que tivesse o mesmo custo sob as hipóteses de 1-port e d-port.

3.1.1 Difusão e Acumulação Singular (DS e AS)

O tempo mínimo para se efetuar a DS em qualquer topologia tem como limite inferior o seu diâmetro. No $H(d)$, é possível realizar a DS em d passos, correspondendo a um tempo $O(d)$, o que significa que existe um algoritmo ótimo para esse problema. DS nada mais é do que uma seqüência de transmissões que pode ser representada graficamente como uma árvore de difusão com os vértices do hipercubo, onde as arestas indicam o passo em que a transmissão é efetuada.

Um possível modo de se fazer a DS é aquele em que cada vértice i que já tenha recebido a mensagem, no passo b a envia para o vértice $i^{(b)}$, $0 \leq b < d$. A figura abaixo mostra a árvore de difusão a partir do vértice (0000) no $H(4)$, onde os rótulos das arestas indicam os números dos passos:

[1] Como já foi observado, por associação de mensagens entende-se justaposição, ou seja, obtém-se uma nova mensagem cujo tamanho é a soma dos tamanhos iniciais.



A árvore de difusão a partir do vértice i pode ser obtida de modo análogo, ou através da árvore acima, trocando cada vértice j por $j \oplus i$ (aplicação do operador ou-exclusivo).

A vantagem dessa árvore é que, em cada passo da DS, cada vértice comunica-se com no máximo um outro vértice, o que garante tempo $O(d)$ mesmo sob a hipótese de 1-port.

AS pode ser resolvida de modo análogo, bastando efetuar as transmissões na ordem inversa dos passos, gastando também tempo $O(d) = O(\log p)$.

3.1.2 Difusão e Acumulação Multi-nó (DM e AM)

Na DM, cada vértice deve receber $p - 1$ mensagens. Supondo a hipótese de d-port, o tempo mínimo necessário para o término da DM no hipercubo será então proporcional a $r = \lceil (p-1)/\log p \rceil$, pois cada vértice está ligado a outros $\log p$ vértices. r é portanto o número mínimo de passos necessários para DM e AM.

Um modo de resolver este problema é encontrar uma árvore de difusão para cada vértice. Para isso, é interessante utilizar o mesmo método anterior, ou seja, encontrar uma árvore para o vértice 0 e, a partir dessa árvore, aplicar o operador ou-exclusivo para se obter as árvores correspondentes aos outros vértices.

Entretanto, deve-se tomar um cuidado: Como a DM corresponde a p DS simultâneas, o tempo gasto em cada passo da DM será determinado pela ligação mais sobrecarregada nesse passo, ou seja, a que tiver que transmitir mais mensagens. O tempo será mínimo se em cada passo houver exatamente uma única transmissão em cada ligação. A partir do modo como as árvores de difusão são obtidas (da árvore do vértice 0, através do operador ou-exclusivo), para que cada ligação transmita uma única mensagem em cada passo (ou seja, para que em cada passo pertença a uma única árvore de difusão), é suficiente a seguinte condição: dadas duas ligações (x, y) e (x', y') pertencentes a um mesmo passo da árvore de difusão do vértice 0, o *bit* no qual x e y diferem não é o mesmo daquele no qual x' e y' diferem.

O problema passa a ser encontrar uma árvore de difusão para o vértice 0 que satisfaz a condição acima. A árvore que será descrita a seguir garante que cada ligação transmita uma única mensagem em cada um dos passos da DM, e permite um algoritmo com r passos. Dessa forma, tem-se uma DM ótima, pois o número de passos e o tempo de cada passo são mínimos.

Descrição da árvore de difusão do vértice 0: Seja N_k o conjunto das representações binárias que possuem k *bits* 1 e $d-k$ *bits* 0. As representações binárias correspondem aos nós do $H(d)$.

Sabe-se que:

$$|N_k| = \binom{d}{k}$$

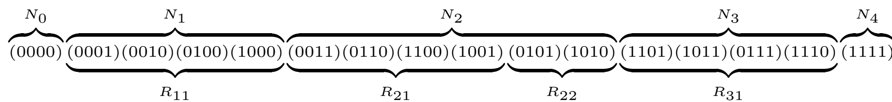
Portanto,

$$\begin{cases} |N_0| = |N_d| = 1, \text{ pois } N_0 = \{(00\dots0)\} \text{ e } N_d = \{(11\dots1)\} \\ |N_1| = |N_{d-1}| = d. \end{cases}$$

Além disso, para $1 < k < d-1$, $|N_k| \geq 2d$ quando $d \geq 5$, pois:

$$\binom{d}{k} \geq \binom{d}{2} = \frac{d(d-1)}{2!} \geq 2d, \text{ quando } d \geq 5.$$

Sejam $R_{k1}, R_{k2}, \dots, R_{kn_k}$ os sub-conjuntos de N_k que formam n_k classes de equivalência sob rotação à esquerda. Para $d = 4$, a figura abaixo mostra quais são os elementos desses sub-conjuntos:



Sabe-se que $n_1 = n_{d-1} = 1$, pois as representações binárias de N_1 e N_{d-1} possuem $d-1$ *bits* iguais.

R_{k1} será a classe de equivalência do número com k *bits* 1 mais à direita:

$$(00\dots0\underbrace{11\dots1}_{k})$$

Com isso, $|R_{k1}| = d$. Além disso, é fácil ver que $|R_{ki}| \leq d$.

Será associado um número $n(t) \in \{0, 1, \dots, p-1\}$ a cada representação binária t de acordo com ordem

$$\overbrace{(00\dots0)}^{N_0}\overbrace{R_{11}}^{N_1}\overbrace{R_{21}\dots R_{2n_2}}^{N_2}\overbrace{R_{31}\dots R_{3n_3}}^{N_3}\dots R_{(d-2)n_{d-2}}\overbrace{R_{(d-1)1}}^{N_{d-1}}\overbrace{(11\dots1)}^{N_d},$$

de forma que $n(00\dots0) = 0$ e $n(11\dots1) = p-1$.

Definindo $m(t) = 1 + [(n(t)-1) \bmod d]$, temos que a seqüência de $m(t)$ associada às representações binárias que estão entre $(00\dots0)$ e $(11\dots1)$ na ordem acima é $1,2,\dots,d,1,2,\dots,d,1,2,\dots$

$m(t)$ servirá para determinar a ordem das representações binárias dentro de cada R_{ki}: o primeiro elemento t (elemento mais à esquerda da seqüência) de R_{ki} é escolhido de tal forma que o *bit* na posição $m(t)$ a partir da direita seja 1. Além disso, para os elementos de R_{k1}, é necessário que o *bit* na posição $m(t)-1$ (se $m(t) > 1$) ou d (se $m(t) = 1$) a partir da direita seja 0.

No mesmo exemplo anterior, onde $d = 4$, os *bits* que obedecem essas regras estão indicados com um ponto:

	N_0	N_1				N_2						N_3				N_4
	(0000)	$(\dot{0}00\dot{1})$	$(00\dot{1}\dot{0})$	$(0\dot{1}\dot{0}0)$	$(\dot{1}\dot{0}00)$	$(\dot{0}01\dot{1})$	$(01\dot{1}\dot{0})$	$(1\dot{1}\dot{0}0)$	$(\dot{1}\dot{0}01)$	$(010\dot{1})$	$(10\dot{1}0)$	$(1\dot{1}\dot{0}1)$	$(\dot{1}\dot{0}11)$	$(\dot{0}11\dot{1})$	$(11\dot{1}\dot{0})$	(1111)
m:		1	2	3	4	1	2	3	4	1	2	3	4	1	2	3
n:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		R_{11}				R_{21}				R_{22}		R_{31}				

Uma vez dispostas as representações de forma a obedecer as restrições descritas, cada grupo de d representações consecutivas, a partir de $(00\dots0)$, pertencerá a um mesmo conjunto:

$$
\begin{aligned}
E_0 &= \{(00\dots0)\};\\
E_i &= \{t \mid (i-1)d < n(t) \le id\},\ i = 1,2,\dots,r-1;\\
E_r &= \{t \mid (r-1)d < n(t) \le p-1\}.
\end{aligned}
$$

Pode-se notar que $|E_0| = 1$, $|E_i| = d$, para $1 \le i < r$, e $|E_r| \le d$. Mais uma vez utilizaremos como exemplo o caso em que $d = 4$:

E_0	E_1				E_2				E_3				E_4		
(0000)	(0001)	(0010)	(0100)	(1000)	(0011)	(0110)	(1100)	(1001)	(0101)	(1010)	(1101)	(1011)	(0111)	(1110)	(1111)

Esses conjuntos definem quais vértices receberão a mensagem em cada passo: os vértices cujas representações pertençam ao conjunto E_i receberão a mensagem no passo i, $i = 1,2,\dots,r$. Observando-se o exemplo acima, pode-se notar que esses conjuntos correspondem aos grupos de vértices que receberam valores $1,2,\dots,d$ de m.

Cada $t \in E_i$ terá seu correspondente vértice na árvore de difusão conectado com o vértice cuja representação binária é a mesma, invertendo-se apenas o *bit* $m(t)$, que é 1 por construção.

Dessa forma, será válida a propriedade: qualquer $t \in E_i$ será conectado com um elemento de $\bigcup_{k=0}^{i-1} E_k$, ou seja, um vértice que já tenha recebido a mensagem.

Agora, a propriedade pode ser provada. Com certeza, ao se inverter o *bit* $m(t)$ do vértice $t \in E_i$, obtemos uma nova representação binária t' tal que $n(t') < n(t)$, pois o *bit* $m(t)$ era 1. Logo, $t' \notin E_k$, $k > i$. Basta então provar que $t' \notin E_i$. Isso pode ser feito, com exceção de um caso descrito abaixo, provando-se que $n(t) - n(t') \ge d$, uma vez que $|E_i| = d$.

Possíveis casos:

- $t = (11\ldots1)$: Sabe-se que $2^d - 1$ não é divisível por d [BOS91]. Com isso, $m(t) < d$. t será conectado com t', que possui somente um *bit* 0 na posição $m(t)$. Logo, $t' \in R_{(d-1)1}$, e a posição do *bit* 0 por construção é $m(t) = m(t') - 1$ (portanto, $m(t') = m(t) + 1$). Como $t = (11\ldots1) \in E_r$, e é o último elemento, ele possui o maior valor de m em E_r, ou seja, $t' \notin E_r$.

- $t \in R_{kn}$, $n > 1$: Todos os elementos de R_{k1} estão entre t' e t, e como $|R_{k1}| = d$, $n(t) - n(t') \geq d$.

- $t \in R_{k1}$: Segundo a construção descrita, $t' \in R_{(k-1)1}$, e todos os elementos de $R_{(k-1)2}, \ldots, R_{(k-1)n_{k-1}}$ estão entre t' e t, num total de $|N_{k-1}| - d$ elementos. Para $d \geq 5$ e $1 < k-1 < d-1$ (ou seja, $2 < k < d$), $|N_{k-1}| \geq 2d$, e portanto $n(t)-n(t') \geq d$. Os casos $d = 3$ e $d = 4$ são tratados individualmente[2], e os casos $k = 1$ e $k = 2$ não criam dificuldades, pois $R_{11} = E_1$ e $R_{21} = E_2$.

Portanto, cada vértice $t \in E_i$ recebe a mensagem no passo i de um outro vértice que já a possui. Além disso, quaisquer ligações pertencentes ao mesmo passo correspondem a inversões de *bits* em posições diferentes, garantindo assim no máximo uma transmissão por ligação em cada passo da DM. Como a AM utiliza as mesmas árvores, ambas as operações básicas são resolvidas em tempo $O((p-1)/\log p)$ no $H(d)$.

No caso de 1-port, o tempo é $O(p-1)$.

3.1.3 Dispersão e Recolhimento (Dp e Rc)

Na Dp, $p-1$ mensagens devem ser enviadas pelo vértice em questão. Supondo a hipótese de d-port, o tempo mínimo necessário é proporcional a $\lceil (p-1)/\log p \rceil$.

Dp pode ser resolvida pelo algoritmo de AM, no qual somente um vértice envia mensagens. O algoritmo mostrado na seção anterior gasta tempo $O((p-1)/\log p)$, e portanto é ótimo para a Dp.

O mesmo raciocínio pode ser aplicado com relação ao Rc considerando a DM, o que significa que esse algoritmo também é ótimo para esta operação.

3.1.4 Troca Completa (TC)

Se decompusermos o $H(d)$ em dois $H(d-1)$, haverá $p/2$ ligações entre eles. Tais ligações têm a propriedade de serem uma bijeção entre os vértices dos dois $H(d-1)$ (vide definição recursiva do hipercubo já apresentada). Dado um vértice em um desses $H(d-1)$, o outro vértice ligado a ele pela bijeção será chamado sua duplicata.

Na TC, cada $H(d-1)$ deve enviar $(p/2)^2$ mensagens ao outro. Supondo a hipótese de d-port, o tempo mínimo necessário para esta operação é proporcional a $(p/2)^2/(p/2) = p/2 = 2^{d-1}$.

Um algoritmo de "divisão-e-conquista" com duas fases distintas tem complexidade dessa mesma ordem:

[2] As árvores de difusão para o nó $(00\ldots0)$ desses dois casos que satisfazem as restrições exigidas estão em [BeT89] e [BOS91].

- Na primeira fase, ocorre recursivamente e em paralelo a TC em cada um dos $H(d-1)$, e ao mesmo tempo cada vértice envia $p/2 = 2^{d-1}$ mensagens para sua duplicata no outro $H(d-1)$.
- Na segunda fase, cada um dos $H(d-1)$, com as mensagens recebidas do outro, faz em paralelo uma nova TC.

Sendo $T(d)$ o tempo gasto no $H(d)$ para se efetuar a TC, pode-se provar por indução que $T(d) = O(2^d - 1)$:

- Sabe-se que $T(1) = O(1)$: tempo de uma única transmissão.

  Portanto, vale para $d = 1$: $2^d - 1 = 2^1 - 1 = 1$.
- Supondo válida a hipótese de indução para hipercubos com dimensão menor do que d, o tempo máximo da primeira fase do algoritmo é $\max\{2^{d-1}-1,\ 2^{d-1}-1,\ 2^{d-1}\} = 2^{d-1}$, enquanto que o tempo da segunda fase é $T(d-1)$. Logo, $T(d) = T(d-1) + O(2^{d-1})$.

  Aplicando a hipótese de indução, $T(d) = O(2^{d-1} - 1) + O(2^{d-1}) = O(2^d - 1)$.

Logo, TC tem complexidade de tempo $O(2^d - 1) = O(p-1)$ no hipercubo.

3.2 Resultados e Comparações

A tabela abaixo permite uma comparação entre algoritmos ótimos para as operações básicas de comunicação em algumas topologias conhecidas, conforme os algoritmos apresentados em [BeT89]. Supõe-se a hipótese de que cada vértice possa enviar/receber mensagens simultaneamente ao longo de todas as suas ligações, e que todas as redes tenham p vértices.

TOPOLOGIAS	ANEL	ÁRVORE	GRADE	HIPERCUBO
DS e AS	$\Theta(p)$	$\Theta(\log p)$	$\Theta(p^{1/d})$	$\Theta(\log p)$
Dp e Rc	$\Theta(p)$	$\Theta(p)$	$\Theta(p)$	$\Theta(p/\log p)$
DM e AM	$\Theta(p)$	$\Theta(p)$	$\Theta(p)$	$\Theta(p/\log p)$
TC	$\Theta(p^2)$	$\Theta(p^2)$	$\Theta(p^{\frac{d+1}{d}})$	$\Theta(p)$

3.3 Alguns algoritmos SIMD

Esses algoritmos que agora apresentaremos têm como principal objetivo conseguir uma formalização simples de algumas operações básicas, para assim podermos descrever de um modo claro os algoritmos do próximo capítulo. A principal vantagem deles é exigir apenas a hipótese de 1-port, e, nestas condições, têm complexidade de tempo ótima.

Quando descrevemos o hipercubo, pôde ser observado que cada vértice do $H(d)$ é representado por um endereço de d *bits*, isto é, um *bit* em cada dimensão do hipercubo. Uma conseqüência importante deste modo de endereçar os vértices é que se fixarmos k destes d *bits*, os 2^{d-k} endereços possíveis corresponderão a um $H(d-k)$.

Iremos aproveitar esse fato na formalização das operações básicas, passando como parâmetro qual sub-conjunto de *bits* não está fixo, e desse modo indicando em quais sub-hipercubos as correspondentes operações estão sendo efetuadas. Nas descrições abaixo, esse sub-conjunto será chamado de B.

Na tabela abaixo, para o caso do $H(3)$, cujos vértices têm endereços com representação binária $e_2e_1e_0$, estão alguns exemplos de quais sub-hipercubos ficam definidos para determinados valores de B. Os *bits* marcados com um ponto são aqueles que estão fixos em cada caso.

B	Vértices dos sub-hipercubos	Casos
{0,1,2}	000 001 010 011 100 101 110 111	
{0,1}	$\dot{0}00$ $\dot{0}01$ $\dot{0}10$ $\dot{0}11$	$e_2 = 0$
	$\dot{1}00$ $\dot{1}01$ $\dot{1}10$ $\dot{1}11$	$e_2 = 1$
{0,2}	$0\dot{0}0$ $0\dot{0}1$ $1\dot{0}0$ $1\dot{0}1$	$e_1 = 0$
	$0\dot{1}0$ $0\dot{1}1$ $1\dot{1}0$ $1\dot{1}1$	$e_1 = 1$
{1}	$\dot{0}0\dot{0}$ $\dot{0}1\dot{0}$	$e_2e_0 = 00$
	$\dot{0}0\dot{1}$ $\dot{0}1\dot{1}$	$e_2e_0 = 01$
	$\dot{1}0\dot{0}$ $\dot{1}1\dot{0}$	$e_2e_0 = 10$
	$\dot{1}0\dot{1}$ $\dot{1}1\dot{1}$	$e_2e_0 = 11$

3.3.1 Difusão Singular

A árvore de difusão apresentada no algoritmo ótimo para DS permite-nos obter um algoritmo SIMD. Neste algoritmo descrito abaixo, supõe-se que exista um único vértice i em cada sub-hipercubo determinado por B tal que $R(i) \neq$ null, contendo a mensagem a ser difundida. No final, essa informação estará em todos os registradores R. O parâmetro R é passado por referência.

DS(B, R)

Para cada $b \in B$ faça

$$R(i^{(b)}) \leftarrow R(i), \ (R(i) \neq \text{null})$$

A varredura dos $|B|$ elementos pode ser feita em qualquer ordem. Considerando os elementos de B armazenados num vetor de tamanho $|B|$, a complexidade desse algoritmo é $O(|B|)$.

3.3.2 Acumulação Singular

A operação AS considera uma "combinação" de mensagens que deve ser feita em seus estágios intermediários. Essa "combinação" deve gastar um tempo não superior ao da comunicação entre dois vértices, e a mensagem resultante deve possuir também tamanho unitário.

Nos algoritmos descritos no próximo capítulo, a única operação de "combinação" necessária será a soma entre números armazenados em um determinado registrador presente em cada vértice. Conforme suposto na seção 1.3, o resultado dessa soma deve ser um número capaz de ser armazenado num registrador. Como a adição é comutativa, pode-se fazer algo "mais forte" do que a AS: todos os vértices receberão o resultado final, ao invés de um único (isso na verdade corresponde a uma AS seguida de uma DS).

No algoritmo abaixo, supõe-se que as informações estejam no registrador R, e que o resultado final (no caso, a somatória de todos os valores de R no hipercubo) fique no registrador S, que é passado por referência. É utilizado também um registrador auxiliar S'.

SOMA(B, R, S)

$S(i) := 0$
$S'(i) := R(i)$
Para cada $b \in B$ faça
$S'(i) \leftarrow S'(i^{(b)})$
$S(i) := S(i) + S'(i)$
$S'(i) := S(i)$

Da mesma forma que na DS, não é necessário obedecer qualquer ordem na varredura dos *bits*, e supondo as mesmas condições, a complexidade também é $O(|B|)$.

3.3.3 Difusão Multi-nó

Nesta operação cada vértice recebe várias informações (uma de cada outro) e envia uma única. Cada vértice armazenará as informações recebidas em um vetor, e isso será feito de tal forma que a posição da informação nesse vetor corresponda ao endereço do vértice que a enviou dentro do hipercubo considerado.

Esse vetor presente em cada vértice terá $2^{|B|}$ posições, e armazenará inclusive a própria mensagem que deve ser enviada aos outros.

Na descrição abaixo, supõe-se que as informações a serem enviadas estejam no registrador R de cada vértice, que F seja o vetor de $2^{|B|}$ posições, e que AUX seja um outro vetor auxiliar do mesmo tamanho de F. Além disso, F está vazio no início, e & é um operador de concatenação. Dessa forma, $F := F\&X$ corresponde a colocar X no final do vetor F. Esse vetor é passado por referência.

DM(B, R, F)

$F(i) := F(i)\&R(i)$
Para cada $b \in B$ em ordem crescente
$AUX(i) \leftarrow F(i^{(b)})$
$F(i) := F(i)\&AUX(i),\ (i_b = 0)$
$F(i) := AUX(i)\&F(i),\ (i_b = 1)$

Considerando B como um vetor onde os elementos estejam já em ordem crescente, o custo é $O(1) + O(2) + O(4) + \ldots + O(2^{|B|-1}) = O(2^{|B|})$.

Esse algoritmo baseia-se na visualização linear do hipercubo. O exemplo abaixo para o $H(3)$ (onde $B = \{0, 1, 2\}$) mostra as informações em F em cada passo, e ajudará a entender o que ocorre:

Vértices	000	001	010	011	100	101	110	111
No início	a	b	c	d	e	f	g	h
$b = 0$	ab	ab	cd	cd	ef	ef	gh	gh
$b = 1$	abcd	abcd	abcd	abcd	efgh	efgh	efgh	efgh
$b = 2$	abcdefgh	abcdefgh	abcdefgh	abcdefgh	abcdefgh	abcdefgh	abcdefgh	abcdefgh

3.3.4 Complexidades

A tabela abaixo mostra as complexidades dos algoritmos SIMD apresentados para o $H(d)$, sem necessitar da hipótese de que cada vértice possa enviar/receber mensagens simultaneamente ao longo de todas as suas ligações. Esses são os algoritmos que utilizaremos adiante.

ALGORITMO	*COMPLEXIDADE*
DS(B,R)	$O(\log p)$
SOMA(B,R,S)	$O(\log p)$
DM(B,R,F)	$O(p)$

Capítulo 4

Multiplicação de matrizes

A partir do que já foi exposto, veremos neste capítulo um exemplo de como as diferentes visualizações matriciais realmente auxiliam na concepção de algoritmos para o hipercubo. Procuraremos explorar ao máximo as próprias idéias de seus autores com relação a um problema específico: a multiplicação de matrizes.

Nosso objetivo é descrever de uma forma sistemática os algoritmos para multiplicação de matrizes que são possíveis de se conceber utilizando diversas visualizações matriciais. Uma vez feito isso, estabelecemos um critério de comparação entre suas complexidades, através do qual chegamos aos melhores algoritmos. O resultado final é muito interessante: em determinados casos, alguns dos novos algoritmos são melhores do que aqueles apresentados em [DNS81].

Entretanto, a quantidade de algoritmos distintos é muito grande, devido aos seguintes fatores:

- há diversas visualizações matriciais do hipercubo que permitem elaboração de algoritmos;
- pode-se supor diferentes modos de armazenamento inicial dos elementos das matrizes;
- a maioria dos algoritmos baseia-se na multiplicação de sub-matrizes. Para isso, são utilizados outros algoritmos que resolvem o mesmo problema, possibilitando assim diversas combinações.

Para facilitar essa descrição, agruparemos os algoritmos que se originam a partir de uma mesma visualização. Cada um desses grupos será chamado de classe, e receberá um nome que permite associá-la com a respectiva visualização, e também com o número de processadores utilizados. Além disso, nem todos os algoritmos serão descritos formalmente, pois as diferenças entre alguns são muito pequenas e não ocasionam mudança na complexidade.

A seção 4.5 mostra uma visão global e facilita o entendimento dessa divisão. Procura mostrar inclusive como surge a possibilidade de elaboração de algoritmos originais.

Além de alguns novos algoritmos, as formalizações apresentadas também são todas originais. Apesar de utilizarmos a mesmo notação (como já foi comentado no capítulo 2), até os algoritmos conhecidos são reescritos com o objetivo de ganhar clareza e concisão. A utilização das operações básicas, que não está presente em [DNS81] e [NaS82],

permite a descrição de algoritmos de um modo mais simples sem afetar a complexidade de tempo. Além disso, as formalizações receberão como parâmetro um conjunto de *bits* (representado pelo vetor *BITS*) que determina os sub-hipercubos considerados em cada caso (conforme seção 3.3). Isso facilitará a utilização desses algoritmos como sub-rotinas. Para não "carregar" muito a notação, será omitido o índice do vértice para os parâmetros e variáveis.

No cálculo das complexidades de cada algoritmo, caso seja utilizada uma operação básica, se esta gasta tempo $O(x)$ com mensagens de tamanho unitário, gastará tempo $O(tx)$ com mensagens de tamanho t.

Por fim, os índices de cada processador p sempre se referem à visualização que está sendo considerada no momento. Assim, por exemplo, p_i se refere à visualização linear, e $p_{i,j}$ à visualização matricial bi-dimensional.

4.1 O problema e sua complexidade seqüencial

Dadas duas matrizes $\mathcal{A}$ e $\mathcal{B}$ quadradas de ordem n, onde $\mathcal{A}= (a_{i,j})$ e $\mathcal{B}= (b_{i,j})$, $0 \leq i, j < n$, convenientemente dispostas em um $H(d)$ com $p = 2^d$ processadores, deseja-se calcular a matriz produto $\mathcal{C} = \mathcal{AB}$, onde $\mathcal{C}= (c_{i,j})$, tal que $c_{i,j} = \sum_{k=0}^{n-1} a_{i,k}b_{k,j}$. A disposição final da matriz $\mathcal{C}$ no $H(d)$ será análoga às disposições iniciais de $\mathcal{A}$ e $\mathcal{B}$. Supõe-se que cada elemento das matrizes tenha tamanho unitário. Portanto, a transmissão de um elemento de $\mathcal{A}$, $\mathcal{B}$ ou $\mathcal{C}$ gasta $O(1)$ e a transmissão de t elementos gasta $O(t)$.

Existem vários algoritmos seqüenciais para multiplicação de matrizes quadradas de ordem n que gastam tempo $o(n^3)$. O primeiro e mais conhecido é o de Strassen [Str69], de tempo $O(n^{\log_2 7})$. Em seguida, surgiram novos algoritmos, como [Pan80] e [Sch81], e atualmente o melhor gasta tempo $O(n^{2,376})$ [CoW87]. Por outro lado, o limite inferior é ainda um problema em aberto, e o melhor resultado conhecido é $2n^2 + \frac{3}{46}n - 2$ [JaT85]. De qualquer forma, ao longo deste texto, será suficiente considerar que o melhor algoritmo seqüencial conhecido para multiplicação de matrizes quadradas $n \times n$ gasta tempo $O(n^\lambda)$, onde $2 \leq \lambda < 3$.

4.2 Algoritmos paralelos básicos no hipercubo

Existem na literatura alguns algoritmos paralelos para multiplicação de matrizes $n \times n$ no hipercubo nos casos em que o número de processadores p é n, n^2 ou n^3. Esses três tipos de algoritmos têm complexidade $O(n^2)$, $O(n)$ e $O(\log n)$, respectivamente, e serão denominados básicos. Eles pertencerão às classes denominadas N, NN e NNN. Índices serão usados para denotar diferentes algoritmos de uma mesma classe. Por exemplo, NN_1 e NN_2 pertencem à classe NN.

4.2.1 Classe N: com $p = n$ processadores

Nessa primeira classe, o hipercubo é visualizado como uma matriz $1 \times n$, o que equivale à visualização linear. Supõe-se, por exemplo, que cada processador p_j, $0 \leq j < n$, armazene

uma coluna de cada matriz, ou seja, $a_{0,j}, \ldots, a_{n-1,j}$ e $b_{0,j}, \ldots, b_{n-1,j}$ estão em p_j.

Na figura abaixo está esquematizada apenas a situação inicial da matriz $\mathcal{A}$, uma vez que é análoga à de $\mathcal{B}$. Cada quadrado corresponde a um processador.

0	1	$\cdots$	$n-1$
$a_{0,0}$ $\vdots$ $a_{n-1,0}$	$a_{0,1}$ $\vdots$ $a_{n-1,1}$	$\cdots$	$a_{0,n-1}$ $\vdots$ $a_{n-1,n-1}$

O algoritmo abaixo, chamado N_1, calcula a matriz produto $\mathcal{C}$, deixando-a também armazenada por coluna. Cada processador p_j, $0 \leq j < n$, executa:

for $i := 0$ to $n-1$ do
    DM de $a_{i,j}$
    p_j calcula valor de $c_{i,j}$

Como ambos os passos do laço gastam tempo $O(n)$, o algoritmo tem complexidade $O(n^2)$. O espaço necessário em cada processador é também $O(n)$.

Portanto, cada processador contém dois vetores A e B com n posições. Inicialmente, em cada p_j, $0 \leq j < n$, $A[i] = a_{i,j}$ e $B[i] = b_{i,j}$, com $0 \leq i < n$. Conforme a sub-seção 2.5.1, $A[i](j)$ denota o elemento $A[i]$ do processador p_j. O resultado ficará em um terceiro vetor C, também com n posições. F também é um vetor de tamanho n, e *BITS* será $\{0, \ldots, \log_2 n - 1\}$. Com isso, podemos formalizar[1] o algoritmo N_1 em cada processador p_j:

N$_1$(*BITS, A, B, C)*
    for i:=0 to $n-1$ do
        $F(j) := \emptyset$
        $DM(BITS, A[i](j), F(j))$
        $C[i](j) := \sum_{i=0}^{n-1} F[i](j) * B[i](j)$

É fácil observar que existe um algoritmo N_2 análogo para o caso em que as matrizes estejam armazenadas por linha.

4.2.2 Classe NN: com $p = n^2$ processadores

O algoritmo NN_1 descrito em seguida foi sugerido em [BeT89], e supõe que cada processador armazene um elemento de cada matriz: $a_{i,j}$ e $b_{i,j}$ estão em $p_{i,j}$, $0 \leq i, j < n$.

O esquema abaixo mostra a situação inicial das matrizes $\mathcal{A}$ e $\mathcal{B}$, considerando cada quadrado como um processador:

[1] O último parâmetro, em todas as formalizações deste capítulo, será o registrador que armazenará o resultado final. Portanto, sua passagem será por referência, enquanto que os demais parâmetros são passados por valor.

	0	$\cdots$	$n-1$
0	$a_{0,0}$ $b_{0,0}$	$\cdots$	$a_{0,n-1}$ $b_{0,n-1}$
$\vdots$	$\vdots$		$\vdots$
$n-1$	$a_{n-1,0}$ $b_{n-1,0}$	$\cdots$	$a_{n-1,n-1}$ $b_{n-1,n-1}$

Essa disposição aproveita uma das propriedades da visualização matricial: as linhas e as colunas correspondem a sub-hipercubos com n processadores.

O algoritmo consiste em apenas três passos:

- DM de $a_{i,j}$ em cada linha i;
- DM de $b_{i,j}$ em cada coluna j;
- $p_{i,j}$ calcula valor de $c_{i,j}$

Cada passo gasta tempo $O(n)$, que é a complexidade final. Cada elemento da matriz $\mathcal{C}$ fica em um processador, de modo análogo às matrizes $\mathcal{A}$ e $\mathcal{B}$. Devido às duas DM, o espaço necessário em cada processador é $O(n)$.

Na formalização deste algoritmo, cada processador tem registradores A, B e C. Inicialmente, em cada $p_{i,j}$, $A = a_{i,j}$ e $B = b_{i,j}$. No final, o resultado $c_{i,j}$ ficará em C. Neste algoritmo, $BITS = \{0, \ldots, 2\log_2 n - 1\}$. Os registradores LIN e COL armazenam os elementos de $BITS$ que correspondem respectivamente aos sub-hipercubos das linhas e colunas da visualização matricial. Portanto, $LIN = \{0, \ldots, \log_2 n - 1\}$ e $COL = \{\log_2 n, \ldots, 2\log_2 n - 1\}$.

Abaixo está o algoritmo em cada processador p_j:

NN$_1$(*BITS, A, B, C)*
  $q := |BITS|/2$
  $LIN(j) := \{BITS[0], \ldots, BITS[q-1]\}$
  $COL(j) := \{BITS[q], \ldots, BITS[2q-1]\}$
  $F_1(j) := \emptyset$
  $DM(LIN(j), A(j), F_1(j))$
  $F_2(j) := \emptyset$

$DM(COL(j), B(j), F_2(j))$
$C(j) := \sum_{i=0}^{n-1} F_1[i](j) * F_2[i](j)$

Em [DNS81] há um outro algoritmo (que chamaremos de NN_2) para o caso de n^2 processadores com a mesma complexidade de tempo, mas que gasta espaço $O(\log n)$ em cada processador. Ele é uma adaptação para a visualização matricial do algoritmo de Cannon [Can69], elaborado para multiplicação de matrizes em grade toroidal.

O algoritmo NN_2 considera o mesmo armazenamento inicial de NN_1, e consiste basicamente em duas fases:

- Obter um alinhamento inicial tal que em cada processador $p_{i,j}$, $A = a_{i,r}$ e $B = b_{r,j}$, para algum valor de r. No final dessa fase, que deve gastar tempo $O(n)$, pode-se calcular uma das n parcelas de $c_{i,j}$.

- Durante $n - 1$ passos, através da comunicação com processadores vizinhos, obter novos valores para A e B que permitam calcular em cada passo uma outra parcela de $c_{i,j}$.

Com essas idéias, fica mais simples a descrição do algoritmo:

Fase 1: Na visualização $n \times n$ do $H(d)$, é possível conseguir que depois de $\log n$ passos, em cada $p_{i,j}$, $A = a_{i,j \oplus i}$ e $B = b_{i \oplus j,j}$.

Fase 2: Calcula-se uma seqüência $S_{\log_2 n-1}$ de índices dos *bits* pertencentes ao intervalo $[0, \log_2 n - 1]$, tal que a complementação dos *bits* na ordem em que estão nessa seqüência permite que se passe por todos os valores de 0 a $n - 1$.

Essa seqüência pode ser construída de modo recursivo: excetuando-se o *bit* mais siginificativo, quando se complementa um número segundo uma seqüência $S_{\log_2 n-2}$, passa-se por todos os números no intervalo $[0, \lfloor n/2 \rfloor]$. Em seguida, ao se inverter o *bit* mais significativo, basta repetir essa mesma seqüência $S_{\log_2 n-2}$ para se passar por todos os números da segunda metade do intervalo.

Considerando $S_0 = 0$, podemos então obter todas as seqüências:

$$\begin{aligned} S_1 &= 0, 1, 0 \\ S_2 &= 0, 1, 0, 2, 0, 1, 0 \\ S_3 &= 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0 \\ &\vdots \end{aligned}$$

Um exemplo: Para o intervalo $[0, 7]$, partindo-se do número $3 = (011)$ e utilizando-se a seqüência S_2, obtêm-se todos os números na seguinte ordem:

NÚMERO	*BITS*		
	2	1	0
3	0	1	1
2	0	1	0
0	0	0	0
1	0	0	1
5	1	0	1
4	1	0	0
6	1	1	0
7	1	1	1

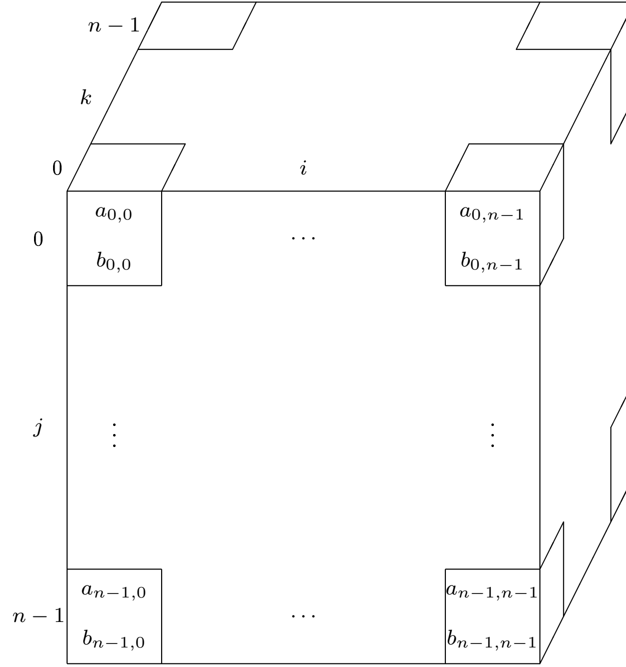
Dessa forma, ao se inverter os *bits* de um número com $\log_2 n$ *bits* seguindo a ordem determinada pela seqüência $S_{\log_2 n-1}$, passa-se sem repetição por todos os números do intervalo $[0, n-1]$. Os elementos desta seqüência podem ser gerados nesta ordem gastando-se espaço $O(\log n)$. A função $SEQ(i, n)$ retorna o i-ésimo elemento de $S_{\log_2 n-1}$.

A partir dessas considerações, podemos formalizar o algoritmo NN_2. Como em NN_1, na sua chamada $BITS = \{0, \ldots, 2\log_2 n - 1\}$. É necessário lembrar que na visualização matricial $n \times n$, os *bits* dos endereços dos processadores que variam em cada linha correspondem à primeira metade de *BITS* (menos significativos), enquanto que os que variam em cada coluna correspondem à segunda metade (mais significativos).

NN$_2$(*BITS, A, B, C***)**
- $k := |BITS|/2$
- for $i := 0$ to $k-1$ do
  - $q := i + k - 1$
  - $A(j) \leftarrow A(j^{(BITS[i])}),\ \ (j_{BITS[q]} = 1)$
  - $B(j) \leftarrow B(j^{(BITS[q])}),\ \ (j_{BITS[i]} = 1)$
- $C(j) := A(j) * B(j)$
- for $i := 1$ to $2^k - 1$ do
  - $s := SEQ(i, 2^k)$
  - $q := s + k - 1$
  - $A(j) \leftarrow A(j^{(BITS[s])})$
  - $B(j) \leftarrow B(j^{(BITS[q])})$
  - $C(j) := C(j) + A(j) * B(j)$

4.2.3 Classe NNN: com $p = n^3$ processadores

A descrição do algoritmo NNN_1 está em [DNS81]. Os processadores são considerados segundo uma disposição cúbica $n \times n \times n$, e supõe-se que as matrizes $\mathcal{A}$ e $\mathcal{B}$ estejam ambas armazenadas na face $k = 0$, um elemento por processador. A situação inicial nessa face é idêntica à situação dos n^2 processadores da classe NN.



Cada processador $p_{i,j,k}$, que corresponde a um pequeno cubo na figura acima, tem registradores A, B, A', B' e C. Considerando $R_{i,j,k}$ como o valor do registrador R de $p_{i,j,k}$, temos que inicialmente $A_{i,j,0} = a_{i,j}$ e $B_{i,j,0} = b_{i,j}$.

Passos:

- A partir de $A_{i,j,0} = a_{i,j}$, DS no eixo k: $A_{i,j,k} = a_{i,j}$;
- A partir de $B_{i,j,0} = b_{i,j}$, DS no eixo k: $B_{i,j,k} = b_{i,j}$;
- A partir de $A_{i,k,k} = a_{i,k}$, DS no eixo j: $A'_{i,j,k} = a_{i,k}$;
- A partir de $B_{k,j,k} = b_{k,j}$, DS no eixo i: $B'_{i,j,k} = b_{k,j}$;
- Cada $p_{i,j,k}$ calcula $A'_{i,j,k}B'_{i,j,k} = a_{i,k}b_{k,j}$;
- AS no eixo k para $p_{i,j,0}$ da soma dos valores calculados, obtendo $c_{i,j}$.

Todos os passos, com exceção do penúltimo, que é efetuado em tempo constante, utilizam operações que requerem tempo $O(\log n)$, sendo esta a complexidade final do algoritmo. O espaço necessário em cada processador é $O(1)$.

Formalizando: em cada $p_{i,j,0}$, temos inicialmente $A = a_{i,j}$ e $B = b_{i,j}$. Nos demais processadores, $A = B =$ null. Em todos os processadores, inicialmente $A' = B' =$ null. Nesse caso, $BITS = \{0, \ldots, 3\log_2 n - 1\}$, e I, J e K são vetores que corresponderão aos *bits* que determinam os sub-hipercubos em cada eixo.

NNN$_1$(*BITS, A, B, C)*
$q := |BITS|/3$
$I(j) := \{BITS[0], \ldots, BITS[q-1]\}$
$J(j) := \{BITS[q], \ldots, BITS[2q-1]\}$
$K(j) := \{BITS[2q], \ldots, BITS[3q-1]\}$
$DS(K(j), A(j))$
$DS(K(j), B(j))$
$A'(j) := A(j), \ (j_{q:2q-1} = j_{2q:3q-1})$
$DS(J(j), A'(j))$
$B'(j) := B(j), \ (j_{0:q-1} = j_{2q:3q-1})$
$DS(I(j), B'(j))$
$A'(j) := A'(j) * B'(j)$
$SOMA(K(j), A'(j), C(j))$

Em [BeT89] é descrita uma variante NNN_2 deste algoritmo, cuja única diferença está em considerar as matrizes $\mathcal{A}$ e $\mathcal{B}$ inicialmente dispostas em faces ortogonais do cubo, mudando assim as DS.

4.3 Os algoritmos de Dekel, Nassimi e Sahni

Em [DNS81] foram propostos algoritmos para multiplicação de matrizes que se baseiam em algumas outras visualizações matriciais do hipercubo, e que utilizam um algoritmo de NN e outro de NNN. São apresentados dois algoritmos que cobrem os casos em que o número de processadores varia desde 1 a n^3. Esses algoritmos, considerados como pertencentes às classes NNM e MM, serão apresentados a seguir.

4.3.1 Classe NNM: com $n^2 \leq p \leq n^3$ processadores

Em [DNS81] considera-se $p = n^2m$ processadores, onde n e m são potências de 2 tal que $1 \leq m \leq n$, arranjados na forma $n \times n \times m$. Inicialmente, as matrizes $\mathcal{A}$ e $\mathcal{B}$ estão armazenadas na face $n \times n$, de modo similar aos algoritmos de NNN.

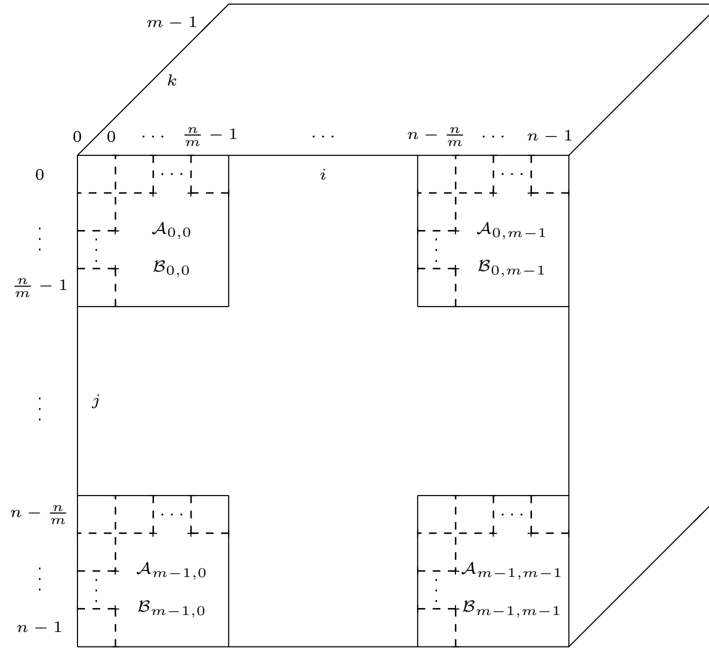
A idéia básica do algoritmo é tratar $\mathcal{A}$ e $\mathcal{B}$ como matrizes $m \times m$, cujos elementos são sub-matrizes $\frac{n}{m} \times \frac{n}{m}$:

$$\mathcal{A} = (\mathcal{A}_{i,j}) \text{ e } \mathcal{B} = (\mathcal{B}_{i,j}),$$

onde $\mathcal{A}_{i,j}$ e $\mathcal{B}_{i,j}$ são sub-matrizes de ordem n/m, com $0 \leq i, j < m$. Aliás, na descrição desses algoritmos, os índices i e j estarão sempre nesse intervalo.

Dessa forma, $\mathcal{C} = (\mathcal{C}_{i,j})$, onde $\mathcal{C}_{i,j} = \sum_{k=0}^{m-1} \mathcal{A}_{i,k}\mathcal{B}_{k,j}$. Um fato muito importante é que como m é potência de 2, $\mathcal{A}_{i,j}$ e $\mathcal{B}_{i,j}$ estão armazenadas em um sub-hipercubo com $(n/m)^2$ processadores, graças a outra das propriedades da visualização matricial.

A figura abaixo visualiza o hipercubo neste arranjo $n \times n \times m$, mostrando a situação inicial das sub-matrizes (os quadrados pontilhados correspondem aos processadores):

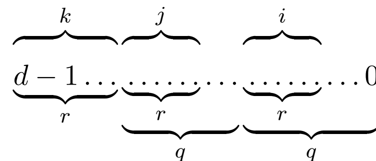


O algoritmo NNM_1 apresentado em [DNS81] consiste em seguir os passos de NNN_1, e utiliza NN_2 como sub-algoritmo para obter o produto entre sub-matrizes:

- Aplicam-se as quatro DS descritas em NNN_1 considerando as sub-matrizes $\mathcal{A}_{i,j}$ e $\mathcal{B}_{i,j}$ como elementos. Tempo necessário: $O(\log m)$, graças à propriedade dos elementos correspondentes da visualização matricial.
- No final do passo anterior, cada par de sub-matrizes $\mathcal{A}_{i,k}$ e $\mathcal{B}_{k,j}$ estará em um mesmo grupo de $(n/m)^2$ processadores. O produto entre essas sub-matrizes pode ser feito em tempo $O(n/m)$ utilizando NN_2 (na verdade, poder-se-ia utilizar também NN_1).
- A somatória necessária para calcular cada sub-matriz $\mathcal{C}_{i,j}$ é feita com uma AS no eixo k, conforme a descrição de NNN_1. O tempo gasto também é $O(\log m)$.

Portanto, o algoritmo NNM_1 gasta tempo total $O(\log m + n/m)$. Considerando $p = n^2m$, o tempo é $O(\log \frac{p}{n^2} + \frac{n^3}{p})$. Cada processador precisa de apenas espaço $O(1)$.

Para formalizarmos esse algoritmo, é necessário observar quais dos d *bits* do endereço de cada processador estão relacionados com as direções i, j e k. Considerando $q = \log_2 n$ e $r = \log_2 m$, a figura abaixo mostra dentre os d *bits* dos processadores de um mesmo grupo, quais variam ao longo das três direções:



Portanto, é necessário passar como parâmetro o valor de $r = \log_2 m$, para se poder fazer a divisão dos *bits*. Além disso, $BITS = \{0, \ldots, 2\log_2 n + \log_2 m - 1\}$.

NNM$_1$(*BITS, r, A, B, C)*
$q := (|BITS| - r)/2$
$I(j) := \{BITS[q-r], \ldots, BITS[q-1]\}$
$J(j) := \{BITS[2q-r], \ldots, BITS[2q-1]\}$
$K(j) := \{BITS[2q], \ldots, BITS[2q+r-1]\}$
$DS(K(j), A(j))$
$DS(K(j), B(j))$
$A'(j) := A(j), \ (j_{2q-r:2q-1} = j_{2q:2q+r-1})$
$DS(J(j), A'(j))$
$B'(j) := B(j), \ (j_{q-r:q-1} = j_{2q:2q+r-1})$
$DS(I(j), B'(j))$
$BITS2 := \{0, \ldots, q-r-1, q, \ldots, 2q-r-1\}$
$NN_2(BITS2, A', B', C)$
$A'(j) := C(j)$
$SOMA(K(j), A'(j), C(j))$

Caso utilizemos NN_1 como sub-algoritmo, temos um novo algoritmo NNM_2 com a mesma complexidade.

Vale a pena considerar alguns casos particulares:

p	m	COMPLEXIDADE	OBSERVAÇÕES
n^2	1	$O(n)$	Equivalente a NN_1.
n^3	n	$O(\log n)$	Equivalente a NNN_1.
$n^3/\log n$	$n/\log n$	$O(\log n)$	Mesmo tempo de NNN_1.

4.3.2 Classe MM: com $1 \leq p \leq n^2$ processadores

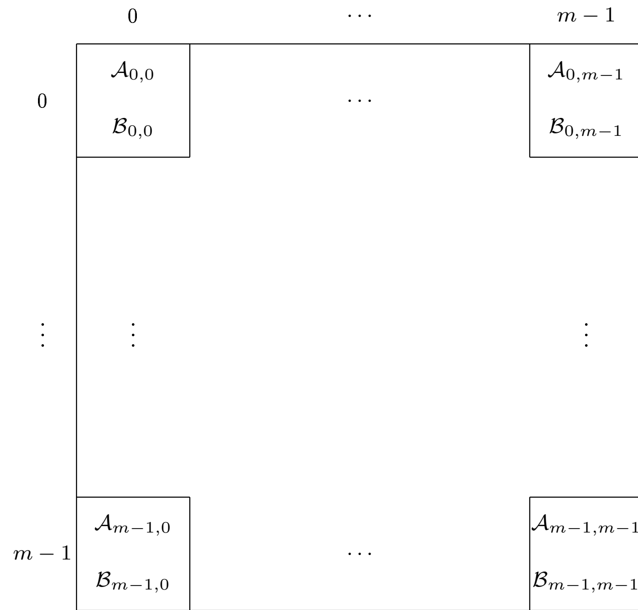
A mesma idéia descrita na classe anterior é novamente utilizada, agora considerando $p = m^2$, onde $1 \leq m \leq n$, e aplicando os passos de um algoritmo de NN tomando as sub-matrizes como elementos.

Inicialmente, cada processador $p_{i,j}$ armazena as sub-matrizes $\mathcal{A}_{i,j}$ e $\mathcal{B}_{i,j}$, $0 \leq i, j < m$, ou seja, contém $2(n/m)^2$ elementos. Novamente os índices i e j estarão sempre neste intervalo durante a descrição do algoritmo.

Baseando-se em NN_1, pode-se aplicar seus passos às sub-matrizes de $\mathcal{A}$ e $\mathcal{B}$, obtendo-se assim o algoritmo MM_1:

- DM de $\mathcal{A}_{i,j}$ em cada linha i. O tempo gasto é $O(m\frac{n^2}{m^2}) = O(n^2/m)$.
- DM de $\mathcal{B}_{i,j}$ em cada coluna j. O tempo gasto é o mesmo do passo anterior: $O(n^2/m)$.
- m multiplicações e $m-1$ adições de sub-matrizes $\frac{n}{m} \times \frac{n}{m}$ em cada $p_{i,j}$ para cálculo de $\mathcal{C}_{i,j}$. Neste passo pode-se utilizar o melhor algoritmo seqüencial conhecido para multiplicação de matrizes de ordem n que tenha complexidade $O(n^\lambda) = o(n^3)$. Com isso, o tempo gasto passa a ser $O(m(\frac{n}{m})^\lambda + m(\frac{n}{m})^2) = O(m(\frac{n}{m})^\lambda)$, uma vez que $\lambda \geq 2$.

Portanto, o tempo total de MM_1 é $O(n^\lambda/m^{\lambda-1}) = O(n^\lambda/p^{\frac{\lambda-1}{2}})$.



Para facilitar a formalização deste algoritmo, vamos supor que A, B e C são "registradores" (mais precisamente, conjuntos de registradores) que armazenam matrizes quadradas com $(n/m)^2$ elementos cada; *MULTSEQ* é uma função que recebe como argumento dois desses registradores e faz a multiplicação seqüencial de matrizes quadradas de ordem n/m em tempo $O((n/m)^\lambda)$, devolvendo a matriz resultado no mesmo formato; e os elementos dos vetores F_1 e F_2 são matrizes quadradas de ordem n/m. Na chamada de MM_1, *BITS* $= \{0, \ldots, 2\log_2 m - 1\}$.

MM$_1$**(***BITS, A, B, C)*
  $q := |BITS|/2$
  $LIN(j) := \{BITS[0], \ldots, BITS[q-1]\}$
  $COL(j) := \{BITS[q], \ldots, BITS[2q-1]\}$
  $F_1(j) := \emptyset$
  $DM(LIN(j), A(j), F_1(j))$
  $F_2(j) := \emptyset$
  $DM(COL(j), B(j), F_2(j))$
  $C(j) := \sum_{i=0}^{n-1} MULTSEQ(F_1[i](j), F_2[i](j))$

O algoritmo sugerido em [DNS81], que poderíamos chamar de MM_2, baseia-se em NN_2, e tem a mesma complexidade.

Alguns casos particulares interessantes:

p	m	COMPLEXIDADE	OBSERVAÇÕES
n	$n^{\frac{1}{2}}$	$O(n^{\frac{\lambda+1}{2}})$	Melhor do que N.
n^2	n	$O(n)$	Equivalente a NN.

4.4 Situação atual

Os resultados até aqui obtidos são os melhores conhecidos para se efetuar a multiplicação de duas matrizes quadradas de ordem n num hipercubo com p processadores, onde $1 \leq p \leq n^3$. Como os algoritmos básicos, correspondentes às classes N, NN e NNN, tornam-se casos particulares (em termos de complexidade) dos algoritmos apresentados em [DNS81], a tabela abaixo resume a situação atual (colocamos apenas um representante para cada complexidade):

PROCESSADORES	ALGORITMO	COMPLEXIDADE
$1 \leq p \leq n^2$	MM_1	$O(n^\lambda / p^{\frac{\lambda-1}{2}})$
$n^2 \leq p \leq n^3$	NNM_1	$O(\log \frac{p}{n^2} + \frac{n^3}{p})$

Johnsson [Joh87] apresenta também alguns algoritmos para multiplicação de matrizes não quadradas no $H(d)$. Sendo $\mathcal{A}$ uma matriz $n_1 \times n_2$ e $\mathcal{B}$ uma matriz $n_2 \times n_3$, seus melhores algoritmos, em termos de ordem, têm complexidades $O(n_1 n_2 n_3 / p + \log p)$. Pode-se observar que no caso particular de matrizes quadradas, isto é, $n_1 = n_2 = n_3 = n$, a complexidade $O(n^3/p + \log p)$ não é melhor que as dos algoritmos acima.

4.5 Uma visão global

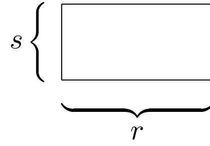
Analisando de um modo mais global as classes de algoritmos que foram descritas até o momento, pode-se chegar à conclusão de que se baseiam em duas maneiras de se arranjar os processadores de forma a permitir o emprego de algoritmos conhecidos: arranjos que poderíamos chamar de bi-dimensional e tri-dimensional.

Para esses dois arranjos são conhecidos alguns algoritmos básicos, como N_1, NN_1, NN_2 e NNN_1, entre outros. Tais algoritmos podem ser adaptados quando sub-matrizes são consideradas como elementos, aproveitando-se uma propriedade do cálculo da multiplicação de duas matrizes, dando origem assim a diversas classes de algoritmos.

A seguir será feita uma descrição sucinta desses arranjos, e quais classes foram ou podem ser obtidas através deles. As classes que estão indicadas como novas serão estudadas na próxima seção. A notação utilizada é a mesma da seção anterior, isto é, as matrizes são de ordem n, e m obedece a $1 \leq m \leq n$.

4.5.1 Arranjo bi-dimensional

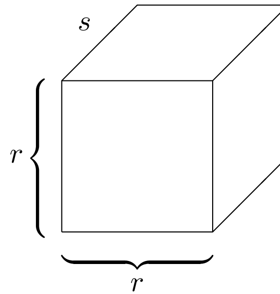
Neste primeiro caso, o hipercubo é visto como rs processadores dispostos segundo uma forma matricial de duas dimensões, onde r é o número de colunas e s o número de linhas.



$$p = rs \begin{cases} r = s \{ \ r = s = m: \ m^2 \text{ processadores} \rightarrow MM \\ r > s \begin{cases} r = n: \ nm \text{ processadores} \rightarrow NM \text{ (nova)} \\ r = m \text{ e } s = 1: \ m \text{ processadores} \rightarrow M \text{ (nova)} \end{cases} \\ r < s \{ \ \text{Pode ser considerado análogo ao caso anterior.} \end{cases}$$

4.5.2 Arranjo tri-dimensional

No segundo caso, são r^2s processadores do hipercubo considerados como uma matriz cúbica com r processadores na duas primeiras dimensões e s na última.



$$p = r^2s \begin{cases} r = s \{ \ r = s = m: \ m^3 \text{ processadores} \rightarrow MMM \text{ (nova)} \\ r > s \begin{cases} r = n: \ n^2m \text{ processadores} \rightarrow NNM \\ r = m \text{ e } s = 1: \ m^2 \text{ processadores} \rightarrow MM \end{cases} \\ r < s \{ \ \text{Não é interessante.} \end{cases}$$

4.6 As novas classes de algoritmos

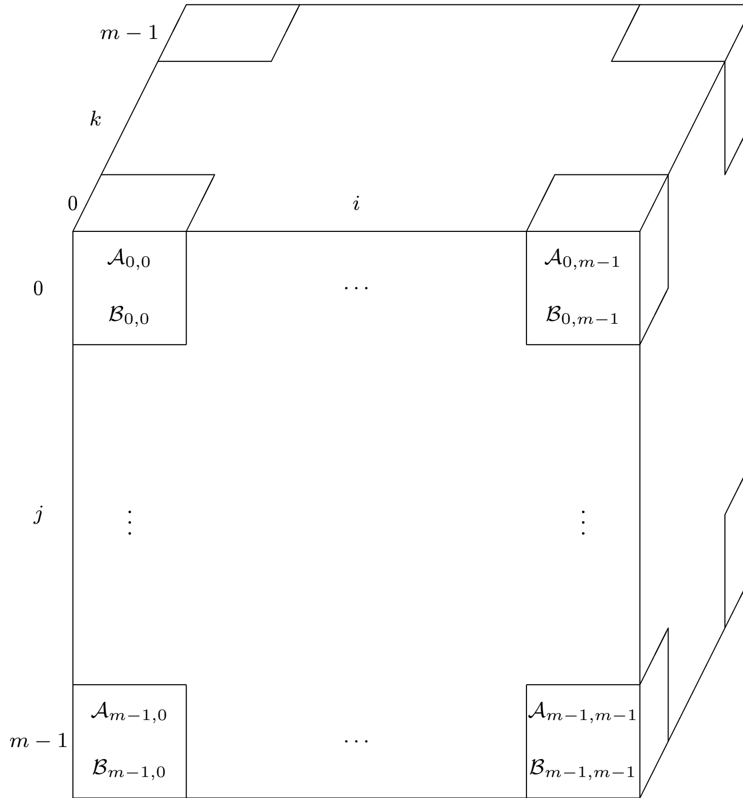
Conforme indicado na seção anterior, podem ser estudadas três novas classes de algoritmos: MMM, NM e M. Além disso, veremos que numa mesma classe pode haver algoritmos com complexidades distintas. Por esse motivo, apresentaremos também um novo algoritmo para a classe NNM: o algoritmo NNM_3.

4.6.1 Classe MMM: com $1 \leq p \leq n^3$ processadores

Os $p = m^3$ processadores, onde m é potência de 2 tal que $1 \leq m \leq n$, são considerados dentro de um arranjo cúbico de lado m. Inicialmente, cada processador $p_{i,j,0}$ contém as sub-matrizes $\mathcal{A}_{i,j}$ e $\mathcal{B}_{i,j}$, $0 \leq i, j < m$, que são de ordem n/m, correspondendo a um total

de $2(n/m)^2$ elementos. O armazenamento inicial, considerando somente a face do cubo formada pelos processadores $p_{i,j,0}$, é análogo ao dos algoritmos MM_1 e MM_2.

Como existem m^3 processadores, faremos a adaptação de um dos algoritmos da classe NNN, no caso o NNN_1. Esse novo algoritmo será chamado MMM_1.



Passos:

- Aplicam-se as quatro DS descritas em NNN_1 considerando as sub-matrizes como elementos. O tempo gasto é $O(\frac{n^2}{m^2} \log m)$.
- Em cada processador $p_{i,j,k}$ é calculado o produto entre as sub-matrizes $\mathcal{A}_{i,k}$ e $\mathcal{B}_{k,j}$, utilizando-se o melhor algoritmo seqüencial conhecido. Tempo necessário: $O((\frac{n}{m})^{\lambda})$.
- Efetua-se a AS para cálculo da somatória. O tempo gasto é o mesmo do primeiro passo: $O(\frac{n^2}{m^2} \log m)$.

O tempo total de MMM_1 é $O(\frac{n^2}{m^2} \log m + (\frac{n}{m})^{\lambda}) = O(\frac{n^2}{p^{2/3}} \log p + \frac{n^{\lambda}}{p^{\lambda/3}})$.

Consideraremos as mesmas suposições para a descrição de MM_1 com relação aos "registradores" A, B, C, A' e B' (cada um dos quais armazenando $(n/m)^2$ elementos). Na

formalização abaixo, *SOMA2* é o mesmo algoritmo *SOMA*, com a diferença de que adiciona um a um os $(n/m)^2$ elementos correspondentes armazenados em cada registrador. Na chamada de MMM_1, $BITS = \{0, \ldots, 3\log_2 m - 1\}$.

MMM$_1$(*BITS, A, B, C*)
$q := |BITS|/3$
$I(j) := \{BITS[0], \ldots, BITS[q-1]\}$
$J(j) := \{BITS[q], \ldots, BITS[2q-1]\}$
$K(j) := \{BITS[2q], \ldots, BITS[3q-1]\}$
$DS(K(j), A(j))$
$DS(K(j), B(j))$
$A'(j) := A(j), \ (j_{q:2q-1} = j_{2q:3q-1})$
$DS(J(j), A'(j))$
$B'(j) := B(j), \ (j_{0:q-1} = j_{2q:3q-1})$
$DS(I(j), B'(j))$
$A'(j) := MULTSEQ(A'(j), B'(j))$
$SOMA2(K(j), A'(j), C(j))$

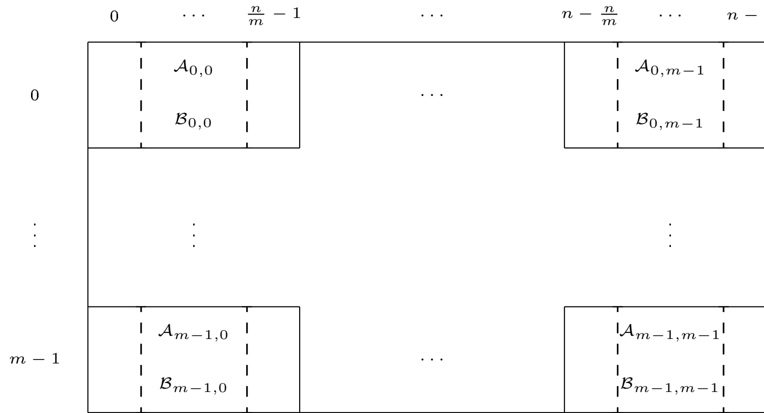
Alguns casos particulares interessantes:

p	m	COMPLEXIDADE	OBSERVAÇÕES
n	$n^{1/3}$	$O(n^{4/3}\log n + n^{2\lambda/3})$	Melhor do que N_1 e MM_1.
n^2	$n^{2/3}$	$O(n^{2/3}\log n + n^{\lambda/3})$	Melhor do que NN_1 e NNM_1.
n^3	n	$O(\log n)$	Equivalente a NNN_1.

4.6.2 Classe NM: com $n \leq p \leq n^2$ processadores

Considerando $p = nm$, onde $1 \leq m \leq n$, os processadores são vistos como um arranjo matricial de n colunas e m linhas. Inicialmente, cada grupo de n/m processadores consecutivos numa mesma linha (existem portanto m grupos em cada linha), que corresponde a um sub-hipercubo quando m é potência de 2, contém uma sub-matriz de ordem n/m de $\mathcal{A}$ e outra de $\mathcal{B}$. Dessa forma, há uma analogia entre essa situação e a situação inicial da classe NN.

A figura abaixo mostra como estão inicialmente as sub-matrizes no hipercubo (as linhas pontilhadas separam os processadores). Dependendo de como é feito o armazenamento inicial das sub-matrizes, surgem três algoritmos: NM_1, NM_2 e NM_3.



Adaptando-se o algoritmo NN_1 à situação acima, pode-se notar que as sub-matrizes de ordem n/m serão multiplicadas em grupos de n/m processadores. Para se efetuar essa operação, contamos com algoritmos de três classes: N, MM e MMM. Tais algoritmos supõem armazenamentos iniciais diferentes das matrizes, sendo que aquele que armazena mais elementos por processador (portanto, que ocasionará maior tempo de comunicação) possui melhor complexidade. Os casos serão analisados separadamente, dando origem a três algoritmos com complexidades distintas.

Algoritmo NM_1: utilizando N_1 como sub-algoritmo. Esse é o caso mais simples: cada $p_{i,j}$, com $0 \leq i < m$ e $0 \leq j < n$, armazena uma coluna de cada sub-matriz, num total de n/m elementos. Dessa forma, há informação em todos os processadores.

Seguindo os passos de NN_1, cada DM gastará tempo $O(\frac{n}{m}m) = O(n)$. É importante notar que nas linhas a DM é feita nos hipercubos determinados pelos elementos correspondentes. Como em cada grupo serão feitos m produtos de sub-matrizes de ordem n/m utilizando N_1, gastar-se-á tempo $O(m(\frac{n}{m})^2) = O(n^2/m)$. Por fim, cada processador deverá somar as m sub-colunas resultantes: $O(m\frac{n}{m}) = O(n)$. O tempo total é portanto $O(n^2/m) = O(n^3/p)$.

Alguns casos particulares interessantes:

p	m	COMPLEXIDADE	OBSERVAÇÕES
n	1	$O(n^2)$	Equivalente a N_1.
n^2	n	$O(n)$	Equivalente a NN_1.

Algoritmo NM_2: utilizando MM_1 como sub-algoritmo.[2] Consideraremos cada um dos grupos de n/m processadores, que é um sub-hipercubo, como um arranjo $m' \times m'$. Isso significa que $m' = (n/m)^{1/2}$. Cada processador, de modo análogo a MM_1, deve armazenar uma sub-matriz de ordem $(n/m)/m' = (n/m)^{1/2}$, isto é, n/m elementos.

Voltando aos passos de NN_1, cada DM gastará tempo $O(\frac{n}{m}m) = O(n)$, os m produtos em cada grupo utilizando MM_1 custarão $O(m(\frac{n}{m})^{\frac{\lambda+1}{2}}) = O(n^{\frac{\lambda+1}{2}}/m^{\frac{\lambda-1}{2}})$, e a somatória final $O(m\frac{n}{m}) = O(n)$. Portanto, o tempo total é $O(n + n^{\frac{\lambda+1}{2}}/m^{\frac{\lambda-1}{2}}) = O(n^{\frac{\lambda+1}{2}}/m^{\frac{\lambda-1}{2}}) =$

[2] Esse algoritmo pode ser aplicado quando n/m é quadrado perfeito.

$O(n^{\lambda}/p^{\frac{\lambda-1}{2}})$, uma vez que $2 \leq \lambda < 3$. Um fato interessante é que a complexidade é a mesma de MM_1.

Alguns casos particulares interessantes:

p	m	COMPLEXIDADE	OBSERVAÇÕES
n	1	$O(n^{\frac{\lambda+1}{2}})$	Melhor do que N_1 e NM_1.
n^2	n	$O(n)$	Equivalente a NN_1 e NM_1.

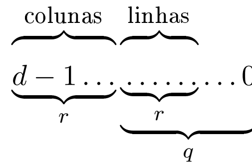
Algoritmo NM_3: utilizando MMM_1 como sub-algoritmo.[3] Neste último caso, cada um dos grupos de n/m processadores será visto como um arranjo cúbico $m' \times m' \times m'$. Logo, $m' = (n/m)^{1/3}$. De modo análogo a MMM_1, as sub-matrizes correspondentes a cada grupo deverão estar armazenadas numa face desse arranjo, ou seja, cada processador dessa face armazena uma sub-matriz da sub-matriz, que tem ordem $(n/m)/m' = (n/m)^{2/3}$, correspondendo a um total de $(n/m)^{4/3}$ elementos.

Aplicando os passos de NN_1, cada DM custará $O((n/m)^{4/3}m)$, as m multiplicações em cada grupo utilizando o algoritmo MMM_1 gastarão tempo $O(m((n/m)^{4/3}\log\frac{n}{m} + (n/m)^{2\lambda/3}))$, e a somatória $O(m(n/m)^{4/3})$. O tempo total deve-se às multiplicações: $O(m((n/m)^{4/3}\log\frac{n}{m} + (n/m)^{2\lambda/3})) = O(\frac{n^{5/3}}{p^{1/3}}\log\frac{n^2}{p} + n^{\frac{4\lambda-3}{3}}/p^{\frac{2\lambda-3}{3}}) = O(\frac{n^{5/3}}{p^{1/3}}\log\frac{n^2}{p} + \frac{n^{\lambda}}{p^{\lambda/3}}(\frac{p}{n})^{1-\lambda/3})$.

Alguns casos particulares interessantes:

p	m	COMPLEXIDADE	OBSERVAÇÕES
n	1	$O(n^{4/3}\log n + n^{2\lambda/3})$	Melhor do que N_1, NM_1 e NM_2.
n^2	n	$O(n)$	Equivalente a NN_1, NM_1 e NM_2.

Na formalização desses algoritmos, devemos observar quais dos d *bits* dos endereços dos processadores de cada grupo estão relacionados com as direções da visualização $m \times m$. Considerando $q = \log_2 n$ e $r = \log_2 m$, a figura abaixo mostra dentre os d *bits* quais estão variando ao longo das linhas e colunas:



Portanto, também é necessário passar como parâmetro o valor de $r = \log_2 m$, para se poder fazer a divisão dos *bits*. Nesse caso, $BITS = \{0, \ldots, \log_2 n + \log_2 m - 1\}$. Para facilitar a descrição, vamos supor novamente que os registradores do algoritmo abaixo armazenam sub-matrizes. Utilizaremos também um registrador auxiliar AUX.

NM$_i$(*BITS, r, A, B, C*)
    $q := |BITS| - r$

[3] Esse algoritmo pode ser aplicado quando n/m é cubo perfeito.

$LIN(j) := \{BITS[q-r], \ldots, BITS[q-1]\}$
$COL(j) := \{BITS[q], \ldots, BITS[q+r-1]\}$
$F_1(j) := \emptyset$
$DM(LIN(j), A(j), F_1(j))$
$F_2(j) := \emptyset$
$DM(COL(j), B(j), F_2(j))$
$BITS2 := \{0, \ldots, q-r-1\}$
for $i := 0$ to $2^{q-r}-1$ do
    $X(BITS2, F_1[i](j), F_2[i](j), AUX(j))$
    $C(j) := C(j) + AUX(j)$

Como se supõe armazenamentos iniciais diferentes em cada um dos três algoritmos, a única coisa que muda na formalização é a chamada do sub-algoritmo X. Para NM_1, X é N_1; para NM_2, X é MM_1; e para NM_3, X é MMM_1. É claro que a soma na última linha do algoritmo é feita entre os elementos correspondentes de C e AUX, pois estes registradores armazenam sub-matrizes.

4.6.3 Classe M: com $1 \leq p \leq n$ processadores

Para manter uma uniformidade na notação, vamos considerar $p = m$, onde $1 \leq m \leq n$. Esses processadores são vistos de uma maneira linear. Inicialmente, supomos que cada um deles contenha uma coluna de sub-matrizes de $\mathcal{A}$ e de $\mathcal{B}$, num total de n^2/m elementos.

Esquema inicial para a matriz $\mathcal{A}$, uma vez que o armazenamento de $\mathcal{B}$ é análogo:

0	1	$\cdots$	$m-1$
$\mathcal{A}_{0,0}$ $\vdots$ $\mathcal{A}_{m-1,0}$	$\mathcal{A}_{0,1}$ $\vdots$ $\mathcal{A}_{m-1,1}$	$\cdots$	$\mathcal{A}_{0,m-1}$ $\vdots$ $\mathcal{A}_{m-1,m-1}$

Dessa forma, pode-se deduzir um algoritmo que se baseia em N_1:

for $i := 0$ to $m-1$ do
    DM de $\mathcal{A}_{i,j}$;
    p_j calcula valor de $\mathcal{C}_{i,j}$

A primeira linha do laço gasta tempo $O(\frac{n^2}{m^2}m) = O(n^2/m)$, enquanto que a segunda gasta $O(m\frac{n^2}{m^2} + m(\frac{n}{m})^\lambda) = O(n^\lambda/m^{\lambda-1})$. Como há m passos, o tempo total é $O(n^2 + n^\lambda/m^{\lambda-2}) = O(n^\lambda/m^{\lambda-2}) = O(n^\lambda/p^{\lambda-2})$.

Considerando agora que os registradores A, B e C sejam vetores de tamanho m, e seus elementos sejam matrizes quadradas de ordem n/m, o mesmo ocorrendo com o vetor F, podemos formalizar o algoritmo M_1. Na sua chamada, $BITS = \{0, \ldots, \log_2 m\}$.

M$_1$**(***BITS, A, B, C)*
  for i:=0 to $m-1$ do
    $F(j) := \emptyset$
    $DM(BITS, A[i](j), F(j))$
    $C[i](j) := MULTSEQ(F[i](j), B[i](j))$

É fácil notar que quando o armazenamento inicial das sub-matrizes for por linhas, é possível adaptar o algoritmo N_2. Dessa forma, surge o algoritmo M_2, com a mesma complexidade que M_1.

Um caso particular interessante ocorre quando $p = n$, onde o tempo gasto é $O(n^2)$: o mesmo de N_1.

4.6.4 Algoritmo NNM_3: um outro algoritmo de NNM

Relembrando a descrição de NNM_1, a idéia básica é visualizar um arranjo $n \times n \times m$ como um novo arranjo $m \times m \times m$, onde os elementos da face $n \times n$ passam a ser sub-matrizes de ordem n/m, que correspondem a $(n/m)^2$ processadores que, por sua vez, formam um sub-hipercubo. Com isso, é possível adaptar o algoritmo NNN_1 a essa situação. Nesta adaptação, se torna necessário multiplicar essas sub-matrizes entre si, e se aplica então o algoritmo NN_1. Entretanto, pudemos verificar que, para $p = n^2$, MMM_1 tem melhor complexidade do que NN_1, exigindo porém uma disposição inicial diferente das matrizes. Entretanto, alguns processadores armazenarão mais dados, aumentando assim o tempo de comunicação. NNM_3 surge a partir desta consideração[4].

Cada grupo de $(n/m)^2$ processadores correspondente a uma sub-matriz será visto como um cubo $m' \times m' \times m'$. Portanto, $m' = (n/m)^{2/3}$. Cada processador pertencente à face que inicialmente armazena as sub-matrizes conterá uma sub-matriz de ordem $(n/m)/m' = (n/m)^{1/3}$, ou seja, $(n/m)^{2/3}$ elementos.

Dessa forma, basta seguir os passos de NNN_1: cada DS custará $O((n/m)^{2/3} \log m)$, as multiplicações são realizadas através de MMM_1 gastando tempo $O((n/m)^{2/3} \log \frac{n}{m} + (n/m)^{\lambda/3})$, e a somatória final gastará tempo da mesma ordem que o primeiro passo. O tempo total é portanto $O((n/m)^{2/3}\ (\log m + \log \frac{n}{m}) + (n/m)^{\lambda/3}) = O(\frac{n^2}{p^{2/3}} \log n + \frac{n^\lambda}{p^{\lambda/3}})$.

No intervalo $n^2 \leq p \leq n^3$, temos que $\log n = \Theta(\log p)$. Portanto, neste intervalo, NNM_3 tem complexidade da mesma ordem que MMM_1.

A formalização de NNM_3 é análoga à de NNM_1, e supõe que os registradores armazenem sub-matrizes de ordem $(n/m)^{1/3}$. Por outro lado, requer, como foi dito, um outro armazenamento inicial.

NNM$_3$**(***BITS, r, A, B, C)*
    $q := (|BITS| - r)/2$
    $I(j) := \{BITS[q-r], \ldots, BITS[q-1]\}$
    $J(j) := \{BITS[2q-r], \ldots, BITS[2q-1]\}$
    $K(j) := \{BITS[2q], \ldots, BITS[2q+r-1]\}$
    $DS(K(j), A(j))$

[4]Esse algoritmo pode ser aplicado quando $(n/m)^2$ é cubo perfeito.

$DS(K(j), B(j))$
$A'(j) := A(j), \ \ (j_{2q-r:2q-1} = j_{2q:2q+r-1})$
$DS(J(j), A'(j))$
$B'(j) := B(j), \ \ (j_{q-r:q-1} = j_{2q:2q+r-1})$
$DS(I(j), B'(j))$
$BITS2 := \{0, \ldots, q-r-1, q, \ldots, 2q-r-1\}$
$MMM_1(BITS2, A', B', C)$
$A'(j) := C(j)$
$SOMA2(K(j), A'(j), C(j))$

4.7 Resultados

Como os algoritmos básicos podem ser considerados casos particulares de alguns dos algoritmos descritos, e alguns dos algoritmos sugeridos não representam novidades em termos de complexidade, temos o seguinte conjunto de resultados, onde somente os dois primeiros [DNS81] não são originais:

ALGORITMO	PROCESSADORES	COMPLEXIDADE
NNM_1	$n^2 \leq p \leq n^3$	$O(\log \frac{p}{n^2} + \frac{n^3}{p})$
MM_1	$1 \leq p \leq n^2$	$O(n^\lambda / p^{\frac{\lambda-1}{2}})$
MMM_1	$1 \leq p \leq n^3$	$O(\frac{n^2}{p^{2/3}} \log p + \frac{n^\lambda}{p^{\lambda/3}})$
NM_1	$n \leq p \leq n^2$	$O(n^3/p)$
NM_2	$n \leq p \leq n^2$	$O(n^\lambda / p^{\frac{\lambda-1}{2}})$
NM_3	$n \leq p \leq n^2$	$O(\frac{n^{5/3}}{p^{1/3}} \log \frac{n^2}{p} + \frac{n^\lambda}{p^{\lambda/3}} (\frac{p}{n})^{1-\lambda/3})$
M_1	$1 \leq p \leq n$	$O(n^\lambda / p^{\lambda-2})$
NNM_3	$n^2 \leq p \leq n^3$	$O(\frac{n^2}{p^{2/3}} \log n + \frac{n^\lambda}{p^{\lambda/3}})$

Agrupando os algoritmos por intervalos e obtendo suas complexidades nos casos em que o número de processadores é o mesmo dos algoritmos básicos, podemos ter uma melhor visão comparativa:

PROCESSADORES	*ALGORITMO*	$p = n$	$p = n^2$	$p = n^3$
$1 \leq p \leq n$	M_1	$O(n^2)$		
	MM_1	$O(n^{\frac{\lambda+1}{2}})$		
	MMM_1	$O(n^{4/3} \log n + n^{2\lambda/3})$		
$n \leq p \leq n^2$	MM_1	$O(n^{\frac{\lambda+1}{2}})$	$O(n)$	
	NM_1	$O(n^2)$	$O(n)$	
	NM_2	$O(n^{\frac{\lambda+1}{2}})$	$O(n)$	
	NM_3	$O(n^{4/3} \log n + n^{2\lambda/3})$	$O(n)$	
	MMM_1	$O(n^{4/3} \log n + n^{2\lambda/3})$	$O(n^{2/3} \log n + n^{\lambda/3})$	
$n^2 \leq p \leq n^3$	NNM_1		$O(n)$	$O(\log n)$
	NNM_3		$O(n^{2/3} \log n + n^{\lambda/3})$	$O(\log n)$
	MMM_1		$O(n^{2/3} \log n + n^{\lambda/3})$	$O(\log n)$

4.8 Comparações

Nesta seção mostraremos quais são os algoritmos que, a partir de todos os resultados obtidos, possuem melhor complexidade, considerando que o número de processadores varia desde 1 a n^3.

Para simplificar a notação, a complexidade do algoritmo X será referida como $O(X)$. Por exemplo, se a complexidade do algoritmo X calculada nas seções anteriores é $O(n^2)$, então escreveremos $O(X) = O(n^2)$ ou $X = n^2$.

Os algoritmos descritos cobrem intervalos do número p de processadores do hipercubo. Esses intervalos dependem da ordem n das matrizes $\mathcal{A}$ e $\mathcal{B}$. Portanto, torna-se necessário considerar p **como uma função de** n para assim ser possível efetuarmos as comparações. Faremos a seguinte suposição: quando dissermos que $g(n) \leq p \leq h(n)$, isso significará que $p = f(n)$ tal que $g(n) \leq f(n) \leq h(n)$ para qualquer n positivo. Portanto, $f(n) = \Omega(g(n))$ e $f(n) = O(h(n))$.

A partir dessas considerações, podemos estabelecer um critério de comparação. Dados dois algoritmos X e Y definidos num mesmo intervalo do número de processadores, X será considerado melhor do que Y se:

- $X = O(Y)$ neste intervalo;
- houver pelo menos algum caso do número de processadores neste intervalo onde $X = o(Y)$.

4.8.1 NM_1 versus NM_2 versus NM_3

Inicialmente, iremos comparar as três versões do algoritmo NM, que cobrem os casos em que $n \leq p \leq n^2$.

Conforme a notação descrita acima, temos que:

$$\begin{aligned} &NM_1 = n^3/p, \\ &NM_2 = n^\lambda/p^{\frac{\lambda-1}{2}}, \\ &NM_3 = \tfrac{n^{5/3}}{p^{1/3}} \log \tfrac{n^2}{p} + n^{\frac{4\lambda-3}{3}}/p^{\frac{2\lambda-3}{3}}. \end{aligned}$$

Comparação entre NM_1 e NM_2:

É fácil observar que $NM_2 = O(NM_1)$ se $p = O(n^2)$:

$$\begin{aligned} NM_2 = O(NM_1) \quad &\Leftrightarrow \quad \frac{n^\lambda}{p^{\frac{\lambda-1}{2}}} = O(\frac{n^3}{p}) \\ &\Leftrightarrow \quad \frac{n^\lambda}{(n\frac{p}{n})^{\frac{\lambda-1}{2}}} = O(\frac{n^3}{n\frac{p}{n}}) \\ &\Leftrightarrow \quad \frac{n^{\frac{\lambda+1}{2}}}{(\frac{p}{n})^{\frac{\lambda-1}{2}}} = O(\frac{n^2}{\frac{p}{n}}) \end{aligned}$$

$$\begin{aligned}
&\Leftrightarrow \quad \frac{\frac{p}{n}}{(\frac{p}{n})^{\frac{\lambda-1}{2}}} = O(\frac{n^2}{n^{\frac{\lambda+1}{2}}}) \\
&\Leftrightarrow \quad (\frac{p}{n})^{\frac{3-\lambda}{2}} = O(n^{\frac{3-\lambda}{2}}) \\
&\Leftrightarrow \quad \frac{p}{n} = O(n), \quad \text{pois} \quad 2 \leq \lambda < 3 \quad \text{e} \quad \frac{p}{n} \geq 1 \\
&\Leftrightarrow \quad p = O(n^2).
\end{aligned}$$

Para o caso em que $p = n$, $NM_2 = n^{\frac{\lambda+1}{2}}$ e $NM_1 = n^2$. Como $n^{\frac{\lambda+1}{2}} = o(n^2)$ para $2 \leq \lambda < 3$, $NM_2 = o(NM_1)$ quando $p = n$.

Conclusão: NM_2 é melhor do que NM_1.

Comparação entre NM_2 e NM_3:

Fatos importantes:

(a) $n^{\frac{4\lambda-3}{3}}/p^{\frac{2\lambda-3}{3}} = O(n^\lambda/p^{\frac{\lambda-1}{2}}) \Leftrightarrow p = O(n^2)$.

Demonstração:

$$\begin{aligned}
\frac{n^{\frac{4\lambda-3}{3}}}{p^{\frac{2\lambda-3}{3}}} = O(\frac{n^\lambda}{p^{\frac{\lambda-1}{2}}}) \quad &\Leftrightarrow \quad \frac{n^{\frac{4\lambda-3}{3}}}{(n\frac{p}{n})^{\frac{2\lambda-3}{3}}} = O(\frac{n^\lambda}{(n\frac{p}{n})^{\frac{\lambda-1}{2}}}) \\
&\Leftrightarrow \quad \frac{n^{2\lambda/3}}{(\frac{p}{n})^{\frac{2\lambda-3}{3}}} = O(\frac{n^{\frac{\lambda+1}{2}}}{(\frac{p}{n})^{\frac{\lambda-1}{2}}}) \\
&\Leftrightarrow \quad \frac{(\frac{p}{n})^{\frac{\lambda-1}{2}}}{(\frac{p}{n})^{\frac{2\lambda-3}{3}}} = O(\frac{n^{\frac{\lambda+1}{2}}}{n^{2\lambda/3}}) \\
&\Leftrightarrow \quad (\frac{p}{n})^{\frac{3-\lambda}{6}} = O(n^{\frac{3-\lambda}{6}}) \\
&\Leftrightarrow \quad \frac{p}{n} = O(n), \quad \text{pois} \quad 2 \leq \lambda < 3 \quad \text{e} \quad \frac{p}{n} \geq 1 \\
&\Leftrightarrow \quad p = O(n^2).
\end{aligned}$$

(b) $\frac{n^{5/3}}{p^{1/3}} \log \frac{n^2}{p} = O(n^\lambda/p^{\frac{\lambda-1}{2}})$.

Demonstração:

$$\begin{aligned}
\frac{n^{5/3}}{p^{1/3}} \log \frac{n^2}{p} = O(\frac{n^\lambda}{p^{\frac{\lambda-1}{2}}}) \quad &\Leftrightarrow \quad \log \frac{n^2}{p} = O(\frac{n^{\frac{3\lambda-5}{3}}}{p^{\frac{3\lambda-5}{6}}}) \\
&\Leftrightarrow \quad \log^6 \frac{n^2}{p} = O((\frac{n^2}{p})^{3\lambda-5}).
\end{aligned}$$

A última condição é sempre verdadeira, pois $2 \leq \lambda < 3$.

(c) Para o caso em que $p = n$, $NM_2 = n^{\frac{\lambda+1}{2}}$ e $NM_3 = n^{4/3}\log n + n^{2\lambda/3}$.

Como $2 \leq \lambda < 3$, $n^{4/3}\log n = o(n^{\frac{\lambda+1}{2}})$ e $n^{2\lambda/3} = o(n^{\frac{\lambda+1}{2}})$. Logo, $NM_3 = o(NM_2)$ quando $p = n$.

Conclusão: A partir dos fatos (a), (b) e (c), pode-se concluir que NM_3 é melhor do que NM_2.

Conclusão final: NM_3 é o melhor dos três algoritmos da classe NM no intervalo $n \leq p \leq n^2$. Como NM_2 tem a mesma complexidade de MM_1 (algoritmo este devido a Dekel, Nassimi e Sahni), pode-se concluir também que neste mesmo intervalo NM_3 é melhor do que MM_1. Portanto, este já é um algoritmo que supera os melhores resultados conhecidos.

4.8.2 MM_1 versus M_1

Considerando $1 \leq p \leq n$, temos:

$$\begin{array}{l} MM_1 = n^\lambda/p^{\frac{\lambda-1}{2}}, \\ M_1 = n^\lambda/p^{\lambda-2}. \end{array}$$

Pode-se comprovar facilmente que $p = \Omega(1)$ garante $MM_1 = O(M_1)$:

$$\begin{aligned} MM_1 = O(M_1) \quad &\Leftrightarrow \quad \frac{n^\lambda}{p^{\frac{\lambda-1}{2}}} = O(\frac{n^\lambda}{p^{\lambda-2}}) \\ &\Leftrightarrow \quad \frac{n^\lambda}{n^\lambda} = O(\frac{p^{\frac{\lambda-1}{2}}}{p^{\lambda-2}}) \\ &\Leftrightarrow \quad 1 = O(p^{\frac{3-\lambda}{2}}) \\ &\Leftrightarrow \quad 1 = O(p) \\ &\Leftrightarrow \quad p = \Omega(1). \end{aligned}$$

Para o caso em que $p = n$, $MM_1 = n^{\frac{\lambda+1}{2}}$ e $M_1 = n^2$. Como $n^{\frac{\lambda+1}{2}} = o(n^2)$ para $2 \leq \lambda < 3$, temos que $MM_1 = o(M_1)$ quando $p = n$.

Conclusão: MM_1 é melhor do que M_1 no intervalo $1 \leq p \leq n$.

4.8.3 MMM_1 versus NM_3

O intervalo considerado é $n \leq p \leq n^2$, onde:

$$\begin{array}{l} MMM_1 = \frac{n^2}{p^{2/3}}\log p + n^\lambda/p^{\lambda/3} \\ NM_3 = \frac{n^{5/3}}{p^{1/3}}\log\frac{n^2}{p} + \frac{n^\lambda}{p^{\lambda/3}}(\frac{p}{n})^{1-\lambda/3} \end{array}$$

Fatos importantes:

(a) $n^\lambda/p^{\lambda/3} = O(\frac{n^\lambda}{p^{\lambda/3}}(\frac{p}{n})^{1-\lambda/3}) \Leftrightarrow p = \Omega(n)$.

Demonstração:

$$\begin{aligned}\frac{n^\lambda}{p^{\lambda/3}} = O(\frac{n^\lambda}{p^{\lambda/3}}(\frac{p}{n})^{1-\lambda/3}) &\Leftrightarrow 1 = O((\frac{p}{n})^{1-\lambda/3}) \\ &\Leftrightarrow 1 = O(\frac{p}{n}) \\ &\Leftrightarrow p = \Omega(n).\end{aligned}$$

(b) $\frac{n^2}{p^{2/3}} \log p = O(\frac{n^{5/3}}{p^{1/3}} \log \frac{n^2}{p}) \Leftrightarrow p = \Omega(n)$ e $p = O(n^2)$.

Demonstração:

$$\begin{aligned}\frac{n^2}{p^{2/3}} \log p = O(\frac{n^{5/3}}{p^{1/3}} \log \frac{n^2}{p}) &\Leftrightarrow \log p = O((\frac{p}{n})^{1/3} \log \frac{n^2}{p}) \\ &\Leftrightarrow \log p + (\frac{p}{n})^{1/3} \log p = O((\frac{p}{n})^{1/3} \log n^2).\end{aligned}$$

A última condição é válida, pois $p = \Omega(n)$ e $p = O(n^2)$.

(c) Quando $p = n^2$, $MMM_1 = o(NM_3)$, pois neste caso $NM_3 = n$ e $MMM_1 = n^{2/3} \log n + n^{\lambda/3}$.

Conclusão: A partir dos fatos (a), (b) e (c), pode-se concluir que, no intervalo $n \leq p \leq n^2$, MMM_1 é melhor do que NM_3. Como conseqüência, neste mesmo intervalo, MMM_1 é melhor do que MM_1.

4.8.4 MMM_1 versus MM_1

O resultado da seção anterior mostra que MMM_1 é melhor do que MM_1 pelo menos no intervalo $n \leq p \leq n^2$. Entretanto, é fácil mostrar que MMM_1 é melhor também no intervalo $1 \leq p \leq n$.

Conforme o que já foi descrito:

$$\begin{aligned}&MMM_1 = \frac{n^2}{p^{2/3}} \log p + n^\lambda/p^{\lambda/3}, \\ &MM_1 = n^\lambda/p^{\frac{\lambda-1}{2}}.\end{aligned}$$

Fatos importantes:

(a) $n^\lambda/p^{\lambda/3} = O(n^\lambda/p^{\frac{\lambda-1}{2}}) \Leftrightarrow p = \Omega(1)$.

Demonstração:

$$\begin{aligned}\frac{n^\lambda}{p^{\lambda/3}} = O(\frac{n^\lambda}{p^{\frac{\lambda-1}{2}}}) &\Leftrightarrow 1 = O(p^{\frac{3-\lambda}{6}}) \\ &\Leftrightarrow p = \Omega(1).\end{aligned}$$

(b) $\frac{n^2}{p^{2/3}}\log p = O(n^\lambda / p^{\frac{\lambda-1}{2}})$.

Demonstração:

$$\frac{n^2}{p^{2/3}}\log p = O(\frac{n^\lambda}{p^{\frac{\lambda-1}{2}}}) \quad \Leftrightarrow \quad \log^6 p = O(\frac{n^{6(\lambda-2)}}{p^{3\lambda-7}})$$

Dois casos devem ser considerados:

- Para $3\lambda - 7 \geq 0$:

$$\log^6 p = O(\frac{n^{6(\lambda-2)}}{p^{3\lambda-7}}) \quad \Leftrightarrow \quad \log^6 p = O((\frac{n}{p})^{3\lambda-7} n^{3\lambda-5}).$$

Neste caso, como $p = O(n)$, $n/p = \Omega(1)$ e $3\lambda - 5 \geq 2$, a condição é verdadeira.

- Para $3\lambda - 7 < 0$:

$$\log^6 p = O(\frac{n^{6(\lambda-2)}}{p^{3\lambda-7}}) \Leftrightarrow \log^6 p = O(n^{6(\lambda-2)} p^{7-3\lambda}).$$

Neste caso, como $0 \leq \lambda - 2 < 1/3$, temos que $n^{6(\lambda-2)} = \Omega(1)$, e portanto a condição é verdadeira.

(c) Quando $p = n$, $MMM_1 = o(MM_1)$, pois neste caso $MMM_1 = n^{4/3}\log n + n^{2\lambda/3}$ e $MM_1 = n^{\frac{\lambda+1}{2}}$.

Conclusão: A partir dos fatos (a), (b) e (c), pode-se concluir que MMM_1 é melhor do que MM_1 também no intervalo $1 \leq p \leq n$. Com o resultado da sub-seção anterior, conclui-se que MMM_1 é melhor do que MM_1 em todo o intervalo $1 \leq p \leq n^2$.

4.8.5 NNM_3 versus NNM_1

Esses algoritmos serão comparados no intervalo $n^2 \leq p \leq n^3$:

$$NNM_3 = \tfrac{n^2}{p^{2/3}}\log n + n^\lambda / p^{\lambda/3},$$
$$NNM_1 = \log \tfrac{p}{n^2} + n^3/p.$$

Fatos importantes:

(a) $n^\lambda / p^{\lambda/3} = O(n^3/p) \Leftrightarrow p = O(n^3)$.

Demonstração:

$$\frac{n^\lambda}{p^{\lambda/3}} = O(\frac{n^3}{p}) \quad \Leftrightarrow \quad \frac{n^\lambda}{(n^2 \frac{p}{n^2})^{\lambda/3}} = O(\frac{n^3}{n^2 \frac{p}{n^2}})$$
$$\Leftrightarrow \quad \frac{n^{\lambda/3}}{(\frac{p}{n^2})^{\lambda/3}} = O(\frac{n}{(\frac{p}{n^2})})$$

$$\Leftrightarrow \quad (\frac{p}{n^2})^{\frac{3-\lambda}{3}} = O(n^{\frac{3-\lambda}{3}})$$
$$\Leftrightarrow \quad (\frac{p}{n^2}) = O(n)$$
$$\Leftrightarrow \quad p = O(n^3).$$

(b) $p = O(n^3) \Rightarrow \log \frac{p}{n^2} = O(\frac{n^2}{p^{2/3}} \log n)$, pois neste caso $p/n^2 = O(n)$ e $n^2/p^{2/3} = \Omega(1)$.

(c) $\frac{n^2}{p^{2/3}} \log n = O(n^3/p) \Leftrightarrow \log n = O(n/p^{1/3}) \Leftrightarrow p = O(n^3/\log^3 n)$.

(d) De modo análogo, $\frac{n^2}{p^{2/3}} \log n = \Theta(n^3/p) \Leftrightarrow p = \Theta(n^3/\log^3 n)$.

(e) De modo análogo, $\frac{n^2}{p^{2/3}} \log n = \Omega(n^3/p) \Leftrightarrow p = \Omega(n^3/\log^3 n)$.

A partir desses fatos, podemos considerar o número de processadores sujeito a quatro casos:

- $n^2 \leq p \leq n^3/\log^3 n$: a partir dos fatos (a) e (c), temos que $NNM_3 = O(NNM_1)$.
- $p = \Theta(n^3/\log^3 n)$: a partir dos fatos (a), (b) e (d), temos que $NNM_3 = \Theta(NNM_1)$.
- $n^3/\log^3 n \leq p \leq n^3$: a partir dos fatos (b) e (e), temos que $NNM_3 = \Omega(NNM_1)$.
- $p = \Theta(n^3)$: $NNM_3 = \Theta(NNM_1)$, pois $NNM_3 = NNM_1 = \log n$.

Portanto, no intervalo $n^2 \leq p \leq n^3$, temos que:

$$\begin{aligned} NNM_3 = O(NNM_1) &\quad\Leftrightarrow\quad p \leq n^3/\log^3 n, \\ NNM_3 = \Theta(NNM_1) &\quad\Leftrightarrow\quad p = \Theta(n^3/\log^3 n) \text{ ou } p = \Theta(n^3), \\ NNM_1 = O(NNM_3) &\quad\Leftrightarrow\quad n^3/\log^3 n \leq p \leq n^3. \end{aligned}$$

Alguns casos particulares:

- Quando $p = n$, $NNM_3 = o(NNM_1)$, pois neste caso $NNM_3 = n^{2/3} \log n + n^{\lambda/3}$ e $NNM_1 = n$.
- Quando $p = n^3/\log n$, $NNM_1 = o(NNM_3)$, pois nesse caso $NNM_1 = \log n$ e $NNM_3 = \log^{5/3} n$.

Conclusão: No intervalo considerado, há uma fronteira ($p = \Theta(n^3/\log^3 n)$) onde à sua esquerda NNM_3 é melhor, e à sua direita, NNM_1 passa a ser o melhor algoritmo, com exceção do caso $p = \Theta(n^3)$, onde eles têm a mesma complexidade. Como nesse mesmo intervalo $\log p = \Theta(\log n)$, a mesma conclusão vale para MMM_1.

4.9 Melhores algoritmos

Os resultados das comparações efetuadas entre os algoritmos podem ser agrupados no seguinte quadro:

$1 \leq p \leq n$	M_1: pior do que MM_1. MM_1: pior do que MMM_1. MMM_1: é o melhor.
$n \leq p \leq n^2$	MM_1: pior do que NM_3. NM_1: pior do que NM_2. NM_2: pior do que NM_3. NM_3: pior do que MMM_1. MMM_1: é o melhor.
$n^2 \leq p \leq n^3$	NNM_1: possui fronteira com NNM_3. NNM_3: possui fronteira com NNM_1. MMM_1: equivalente a NNM_3.

Dessa forma, podemos então concluir quais são os melhores algoritmos em cada intervalo do número de processadores, que varia desde 1 a n^3, e quais as suas complexidades:

PROCESSADORES	ALGORITMOS	COMPLEXIDADES
$1 \leq p \leq n^2$	MMM_1	$O(\frac{n^2}{p^{2/3}} \log p + \frac{n^\lambda}{p^{\lambda/3}})$
$n^2 \leq p \leq n^3/\log^3 n$	MMM_1 ou NNM_3	$O(\frac{n^2}{p^{2/3}} \log n + \frac{n^\lambda}{p^{\lambda/3}})$
$p = \Theta(n^3/\log^3 n)$	MMM_1 ou NNM_3 ou NNM_1	$O(\log^3 n)$
$n^3/\log^3 n \leq p \leq n^3$	NNM_1	$O(\log \frac{p}{n^2} + \frac{n^3}{p})$
$p = \Theta(n^3)$	MMM_1 ou NNM_3 ou NNM_1	$O(\log n)$

4.10 Exemplos

Nesta seção mostraremos alguns exemplos onde comparamos as complexidades dos algoritmos de [DNS81] (MM_1 e NNM_1) com as complexidades dos dois novos algoritmos MMM_1 e NNM_3 nos casos em que esses últimos são melhores.

Assumiremos também um valor fictício para λ (no caso, 8/3) para facilitar a comparação. Obviamente, poderia ter sido utilizado qualquer valor para λ tal que $2 \leq \lambda < 3$.

PROCESSADORES	COMPLEXIDADES	COMPLEXIDADES com $\lambda = 8/3$
$p = \log n$	$MM_1: \quad O(n^{\lambda} / \log^{\frac{\lambda-1}{2}} n)$ $MMM_1: \quad O(\frac{n^2}{\log^{2/3} n} \log\log n + \frac{n^{\lambda}}{\log^{\lambda/3} n})$	$MM_1: \quad O(\frac{n^{8/3}}{\log^{5/6} n})$ $MMM_1: \quad O(\frac{n^{8/3}}{\log^{8/9} n})$
$p = n$	$MM_1: \quad O(n^{\frac{\lambda+1}{2}})$ $MMM_1: \quad O(n^{4/3} \log n + n^{2\lambda/3})$	$MM_1: \quad O(n^{11/6})$ $MMM_1: \quad O(n^{16/9})$
$p = n \log n$	$MM_1: \quad O(n^{\frac{\lambda+1}{2}} / \log^{\frac{\lambda-1}{2}} n)$ $MMM_1: \quad O(n^{4/3} \log^{1/3} n + (\frac{n^2}{\log n})^{\lambda/3})$	$MM_1: \quad O(\frac{n^{11/6}}{\log^{5/6} n})$ $MMM_1: \quad O(\frac{n^{16/9}}{\log^{8/9} n})$
$p = n^{3/2}$	$MM_1: \quad O(n^{\frac{\lambda+3}{4}})$ $MMM_1: \quad O(n \log n + n^{\lambda/2})$	$MM_1: \quad O(n^{17/12})$ $MMM_1: \quad O(n^{4/3})$
$p = n^2$	$NNM_1: \quad O(n)$ $MMM_1: \quad O(n^{2/3} \log n + n^{\lambda/3})$	$NNM_1: \quad O(n)$ $MMM_1: \quad O(n^{8/9})$
$p = n^2 \log n$	$NNM_1: \quad O(n/\log n)$ $NNM_3: \quad O(n^{2/3} \log^{1/3} n + (\frac{n}{\log n})^{\lambda/3})$	$NNM_1: \quad O(\frac{n}{\log n})$ $NNM_3: \quad O((\frac{n}{\log n})^{8/9})$
$p = n^{5/2}$	$NNM_1: \quad O(n^{1/2})$ $MMM_1: \quad O(n^{1/3} \log n + n^{\lambda/6})$	$NNM_1: \quad O(n^{1/2})$ $MMM_1: \quad O(n^{4/9})$

4.11 Algoritmos tipo "divisão-e-conquista"

Toda esta seção é resultado de uma investigação posterior ao desenvolvimento dos algoritmos já descritos. Como foi comentado, a idéia fundamental por trás de tudo o que foi visto em termos de multiplicação de matrizes no hipercubo é adaptar os algoritmos básicos dos casos com n, n^2 e n^3 processadores, considerando sub-matrizes como elementos. Isso é interessante no hipercubo graças às propriedades da visualização matricial.

Partindo desse fato, tentamos obter novos algoritmos básicos, distintos dos já conhecidos, e assim verificar o que aconteceria se repetíssemos neles as adaptações.

Como o hipercubo é uma topologia essencialmente recursiva, surge quase que imediatamente uma pergunta: por que não tentar a elaboração de algoritmos do tipo "divisão-e-conquista"? Essa foi a experiência realizada.

Apresentaremos dois algoritmos básicos originais equivalentes a NN_1 e NNN_1, mas que se baseiam no método de "divisão-e-conquista" aplicados a hipercubos dispostos matricialmente. A partir deles, calcularemos as complexidades de algumas das adaptações já descritas.

Em ambos os algoritmos, as matrizes $\mathcal{A}$, $\mathcal{B}$ e $\mathcal{C}$ serão consideradas como matrizes quadradas de ordem 2, onde seus elementos são sub-matrizes $\frac{n}{2} \times \frac{n}{2}$:

$$\mathcal{A} = \begin{bmatrix} \mathcal{A}_{0,0} & \mathcal{A}_{0,1} \\ \mathcal{A}_{1,0} & \mathcal{A}_{1,1} \end{bmatrix}, \quad \mathcal{B} = \begin{bmatrix} \mathcal{B}_{0,0} & \mathcal{B}_{0,1} \\ \mathcal{B}_{1,0} & \mathcal{B}_{1,1} \end{bmatrix}, \quad \mathcal{C} = \begin{bmatrix} \mathcal{C}_{0,0} & \mathcal{C}_{0,1} \\ \mathcal{C}_{1,0} & \mathcal{C}_{1,1} \end{bmatrix}.$$

Portanto, sabe-se que:

$$\begin{aligned} \mathcal{C}_{0,0} &= \mathcal{A}_{0,0}\mathcal{B}_{0,0} + \mathcal{A}_{0,1}\mathcal{B}_{1,0}, \\ \mathcal{C}_{0,1} &= \mathcal{A}_{0,0}\mathcal{B}_{0,1} + \mathcal{A}_{0,1}\mathcal{B}_{1,1}, \\ \mathcal{C}_{1,0} &= \mathcal{A}_{1,0}\mathcal{B}_{0,0} + \mathcal{A}_{1,1}\mathcal{B}_{1,0}, \\ \mathcal{C}_{1,1} &= \mathcal{A}_{1,0}\mathcal{B}_{0,1} + \mathcal{A}_{1,1}\mathcal{B}_{1,1}. \end{aligned}$$

4.11.1 Algoritmo NN_{DC}: para $p = n^2$ processadores

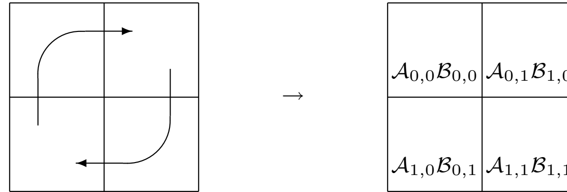
Da mesma forma que os algoritmos básicos anteriores com n^2 processadores, NN_{DC} tem complexidade $O(n)$ e supõe que inicialmente cada processador $p_{i,j}$ armazene os elementos $a_{i,j}$ e $b_{i,j}$, $0 \leq i, j < n$.

Considerando o hipercubo disposto na forma matricial $n \times n$, onde quatro sub-hipercubos armazenam as sub-matrizes conforme a figura abaixo, sabe-se que a troca dos elementos entre quaisquer dois desses sub-hipercubos pode ser feita em tempo constante.

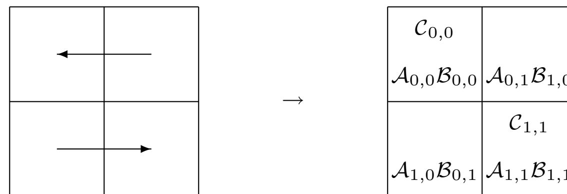
$\mathcal{A}_{0,0}\mathcal{B}_{0,0}$	$\mathcal{A}_{0,1}\mathcal{B}_{0,1}$
$\mathcal{A}_{1,0}\mathcal{B}_{1,0}$	$\mathcal{A}_{1,1}\mathcal{B}_{1,1}$

Seja $T(n)$ o tempo total gasto pelo algoritmo. Seus passos são os seguintes:

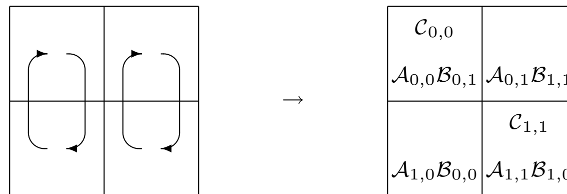
- Troca das sub-matrizes: $\mathcal{B}_{0,1} \leftrightarrow \mathcal{B}_{1,0}$. Tempo: $O(1)$.



- Em cada sub-hipercubo, de maneira recursiva, é calculado o produto entre as duas sub-matrizes armazenadas. Tempo: $T(n/2)$.
- Roteamento e adição dos resultados obtidos no passo anterior, seguindo o esquema abaixo, calculando assim $\mathcal{C}_{0,0}$ e $\mathcal{C}_{1,1}$. Tempo: $O(1)$.

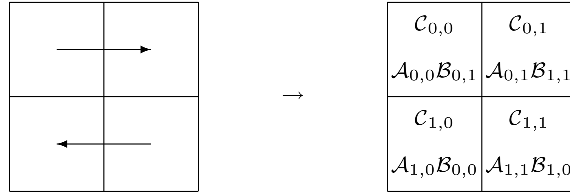


- Troca das sub-matrizes: $\mathcal{B}_{0,0} \leftrightarrow \mathcal{B}_{0,1}$ e $\mathcal{B}_{1,1} \leftrightarrow \mathcal{B}_{1,0}$. Tempo: $O(1)$.



- Cálculo recursivo dos produtos referentes às sub-matrizes de $\mathcal{A}$ e $\mathcal{B}$ armazenadas em cada sub-hipercubo. Tempo: $T(n/2)$.

- Roteamento e adição dos resultados obtidos no passo anterior, seguindo o esquema abaixo, calculando assim $\mathcal{C}_{0,1}$ e $\mathcal{C}_{1,0}$. Tempo: $O(1)$.



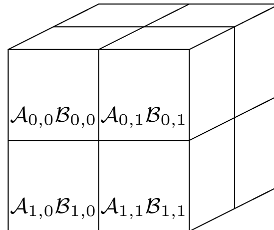
Caso deseje-se que as sub-matrizes de $\mathcal{A}$ e $\mathcal{B}$ voltem às suas posições iniciais, isso pode ser feito em tempo $O(1)$ através de novos roteamentos.

O tempo total é $T(n) = 2T(n/2) + O(1)$. Como $T(1) = O(1)$, segue que $T(n) = O(n)$. Este algoritmo não exige nenhuma capacidade extra de armazenamento em cada processador, pois somente houve roteamentos e trocas de valores. Portanto, cada processador necessita apenas de espaço $O(1)$.

4.11.2 Algoritmo NNN_{DC}: com $p = n^3$ processadores

Considerando o hipercubo como um cubo com n^3 processadores, a disposição inicial das matrizes $\mathcal{A}$ e $\mathcal{B}$ é a mesma exigida pelo algoritmo NNN_1 descrito anteriormente, ou seja, ambas as matrizes estão na face $k = 0$, um elemento de cada matriz por processador.

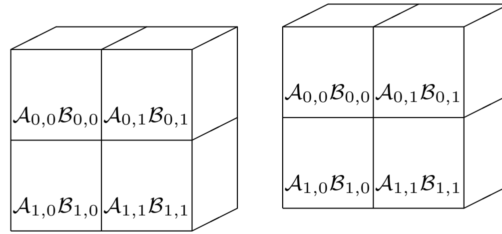
O esquema abaixo mostra a disposição inicial das matrizes $\mathcal{A}$ e $\mathcal{B}$ no hipercubo, onde cada sub-cubo corresponde a um sub-hipercubo com $n^3/8$ processadores:



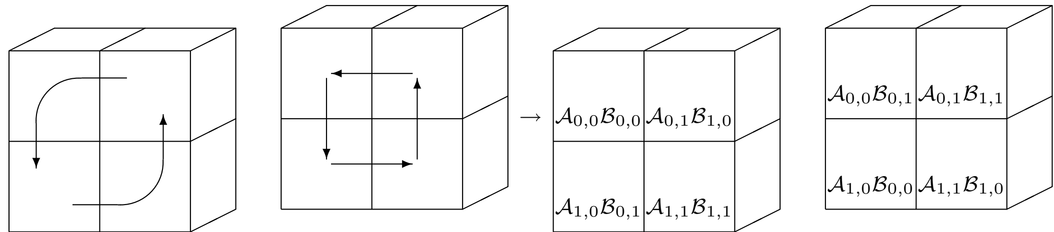
Chamemos novamente de $T(n)$ o tempo gasto por este algoritmo. Seus passos são os seguintes:

- Cópia das sub-matrizes na correspondente face $k = 0$ dos sub-cubos de trás. Tempo: $O(1)$.

  Separando-se os quatro sub-cubos da frente dos outros quatro de trás, pode-se observar melhor a situação após esse passo:



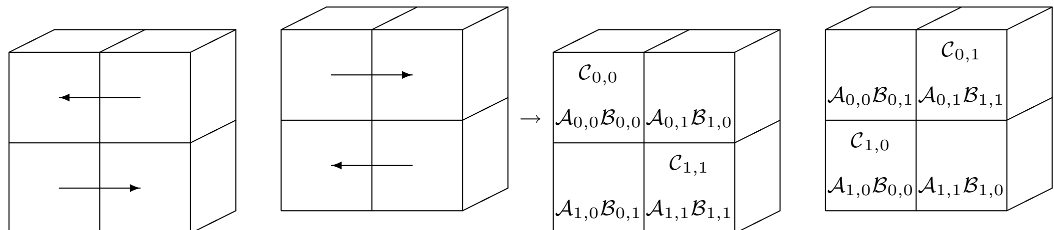
- Troca das sub-matrizes de $\mathcal{B}$ segundo o esquema abaixo:



  O tempo gasto nestes roteamentos é $O(1)$.

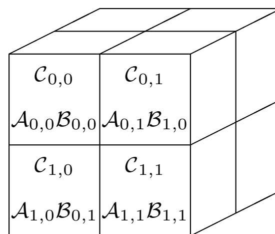
- Cálculo recursivo dos produtos das sub-matrizes em cada um dos 8 sub-hipercubos. Tempo necessário: $T(n/2)$.

- Roteamento dos resultados obtidos no passo anterior com somatória para cálculo de $\mathcal{C}_{0,0}$, $\mathcal{C}_{0,1}$, $\mathcal{C}_{1,0}$ e $\mathcal{C}_{1,1}$, conforme a figura abaixo:



  O tempo gasto neste passo é $O(1)$.

- Cópia de $\mathcal{C}_{1,0}$ e $\mathcal{C}_{0,1}$ nos correspondentes sub-hipercubos da frente:



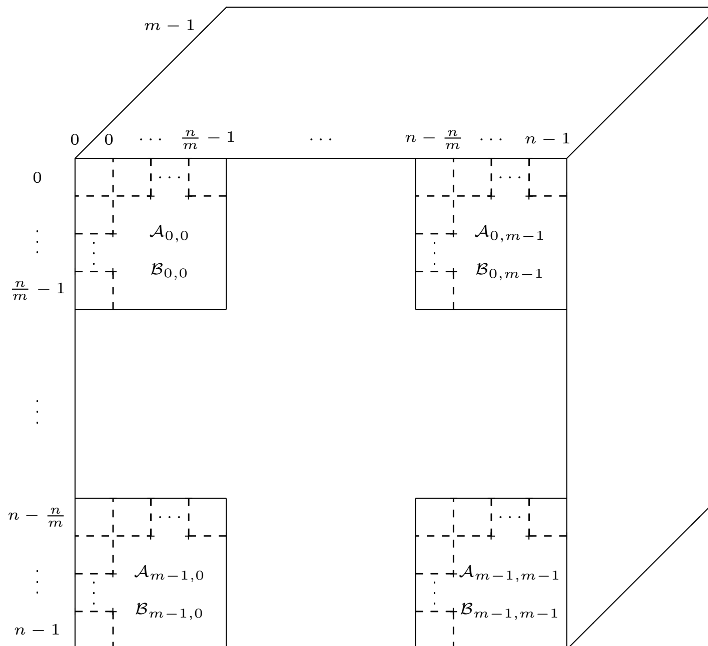
Este último passo também gasta tempo $O(1)$.

Caso seja necessário, as sub-matrizes de $\mathcal{B}$ podem ser recolocadas em suas posições iniciais em tempo $O(1)$ fazendo-se roteamentos inversos aos descritos no segundo passo.

O tempo total do algoritmo é $T(n) = T(n/2) + O(1)$, ou seja, $T(n) = O(\log n)$, uma vez que $T(1) = O(1)$. Do mesmo modo que o algoritmo anterior, o espaço necessário em cada processador é constante.

4.11.3 Algoritmo NNM_{DC}: com $n^2 \leq p \leq n^3$ processadores

De modo análogo ao desenvolvimento de NNM_1, a idéia é considerar cada grupo de $(n/m)^2$ processadores como uma sub-matriz, e adaptar NNN_{DC}. Supõe também que as matrizes estejam inicialmente armazenadas como em NNM_1: um elemento por processador na face $k = 0$ da visualização matricial.



Não descreveremos os passos por ser desnecessário, uma vez que são os mesmos de NNN_{DC}, considerando as sub-matrizes de ordem n/m como elementos e utilizando NN_{DC} como sub-algoritmo.

Chamando de $T(n, m)$ o tempo necessário para o cálculo da matriz $\mathcal{C}$ utilizando um hipercubo com $p = n^2 m$ processadores através do algoritmo NNM_{DC}, temos que:

$$\begin{cases} T(n,1) &= O(n) \\ T(n,m) &= T(n/2, m/2) + O(1) \end{cases}$$

Desenvolvendo a fórmula de recorrência:

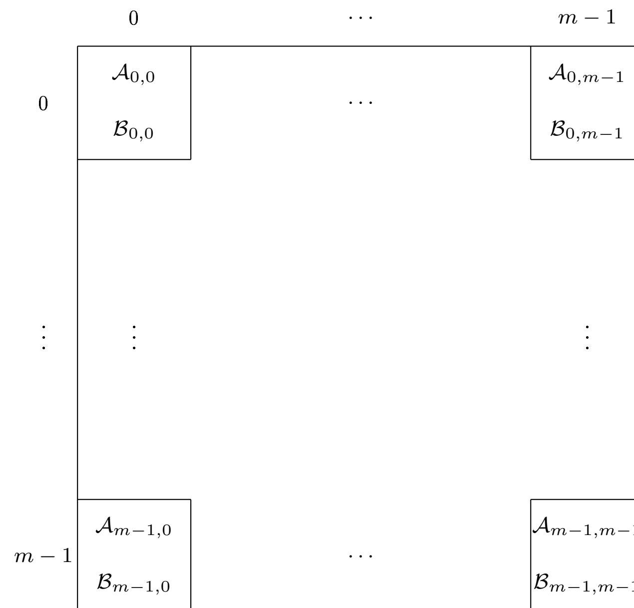
$$\begin{aligned} T(n,m) &= T(n/2, m/2) + O(1) \\ &= T(n/2^2, m/2^2) + O(1) + O(1) \\ &\vdots \\ &= T(n/2^k, m/2^k) + O(k) \end{aligned}$$

Para $m = 2^k$:

$$\begin{aligned} T(n,m) &= T(n/m, 1) + O(\log m) \\ &= O(n/m) + O(\log m) \\ &= O(n^3/p + \log \frac{p}{n^2}) \end{aligned}$$

4.11.4 Algoritmo MM_{DC}: com $1 \leq p \leq n^2$ processadores

Esse algoritmo é desenvolvido de maneira análoga a MM_1: cada processador armazena $2(n/m)^2$ elementos, segundo o esquema abaixo, mas ele segue os passos de NN_{DC}, utilizando também o melhor algoritmo seqüencial para multiplicação de matrizes. Sabe-se portanto que os roteamentos e adições entre sub-matrizes custam $O((n/m)^2)$.



Novamente, chamando agora de $T(m)$ o tempo necessário para o cálculo da matriz $\mathcal{C}$ através de MM_{DC}, temos que:

$$\begin{cases} T(1) & = & O((n/m)^{\lambda}) \\ T(m) & = & 2T(m/2) + O((n/m)^2) \end{cases}$$

Desenvolvendo a fórmula de recorrência:

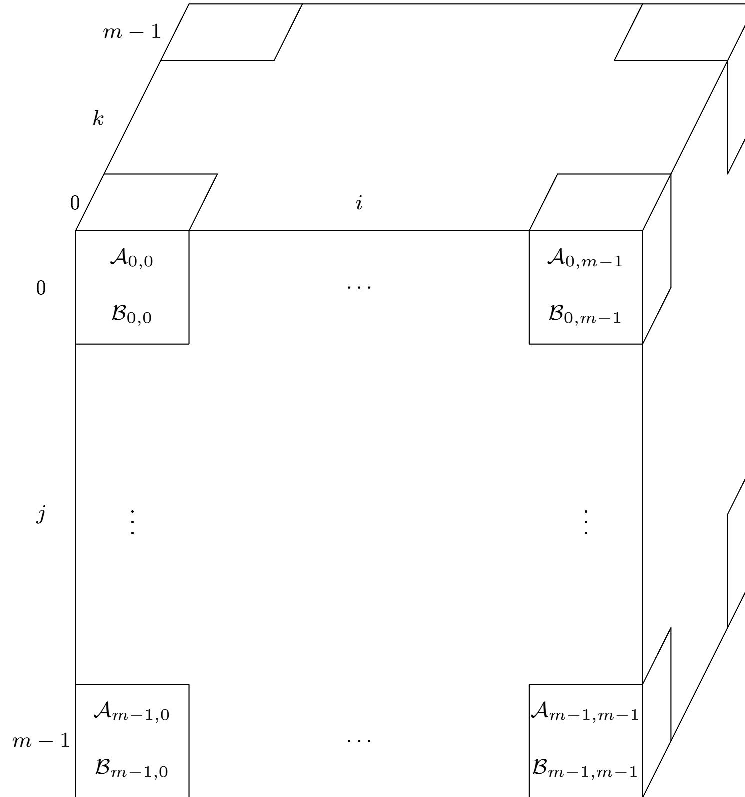
$$\begin{aligned}
T(m) &= 2T(m/2) + O(n^2/m^2) \\
&= 2[2T(m/2^2) + O(n^2/(m/2)^2)] + O(n^2/m^2) \\
&= 2^2T(m/2^2) + O(\frac{n^2}{m^2}(2^2+1)) \\
&= 2^2[2T(m/2^3) + O(n^2/(m/2^2)^2)] + O(\frac{n^2}{m^2}(2^2+1)) \\
&= 2^3T(m/2^3) + O(\frac{n^2}{m^2}(2^4+2^2+1)) \\
&\vdots \\
&= 2^kT(m/2^k) + O(\frac{n^2}{m^2}(2^{2(k-1)} + 2^{2(k-2)} + \ldots + 1)) \\
&= 2^kT(m/2^k) + O(\frac{n^2}{m^2}\frac{2^{2k}-1}{3})
\end{aligned}$$

Para $m = 2^k$:

$$\begin{aligned}
T(m) &= O(m(n/m)^{\lambda}) + O(\frac{n^2}{m^2}\frac{m^2-1}{3}) \\
&= O(n^{\lambda}/m^{\lambda-1} + n^2) \\
&= O(n^{\lambda}/p^{\frac{\lambda-1}{2}} + n^2)
\end{aligned}$$

4.11.5 Algoritmo MMM_{DC}: com $1 \leq p \leq n^3$ processadores

O desenvolvimento de MMM_{DC} é análogo ao de MMM_1, pois cada processador da face $k = 0$ da visualização abaixo armazena sub-matrizes de $\mathcal{A}$ e $\mathcal{B}$ de ordem n/m. Seus passos seguem NNN_{DC}, e também utilizam o melhor algoritmo seqüencial para multiplicação de matrizes.



Agora $T(m)$ será o tempo necessário para o cálculo da matriz $\mathcal{C}$ através de MMM_{DC}:

$$\begin{cases} T(1) & = & O((n/m)^{\lambda}) \\ T(m) & = & T(m/2) + O((n/m)^2) \end{cases}$$

Desenvolvendo a fórmula de recorrência:

$$\begin{aligned}
T(m) &= T(m/2) + O(n^2/m^2) \\
&= T(m/2^2) + O(n^2/(m/2)^2) + O(n^2/m^2) \\
&= T(m/2^2) + O(\frac{n^2}{m^2}(2^2+1)) \\
&= T(m/2^3) + O(n^2/(m/2^2)^2) + O(\frac{n^2}{m^2}(2^2+1)) \\
&= T(m/2^3) + O(\frac{n^2}{m^2}(2^4+2^2+1)) \\
&\vdots \\
&= T(m/2^k) + O(\frac{n^2}{m^2}(2^{2(k-1)} + 2^{2(k-2)} + \ldots + 1)) \\
&= T(m/2^k) + O(\frac{n^2}{m^2}\frac{2^{2k}-1}{3})
\end{aligned}$$

Para $m = 2^k$:

$$\begin{aligned} T(m) &= O((n/m)^\lambda) + O(\frac{n^2}{m^2}\frac{m^2-1}{3}) \\ &= O((n/m)^\lambda + n^2) \\ &= O(n^\lambda/p^{\lambda/3} + n^2) \end{aligned}$$

4.11.6 Comparações

Na tabela abaixo estão as complexidades desses novos algoritmos do tipo "divisão-e-conquista" diante de algum de seus similares anteriores já descritos:

PROCESSADORES	"DIVISÃO-E-CONQUISTA"	ANTERIORES
$p = n^2$	NN_{DC}: $O(n)$	NN_1: $O(n)$
$p = n^3$	NNN_{DC}: $O(\log n)$	NNN_1: $O(\log n)$
$n^2 \leq p \leq n^3$	NNM_{DC}: $O(\log \frac{p}{n^2} + \frac{n^3}{p})$	NNM_1: $O(\log \frac{p}{n^2} + \frac{n^3}{p})$
$1 \leq p \leq n^2$	MM_{DC}: $O(n^\lambda/p^{\frac{\lambda-1}{2}} + n^2)$	MM_1: $O(n^\lambda/p^{\frac{\lambda-1}{2}})$
$1 \leq p \leq n^3$	MMM_{DC}: $O(n^2 + \frac{n^\lambda}{p^{\lambda/3}})$	MMM_1: $O(\frac{n^2}{p^{2/3}} \log p + \frac{n^\lambda}{p^{\lambda/3}})$

Podemos concluir que os algoritmos do tipo "divisão-e-conquista", apesar de serem todos originais, não são melhores que os anteriores em termos de complexidade de tempo. Isso se deve ao fato de se baseiam principalmente em roteamentos (não há neles qualquer necessidade de armazenamento extra), e portanto têm uma eficiência ruim quando os processadores inicialmente contêm sub-matrizes. Por outro lado, nos casos em que há um único elemento de cada matriz por processador, obtêm-se complexidades equivalentes, com a vantagem de gastar espaço $O(1)$.

Capítulo 5

Multiplicação de matriz por vetor

Apresentamos aqui um outro exemplo original da utilidade da visualização matricial na elaboração de algoritmos para o hipercubo. Como há uma semelhança muito grande entre o problema da multiplicação de matrizes e o da multiplicação de matriz por vetor, é possível aproveitar algumas das idéias presentes no capítulo anterior. Utilizaremos inclusive a mesma nomenclatura, sem a necessidade de classes, para facilitar o entendimento.

Por não ser o tema mais importante deste trabalho, todas as descrições serão apenas informais.

5.1 O problema

A multiplicação de matriz por vetor consiste em calcular um vetor $\mathcal{Y}=(y_i)$, $0 \leq i < n$, que é o resultado da multiplicação entre uma matriz $\mathcal{A}=(a_{i,j})$ e outro vetor $\mathcal{X}=(x_i)$, $0 \leq i, j < n$. Dessa forma, $\mathcal{A}\mathcal{X}=\mathcal{Y}$, onde $y_i = \sum_{k=0}^{n-1} a_{i,k}x_k$, $0 \leq i < n$. Também se supõe que tanto a matriz $\mathcal{A}$ como o vetor $\mathcal{X}$ estejam dispostos no hipercubo de uma maneira conveniente, e a disposição final do vetor $\mathcal{Y}$ será análoga à disposição inicial de $\mathcal{X}$. Além disso, cada elemento da matriz ou dos dois vetores tem tamanho unitário, permitindo assim sua transmissão em tempo $O(1)$.

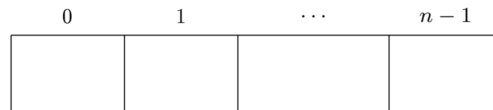
Pela própria definição do problema, percebe-se que sua complexidade seqüencial é $O(n^2)$.

5.2 Algoritmos paralelos conhecidos

Em [BeT89], há dois algoritmos para os casos em que o número p de processadores do hipercubo é n ou n^2. Suas complexidades são $O(n)$ e $O(\log n)$, respectivamente, e os chamaremos de N e NN, seguindo a mesma notação empregada no capítulo 4. Vamos descrevê-los rapidamente.

5.2.1 Algoritmo N: com $p = n$ processadores

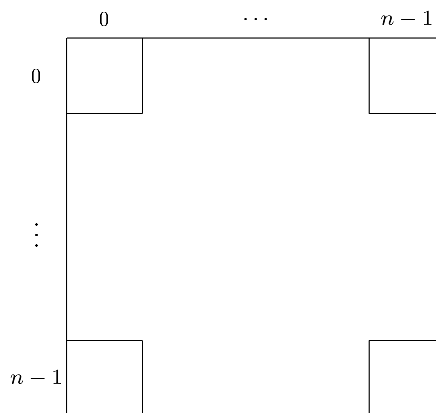
Esse algoritmo considera o hipercubo disposto linearmente, conforme a figura abaixo.



Cada p_i armazena inicialmente o vetor $\mathcal{X}$ e i-ésima linha de $\mathcal{A}$, e pode calcular em tempo $O(n)$ o elemento y_i do vetor $\mathcal{Y}$. Através de uma DM, que custa também $O(n)$, cada processador terá todos os elementos do vetor $\mathcal{Y}$.

5.2.2 Algoritmo NN: com $p = n^2$ processadores

Como era de se esperar, NN considera o hipercubo na visualização matricial $n \times n$.



Inicialmente, cada $p_{i,j}$ armazena o elemento $a_{i,j}$ de $\mathcal{A}$. O vetor $\mathcal{X}$ está armazenado numa linha da matriz, um elemento por processador. Sem perda de generalidade, vamos considerar que essa linha seja a de número 0. Dessa forma, no início $p_{0,j}$ contém o valor de x_j, além de $a_{0,j}$, para $0 \leq j < n$.

Os passos são os seguintes:

- A partir de $p_{0,j}$, DS de x_j em cada coluna. No final desse passo, que custa $O(\log n)$, o vetor $\mathcal{X}$ estará em todas as linhas da matriz, um elemento por processador.
- Cada processador $p_{i,j}$ calcula $a_{i,j}x_j$, que é uma das n parcelas de y_j. Custo: $O(1)$.
- Através de AS em cada linha j, pode-se somar todas as parcelas de y_j, armazenando o resultado em $p_{j,j}$. Custo: $O(\log n)$.
- DS em cada coluna j a partir de $p_{j,j}$: $O(\log n)$.

Portanto, o tempo total deste algoritmo é $O(\log n)$.

5.3 Os novos algoritmos

Lembrando das idéias descritas no último capítulo, fica simples, a partir dos dois algoritmos anteriores, elaborar outros mais gerais e originais: M e MM[1].

A matriz $\mathcal{A}$ é vista como formada por sub-matrizes quadradas de ordem n/m, onde m e n são potências de 2 tais que $1 \leq m \leq n$, e $\mathcal{X}$ e $\mathcal{Y}$ são vetores compostos de m sub-vetores com n/m elementos cada um:

$$\mathcal{A} = (\mathcal{A}_{i,j}),\ \mathcal{X} = (\mathcal{X}_i) \text{ e } \mathcal{Y} = (\mathcal{Y}_i),$$

onde $0 \leq i, j < m$. Sabe-se portanto que $\mathcal{Y}_i = \sum_{k=0}^{m-1} \mathcal{A}_{i,k}\mathcal{X}_k$.

5.3.1 Algoritmo M: com $1 \leq p \leq n$ processadores

O hipercubo com $p = m$ processadores é visualizado linearmente. Inicialmente, cada processador p_i armazena a i-ésima linha de sub-matrizes de $\mathcal{A}$, isto é, $\mathcal{A}_{i,j}$ para $0 \leq j < m$, e o vetor $\mathcal{X}$ completo. O algoritmo baseia-se em N.

Passos:

- Em cada p_i, calcula-se $\mathcal{Y}_i$ gastando tempo $O(n^2/m)$.
- DM de $\mathcal{Y}_i$: $O(\frac{n}{m}m) = O(n)$.

Tempo total: $O(n + n^2/m) = O(n^2/m) = O(n^2/p)$.

Alguns casos interessantes:

p	COMPLEXIDADE
$\log n$	$O(n^2/\log n)$
$n^{1/2}$	$O(n^{3/2})$
n	$O(n)$

5.3.2 Algoritmo MM: com $1 \leq p \leq n^2$ prodessadores

Neste caso, o hipercubo de $p = m^2$ processadores é visualizado como uma matriz $m \times m$, e cada processador $p_{i,j}$ inicialmente armazena a sub-matriz $\mathcal{A}_{i,j}$. A linha 0 contém o vetor $\mathcal{X}$, um sub-vetor por processadoo.

Os passos são análogos aos do algoritmo NN:

- DS dos sub-vetores de $\mathcal{X}$ nas colunas: $O(\frac{n}{m}\log m)$.
- Cada processador $p_{i,j}$ calcula $\mathcal{A}_{i,j}\mathcal{X}_j$, que é um sub-vetor de tamanho n/m. Tempo: $O((n/m)^2)$.
- AS em cada linha j, colocando em $p_{j,j}$ a somatória dos sub-vetores calculados no passo anterior, que corresponde ao sub-vetor $\mathcal{Y}_j$: $O(\frac{n}{m}\log m)$.

[1]Também é possível criar mais dois algoritmos, utilizando para isso a visualização NM, mas como eles têm complexidades $O(\frac{n^2}{p}\log n)$, para $n \leq p \leq n^2$, não os descreveremos.

- DS do sub-vetor $\mathcal{Y}_j$ em cada coluna j, a partir de $p_{j,j}$: $O(\frac{n}{m}\log m)$.

Tempo total: $O(\frac{n}{m}\log m + (n/m)^1) = O(\frac{n}{p^{1/2}}\log p + n^2/p)$.
Alguns casos interessantes:

p	COMPLEXIDADE
$\log n$	$O(n^2/\log n)$
$n^{1/2}$	$O(n^{3/2})$
n	$O(n)$
$n\log n$	$O(n/\log n)$
$n^{3/2}$	$O(n^{1/2})$
$n^2/\log n$	$O(\log^{3/2} n)$
n^2	$O(\log n)$

5.4 Resultados

Conforme as descrições acima, temos os seguintes algoritmos para multiplicação de matriz por vetor, uma vez que N e NN são casos particulares:

ALGORITMO	PROCESSADORES	COMPLEXIDADE
M	$1 \leq p \leq n$	$O(n^2/p)$
MM	$1 \leq p \leq n^2$	$O(\frac{n}{p^{1/2}}\log p + n^2/p)$

5.5 Comparação

Seguindo a mesma notação e critérios do capítulo 4, somente podemos comparar ambos os algoritmos no intervalo $1 \leq p \leq n$:

$$M = n^2/p,$$
$$MM = \frac{n}{p^{1/2}}\log p + n^2/p.$$

Sabe-se que:

$$\frac{n}{p^{1/2}}\log p = O(n^2/p) \Leftrightarrow p^{1/2}\log p = O(n)$$

No intervalo considerado, $p^{1/2} = O(n^{1/2})$, e $\log p = O(\log n) = o(n^{1/2})$. Portanto, a última condição acima é verdadeira, o que significa que a complexidade de MM no intervalo $1 \leq p \leq n$ é $O(n^2/p)$: a mesma de M.

5.6 Melhores algoritmos

A tabela abaixo mostra a complexidade dos melhores algoritmos de multiplicação de matriz por vetor no intervalo $1 \leq p \leq n^2$:

PROCESSADORES	ALGORITMOS	COMPLEXIDADE
$1 \leq p \leq n$	M ou MM	$O(n^2/p)$
$1 \leq p \leq n^2$	MM	$O(\frac{n}{p^{1/2}} \log p + n^2/p)$

Conseguimos assim generalizar os algoritmos N e NN (válidos para n e n^2 processadores, respectivamente) para o número de processadores no intervalo citado acima.

Capítulo 6

Comentários e perspectivas

A exemplo dos trabalhos de [DNS81, NaS81, NaS82], em que não são comentados resultados de qualquer implementação, os resultados deste trabalho são de cunho teórico. Podemos dizer que os algoritmos "não saíram do papel". A superioridade de alguns dos algoritmos apresentados demonstra-se pelas suas complexidades de tempo.

A utilização das operações básicas de comunicação foi motivada graças às suas descrições em [BeT89], empregadas inclusive em algoritmos básicos de multiplicação de matrizes. É claro que seu uso não é necessário, mas, uma vez que se tornam familiares ao leitor, permitem maior simplicidade e clareza tanto na descrição como na formalização de algoritmos. No nosso caso, a única desvantagem que observamos foi a de exigir em alguns algoritmos um maior espaço em cada processador.

Um fato que julgamos necessário ressaltar é a importância das visualizações matriciais do hipercubo como instrumento de concepção de algoritmos. Seus autores, sem lhe darem esse nome, a utilizaram em muitos algoritmos, mas não chegaram a enumerar algumas de suas propriedades da forma como fizemos no capítulo 2. Consideramos que esse conhecimento facilita muito a compreensão dos algoritmos. Por esse mesmo motivo, os exemplos citados (intercalação bitônica, permutação, ordenação e transposição de matrizes) foram descritos baseados nestas propriedades, utilizando também as operações básicas, e permitem uma rápida assimilação das idéias que os originaram, o que não acontece de modo claro nos artigos. Os algoritmos originais para multiplicação de matriz por vetor apresentados no capítulo 5 são mais um exemplo que destaca a importância deste instrumento.

Quanto ao problema da multiplicação de matrizes no hipercubo, como já comentamos, a seção 4.5 mostra qual foi o ponto de partida em que nos apoiamos na elaboração dos novos algoritmos. Todo o mérito da aplicação da visualização matricial a esse problema, aproveitando algumas de suas características próprias, é dos autores já citados. Simplesmente, eles não a exploraram completamente. Não julgamos que o nosso trabalho o faça, longe disso, mas contribui de uma forma satisfatória mostrando grandes famílias de algoritmos e deixando possibilidades para a concepção de vários outros. Essa grande quantidade de algoritmos nos obrigou a criar uma divisão entre eles, agrupando-os em classes, pois descrevê-los um a um sem seguir uma ordem seria muito enfadonho.

A importância da utilização da visualização matricial do hipercubo para elaborar novos

algoritmos pode ser ainda corroborada pelo seguinte fato, ao menos curioso. Ao tomarmos conhecimento da visualização matricial através de um trabalho de Nassimi e Sahni [NaS82], desconhecíamos os trabalhos de [DNS81]. Os primeiros algoritmos para multiplicação de matrizes que descobrimos perteciam justamente à classe NNM. Somente posteriormente, após conhecermos o trabalho de [DNS81], descobrimos que na verdade teríamos reinventado resultados já conhecidos.

As formalizações são todas originais. Convidamos o leitor a compará-las com as que estão apresentadas em [DNS81], para que note o quanto se lucrou em clareza e concisão através do emprego das operações básicas. Aproveitamos a vantagem de se dividir um hipercubo em outros através de um sub-conjunto dos *bits* dos endereços de seus vértices para utilizar as formalizações como sub-algoritmos.

A primeira dificuldade enfrentada foi a determinação do modelo de medida do tempo de comunicação entre os processadores. No capítulo 3, relatamos quais foram as hipóteses consideradas, com o objetivo de tornar os novos resultados comparáveis aos já conhecidos. Em seguida, surgiu o grande obstáculo: comparar as complexidades.

Uma vez que as complexidades obtidas dependiam do número p de processadores, que por sua vez estava sujeito a um intervalo relacionado com a ordem das matrizes, fomos obrigados a sugerir um critério de comparação. Não encontramos na literatura nada que pudesse nos ajudar, o que não significa que não exista. O fato de que há uma omissão por parte dos autores citados quanto a certos detalhes, forçou-nos a fixar algumas regras. O critério que estabelecemos, também descrito no capítulo 4, foi o que nos pareceu mais adequado, mas o consideramos sujeito a vários aperfeiçoamentos. De qualquer forma, os exemplos que apresentamos após as comparações nos dão uma certa garantia da corretude das conclusões.

Por fim, apresentamos algo extra: a aplicação do paradigma da "divisão-e-conquista" ao problema da multiplicação de matrizes no hipercubo, considerando o que já havia sido exposto até então. Não descrevemos todos os algoritmos; apenas indicamos o modo de como é possível obtê-los. Conforme já frisamos, parecem não representar progressos quanto à complexidade de tempo, inclusive sendo piores em alguns casos, mas possuem a vantagem de não necessitar de espaço extra de armazenamento.

Há dois aspectos que ficaram pendentes e que podem ser objetos de pesquisa:

- Não foi estudada a generalização do problema da multiplicação de matrizes para o caso em que elas não sejam quadradas, nem quando sua ordem não seja potência de 2. Esse último aspecto parece ser de simples solução, utilizando para isso o menor hipercubo capaz de conter as matrizes.

- Muitos outros problemas podem ser resolvidos através da multiplicação de matrizes. Dessa forma, os resultados aqui apresentados podem ter outras aplicações.

Apêndice A

Lista de símbolos

SÍMBOLO	SIGNIFICADO	PRIMEIRA OCORRÊNCIA
L	Comprimento da mensagem a ser enviada	4
β	Custo de *start-up* para comunicação	4
τ	Tempo de transmissão	4
DS	Difusão Singular	5
DM	Difusão Multi-nó	5
Dp	Dispersão	5
TC	Troca Completa	5
AS	Acumulação Singular	5
AM	Acumulação Multi-nó	5
Rc	Recolhimento	5
$H(d)$	Hipercubo de dimensão d	7
p	Número de processadores de $H(d)$: $p = 2^d$	7
$e_{d-1} \dots e_1 e_0$	Representação binária do endereço do vértice e de $H(d)$	7
$\oplus$	Operador ou-exclusivo	7
$R(i)$	Registrador R no vértice i de $H(d)$	10
$R[j](i)$	j-ésima posição do vetor R do vértice i de $H(d)$	10
i_b	b-ésimo *bit* da representação binária de i	10
$i_{b_2:b_1}$	*Bits* $b_2 \dots b_1$ da representação binária de i	10
$i^{(b)}$	Corresponde a $i_{d-1} \dots i_{b+1} \bar{i_b} i_{b-1} \dots i_0$, onde $\bar{i_b}$ é o complemento de i_b	10
$:=$	Atribuição entre registradores de um mesmo vértice	11
$\leftarrow$	Atribuição entre registradores de vértices vizinhos	11
$\leftrightarrow$	Troca entre registradores de vértices vizinhos	11
$P = (p_{i,j,\dots})$	Matriz de processadores que representa visualização matricial de $H(d)$	16
&	Operador de concatenação	31
n	Ordem das matrizes (potência de 2)	34
$\mathcal{A}= (a_{i,j}), \mathcal{B}= (b_{i,j}), \mathcal{C}= (c_{i,j})$	Matrizes quadradas de ordem n	34
λ	Expoente de n na complexidade do melhor algoritmo seqüencial para calcular $\mathcal{AB}$	34
BITS	Vetor que contém índices dos *bits* que definem os sub-hipercubos	34
m	Potência de 2 pertencente ao intervalo $[1, n]$	40
$\mathcal{X}= (x_i)$, $\mathcal{Y}= (y_i)$	Vetores de tamanho n	69

Referências Bibliográficas

[Bat68] Batcher, K.E., "Sorting Networks and their Applications", *Proc. AFIS 1968 SJCC*, AFIPS Press, Montvale, N.J., Vol. 32, p. 307-314, 1968.

[Bel92] Bell, G., "Ultracomputers: a Teraflop Before its Time", *Communications of ACM*, Vol. 35, N. 8, p. 27-47, August 1992.

[BOS91] Bertsekas, D.P.; Özveren, C.; Stamoulis, G.D.; Tseng, P. and Tsitsiklis, J.N., "Optimal Communication Algorithms for Hypercubes", *Journal of Parallel and Distributed Computing*, Vol. 11, p. 263-275, 1991.

[BeT89] Bertsekas, D.P. and Tsitsiklis, J.N., *Parallel and Distributed Computation - Numerical Methods*, Prentice-Hall, 1989.

[Can69] Cannon, L.E., *A Cellular Computer to Implement the Kalman Filter Algorithm*, Ph.D. thesis, Montana State University, 1969.

[CoW87] Coppersmith, D. and Winograd, S., "Matrix Multiplication via Arithmetic Progression", *Proc. 19th Annual ACM Symp. on the Theory of Computing*, p. 1-6, 1987.

[DNS81] Dekel, E.; Nassimi, D. and Sahni, S., "Parallel Matrix and Graph Algorithms", *SIAM J. Comp.*, Vol. 10, N. 10, p. 657-675, November 1981.

[FrL91] Fraigniaud, P. and Lazard, E., *Methods and Problems of Communication in Usual Networks*, Technical Report N. 91-33, Laboratoire de l'Informatique du Parallélslime, École Normale Supérieure de Lyon, October 1991.

[Hil85] Hillis, W.D., *The Connection Machine*, MIT Press, Cambridge, Mass., 1985.

[JaT85] Ja'Ja', J. and Takche, J., "Improved Lower Bounds for some Matrix Multiplication Problems", *Information Processing Letters*, Vol. 21, N. 3, p. 123-127, September 1985.

[Joh87] Johnsson, S.L., "Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures", *Journal of Parallel and Distributed Computing*, Vol. 4, p. 133-172, 1987.

[Knu73] Knuth, D.E., *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*, Addison-Wesley, 1973.

[McV87] McBryan, O.A. and Van de Velde, E.F., "Hypercube Algorithms and Implementations", *SIAM J. Sci. Stat. Comput.*, Vol. 8, N. 2, p. s227-287, March 1987.

[MoS88] Monien, B. and Sudborough, H., *Comparing Interconnection Networks*, Technical Report, October 1988.

[NaS81] Nassimi, D. and Sahni, S., "Data Broadcasting in SIMD Computers", *IEEE Trans. Comput.*, Vol. C-30, N. 2, p. 101-107, February 1981.

[NaS82] Nassimi, D. and Sahni, S., "Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network", *Journal of the ACM*, p. 642-667, July 1982.

[Pan80] Pan, V. Ya., "New Fast Algorithms for Matrix Operations", *SIAM J. Comp.*, Vol. 9, N. 2, p. 321-342, May 1980

[Pre78] Preparata, F.P., "New Parallel-Sorting Schemes", *IEEE Trans. Comput.*, Vol. C-27, N. 7, p. 669-673, July 1978.

[SaS88] Saad, Y. and Schultz, M.H., "Topological Properties of Hypercubes", *IEEE Trans. Comput.*, Vol. 37, N. 7, p. 867-872, July 1988.

[SaS89] Saad, Y. and Schultz, M.H., "Data Communication in Hypercubes", *Journal of Parallel and Distributed Computing*, Vol. 6, p. 115-135, 1989.

[Sch81] Schönhage, A., "Partial and Total Matrix Multiplication", *SIAM J. Comp.*, Vol. 10, N. 3, p. 434-455, August 1981.

[Sei85] Seitz, C.L., "The Cosmic Cube", *Communications of the ACM*, 28, p. 22-33, 1985.

[Sie79] Siegel, H.J., "Interconnection Networks for SIMD machines", *Computer*, p. 57-65, June 1979.

[Son91] Song, S.W., *Síntese de Algoritmos Paralelos para o n-cubo Binário*, tese de livre-docência, IME-USP, Departamento de Ciência da Computação, junho de 1991.

[Str69] Strassen, V., "Gaussian Elimination is not Optimal", *Numer. Math.*, 13, p. 354-356, 1969.

[Wil86] Wilf, H.S., *Algorithms and Complexity*, Prentice-Hall, 1986.