Introdução a Algoritmos Paralelos E Suas Complexidades

Marcelo H. Carvalho

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. Routo Terada

-SÃO PAULO, Maio de 1988-

À minha mãe,
e aos meus tios
Fraga e Belita.

Agradecimentos

Agradeço a todos os colegas e amigos do IME-USP e DMT-UFMS que direta ou indiretamente colaboraram para a realisação deste trabalho. Em particular, agradeço ao Prof. Routo Terada, meu orientador, pela sua amisade, e dedicação a esta dissertação.

RESUMO

Computação Paralela é uma forma de resolver problemas computacionais na qual vários processadores trabalham simultaneamente para a obtenção da solução. A idéia básica para se conseguir colocar em prática este processo, é dividir o problema em várias partes onde cada uma delas pode ser resolvida, ao mesmo tempo, por um processador.

Neste trabalho propomos um estudo sobre algoritmos paralelos e suas complexidades para alguns problemas computacionais, visando compreender os mecanismos e as idéias que regem a teoria da computação paralela na ciência da computação. Mais precisamente, trataremos dos problemas do cálculo do determinante e ordenação; Apresentaremos, também, um estudo sobre limites inferiores paralelos para alguns problemas, e uma breve introdução aos principais modelos de computação paralela.

ABSTRACT

Parallel computation is a form to solve computational problems in that several processors work simultaneously to get the solution. The basic idea is to break a problem into smaller parts, which are solved simultaneously, each by a different processor.

Our aim is to study about parallel algorithms, and their complexity, to solve some computational problems, to understand the mechanisms and ideas in this area. More precisely, we deal with the problems of computing the determinant and sorting; we further present a study about parallel lower bonds to some problems, and a short introduction to the main models of parallel computation.

índice

Introdução 1

Capítulo I - **A Classe NC e o Determinante**

1.1 Introdução 2
1.2 Avaliação de algoritmos 3
1.3 Famílias de circuitos uniformes 4
1.4 A classe NC^1 e redutibilidade NC^1 5
1.5 Anéis bem-dotados 6
1.6 Algoritmo paralelo para o determinante 15
1.7 A classe DET 22
1.8 Comentários finais 23

Capítulo II - **Ordenação**

2.1 Introdução 26
2.2 Ordenação por intercalação par-ímpar 27
2.3 Ordenação por intercalação bitônica 31
2.4 O embaralhamento perfeito 38
2.5 Comentários Finais 50

Capítulo III - **Limites Inferiores**

3.1 Introdução 51
3.2 O máximo 52
3.3 Intercalação 55
3.4 Ordenação 58
3.5 Comentários finais 59

Apêndice - **Introdução a Modelos de Computação Paralela**

4.1 Introdução 60
4.2 Modelo SIMD 61
4.3 Modelo SIMDAG 62
4.4 Modelo máquina vetorial 63
4.5 Modelo máquina de Turing alternada 64
4.6 Comentários Finais 67

INTRODUÇÃO

Devido ao crescente número de áreas nas quais os computadores vem sendo usados, tem sido crescente a necessidade de computadores cada vez mais rápidos do que os que existem atualmente. No entanto está se tornando aparente que será impossível encontrar um crescimento significativo na velocidade de processamento usando simplesmente dispositivos eletrônicos mais rápidos, como foi feito nas últimas décadas. Isto ocorre, por um lado, devido ao fato de que o tempo gasto para que uma informação seja levada de um dispositivo eletrônico para outro está se tornando maior do que o tempo gasto para processá-la. Por outro lado, a redução das distâncias entre os dispositivos, através da tecnologia VLSI, está encontrando um limite através do qual decresce a velocidade e a confiança na transferência dos dados.

Uma alternativa para se conseguir crescimento substancial na velocidade de processamento, é o uso de um computador paralelo, ou seja, um que possui várias unidades de processamento, ou processadores, onde todos trabalham simultaneamente para a solução de um determinado problema. A idéia básica aqui, é dividir o problema em várias partes, e cada uma das quais é resolvida ao mesmo tempo por um processador. Como um simples exemplo, considere o problema de buscar um elemento em um arquivo. Com n processadores disponíveis, o arquivo pode ser subdividido em n subarquivos, cada um dos quais sendo pesquisado por um processador. Um computador deste tipo levaria $1/n$ do tempo gasto por um computador sequencial.

O nosso objetivo neste trabalho é estudar os mecanismos e as idéias que regem a teoria da computação paralela na ciência da computação. Nós o desenvolvemos do seguinte modo: O capítulo I é desenvolvido inteiramente com o objetivo de apresentar um algoritmo paralelo para o determinante. Neste capítulo apresentamos também os principais critérios para a avaliação de algoritmos paralelos, e definimos as classes NC^k de problemas; estas classes são importantes para a classificação de problemas segundo suas complexidades paralelas. No capítulo II tratamos do problema da ordenação; neste capítulo apresentamos três algoritmos para a ordenação e discutimos brevemente os demais algoritmos existentes. No capítulo III, tratamos de limites inferiores paralelos para os problemas do máximo, intercalação e ordenação; e como apêndice apresentamos uma breve introdução aos principais modelos de computação paralela.

No decorrer deste trabalho, são feitas várias referências a objetos da Teoria dos Grafos, Teoria das Linguagens Formais e Autômatos, e Teoria da Complexidade. Para o leitor que não tiver formação nestas áreas, uma explicação melhor para estes objetos pode ser encontrada em [BM], [HU] e [AHU].

Capítulo I

A Classe NC
e o Determinante

1.1 - INTRODUÇÃO

Nós estamos interessados em classes de problemas que são resolvidos rapidamente (em tempo polinomial em $\log n$, onde n é o comprimento da instância do problema) por um Computador Paralelo com um número viável (isto é, polinomial em n) de processadores. Estas classes foram identificadas e caracterizadas por Nick Pippenger e são agora chamadas NC ("Nick's Class"). Desde então estas classes têm sido estudadas e atualmente elas incluem uma grande variedade de problemas. O que faremos agora é estudar essas classes e dar exemplos de problemas que estão nelas.

Até o presente momento não existe um modelo de Computador Paralelo que sirva de base para produção em grande escala. Um modelo mais atraente e que também é candidato a amparar uma teoria matemática duradoura é um modelo chamado *Família de Circuitos Booleanos Uniformes*. Como todos os computadores reais são construídos a partir de circuitos, circuitos representam um modelo mais fundamental que outros usualmente considerados. Também a complexidade de circuitos tem sido estudada desde 1949 [Co]. Finalmente, as Classes de Complexidade definidas em termos de famílias de circuitos têm uma caracterização precisa em termos de Máquina de Turing Alternada. Sendo assim, de agora em diante, estudaremos algoritmos que consideram o modelo de Circuitos Booleanos Uniformes.

O que faremos neste capítulo é o seguinte: Na seção 1.2 introduziremos os principais critérios para a avaliação de algoritmos. Na seção 1.3 definiremos Família de Circuitos Booleanos Uniformes e a Classe NC^k de problemas que podem ser resolvidos por uma Família de Circuitos Booleanos Uniformes em tempo $O(\log^k n)$ usando um número polinomial de processadores. Na seção 1.4 conceituaremos Redutibilidade NC^1. Na seção 1.5 introduziremos o conceito de Anel Bem-Dotado. Tal anel tem representação na qual soma e multiplicação podem ser eficientemente implementadas (em NC^0 e NC^1 respectivamente). Estes resultados serão aplicados na seção 1.6 onde desenvolveremos um algoritmo [Be] para calcular o determinante de uma matriz em tempo paralelo $O(\log^2 n)$. Na seção 1.7 mostraremos a equivalência, em tempo paralelo, entre os problemas: determinante, potência de matriz, produtos iterados de matrizes, e inversa de matriz.

1.2 - AVALIAÇÃO DE ALGORITMOS

Nos parágrafos que segue, definiremos várias funções que serão utilizadas na avaliação e comparação de algoritmos paralelos.

1.2.1 - Tempo de Execução

Como a rapides de execução é a razão de ser para a computação paralela, *tempo de execução* em paralelo é, provavelmente, a mais importante medida na avaliação de um algoritmo paralelo. É definido como sendo o tempo nacessário para resolver um problema, ou seja, o tempo gasto do início ao fim do algoritmo. O tempo de execução é geralmente obtido contando dois tipos de instruções executadas pelo algoritmo: As instruções de transferência de dados entre os processadores, e as operações aritméticas e lógicas executadas dentro de cada processador.

Uma boa indicação da qualidade de um algoritmo paralelo é o *aumento da velocidade* de execução que ele produz. Este aumento é definido como:

$$aumento = \frac{\textit{tempo de execução, no pior caso, do mais rápido algoritmo sequencial conhecido para o problema}}{\textit{tempo de execução, no pior caso, do algoritmo paralelo}}.$$

É claro que quanto maior for esta taxa, melhor será o algoritmo paralelo. No melhor caso, o que se espera, é encontrar um aumento de velocidade de N quando se resolve um problema usando N processadores. Na prática, tal aumento, em geral, é difícil de ser encontrado, pois

(1) Na maioria dos casos é difícil decompor um problema em N partes, cada um requerendo $1/N$ do tempo gasto por um processador para resolver o problema original, e

(2) O modelo de computador paralelo usado para resolver o problema impõe restrições que torna o tempo desejado inatingível.

1.2.2 - Número de Processadores

Outro critério de avaliação de um algoritmo paralelo é o *número de processadores* necessários para resolver um problema. Por este critério, é evidente que quanto menor for o número de processadores utilizados, melhor é o algoritmo.

1.2.3 - Custo

O *custo* de um algoritmo paralelo é definido como o produto das duas medidas anteriores. Ou seja,

$$custo = \textit{tempo de exec.} \times \textit{número de proc. utilizados.}$$

Em outras palavras, custo é igual o número de passos (sequenciais) executados na solução do problema, no pior caso. Se o custo de um algoritmo paralelo é igual ao limite inferior

sequencial para um dado problema, o algoritmo paralelo é dito ser de *custo ótimo*. No caso particular da ordenação, um algoritmo paralelo cujo custo é $O(n \log n)$, é de custo ótimo. Além disso, quando não se conhece um limite inferior, a *eficiência* do algoritmo paralelo, definida por

$$\textit{eficiência} = \frac{\textit{tempo de execução, no pior caso, do mais rápido algoritmo sequencial conhecido para o problema}}{\textit{custo do algoritmo paralelo}},$$

é usada para a avaliação.

Na maior parte dos casos, eficiência ≤ 1; caso contrário, um algoritmo sequencial mais rápido pode ser obtido a partir do paralelo.

1.2.4 - Outras Medidas

Além dos três critérios descritos acima, outras medidas são, às vezes, usadas na avaliação de um algoritmo paralelo. Por exemplo, se o computador paralelo é construído usando tecnologia VLSI, onde aproximadamente 10^6 portas lógicas podem ser colocadas em uma placa de 1 cm^2, então a *área ocupada* pelos processadores e suas interligações (comprimento dos fios) devem ser levadas em consideração.

1.3 - FAMÍLIAS DE CIRCUITOS UNIFORMES [Co]

Um *circuito* [Co] (booleano) C com n entradas e m saídas é um grafo orientado acíclico e finito com nós (chamados portas) rotulados como segue: O circuito C tem n "nós de entrada" $x_1, \ldots, x_n$, cada um com grau de entrada zero. Os outros nós que têm grau de entrada zero são rotulados com 0 ou 1. Todos os nós com grau de entrada 1 são rotulados "inversor". Todos os outros nós restantes têm grau de entrada 2 e são rotulados $\wedge$ ("e") ou $\vee$ ("ou"). Os nós de saída são $y_1, \ldots, y_m$. Para cada nó de entrada existe um caminho que vai dele a algum nó de saída. Usamos $s(C)$ para denotar o número de nós de C e $d(C)$ para denotar o comprimento de um passeio mais longo de um nó de entrada a um nó de saída. O circuito C computa uma função $f : \{0,1\}^n \mapsto \{0,1\}^m$ se $f(x) = y$ se e somente se C com entrada x obtém y como saída (Fig. abaixo).

$$f : \{0,1\}^n \mapsto \{0,1\}^m \quad f(x) = y \quad \Leftrightarrow \quad x \in \{0,1\}^n \left\{ \begin{array}{c} \rightarrow \\ \rightarrow \\ \vdots \\ \rightarrow \end{array} \right. \boxed{\quad C \quad} \left. \begin{array}{c} \rightarrow \\ \rightarrow \\ \vdots \\ \rightarrow \end{array} \right\} y \in \{0,1\}^m$$

Fig. 1.1 - O Circuito C computa a função f.

Em geral, nós estamos interessados em computar uma função $f = < f_n >$ onde $f_n : \{0,1\}^{g(n)} \mapsto \{0,1\}^{h(n)}$, $g(n)$ é monotônica e estritamente crescente e $g(n) = n^{o(1)}$*,

* $g(n) = n^{o(1)}$ se e somente se $g(n) = O(n^k)$ para alguma constante k

(a função f será tratada como a união sobre n de f_n). Uma *Família de Circuitos* com entrada de tamanho g e saída de tamanho h é uma sequência $< C_n >$, onde C_n é um circuito com $g(n)$ entradas e $h(n)$ saídas. A família C_n *computa* a função f se e somente se C_n computa f_n para todo n.

Por exemplo, a função "fecho transitivo em um grafo" tem $g(n) = h(n) = n^2$. O argumento x para $f_n(x)$ é uma sequência de n^2 bits representando a matriz $n \times n$ de adjacência linha por linha para um grafo orientado G com n vértices. O valor $f_n(x)$ é uma sequência de n^2 bits representando o fecho transitivo de G.

Neste estudo restringiremos nossa atenção para Famílias de Circuitos "Uniformes", ou seja, Famílias $< C_n >$ para as quais existe algum algoritmo que, dado n, gera efetivamente o n-ésimo circuito C_n. Existem várias razões para exigir uniformidade. Primeiro, pode-se querer exibir C_n para vários valores de n, e isto não seria possível sem alguma condição de uniformidade. Além disso, Famílias de Circuitos Uniformes definem classes de complexidade que têm relações interessantes com classes tradicionais definidas por tempo e espaço. O conceito de uniformidade não será formalizado aqui, mas pode ser encontrado em [Ru].

Definição [Co]. Um *problema* $\mathcal{R}$ é uma família $< R_n >$ de relações binárias tal que $R_n \subseteq \{0,1\}^{g(n)} \times \{0,1\}^{h(n)}$. Uma família de circuitos $< C_n >$ *resolve* o problema $\mathcal{R}$ se e somente se a função $< f_n >$ computada por $< C_n >$ realiza $\mathcal{R}$ no seguinte sentido: Para cada n e cada x em $\{0,1\}^{g(n)}$, se $R_n(x,y)$ existe para algum y, então $R_n(x, f_n(x))$ existe.

Para ilustrar a definição acima, se o problema $\mathcal{R}$ é encontrar uma floresta espalhada, então $R_n(x,y)$ existe sempre que x codificar um grafo não orientado G e y codificar uma floresta espalhada para G. A família $< \alpha_n >$ resolve $\mathcal{R}$ se e somente se para todo n, quando as entradas para α_n codificarem um grafo G, as saídas de α_n codificarem uma floresta espalhada para G.

Definição [Co]. NC^k é o conjunto de todos os problemas $\mathcal{R}$ resolvidos por uma Família de Circuitos Uniformes $< C_n >$ com $s(C_n) = n^{o(1)}$ (isto é, $s(C_n)$ é limitada por algum polinômio em n) e $d(C_n) = O(\log^k n)$. $NC = \bigcup_k NC^k$.

1.4 - A CLASSE NC^1 E REDUTIBILIDADE NC^1 [Co]

A classe NC^1 consiste de todos os problemas que são resolvidos por uma Família de Circuitos Booleanos Uniformes $< C_n >$ com complexidade $s(C_n) = n^{o(1)}$ e $d(C_n) = O(\log n)$, onde n é o tamanho da entrada. Exemplos de problemas em NC^1 são: Soma e produto de dois inteiros com n dígitos, soma de n números inteiros, multiplicação de matrizes inteiras e ordenação de n números inteiros. Circuitos para estes problemas, com exceção do último, serão descritos neste capítulo.

Em estudo da complexidade de tempo sequencial, redutibilidade em tempo polinomial tem sido padrão. No estudo da Computação Paralela, aparentemente a redutibilidade NC^1 é a mais apropriada. O conceito de redutibilidade introduzido agora é uma ferramenta muito importante para a classificação de problemas segundo as classes NC^k. A seção 1.6 é um exemplo disto.

Definição [Co]. Um problema $\mathcal{R}$ é NC^1 *redutível* para $\mathcal{S}$ ($\mathcal{R} \preceq \mathcal{S}$) se e somente se existe uma Família de Circuitos Uniformes $< \alpha_n >$ que resolve $\mathcal{R}$, onde $d(\alpha_n) = O(\log n)$ e para α_n é permitido ter um número constante (em paralelo) de nós oráculos para $\mathcal{S}$. Um nó *oráculo* para $\mathcal{S}$ é um nó com uma sequência $< y_1, \ldots, y_r >$ de arestas de entrada e uma sequência $< z_1, \ldots, z_s >$ de arestas de saída tais que $\mathcal{S}(y_1 \ldots y_r, z_1 \ldots z_s)$ existe.

Exemplos de reduções NC^1 são encontrados no Teorema 1.1, onde o problema de computar o produto de duas matrizes é reduzido à multiplicação de dois números (esta redução também pode ser vista em relação a somas iteradas), e na seção 1.7.

Observe que a relação $\preceq$ é transitiva e reflexiva. O *Fecho* C^* de uma classe C de problemas (sob $\preceq$) consiste de todos os problemas $\mathcal{R}$ tais que $\mathcal{R} \preceq \mathcal{S}$ para algum $\mathcal{S}$ em C. A classe C é *Fechada* se e somente se $C = C^*$.

Proposição 1.1 [Co]. A classe NC^k é fechada sob $\preceq$ para todo $K \geq 1$.

Dem. Suponha $\mathcal{R} \preceq \mathcal{S}$ e $\mathcal{S} \in NC^k$. Suponha que $< \alpha_n >$ realiza a redução $\mathcal{R} \preceq \mathcal{S}$, e $< \beta_n >$ resolve $\mathcal{S}$ em NC^k. Então uma família $< \gamma_n >$ para resolver $\mathcal{R}$ em NC^k pode ser construída tomando γ_n como sendo α_n onde cada nó oráculo para S_m é trocado por β_m. Para ver que $d(\gamma_n) = O(\log^k n)$, considere qualquer caminho p em γ_n e suponha que p passa pelos circuitos $\beta_{m_1}, \ldots, \beta_{m_t}$ que substituíram nós oráculos em α_n. Então o comprimento de p é no máximo $\sum_{j=1}^{t} d(\beta_{m_j}) + O(\log n) = O(\sum_{j=1}^{t} \log^k m_j) + O(\log n) = O(\log^k n)$. A uniformidade de $< \gamma_n >$ é provada a partir da uniformidade de $< \alpha_n >$ e $< \beta_n >$.

Definição [Co]. Um problema $\mathcal{R}$ é *Difícil* (ou NC^1 *Difícil*) para a classe C se e somente se $\mathcal{S} \preceq \mathcal{R}$ para todo $\mathcal{S}$ em C. $\mathcal{R}$ é *Completo* (ou NC^1 *Completo*) para C se e somente se $\mathcal{R}$ é Difícil para C e $\mathcal{R} \in C$.

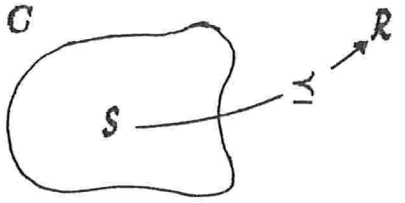


Fig. 1.2 (a) - $\mathcal{R}$ é difícil para C

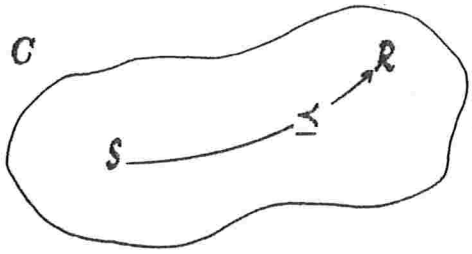


Fig. 1.2 (b) - $\mathcal{R}$ é completo para C

1.5 - ANÉIS BEM-DOTADOS [BCP]

Informalmente, um Anel Bem-Dotado é aquele cujos elementos podem ser representados por sequências de bits de tal forma que a soma possa ser computada em tempo (paralelo) constante e a multiplicação possa ser computada em tempo $O(\log n)$. A representação dos elementos do Anel pode ser redundante, ou seja, um elemento do anel pode ter mais que um representante, mas deve ser sucinta no sentido que elementos "pequenos" usem poucos bits para serem representados. Dessa maneira, no Anel deve existir uma noção de tamanho, ou comprimento para seus elementos.

Seja $\mathcal{A}$ um anel. Uma *função de comprimento* para $\mathcal{A}$ é uma função $\alpha : \mathcal{A} \mapsto N$ satisfazendo:

$$1) \alpha(x+y) \leq max\{\alpha(x), \alpha(y)\} + O(1)$$
$$2) \alpha(x.y) \leq \alpha(x) + \alpha(y) + O(\log max\{\alpha(x), \alpha(y)\})$$

para todo x, y em $\mathcal{A}$. Para ser conciso diremos "o Anel $(\mathcal{A}, \alpha)$" para representar "o Anel $\mathcal{A}$ com função de comprimento α".

Seja $f : \mathcal{A} \mapsto \mathcal{B}$ um mapeamento do Anel $(\mathcal{A}, \alpha)$ no Anel $(\mathcal{B}, \beta)$. Uma *função limite* para f é uma função $\phi : N \mapsto N$ tal que

$$\beta(f(x)) \leq \phi(\alpha(x)).$$

Denotaremos "mapeamento f com função limite ϕ" por "mapeamento (f, ϕ)".

Para um Anel $(\mathcal{A}, \alpha)$ e um número natural n, denotaremos por $\mathcal{A}_n$ o conjunto dos elementos x em $\mathcal{A}$ tal que $\alpha(x) \leq n$.

Uma *representação* para um Anel $(\mathcal{A}, \alpha)$ é um par (l, r) composto por uma função $l : N \mapsto N$ e uma sequência $r = \{r_1, r_2, \ldots\}$ de funções tais que $r_n : \{0,1\}^{l(n)} \mapsto \mathcal{A}$ e $\mathcal{A}_n$ está contido em $r_n(\{0,1\}^{l(n)})$ para todo n em N. Uma representação será chamada *sucinta* se $l(n) = n^{o(1)}$ (isto é, se $l(n)$ é limitada superiormente por um polinômio em n), e *uniforme* se para todo $x \in \mathcal{A}$ existe uma sequência $\{u_1, u_2, \ldots\}$ de espaço-log e uniforme tal que u_k está em $\{0,1\}^{l(\alpha(x)+k)}$ e $r_{\alpha(x)+k}(u_k) = x$ para todo $k \geq 1$. (espaço-log e uniforme significa que alguma Máquina de Turing pode, para todo k, gerar u_k sobre uma fita de saída usando espaço de trabalho $O(\log |u_k|)$). Usaremos a notação "o Anel $(\mathcal{A}, \alpha, l, r)$" para "o Anel $(\mathcal{A}, \alpha)$ com representação (l, r)".

Uma *implementação* para um mapeamento (f, ϕ) de um anel $(\mathcal{A}, \alpha, l, r)$ para um anel $(\mathcal{B}, \beta, m, t)$ é uma sequência $F = \{F_1, F_2, \ldots\}$ de funções tal que $F_n : \{0,1\}^{l(n)} \mapsto \{0,1\}^{m(\phi(n))}$ e

$$t_{\phi(n)}(F_n(u)) = f(r_n(u))$$

Para todo n em N e u em $\{0,1\}^{l(n)}$. No caso mais geral em que $f : \mathcal{A}^k \mapsto \mathcal{B}$, uma implementação para f é definida de maneira similar à acima.

Por *soma* em $\mathcal{A}$ denotaremos o mapeamento $f : \mathcal{A} \times \mathcal{A} \mapsto \mathcal{A}$ tal que $f(x, y) = x + y$. Por *negação* em $\mathcal{A}$ denotaremos o mapeamento $f : \mathcal{A} \mapsto \mathcal{A}$ tal que $f(x) = -x$ e por *multiplicação* em $\mathcal{A}$ denotaremos o mapeamento $f : \mathcal{A} \times \mathcal{A} \mapsto \mathcal{A}$ tal que $f(x, y) = xy$. Uma implementação para soma ou negação será chamada *eficiente* se ela está em NC^0 e uma implementação para multiplicação será chamada eficiente se ela está em NC^1. Dizemos que um Anel $(\mathcal{A}, \alpha)$ é *bem-dotado* se existe uma representação sucinta e uniforme para a qual existem implementações eficientes para soma, negação e multiplicação.

Considerando o Anel dos inteiros, seja $\xi(x) = \lceil \log_p(|x| + 1) \rceil$ (o número de dígitos p-ários necessários para representar x).

Fato 1. $\xi(x) = \lceil \log_p(|x| + 1) \rceil$ é uma função de comprimento para $\mathbb{Z}$.

Dem. Sejam x e y em $\mathbb{Z}$ e suponha, sem perda de generalidade, que $|x| \geq |y|$.

$$|x| \geq |y| \Rightarrow \lceil \log_p(|x| + 1) \rceil \geq \lceil \log_p(|y| + 1) \rceil \Rightarrow \xi(x) \geq \xi(y) \Rightarrow$$
$$\Rightarrow \xi(x) = max\{\xi(x), \xi(y)\}.$$

$$\xi(x + y) = \lceil \log_p(|x + y| + 1) \rceil \leq \lceil \log_p(|x| + |y| + 1) \rceil \leq$$
$$\leq \lceil \log_p(2|x| + 2) \rceil = \lceil \log_p 2(|x| + 1) \rceil \leq \lceil \log_p p(|x| + 1) \rceil =$$

$$= 1 + \lceil \log_p(|x|+1) \rceil = 1 + \xi(x) = max\{\xi(x), \xi(y)\} + 1.$$
$$\Rightarrow \xi(x+y) \leq max\{\xi(x), \xi(y)\} + O(1).$$

Se $x = 0$ ou $y = 0$ então $\xi(x.y) = 1 \leq \xi(x) + \xi(y) + O(\log max\{\xi(x), \xi(y)\})$. Suponha então $x \neq 0$ e $y \neq 0$.

$$\begin{aligned}
\xi(x.y) &= \lceil \log_p(|x.y|+1) \rceil = \lceil \log_p(|x|.|y|+1) \rceil \leq \\
&\leq \lceil \log_p(2|x|.|y|) \rceil \leq \lceil \log_p(p|x|.|y|) \rceil = 1 + \lceil \log_p |x| + \log_p |y| \rceil \leq \\
&\leq 1 + \lceil \log_p |x| \rceil + \lceil \log_p |y|) \rceil \leq \\
&\leq 1 + \lceil \log_p(|x|+1) \rceil + \lceil \log_p(|y|+1) \rceil. \\
&\Rightarrow \xi(x.y) \leq \xi(x) + \xi(y) + O(\log max\{\xi(x), \xi(y)\}).
\end{aligned}$$

O que prova o fato 1. ∎

A representação comumente usada para os inteiros (o anel Z_n, $n \geq 2$) permite a construção de algoritmos para a soma e multiplicação cuja natureza é inerentemente sequencial, devido a propagação do "vai um"; Para quebrar essa propagação e possibilitar a construção de algoritmos mais rápidos, é preciso mudar a representação. Isso é o que faremos agora.

Para $p \geq 2$, *notação p-ária é uma representação sucinta e uniforme para* (Z, ξ), mas não produz uma implementação para soma em NC^0. Para $p \geq 3$ nós construiremos agora uma representação sucinta e uniforme (l, r) para (Z, ξ) para a qual existe implementação para soma e negação em NC^0 e implementação para multiplicação em NC^1.

Para $n \geq 1$ em N, defina $\rho_n : \{-(p-1), \ldots, -1, 0, 1, \ldots, (p-1)\}^n \mapsto Z$ por:

$$\rho_n(u_1, \ldots, u_n) = \sum_{1 \leq k \leq n} u_k p^{k-1}$$

$u_1, \ldots, u_n$ em $\{-(p-1), \ldots, -1, 0, 1, \ldots, (p-1)\}$. Observe que esta representação é redundante no sentido que um elemento inteiro pode ter mais que um representante. Por exemplo, se $p = 10$ então o elemento 3 pode ser representado por (3) ou $(-7, 1)$ ou ainda $(-7, -2, 3)$.

Z_n está contido em $\rho_n(\{-(p-1), \ldots, -1, 0, 1, \ldots, (p-1)\}^n)$ para todo $n \geq 1$ em N. Codificando cada símbolo de $\{-(p-1), \ldots, -1, 0, 1, \ldots, (p-1)\}$ como uma sequência de $\lceil \log_2 p + 1 \rceil$ símbolos em $\{0, 1\}$, podemos obter a partir de $\rho = (\rho_1, \rho_2, \ldots)$ uma representação (l, ρ) com $l(n) = \lceil \log_2 p + 1 \rceil n$. Esta representação é sucinta e uniforme, e será chamada representação *p-ária balanceada* para (Z, ξ). Esta representação é de autoria de Avizienis [BCP].

Para ver que soma (com função limite $\phi(n) = n + 1$) tem uma implementação em NC^0, suponha que são dados $u_1, \ldots, u_n$ e $v_1, \ldots, v_n$ em $\{-(p-1), \ldots, -1, 0, 1, \ldots, (p-1)\}$. Desejamos calcular $w_1, \ldots, w_{n+1}$ em $\{-(p-1), \ldots, -1, 0, 1, \ldots, (p-1)\}$ tal que

$$\rho_{n+1}(w_1, \ldots, w_{n+1}) = \rho_n(u_1, \ldots, u_n) + \rho_n(v_1, \ldots, v_n). \qquad (*)$$

Como, para $1 \leq k \leq n$, $|u_k| \leq p - 1$ e $|v_k| \leq p - 1$, temos $|u_k + v_k| \leq 2p - 2$. Então é possível escrever $u_k + v_k = px_k + y_k$ com $|x_k| \leq 1$ e $|y_k| \leq p - 2$ (aqui nós usamos a

hipótese $p \geq 3$). Se, para $1 \leq k \leq n+1$ fizermos $w_k = x_{k-1} + y_k$, onde $x_0 = 0$ e $y_{n+1} = 0$, então $|w_k| \leq p-1$ e a condição $(*)$ é satisfeita. Como w_k depende somente de u_k, u_{k-1}, v_k e v_{k-1}, w pode ser computado a partir de u e v em NC^0.

Uma implementação para este algoritmo pode ser feita por uma família de circuitos construída do seguinte modo: Cada circuito C_n é composto por dois tipos de processadores, P_1 e P_2. Os processadores do tipo P_1 têm as seguintes características:

(1) Cada processador tem dois nós de entrada e dois de saída.

(2) cada processador deve somar os valores contidos nos nós de entrada, e colocar o resultado da soma na forma $px + y$ com $|x| \leq 1$ e $|y| \leq p-2$. Feito isto, o valor de x deve ser apresentado em um nó de saída e o valor de y no outro (Fig. 1.3 (a)).

Os processadores do tipo P_2 têm as seguintes características:

(1) Cada processador tem dois nós de entrada e um nó de saída.

(2) cada processador deve somar os valores contidos nos nós de entrada, e apresentar o resultado da soma no nó de saída (Fig. 1.3 (b)).

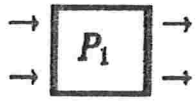


Fig. 1.3 (a) - Processador tipo P_1

Fig. 1.3 (b) - Processador tipo P_2

O circuito C_n para somar $\rho_n(u_1, \ldots, u_n) + \rho_n(v_1, \ldots, v_n)$ está ilustrado na figura 1.4. Observe que a soma é executada em tempo constante e o número de processadores utilizados é $2n+1$.

Exemplo. Suponha $p = 10$ e que queremos somar $(u_1, \ldots, u_4) = (2, -1, 5, 7)$ e $(v_1, \ldots, v_4) = (7, -8, 4, -2)$. A execução desta soma está ilustrada no circuito da figura 1.5.

Para ver que a negação (com função limite $\phi(n) = n$) tem implementação em NC^0, suponha que seja dado $u_1, \ldots, u_n$ em $\{-(p-1), \ldots, -1, 0, 1, \ldots, (p-1)\}$. Desejamos calcular $w_1, \ldots, w_n$ em $\{-(p-1), \ldots, -1, 0, 1, \ldots, (p-1)\}$ tal que

$$\rho_n(w_1, \ldots, w_n) = -\rho_n(u_1, \ldots, u_n).$$

Basta tomarmos $w_k = -u_k$ para $1 \leq k \leq n$ que teremos todas as condições satisfeitas. Como w_k depende somente de u_k, w pode ser computado a partir de u em NC^0.

O circuito para a negação não será feito aqui, mas pode ser construído a partir de processadores com as seguintes características: Cada processador tem um nó de entrada e um de saída, e sua função é tomar o nó de entrada u_k, e produzir como saída o valor $-u_k$.

Para ver que a multiplicação (com função limite $\phi(n) = 2n + \lceil \log_2 2n \rceil$) tem implementação em NC^1, suponha que sejam dados $u_1, \ldots, u_n$ e $v_1, \ldots, v_n$ em $\{-(p-1), \ldots, -1, 0, 1, \ldots, (p-1)\}$ e desejamos computar $w_1, \ldots, w_{\phi(n)}$ em $\{-(p-1), \ldots, -1, 0, 1, \ldots, (p-1)\}$ tal que

$$\rho_{\phi(n)}(w_1, \ldots, w_{\phi(n)}) = \rho_n(u_1, \ldots, u_n) \times \rho_n(v_1, \ldots, v_n).$$

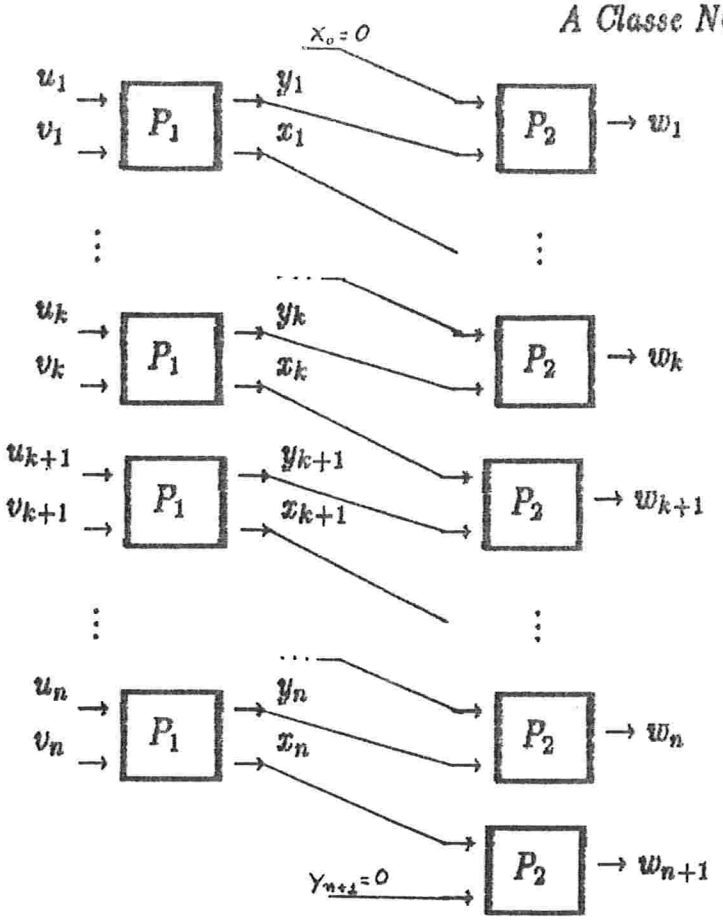


Fig. 1.4 - Circuito para somar $\rho_n(u_1, \ldots, u_n) + \rho_n(v_1, \ldots, v_n)$

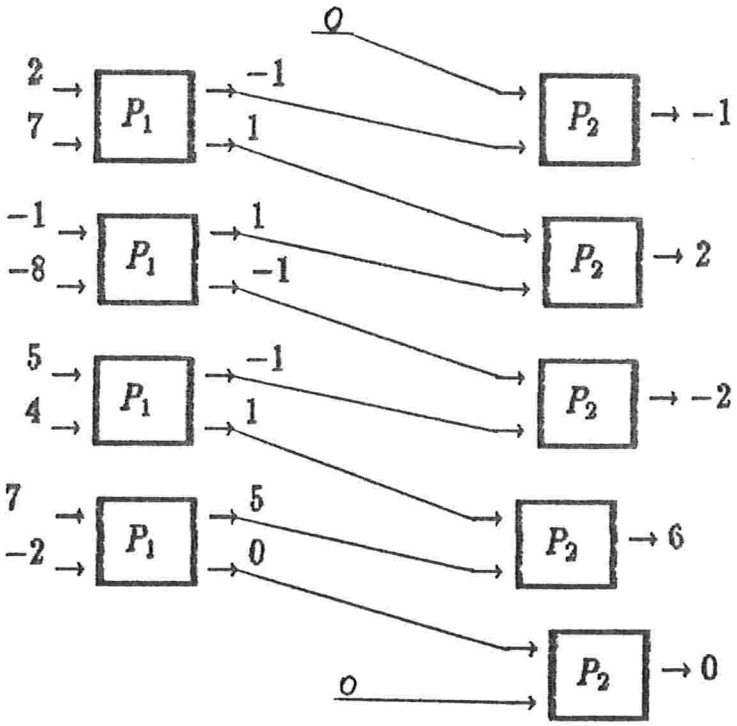


Fig. 1.5 - Circuito para somar $(u_1, \ldots, u_4) + (v_1, \ldots, v_4)$.

Como para $1 \leq i, j \leq n$, $|u_i| \leq p-1$ e $|v_j| \leq p-1$, temos $|u_i v_j| \leq (p-1)^2 \leq p^2 - 1$. Portanto, é possível escrever $|u_i v_j| = p.x_{ij} + y_{ij}$ com $|x_{ij}| \leq p-1$ e $|y_{ij}| \leq p-1$. Se fizermos

$$z_{ij} = \begin{cases} x_{i,j-i} & \text{se } i+1 \leq j \leq n+i \\ 0 & \text{caso contrário} \end{cases}$$

e

$$z_{n+i,j} = \begin{cases} y_{i,j-i+1} & \text{se } i \leq j \leq n+i-1 \\ 0 & \text{caso contrário} \end{cases}$$

Para $1 \leq i \leq n$ e $1 \leq j \leq 2n$, teremos então

$$\sum_{1 \leq i \leq 2n} \rho_{2n}(z_{i,1} \ldots, z_{i,2n}) = \rho_n(u_1, \ldots, u_n) \times \rho_n(v_1, \ldots, v_n).$$

Como z_{ij} depende somente de u_i e v_{j-i}, e $z_{n+i,j}$ depende somente de u_i e v_{j-i+1}, z pode ser computado a partir de x e y em NC^0. Se computarmos $w_1, \ldots, w_{\phi(n)}$ em $\{-(p-1), \ldots, -1, 0, 1, \ldots, (p-1)\}$ tal que

$$\rho_{\phi(n)}(w_1, \ldots, w_{\phi(n)}) = \sum_{1 \leq i \leq 2n} \rho_{2n}(z_{i,1} \ldots, z_{i,2n})$$

teremos terminado a prova. Por somas iteradas (veja Lema 1.1) podemos computar w a partir de z em NC^1.

A implementação do algoritmo acima para a multiplicação pode ser feita por um circuito construído do seguinte modo: O circuito é composto por dois tipos de processadores, P_1 e P_2. Os processadores do tipo P_1 possuem as seguintes características:

(1) Cada processador tem dois nós de entrada, u_i e v_{j-i}, e dois de saída, $x_{i,j-i}$ e $y_{i,j-i}$.

(2) Cada processador deve multiplicar os valores os valores de u_i e v_{j-i}, obter o resultado na forma $px_{i,j-i} + y_{i,j-i}$ e colocar $x_{i,j-i}$ e $y_{i,j-i}$ nos nós de saída.

Os processadores do tipo P_2 são rotulados por z_{pq} ou w_t, e possuem um nó de entrada e um de saída. Sua função é simplesmente armazenar o valor contido no nó de entrada e apresentá-lo ao nó de saída para posteriores operações.

Os processadores estão interconectados do seguinte modo: O nó de saída de cada processador do tipo P_1 que contém o valor $x_{i,j-i}$ ($y_{i,j-i}$) é interconectado ao nó de entrada do processador $z_{i,j}$ ($z_{n+i,j-1}$) (fig. abaixo).

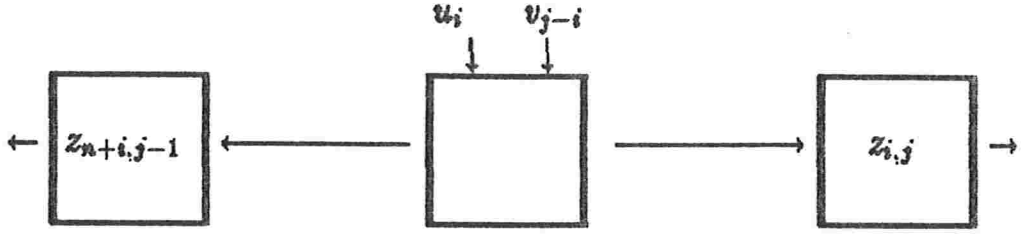


Fig. 1.6 - Interconexão entre os Processadores no circuito para a multiplicação.

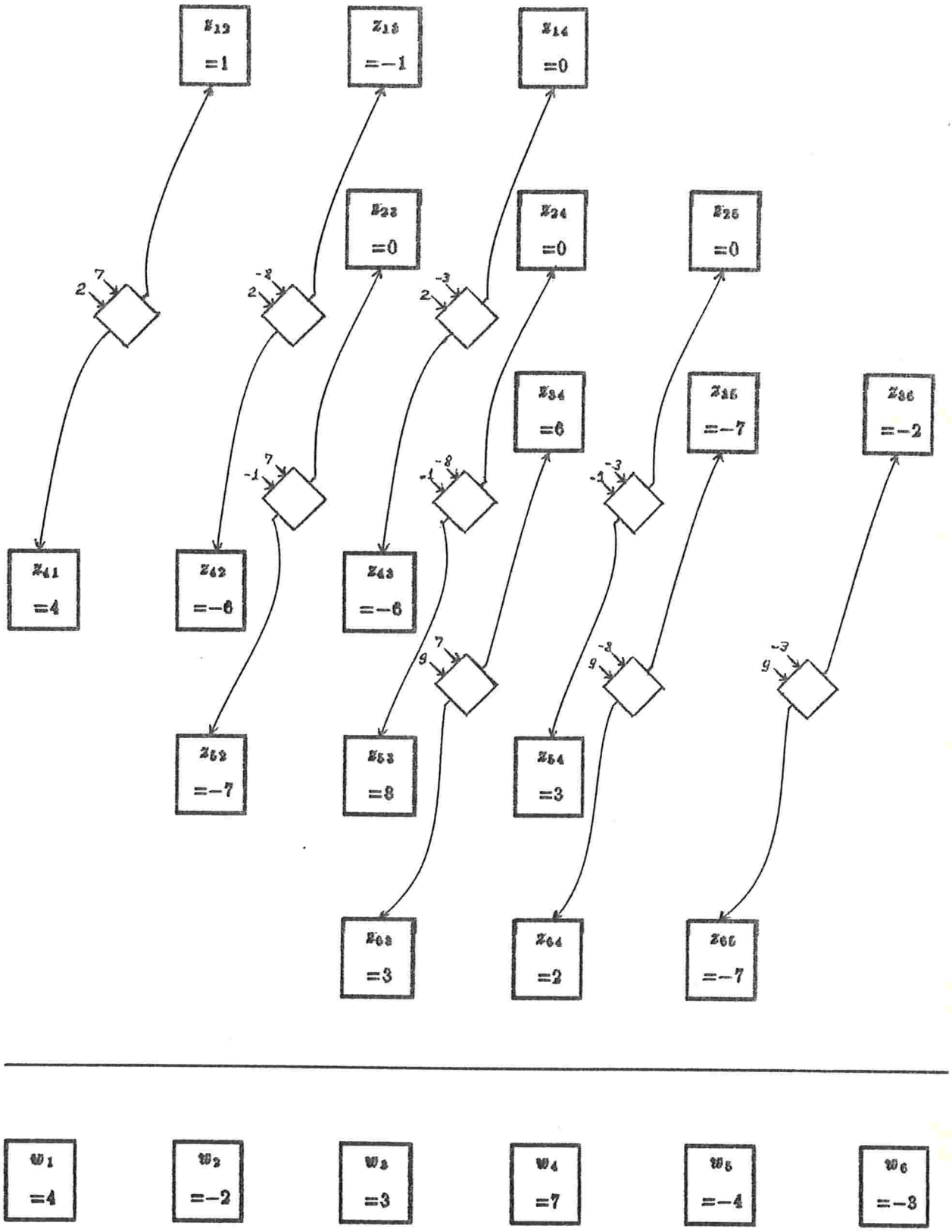


Fig. 1.7 - Circuito para multiplicar $(u_1, u_2, u_3) \times (v_1, v_2, v_3)$

Pela aplicação do Lema 1.1, os processadores $z_{i,1}, \ldots, z_{i,2n}$ são interconectados em forma de uma árvore binária com $z_{i,1}, \ldots, z_{i,2n}$ nas folhas. Na raiz da árvore encontra-se o processador w_i que conterá o valor $z_{i,1} + \cdots + z_{i,2n}$.

A partir de u e v podemos obter z em tempo constante. De posse de $z_{i,1}, \ldots, z_{i,2n}$ podemos obter o valor w_i em tempo $O(\log n)$. Portanto, o circuito acima computa a multiplicação de inteiros em NC^1.

Exemplo. *Suponha $p = 10$ e que queremos multiplicar $(u_1, u_2, u_3) = (2, -1, 9)$ por $(v_1, v_2, v_3) = (7, -8, -3)$.* O circuito da figura 1.7 executa esta multiplicação.

O que acabamos de demonstrar foi o seguinte:

Proposição 1.2 [BCP]. O anel dos inteiros é bem-dotado, usando representações p-árias balanceadas para $p \geq 3$. ∎

Lema 1.1 [BCP]. Somas iteradas em Z tem implementação em NC^1 e função limite $\phi^*(n) = n + \lceil \log_2 k \rceil$, onde k é o número de elementos a ser somado e n é o número máximo de dígitos em cada elemento.

Dem. Usando representação p-ária balanceada ($p \geq 3$) para Z, suponha que sejam dados $u_1, \ldots, u_k$ em $\{-(p-1), \ldots, -1, 0, 1, \ldots, (p-1)\}^n$ e desejamos computar $u_1 + \cdots + u_k$. O circuito para a implementação de somas iteradas tem a forma de uma árvore binária com k folhas. Os processadores são representados pelos vértices da árvore e as interconexões pelas arestas. Em cada folha está colocado um elemento u_j e cada vértice que não é folha contém a soma dos valores do(s) seu(s) (dois) filho(s). Dessa forma, como a soma é associativa, teremos na raiz da árvore o valor $u_1 + \cdots + u_k$ (fig. 1.8).

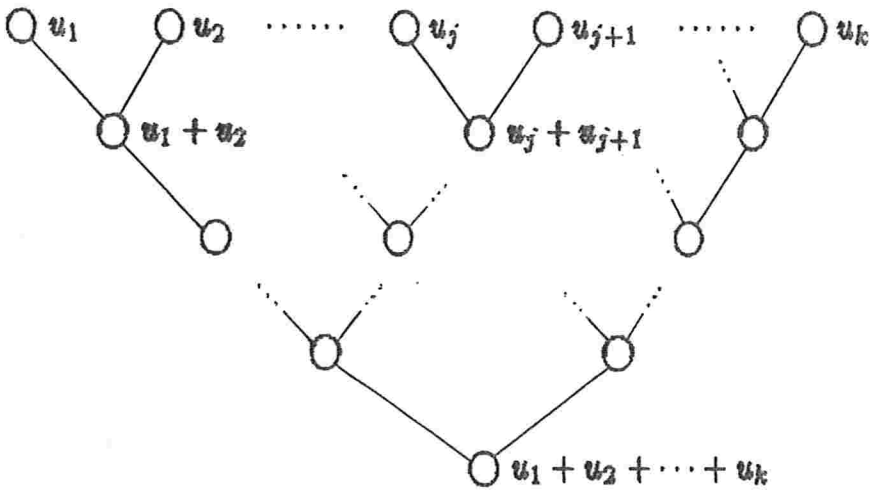


Fig. 1.8 - Circuito para a implementação de somas iteradas.

O número de nós deste circuito é no máximo $2k - 1$ e o comprimento de um caminho da folha até a raiz é $O(\log_2 k)$. Como a soma de dois elementos em Z pode ser feita em NC^0, temos que somas iteradas em Z tem implementação em NC^1.

Com relação a função limite ϕ^* para somas iteradas, vamos mostrar que $\phi^*(n) = n + \lceil \log_2 k \rceil$ fazendo indução em k. Para $k = 2$ temos $\phi^*(n) = \phi(n) = n + 1 = n + \lceil \log_2 k \rceil$. Suponha então que desejamos somar $k \geq 3$ elementos em Z pelo processo da árvore

binária descrito acima. Se k não é uma potência de dois então complete o conjunto com zeros até que tenhamos $k = 2^x$ elementos para somar. Seja r a raiz da árvore, r_1 e r_2 seus filhos esquerdo e direito respectivamente. Então as subárvores esquerda e direita da raiz contém cada uma $k/2$ folhas, r_1 contém a soma dos elementos $u_1, \ldots, u_{k/2}$ e r_2 contém a soma dos elementos $u_{(k/2)+1}, \ldots, u_k$. Por hipótese de indução, temos em r_1 e r_2 que $\phi^*(n) = n + \lceil \log_2 k/2 \rceil$. Temos então em r que

$$\phi^*(n) = n + \lceil \log_2 k/2 \rceil + 1 = n + \lceil \log_2 k \rceil. \qquad \rfloor$$

Teorema 1.1. O produto de matrizes pode ser computado em NC^1 com $O(n^3)$ processadores.

Dem. Seja $\mathcal{A}$ um anel bem-dotado. Sejam A e B duas matrizes de ordem $n \times n$ em $\mathcal{A}$ e $C = A * B$. Cada elemento C_{ij} é determinado por $C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$. Ou seja, C_{ij} é determinado por n produtos e $n-1$ somas. usando n processadores podemos fazer os produtos, em paralelo, em uma unidade de tempo. Feito os produtos, usando mais $n-1$ processadores podemos fazer as somas em tempo $O(\log n)$ (Lema 1.1). Temos então determinado um elemento C_{ij} em tempo $O(\log n)$ usando $2n-1$ processadores. Como temos n^2 elementos na matriz C para serem calculados, podemos executar este processo n^2 vezes em paralelo e determinarmos a matriz produto C em tempo $O(\log n)$ usando $n^2(2n-1)$ processadores. Portanto, produto de matrizes pertence a classe NC^1 com $O(n^3)$ procesadores. $\rfloor$

A partir do teorema acima, podemos construir um circuito para computar $C_{ij} = \sum_{k=1}^{n} A_{ik} * B_{kj}$ do seguinte modo: Tomamos n circuitos para multiplicação de dois números inteiros (Proposição 1.1) e um circuito para somas iteradas de n números inteiros (Lema 1.1). C_{ij} é computado pelo seguinte circuito.

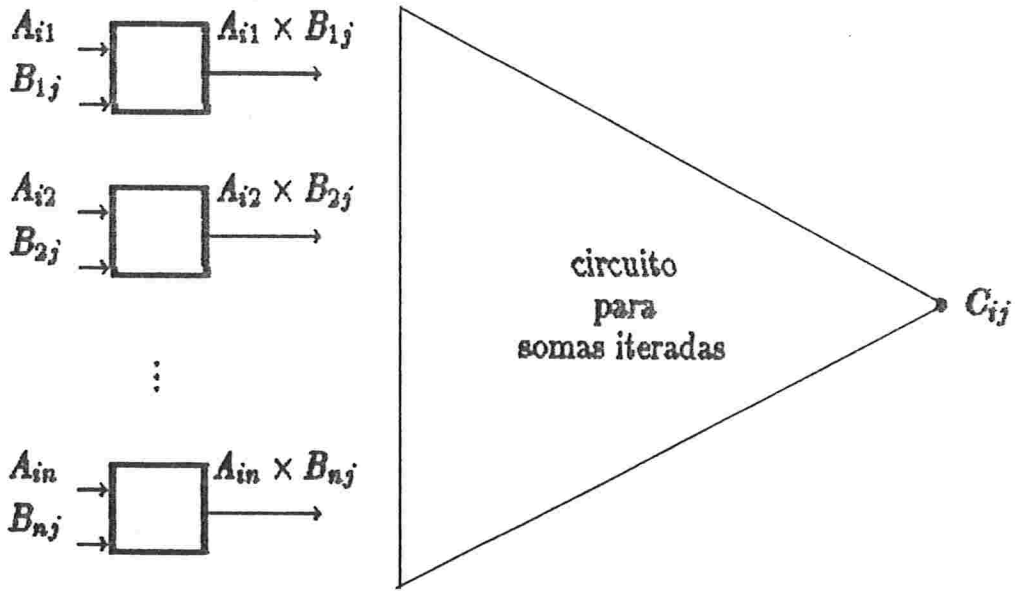


Fig. 1.9 - Circuito para computar C_{ij}

A matriz $C = A \times B$ pode ser computada a partir de n^2 cópias do circuito acima.

Observe que este algoritmo exige que sejam feitas n cópias de cada elemento das matrizes A e B, e que estas cópias não foram computadas no cálculo da complexidade do algoritmo. Nós estamos supondo desprezável este "tempo para tirar cópias de um elemento".

Usando o mesmo raciocínio do lema 1.1 podemos provar o seguinte:

Lema 1.2. Produtos iterados de matrizes com elementos em Z tem implementação em NC^2 usando $O(n^4)$ processadores. ∎

1.6 - ALGORITMO PARALELO PARA O DETERMINANTE

O algoritmo que apresentaremos a seguir é devido a Berkowitz [Be]. Este algoritmo computa o determinante de uma matriz A de ordem $n \times n$, via polinômio característico de A ($p(\lambda) = det(A - \lambda * I)$), em tempo paralelo $O(\log^2 n)$ usando $O(n^4)$ processadores.

Seja $\mathcal{A}$ um Anel Bem-Dotado comutativo com elemento unidade. Seja A uma matriz de ordem $n \times n$ sobre $\mathcal{A}$. Para $1 \leq t \leq n-1$ definimos as submatrizes R_t, S_t e M_t como segue:

$$R_t = (a_{t,t+1}, \ldots, a_{t,n})$$
$$S_t = (a_{t+1,t}, \ldots, a_{n,t})$$
$$M_t = \begin{pmatrix} a_{t+1,t+1} & \cdots & a_{t+1,n} \\ \vdots & \ddots & \vdots \\ a_{n,t+1} & \cdots & a_{nn} \end{pmatrix}.$$

Façamos $R = R_1$, $S = S_1$ e $M = M_1$. A matriz A se relaciona com R, S e M por:

$$A = \begin{pmatrix} a_{11} & R \\ S & M \end{pmatrix}.$$

Sejam $p(\lambda)$ e $q(\lambda)$ os polinômios característicos de A e M respectivamente. Ou seja,

$$p(\lambda) = \sum_{i=0}^{n} p_{n-i}\lambda^i = det(A - \lambda * I),$$

$$q(\lambda) = \sum_{i=0}^{n-1} q_{n-i-1}\lambda^i = det(M - \lambda * I).$$

Podemos observar então que se calcularmos o polinômio característico $p(\lambda)$ da matriz A temos calculado $det(A)$, pois $det(A) = p(0)$. O que faremos agora é calcular $p(\lambda)$.

O determinante pode ser definido indutivamente pelo método da expansão dos cofatores sobre linhas ou colunas, ou seja:

$$det(A) = \sum_{i=1}^{n} A_{ij} * det(A(i|j)).(-1)^{i+j}$$
$$= \sum_{j=1}^{n} A_{ij} * det(A(i|j)).(-1)^{i+j}$$

onde $A(i|j)$ é a matriz de ordem $(n-1)\times(n-1)$ obtida de A removendo-se a i-ésima linha e a j-ésima coluna.

A adjunta clássica é definida por:

$$adj(A)_{ij} = det\big(A(j|i)\big).(-1)^{i+j}.$$

Daí segue que se $det(A) \neq 0$, então $adj(A)$ é a única matriz que satisfaz

$$adj(A) * A = A * adj(A) = I * det(A).$$

As duas afirmações seguintes servirão para deduzirmos o método de Samuelson que relaciona os polinômios característicos de A e M a fim de obtermos uma solução recursiva para $p(\lambda)$.

Lema 1.3 [Be].† $p(\lambda) = (a_{11}-\lambda) * det(M-\lambda * I) - R * adj(M-\lambda * I) * S.$

Dem. Façamos a expansão por cofatores de $det(A-\lambda * I)$ sobre a primeira linha e teremos

$$\begin{aligned} p(\lambda) = {} & (a_{11}-\lambda) * det(M-\lambda * I) + \\ & + a_{12} * det\big((A-\lambda * I)(1|2)\big).(-1)^{1+2} + a_{13} * det\big((A-\lambda * I)(1|3)\big).(-1)^{1+3} \\ & + \ldots + \\ & + a_{1n} * det\big((A-\lambda * I)(1|n)\big).(-1)^{1+n} \\ = {} & (a_{11}-\lambda) * det(M-\lambda * I) + R * \begin{bmatrix} det\big((A-\lambda * I)(1|2)\big).(-1)^{1+2} \\ det\big((A-\lambda * I)(1|3)\big).(-1)^{1+3} \\ \vdots \\ det\big((A-\lambda * I)(1|n)\big).(-1)^{1+n} \end{bmatrix}. \end{aligned}$$

Façamos agora a expansão por cofatores de $det\big((A-\lambda * I)(1|j)\big).(-1)^{1+j}$, $2 \leq j \leq n$

† No artigo original de Berkowitz encontramos um engano no enunciado deste lema. O operador, no centro da expressão, está $+$. Este engano teve como consequência outra troca de sinal na fórmula do método de Samuelson.

sobre a primeira coluna.

$$
\begin{aligned}
&det\big((A-\lambda * I)(1|j)\big).(-1)^{1+j} \\
&\qquad = a_{21} * det\big(((A-\lambda * I)(1|j))(1|1)\big).(-1)^{1+1}.(-1)^{1+j}+ \\
&\qquad\quad + a_{31} * det\big(((A-\lambda * I)(1|j))(2|1)\big).(-1)^{1+2}.(-1)^{1+j} + \cdots + \\
&\qquad\quad + a_{n1} * det\big(((A-\lambda * I)(1|j))(n-1|1)\big).(-1)^{1+n-1}.(-1)^{1+j} \\
&\qquad = a_{21} * det\big((M-\lambda * I)(1|j-1)\big).(-1)^{1+j}+ \\
&\qquad\quad + a_{31} * det\big((M-\lambda * I)(2|j-1)\big).(-1)^{2+j} + \cdots + \\
&\qquad\quad + a_{n1} * det\big((M-\lambda * I)(n-1|j-1)\big).(-1)^{n-1+j} \\
&\qquad = -\big[a_{21} * det\big((M-\lambda * I)(1|j-1)\big).(-1)^{1+(j-1)}+ \\
&\qquad\quad + a_{31} * det\big((M-\lambda * I)(2|j-1)\big).(-1)^{2+(j-1)} + \cdots + \\
&\qquad\quad + a_{n1} * det\big((M-\lambda * I)(n-1|j-1)\big).(-1)^{n-1+(j-1)}\big] \\
&\qquad = -\big[a_{21} * adj(M-\lambda * I)_{j-1,1}+ \\
&\qquad\quad + a_{31} * adj(M-\lambda * I)_{j-1,2} + \cdots + \\
&\qquad\quad + a_{n1} * adj(M-\lambda * I)_{j-1,n-1}\big] \\
&= -\big[adj(M-\lambda * I)_{j-1,1} \quad adj(M-\lambda * I)_{j-1,2} \quad \cdots \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad adj(M-\lambda * I)_{j-1,n-1}\big] * S.
\end{aligned}
$$

Portanto: $p(\lambda) = (a_{11}-\lambda) * det(M-\lambda * I) - R * adj(M-\lambda * I) * S.$ ∎

Lema 1.4 [Be]. $adj(M-\lambda * I) = -\sum_{k=2}^{n}\left(M^{k-2} * q_0 + \cdots + I * q_{k-2}\right) * \lambda^{n-k}.$

Dem. Multiplicando ambos os lados desta expressão por $(M-\lambda * I)$, o lado esquerdo ´ igual a $q(\lambda) * I$. Do lado direito temos:

$$
\begin{aligned}
-\sum_{k=2}^{n}(\cdots) * \lambda^{n-k} * (M-\lambda * I) = \\
&= -\sum_{k=2}^{n}\left(M^{k-1} * q_0 + \cdots + M * q_{k-2}\right) * \lambda^{n-k}+ \\
&= +\sum_{k=2}^{n}\left(M^{k-2} * q_0 + \cdots + I * q_{k-2}\right) * \lambda^{n-k+1}.
\end{aligned}
$$

Reordenando os índices do segundo termo temos:

$$
\begin{aligned}
-\sum_{k=2}^{n}(\cdots) * \lambda^{n-k} * (M-\lambda * I) = \\
&= -\sum_{k=2}^{n}\left(M^{k-1} * q_0 + \cdots + M * q_{k-2}\right) * \lambda^{n-k}+ \\
&= +\sum_{k=1}^{n-1}\left(M^{k-1} * q_0 + \cdots + I * q_{k-1}\right) * \lambda^{n-k}.
\end{aligned}
$$

Sabemos, pelo teorema de Cayley-Hamilton, que: $M^{n-1} * q_0 + \cdots + I * q_{n-1} = 0$. Somando então $M^{n-1} * q_0 + \cdots + I * q_{n-1}$ no segundo termo acima, temos:

$$
\begin{aligned}
-\sum_{k=2}^{n}(\cdots) * \lambda^{n-k} * (M - \lambda * I) = \\
&= -\sum_{k=2}^{n}\left(M^{k-1} * q_0 + \cdots + M * q_{k-2}\right) * \lambda^{n-k} + \\
&= +\sum_{k=1}^{n}\left(M^{k-1} * q_0 + \cdots + I * q_{k-1}\right) * \lambda^{n-k} \\
&= \left(q_{n-1} + q_{n-2} * \lambda + \cdots + q_0 \lambda^{n-1}\right) * I = q(\lambda) * I.
\end{aligned}
$$

Como $q(\lambda) \neq 0$ temos provado o lema 1.4. ∎

Substituindo Lema 1.4 no Lema 1.3 temos o método de Samuelson que relaciona os polinômios característicos de A e M:

$$
\begin{aligned}
p(\lambda) = (a_{11} - \lambda) * det(M - \lambda * I) \\
+ R * \left(\sum_{k=2}^{n}(M^{k-2} * q_0 + \cdots + I * q_{k-2}) * \lambda^{n-k}\right) * S.
\end{aligned}
$$

Observe que a igualdade acima nos dá uma solução recursiva para o polinômio característico de A. Tudo depende, então, de computarmos eficientemente o somatório que aparece no segundo membro. Caminharemos agora neste sentido.

Lema 1.5 [Be]. Sejam R, S e M matrizes $1 \times m$, $m \times 1$ e $m \times m$ respectivamente, para $m < n$. Seja $T = \{R * M^i * S\}_{i=0}^{m}$. Então T pode ser computado em tempo $O(\log^2 n)$ com $O(n^3)$ processadores.

Dem. Defina:

$$U = \{R * M^i\}_{i=0}^{\lceil\sqrt{n}\rceil} \quad e \quad V = \{M^{j*\lceil\sqrt{n}\rceil} * S\}_{j=0}^{\lceil\sqrt{n}\rceil}.$$

Então qualquer elemento de T pode ser determinado por um único produto de um elemento de U por um elemento em V. Isto ocorre porque qualquer expoente k de M em T pode ser escrito (unicamente) na forma $k = i + j * \lceil\sqrt{n}\rceil$, onde $i, j \leq \lceil\sqrt{n}\rceil$. Portanto, todo o conjunto T pode ser computado a partir de U e V por n produtos. Como cada produto de um elemento de U por um elemento de V pode ser computado em tempo $O(\log n)$ com $O(n)$ processadores, T pode ser computado em tempo $O(\log n)$ com $O(n^2)$ processadores.

U e V são computados da seguinte maneira: Apresentaremos primeiro a computação de U. A partir de M computamos M^2, depois M^4, M^8, e assim sucessivamente até $M^{2^{\lceil\log\sqrt{n}\rceil}}$. Isto pode ser feito em tempo $O(\log^2 n)$ com $O(n^3)$ processadores por $\lceil\log\sqrt{n}\rceil$ aplicações sequenciais de produto de matrizes. A seguir calculamos RM, RM^2 e de posse de $U' = \{RM^{2^j}, RM^{2^j-1}, \ldots, RM^{2^{j-1}+1}\}$ e $U'' = \{M^{2^j}, M^{2^{j-1}}\}$, obtemos $RM^{2^j+1}, \ldots,$

$RM^{2^{j+1}}$ multiplicando cada elemento de U' por cada elemento de U'' (fig. 1.10). Isto pode ser feito em tempo $O(\log^2 n)$ usando $O(n^3)$ procesadores. Desta maneira, U pode ser computado, iterativamente, a partir de M, M^2 e R em tempo $O(\log^2 n)$ com $O(n^3)$ processadores. V é computado de maneira análoga a partir de $M^{\lceil\sqrt{n}\rceil}$.

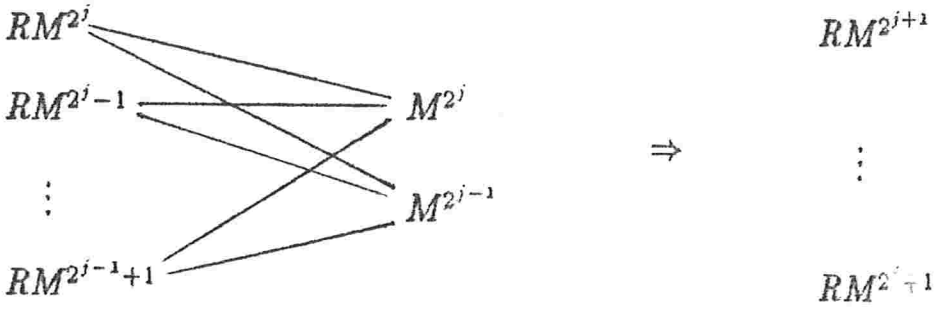


Fig. 1.10 - Cálculo de U

Definição [Be]. Uma matriz B de ordem $n \times m$ é dita *Super-Diagonal* se

$$B_{i,j} = B_{i-min\{i,j\}+1,j-min\{i,j\}+1}.$$

Ou seja, uma matriz $n \times m$ é Super-Diagonal se cada diagonal possui todos os seus elementos com o mesmo valor.

Uma matriz é *triangular inferior* se ela possui todos os elementos acima da diagonal principal iguais a zero ($B_{ij} = 0$ se $j > i$).

Lema 1.6 [Be]. Sejam A, B $n \times m$ e $m \times p$ matrizes triangular inferior e super-diagonal. Então $C = A * B$ é triangular inferior e super-diagonal e pode ser computada em tempo $O(\log n)$ com $O(n^2)$ processadores.

Dem. Se $i < j$, temos

$$C_{ij} = \sum_{k=1}^{m} A_{ik} * B_{kj} = \sum_{k=1}^{j-1} A_{ik} * B_{kj} + \sum_{k=j}^{m} A_{ik} * B_{kj} = 0.$$

Pois $B_{kj} = 0$ para $k = 1, \ldots, j-1$ e $A_{ik} = 0$ para $k = j, \ldots, m$ ($i < j \Rightarrow i < k$ para $k = j, \ldots, m$). Ou seja, C é triangular inferior.

Vamos mostrar agora que C é super-diagonal. Para isso precisamos mostrar que

$C_{ij} = C_{i-j+1,1}$ para $i \geq j$. Por um lado temos:

$$C_{ij} = \sum_{k=1}^{m} A_{ik} * B_{kj} = \sum_{k=j}^{m} A_{ik} * B_{kj} =$$

$$\stackrel{\text{def}}{=} \sum_{k=j}^{m} A_{i-min\{i,k\}+1,k-min\{i,k\}+1} * B_{k-min\{j,k\}+1,j-min\{j,k\}+1} =$$

$$= \sum_{k=j}^{i} A_{i-k+1,1} * B_{k-j+1,1} + \sum_{k=i+1}^{m} \underbrace{A_{1,k-i+1}}_{=0} * B_{k-j+1,1} =$$

$$= \sum_{k=j}^{i} A_{i-k+1,1} * B_{k-j+1,1}.$$

Por outro lado:

$$C_{i-j+1,1} = \sum_{k=1}^{m} A_{i-j+1,k} * B_{k,1} = \sum_{k=1}^{i-j+1} A_{i-j+1,k} * B_{k,1} =$$

$$= \sum_{k=1}^{i-j+1} A_{i-j+1-k+1,1} * B_{k,1}.$$

Reordenando os índices podemos ver que $C_{ij} = C_{i-j+1}$. Portanto C é super-diagonal.

Para computarmos C basta então determinarmos os n elementos C_{j1} , $1 \leq j \leq n$, já que C é triangular inferior e super-diagonal. Por n aplicações em paralelo do Teorema 1.1, Isto pode ser feito em tempo $O(\log n)$ com $O(n^2)$ processadores. ‖

Estamos agora em condições de apresentar o teorema principal deste capítulo.

TEOREMA 1.2 [Be]. Os coeficientes do polinômio característico $p(\lambda)$ podem ser computados em tempo $O(\log^2 n)$ com $O(n^4)$ processadores.

DEM. Para $1 \leq t \leq n$ seja C_t uma matriz de ordem $(n-t+2) \times (n-t+1)$, triangular inferior e super-diagonal definida por:

$$[C_t]_{i,1} = \begin{cases} -1 & \text{se } i = 1 \\ a_{t,t} & \text{se } i = 2 \\ R_t * M_t^{i-3} * S_t & \text{se } i > 2 \end{cases}$$

Para $t = 1, \ldots, n-1$, e $C_n = \begin{bmatrix} -1 \\ a_{nn} \end{bmatrix}$.

Pelo método de Samuelson temos uma relação linear entre os coeficientes de $p(\lambda)$ e os coeficientes de $q(\lambda)$ que pode ser expressa por:

$$(p_0, p_1, \ldots, p_n)^t = C_1 * (q_0, \ldots, q_{n-1})^t.$$

Onde $(\cdots)^t$ é a matriz transposta. Aplicando isto recursivamente, temos

$$(p_0, p_1, \ldots, p_n)^t = \prod_{i=1}^{n} C_i.$$

As matrizes $\{C_i\}_{i=1}^n$ podem ser computadas fazendo n aplicações, em paralelo, do Lema 1.5 em tempo $O(\log^2 n)$ com $O(n^4)$ processadores. Daí pelo Lema 1.6 podemos computar $\prod_{i=1}^n$ usando uma árvore binária balanceada de multiplicação de matrizes em tempo $O(\log^2 n)$ com $O(n^3)$ processadores.]

Temos então calculado o polinômio característico da matriz A em tempo $O(\log^2 n)$ com $O(n^4)$ processadores, e temos, como cosequência, calculado $det(A) = p(0)$ com a mesma complexidade.

Exemplo. Suponha que queremos calcular o determinante da matriz

$$A = \begin{bmatrix} -3 & 7 & -5 & 2 \\ 4 & 6 & 0 & 0 \\ 5 & -2 & 4 & -1 \\ 0 & 0 & 1 & 3 \end{bmatrix}$$

pelo algoritmo acima.

O que vamos fazer, então, é calcular os coeficientes do polinômio característico $p(\lambda) = det(A - \lambda * I)$.

A partir de A temos as matrizes

$$M_1 = \begin{bmatrix} 6 & 0 & 0 \\ -2 & 4 & -1 \\ 0 & 1 & 3 \end{bmatrix} \quad , \quad R_1 = \begin{bmatrix} 7 & -5 & 2 \end{bmatrix} \quad , \quad S_1 = \begin{bmatrix} 4 \\ 5 \\ 0 \end{bmatrix}$$

$$M_2 = \begin{bmatrix} 4 & -1 \\ 1 & 3 \end{bmatrix} \quad , \quad R_2 = \begin{bmatrix} 0 & 0 \end{bmatrix} \quad , \quad S_2 = \begin{bmatrix} -2 \\ 0 \end{bmatrix}$$

$$M_3 = \begin{bmatrix} 3 \end{bmatrix} \quad , \quad R_3 = \begin{bmatrix} -1 \end{bmatrix} \quad , \quad S_3 = \begin{bmatrix} 1 \end{bmatrix}.$$

As matrizes C_t são:

$$C_1 = \begin{bmatrix} -1 & 0 & 0 & 0 \\ a_{11} & -1 & 0 & 0 \\ R_1S_1 & a_{11} & -1 & 0 \\ R_1M_1S_1 & R_1S_1 & a_{11} & -1 \\ R_1M_1^2S_1 & R_1M_1S_1 & R_1S_1 & a_{11} \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ -3 & -1 & 0 & 0 \\ 3 & -3 & -1 & 0 \\ 118 & 3 & -3 & -1 \\ 1087 & 118 & 3 & -3 \end{bmatrix}$$

$$C_2 = \begin{bmatrix} -1 & 0 & 0 \\ a_{22} & -1 & 0 \\ R_2S_2 & a_{22} & -1 \\ R_2M_2S_2 & R_2S_2 & a_{22} \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 6 & -1 & 0 \\ 0 & 6 & -1 \\ 0 & 0 & 6 \end{bmatrix}$$

$$C_3 = \begin{bmatrix} -1 & 0 \\ a_{33} & -1 \\ R_3S_3 & a_{33} \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 4 & -1 \\ -1 & 4 \end{bmatrix} \quad e \quad C_4 = \begin{bmatrix} -1 \\ 3 \end{bmatrix}.$$

Fazendo o produto $C_1.C_2.C_3.C_4$ temos:

$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ -3 & -1 & 0 & 0 \\ 3 & -3 & -1 & 0 \\ 118 & 3 & -3 & -1 \\ 1087 & 118 & 3 & -3 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 & 0 \\ 6 & -1 & 0 \\ 0 & 6 & -1 \\ 0 & 0 & 6 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 \\ 4 & -1 \\ -1 & 4 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ -3 & 1 & 0 \\ -21 & -3 & 1 \\ -100 & -21 & -3 \\ -379 & -100 & -21 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ -7 \\ 13 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ -10 \\ 13 \\ 8 \\ 48 \end{bmatrix}$$

Portanto: $p(\lambda) = \lambda^4 - 10\lambda^3 + 13\lambda^2 + 8\lambda + 48$ e $det(A) = p(0) = 48$.

1.7 - A CLASSE DET [Co]

Seja *intdet* o problema de calcular $det(A)$ dada uma matriz A de ordem $n \times n$ com elementos inteiros, e seja *potmat* o problema de calcular as potências $A^1, A^2, \ldots, A^n$, dada uma matriz A. Como o problema da multiplicação de duas matrizes inteiras está em NC^1, podemos ver que *potmat* está em NC^2.

O que faremos nesta seção é tratar de alguns problemas $\mathcal{R}$ tais que $\mathcal{R} \preceq intdet$ (relembre que $\preceq$ significa "é NC^1 redutível para"). Mais formalmente,

Definição [Co]. $DET = \{intdet\}^* = \{f / f \preceq intdet\}$.

Por exemplo, o problema "produtos iterados de inteiros" (dado n inteiros $a_1, a_2, \ldots, a_n$ calcular o produto $a_1 a_2 \ldots a_n$) está em DET, pois, basta computarmos o determinante da matriz com $a_1, a_2, \ldots, a_n$ na diagonal principal e zero nas demais posições.

Uma classe abrangente de problemas inclusos em DET é a classe NL^*, onde NL é a classe dos conjuntos aceitos por uma Máquina de Turing não-determinística em espaço $O(\log n)$.

Proposição 1.3 [Co]. Os seguintes problemas são completos para DET:

(1) *intdet*

(2) *potmat*

(3) *itprodmat* (produtos iterados de matrizes)

(4) *invmat* (matriz inversa)

entrada: matriz inteira A de ordem $n \times n$

saída: A^{-1} na forma $< adj(A), det(A) >$.

Dem.

(a) *intdet* $\preceq$ *potmat* : (seção 1.5, algoritmo de Berkowitz).

(b) *potmat* $\preceq$ *invmat* : Seja N a matriz de ordem $n^2 \times n^2$ consistindo de $n \times n$ blocos dispostos do seguinte modo: Os $n-1$ blocos acima da diagonal principal contém a matriz A e os demais blocos são nulos (fig. abaixo).

$$N = \begin{bmatrix} 0 & A & 0 & \dots & 0 \\ 0 & 0 & A & \dots & 0 \\ & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \dots & A \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

Então, $N^n = 0$ e $(I-N)^{-1} = I + N + N^2 + \cdots + N^{n-1} =$

$$\begin{bmatrix} I & A & A^2 & \dots & A^{n-1} \\ 0 & I & A & \dots & A^{n-2} \\ & \vdots & & \ddots & \vdots \\ & & & \dots & A \\ 0 & 0 & 0 & \dots & I \end{bmatrix}$$

(c) *invmat* $\preceq$ *intdet* :

$$A^{-1} = \frac{adj(A)}{det(A)},$$

onde $adj(A)_{ij} = det(A(j|i)).(-1)^{i+j}$.

(d) *potmat* $\preceq$ *itprodmat* : Óbvio.

(e) *itprodmat* $\preceq$ *potmat* : Seja $A_1, \dots, A_n$ matrizes de ordem $n \times n$. Seja B uma matriz de ordem $(n^2+n) \times (n^2+n)$ onde todos os blocos de tamanho $n \times n$ são nulos com excessão daqueles que ficam acima da diagonal principal, aí estarão as matrizes $A_1, \dots, A_n$.

$$B = \begin{bmatrix} 0 & A_1 & 0 & \dots & 0 \\ 0 & 0 & A_2 & \dots & 0 \\ & \vdots & & \ddots & \vdots \\ & & & & A_n \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

Então B^n conterá o produto $A_1 A_2 \dots A_n$ no canto superior direito.

1.8 - COMENTÁRIOS FINAIS

Na seção 1.3 definimos a classe NC^k de problemas que podem ser resolvidos por uma família de circuitos booleanos uniformes em tempo $O(\log^k n)$ usando um número polinomial de processadores. Uma outra forma de classificar problemas segundo suas complexidades

paralelas é considerar a classe RNC (Random NC) [Co]. A importância desta classificação é que ela abrange problemas que ainda não se sabe se pertencem à classe NC, mas que podem ser resolvidos por circuitos em tempo polinomial em log e número polinomial de processadores, com probabilidade de erro muito pequena.

Mais precisamente, um problema R está em RNC^k se ele pode ser resolvido por uma função f computada por uma família $< \alpha_n >$ de circuitos probabilísticos uniformes, com probabilidade de erro de cada bit de no máximo $1/4$, onde $d(\alpha_n) = O(\log^k n)$ e $s(\alpha_n) = n^{O(1)}$. Um *circuito probabilístico* é um circuito booleano com entrada x e uma sequência y de dígitos binários aleatórios. A probabilidade de um bit de saída v ser 1 é definida como sendo a fração das sequências y tal que o valor de v é 1 quando a entrada de α_n é (x, y). Se cada bit de $f(x)$ é computado corretamente com probabilidade de pelo menos $3/4$, então podemos arranjar vários circuitos em paralelo computando o mesmo bit, e o voto da maioria pode ser tomado como um valor confiável. Portanto, se R está em RNC^k, então para cada l, uma outra família de circuitos probabilísticos uniformes $< \beta_n >$ de tamanho polinomial e profundidade $O(\log^k n)$ computa todos os bits da função f que resolve R corretamente com a probabilidade de erro de no máximo 2^{-n^l}.

NC está contida em RNC, mas nada se sabe a respeito da inclusão contrária [Co].

Alguns problemas em RNC^2 são: Calcular o posto de uma matriz sobre um corpo finito, calcular o tamanho do emparelhamento máximo em um grafo bipartido, e calcular o valor do fluxo máximo em um grafo com capacidade nas arestas expressas em notação unária.

Vimos que as operações básicas sobre anéis podem ser implementadas em NC^1. Usamos, porém, uma representação redundante para implementar a soma em NC^0. Se tentarmos uma representação não redundante para os inteiros, Winograd mostrou que é impossível uma implementação para a soma em NC^0 [BCP]. Observe que, mesmo usando uma representação redundante, se também for possível uma implementação para a multiplicação em NC^0, essa melhora será refletida em diversos problemas; Por exemplo, Produto de Matrizes passa a ser computado em NC^0 e o Determinante, como consequência, passa a ser computado em NC^1. Para o anel dos inteiros, é possível conseguirmos uma representação para a qual a multiplicação possa ser implementada em NC^0, mas a soma passa a ter um custo extremamente alto. Por exemplo, representando os inteiros como produto de fatores primos, a multiplicação é reduzida a soma dos expoentes. Com relação às operações básicas sobre anéis, alguns problemas em aberto são: assumindo uma representação sucinta e uniforme para a qual a multiplicação possa ser implementada em NC^0, qual é a melhor implementação para a soma de inteiros? Em particular, podemos provar que soma e multiplicação não podem ambas serem implementadas em NC^0?

Nos lemas 1.1 e 1.2 mostramos que somas iteradas e produtos iterados de matrizes têm implementações em NC^1 e NC^2 respectivamente. Na verdade, usando a mesma idéia da árvore binária, podemos provar que "se A é um anel Bem-Dotado e $f : A \times A \mapsto A$ é um mapeamento associativo com elemento neutro e com uma implementação em NC^k para algum k em N, então iterada f tem uma implementação em NC^{k+1}" [BCP].

Na seção 1.5 vimos que o determinante de uma matriz, cujos elementos pertencem a um anel Bem-Dotado A, pode ser computado em NC^2. Este resultado também é válido

se considerarmos o corpo de frações de $\mathcal{A}$ [BCP].

Na tabela a seguir apresentamos as complexidades sequenciais e paralelas dos algoritmos estudados neste capítulo, bem como seus respectivos aumentos, custos e eficiências, conforme as definições das páginas 3 e 4.

	Soma	Produto	Prodmat	Determinante
Sequencial	$O(n)$	$O(n^2)$	$O(n^{\log_2 7})$	$O(n^{\log_2 7})$
Paralelo	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log^2 n)$
Aumento	$O(n)$	$O(n^2/\log n)$	$O(n^{\log_2 7}/\log n)$	$O(n^{\log_2 7}/\log^2 n)$
Custo	$O(n)$	$O(n^2 \log n)$	$O(n^3 \log n)$	$O(n^4 \log^2 n)$
Eficiência	$O(1)$	$O(1/\log n)$	$O(1/\log n)$	$O(1/n \log^2 n)$

Tab. 1.1 - Complexidades dos algoritmos do Cap. 1.

Observe que segundo as definições das páginas 3 e 4, o ideal é que tenhamos valores da eficiência próximos de uma constante (o que só ocorre no problema da soma), e os valores do custo próximos do valor da complexidade sequencial dos algoritmos.

Capítulo II

Ordenação

2.1 - INTRODUÇÃO

Ordenação é definido como o problema de rearranjar uma sequência de valores em ordem crescente ou decrescente. Programas computacionais tais como compiladores e editores utilizam a ordenação de tabelas de símbolos e listas para aumentar a eficiência de buscas e inserções de novos elementos nestes conjuntos. Devido à sua importância tanto prática quanto teórica, ordenação é, provavelmente, o problema mais estudado em ciência da computação. Primeiramente, algoritmos sequenciais foram investigados, e muitos são conhecidos dentre os que ordenam n elementos em $O(n \log n)$ comparações no pior caso, que é o limite inferior para o problema [Kn]. Depois, com o advento do processamento paralelo, algoritmos paralelos para ordenação têm sido uma área de pesquisa muito ativa.

Neste capítulo trataremos de estudar algoritmos paralelos para ordenação. Mais precisamente, apresentaremos os algoritmos de ordenação por intercalação par-ímpar e ordenação por intercalação bitônica, de autoria de Batcher [Ba], e o algoritmo de ordenação usando embaralhamento perfeito, de autoria de Stone [St]. Estes algoritmos não são os mais rápidos dentre os existentes, nós os escolhemos devido a originalidade da construção e a viabilidade de implementação de cada um deles.

2.2 - ORDENAÇÃO POR INTERCALAÇÃO PAR-ÍMPAR

Os circuitos descritos nesta seção e na seguinte são compostos por uma coleção de processadores com as seguintes características:

(1) Cada processador tem duas linhas de entrada e duas de saída;

(2) Cada processador pode comparar os dois elementos de entrada, colocar o menor dos dois em uma linha de saída denotada por L, e o maior na outra denotada por H. Se as duas entradas possuem o mesmo valor, suas posições são conservadas, ou seja, o elemento que entrou pela linha de entrada superior (inferior) é colocado na linha de saída superior (inferior).

Cada processador deste tipo é conhecido como elemento de comparação ou comparador (fig. 2.1).

Comparadores são usados para construção de circuitos de intercalação do seguinte modo: Assuma que queremos intercalar duas sequências ordenadas $\{a_1, a_2, \ldots, a_n\}$ e $\{b_1, b_2, \ldots, b_n\}$, para formar a sequência ordenada $\{c_1, c_2, \ldots, c_{2n}\}$, onde n é uma potência de 2. Se $n = 1$, então, obviamente, um comparador é suficiente para a intercalação. Se $n = 2$, então é possível verificar exaustivamente que o circuito de intercalação 2×2 da figura 2.2 intercala corretamente duas sequências ordenadas, cada uma com dois elementos. No caso geral, os elementos com índice ímpar, ou seja, $\{a_1, a_3, a_5, \ldots\}$ e $\{b_1, b_3, b_5, \ldots\}$, são intercalados usando um circuito de intercalação $(n/2) \times (n/2)$ para produzir a sequência $\{d_1, d_2, d_3, \ldots\}$. Simultaneamente, os elementos com índice par das duas sequências, ou seja, $\{a_2, a_4, a_6, \ldots\}$ e $\{b_2, b_4, b_6, \ldots\}$, são intercalados para produzir $\{e_1, e_2, e_3, \ldots\}$. A sequência final $\{c_1, c_2, \ldots, c_{2n}\}$ é obtida fazendo $c_1 = d_1$, $c_{2i} = min\{d_{i+1}, e_i\}$ e $c_{2i+1} = max\{d_{i+1}, e_i\}$ para $i = 1, 2, \ldots, n-1$, e $c_{2n} = e_n$.

Um circuito de intercalação $n \times n$ é ilustrado na figura 2.3. Note que cada um dos dois circuitos de intercalação $(n/2) \times (n/2)$ é construído aplicando a mesma regra recursivamente, ou seja, usando dois circuitos de comparação $(n/4) \times (n/4)$ seguido por $(n/2) - 1$ comparadores. A certificação deste método, conhecido como intercalação par-ímpar, é estabelecida pelo seguinte teorema:

Teorema 2.1 [Ba]. Sejam $\{a_1, a_2, \ldots, a_n\}$ e $\{b_1, b_2, \ldots, b_n\}$ duas sequências ordenadas. intercalação par-ímpar intercala corretamente estas duas sequências produzindo $\{c_1, c_2, \ldots c_{2n}\}$ por

(1) Intercalando os elementos de índice ímpar e de índice par das duas sequências para produzir $\{d_1, d_2, d_3, \ldots\}$ e $\{e_1, e_2, e_3, \ldots\}$, respectivamente;

(2) Fazendo $c_{2i} = min\{d_{i+1}, e_i\}$ e $c_{2i+1} = max\{d_{i+1}, e_i\}$ para $i = 1, 2, \ldots, n-1$;

(3) E, finalmente, fazendo $c_1 = d_1$ e $c_{2n} = e_n$.

Dem. Considere a subsequência $\{d_1, d_2, \ldots, d_{i+1}\}$ para algum i. Se k elementos desta subsequência pertencem a $\{a_1, a_3, a_5, \ldots\}$, então $i + 1 - k$ elementos pertencem a $\{b_1, b_3, b_5, \ldots\}$. Portanto, $2k - 1$ elementos de $\{a_1, a_2, a_3, \ldots\}$ e $2i + 1 - 2k$ elementos de $\{b_1, b_2, b_3, \ldots\}$ são menores ou iguais a d_{i+1}. Consequentemente,

$$d_{i+1} \geq c_{2i}. \tag{2.1}$$

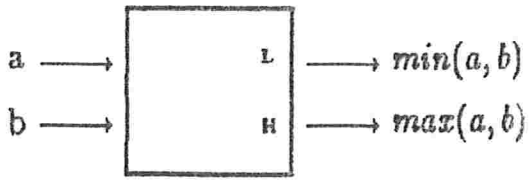


Fig. 2.1 - elemento de comparação

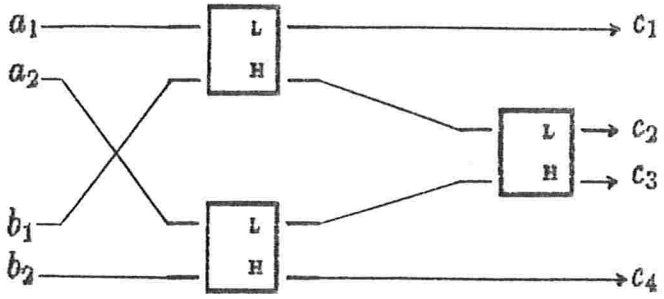


Fig. 2.2 - circuito de intercalção 2×2.

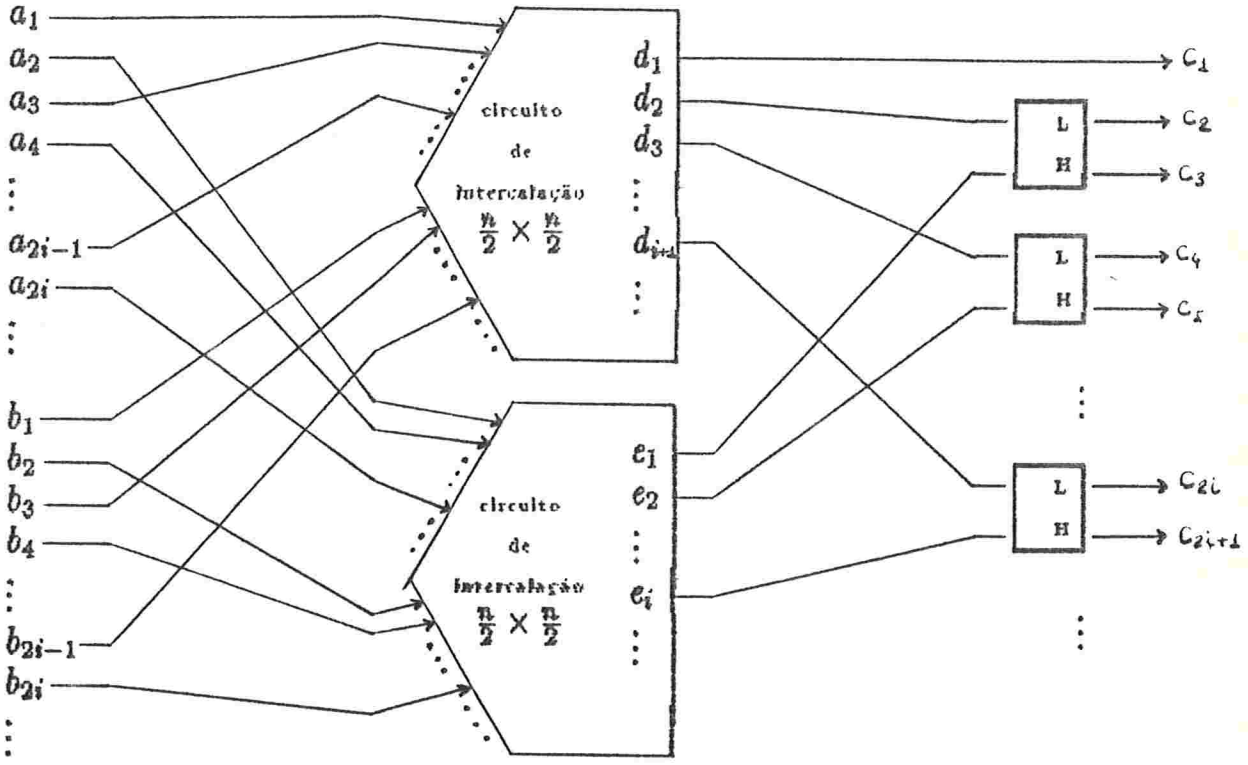


Fig. 2.3 - circuito de intercalação $n \times n$.

Pela mesma razão,

$$e_i \geq c_{2i}. \tag{2.2}$$

Agora assuma que k elementos de $\{c_1, c_2, \ldots, c_{2i+1}\}$ pertencem a $\{a_1, a_2, a_3, \ldots\}$ e que $2i+1-k$ elementos pertencem a $\{b_1, b_2, b_3, \ldots\}$. Concluímos, então, que c_{2i+1} é maior ou igual a

k elementos de $\{a_1, a_2, a_3, \ldots\}$,

$k/2$ elementos de $\{a_1, a_3, a_5, \ldots\}$ (ou $(k+1)/2$ se k for ímpar),

$2i+1-k$ elementos de $\{b_1, b_2, b_3, \ldots\}$, e

$i+1-(k/2)$ elementos de $\{b_1, b_3, b_5, \ldots\}$ (ou $(2i+1-k)/2$ se k for ímpar).

Ou seja,

$$c_{2i+1} \geq d_{i+1}. \tag{2.3}$$

Pela mesma razão

$$c_{2i+1} \geq e_i. \tag{2.4}$$

Como $c_1 \leq c_2 \leq c_3 \leq \cdots$, as desigualdades (2.1) a (2.4) implicam que

$$c_{2i} = min\{d_{i+1}, e_i\} \qquad e \qquad c_{2i+1} = max\{d_{i+1}, e_i\}.$$

Finalmente, como

$$d_1 = min(a_1, b_1) \qquad e \qquad e_n = max(a_n, b_n),$$

a prova está terminada. ∎

Tendo estabelecido que é possível intercalar duas sequências ordenadas usando um circuito de intercalação, é imediato que um circuito para ordenação pode ser construído baseado no mesmo conceito. A idéia é simplesmente tomar uma sequência não ordenada S com n elementos, e, usando $n/2$ comparadores, cria-se $n/2$ sequências ordenadas, cada uma com 2 elementos. Pares destas sequências são agora intercalados usando circuitos de intercalação 2×2 para formar sequências ordenadas com 4 elementos. Pares destas sequências são agora intercalados usando circuitos de intercalação 4×4 para formar sequências ordenadas com 8 elementos. E o processo continua até que uma sequência ordenada com n elementos seja obtida. Este algoritmo é conhecido como algoritmo par-ímpar para ordenação.

Exemplo. Um circuito para ordenar a sequência $S = \{8, 7, 6, 5, 4, 3, 2, 1\}$ usando o algoritmo de intercalação par-ímpar é ilustrado na figura 2.4.

O número total de comparadores e o tempo de execução em paralelo para ordenar uma sequência com n elementos, onde $n = 2^m$, para algum inteiro positivo m, usando ordenação par-ímpar é obtido como segue. Como o número de elementos das sequências dobra após cada estágio, existe $\log n$, (isto é, m) estágios ao todo. se denotarmos por $q(2^i)$ o número de comparadores utilizados no i-ésimo estágio para intercalar duas sequências ordenadas com 2^{i-1} elementos cada uma, então teremos a recorrência

$$\begin{aligned} q(2) &= 1 && para\ i = 1 \\ q(2^i) &= 2q(2^{i-1}) + 2^{i-1} - 1 && para\ i \geq 1. \end{aligned}$$

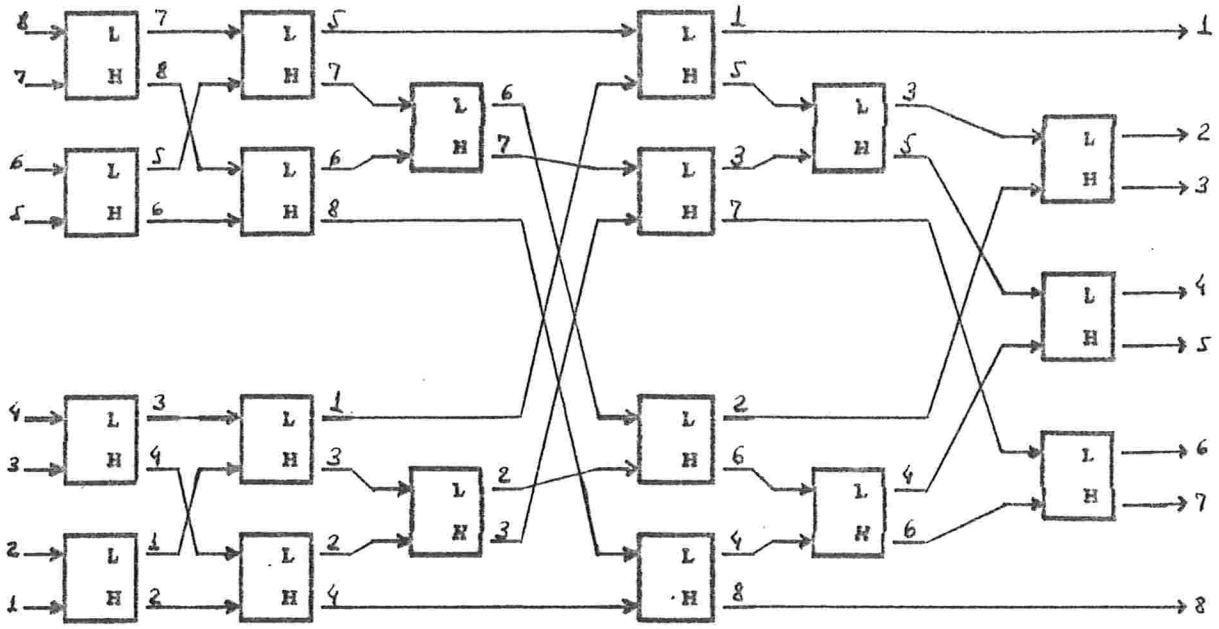


Fig. 2.4 - circuito para ordenar a sequência $S = \{8, 7, 6, 5, 4, 3, 2, 1\}$.

Resolvendo esta recorrência temos

$$\begin{aligned}
q(2^i) &= 2(2q(2^{i-2}) + 2^{i-2} - 1) + 2^{i-1} - 1 \\
&= 2^2 q(2^{i-2}) + 2^{i-1} + 2^{i-1} - 2 - 1 \\
&= 2^2(2q(2^{i-3}) + 2^{i-3} - 1) + 2^{i-1} + 2^{i-1} - 2 - 1 \\
&= 2^3 q(2^{i-3}) + 2^{i-1} + 2^{i-1} + 2^{i-1} - 2^2 - 2 - 1 \\
&= \cdots = \\
&= 2^j q(2^{i-j}) + j2^{i-1} - (2^{j-1} + 2^{j-2} + \cdots + 2^0)
\end{aligned}$$

Quando $j = i - 1$, temos

$$\begin{aligned}
q(2^i) &= 2^{i-1}.1 + (i-1)2^{i-1} - (2^{i-1} - 1) \\
&= (i-1)2^{i-1} + 1.
\end{aligned}$$

Portanto, o número total de comparadores utilizados para ordenar uma sequência com 2^m elementos é

$$\begin{aligned}
\sum_{i=1}^{m} 2^{m-i} q(2^i) &= \sum_{i=1}^{m} 2^{m-i}((i-1)2^{i-1} + 1) \\
&= 2^{m-1} \sum_{i=1}^{m} (i-1) + \sum_{i=1}^{m} 2^{m-i} \\
&= 2^{m-1} \frac{m(m-1)}{2} + 2^m - 1 \\
&= 2^{m-2}(m^2 - m + 4) - 1.
\end{aligned}$$

Portanto,

$$\begin{aligned} p(n) &= O(m^2 2^{m-2}) \\ &= O(n \log^3 n). \end{aligned}$$

Se denotarmos $s(2^i)$ o número máximo de comparações, em paralelo, executadas no i-ésimo estágio para intercalar duas sequências ordenadas com 2^{i-1} elementos cada uma, então teremos a recorrência

$$\begin{aligned} s(2) &= 1 && \textit{para } i = 1, \\ s(2^i) &= s(2^{i-1}) + 1 && \textit{para } i > 1, \end{aligned}$$

cuja solução é

$$s(2^i) = i.$$

Portanto, o mais longo caminho em um circuito para ordenar uma sequência com 2^m elementos consiste de

$$\sum_{i=1}^{m} s(2^i) = \sum_{i=1}^{m} i = \frac{m(m+1)}{2}$$

passos. Portanto,

$$t(n) = O(m^2) = O(\log^2 n).$$

Consequentemente,

$$c(n) = t(n) \times p(n) = O(n \log^4 n).$$

2.3 - ORDENAÇÃO POR INTERCALAÇÃO BITÔNICA

Nesta seção introduziremos um segundo método para construção de circuitos de ordenação usando comparadores. Como no circuito anterior, este também é baseado na idéia de intercalações sucessivas. A definição e o teorema seguinte contém as informações necessárias para o entendimento deste novo algoritmo.

Definição 2.1 [Ba]. Uma sequência $\{a_1, a_2, \ldots, a_{2n}\}$ é dita ser *bitônica* se

(i) Existe um inteiro j, $1 \leq j \leq 2n$, tal que

$$a_1 \leq a_2 \leq \cdots \leq a_j \geq a_{j+1} \geq \cdots \geq a_{2n},$$

ou

(ii) A sequência não satisfaz a condição (i) inicialmente, mas pode ser deslocada ciclicamente até a condição (i) ser satisfeita.

Por exemplo, $\{1, 3, 5, 6, 7, 9, 4, 2\}$ *é uma sequência bitônica, pois satisfaz a condição* (i). A sequência $\{7, 8, 6, 4, 3, 1, 2, 5\}$, que não satisfaz a condição (i), também é bitônica, pois ela pode ser deslocada ciclicamente para obter $\{2, 5, 7, 8, 6, 4, 3, 1\}$.

Teorema 2.2 [Ba]. Seja $\{a_1, a_2, \ldots, a_{2n}\}$ uma sequência bitônica. Se $d_i = min(a_i, a_{n+i})$ e $e_i = max(a_i, a_{n+i})$ para $1 \leq i \leq n$, então

(I) $\{d_1, d_2, \ldots, d_n\}$ e $\{e_1, e_2, \ldots, e_n\}$ são sequências bitônicas e

(II) $max(d_1, d_2, \ldots, d_n) \leq min(e_1, e_2, \ldots, e_n)$.

Dem. Como um deslocamento cíclico de $\{a_1, a_2, \ldots, a_{2n}\}$ resulta em um deslocamento cíclico de $\{d_1, d_2, \ldots, d_n\}$ e $\{e_1, e_2, \ldots, e_n\}$, e isto não afeta as propriedades (I) e (II), é suficiente provar o teorema para o caso em que

$$a_1 \leq a_2 \leq \cdots \leq a_j \geq a_{j+1} \geq \cdots \geq a_{2n}$$

ocorre para algum $1 \leq j \leq 2n$.

Além disso, como a sequência reversa $\{a_{2n}, a_{2n-1}, \ldots, a_1\}$ é também bitônica e as propriedades (I) e (II) não são afetadas pela reversão, assumimos, sem perda de generalidade, que $n < j \leq 2n$, e vamos provar o teorema nestas condições.

Caso 1: Se $a_n \leq a_{2n}$, então $a_i \leq a_{n+i}$. Consequentemente $d_i = a_i$ e $e_i = a_{n+i}$ para $1 \leq i \leq n$, e as propriedades (I) e (II) são satisfeitas.

Caso 2: Se $a_n > a_{2n}$, então como $a_{j-n} \leq a_j$, um índice k, $j \leq k \leq 2n$, tal que

$$a_{k-n} \leq a_k \qquad e \qquad a_{k-n+1} > a_{k+1} \tag{4.5}$$

pode ser encontrado.

Para ver isto, tome $k = j$, que temos satisfeita a primeira desigualdade. Se a segunda desigualdade não é satisfeita, ou seja, $a_{k-n+1} \leq a_{k+1}$, tome $k = j + 1$ e repita o processo (observe que a primeira desigualdade continua satisfeita para $k = j + 1$). Se nenhum valor de $j \leq k < 2n - 1$ satisfaz as duas desigualdades, então devemos ter $a_{n-1} \leq a_{2n-1}$. Mas então $k = 2n - 1$ satisfaz as duas desigualdades, pois $a_n > a_{2n}$. Portanto, um índice nas condicões de (4.5) pode sempre ser encontrado.

Daí segue que (fig. 2.5)

$$d_i = a_i \qquad e \qquad e_i = a_{n+i} \qquad para\ 1 \leq i \leq k - n$$

e

$$d_i = a_{n+i} \qquad e \qquad e_i = a_i \qquad para\ k - n < i \leq n.$$

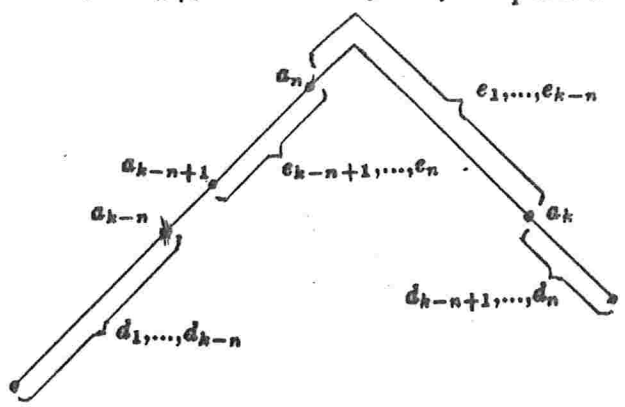


Fig. 2.5 - Valores de $\{d_1, d_2, \ldots, d_n\}$ e $\{e_1, e_2, \ldots, e_n\}$ a partir de $\{a_1, a_2, \ldots, a_{2n}\}$.

Para provar (I), note que

$$a_1 \leq \cdots \leq a_{k-n} \geq a_{k+1} \geq \cdots \geq a_{2n},$$

o que prova que $\{d_1, d_2, \ldots, d_n\}$ é bitônica. Também ocorre que

$$\begin{array}{ll} e_i \leq e_{i+1} & para\ k-n < i < n, \\ e_n \leq e_1, & \\ e_i \leq e_{i+1} & para\ 1 \leq i < j-n, \\ e_i \geq e_{i+1} & para\ j-n \leq i < k-n, \end{array}$$

o que prova que $\{e_1, e_2, \ldots, e_n\}$ também é bitônica. Isto completa a prova de (I). Para provar (II), note que

$$max(d_1, d_2, \ldots, d_n) = max(d_{k-n}, d_{k-n+1}) = max(a_{k-n}, a_{k+1}),$$

e

$$min(e_1, e_2, \ldots, e_n) = min(e_{k-n}, e_{k-n+1}) = max(a_k, a_{k-n+1}).$$

Como

$$a_k \geq a_{k+1}, \quad a_k \geq a_{k-n}, \quad a_{k-n+1} \geq a_{k-n}, \quad e a_{k-n+1} \geq a_{k+1},$$

temos

$$max(a_{k-n}, a_{k+1}) \leq max(a_k, a_{k-n+1}). \qquad \rceil$$

Teorema 4.2 implica que nós podemos ordenar uma sequência bitônica $\{a_1, a_2, \ldots, a_{2n}\}$ em ordem crescente, do seguinte modo:

(1) Usando n comparadores, as duas subsequências

$$min(a_1, a_{n+1}),\ min(a_2, a_{n+2}), \ldots, min(a_n, a_{2n})$$

e

$$max(a_1, a_{n+1}),\ max(a_2, a_{n+2}), \ldots, max(a_n, a_{2n})$$

são criadas.

(2) Cada uma dessas duas subsequências, sendo bitônicas, podem ser ordenadas recursivamente pelo mesmo processo. Como nenhum elemento da primeira subsequência é maior que qualquer elemento da segunda, a primeira subsequência produzirá os n primeiros elementos da sequência ordenada e a segunda produzirá os n últimos. O circuito que executa esta ordenação está ilustrado na figura 2.6.

Um intercalador bitônico para uma sequência bitônica com 2 elementos é, obviamente, um simples comparador. Exemplos de circuitos de intercalação bitônica para sequências com 4 e 8 elementos estão ilustrados nas figuras 2.7 e 2.8, respectivamente.

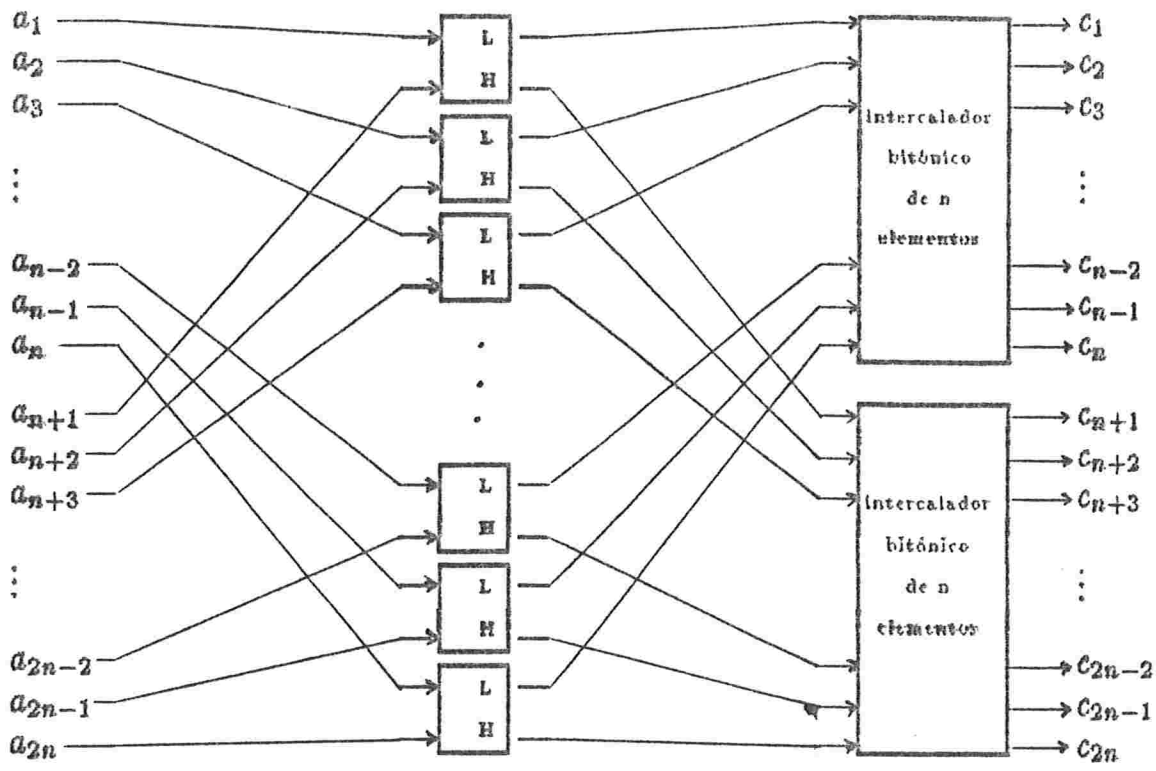


Fig. 2.6 - Intercalador Bitônico.

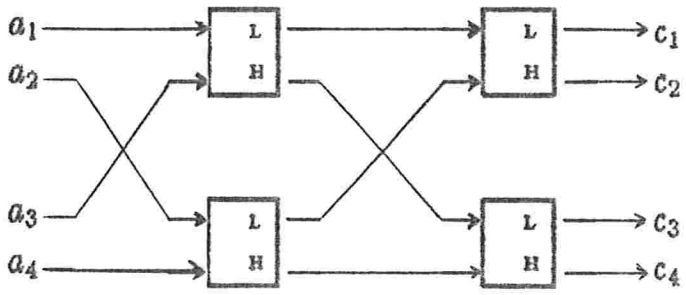


Fig. 2.7 - Intercalador bitônico para uma sequência com 4 elementos.

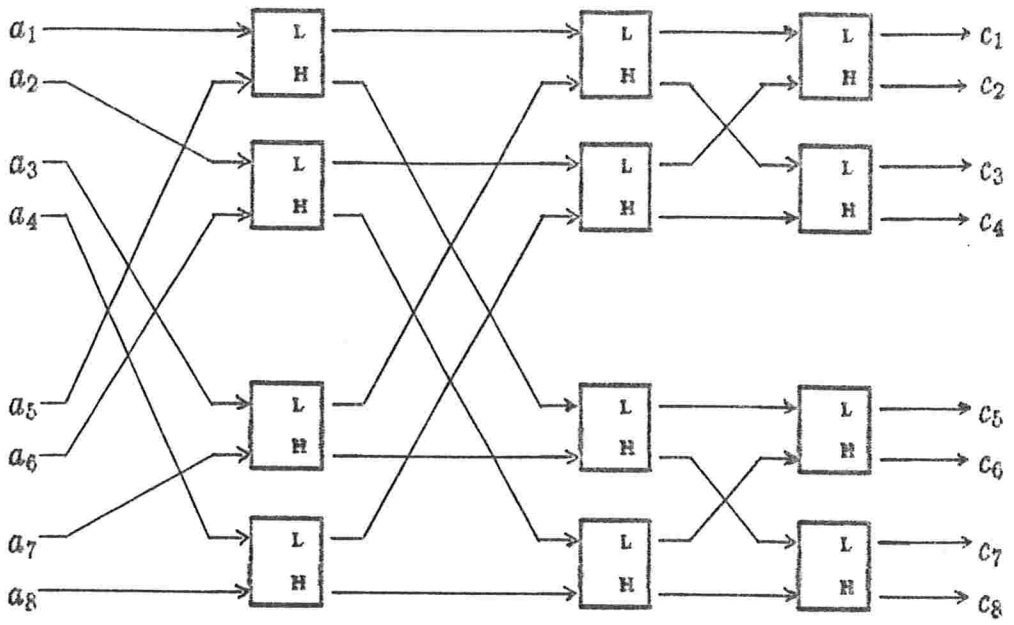


Fig. 2.8 - Intercalador bitônico para uma sequência com 8 elementos.

Se desejamos ordenar uma sequência S com n elementos, então subsequências bitônicas de S são ordenadas e combinadas para formar subsequências bitônicas maiores, até que uma sequência bitônica com n elementos é obtida, a qual é finalmente ordenada. As subsequências bitônicas são ordenadas usando intercaladores bitônicos como descrito acima. Este algoritmo é conhecido como ordenação bitônica.

Exemplo. Um circuito para ordenar a sequência $\{4, 8, 1, 3, 2, 7, 5, 6\}$, usando ordenação bitônica é ilustrado na figura 2.9. Note que:

(1) Para produzir a parte decrescente da sequência bitônica, as linhas de saída de alguns comparadores são invertidas.

(2) Após a sequência S ser colocada para a primeira coluna de comparadores, duas sequências bitônicas, cada uma com 4 elementos, são produzidas. Cada uma delas é, então, ordenada (uma em ordem crescente e outra em ordem decrescente) usando um intercalador bitônico para sequências com 4 elementos (os comparadores das colunas 2 e 3). Isto resulta em uma única sequência bitônica com 8 elementos, a qual é agora ordenada usando um intercalador bitônico para sequências com 8 elementos (os comparadores das colunas 4, 5 e 6).

O número total de comparadores e de comparações em paralelo necessárias para ordenar uma sequência com n elementos, onde $n = 2^m$, para algum inteiro positivo m, usando ordenação bitônica, é obtido como segue. Como o número de elementos das subsequências bitônicas dobra a cada iteração, o circuito consiste de $\log n$ (isto é, m) iterações ao todo.

Se denotarmos por $q(2^i)$ o número de comparadores necessários para ordenar uma sequência bitônica com 2^i elementos, na i-ésima iteração, temos a recorrência

$$\begin{aligned} q(2) &= 1 && \textit{para } i = 1, \\ q(2^i) &= 2^{i-1} + 2q(2^{i-1}) && \textit{para } i > 1. \end{aligned}$$

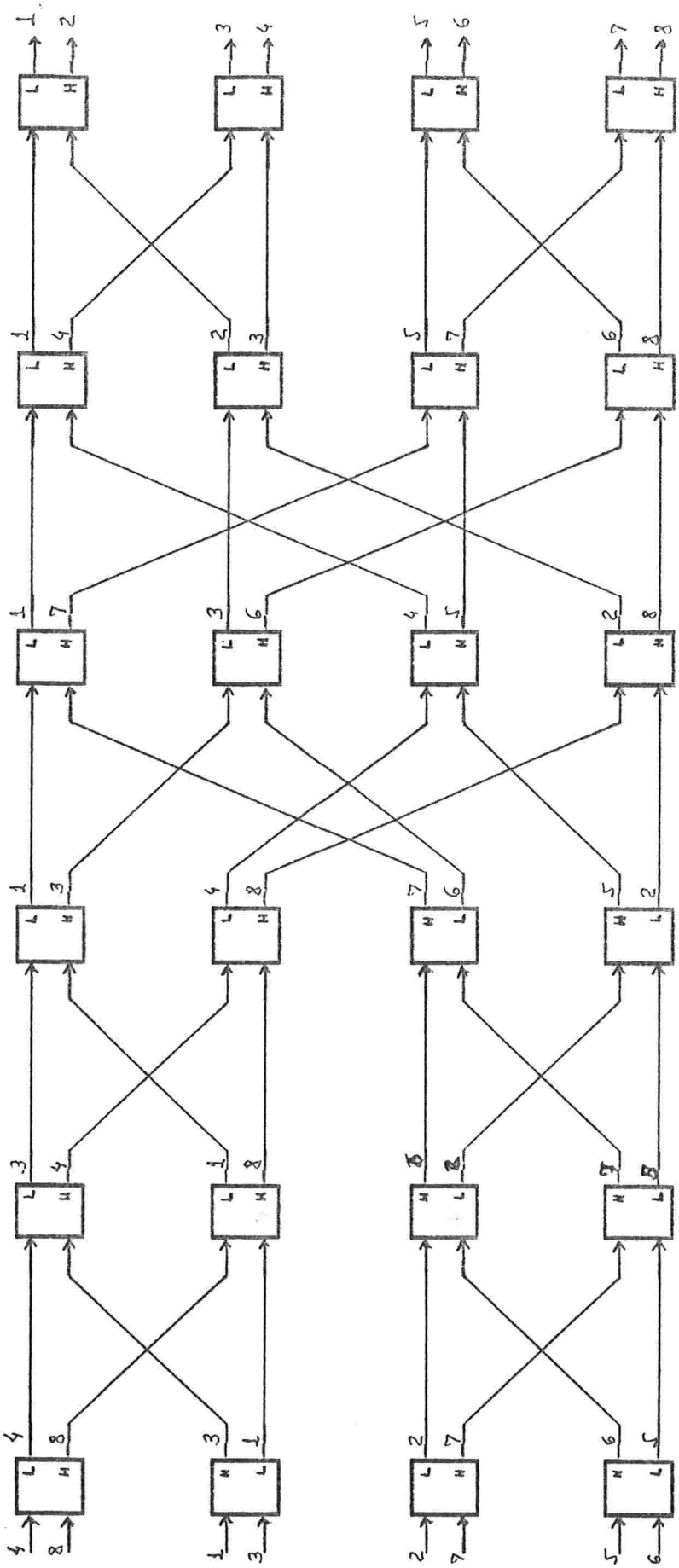


Fig. 2.9 - Circuito para ordenar {4, 8, 1, 3, 2, 7, 5, 6}.

Resolvendo esta recorrência, temos

$$\begin{aligned}
q(2^i) &= 2^{i-1} + 2q(2^{i-1}) \\
&= 2^{i-1} + 2(2^{i-2} + 2q(2^{i-2})) \\
&= 2.2^{i-1} + 2^2 q(2^{i-2}) \\
&= 2.2^{i-1} + 2^2(2^{i-3} + 2q(2^{i-3})) \\
&= 3.2^{i-1} + 2^3 q(2^{i-3}) \\
&= \cdots = \\
&= j.2^{i-1} + 2^j q(2^{i-j}).
\end{aligned}$$

para $j = i - 1$, temos

$$\begin{aligned}
q(2^i) &= (i-1)2^{i-1} + 2^{i-1} \\
&= i2^{i-1}.
\end{aligned}$$

Portanto, o número total de comparadores necessários para ordenar uma sequência com 2^m elementos é

$$\sum_{i=1}^{m} 2^{m-i} q(2^i) = \sum_{i=1}^{m} 2^{m-i}(i2^{i-1}) = 2^{m-1}\frac{m(m+1)}{2}.$$

Ou seja,

$$p(n) = O(n \log^2 n).$$

Para obter o número de comparações, em paralelo, necessárias para a ordenação, procedemos do seguinte modo: Se denotarmos por $s(2^i)$ o número de comparações (em paralelo) executadas para ordenar uma sequência bitônica com 2^i elementos, na i-ésima iteração, temos a recorrência

$$\begin{aligned}
s(2) &= 1 && \textit{para } i = 1, \\
s(2^i) &= 1 + s(2^{i-1}) && \textit{para } i > 1,
\end{aligned}$$

cuja solução é

$$s(2^i) = i.$$

Portanto, o número total de comparações, em paralelo, em um circuito para ordenar uma sequência com 2^m elementos é

$$\sum_{i=1}^{m} s(2^i) = \sum_{i=1}^{m} i = \frac{m(m+1)}{2}.$$

Ou seja,

$$t(n) = O(\log^2 n),$$

e o custo deste algoritmo é

$$c(n) = t(n) \times p(n) = O(n \log^4 n).$$

A primeira vista, este algoritmo parece não apresentar vantagens sobre a o algoritmo de ordenação por intercalação par-ímpar visto na seção anterior. O tempo de execução é o mesmo e o número de processadores utilizados é ainda maior. Porém, este algoritmo para ordenação bitônica apresenta uma propriedade para ser explorada e que pode contribuir para melhorar a eficiência dos algoritmos: O circuito para ordenar uma sequência com 2^m elementos consiste de $m(m+1)/2$ passos de comparação, cada um com 2^{m-1} comparadores. Devido a esta regularidade, a ordenação bitônica, quando implementada em outras arquiteturas, pode resultar em algoritmos para ordenação mais eficientes, como veremos na seção 2.5 a seguir.

2.5 - O EMBARALHAMENTO PERFEITO

O que faremos nesta seção é mostrar como o embaralhamento perfeito pode ser usado para ordenar uma sequência $S = \{x_0, x_1, \ldots, x_{n-1}\}$ em ordem crescente. A idéia é adaptar o algoritmo de ordenação bitônica da seção anterior para um conjunto de processadores interconectados na forma de um embaralhamento perfeito. O resultado disto é um algoritmo para ordenação cujo custo é menor do que os anteriores.

Dado um vetor com n elementos, o embaralhamento perfeito deste vetor é uma permutação dos elementos que é idêntica a um embaralhamento perfeito de cartas de baralho. Elementos da primeira metade do vetor são intercalados com elementos da segunda metade do mesmo modo (fig. 2.10).

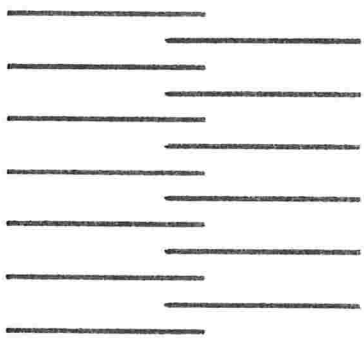


Fig. 2.10 - Embaralhamento perfeito das cartas de baralho.

Um estudo cuidadoso da figura 2.11 mostra que os índices dos processadores da esquerda são mapeados nos da direita segundo a regra da definição 2.2 abaixo.

Considere n processadores $P_0, P_1, P_2, \ldots, P_{n-1}$, onde $n = 2^m$ para algum inteiro m.

Definição 2.2 [St]. Um *Embaralhamento Perfeito* é uma forma de interconexão entre os processadores tal que o processador P_i está interligado a p_j se

$$j = \begin{cases} 2i & \text{para } 0 \leq i \leq n/2 - 1 \\ 2i + 1 - n & \text{para } n/2 \leq i \leq n - 1. \end{cases}$$

∎

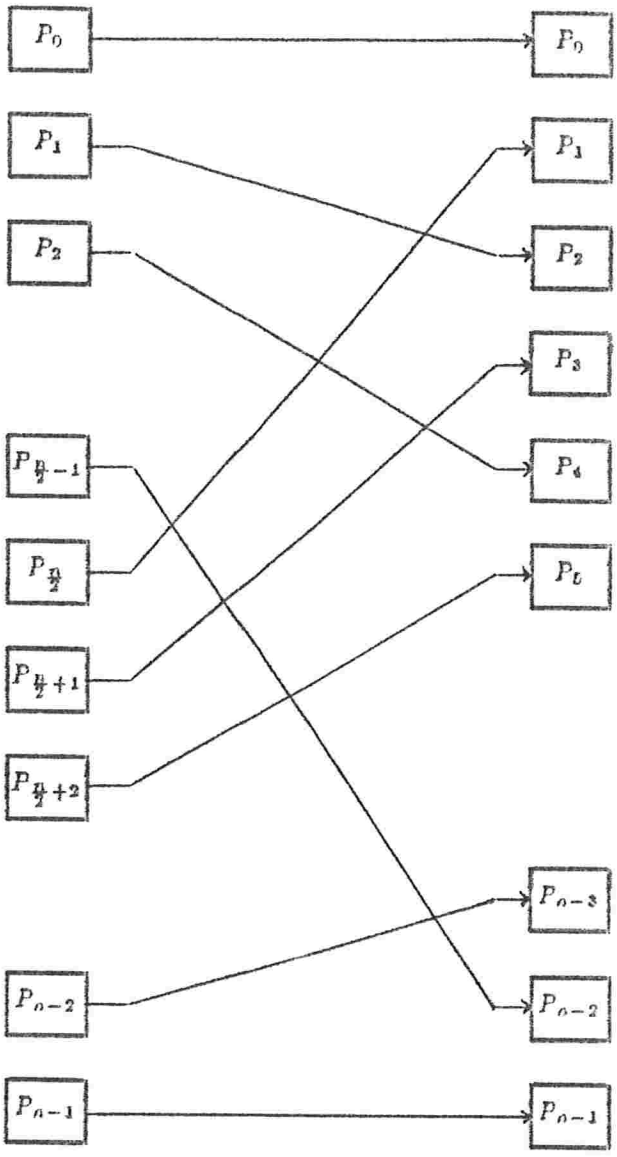


Fig. 2.11 - Embaralhamento perfeito visto como um mapeamento.

Uma segunda definição de embaralhamento perfeito usa a representação binária dos índices dos processadores.

Definição 2.3 [St]. Seja $b_{m-1}b_{m-2}\ldots b_1, b_0$ a representação binária de i, onde $b_k = 0$ ou 1, para $0 \leq k \leq m-1$, ou seja

$$i = b_{m-1}2^{m-1} + b_{m-2}2^{m-2} + \cdots + b_1 2 + b_0.$$

Em um *Embaralhamento Perfeito*, P_i está interligado a P_j se a representação binária de j for $b_{m-2}b_{m-3}\ldots b_0, b_{m-1}$, ou seja

$$j = b_{m-2}2^{m-1} + b_{m-3}2^{m-2} + \cdots + b_0 2 + b_{m-1}.$$

]

Em outras palavras, a representação binária de j é obtida por um deslocamento cíclico da representação binária de i de uma posição para a esquerda. Assim, para $m = 3$, temos

$$\begin{array}{llll} 000 \rightarrow 000, & 001 \rightarrow 010, & 010 \rightarrow 100, & 011 \rightarrow 110, \\ 100 \rightarrow 001, & 101 \rightarrow 011, & 110 \rightarrow 101, & 111 \rightarrow 111. \end{array}$$

As duas definições acima são equivalentes, pois,

$$j = 2i \Leftrightarrow 0 \leq i \leq n/2 - 1 \Leftrightarrow b_{m-1} = 0$$

e

$$j = 2i + 1 - n \Leftrightarrow n/2 \leq i \leq n - 1 \Leftrightarrow b_{m-1} = 1.$$

Isto implica que um deslocamento para a esquerda da representação binária de i, para a obtenção de j, resulta em

$$\begin{array}{ll} j = 2i & \text{se } 0 \leq i \leq n/2 - 1 \\ j = 2i + 1 - n & \text{se } n/2 \leq i \leq n - 1. \end{array}$$

É interessante visualisar o embaralhamento perfeito como uma aplicação do conjunto de processadores nele mesmo. Uma ilustração desta idéia é feita na figura 2.11, onde o conjunto de processadores é desenhado duas vezes, e as interligações do embaralhamento perfeito partem dos processadores da esquerda para os processadores da direita.

2.5.1 - Propriedades do Embaralhamento Perfeito

O embaralhamento perfeito possui duas propriedades interessantes que examinaremos agora. Suponha que n elementos $x_0, x_1, \ldots, x_{n-1}$, onde $n = 2^m$ estejam distribuídos para os processadores $P_0, P_1, P_2, \ldots, P_{n-1}$, de tal forma que x_i esteja inicialmente localizado em P_i. Em um embaralhamento perfeito ocorre o deslocamento dos dados de P_i a P_j.

Propriedade 2.1 [St]. *Após m aplicações do embaralhamento perfeito, cada elemento retorna ao processador inicial.*

Esta propriedade pode ser entendida olhando para a definição 2.3: Após m deslocamentos cíclicos, a representação binária de i retorna a sua forma inicial.

Exemplo. Na figura 2.12 podemos ver que após 3 aplicações do embaralhamento perfeito, os elementos $x_0, x_1, x_2, \ldots, x_7$ retorna às suas posições originais.

A segunda propriedade trata dos dados que estão localizados em pares de processadores que são adjacentes fisicamente e cuja representação binária de seus índices difere somente no bit mais a direita. Estes pares de processadores são $(P_0, P_1), (P_2, P_3), (P_4, P_5), \ldots, (P_{n-2}, P_{n-1})$.

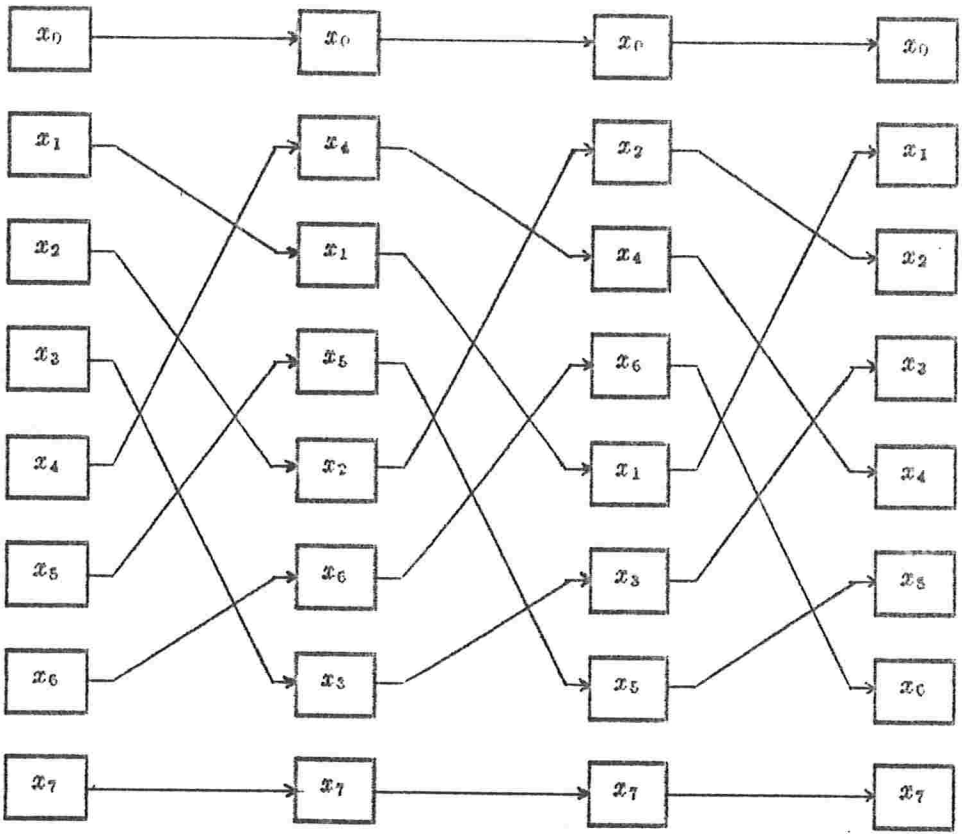


Fig. 2.12 - Ilustração da propriedade 4.1.

Propriedade 2.2 [St]. Considere os dados que estão localizados inicialmente em pares de processadores cujos índices têm representação binária $b_{m-1}b_{m-2}\ldots b_1, b_0$ e b'_{m-1} $b'_{m-2}\ldots b'_1, b'_0$, as quais diferem somente na posição $m-k$ para algum $1 \leq k \leq m$. Após k embaralhamentos, estes dados estarão localizados em processadores adjacentes.

Esta propriedade também pode ser entendida olhando para a definição 2.3: Após k deslocamentos cíclicos para a esquerda, estes dados estarão localizados em processadores cujos índices diferem somente na primeira posição, ou seja, em processadores adjacentes.

Propriedade 2.3 [St]. Assuma que os elementos $x_0, x_1, \ldots, x_{n-1}$, onde $n = 2^m$ estejam localizados nos processadores $P_0, P_1, P_2, \ldots, P_{n-1}$, de tal modo que x_i esteja inicialmente localizado em P_i. Então a posição na qual difere a representação binária dos índices dos elementos localizados em processadores adjacentes, move para a direita após cada operação de embaralhamento.

Esta propriedade decorre da anterior. Se o índice j do processador contendo x_i após k embaralhamentos é obtido deslocando a representação binária de i k vezes para a esquerda, então, certamente, o índice i do elemento em P_j, após k embaralhamentos, é obtido deslocando a representação binária de j k vezes para a direita.

Exemplo. A figura 2.13 ilustra as propriedades 2.2 e 2.3 para $m = 3$. A representação binária do índice i de x_i é colocada dentro do processador no qual x_i está localizado a cada estágio. Uma linha vertical é usada para localizar processadores adjacentes.

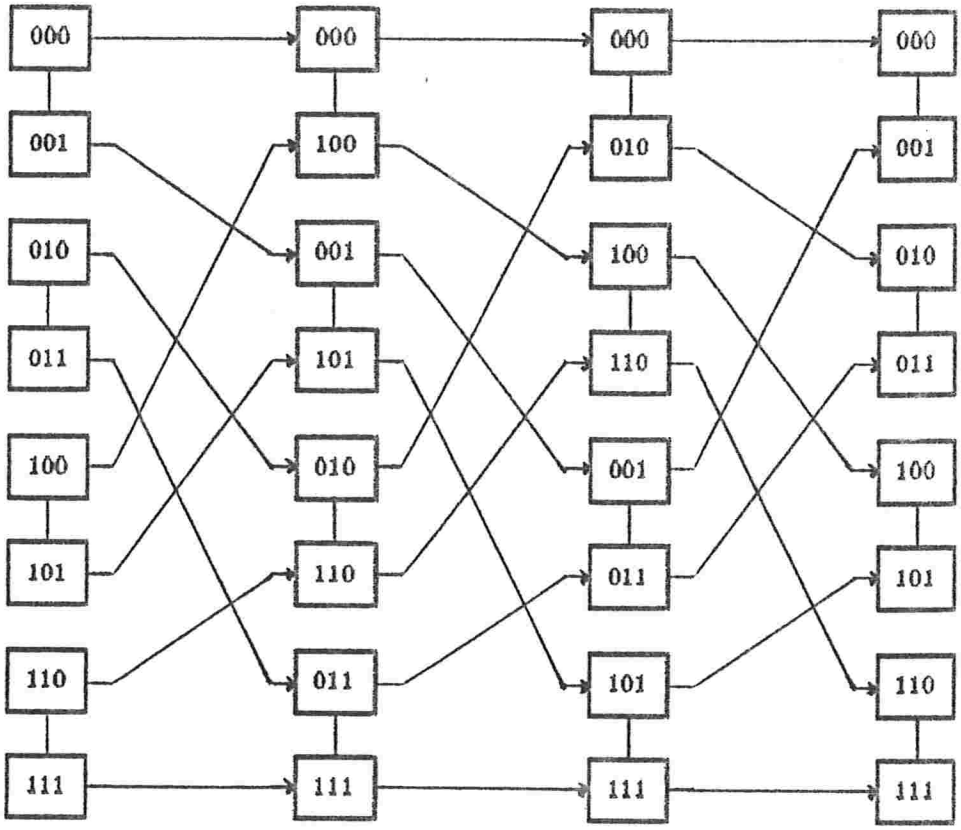


Fig. 2.13 - Ilustração das propriedades 4.2 e 4.3.

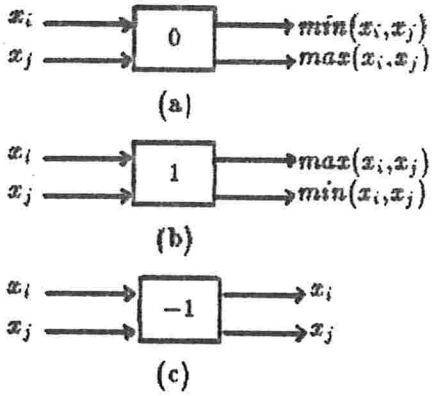


Fig. 2.14 - Três tipos de comparadores.

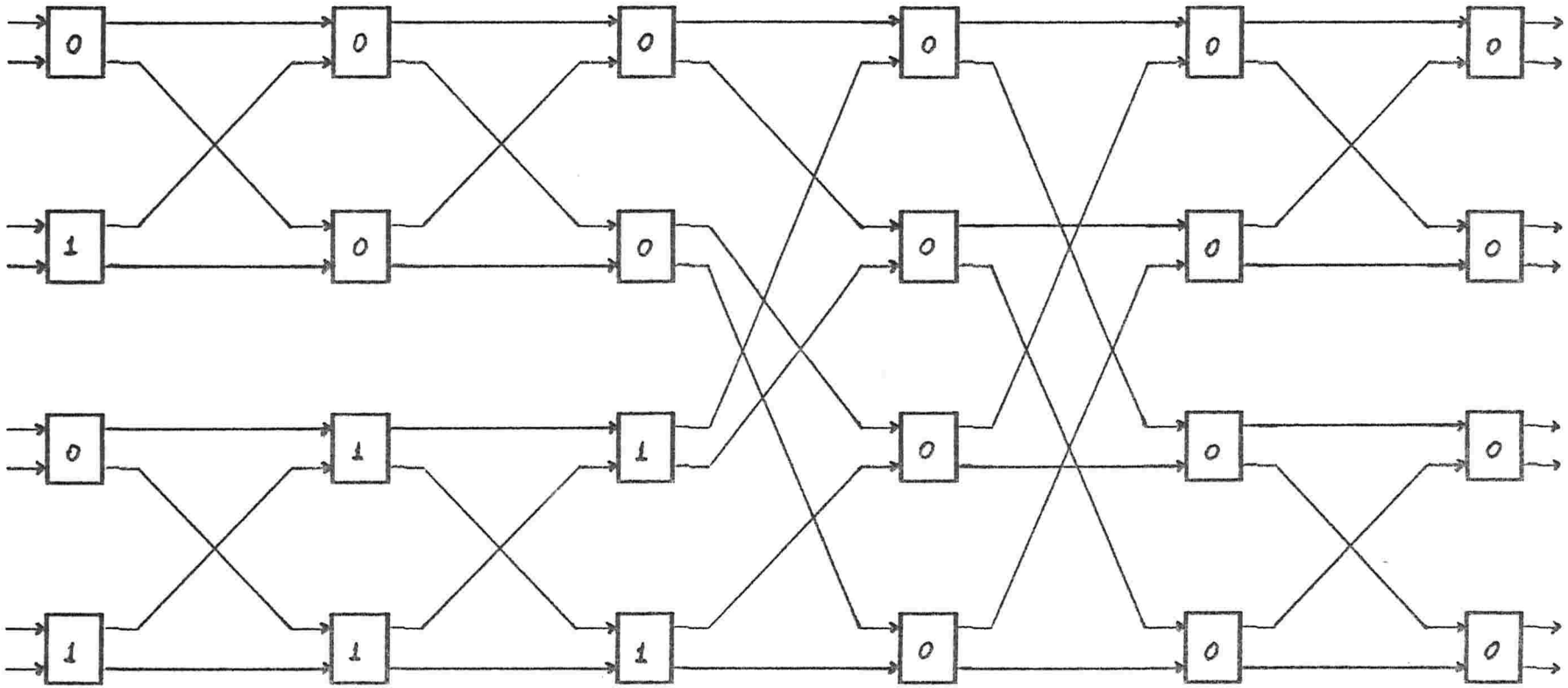


Fig. 2.15 - Circuito de Ordenação Bitônica.

2.5.2 - Ordenação Bitônica usando Embaralhamento Perfeito

Começaremos introduzindo algumas notações que serão usadas nos diagramas desta seção. Relembre que um comparador é um simples processador que executa uma operação de comparação e troca de posição sobre as duas entradas para produzir as duas saídas. De agora em diante um comparador convencional, ou seja, aquele que coloca o menor das duas entradas na linha da saída superior e o maior na inferior, será rotulado com um 0, como mostra a figura 2.14(a). Um comparador que inverte esta ordem será rotulado com um 1, como mostra a figura 2.14(b). Necessitamos, também, de um comparador que não altere a ordem das entradas; tal comparador será rotulado com um -1, como mostra a figura 2.14(c).

Relembremos o circuito de ordenação bitônica, para $n = 2^3$, da figura 2.9. Este circuito está reproduzido por conveniência na figura 2.15 usando as convenções introduzidas agora.

Para explicar como o algoritmo de ordenação bitônica será simulado pelo embaralhamento perfeito, é necessário representarmos o circuito da figura 2.15 como na figura 2.16. Aqui, as linhas horizontais representam os elementos de entrada para o circuito, numeradas de 0 a 7 por dígitos binários. Uma seta vertical representa um comparador. As duas linhas ligadas pelos extremos da seta são as entradas para o comparador. A seta aponta no sentido em que deve ser colocado o maior das duas entradas quando produzida a saída. Note que cada par de linhas conectadas por uma seta difere por um único bit.

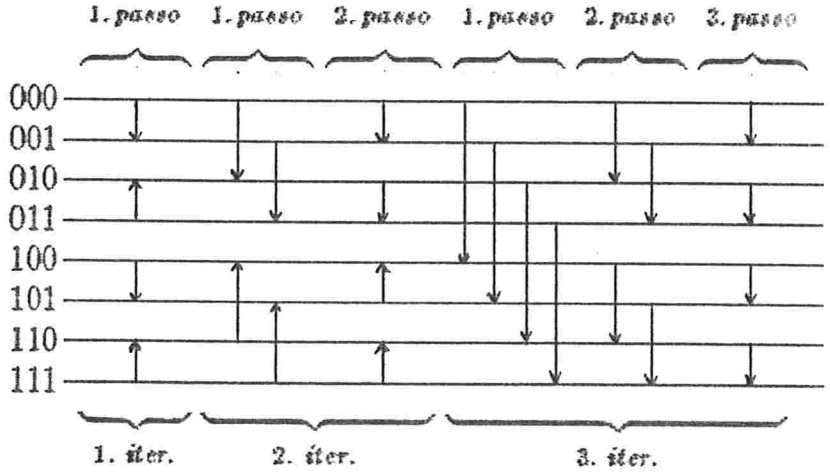


Fig. 2.16 - Representação diferente do circuito de ordenação bitônica.

Definição 2.4 [St]. A cada passo em uma iteração na ordenação bitônica, o bit *Pivô* é definido como sendo o bit no qual cada par de linhas conectadas por uma seta difere.]

Seja $b_{m-1}b_{m-2}\ldots b_1b_0$ a representação binária da linha i, onde $n = 2^m$ é o número total de linhas. Então os bits pivô para iterações sucessivas de ordenação bitônica são:

1. iteração : b_0
2. iteração : b_1, b_0
3. iteração : b_2, b_1, b_0

⋮

m. iteração : $b_{m-1}, b_{m-2}, \ldots, b_1, b_0$.

Em outras palavras, a sequência de pivôs consiste de m subsequências: A subsequência na j-ésima iteração tem j elementos e consiste dos bits b_{j-1} até b_0, nesta ordem.

O que precisamos, então, para implementar o algoritmo de ordenação bitônica, é embaralhar a sequência S um número de vezes antes de cada iteração até certificar que o bit pivô no início da j-ésima iteração é b_{j-1}. Então, para cada um dos pivôs, um passo de comparação seguido por um embaralhamento é executado sobre S até que o bit b_0, o fim da subsequência de pivôs associada à j-ésima iteração, ser alcançado. Como existe j pivôs associados à j-ésima iteração, a sequência S deve ser embaralhada $m - j + 1$ vezes antes da j-ésima iteração seguida de j comparações e $j - 1$ embaralhamentos.

Partindo das observações dadas acima e das propriedades do embaralhamento perfeito, uma sequência $S = \{x_0, x_1, \ldots, x_{n-1}\}$, $n = 2^m$, pode ser ordenada usando o circuito que está na figura 2.17. Aqui temos n módulos de armazenamento, numerados de 0 a $n-1$, que contém o vetor a ser ordenado, e $n/2$ comparadores numerados de 0 a $n/2-1$. As n linhas de saída dos módulos de armazenamento, numeradas de 0 a $n-1$, são ligadas às $2(n/2)$ linhas de entrada dos comparadores, numeradas de 0 a $n-1$, por um embaralhamento perfeito. As $2(n/2)$ linhas de saída dos comparadores, numeradas de 0 a $n-1$, realimentam as n linhas de entrada dos módulos de armazenamento, numeradas de 0 a $n-1$, tais que a linha de saída i dos comparadores está ligada ao módulo de armazenamento i.

Assuma que a sequência S esteja colocada inicialmente nos módulos de armazenamento tal que x_i esteja localizado no módulo i. Após um embaralhamento, cada comparador recebe dois elementos de S cuja representação binária de seus índices diferem apenas no bit mais à esquerda. Cada embaralhamento subsequente move os bits diferentes uma posição para a direita (Propriedade 2.3). Portanto, se a um dado momento o pivô é b_j, então, com um embaralhamento a mais, o pivô passa a ser $b_{(j-1) \bmod m}$.

A questão que surge neste momento é: Como os comparadores devem se comportar? Obviamente, durante os $m - j + 1$ embaralhamentos que precedem a j-ésima iteração, os comparadores encontram-se inativos, permitindo que as duas entradas sejam colocadas nas saídas sem alterar a ordem. Durante os j passos de comparação, ele opera sobre as entradas colocando a menor na linha de saída superior e a maior na linha de saída inferior, ou vice versa, dependendo se o seu rótulo for 0 ou 1. Para distinguir entre estes três estados, utilizaremos um vetor *estado* cuja i-ésima componente determina o comportamento do i-ésimo comparador, Ou seja

$$estado(i) = \begin{cases} 0 & \text{quando o comparador } i \text{ se comporta como o da fig. 4.13(a)} \\ 1 & \text{quando o comparador } i \text{ se comporta como o da fig. 4.13(b)} \\ -1 & \text{quando o comparador } i \text{ se comporta como o da fig. 4.13(c).} \end{cases}$$

Durante a execução, o vetor *estado* deverá ser embaralhado, como explicaremos no parágrafo a seguir, e o embaralhamento necessita de n elementos. Por isso assumiremos que o vetor *estado* tem n elementos.

Quando o embaralhamento perfeito é usado para ordenar uma sequência S, a sequência deve ser embaralhada $m-j+1$ vezes na j-ésima iteração, antes de ser executada a primeira

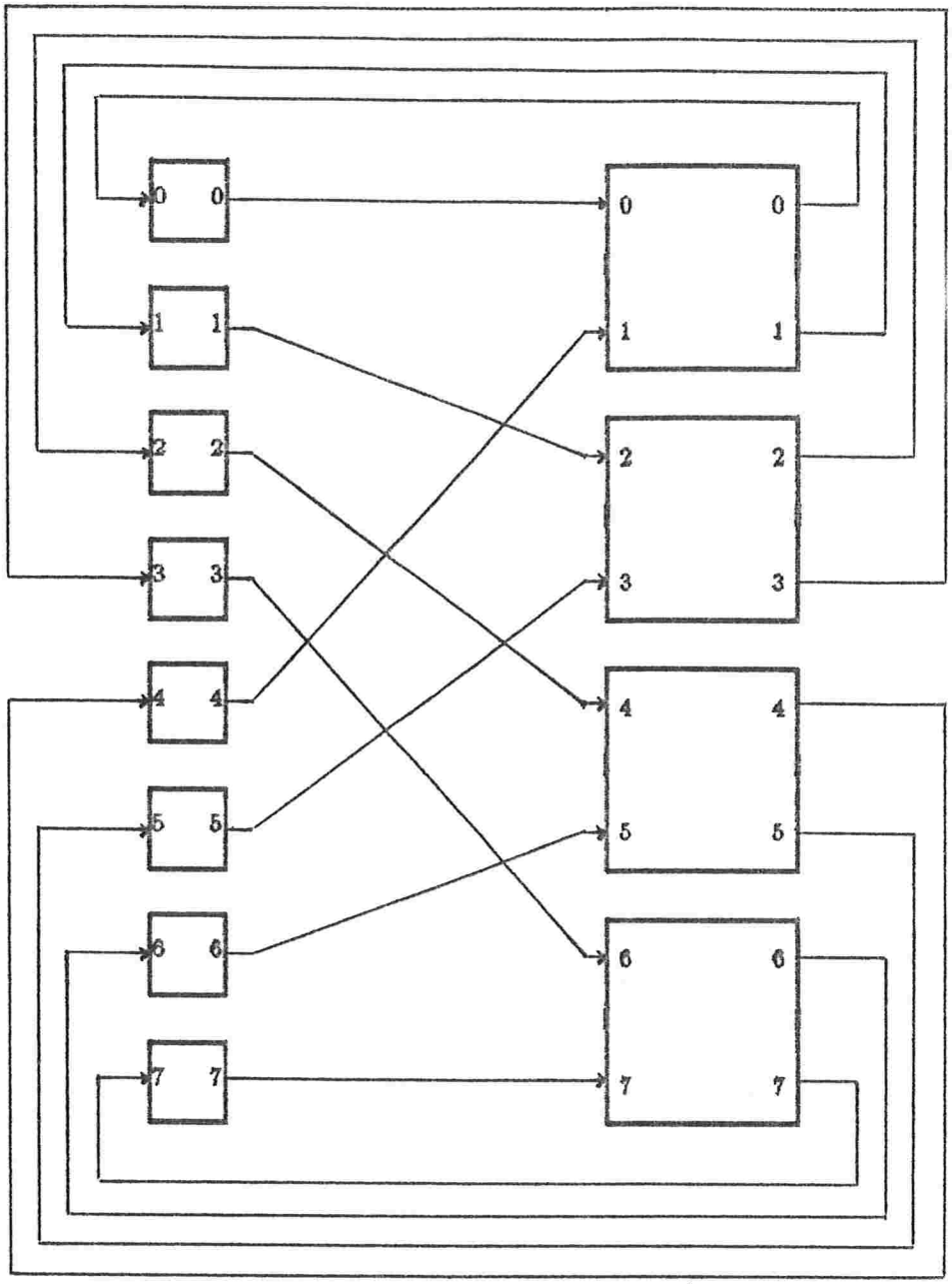


Fig. 2.17 - Circuito para ordenação usando embaralhamento perfeito.

comparação, para obter a posição correta dos pivôs. Portanto, é necessário, também, embaralhar o vetor *estado* para não perdermos o comportamento de cada processador durante a execução do algoritmo.

Na j-ésima iteração da ordenação bitônica de $n = 2^m$ elementos, Os valores para o vetor *estado* são cópias de 2^{j-1} 0s seguido de 2^{j-1} 1s, num total de 2^{m-1} comparadores. Por exemplo, quando $m = 3$ e $j = 2$, o vetor estado, com 8 elementos, deve conter inicialmente duas cópias dos valores $0, 0, 1, 1$. Estes valores são embaralhados $m - j + 1$ vezes, ou seja, 2 vezes, para obter 2 cópias dos valores $0, 1, 0, 1$. No caso geral, quando uma sequência de 2^m dígitos binários, composta por 2^{j-1} 0s seguido de 2^{j-1} 1s, é embaralhada $m - j + 1$ vezes, o resultado é uma sequência de 0s e 1s alternados $(0, 1, 0, 1, 0, 1, \ldots)$. Portanto, podemos economizar estes passos, e assumir que no início de cada uma das m iterações da ordenação bitônica o vetor *estado* (com 2^m elementos) é composto por $0, 1, 0, 1, 0, 1, \ldots$. E após cada um dos $j - 1$ embaralhamentos seguintes da sequência S, o vetor *estado* também deve ser embaralhado. Portanto, quando o último passo da j-ésima iteração é executado, o vetor *estado* terá sido embaralhado m vezes ao todo, retornando assim à sua configuração inicial, que é uma sequência com 2^m elementos consistindo de cópias de 2^{j-1} 0s seguido de 2^{j-1} 1s.

Estamos agora em condições de descrever formalmente o algoritmo de ordenação bitônica usando embaralhamento perfeito. Como mencionamos acima, a sequência S que vai ser ordenada está colocada inicialmente nos módulos de armazenamento, com um elemento por módulo. As instruções *circula*(S), *embaralha*(*estado*) e *estado*(j) são usadas no algoritmo e explicadas abaixo.

(1) *circula*(S)

Esta instrução compreende as seguintes operações:

(a) Cada módulo de armazenamento envia o elemento de S que ele contém para o processador em que ele está intercalado através do embaralhamento perfeito.

(b) Após receber os dois valores de S, o comparador P_i comporta-se como o comparador da figura 4.13(a), (b) ou (c), dependendo do valor *estado*(i).

(c) O valor da saída de cada comparador é enviado de volta ao módulo de armazenamento ao qual ela está interligada.

(2) *embaralha*(*estado*)

Esta instrução é usada para embaralhar o vetor *estado* como segue:

(a) Para cada i, o módulo de armazenamento i envia o valor *estado*(i) para o processador ao qual ele está interligado através do embaralhamento perfeito.

(b) As duas entradas do *estado* em cada processador são enviadas de volta (sem trocar a ordem) ao módulo de armazenamento interligado às linhas de saída.

(3) *estado*(j)

Esta instrução computa a entrada do vetor *estado*. Dado um inteiro j, um vetor com 2^m elementos é criado como segue:

$$estado = \begin{cases} -1, -1, -1, \ldots, -1 & \text{se } j = -1 \\ 0, 1, 0, 1, \ldots, 0, 1 & \text{se } 1 \leq j \leq m - 1 \\ 0, 0, 0, 0, \ldots, 0, 0 & \text{se } j = m. \end{cases}$$

O algoritmo é dado abaixo.

Algoritmo:

```
para j = 1 até m faça
início
    estado(-1);
    para k = 1 até m - j faça circula(S);
    estado(j);
    para k = m - j + 1 até m faça
    início
        circula(S);
        embaralha(estado);
    fim
fim.
```

O passo principal no algoritmo e o que é executado mais frequentemente é o passo *circula*(S), que consiste de uma comparação e um envio de valores para os módulos de armazenamento. Para cada uma das m iterações (ou seja, para cada valor de j) *circula*(S) é executada m vezes, num total de m^2 vezes. Como $m = \log n$, temos

$$t(n) = O(\log^2 n).$$

Como são usados $n/2$ processadores, temos

$$p(n) = n/2.$$

Portanto,

$$c(n) = t(n) \times p(n) = O(n \log^2 n),$$

o que representa uma melhora em relação aos algoritmos anteriores. O menor custo deste algoritmo é devido ao uso de menos processadores que a ordenação bitônica da seção anterior.

Exemplo. O algoritmo apresentado acima é ilustrado na figura 2.18 para a sequência $S = \{4, 8, 1, 3, 2, 7, 5, 6\}$. Como $n = 2^3$, existem três iterações, cada uma composta por três passos. Para melhor entendimento, desenhamos o conjunto de comparadores e módulos de armazenamento 9 vezes. Os elementos de S são mostrados dentro dos módulos de armazenamento, enquanto os valores do vetor *estado* aparecem dentro dos comparadores.

Embora mais eficiente que o circuito para a ordenação bitônica da seção anterior, este algoritmo ainda não apresenta um custo ótimo para a ordenação. Uma melhora a mais do custo, na implementação da ordenação bitônica pode ser encontrada em [BS78]. A idéia básica utilizada é também o embaralhamento perfeito. Para reduzir o custo do algoritmo paralelo, menos processadores são utilizados, porém cada um deles não é mais um simples comparador, mas sabe ordenar e intercalar pequenas sequências.

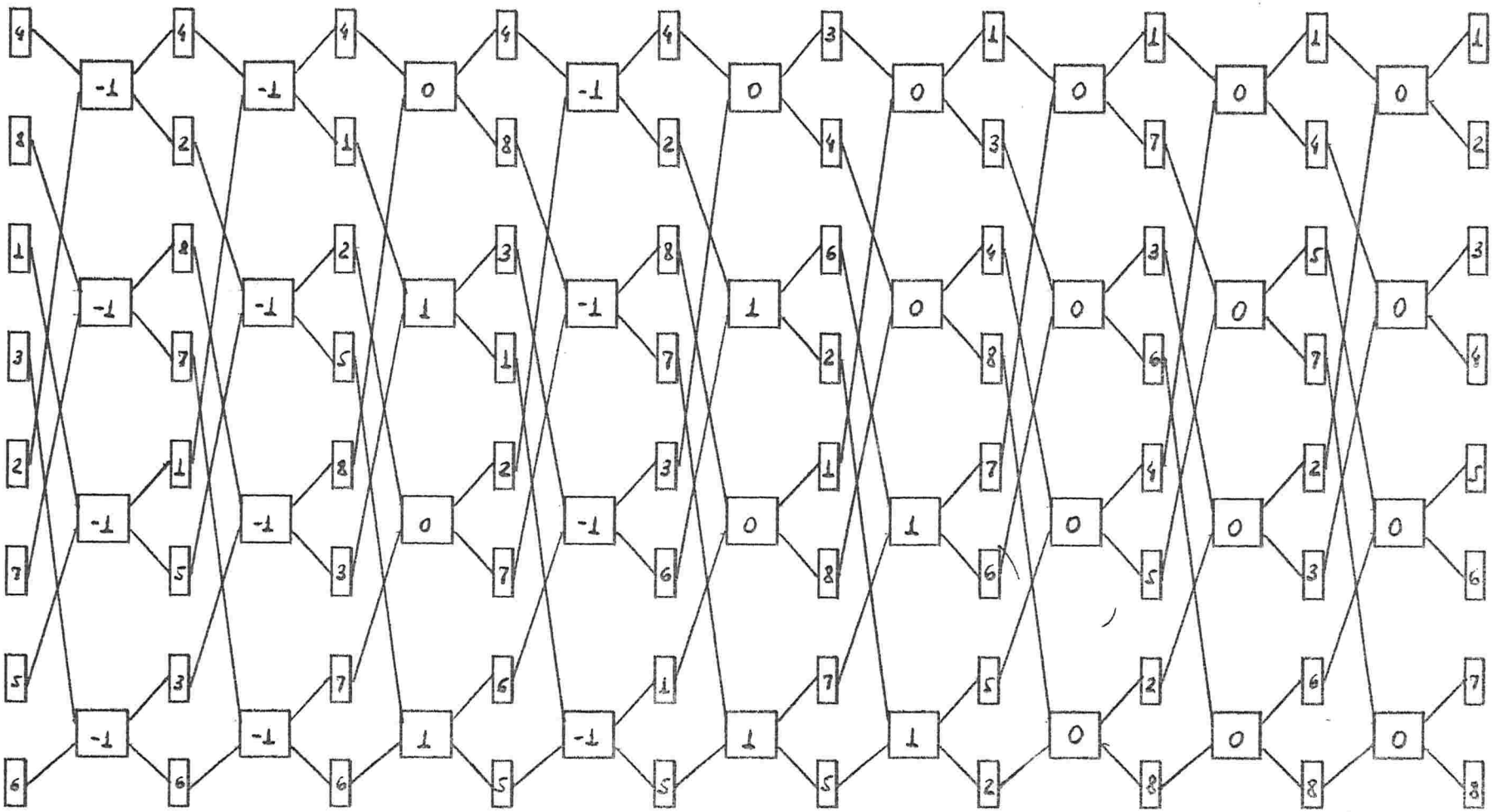


Fig. 2.18 - Circuito para ordenar $S = \{4, 8, 1, 3, 2, 7, 5, 6\}$.

2.5 - COMENTÁRIOS FINAIS

O problema da ordenação em paralelo tem sido importante foco de pesquisa nos últimos dez anos. Muitos algoritmos paralelos para ordenação são conhecidos atualmente, e novos algoritmos estão sendo desenvolvidos. Do ponto de vista teórico, a principal dificuldade é explorar sistematicamente o intrínseco paralelismo da ordenação e da intercalação. Se este problema fosse contornado, certamente teríamos algoritmos que ordenariam n elementos em tempo $O(\log n)$ usando $O(n)$ processadores.

O método de intercalação par-ímpar de Batcher envolve um número fixo de comparações. Sabemos, a partir dos cálculos que esse número é $O(n \log^2 n)$; Porém, o problema de determinar quantas comparações resultam em troca de posição das entradas é mais difícil. [Se] e [Ke] apresentam um estudo deste problema com os seguintes resultados: Na análise do pior caso, a razão entre o número de trocas e de comparações tende a 1 quando $n \to \infty$; E na análise do caso médio, apenas 1/4 dos comparadores estão envolvidos em trocas.

Na tabela 1 apresentamos as complexidades assintóticas dos principais algoritmos paralelos para ordenação em termos do número de processadores utilizados e tempo de execução.

Algoritmo	Processadores	Tempo
Batcher (interc. par-ímpar)	$n \log^2 n$	$O(\log^2 n)$
Batcher (interc. Bitônica)	$n \log^2 n$	$O(\log^2 n)$
Stone (emb. perf.)	$n/2$	$O(\log^2 n)$
Thompson & Kung (Malha-bit.)	n	$O(\sqrt{n})$
Muller-preparata	n^2	$O(\log n)$
Hirschberg (1)	n	$O(\log n)$
Hirschberg (2)	$n^{1+1/k}$	$O(k \log n)$
Preparata (1)	$n \log n$	$O(\log n)$
Preparata (2)	$n^{1+1/k}$	$O(k \log n)$
Ajtai e outros	$n \log n$	$O(\log n)$

Tabela 1 - Número de processadores e tempo de execução dos principais algoritmos paralelos para ordenação.

O algoritmo Hirschberg (1) estabelece que os valores a ser ordenados devem pertencer ao conjunto $\{0, 1, \ldots, m-1\}$, e não deve ocorrer valores repetidos. O problema da não ocorrência de valores repetidos é resolvido em Hirschberg (2) [H].

Capítulo III

Limites Inferiores

3.1 - INTRODUÇÃO

Neste capítulo trataremos da análise da complexidade de tempo paralelo para os problemas:

1 - Encontrar o elemento *Máximo* em um conjunto;

2 - *Intercalação* de duas listas ordenadas;

3 - *Ordenação*.

Esta análise será feita somente para algoritmos que são baseados em comparações binárias.

Para cada problema a entrada consiste de um conjunto de elementos sobre os quais existe uma ordenação linear.

O modelo considerado aqui possui k processadores disponíveis, e portanto, k comparações podem ser executadas simultaneamente. Os processadores estão sincronizados de tal modo que em um intervalo de tempo, cada um deles executa uma comparação. Definimos, então:

a) $max_k(n)$ como a medida de complexidade para o problema de encontrar o máximo entre n elementos sobre uma máquina com k processadores.

b) $intercala_k(m,n)$ como a medida de complexidade para o problema de intercalar duas listas ordenadas sobre uma máquina com k processadores.

c) $ordena_k(n)$ como a medida de complexidade para o problema de ordenar n elementos sobre uma máquina com k processadores.

Nas seções seguintes, trataremos dos limites inferiores para os problemas definidos acima.

3.2 - O MÁXIMO

Trataremos aqui da análise de pior caso do problema de encontrar o máximo entre n elementos usando k processadores. Consideraremos primeiramente o caso $k = n$ e a partir disto, derivaremos soluções para os demais casos.

Teorema 3.1 [Va]. Para $k = n > 1$,

$$max_k(n) \geq O(\log\log n).$$

Dem. Considere a execução de um algoritmo qualquer, baseado em comparações, para encontrar o máximo entre n elementos. Seja C_i o conjunto dos elementos que até o i-ésimo passo (em paralelo) de computação ainda concorrem ao máximo, isto é, $x \in C_i$ se e somente se até o tempo i, não foi possível mostrar que x é menor do que qualquer outro elemento. O número de elementos em C_i será denotado por c_i. O que faremos é limitar o valor de c_{i+1} em termos de c_i e k.

Considere um grafo G definido da seguinte maneira: Identifique os vértices como sendo os c_i elementos de C_i e os arcos como sendo as comparações, isto é, existe um arco de x para y se x está sendo comparado com y. Então o valor de c_{i+1} está limitado inferiormente por

$$c_{i+1} \geq min\{max\{h/\ G\ contém\ um\ conjunto\ estável\ de\ tamanho\ h\}/G\ é\ um\ grafo\ com\ c_i\ vértices\ e\ k\ arcos\},$$

pois

(1) As k comparações devem ser feitas sobre os c_i elementos que concorrem ao máximo; Há várias maneiras de dispor os k arcos entre os c_i elementos.

(2) Para cada disposição destes arcos, um conjunto estável* máximo representa os candidatos, no próximo passo, ao máximo.

(3) Entre esses conjuntos estáveis máximos, o de menor cardinalidade representa o melhor caso.

(4) Portanto, c_{i+1} está limitado inferiormente pela cardinalidade do menor conjunto esta'vel.

A partir disto podemos concluir que: ([Bg] pg. 282 - corol. 2)

$$c_{i+1} \geq \frac{c_i^2}{2k + c_i}.$$

---

* Um conjunto S de vértices de um grafo G é *estável* se não existe arestas em G ligando quaisquer dois vértices de S

Resolvendo esta desigualdade para $c_0 = n = k$, temos:

$$\begin{aligned} c_{i+1} &\geq \frac{c_i^2}{3n} = \frac{1}{3n}c_i^2 \\ &\geq \frac{1}{3n}\left(\frac{c_{i-1}^2}{3n}\right)^2 = \frac{1}{(3n)^{2^2-1}}c_{i-1}^{2^2} \\ &\geq \frac{1}{(3n)^{2^2-1}}\left(\frac{c_{i-2}^2}{3n}\right)^{2^2} = \frac{1}{(3n)^{2^3-1}}c_{i-2}^{2^3} \geq \cdots \geq \\ &\geq \frac{1}{(3n)^{2^{j+1}-1}}c_{i-j}^{2^{j+1}}. \end{aligned}$$

Para $j = i$ e $c_0 = n$, temos

$$c_{i+1} \geq \frac{1}{(3n)^{2^{i+1}-1}}n^{2^{i+1}} = 3n\left(\frac{n}{3n}\right)^{2^{i+1}} = 3n.\left(\frac{1}{3}\right)^{2^{i+1}}.$$

Vamos determinar, agora, para quais valores de i tem-se $3n\left(\frac{1}{3}\right)^{2^{i+1}} > 1$.

$$\begin{aligned} 3n\left(\frac{1}{3}\right)^{2^{i+1}} > 1 &\Rightarrow -2^{i+1}\log 3 > -\log 3n \\ &\Rightarrow 2^{i+1} < \frac{1}{\log 3}\log 3n \\ &\Rightarrow i < \log\log 3n - (\log\log 3 + 1). \end{aligned}$$

Ou seja, c_{i+1} excede a unidade enquanto $i < \log\log 3n - const$. Portanto, $max_k(n) \geq O(\log\log n)$.]

Corolário 3.1 [Va]. Se $4 \leq 2n \leq k \leq n(n-1)/2$, então

$$max_k(n) \geq O(\log\log n - \log\log(k/n)).$$

Dem. Usando a mesma idéia do teorema anterior, chegaremos à seguinte desigualdade:

$$\begin{aligned} c_{i+1} &\geq \frac{c_i^2}{2k+c_i} \geq \frac{c_i^2}{2k+n} \geq \frac{1}{2k+n}.\left(\frac{c_{i-1}^2}{2k+n}\right)^2 = \frac{1}{(2k+n)^{2^2-1}}c_{i-1}^{2^2} \\ &\geq \cdots \geq \\ &\geq \frac{1}{(2k+n)^{2^{j+1}-1}}c_{i-j}^{2^{j+1}}. \end{aligned}$$

Para $j = i$ e $c_0 = n$, temos

$$c_{i+1} \geq \frac{1}{(2k+n)^{2^{i+1}-1}}n^{2^{i+1}} = (2k+n)\left(\frac{n}{2k+n}\right)^{2^{i+1}}.$$

A partir daqui, temos:

$$(2k+n)\left(\frac{n}{2k+n}\right)^{2^{i+1}} > 1 \Rightarrow \log(2k+n) + 2^{i+1}\log\left(\frac{n}{2k+n}\right) > 0$$
$$\Rightarrow 2^{i+1}\log\left(\frac{2k+n}{n}\right) < \log(2k+n)$$
$$\Rightarrow 2^{i+1} < \frac{\log(2k+n)}{\log(2\frac{k}{n}+1)}$$
$$\Rightarrow i < \log\log(2k+n) - (\log\log(2\frac{k}{n}+1)+1.$$

Portanto, $max_k(n) \geq O(\log\log n - \log\log(\frac{k}{n}))$.]

No teorema a seguir apresentaremos um algoritmo ótimo para o problema de encontrar o máximo entre n elementos, para o caso $k = n$.

Teorema 3.2 [Va]. Para $k = n > 1$,

$$max_k(n) \leq O(\log\log n).$$

Dem. É sabido [Bg] que qualquer grafo G, com p vértices, que não contém um conjunto estável maior que x tem pelo menos tantos arcos quanto o grafo $G_{p,x}$. $G_{p,x}$ é definido como sendo o grafo com p vértices que consiste de x cliques disjuntos, dos quais $p - x(q-1)$ têm q vértices e os $xq - p$ restantes têm $q-1$ vértices, onde $q = \lceil p/x \rceil$. Pode-se mostrar [Bg] que o grafo $G_{p,x}$ tem $(q-1)(2p-xq)/2$ arestas ao todo.

No nosso algoritmo, o i-ésimo passo de comparação (em paralelo) será baseado em um grafo tal que $p = c_i$, onde cada clique resultará em um candidato a C_{i+1}. Para minimizar c_{i+1} temos que escolher entre os grafos

$$\{G_{c_i,x}/x = 1, 2, \ldots;\ G_{c_i,x} \textit{ tem menos que } k \textit{ arcos}\}$$

aquele que minimiza o índice x.

Como o grafo $G_{c_i,x}$ tem $(q-1)(2p-xq)/2$ arestas, concluímos que

$$c_{i+1} = min\{x/(\lceil \frac{c_i}{x} \rceil - 1).(2c_i - x\lceil \frac{c_i}{x} \rceil)/2 \leq k\}.$$

Resolvendo esta desigualdade, temos:

$$2c_i\lceil \frac{c_i}{x} \rceil - x\lceil \frac{c_i}{x} \rceil^2 - 2c_i + x\lceil \frac{c_i}{x} \rceil \leq 2k$$
$$\Rightarrow 2\frac{c_i^2}{x} - x\frac{c_i^2}{x^2} - 2c_i + x\frac{c_i}{x} \leq 2k$$
$$\Rightarrow \frac{c_i^2}{x} - c_i \leq 2k$$
$$\Rightarrow x \geq \frac{c_i^2}{2k + c_i}.$$

Portanto,

$$c_{i+1} \leq \lceil \frac{c_i^2}{2k + c_i} \rceil.$$

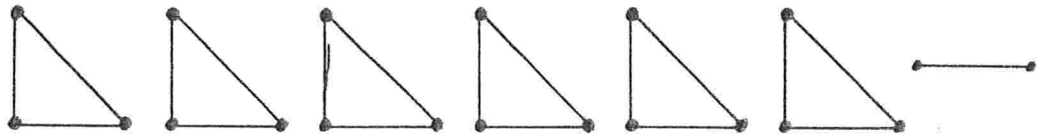
Observando a demonstração do Teorema 3.1, concluímos que este algoritmo é executado em tempo (paralelo) $O(\log \log n)$. ∎

Exemplo. Suponha que queremos determinar o máximo entre 20 elementos, usando 20 processadores e aplicando a algoritmo acima. Temos, então, $c_0 = n = k = 20$. Precisamos determinar c_1 tal que:

$$c_1 = min\{x / G_{c_1,x} \textit{ tem menos que } k \textit{ arcos}\}.$$

Teorema 3.2 nos diz que $c_1 \leq \lceil \frac{c_0^2}{2k+c_0} \rceil = 7$ e $q = \lceil \frac{c_0}{x} \rceil = 3$. Podemos, agora, construir $G_{c_1,x}$ com $x = 7$ cliques disjuntos dos quais $p - x(q-1) = 6$ têm $q = 3$ vértices e os $xq - p = 1$ restante tem $q - 1 = 2$ vértices. As comparações são, então, efetuadas como no grafo $G_{c_1,x}$ abaixo.

$G_{c_1,x}$:

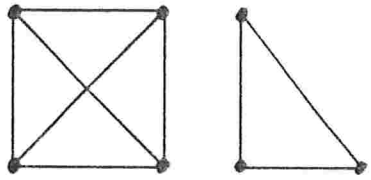


A partir destas comparações temos $c_1 = 7$ candidatos ao máximo. Calculemos c_2

$$c_2 = min\{x / G_{c_2,x} \textit{ tem menos que } k \textit{ arcos}\}.$$

$c_2 \leq \lceil \frac{c_1^2}{2k+c_1} = x \rceil = 2$ e $q = \lceil \frac{c_1}{x} \rceil = 4$. O grafo $G_{c_2,x}$ tem $x = 2$ cliques disjuntos dos quais $p - x(q-1) = 1$ tem $q = 4$ vértices e o outro $(xq - p = 1)$ tem $q - 1 = 3$ vértices.

$G_{c_2,x}$:



Temos agora $c_2 = 2$ candidatos ao máximo, e uma iteração a mais é suficiente para encontrar o máximo.

O que provamos nos Teoremas 3.1 e 3.2 foi:

Teorema 3.3. Para $k = n > 1$,

$$max_k(n) = \Theta(\log \log n).$$ ∎

O mesmo algoritmo do teorema 3.2 nos garante que:

Corolário 3.2. Para $4 \leq 2n \leq k \leq n(n-1)/2$,

$$max_k(n) = \Theta(\log\log n - \log\log(k/n)).$$]

O caso restante, em que $k < n$, pode ser resolvido do seguinte modo: com k comparações simultâneas podemos reduzir c_i de no máximo k elementos. Portanto, enquanto $c_i \geq 2k$, podemos encontrar uma ótima redução comparando k pares disjuntos em c_i. Quando $c_i < 2k$, podemos aplicar o algoritmo do Teorema 3.2. Concluímos, então o seguinte:

Corolário 3.3. Para $1 < k < n$,

$$O(\frac{n}{k} + \log\log k) \leq max_k(n) \leq O(\frac{n}{k} + \log\log k).$$]

3.3 - INTERCALAÇÃO

Sabemos que o problema de intercalar duas listas ordenadas, com m e n elementos, respectivamente, requer $m + n - 1$ comparações, no pior caso. Ainda não se conhece um algoritmo paralelo de custo ótimo para este problema. O que faremos aqui é apresentar um algoritmo que é o mais eficiente entre os que existem até o momento. Trataremos, primeiramente, do caso $k = \sqrt{nm}$, e a partir deste derivaremos soluções para os demais casos.

Teorema 3.4 [Va]. Para $k = \lfloor\sqrt{mn}\rfloor$ e $1 < n \leq m$,

$$intercala_k(n, m) \leq O(\log\log n).$$

Dem. Procederemos indutivamente, mostrando como, dado $\lfloor\sqrt{mn}\rfloor$ processadores, podemos, em dois passos de computação, reduzir o problema de intercalar duas listas de comprimento n e m, respectivamente, ao de intercalar vários pares de listas, onde a menor, em cada par, tem comprimento menor que $\sqrt{n}$. Os pares de listas são criados de tal modo que podemos distribuir os $\sqrt{mn}$ processadores entre eles no próximo estágio de modo a assegurar que para cada par será alocado um número suficiente de processadores para satisfazer a hipótese de indução.

Considere o seguinte algoritmo para intercalar as listas ordenadas $X = (x_1, \ldots, x_n)$ e $Y = (y_1, \ldots, y_m)$.

(a) Marque os elementos de X e Y cujos índices são $i\lceil\sqrt{n}\rceil$ e $i\lceil\sqrt{m}\rceil$, respectivamente, para $i = 1, 2, \ldots$. Existem no máximo $\lfloor\sqrt{n}\rfloor + \lfloor\sqrt{m}\rfloor$ elementos marcados. As sublistas entre elementos marcados sucessivos e as sublistas dos extremos são chamadas *segmentos*.

(b) Compare cada elemento marcado de X com cada elemento marcado de Y. Isto requer não mais que $\lfloor\sqrt{nm}\rfloor$ comparações e pode ser feito em uma unidade de tempo.

(c) As comparações de (b) decidirão, para cada elemento marcado, o segmento da outra lista no qual ele precisa ser inserido. Agora compare cada elemento de X com

cada elemento do segmento de Y no qual ele deve ser inserido. Isto requer no máximo $\lfloor\sqrt{n}\rfloor.(\lceil\sqrt{m}\rceil - 1) < \lfloor\sqrt{mn}\rfloor$ comparações e também pode ser feito em uma unidade de tempo.

Feito (a), (b) e (c) nós identificamos em que posição cada elemento marcado de X se encontra em Y. Portanto, resta-nos intercalar os pares disjuntos de sublistas $(X_1, Y_1), (X_2, Y_2), \ldots$ onde cada X_i é um segmento em X e, portanto, de comprimento $|X_i| \leq \lfloor\sqrt{n}\rfloor$. Além disso, $\sum|X_i| < n$ e $\sum|Y_i| < m$, pois as sublistas são disjuntas. Para aplicarmos o mesmo processo indutivamente, cada par de sublista requer $\lfloor\sqrt{|X_i|.|Y_i|}\rfloor$ processadores, o que resulta num total de $\sum\lfloor\sqrt{|X_i|.|Y_i|}\rfloor$ ao todo. Uma desigualdade de Cauchy afirma que

$$\sum\sqrt{|X_i|.|Y_i|} \leq \sqrt{\sum|X_i|.|Y_i|}.$$

Daí segue que

$$\sum\lfloor\sqrt{|X_i|.|Y_i|}\rfloor \leq \sum\sqrt{|X_i|.|Y_i|} \leq \lfloor\sqrt{mn}\rfloor.$$

Existe, portanto, número suficiente de processadores para serem distribuídos para a intercalação dos pares (X_i, Y_i) simultâneamente.

Vimos, então, que o processo indutivo de divisões sucessivas de um par de listas em um conjunto de pares de sublistas pode continuar com o número de processadores dado. Além disso, o comprimento da menor componente de cada par de sublista é indutivamente limitado superiormente pela raiz quadrada da menor componente entre o par de lista inicial. Portanto, após $2i$ unidades de tempo, cada par de listas produzidas tem um componente de comprimento não maior que λ_i onde

$$\lambda_i = \lfloor\sqrt{\lambda_{i-1}}\rfloor$$

e $\lambda_0 = n$. A solução da recorrência $\lambda_i = \sqrt{\lambda_{i-1}}$ é $\lambda_i \leq n^{(1/2)^i}$. O processo de intercalação termina localmente quando é produzido um par de sublista onde uma delas é vazia. Ou seja, $\lambda_i = 0$. Resolvendo a inequação $n^{(1/2)^i} \leq 2$ temos

$$\begin{aligned} n^{(1/2)^i} \leq 2 &\Rightarrow \frac{1}{2^i}\log n \Rightarrow \log n \leq 2^i \\ &\Rightarrow i \geq \log\log n. \end{aligned}$$

Portanto, $intercala_k(n, m) \leq O(\log\log n)$. ∎

Os corolários seguintes tratam dos casos em que $k \leq n$ e k é múltiplo de $\sqrt{mn}$.

Corolário 3.4 [Va]. Para $1 < k \leq n \leq m$,

$$intercala_k(n, m) \leq O((n + m)/k + \log((mn\log k)/k).$$

Dem. Marque $k-1$ elementos em cada lista de tal modo a obter segmentos de aproximadamente mesmo tamanho (isto é, n/k e m/k) em cada uma. Intercale as duas listas de

elementos marcados pelo algoritmo do teorema acima. Insira cada um dos $2(k-1)$ elementos marcados no segmento ao qual ele pertence na outra lista. Isto pode ser feito em tempo $\log(n/k) + \log(m/k) = O(\log(mn/k^2)$. Temos agora $2k$ pares de sublistas disjuntas para serem intercaladas nas quais nenhum par contém mais que $(n+m)/k$ elementos. Observe que o tempo gasto até agora foi $O(\log\log k) + O(\log(mn/k)) = O(\log((mn\log k)/k)$. Só nos resta determinar como estas intercalações deverão ser feitas com k processadores em tempo $(n+m)/k$ (uma melhora do que poderia ser feito em tempo $2(n+m)/k$.

A intercalação dos k pares de sublistas será feita da seguinte maneira: Associe o i-ésimo processador ao i-ésimo segmento das duas listas. Este par de segmento tem $(n+m)/k$ elementos. A cada par de sublista que tenha, digamos, z elementos em comum com este par de segmento, serão suficientes z passos do i-ésimo processador para a intercalação. Portanto, cada processador executará não mais que $(n+m)/k$ passos de computação ao todo. Além disso, como, por construção, cada sublista está contida totalmente em algum segmento, cada par de sublista será atribuído a no máximo dois processadores. Com este esquema, poderemos intercalar os $2k$ pares de sublistas em tempo $(n+m)/k$.]

Corolário 3.5 [Va]. Para $k = \lfloor r\sqrt{nm}\rfloor$ onde $1 < n \leq m$ e $r \geq 2$,

$$intercala_k(n,m) \leq O(\log\log n - \log\log r).$$

Dem. Usamos aqui o mesmo algoritmo do teorema 3.3, exceto que no passo (a) os elementos marcados são aqueles indexados por $i\lceil\sqrt{(n/r)}\rceil$ em X e por $i\lceil\sqrt{(m/r)}\rceil$ em Y, para $i = 1, 2, \ldots$. Dessa forma, os passos (b) e (c) requerem, cada um, não mais que k comparações e podem ser executados em uma unidade de tempo. Teremos agora $\lambda_i < \sqrt{(\lambda_{i-1}/r)}$, e a partir disto segue o resultado.]

3.4 - ORDENAÇÃO

Devido à importância tanto prática quanto teórica, algoritmos para ordenação têm sido objeto de intensa pesquisa em ciência da computação. Sabemos, hoje, que o limite inferior sequencial de complexidade para o problema da ordenação é $O(n\log n)$ comparações. Disto resulta, imediatamente, que o limite inferior paralelo, usando n processadores é:

$$ordena_n(n) \geq O(\log n).$$

Existem dezenas de algoritmos para ordenação em paralelo. O que apresentaremos aqui não é o que possui o melhor tempo de execução, mas é o de menor custo.

Teorema 3.5 [Va].

$$ordena_{n/2}(n) \leq O(\log n.\log\log n).$$

Dem. Este algoritmo é uma paralelização do algoritmo sequencial baseado em intercalações sucessivas, utilizando o algoritmo de intercalação do teorema 3.2. Considere, primeiramente, o caso $n = 2^j$ para algum j. O processo é executado de tal forma que,

após o i-ésimo estágio teremos 2^{j-i} listas ordenadas disjuntas, cada uma com 2^i elementos. Atribuindo 2^i processadores a cada par de listas e utilizando o algoritmo para intercalação do teorema 3.2, atingimos o próximo estágio de iteração após o tempo $O(\log i)$. A ordenação estará completa quando $i = j$. O tempo de execução do algoritmo é limitado superiormente por:

$$\sum_{i=1}^{\log n} O(\log i) \leq O(\log n . \log\log n).$$

No caso geral, quando n não é uma potência de dois, em cada estágio haverá em um extremo um fragmento de lista ordenada. O algoritmo acima se aplica da mesma maneira neste caso. ∎

Corolário 3.6 [Va]. Para $4 \leq 2n < k = rn \leq n(n-1)/2$,

$$ordena_k(n) \leq O((\log n - \log(k/n))(\log\log n - \log\log(k/n))).$$

Dem. Divida o conjunto inicial em subconjuntos com k/n elementos cada um. Atribua $(k/n)^2$ processadores a cada subconjunto e ordene-os em $\log(k/n)$ passos de computação (compare cada elemento com todos os outros em um passo de computação, e determine a sua posição em mais $\log(k/n)$ passos). Agora aplique $\log n - \log(k/n)$ vezes o algoritmo de intercalação do corolário 3.5. ∎

Corolário 3.7 [Va]. Para $1 < k < n$,

$$ordena_k(n) \leq O((n\log n)/k + \log k . \log((n\log k)/k)).$$

Dem. Divida o conjunto inicial em k subconjuntos de tamanhos iguais e ordene cada um deles sequencialmente em tempo $O(n/k\log(n/k))$. Em $\log k$ estágios intercale sucessivamente cada par de subconjunto ordenado, usando o algoritmo do corolário 3.4. A cada estágio haverá o dobro de processadores disponíveis para cada intercalação em relação ao estágio anterior. Assim, o tempo de execução em cada estágio será de aproximadamente n/k. O tempo total, gasto para as intercalações, será, então, de

$$\log k(\frac{n}{k} + \log((n^2\log k)/k)) = O((n\log n)/k + \log k . \log((n\log k)/k)).$$ ∎

3.5 - COMENTÁRIOS FINAIS

Nos três problemas estudados neste capítulo, observamos uma relação entre tempo de execução e número de processadores utilizados, que é comum na solução de problemas em paralelo: Quanto maior for o número de processadores utilizados, menor é o tempo de execução. Como já dissemos na seção 3.1, o que se busca, quando aumentamos o número de processadores de n, é uma diminuição no tempo de execução de $1/n$.

Outro algoritmo para intercalação é encontrado em [G]. Este algoritmo é baseado em inserções binárias e requer tempo $2\log(n+1)+4(n/p)$, onde p é o número de processadores, $p \leq n$ e $n = m$.

Apêndice

Introdução a Modelos de Computação Paralela

A.1 - INTRODUÇÃO

Uma motivação para a pesquisa em *Computação Paralela* é que podemos encontrar crescimentos substanciais na velocidade de processamento sobre os modelos clássicos sequenciais, na solução de problemas computacionais. Uma grande variedade de modelos de Computação Paralela têm sido definidos, e é interessante estudar as relações que existem entre eles, a viabilidade de construção de cada um, e quais outros modelos de Computação Paralela podem ser imaginados e construídos.

Os modelos de Computação Paralela diferem entre si de várias maneiras, incluindo o arranjo topológico e a interconexão de seus componentes (processadores). No Modelo Sequencial de Computação, o conjunto de instruções válidas é pré-definido pelo processador. Nos Modelos de Computação Paralela, além das instruções internas a cada processador, existem as instruções inter-processadores. Estas instruções não estão pré-definidas, e ainda não se sabe qual a melhor maneira de definí-las para obter um "bom" Modelo de Computação Paralela. Basicamente, a partir das instruções inter-processadores é que são definidos os Modelos de Computação Paralela. Estas formas de interconexão entre os processadores são também importantes na determinação do potencial de Computação Paralela, e são determinadas pelas instruções dos Algoritmos Paralelos. Por exemplo, a instrução "se o processador P_j contém o valor k então armazene j no processador P_k" exige que cada processador tenha acesso a todos os outros, enquanto que a instrução "P_j toma o valor que está em P_{2j} e P_{2j+1}, compara-os, e retém o maior" exige que cada processador P_j tenha acesso somente aos processadores P_{2j} e P_{2j+1}. Ou seja, é a partir de instruções destes tipos que determinamos a interconexão entre os processadores. portanto, modelos diferentes de Computação Paralela existem, devido aos Algoritmos Paralelos de natureza diferente.

Entre os principais modelos de Computação Paralela estão os modelos SIMD, SIMDAG, MÁQUINA VETORIAL e MÁQUINA DE TURING ALTERNADA. O nosso objetivo neste capítulo é descrever brevemente estes modelos.

A.2 - MODELO SIMD

O modelo SIMD [BWHM] (Single Instruction Stream, Multiple Data Stream), também chamado modelo de Interconexão fixa, é um sistema composto por uma unidade de controle (CPU) e um conjunto de processadores com memória local, interconectados por uma rede de interconexão (figura A.1). Os processadores são altamente sincronizados (ou seja, todos os processadores devem completar a execução da instrução corrente antes de qualquer processador começar a próxima instrução). A CPU contém o programa a ser executado e distribui as instruções "em paralelo" para os demais processadores.

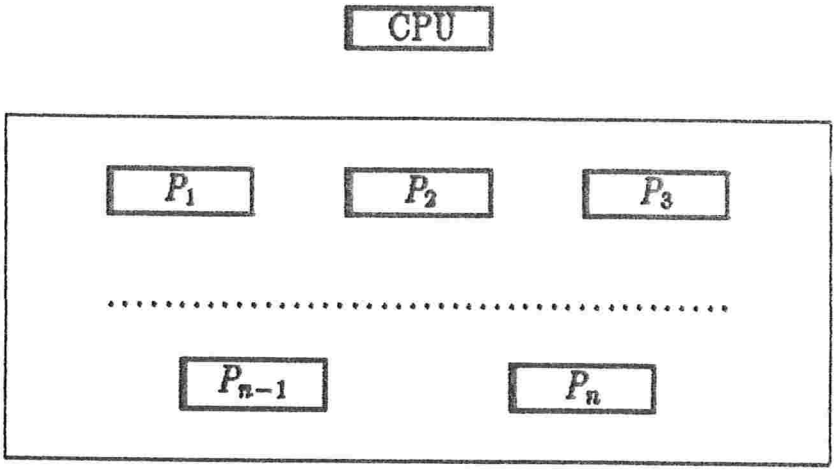


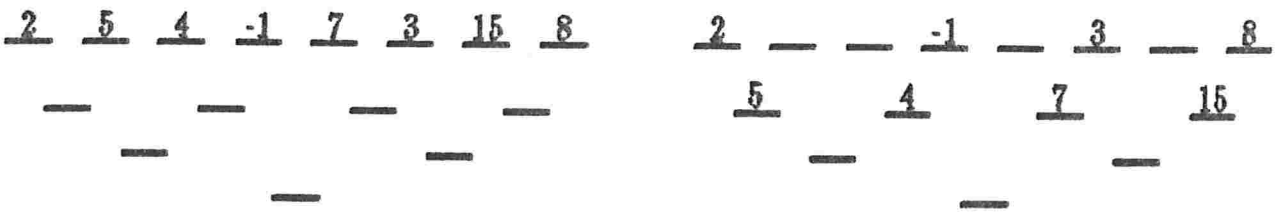
Fig. A.1 - Modelo SIMD

Um processador paralelo está ativo somente quando alguma instrução é delegada a ele. Dois processadores paralelos devem estar executando a mesma instrução se ambos estão ativos ao mesmo tempo (Single Instruction). No entanto, eles podem operar sobre valores distintos (Multiple Data).

Exemplos de algoritmos baseados no modelo SIMD:

Exemplo 1: Usando uma estrutura de árvore binária, podemos determinar o maior entre n números dados, iterativamente, comparando cada dois números vizinhos e movendo o maior para o nó pai. Se executarmos simultaneamente todas as comparações ao mesmo nível na árvore binária, temos um algoritmo em paralelo para determinar o máximo entre n elementos.

Mais precisamente, considere um conjunto de $2n - 1$ processadores interconectados em forma de uma árvore binária com um processador em cada uma das n folhas e em cada nó interior da árvore. Começando com um número em cada processador que está na folha,



1º passo 2º passo

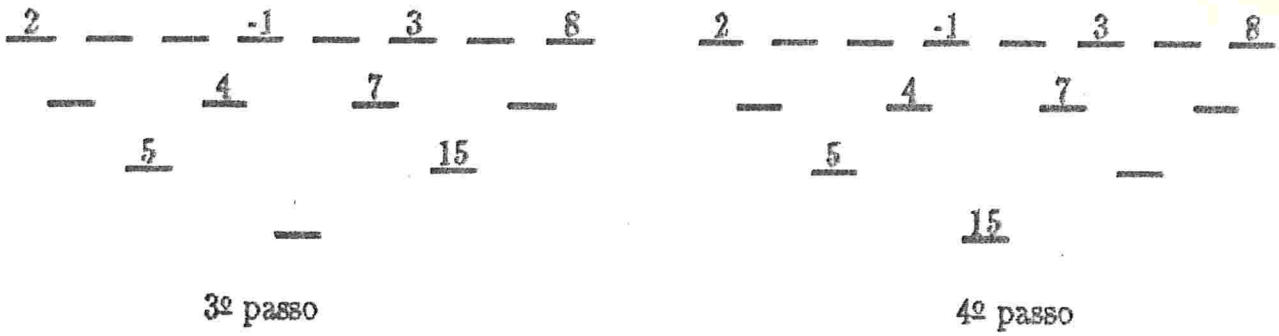


Fig. A.2 - Algoritmo para o modelo SIMD

a cada passo, um pai recebe um elemento de cada um de seu(s) filho(s), compara-os, retém o maior, e devolve o menor (veja figura acima). Dessa maneira o máximo pode ser transferido até a raiz em $\log_2 n$ operações, em paralelo, de comparação e transferência. Portanto, por este algoritmo podemos resolver o problema de achar o máximo de n elementos em tempo (paralelo) $O(\log n)$ usando $2n-1$ processadores.

Na seção 3.2 resolvemos este mesmo problema em tempo $O(\log\log n)$ usando n processadores. Porém, no modelo utilizado lá (Modelo de Memória Compartilhada), o acesso à memória de um processador por parte dos outros é irrestrito, enquanto que aqui há restrição. Informalmente, isso justifica a ineficiência relativa deste algoritmo.

Exemplo 2: Algoritmo de Batcher [Ba] para ordenação, apresentado no capítulo II.

A.3 - MODELO SIMDAG

O modelo SIMDAG [Go] (Single Instruction Stream, Multiple Data Stream, Global Memory) consiste de uma memória global de acesso aleatório, Um processador de controle (CPU), e um conjunto de unidades de processamento paralelo (UPP), arranjados como segue:

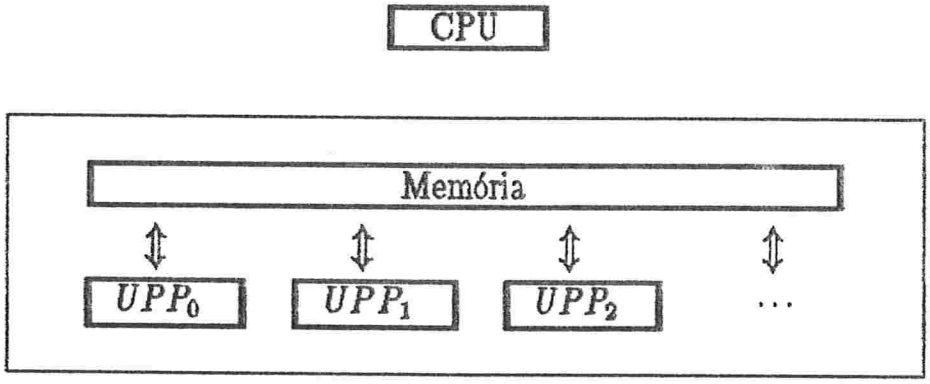


Fig. A.3 - Modelo SIMDAG

(Este modelo é um caso particular dos *Modelos de Memória Compartilhada (Shared Memory Model)* [BWHM]). Todos os processadores são máquinas RAM (Random Access Machine) com o conjunto de instruções usuais acrescido pelas instruções paralelas.

Assim, como no modelo SIMD, a CPU contém as instruções da máquina, executa as instruções "em série" e as distribui "em paralelo" para as UPP's. Um processador paralelo

está ativo somente quando uma instrução é atribuída a ele. Dois processadores ativos ao mesmo tempo devem estar executando a mesma instrução, e no entanto, podem estar operando sobre diferentes valores.

Ao contrário do modelo anterior, neste modelo não é permitido a interconexão entre os processadores.

Como exemplo de algoritmo baseado no modelo SIMDAG podemos citar um algoritmo de ordenação. Este algoritmo, derivado do algoritmo de Knuth ([Kn], pag. 224), é uma paralelização do Bubblesort. Suponha que os n elementos $x_1, x_2, \ldots, x_n$ a serem ordenados estão colocados nas n primeiras posições da Memória Global. Considere $n-1$ processadores $P_1, P_2, \ldots, P_{n-1}$ dispostos linearmente. Suponha que cada processador P_i tenha acesso apenas às posições de memória i e $i+1$. Este algoritmo é composto de n fases. Em cada fase são executadas $n/2$ comparações simultâneas. Durante uma fase ímpar, os elementos $(x_1, x_2), (x_3, x_4), \ldots$ são comparados (e trocadas as posições caso seja necessário) pelos processadore $P_1, P_3, \ldots$ respectivamente. durante uma fase par, os elementos $(x_2, x_3), (x_4, x_5), \ldots$ são comparados pelos processadores $P_2, P_4, \ldots$ respectivamente (veja fig. abaixo).

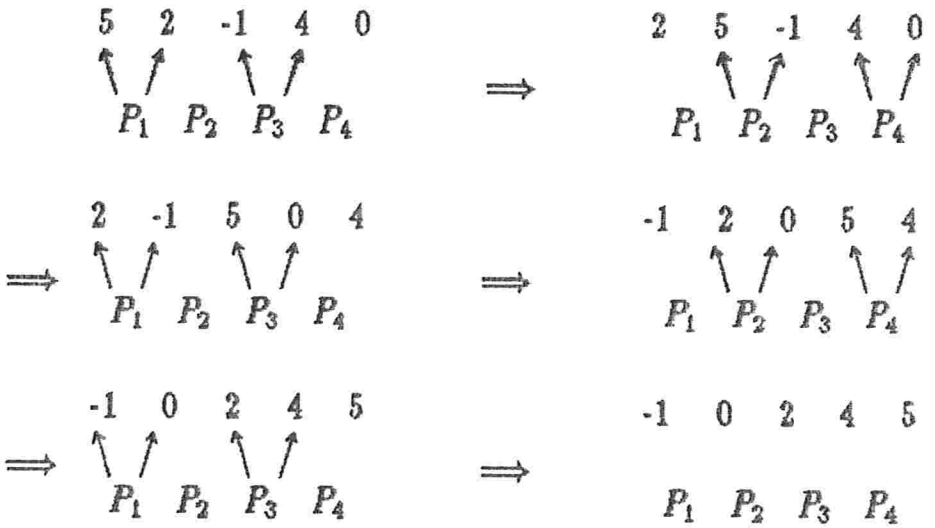


Fig. A.4 - Algoritmo de ordenação para o Modelo SIMDAG.

Pode-se demonstrar que n fases são suficientes para ordenar a sequência dada. Através deste algoritmo podemos, então, resolver o problema da ordenação em tempo paralelo $O(n)$ usando $n-1$ processadores.

A.4 - MODELO MÁQUINA VETORIAL

Uma máquina vetorial [PS] é um modelo que possui uma memória que é um conjunto de vetores de bits arbitrariamente longos e cujo programa é um grafo dirigido finito com um vértice inicial e um conjunto de vértices de aceitação. Os vértices do grafo são formados por cada uma das seguintes instruções ou asserções:

(i) $A \leftarrow$ *Constante*, uma instrução para armazenar um vetor de bit constante no registro A.

(ii) $A \leftarrow \sim B$ e $A \leftarrow B \wedge C$, operações booleanas bit a bit em paralelo.

(iii) $A \leftarrow B \uparrow C$ $(A \leftarrow B \downarrow C)$, *deslocamento de B para a esquerda (direita) de C* posições. No deslocamento para a esquerda as posições vagas são preenchidas com zeros, e no deslocamento para a direita os bits que caem fora do registro são descartados.

(iv) $A = 0$ *e* $A \neq 0$.

É adotada a tradicional convenção de escrever a sequência de bits da direita para a esquerda.

Uma *Computação Aceitável* na Máquina Vetorial é um passeio que vai do vértice inicial até algum vértice de aceitação.

Certos registros são designados como *registros de entrada*. Se um programa menciona m registros $A_1, A_2, \ldots, A_m$, então uma *configuração* consiste de um nó do grafo e vetores de bits $v_1, v_2, \ldots, v_m$ que especificam o conteúdo atual dos registros $A_1, A_2, \ldots, A_m$.

Uma máquina vetorial *determinística* é tal que para qualquer vértice e qualquer configuração apenas uma aresta pode ser seguida. Portanto, em uma máquina vetorial determinística o único vértice com mais de uma aresta de saída é aquele com as asserções $A = 0$ *e* $A \neq 0$.

No programa exemplo abaixo usamos também as operações boleanas $\vee$ e $\oplus$ (ou exclusivo), porém, elas podem ser derivadas de $\wedge$ e $\sim$ pelas identidades:

$$A \vee B = \sim ((\sim A) \wedge (\sim B))$$
$$A \oplus B = (A \vee B) \wedge (\sim (A \wedge B))$$

Exemplo de programa em Máquina Vetorial: Soma de dois números A e B em binário.

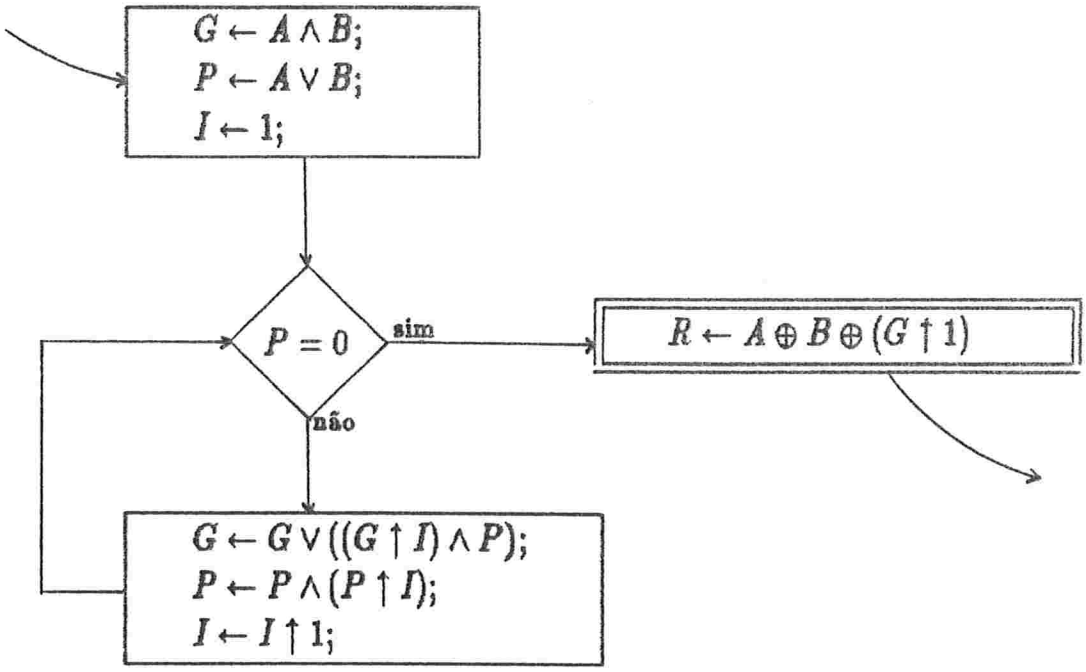


Fig. A.5 - Programa em Máquina Vetorial para somar dois números.

No final da execução a soma de A e B está armazenada no registro R.

O laço no programa calcula o vetor G que é o vetor de "vai um". Pode-se demonstrar que a complexidade de tempo deste algoritmo é $O(\log n)$ [PS].

A.5 - MODELO MÁQUINA DE TURING ALTERNADA (MTA)

Alternação é uma generalização do conceito de *não-determinismo* no qual quantificadores existenciais e universais podem alternar durante uma computação.

O conceito de *não-determinismo* tem desempenhado importantes papéis na Teoria da Computação. Por exemplo, na Teoria de Linguagens Formais, certas classes de linguagens têm sido caracterizadas em termos de aceitação por autômatos não-determinísticos; Em teoria da complexidade, o conceito de não determinismo foi a base para a importante noção de completicidade *NP*.

No caso de máquinas não determinísticas, as regras de transição permitem que uma configuração α dê origem a várias configurações $\beta_1, \ldots, \beta_k$ em um passo de computação; Por definição, a configuração α é *aceita* se e somente se existe um sucessor β_i que leva a uma aceitação. Além desses "estados existenciais", uma MTA pode também incluir "estados universais". Em um estado universal, uma configuração α pode novamente dar origem às configurações $\beta_1, \ldots, \beta_k$ em um passo de computação, mas agora α é aceita se e somente se todos os sucessores $\beta_1, \ldots, \beta_k$ são aceitos. Uma formalização desta idéia resulta na definição da Máquina de Turing Alternada (MTA).

Definição. Uma *Máquina de Turing Alternada* [CS] é uma 7-upla

$$M = (K, Q, \Sigma, \Gamma, \delta, q_0, g)$$

onde:

K é o número de fitas de trabalho,
Q é um conjunto finito de estados,
Σ é um alfabeto de entrada ($c \notin \Sigma$ é uma marca de fim de cadeia),
Γ é um alfabeto de trabalho ($\# \in \Gamma$ representa um espaço em branco),
$\delta \subseteq (Q \times \Gamma^k \times (\Sigma \cup \{c\})) \times (Q \times (\Gamma - \{\#\})^k \times \{L, R\}^{k+1})$ é a regra de transição,
$q_0 \in Q$ é o estado inicial,
$g : Q \rightarrow \{\wedge, \vee, a, r\}$.

Se $g(q) = \wedge$ ($\vee$) então q é dito ser um estado universal (existencial).

A MTA tem uma fita de entrada onde é permitido somente a leitura, e K fitas de trabalho, inicialmente vazias. Um *passo de computação em* M consiste em ler um símbolo de cada fita, escrever um símbolo em cada fita de trabalho, mover cada cabeça de leitura uma célula para a esquerda ou direita, e entrar num novo estado de acordo com a relação de transição δ.

Definição. Uma *configuração* de uma MTA M é um elemento do tipo

$$C_M = Q \times \Sigma^* \times ((\Gamma - \{\#\})^*)^k \times N^{k+1}$$

Representando o estado atual, a entrada, o conteúdo das K fitas de trabalho, e as $K+1$ posições da cabeça de leitura, respectivamente. Se q é um estado associado a uma

configuração α, então dizemos que α é *configuração universal* (*existencial*) se q é um estado universal (existencial). A *configuração inicial* de M com entrada x é

$$\sigma_M(x) = (q_0, x, \underbrace{\lambda, \ldots, \lambda}_{k}, \underbrace{0, \ldots, 0}_{k+1})$$

Onde λ é a palavra vazia.

Uma configuração representa uma descrição instantânea em algum ponto, em uma computação.

Definição. Dada uma MTA M, dizemos que β é *sucessor* de α ($\alpha \mapsto \beta$) se a configuração β é obtida de α em um passo de computação de M de acordo com a regra de transição δ. O *fecho reflexivo transitivo* de $\mapsto$ é denotado por $\mapsto^*$.

Uma *sequência de computações* de M sobre x é uma sequência $\alpha_0 \mapsto \alpha_1 \mapsto \cdots \mapsto \alpha_n$, onde $\alpha_0 = \sigma_M(x)$.

A definição de aceitação para uma MTA é feita de modo a capturar a seguinte idéia: O processo de computação se inicia na configuração inicial $\sigma(x)$. A seguir, se o processo se encontra em uma configuração existencial α, e $\beta_1, \ldots, \beta_m$ são todas as configurações que podem ser alcançadas a partir de α em um passo de computação, então este processo se subdivide em m processos distintos que buscam simultaneamente uma aceitação para cada β_i. Alguns destes processos podem ser infinitos, mas se pelo menos um é aceito, então α também é aceita. Se o processo se encontra em uma configuração universal α', então α' é aceita se todos os seus sucessores $\beta_1, \ldots, \beta_k$ são aceitos. A formalização desta idéia não será feita aqui, mas pode ser encontrada em [CKS].

Um exemplo de aplicação de MTA encontra-se no estudo de jogos combinatoriais de duas pessoas; O problema aqui é estabelecer limites de complexidade computacional para reconhecer o conjunto de posições a partir da qual um jogador (o primeiro, por exemplo) sempre vence o jogo. este problema pode ser resolvido explorando a conexão natural existente entre MTA's e jogos que pode ser resumida no seguinte: As configurações existenciais (universais) da MTA correspondem a posições a partir da qual o jogador I (jogador II) tem a vez para jogar. Para estes problemas, as configurações da MTA podem ser vistas como uma árvore onde para uma configuração α os seus sucessores $\beta_1, \ldots, \beta_k$ são seus filhos (veja exemplo abaixo). Logo, a MTA pode também ser considerada como uma máquina que trabalha em paralelo onde os filhos $\beta_1, \ldots, \beta_k$ de α buscam uma aceitação simultânea. Comunicações ocorrem somente entre pais e filhos. após ter sido aceito ou rejeitado, cada filho β_i comunica o resultado ao pai α. α combina as respostas dependendo do seu rótulo existencial ou universal, e comunica, por sua vez, o resultado ao seu pai.

Para ilustrar esta idéia, considere os jogos sobre fórmulas proposicionais. Nestes jogos, a posição inicial é uma fórmula proposicional e uma atribuição de valores lógicos às variáveis boleanas. Um movimento no jogo é a troca do valor lógico de alguma variável, e o vencedor é o que faz primeiro a fórmula ter como resultado o valor lógico verdadeiro.

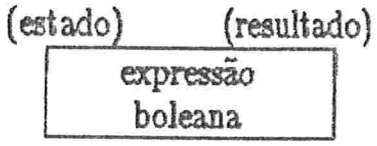
Suponha que a fórmula seja

$$L = (x \vee z) \wedge (x \wedge y)$$

e que a atribuição de valores seja:

$$x = F, \quad y = F, \quad z = F \quad (F = falso, T = verdadeiro).$$

Relembre que o que queremos é determinar um conjunto C a partir do qual o jogador I sempre vence o jogo. Vamos supor que o jogador I inicia o jogo. A árvore de configurações para esta fórmula está apresentada na figura 1.6 onde cada nó tem a seguinte notação:



A variável *"estado"* assume valores no conjunto (e,u,a,r) e nos informa se este estado é existencial (e), universal (u), estado final de aceitação (a) ou rejeição (r). A variável *"resultado"* assume valores no conjunto (F,T) e nos informa se o resultado da expressão booleana é falso (F) ou verdadeiro (T).

O sinal ⊗ do lado esquerdo do nó significa que esta configuração pertence ao conjunto C.

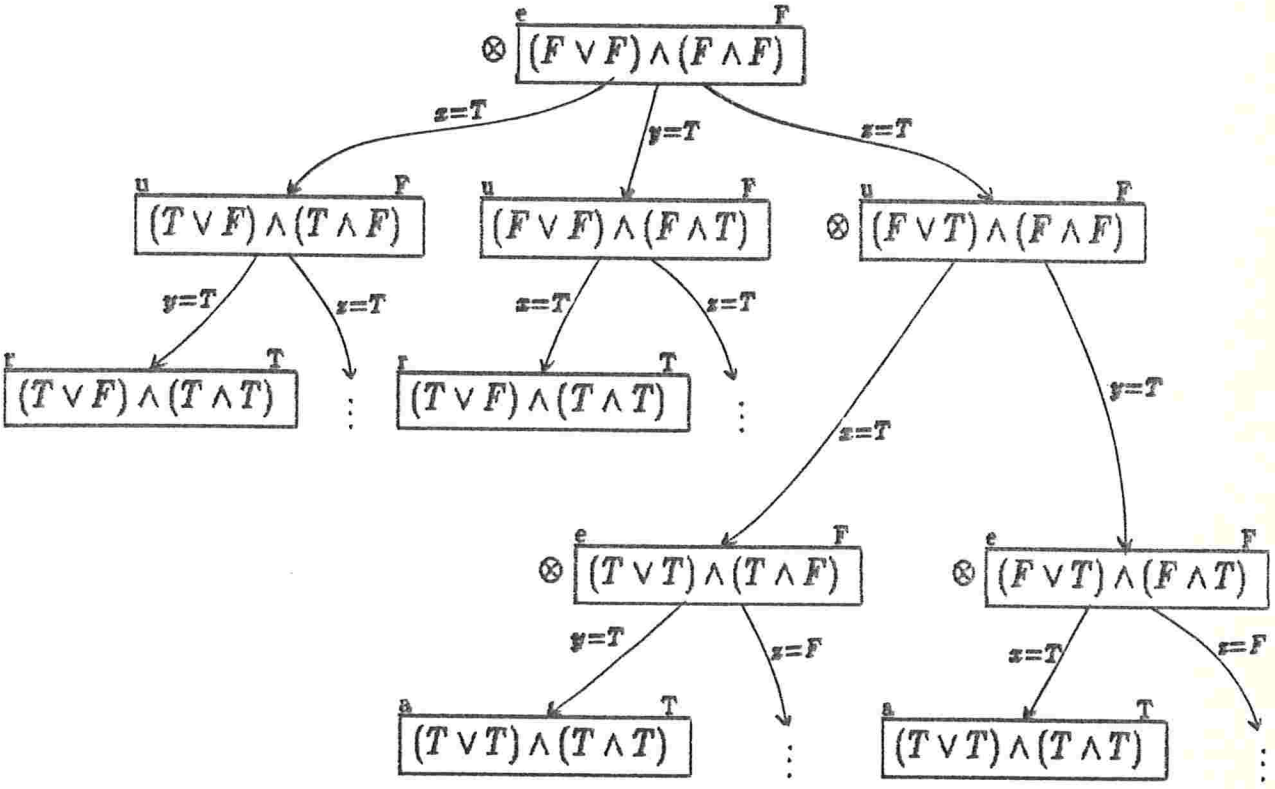


Fig A.6 - árvore de configurações para a fórmula $L = (x \vee z) \wedge (x \wedge y)$

Observe que neste exemplo, como a configuração inicial pertence a C, existe uma estratégia que garante que o jogador I sempre ganhe o jogo.

A.6 - COMENTÁRIOS FINAIS

Nas seções anteriores introduzimos quatro modelos de Computação Paralela. Existem outros ainda. Como já foi dito, um grande objetivo a ser atingido é a determinação de um

bom modelo com relação a viabilidade de construção e implementação de algoritmos. No momento, o modelo SIMD é um candidato mais forte do que os outros, pois, sendo um modelo de Interconexão Fixa, sua viabilidade de construção parece ser maior atualmente. Nos modelos mais recentes, conhecidos como Modelo de Memórias Compartilhadas (nesta classe se encontra o Modelo SIMDAG), cada processador tem acesso a memória dos outros processadores. A implementação de um modelo deste tipo não é viável a curto praso [BWHM].

Na busca por um bom modelo foram encontradas equivalências entre modelos já existentes e simulação de alguns modelos por outros. Ruzzo [Ru] mostrou a equivalência entre os modelos Circuitos Booleanos Uniformes e Máquina de Turing Alternada. Mais precisamente, ele mostrou que qualquer um desses modelos pode ser simulado pelo outro em tempo linear.

Outro fato importante a respeito de Modelos de Computação é a chamada *Tese da Computação Paralela* que se resume no seguinte: Tempo em Máquina Paralela está polinomialmente relacionado a Espaço em Máquina de Turing. Mais precisamente,

Tese da Computação Paralela [Go]. Para qualquer função T(n),

$$\bigcup_k Tempo - Paralelo(T^k(n)) = \bigcup_k Espaco(T^k(n)),$$

onde k é inteiro positivo. Esta afirmação não foi demonstrada para o caso geral, mas já se sabe que os modelos SIMD, SIMDAG, MÁQUINA VETORIAL e MTA satisfazem esta tese [Go].

Uma análise dos algoritmos apresentados nos capítulos anteriores mostra que: O modelo subjacente aos algoritmos dos capítulos I e II é o modelo SIMD, enquanto que aos algoritmos do capítulo III, o modelo subjacente é o SIMDAG.

BIBLIOGRAFIA

[A] Ajtai, M., Komlós, J. and Szemerédi, E. (1983). *An $O(n \log n)$ sorting network.* In Proceedings of The 15th Annual ACM Symposium on Theory of Computing (Boston, apr. 25–27). ACM, New York, pp. 1–9.

[Ak] Akl, S. G. (1985). *Parallel sorting algorithms.* Notes and Report in Computer Sciense and Applied Mathematics, Academic Press Inc.

[AHU] Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass.

[Ba] Batcher, K. E. (1968). *Sorting networks and their applications.* In Preceedings of The 1968 Spring Joint Computer Conference (Atlantic City, N. J., apr. 30–may 2), vol. 32, AFIPS Press, Reston, Va., pp. 307–314.

[Be] Berkowitz, S. J. (1984). *On computing the determinant in small parallel time using a small number of processors.* Information Processing Letters 18, pp. 147–150.

[Bg] Berge, C. (1973). *Graphs and Hipergraphs.* North-Holland, London.

[Bo] Borodin, A. (1977). *On relating time and space to size and depth.* SIAM J. Comput., 6, pp. 733–744.

[BCP] Borodin, A., Cook, S. A. and Pippenger, N. (1983). *Parallel computation for well-endowed rings and space-bounded probabilistic machines.* Information and Control 58, pp. 113–136.

[BH] Borodin, A. and Hopcroft, J. E. (1982). *Routing, merging and sorting on parallel models of computation.* In Proceedings of The 14th Annual ACM Symposium on Theory of Computing (San Francisco, Calif., may 5–7). ACM, New York, pp. 338–344.

[BM] Bondy, J. A. and Murty, U. S. R. (1976). *Graph Theory with Applications.* The Macmillan Press LTD., London.

[BWHM] Bitton, D., DeWitt, D. J., Hsiao, D. K. and Menon, J. (1984). *A taxonomy of parallel sorting.* Computing surveys, vol. 16, No. 3, pp 287–318.

[Co] Cook, S. A. (1984). *A taxonomy of problems with fast parallel algorithms.* Technical Report da Universidade de Toronto.

[CKS] Chandra, A. K., Kozen, D. C. and Stockmeyer, L. J. (1981). *Alternation.* J. ACM vol. 28, No. 1, pp. 114–133.

[CW] Coppersmith, D. and Winograd, S. (1982). *On the asymptotic complexity of matrix multiplication.* SIAM J. Comput., vol. 11, No. 3, pp. 472–492.

[G] Gavril, F. (1975). *Merging with parallel processors.* Commun. ACM vol. 18, No. 10 (oct.), pp. 588–591.

[Go] Goldschlager, L. M. (1982). *A universal interconnection pattern for parallel computers.* J. ACM, vol. 29, No. 3 (july), pp. 1073–1086.

[H] Hirschberg, D. S. (1978). *Fast parallel sorting algorithms.* Commun. ACM vol. 21, No. 8 (aug.), pp. 657–666.

[HU] Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages and Computation.* Reading, Addison-Wesley.

[Ke] Kemp, R. (1984). *Fundamentals of the Analysis of Particular Algorithms.* Wiley - Teubner Series in Computer Science.

[Kn] Knuth, D. E. (1973). *The Art of Computer Programming. Sorting and Searching.* vol. 3, Addison-Wesley, Reading, Mass.

[MP] Muller, D. E. and Preparata, F. P. (1975). *Bounds to complexities of networks for sorting and for switching.* J. ACM, vol. 22, No. 2 (apr.), pp. 195–201.

[Pr] Preparata, F. P. (1978). *New parallel sorting schemes.* IEEE Trans. Comput., C-27,7 (july).

[PS] Pratt, V. R. and Stockmeyer, L. J. (1976). *A characterization of the power of vector machines.* Journal of Computer and System Sciences, vol. 12, pp. 198–221.

[Ru] Ruzzo, W. L. (1981). *On uniform circuit complexity.* J. Comput. and System Sciences, vol. 22, No. 3, pp. 365–383.

[Sa] Savage, J. E. (1976). *The Complexity of Computing.* Wiley, New York.

[Se] Sedgewick, R. (1978). *Data movement in odd-even merging.* SIAM J. Comput., Vol. 7, pp. 239–273.

[St] Stone, H. S. (1971). *Parallel processing with the perfect shuffle.* IEEE Trans. Comput. C-20,2 (feb.).

[TK] Thompson, C. D. and Kung, H. T. (1977). *Sorting on a mesh-connected parallel computer.* Commun. ACM, vol 20, No. 4 (apr.), pp. 263–271.

[Va] Valiant, L. G. (1975). *Parallelism in comparison problems.* SIAM J. Comput., vol. 3, No. 4 (sept.).