


# CT-234



## Estruturas de Dados, Análise de Algoritmos e Complexidade Estrutural

**Carlos Alberto Alonso Sanches**

CT-234



# 11) Algoritmos paralelos

Conceitos básicos, PRAM, Hipercubo

# Motivações

- O desempenho dos *processadores sequenciais* (também chamados de *seriais*) está cada vez mais próximo do limite físico, imposto pela velocidade da luz, pela dissipação de calor e pelas dimensões dos componentes.
- Este desempenho é insuficiente em muitas áreas:
  - Química quântica e física relativista
  - Cosmologia e astrofísica
  - Dinâmica de fluidos e turbulência
  - Sequenciamento de genomas e síntese de proteínas
  - Climatologia global
  - Reconhecimento de voz em tempo real
  - Criptoanálise

# Paralelismo



- Em um sistema paralelo, vários processadores cooperam simultaneamente na resolução de um mesmo problema.
- Esta alternativa vem se tornando cada vez mais viável graças ao declínio do custo e do tamanho dos componentes eletrônicos.
- Além do aumento de velocidade, o paralelismo também proporciona a habilidade de "estar em mais de um lugar ao mesmo tempo". Esta característica é conveniente, por exemplo, para problemas com múltiplos fluxos de dados, ou em situações em que uma computação já realizada não pode ser revertida.

# Paralelismo *versus* Sistema Distribuído

## Sistema Paralelo

Múltiplos processadores

Tarefas simultâneas

Acoplamento forte

Procura-se a aceleração de uma única aplicação

Arquiteturas homogêneas

Possibilidade de uso de memória compartilhada

## Sistema Distribuído

Computadores autônomos

Distribuição de tarefas

Recursos fisicamente distantes

Há várias aplicações e usuários

Estruturas heterogêneas e dinâmicas

Nunca se utiliza memória compartilhada

# Modelos de paralelismo



- É praticamente impossível falar de algoritmos paralelos sem antes estabelecer um modelo de computação.
- Ao contrário do caso sequencial, onde os computadores geralmente pertencem ao modelo de *Von Neumann*, em paralelismo há uma grande diversidade.
- Basicamente, são duas características que os diferenciam:
  - Presença ou não de sincronismo entre os processadores.
  - Modo de comunicação entre eles (rede de interconexão ou memória compartilhada).

# Fluxos de instruções e de dados

- Todo computador (sequencial ou paralelo) executa um conjunto de instruções sobre dados de entrada:
  - Fluxo de instruções (programa)
  - Fluxo de dados (entrada)
- Quando o fluxo de instruções é o mesmo para todos os processadores, podemos dizer (embora isso não seja necessário) que eles trabalham *sincronamente*.
- Em 1966, Flynn estabeleceu uma classificação que se tornou tradicional, embora existam nomenclaturas alternativas.

# Classificação de Flynn



- *SISD (Single Instruction / Single Data)*
  - Modelo sequencial (conhecido como *Von Neumann*)
- *SIMD (Single Instruction / Multiple Data)*
  - Modelo paralelo síncrono
- *MIMD (Multiple Instruction / Multiple Data)*
  - Modelo paralelo assíncrono
- *MISD (Multiple Instruction / Single Data)*
  - Inexistente na prática

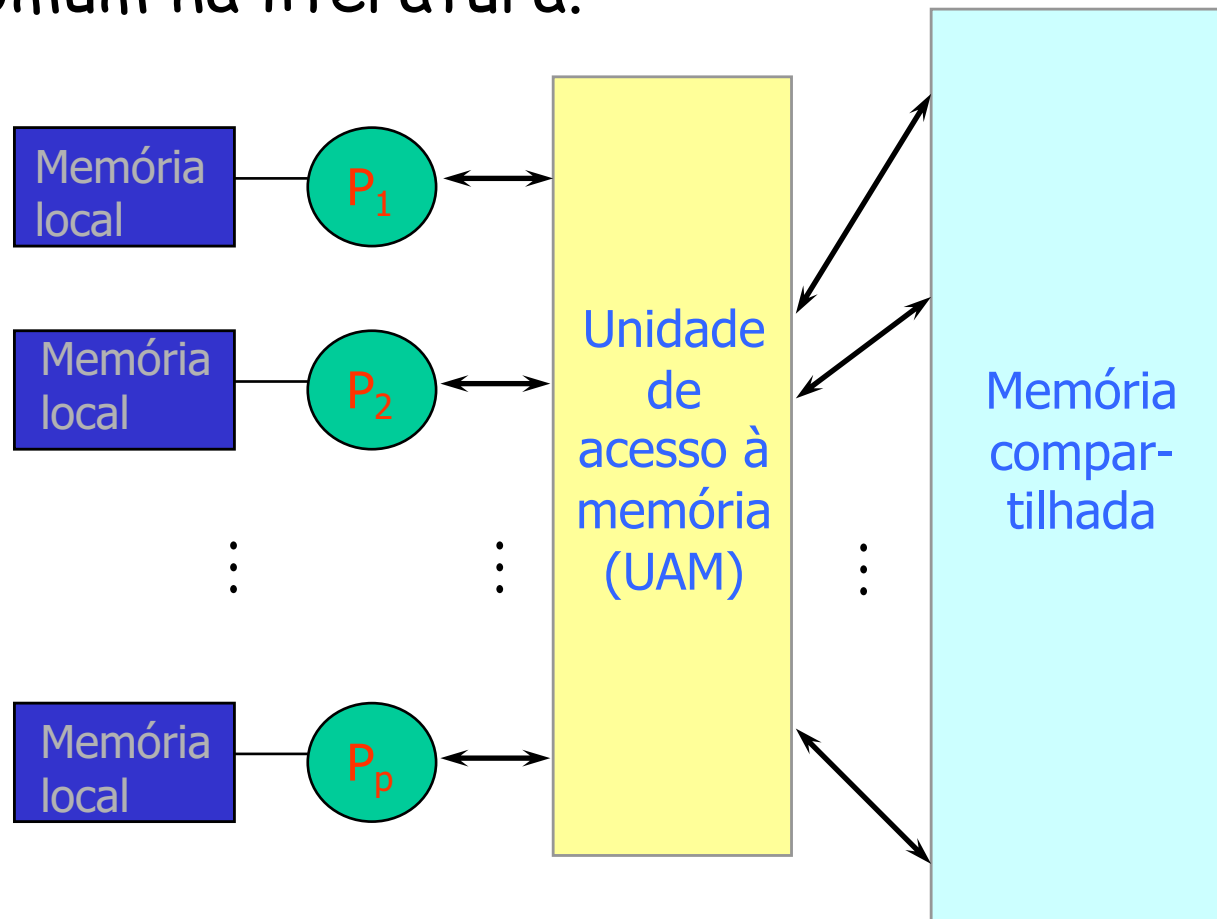


# Análise de algoritmos paralelos

- Passos elementares no tempo de execução paralelo:
  - *Computacionais* : idem ao modelo sequencial.
  - *De roteamento* : transmissão de um dado de tamanho constante de um processador a outro.
- Custo ( $C = p \cdot T_p$ ): produto entre o número  $p$  de processadores e o tempo paralelo  $T_p$  de execução.
  - Estabelece um limite máximo para o número de passos elementares.
  - Portanto, mede o tempo máximo que um computador serial gastaria na simulação desse algoritmo paralelo.
  - Uma paralelização será ótima se seu custo for da mesma ordem do *upper-bound*  $T_s$  de tempo sequencial do mesmo problema:  $C = p \cdot T_p = \Theta(T_s)$ .
  - Não faz sentido  $T_p = o(T_s/p)$ . Por quê?
- Escalabilidade: capacidade de um algoritmo paralelo resolver eficientemente uma mesma instância de um problema em máquinas com diferentes números de processadores.

# Modelo PRAM

- *Parallel Random Access Machine (PRAM)*: modelo paralelo muito comum na literatura.



- É geralmente definida como SIMD (síncrona).

# Passos de execução na PRAM

- Na PRAM, cada passo de um algoritmo consiste em três fases:
  - Leitura: até  $p$  processadores leem simultaneamente uma correspondente posição da memória compartilhada, e armazenam esse valor em um registro local.
  - Computação: até  $p$  processadores executam uma operação aritmética ou lógica em seus dados locais.
  - Escrita: até  $p$  processadores escrevem simultaneamente em uma correspondente posição da memória compartilhada.
- "Até  $p$  processadores" significa que eventualmente alguns deles permanecem inativos.

# Acesso à memória compartilhada

- ER (*Exclusive Read*) e EW (*Exclusive Write*)
  - Não há acessos simultâneos a uma mesma posição da memória compartilhada.
- CR (*Concurrent Read*)
  - São permitidos acessos simultâneos à memória compartilhada para leitura.
- CW (*Concurrent Write*)
  - São permitidos acessos simultâneos à memória compartilhada para escrita. Neste caso, é preciso estabelecer uma política de tratamento para os eventuais conflitos.

# Variantes das instruções CW



- *Priority* : somente escreve o processador de maior prioridade.
- *Common* : a escrita somente é permitida se os processadores desejarem escrever um mesmo valor.
- *Arbitrary* : o algoritmo estabelece qual processador irá escrever.
- *Random* : o processador que irá escrever é escolhido aleatoriamente.
- *Combining* : todos os valores destinados a uma mesma posição são combinados através de uma função determinada (*sum, and, or, max, min, etc.*).

# Comandos na PRAM SIMD

- De modo geral, a descrição de algoritmos em uma PRAM SIMD é muito semelhante ao utilizado no modelo sequencial.
- Na verdade, `forall where` é o único comando novo:
  - Especifica um intervalo de processadores que executarão simultaneamente um bloco de comandos.
  - A cláusula `where`, que é opcional, indica uma restrição neste intervalo.
  - Embora pareça, este comando não é um laço...
- Durante a execução dos comandos internos ao `forall`, os eventuais processadores excluídos pela cláusula `where` permanecem inativos.
- Os algoritmos que apresentaremos a seguir, por serem elaborados em uma PRAM SIMD, utilizam este comando.

# Posição do primeiro 1

- Dada uma sequência de  $n$  bits  $B[1..n]$ , deseja-se encontrar a posição do primeiro valor 1.
- Sabemos que a solução sequencial desse problema gasta tempo  $T_s = O(n)$ .
- Na PRAM CRCW-or com  $n$  processadores (aplica o operador or em caso de conflito de escrita), seria possível resolver esse problema em tempo constante?
- Inicialmente, elaboraremos um algoritmo paralelo de tempo  $T_p = \Theta(1)$  utilizando  $p = n^2$  processadores.
  - Como  $C = p \cdot T_p = \omega(T_s)$ , essa paralelização não é ótima...
- Em seguida, esse algoritmo será utilizado na elaboração de uma paralelização ótima.

# Ideia

- Na PRAM CRCW-or com  $p = n^2$  processadores  $P_{i,j}$  ( $1 \leq i, j \leq n$ ), é possível encontrar a resposta em tempo  $\Theta(1)$ :
  - Se  $B[i] = 1$ ,  $B[j] = 1$  e  $j > i$ , então  $P_{i,j}$  anula  $B[j]$  (conflitos simples: somente são escritos valores nulos)
  - Se  $B[i] = 1$ , então  $P_{1,i}$  escreve  $i$  em  $X$
- Algoritmo ótimo para a PRAM CRCW-or com  $p = n$  processadores:
  - O vetor  $B$  é considerado como  $q = n^{1/2}$  blocos de  $q$  elementos cada.
  - Cada  $P_i$  ( $1 \leq i \leq n$ ) copia  $B[i]$  em  $C[\lceil i/q \rceil]$ , onde  $C$  também é um vetor de tamanho  $q$  (pode haver conflitos de escrita). Tempo:  $\Theta(1)$ .
  - Através do algoritmo anterior, encontrar em  $C$  o primeiro *bit* 1: será o primeiro bloco de  $B$  no qual aparece um *bit* 1. Tempo:  $\Theta(1)$ .
  - Identificado esse bloco em  $B$ , aplicar nele o mesmo algoritmo anterior. Tempo:  $\Theta(1)$ .



# Busca do valor máximo

- Dada um vetor  $A[1..n]$ , deseja-se encontrar seu valor máximo.
- A solução sequencial para esse problema é  $\Theta(n)$ .
- Na mesma PRAM CRCW-or com  $n$  processadores, também seria possível resolver esse problema em tempo constante?
- Inicialmente, elaboraremos um algoritmo de tempo  $\Theta(1)$  utilizando  $p = n^2$  processadores.
- Em seguida, esse algoritmo servirá de inspiração para uma paralelização quase ótima...

# Ideia

- Na PRAM CRCW-or com  $p = n^2$  ( $P_{i,j}$ ,  $1 \leq i, j \leq n$ ), é possível encontrar a resposta em tempo  $\Theta(1)$ :
  - Se  $(A[i] > A[j])$  ou  $(A[i] = A[j]$  e  $i < j)$ , então  $P_{i,j}$  anula  $A[j]$  (conflitos simples: somente são escritos valores nulos).
  - Se  $A[i] \neq 0$ , então  $P_{1,i}$  escreve  $A[i]$  em  $X$
- Algoritmo para a PRAM CRCW-or com  $p = n \cdot n^{1/2}$ :
  - Dividir o vetor  $A$  em  $n^{1/2}$  blocos de  $n^{1/2}$  elementos cada.
  - Através do algoritmo anterior, encontrar em paralelo os valores máximos de cada bloco (utilizam-se  $n$  processadores por bloco), armazenando os resultados em um vetor  $M$  de tamanho  $n^{1/2}$ . Isso gasta tempo  $\Theta(1)$ .
  - Com o mesmo algoritmo anterior, encontrar em  $M$  o valor máximo, também em tempo  $\Theta(1)$  (bastam  $n$  processadores).

# E se $p = n \cdot n^{1/4}$ ?

- Divide-se  $A$  em blocos de tamanho  $n^{1/4}$ :
  - Haverá  $n/n^{1/4} = n^{3/4}$  blocos. É possível calcular em tempo  $\Theta(1)$  um vetor  $M$  de tamanho  $n^{3/4}$  com as subsoluções (são necessários  $n^{(1/4) \cdot 2} \cdot n^{3/4} = n \cdot n^{1/4} = p$  processadores).
- Também divide-se  $M$  em blocos de tamanho  $n^{1/4}$ :
  - Haverá  $n^{3/4}/n^{1/4} = n^{1/2}$  blocos. É possível obter em tempo  $\Theta(1)$  um novo vetor  $M'$  de tamanho  $n^{1/2}$  com as subsoluções (bastam  $n^{(1/4) \cdot 2} \cdot n^{1/2} = n$  processadores).
- Por fim, a melhor solução de  $M'$  pode ser obtida também em tempo  $\Theta(1)$ .

# Generalização

- Dispõem-se de  $p = n^{1+x}$  processadores,  $0 < x < 1$ . Divide-se o vetor  $A$  em  $n^{1-x}$  blocos de tamanho  $n^x$ . Para se calcular em tempo  $\Theta(1)$  o vetor  $M$  de tamanho  $n^{1-x}$  com as subsoluções, são necessários  $n^{2x}n^{1-x} = n^{1+x} = p$  processadores.
- Divide-se  $M$  em  $n^{1-x-y}$  blocos de tamanho  $n^y$ . Para se calcular em tempo  $\Theta(1)$  o vetor  $M'$  de tamanho  $n^{1-x-y}$  com as subsoluções, são necessários  $n^{2y}n^{1-x-y} = n^{1-x+y}$  processadores. Como  $p = n^{1+x}$ , basta escolher  $y = 2x$ , e portanto  $M'$  terá tamanho  $n^{1-3x}$ .
- Se a solução de  $M'$  não puder ser calculada em tempo  $\Theta(1)$ , ou seja, se  $1+x < 2(1-3x)$ , o processo continuará...
- É possível demonstrar que, quando  $p$  tende  $\Theta(n)$ , o tempo para se encontrar a resposta tende a  $\Theta(\log n)$ .

# Ordenação na PRAM CRCW-sum

- Para ordenar um vetor  $A[1..n]$  presente na memória compartilhada, utilizaremos uma PRAM CRCW-sum com  $p = n^2$  processadores  $P_{i,j}$ ,  $1 \leq i, j \leq n$ . Quando houver conflito de escrita, será armazenada a soma dos valores.
- Ideia: calcular o vetor  $C[1..n]$ , onde  $C[i]$  é a quantidade de elementos menores que  $A[i]$ .
- Se dois elementos de  $A$  tiverem o mesmo valor, será considerado menor o de menor índice.
- Desse modo,  $A[i]$  ocupará a  $(C[i]+1)$ -ésima posição na ordenação.

# Exemplo

A =

9	4	6
---	---	---

Escrita concorrente → Soma

$P_{1,1}$

9 & 9

$P_{2,1}$

4 & 9

$P_{3,1}$

6 & 9

$P_{1,2}$

9 & 4

$P_{2,2}$

4 & 4

$P_{3,2}$

6 & 4

$P_{1,3}$

9 & 6

$P_{2,3}$

4 & 6

$P_{3,3}$

6 & 6

C =

2	0	1
---	---	---

A =

4	6	9
---	---	---

# Algoritmo

```
EnumerationSort() {  
    forall  $P_{i,j}$   
        if ( $A[i] > A[j]$  || ( $A[i] == A[j]$  &&  $i > j$ ))  
             $C[i] = 1$ ;  
        else  
             $C[i] = 0$ ;  
    forall  $P_{i,1}$   
         $A[C[i]+1] = A[i]$ ;  
}
```

- Tempo:  $\Theta(1)$
- Custo:  $\Theta(n^2)$
- Não é ótimo, pois o melhor algoritmo sequencial é  $O(n \cdot \log n)$ .
- Há algoritmos ótimos para ordenação na PRAM EREW (Cole).

# Problema da mochila

- A resolução sequencial do *Knapsack Problem 0/1* através de *Programação Dinâmica* gasta tempo  $\Theta(n.c)$ .
- $B[k,w]$ : solução ótima considerando apenas os  $k$  primeiros itens e uma mochila de capacidade  $w$ .

```
Knapsack01() {  
    for (i=0; i<=c; i++)  
        B[0,i] = 0;           // nenhum item é considerado  
    for (k=1; k<=n; k++)     // incremento nos itens  
        for (i=0; i<=c; i++) // incremento na capacidade  
            if (w[k] > i) B[k,i] = B[k-1,i];  
            else B[k,i] = max {B[k-1,i], B[k-1,i-w[k]] + p[k]};  
    return B[n,c]  
}
```



# Resolução na PRAM EREW (com $p = c+1$ )

- Ideia: cada processador  $P_i$  ( $0 \leq i \leq c$ ) calculará as soluções  $B[k,i]$ ,  $0 \leq k < n$ , incrementando passo a passo o número de objetos considerados.

```
ParallelKnapsack01() {  
    forall  $P_i$  Tempo:  $\Theta(n)$   
         $B[0,i] = 0$ ;  
    for ( $k=1$ ;  $k \leq n$ ;  $k++$ )  
        forall  $P_i$  //  $P_i$  calcula soluções para capacidade  $i$   
            if ( $w[k] > i$ )  $B[k,i] = B[k-1,i]$ ;  
            else  $B[k,i] = \max \{B[k-1,i], B[k-1,i-w[k]] + p[k]\}$ ;  
}
```

- Durante o cálculo da linha  $k$  de  $B$ , os processadores utilizam um mesmo deslocamento  $w_k$  na linha  $k-1$ , ocasionando no máximo duas leituras numa mesma posição da memória. Estes eventuais conflitos podem ser resolvidos em tempo constante na PRAM EREW.

# Resolução escalável

- E se não houver tantos processadores disponíveis?
- Lin e Storer (1991) mostraram que é possível tornar escalável essa resolução.
- Considerando  $p < c$ , basta que cada processador cuide de  $q = \lceil (c+1)/p \rceil$  capacidades da mochila. Em concreto,  $P_i$  ( $0 \leq i < p$ ) calculará as soluções ótimas das mochilas com capacidades que variam entre  $i \cdot q$  e  $(i+1) \cdot q - 1$ .
- Desse modo, a resolução gastará tempo  $\Theta(n \cdot c / p)$ , ou seja, é uma paralelização ótima.

# Algoritmo escalável

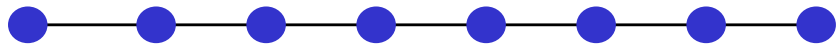
```
ParallelScalableKnapsack01() {  
    q =  $\lceil (c+1)/p \rceil$ ;  
    forall  $P_i$  //  $P_i$  calcula sequencialmente q capacidades  
        for (j=0; j<q; j++)  
            B[0, iq+j] = 0;  
    for (k=1; k<=n; k++)  
        forall  $P_i$  //  $P_i$  calcula sequencialmente q capacidades  
            for (j=0; j<q, j++) {  
                t = iq+j;  
                if ((w[k]-j)/q > i) B[k,t] = B[k-1,t];  
                else B[k,i] = max {B[k-1,t], B[k-1,t-w[k]] + p[k]};  
            }  
}
```

Tempo:  $\Theta(n.c/p)$

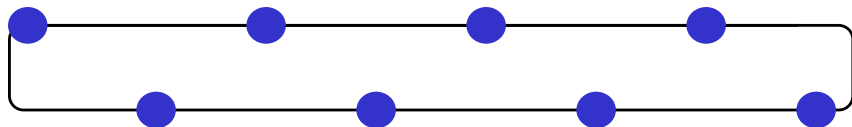
# Redes de interconexão

- Este modelo também é chamado de *message-passing*.
- Critérios que costumam ser considerados:
  - Grau: número de vizinhos de cada processador
  - Diâmetro: maior distância entre dois processadores
  - Comprimento das conexões: o ideal é que não varie em função do número de processadores
- Topologias mais conhecidas:
  - Arranjo linear, anel, grade (*mesh*), toróide, árvore, hipercubo, borboleta (*butterfly*), embaralhador perfeito (*shuffle exchange*), De Bruijn, estrela, etc.

# Arranjos lineares e anéis



Diâmetro:  $p-1$

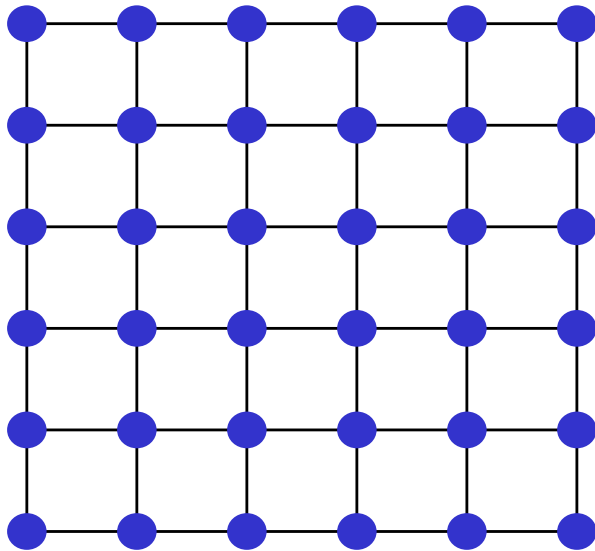


Diâmetro:  $p/2$

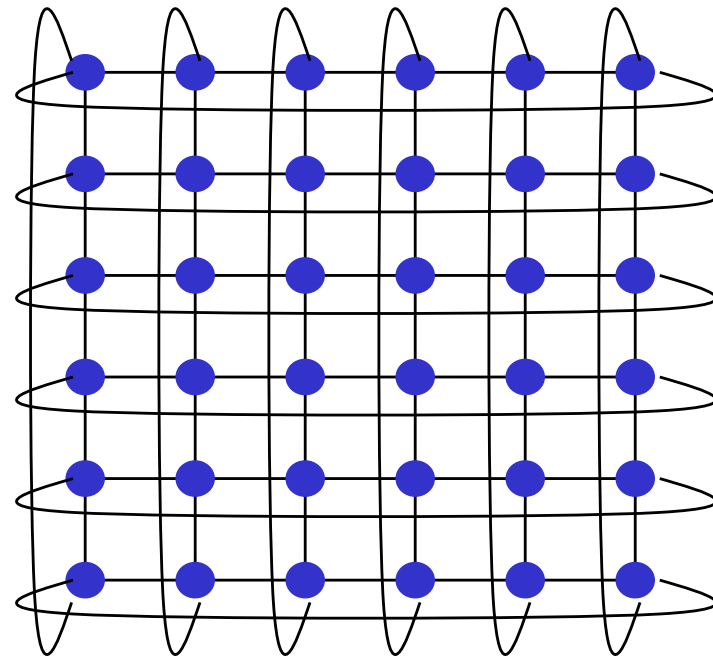
Adequados para algoritmos que trabalham  
com vetores de uma dimensão

# Grades (*meshes*) e toróides

Exemplos para duas dimensões:



Diâmetro:  $2 \cdot p^{1/2}$



Diâmetro:  $p^{1/2}$

Adequados para algoritmos que trabalham com matrizes

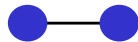
# Hipercubos

- Definição recursiva:

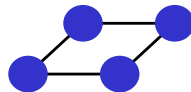
- $H(0)$  é um único vértice, *sem endereço*.
- $H(d)$ ,  $d > 0$ , tem  $p = 2^d$  vértices e é formado por dois  $H(d-1)$ : unem-se os vértices com mesmo endereço, acrescentando-se no início do seu endereço os *bits* 0 ou 1, conforme o  $H(d-1)$ .



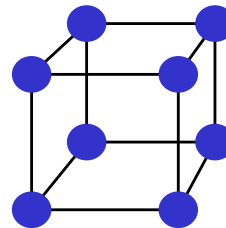
H(0)



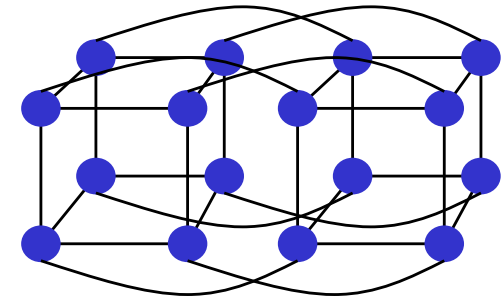
H(1)



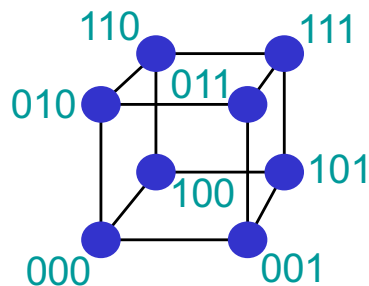
H(2)



H(3)



H(4)



Diâmetro:  $d$

# Hipercubo

- O *hipercubo* é uma das mais conhecidas redes de interconexão, estudado tanto do ponto de vista prático como teórico.
- Alguns protótipos construídos: *Cosmic Cube* ( $p=2^6$ ), *Connection Machine* ( $p=2^{16}$ ), *iPSC/2* ( $p=2^7$ ).
- Além de vantajosas propriedades topológicas, sua definição recursiva facilita a elaboração de algoritmos, principalmente através do paradigma da *Divisão-e-Conquista*.
- Apresentaremos alguns algoritmos SIMD para o hipercubo.

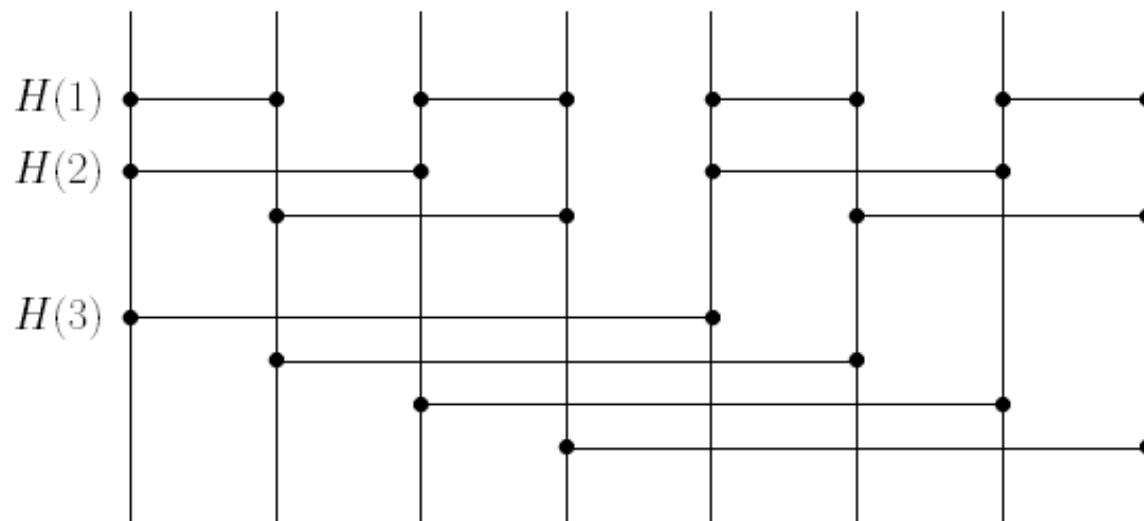


# Notação para algoritmos no hipercubo

- $R(i)$ : registrador  $R$  no vértice  $i$  do hipercubo,  $0 \leq i < p$ .
- $i_b$ :  $b$ -ésimo *bit* (a partir da esquerda) da representação binária de  $i$ ,  $0 \leq b < d$ .
- $i^{(b)}$ : número  $i$  com o *bit*  $i_b$  trocado.
- Comandos de atribuição:
  - $=$  Atribuição realizada entre registradores presentes em um mesmo processador (não exige comunicação)
  - $\leftarrow$  Atribuição que requer uma comunicação entre vértices vizinhos
  - $\leftrightarrow$  Troca de informação com uma comunicação entre vértices vizinhos
- Habilitação de vértices:
  - $R(i) \leftarrow R(i^{(b)}) + 1, (i_4 = 1)$ : executado apenas nos processadores com  $i_4 = 1$

# Visualização linear

Exemplo:  $H(3)$ , ou seja,  $p = 8$



- Vértices aparecem em ordem crescente: de 0 até  $p-1 = 2^d-1$ .
- É fácil observar como o  $H(d)$  é formado por dois  $H(d-1)$ , mais ligações entre os vértices análogos desses sub-hipercubos.

# Inversão de posições

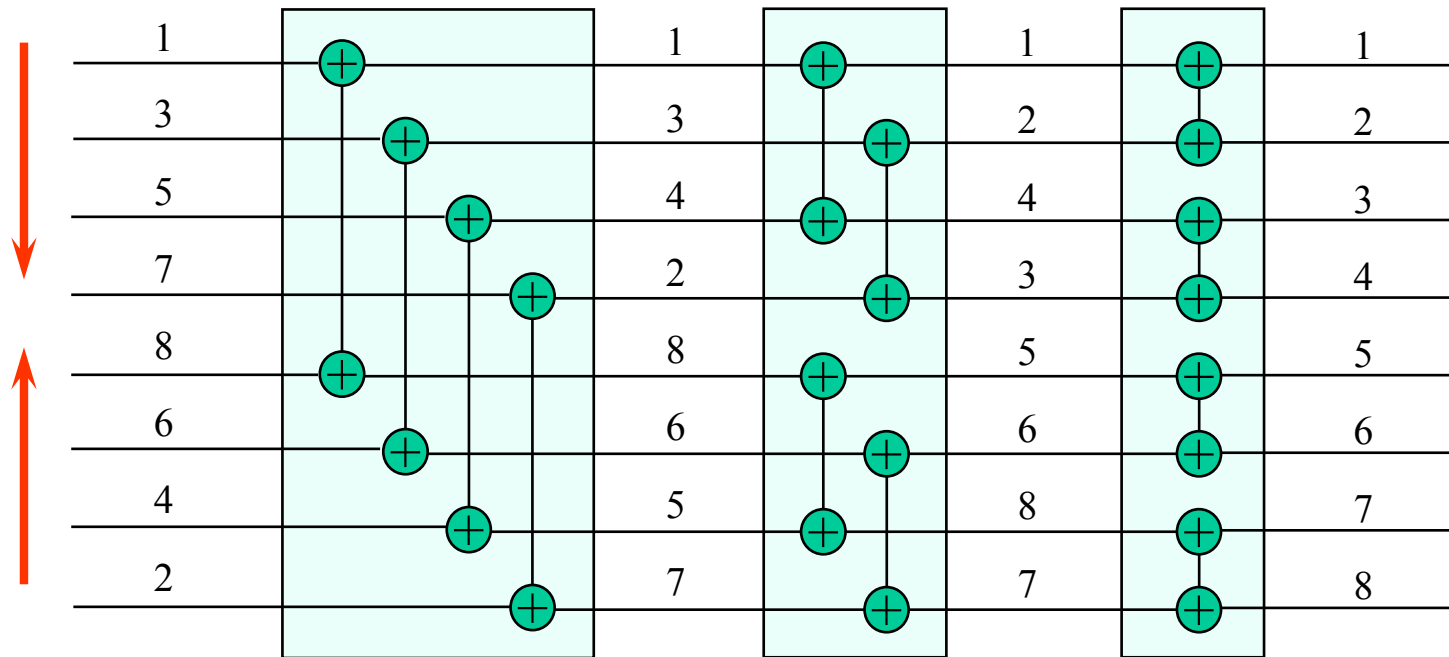
- Cada processador  $P_i$  armazena um valor em  $R(i)$ ,  $0 \leq i < p$ .
- Deseja-se inverter suas posições, ou seja, trocar  $R(i)$  com  $R(p-i-1)$ ,  $0 \leq i < p$ .
- Ideia: observar que os endereços desses pares de processadores têm todos os *bits* trocados entre si.

```
HypercubeInversion(b1, b2) {  
    for (b=b1; b<=b2; b++)  
        R(i(b)) ↔ R(i);  
}
```

Chamada: HypercubeInversion(0, d-1)

Tempo:  $\Theta(d) = \Theta(\log p)$

# Intercalação bitônica (*Batcher*)



n números ordenados em  $\log n$  estágios

# Intercalação (*merge*)

- Há duas seqüências ordenadas em  $R(i)$ ,  $0 \leq i < p$ , que devem ser intercaladas.
- Ideia: inverter a segunda seqüência e aplicar a intercalação bitônica.

```
HypercubeMerge (b1, b2) {  
    HypercubeInversion (b1+1, b2), ( $i_{b1} = 1$ );  
    for (b=b2; b>=b1; b--) {  
        S(i)  $\leftarrow$  R( $i^{(b)}$ );  
        R(i) = min {S(i), R(i)}, ( $i_b = 0$ );  
        R(i) = max {S(i), R(i)}, ( $i_b = 1$ );  
    }  
}
```

Chamada: HypercubeMerge (0, d-1)

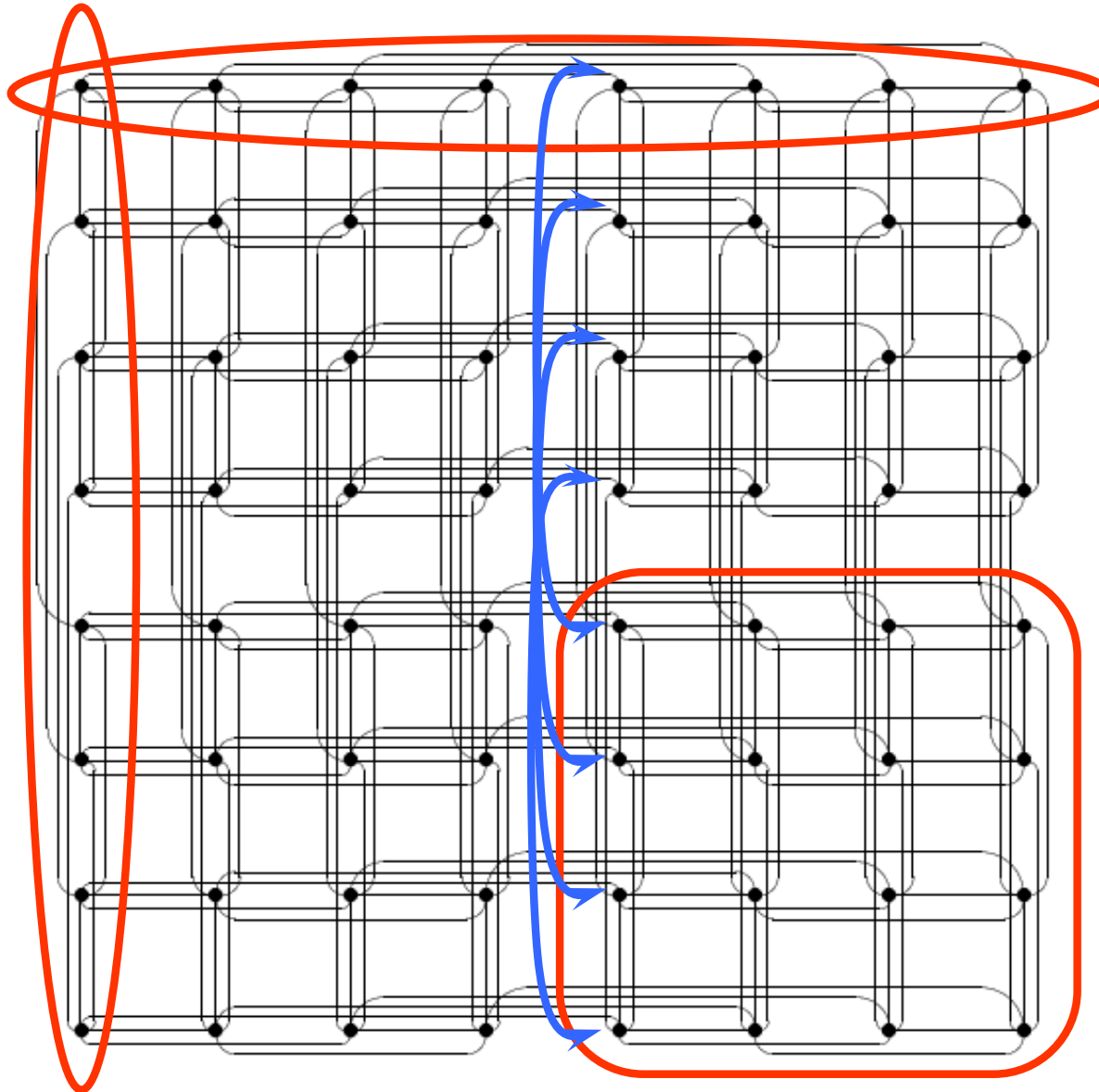
Tempo:  $\Theta(d) = \Theta(\log p)$

# Ordenação

- A partir do algoritmo de intercalação, fica simples elaborar o *MergeSort* paralelo no  $H(d)$ :
  - a) Ordenações recursivas em paralelo em cada  $H(d-1)$
  - b) Intercalação paralela
- Tempo total:  $T(d) = T(d-1) + \Theta(d)$
- $T(d) = \Theta(d^2)$
- Quantidade de valores a serem ordenados:  
 $n = p = 2^d$ .
- Logo,  $T(n) = \Theta(\log^2 n)$ .

# Visualização matricial do hipercubo

Exemplo:  
 $H(6)$   
 $p = 64$



Sub-hipercubos

Conexões entre  
elementos  
correspondentes  
nos sub-hipercubos

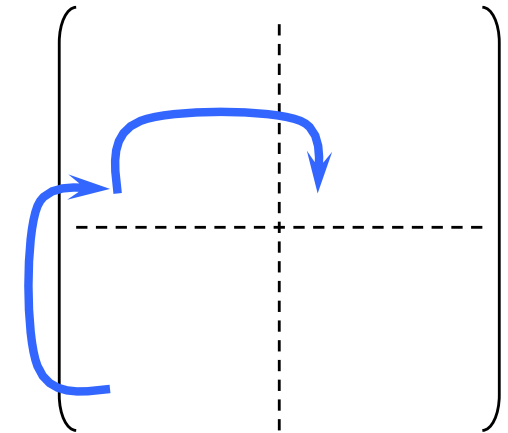
Análogo para  
visualizações  
matriciais de  
dimensões  
maiores

# Transposição de matriz

Matriz quadrada  $n \times n$

Hipercubo de dimensão  $d$ , onde  $2^d = n^2$

Inicialmente armazenada segundo  
a visualização matricial



$T(n)$ : tempo para realizar a transposição

$$T(n) = T(n/2) + \Theta(1)$$

$$T(n) = \Theta(\log n)$$



# Multiplicação de matrizes quadradas

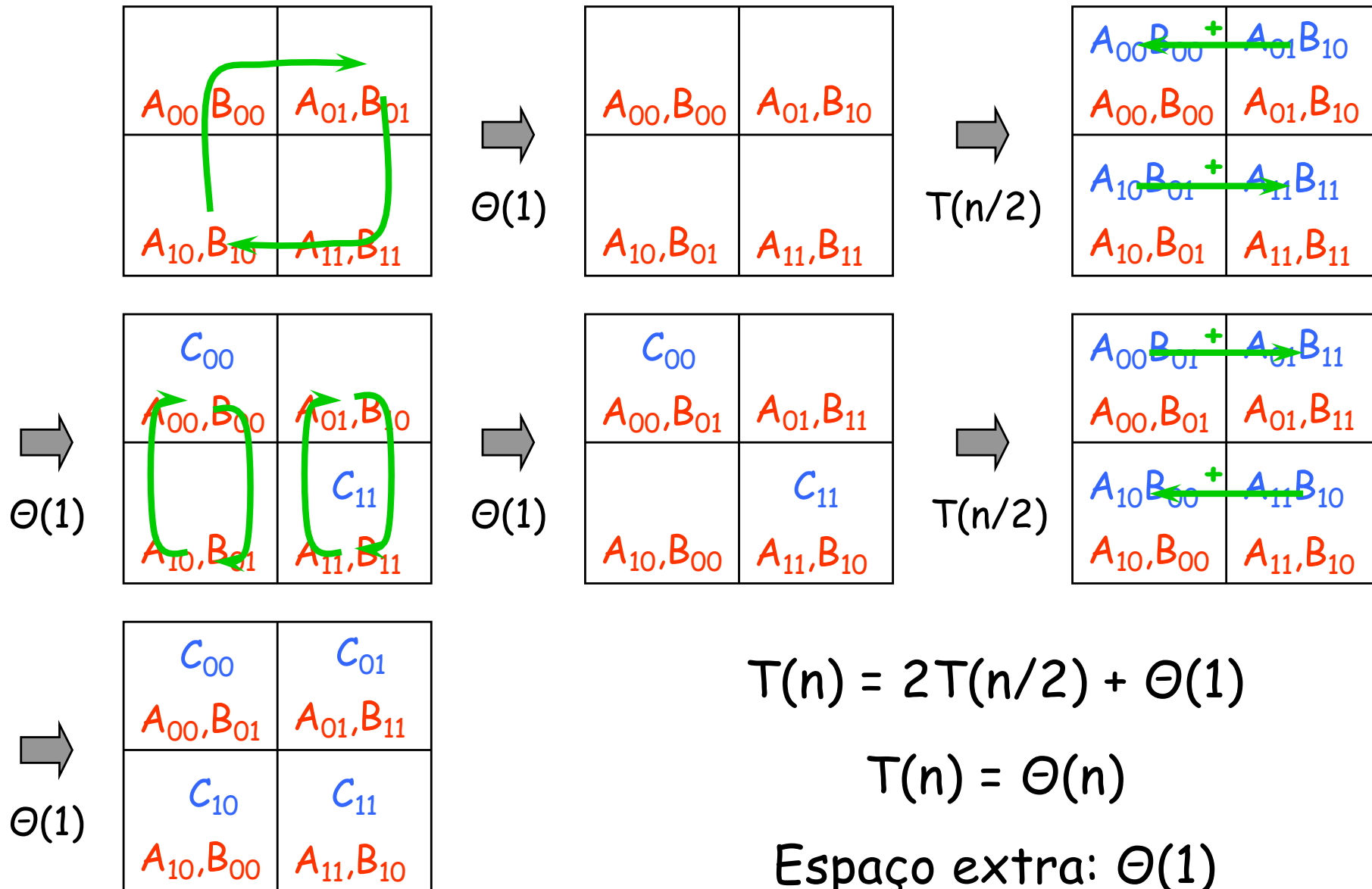
$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix}$$

$$C_{00} = A_{00}B_{00} + A_{01}B_{10} \quad C_{01} = A_{00}B_{01} + A_{01}B_{11}$$

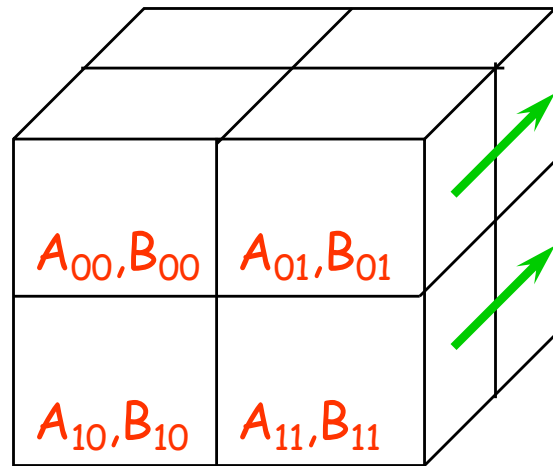
$$C_{10} = A_{10}B_{00} + A_{11}B_{10} \quad C_{11} = A_{10}B_{01} + A_{11}B_{11}$$

$$C_{ij} = A_{i0}B_{0j} + A_{i1}B_{1j}$$

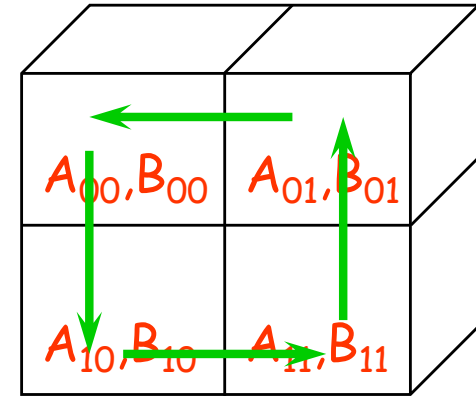
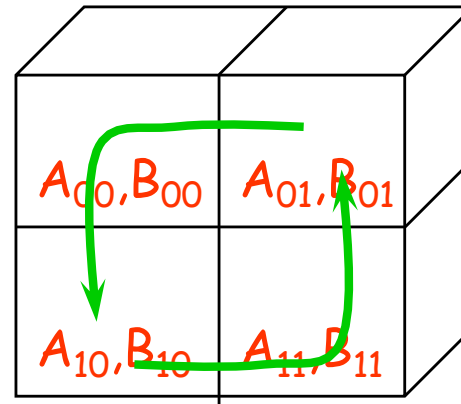
# Multiplicação no hipercubo com $p = n^2$



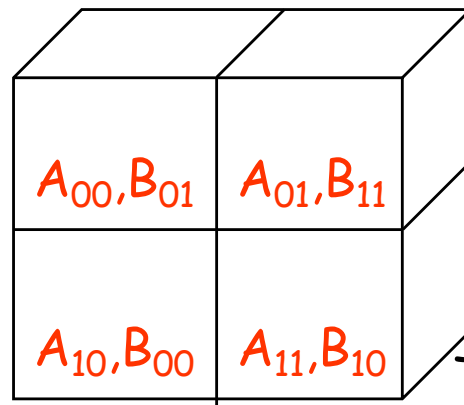
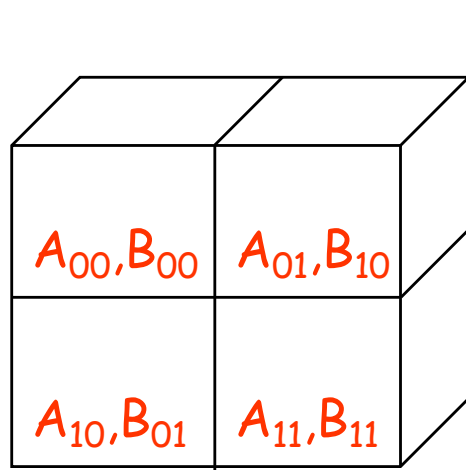
# Multiplicação no hipercubo com $p = n^3$



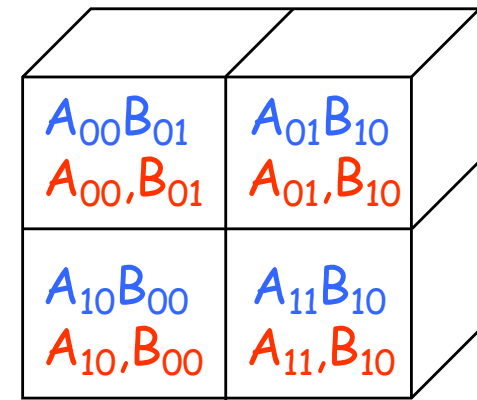
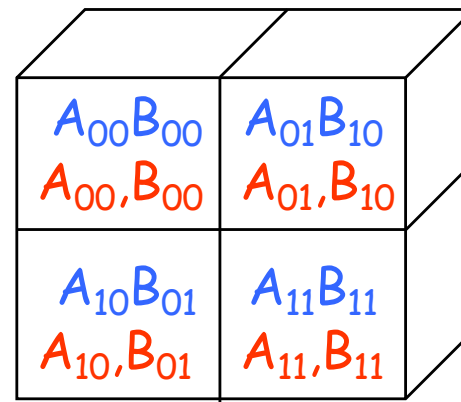
$\Theta(1)$



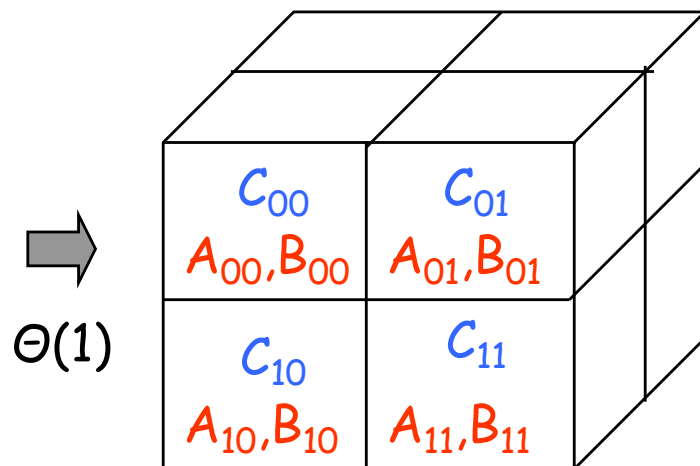
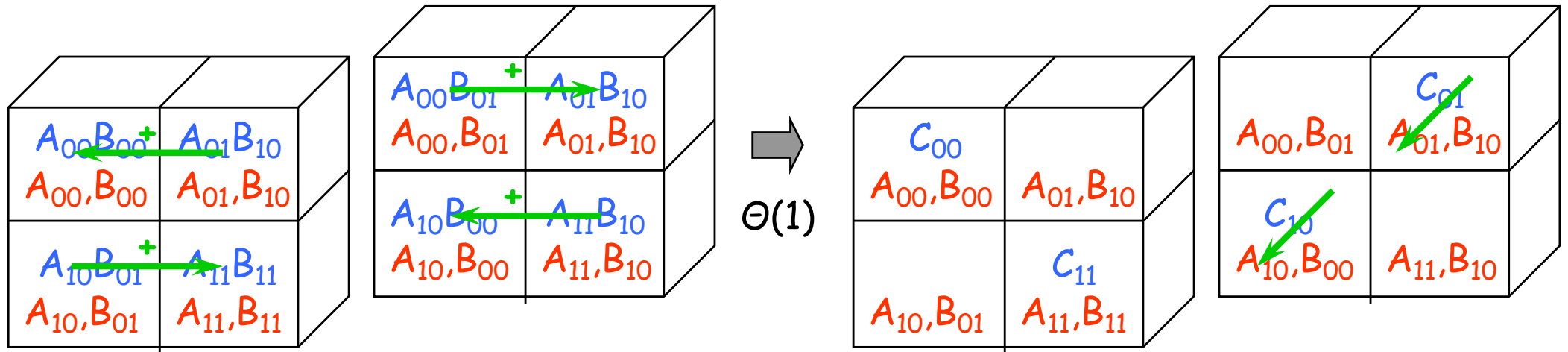
$\Theta(1)$



$T(n/2)$



# Multiplicação no hipercubo com $p = n^3$



$$T(n) = T(n/2) + \Theta(1)$$

$$T(n) = \Theta(\log n)$$

Espaço extra:  $\Theta(1)$

Estes resultados podem ser generalizados para  $1 \leq p \leq n^3$