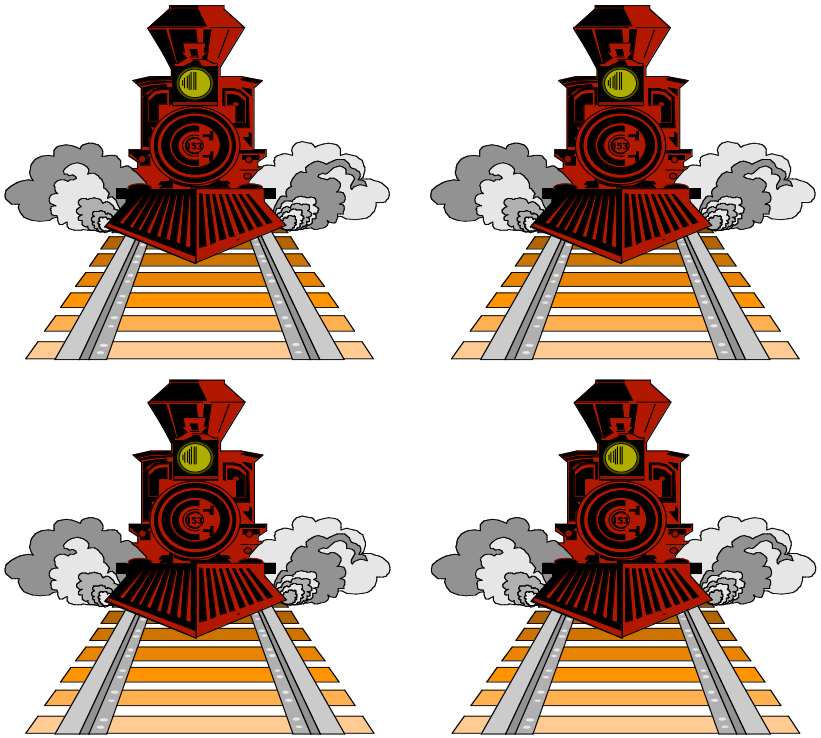
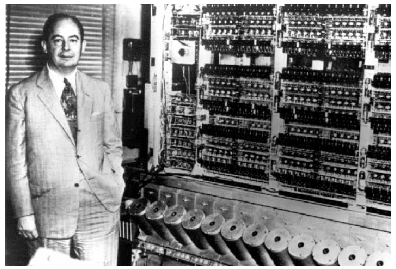
Parallelle algoritmer



Von Neumann's model



John von Neumann 1903-57

Von Neumanns model: Instruktioner og data er lagret i samme lager, og én processor henter instruktioner fra lageret og udfører dem én ad gangen

De fleste computere er variationer over denne model

Nye maskiner vinder frem, som tillader udførelse af et stort antal instruktioner samtidigt (i **parallel**)

Der kan være tale om generelt anvendelige maskiner, eller maskiner, der er specialbyggede til løsning af specifikke opgaver (eller opgave-typer)

Hastighedsforøgelse

(speedup)

Mulig fremgangsmåde:

(1) Flyt eksisterende effektiv algoritme uændret til parallelcomputer

(2) Tilpas derefter algoritmen til computeren

Ikke altid nogen god ide! Algoritmen kan være uegnet til parallelisering

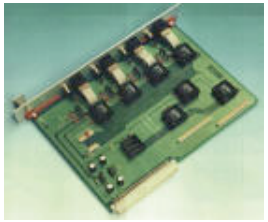
Lad $T(n, p)$ betegne køretiden for en algoritme med inputstørrelse n på p processorer

$S(p) = T(n,1)/T(n, p)$ kaldes algoritmens **speedup**
Hvis $S(p) = p$ siges algoritmen at have **perfekt speedup**

Ved **effektiviteten** af en parallel algoritme forstås $E(p) = S(p)/p$

[udtrykker udnyttelsesgraden af processorerne]

Eksempel
Parallel addition



Problem: Find summen af to n-bit binære tal

En sekventiel løsning har et tidsforbrug $O(n)$

Parallel løsning:
Benyt del-og-hersk til at opnå en parallel algoritme
Hvert problem opdeles i to uafhængige delproblemer

	n-m bit	*m* bit (*m* = *n*/2)
`a:`	`aL`	`aR`
`b:`	`bL`	`bR`

Beregn i **parallel**:

```
sR = aR + bR + carry;
                    // carry = 0/1
sL0 = aL + bL + 0;
sL1 = aL + bL + 1;
```

Hvis `aR + bR + carry` ikke giver mente, benyttes `sL0` i resultatet:

`sL0`	`sR`

Ellers benyttes `sL1`:

`sL1`	*m* sidste bit af `sR`

Java-metode til addition ved del-og-hersk

```
int sum(int a, int b, int carry, int n) {
    if (n == 1) return a + b + carry;
    int m = n / 2;
    int aL = a >> m, aR = a & ~(~0 << m);
    int bL = b >> m, bR = b & ~(~0 << m);
    int sR =  sum(aR, bR, carry, m);
    int sL0 = sum(aL, bL, 0, n - m);
    int sL1 = sum(aL, bL, 1, n - m);
    if (sR >> m == 0)
        return (sL0 << m) | sR;
    else
        return (sL1 << m) | (sR & ~(~0 << m));
}
```

udføres i parallel

Eksempel på kald: `s = sum(25, 29, 0, 16);`

Tidsforbrug: $T(n) = T(n/2) + O(1) \Rightarrow T(n) = O(\log n)$

Problemer ved parallelisering

Kommunikation imellem processorer:

- ikke alle processorer kan være forbundet (hvis der er mange processorer)
- langsomme forbindelser

Synkronisering af processorer:

- en processor kan blive nødt til at vente på en anden processor
- en processors aktivitet bør ikke baseres på andre processorers relative hastigheder

Synkronisering

Når to eller flere processorer, processer eller tråde samtidigt opdaterer de samme data, kan resultatet blive forkert

Bruger 1

```
s1 = k.hentSaldo();

s1 += beløb1;

k.saetSaldo(s1);
```

Bruger 2

```
s2 = k.hentSaldo();

s2 += beløb2;

k.setSaldo(s2);
```

Fejl: saldoen bliver kun øget med `beløb2`

```
class Konto {
    private int saldo;
    public Konto(int startSaldo)
      { saldo = startSaldo; }
    public int hentSaldo()
      { return saldo; }
    public void saetSaldo(int nySaldo)
      { saldo = nySaldo; }
}
```

Kritiske regioner

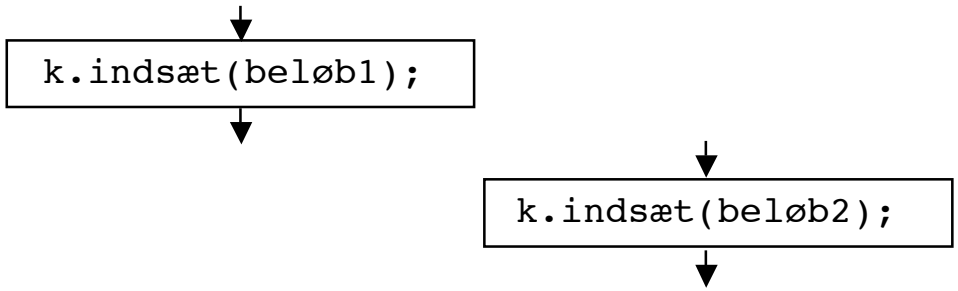
Et stykke kode, som kan give et utilsigtet resultat, hvis det udføres af flere processer/tråde samtidig, kaldes for en **kritisk region**

Java gør det muligt - gennem **`synchronized`** metoder og **`synchronized`** sætninger - at sikre, at kun én tråd ad gangen kan være i en kritisk region
Hvis en tråd ønsker at udføre en kritisk region, som en anden tråd udfører, må den vente, indtil ingen anden tråd udfører denne kode

Synkronisering for kontoeksemplet

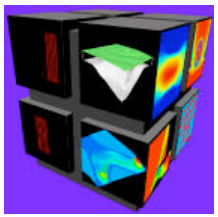
```
class Konto {
    private int saldo;
    public Konto(int startSaldo)
      { saldo = startSaldo; }
    public synchronized int hentSaldo()
      { return saldo; }
    public synchronized void indsæt(int beløb)
      { saldo += beløb; }
}
```

Under udførelsen af en **`synchronized`** metode ”låses” objektet

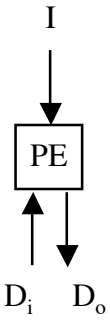


Flynns klassifikation

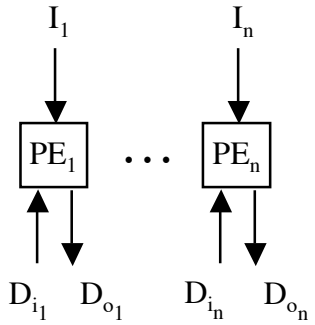
1966



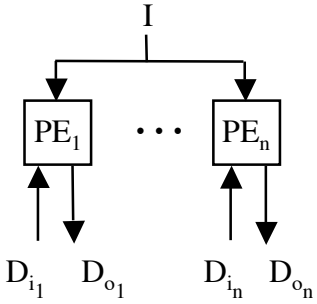
SISD (**S**ingle-**I**nstruction **S**ingle-**D**ata):
Én instruktionsstrøm, én datastrøm
[en sædvanlig computer]



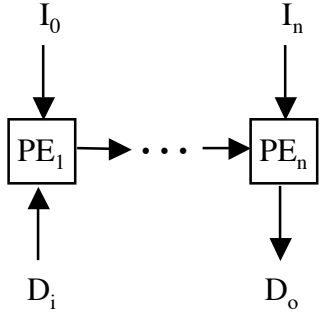
MIMD (**M**ultiple-**I**nstruction **M**ultiple-**D**ata):
Flere instruktionsstrømme, flere datastrømme.
[f.eks. forbundne computere]



SIMD (**S**ingle-**I**nstruction **M**ultiple-**D**ata):
Én instruktionsstrøm, flere datastrømme
[f.eks. en vektorcomputer]



MISD (**M**ultiple-**I**nstruction **S**ingle-**D**ata):
Flere instruktionsstrømme, én datastrøm.
[f.eks. systoliske maskiner]



Kommunikation i MIMD-arkitekturen

Kommunikationen i MIMD kan enten ske:

(1) ved brug af fælles lager

(2) ved udveksling af beskeder (meddelelser)

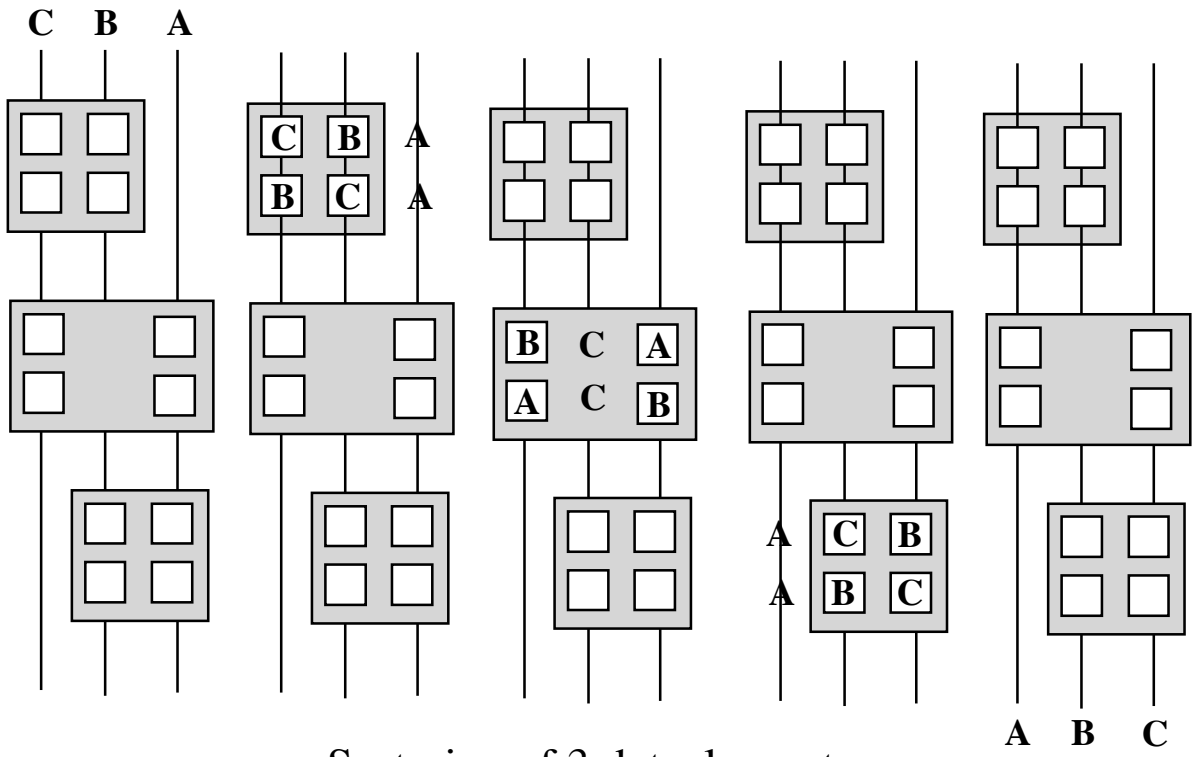
Parallelle algoritmer

Design, analyse og verifikation af parallelle algoritmer er langt vanskeligere end for traditionelle algoritmer

I det følgende betragtes specialbyggede maskiner til løsning af specifikke opgaver

De viser effekten af maskinarkitektur på algoritmedesign - og omvendt

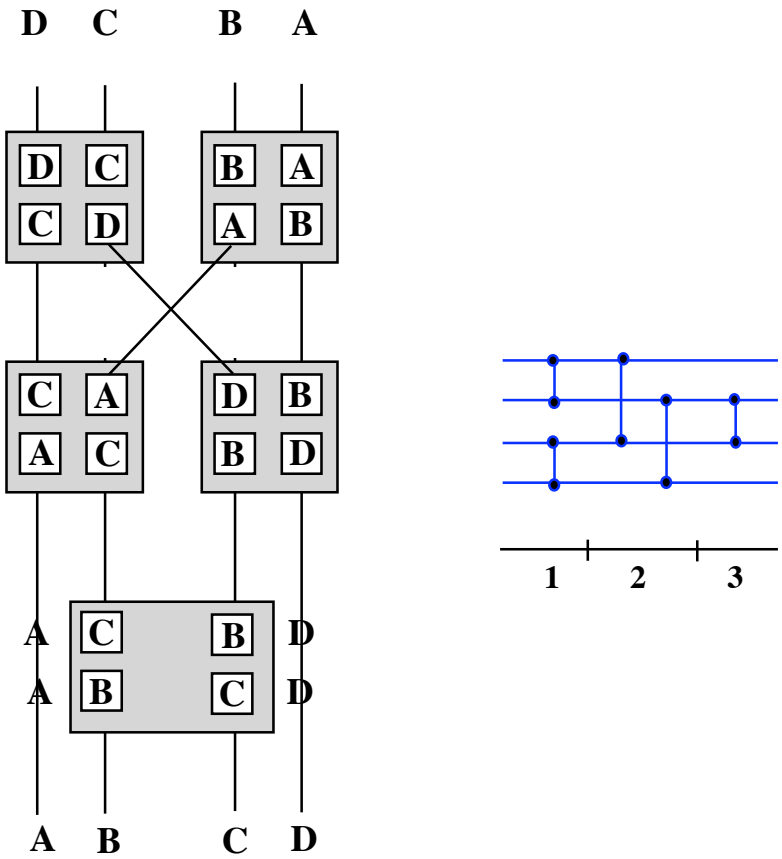
Sorteringsnetværk



Sortering af 3 dataelementer

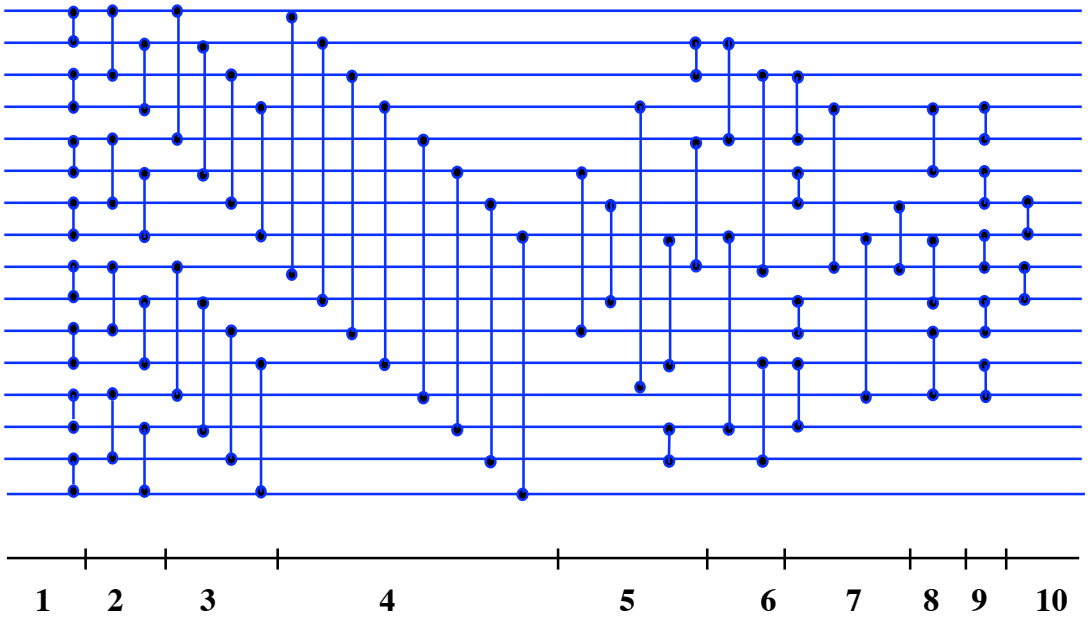
Ikke-adaptiv algoritme: Ligegyldigt hvilke dataelementer, der skal sorteres, udføres en fast sekvens af sammenligning-ombytnings-operationer

Sortering af 4 dataelementer



Sortering af 4 dataelementer kan foretages i 3 parallelle skridt

Sortering af 16 dataelementer



60 komponenter, 10 parallelle skridt

Det er muligt at bygge sorteringsnetværk bestående af cirka $n \lg^2 n / 4$ komponenter, der kan sortere i cirka $\lg^2 n / 2$ parallelle skridt

Matrixmultiplikation

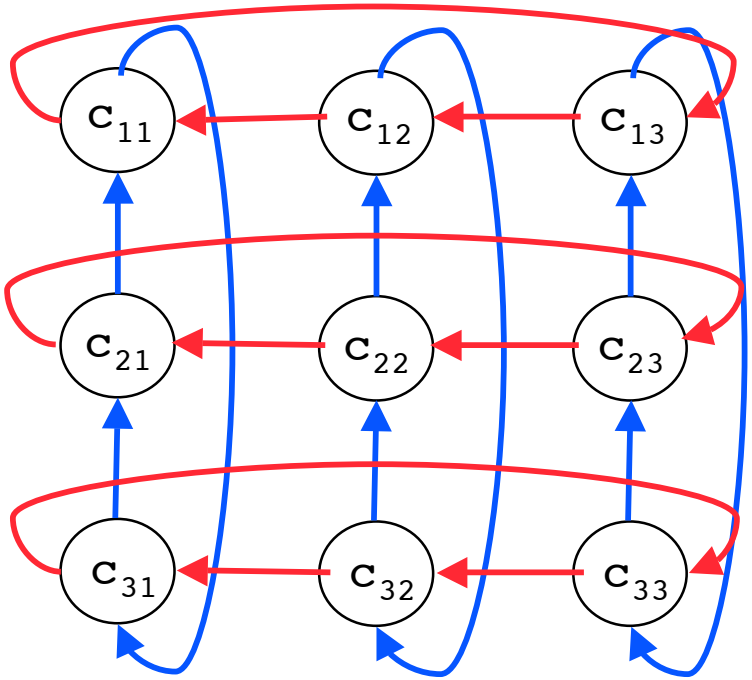
$$\begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$$

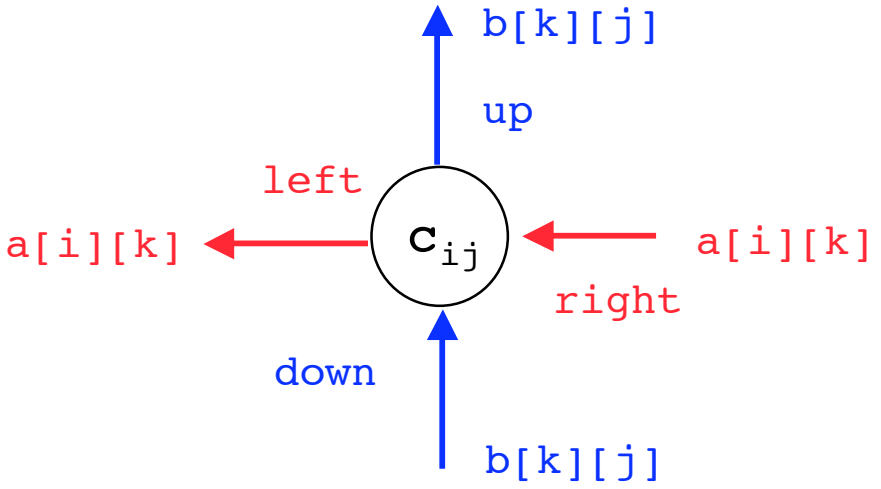
$$c_{ij} = \sum_k a_{ik} b_{kj}$$

```
for (i = 1; i <= N; i++)
    for (j = 1; j <= N; j++) {
        c[i][j] = 0;
        for (k = 1; k <= N; k++)
            c[i][j] += a[i][k] * b[k][j];
    }
```

Tidsforbrug: $O(N^3)$

Matrixmultiplikation på en systolisk maskine





$$c_{ij} = \sum_k a_{ik} b_{kj}$$

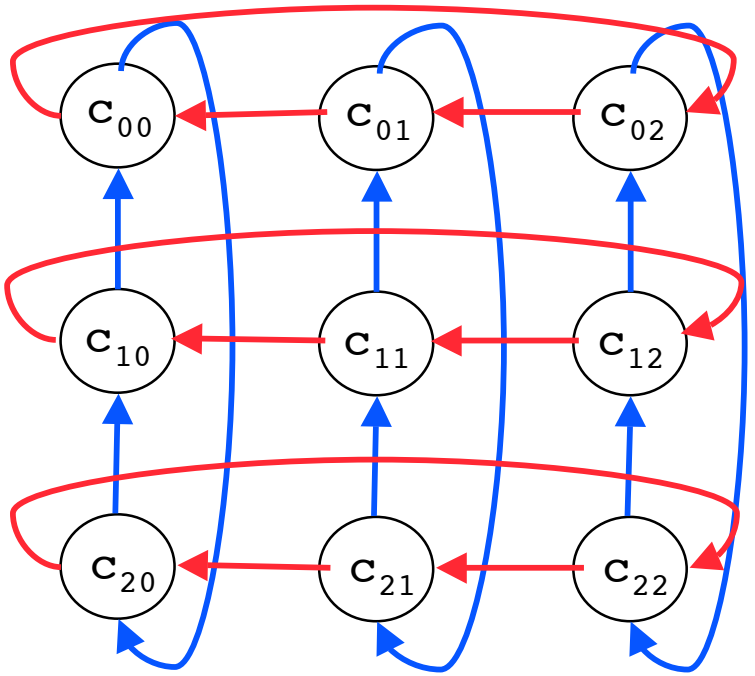
Ressourceforbrug

Maskinen multiplicerer to N x N matricer ved brug af N^2 processorer i N parallelle skridt

```
class Processor extends Thread {
    private int i, j, a, b, c, n;
    private Channel left, right, up, down;

    public Processor(int I, int J,
                     int A, int B, int N,
                     Channel L, Channel R,
                     Channel U, Channel D) {
        i = I; j = J; a = A; b = B; n = N;
        left = L; right = R; up = U; down = D;
        start();
    }

    public int getResult() {
        try { join(); }
        catch (InterruptedException e) {}
        return c;
    }

    void run() {
       ...
    }
}
```

```
class Channel {
    private int message;
    private boolean full = false;

    public synchronized void send(int message) {
        while (full)
            try { wait(); }
            catch (InterruptedException e) {}
        this.message = message;
        full = true;
        notify();
    }

    public synchronized int receive() {
        while (!full)
            try { wait(); }
            catch (InterruptedException e) {}
        full = false;
        notify();
        return message;
    }
}
```

```
public void run() {
    // a == a[i][j] && b == b[i][j]
    for (int k = 0; k <= i; k++) {
        left.send(a);
        a = right.receive();
    }
    for (int k = 0; k <= j; k++) {
        up.send(b);
        b = down.receive();
    }
    // a == a[i][0] && b == b[0][j]
    c = a * b;
    for (int k = 1; k < n; k++) {
        left.send(a);
        up.send(b);
        a = right.receive();
        b = down.receive();
        c += a * b;
    }
}
```



```
public static void main(String args[]) {
    int[][] a = {{ 1, 3,-4},
                 { 1, 1,-2},
                 {-1,-2, 5}};
    int[][] b = {{ 8, 3, 0},
                 { 3,10, 2},
                 { 0, 2, 6}};
    Channel[][][] c = new Channel[2][n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            c[0][i][j] = new Channel();
            c[1][i][j] = new Channel();
        }
    Processor[][] p = new Processor[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            p[i][j] = new Processor(i, j, a[i][j], b[i][j], n,
                                    c[0][i][j], c[0][i][(j + 1) % n],
                                    c[1][i][j], c[1][(i + 1) % n][j]);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            System.out.print(p[i][j].getResult() + " ");
        System.out.println();
    }
}
```