IV100 Distribuované výpočty

Rasťo Královič

Katedra informatiky, FMFI UK Bratislava
kralovic@dcs.fmph.uniba.sk

- **Gerard Tel:** *Introduction to Distributed Algorithms*, Cambridge University Press, 2000, ISBN 0521794838
- **Nancy Lynch:** *Distributed Algorithms*, Morgan Kaufmann Publishers, 1996, ISBN 1558603484
- **Frank Thomson Leighton:** *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, 1991, ISBN 1558601171
- **Joseph JáJá:** *Introduction to Parallel Algorithms*, Addison-Wesley Professional, 1992, 978-0201548563

sekvenčné výpočty

- algoritmus (počítanie funkcie)
- čas
  – problém: presný, ale neprenositeľný
- riešenie: počet inštrukcií v modeli
  – problém: aký model?
-

jednoduchý "assembler"

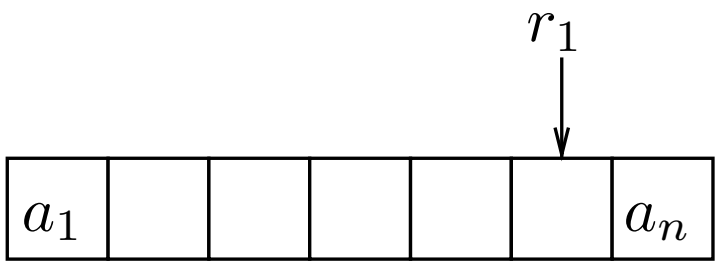
- registre $r_0, r_1, r_2, r_3, \ldots$
- vstup $a_0, a_1, a_2, a_3, \ldots$
- program = postupnosť inštrukcií
- pozícia v programe

inštrukcie

priradenie	$r_X := r_Y$	X, Y je konštanta alebo register
	$r_X := a_Y$	
	$r_X := c$	c je konštanta
výpočet	$r_X := r_Y \square r_Z$	$\square$ je $+, -$
skok	**goto** i	
podmienka	**if** $r_X \begin{matrix} \leq \\ = \\ \geq \end{matrix} r_Y$ **then goto** i	
	if $r_X \begin{matrix} \leq \\ = \\ \geq \end{matrix} c$ **then goto** i	

príklad: čo robí tento program?

1: vstup: $a_0 = n \qquad a_1, a_2, \ldots, a_n, \quad a_i \geq 0$
2: $r_0 := -1$
3: $r_1 := a_0$
4: $r_1 := r_1 + 1$
5: $r_1 := r_1 - 1$
6: **if** $r_1 = 0$ **then goto** 10
7: **if** $r_0 \geq a_{r_1}$ **then goto** 5
8: $r_0 := a_{r_1}$
9: **goto** 5
10: výstup: r_0



r_0 *maximum*

sekvenčné výpočty

- algoritmus (počítanie funkcie)
- čas
  – problém: presný, ale neprenositeľný
- riešenie: počet inštrukcií v modeli
  – problém: aký model?
- riešenie: zaujíma nás asymptotický rast

$$f \in O(g) \Leftrightarrow \exists c, n_0 \; \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

$$f \in \Omega(g) \Leftrightarrow \exists c, n_0 \; \forall n \leq n_0 : f(n) \leq c \cdot g(n)$$

distribuované algoritmy

dôvody

- lepší návrh
  OS, browser, ...
- rýchlejšie riešenie problémov
  scheduling, FEM, ...
- veľké dáta
- využitie hardvéru
  GPU, seti@home, ...
- fyzická vzdialenosť
  ATM, social networking, ...

výzvy

- rôzne ciele
- rôzne úlohy
- rôzny hardvér
- narastanie zložitosti

taxonómia

- Flynn (SISD, SIMD, MIMD)
- tightly/loosely coupled

nie je jeden model

bez zdieľanej pamäte – procesy

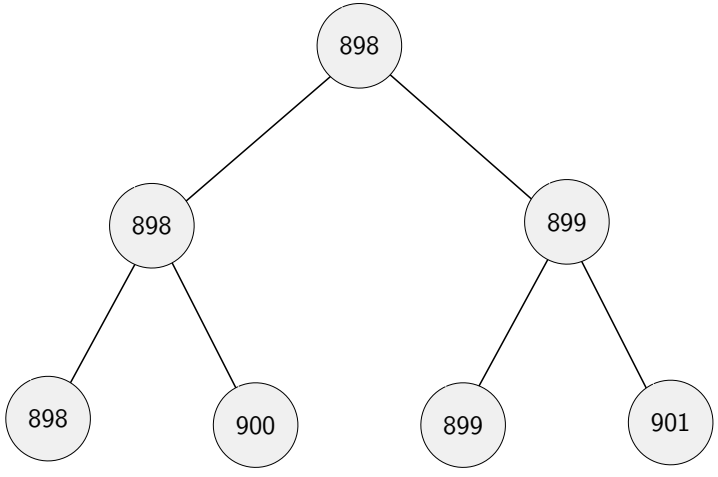
```
void do_child(int data_pipe[]) {
  int c,rc;

  close(data_pipe[1]);
  while ((rc = read(data_pipe[0], &c, 1)) > 0)
    putchar(c);
  exit(0);
}
```

```
void do_parent(int data_pipe[]) {
  int c,rc;

  close(data_pipe[0]);
  while ((c = getchar()) > 0)
    rc = write(data_pipe[1], &c, 1);
  close(data_pipe[1]);
  exit(0);
}
```

```
int main() {
  int data_pipe[2];
  int pid,rc;

  rc = pipe(data_pipe);
  pid = fork();
  switch (pid) {
    case 0:
      do_child(data_pipe);
    default:
      do_parent(data_pipe);
  }
}
```

```
int main(void) {
  int value1, value2;
  int a=42;

  printf("%d started (a=%d)\n",
      getpid(),a);
  value1 = fork();
  printf("%d: value1=%d (a=%d)\n",
      getpid(),value1,a);

  value2 = fork();
  printf("%d: value2=%d (a=%d)\n",
      getpid(),value2,a);
  return 0;
}
```

```
898 started (a=42)
898: value1=899 (a=42)
899: value1=0 (a=42)
898: value2=900 (a=42)
899: value2=901 (a=42)
900: value2=0 (a=42)
901: value2=0 (a=42)
```



bez zdieľanej pamäte – procesy

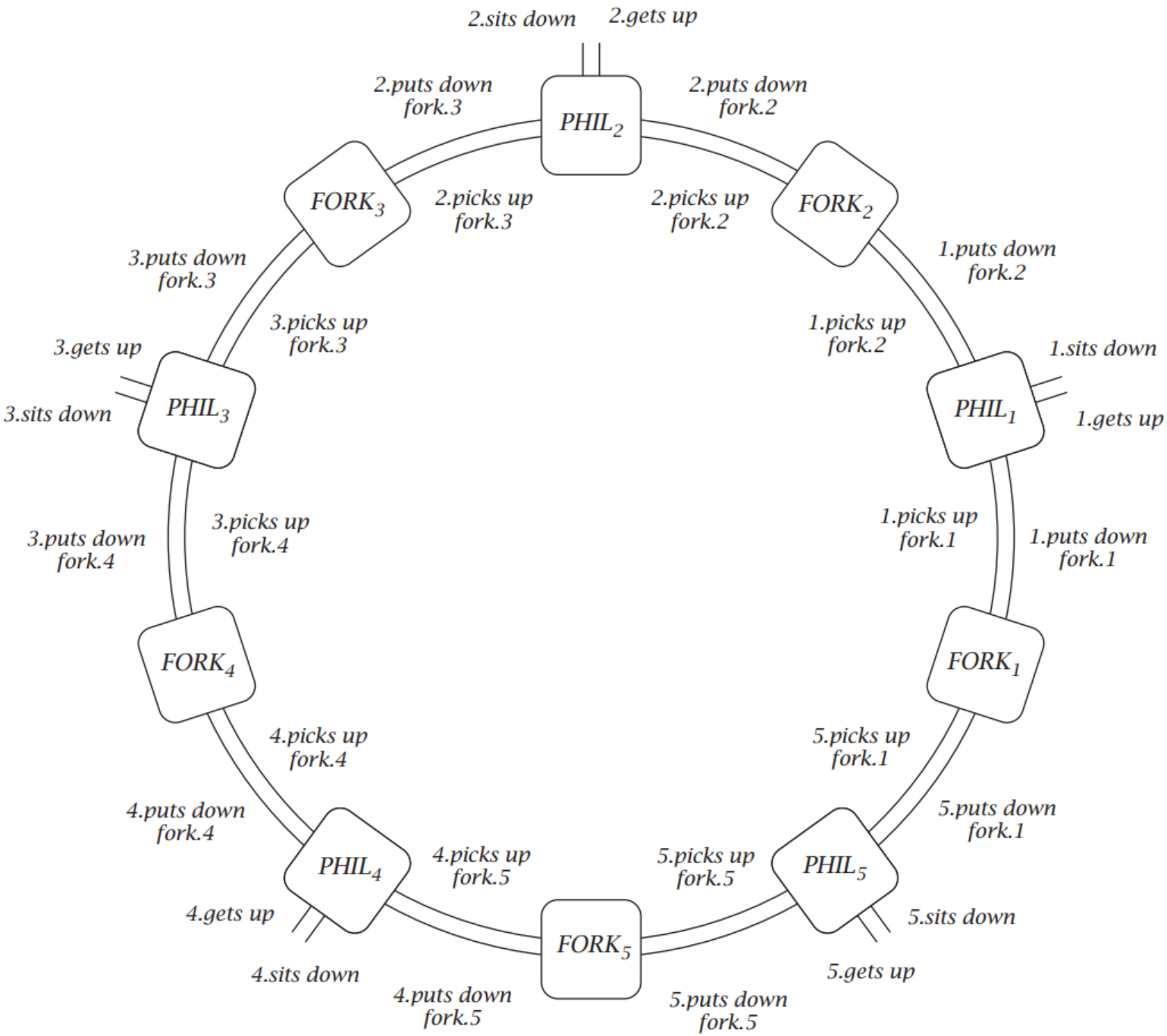
```
void proc(int p,int fd[2])  {
  int i=0;
  close(fd[0]);
  while(1) {
    sleep(rand()%4);
    sprintf(line,"h%d from %d\n",
      ++i,p);
    write(fd[1], line, strlen(line));
  }
}
```

```
int main(void) {
  int fds[NPROC][2];
  int n,c,i;
  fd_set r;
  int maxf = 0;
  for (i=0;i<NPROC;i++) {
    pipe(fds[i]);
    if (fork()==0) proc(i,fds[i]);
    else  {
      close(fds[i][1]);
      if (fds[i][0]>maxf) maxf=fds[i][0];
    }
  }
  while(1) {
    FD_ZERO(&r);
    for (i=0;i<NPROC;i++)
      FD_SET(fds[i][0],&r);
    select(maxf+1,&r,NULL,NULL,NULL);
    for (i=0;i<NPROC;i++)
      if (FD_ISSET(fds[i][0],&r)) {
        n = read(fds[i][0],line,MAXLINE);
        printf("%s",line);
      }
}}
```

CSP (Communicating Sequential Processes)

- sémantika
- pozorovanie procesu: postupnosť udalostí
- proces: množina pozorovaní (**trace**)
- vytváranie procesov:
  - **prefix**: $(x \mapsto P)$
  - **rekurzia**: `CLOCK = (tick↦CLOCK)`
    $F \equiv \mu X.F(X)$
  - **deterministický výber**: `(x↦P | y↦Q)`
  - **paralelizmus**: `(P || Q)` všetky "premiešania" pozorovaní
- **komunikácia**: zdieľaním udalostí (rendezvous)

`K` = μ `X.minca↦(káva↦X | čaj↦X)`
`Z` = μ `X.( čaj↦X | káva↦X | minca↦čaj↦X)`
`(Z || K)` = μ `X.(minca↦čaj↦X)`


2.sits down
2.gets up
2.puts down fork.3
2.puts down fork.2
PHIL2
FORK3
FORK2
2.picks up fork.3
2.picks up fork.2
3.puts down fork.3
1.puts down fork.2
3.picks up fork.3
1.picks up fork.2
3.gets up
1.sits down
3.sits down
PHIL3
PHIL1
1.gets up
3.picks up fork.4
1.picks up fork.1
3.puts down fork.4
1.puts down fork.1
FORK4
FORK1
4.picks up fork.4
5.picks up fork.1
4.puts down fork.4
5.puts down fork.1
PHIL4
PHIL5
4.picks up fork.5
5.picks up fork.5
4.gets up
5.sits down
FORK5
4.sits down
5.gets up
4.puts down fork.5
5.puts down fork.5

$$FORK_i = (i.picks\ up\ fork.i \rightarrow i.puts\ down\ fork.i \rightarrow FORK_i$$
$$\mid (i \ominus 1).picks\ up\ fork.i \rightarrow (i \ominus 1).puts\ down\ fork.i \rightarrow FORK_i)$$
$$LEFT_i = (i.sits\ down \rightarrow i.picks\ up\ fork.i \rightarrow$$
$$i.puts\ down\ fork.i \rightarrow i.gets\ up \rightarrow LEFT_i)$$
$$RIGHT_i = (i.sits\ down \rightarrow i.picks\ up\ fork.(i \oplus 1) \rightarrow$$
$$i.puts\ down\ fork.(i \oplus 1) \rightarrow i.gets\ up \rightarrow RIGHT_i)$$
$$PHIL_i = LEFT_i \parallel RIGHT_i$$
$$PHILOS = (PHIL_0 \parallel PHIL_1 \parallel PHIL_2 \parallel PHIL_3 \parallel PHIL_4)$$
$$FORKS = (FORK_0 \parallel FORK_1 \parallel FORK_2 \parallel FORK_3 \parallel FORK_4)$$
$$COLLEGE = PHILOS \parallel FORKS$$

$$FORK_i = (i.picks\ up\ fork.i \rightarrow i.puts\ down\ fork.i \rightarrow FORK_i \mid (i \ominus 1).picks\ up\ fork.i \rightarrow (i \ominus 1).puts\ down\ fork.i \rightarrow FORK_i)$$

$$LEFT_i = (i.sits\ down \rightarrow i.picks\ up\ fork.i \rightarrow i.puts\ down\ fork.i \rightarrow i.gets\ up \rightarrow LEFT_i)$$

$$RIGHT_i = (i.sits\ down \rightarrow i.picks\ up\ fork.(i \oplus 1) \rightarrow i.puts\ down\ fork.(i \oplus 1) \rightarrow i.gets\ up \rightarrow RIGHT_i)$$

$$PHIL_i = LEFT_i \parallel RIGHT_i$$

$$PHILOS = (PHIL_0 \parallel PHIL_1 \parallel PHIL_2 \parallel PHIL_3 \parallel PHIL_4)$$

$$FORKS = (FORK_0 \parallel FORK_1 \parallel FORK_2 \parallel FORK_3 \parallel FORK_4)$$

$$COLLEGE = PHILOS \parallel FORKS$$

$$U = \bigcup_{i=0}^{4} \{i.gets\ up\} \qquad D = \bigcup_{i=0}^{4} \{i.sits\ down\}$$

$$FOOT_0 = (x : D \rightarrow FOOT_1)$$

$$FOOT_j = (x : D \rightarrow FOOT_{j+1} \mid y : U \rightarrow FOOT_{j-1}) \qquad \text{for } j \in \{1, 2, 3\}$$

$$FOOT_4 = (y : U \rightarrow FOOT_3)$$

$$NEWCOLLEGE = (COLLEGE \parallel FOOT_0)$$

nenastane deadlock

$$U = \cup_{i=0}^{4}\{i.gets\ up\} \qquad D = \cup_{i=0}^{4}\{i.sits\ down\}$$

$(newcollege/s) \neq STOP$ for all $s \in traces(NEWCOLLEGE)$

- $seated(s) := \#(s \upharpoonright D) - \#(s \upharpoonright U)$
- $seated \leq 4$, lebo $FOOT_0$
- $seated =\leq 3$, môže sa posadiť
- nech $seated = 4$
  - niekto je jediaci, môže prestať
  - max 3 zdvihnuté vidličky $\Rightarrow$ aspoň 1 môže zdvihnúť ľavú
  - 4 zdvihnuté $\Rightarrow$ jeden môže zobrať pravú
  - 5 zvdihnutých $\Rightarrow$ jeden je jediaci

bez zdieľanej pamäte – procesy

```
void do_child(int data_pipe[]) {
  int c,rc;

  close(data_pipe[1]);
  while ((rc = read(data_pipe[0], &c, 1)) > 0)
    putchar(c);
  exit(0);
}
```

```
void do_parent(int data_pipe[]) {
  int c,rc;

  close(data_pipe[0]);
  while ((c = getchar()) > 0)
    rc = write(data_pipe[1], &c, 1);
  close(data_pipe[1]);
  exit(0);
}
```

```
int main() {
  int data_pipe[2];
  int pid,rc;

  rc = pipe(data_pipe);
  pid = fork();
  switch (pid) {
    case 0:
      do_child(data_pipe);
    default:
      do_parent(data_pipe);
  }
}
```

v sieti: posielanie správ (sokety)

server

```
int main(int argc, char *argv[]) {
    int sockfd, newsockfd, portno;
    socklen_t clilen;
    char buffer[256];
    struct sockaddr_in serv_addr, cli_addr;
    int n;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    bzero((char *) &serv_addr, sizeof(serv_addr));
    portno = atoi(argv[1]);

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);

    bind(sockfd, (struct sockaddr *) &serv_addr,
        sizeof(serv_addr));
    listen(sockfd,5);
    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd,
        (struct sockaddr *) &cli_addr, &clilen);
    bzero(buffer,256);

    n = read(newsockfd,buffer,255);
    printf("Here is the message: %s\n",buffer);

    n = write(newsockfd,"I got your message",18);
    close(newsockfd);
    close(sockfd);
 }
```

klient

```
int main(int argc, char *argv[]) {
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    char buffer[256];

    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    server = gethostbyname(argv[1]);

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
    (char *)&serv_addr.sin_addr.s_addr,
    server->h_length);
    serv_addr.sin_port = htons(portno);

    connect(sockfd,(struct sockaddr *) &serv_addr,
        sizeof(serv_addr));

    printf("Please enter the message: ");
    bzero(buffer,256);
    fgets(buffer,255,stdin);
    n = write(sockfd,buffer,strlen(buffer));
    bzero(buffer,256);
    n = read(sockfd,buffer,255);
    printf("%s\n",buffer);
    close(sockfd);
    return 0;
}
```

v sieti: posielanie správ (MPI)

```
int main(int argc, char *argv[]) {
    const int tag = 47;
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);

    if (id == 0) {
      MPI_Get_processor_name(msg, &length);
      printf("Hello World from process %d running on %s\n", id, msg);
      for (i=1; i<ntasks; i++) {
        MPI_Recv(msg, 80, MPI_CHAR, MPI_ANY_SOURCE,
                      tag, MPI_COMM_WORLD, &status);
        source_id = status.MPI_SOURCE;
        printf("Hello World from process %d running on %s\n", source_id, msg);
      }
    }
    else {
      MPI_Get_processor_name(msg, &length);
      MPI_Send(msg, length, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    if (id==0) printf("Ready\n");
}
```

sieťový model

- nezávislé zariadenia (procesy, procesory, uzly)
- posielanie správ
- lokálny pohľad (číslovanie portov)
- asynchrónne, spoľahlivé správy
- topológia siete
- wakeup/terminácia

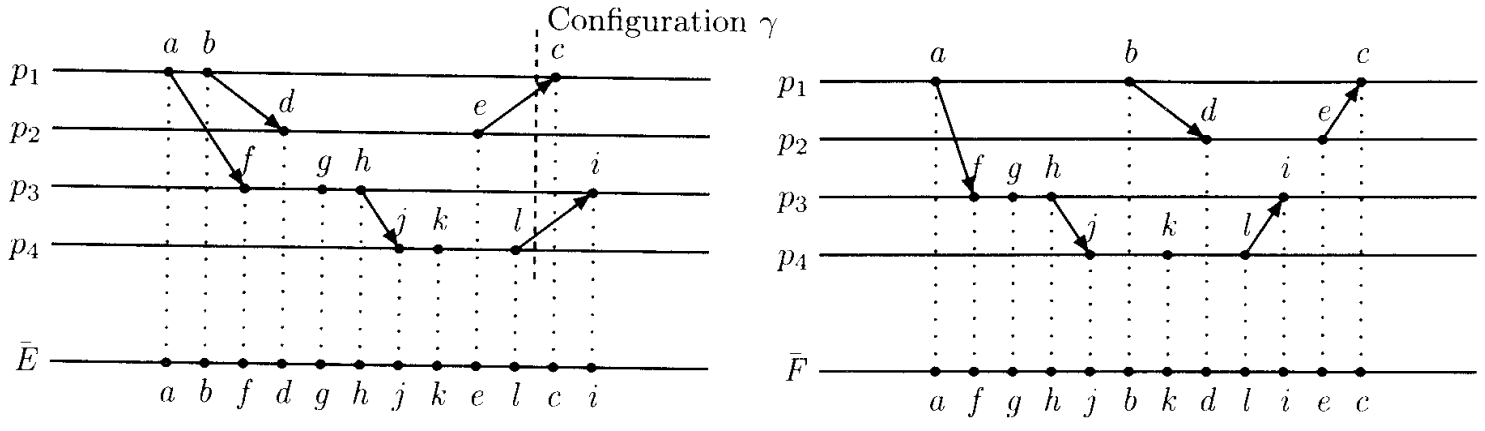
zložitosť: v závislosti od počtu procesorov

- počet správ / bitov
- čas (beh normovaný na max. dĺžku 1)

sieťový model

formálny model: prechodové systémy

- systém je trojica $(\mathcal{C}, \mapsto, \mathcal{I})$
- beh (execution) je postupnosť $\gamma_0 \mapsto \gamma_1 \mapsto \gamma_2 \mapsto \cdots$, kde $\gamma_0 \in \mathcal{I}$
- $\mathcal{C} = \mathcal{C}_L^n \times \mathcal{M}$: stavy systému = stavy procesorov + správy na linkách
- $\mapsto$: krok jedného procesora (výpočet, poslanie správy, prijatie správy)
- fair scheduler?



Sieťový model

formálny model: prechodové systémy

- (ne)závislosť udalostí:
  1) poslanie pred prijatím
  2) časovanie v rámci procesora
  3) tranzitivita
- výpočet (computation): trieda ekvivalencie behov

logické hodiny (Lamport)

var θ_p : integer **init** 0 ;

(* An internal event *)
$\theta_p := \theta_p + 1$;
Change state

(* A send event *)
$\theta_p := \theta_p + 1$:
send $\langle$ **messg**, θ_p $\rangle$; Change state

(* A receive event *)
receive $\langle$ **messg**, θ $\rangle$; $\theta_p := \max(\theta_p, \theta) + 1$;
Change state

sieťový model

horný odhad pre problém

Existuje algoritmus, ktorý **pre všetky** topológie, vstupy, časovania, ...
... pracuje správne a vymení najviac $f(n)$ správ

dolný odhad pre problém

Pre každý algoritmus, **existuje kombinácia** topológie, vstupu, časovania, ...
... že buď nepracuje správne alebo vymení aspoň $f(n)$ správ

sieťový model: broadcast a convergecast (maximum)

const: *deg* : integer
ID : integer
Neigh : [1...*deg*] link
parent : link
var: *msg* : text

Init:

if $parent = NULL$
send $\langle$**dispatch**, *msg*$\rangle$ **to self**

Code:

loop forever

On receipt $\langle$**dispatch**, *new_msg*$\rangle$ from *parent* or self :

msg := *new_msg*
for all $l \in Neigh - \{parent\}$ **do**
send $\langle$**dispatch**, *msg*$\rangle$ **to** *l*
skonči algoritmus

sieťový model: broadcast a convergecast (maximum)

const: *deg* : integer
ID : integer
Neigh : [1...*deg*] link
parent : link
msg : integer
var: *count* : integer
max : integer

Init:

$count = 0$
$max = msg$
if $deg = 1$
  send $\langle$**my_max**, $max\rangle$ **to** *parent*

Code:

loop forever

On receipt $\langle$**my_max**, $x\rangle$ from *Neigh*[*i*]:

  $max = \mathrm{maximum}\{max, x\}$
  $count++$
  if $count = deg - 1$
    send $\langle \mathrm{my\_max}, max\rangle$ **to** *parent*
    skonči algoritmus

cvičenia

- voľba šéfa na (anonymných) stromoch
- voľba šéfa na (anonymnej) mriežke

voľba šéfa: úplné grafy

const: N : integer
ID : integer
$Neigh$: [1...N-1] link
var: $leader$: boolean
$count$: integer
i : integer

Init:

$count := 0$
$leader :=$ **false**

Code:

for $i = 1$ **to** $N - 1$ **do**
    send $\langle$**elect**, $ID\rangle$ **to** $Neigh[i]$
while $count < N - 1$ **wait**
for $i = 1$ **to** $N - 1$ **do**
    send $\langle$**leader**, $ID\rangle$ **to** $Neigh[i]$
$leader :=$ **true**

On receipt $\langle$**elect**, $id_i\rangle$ from $Neigh[i]$:

if $id_i > ID$ **send** $\langle$**accept**$\rangle$ **to** $Neigh[i]$

On receipt $\langle$**accept**$\rangle$ from $Neigh[i]$:

$count++$

On receipt $\langle$**leader**, $id_i\rangle$ from $Neigh[i]$:

Skonči algoritmus

voľba šéfa: úplné grafy II

const: N : integer
ID : integer
$Neigh$: [1...N-1] link
var: $leader$: boolean
$state$: {active, captured, killed}
$level$: integer
$parent$: link
msg : {victory, defeat}
i : integer

Init:

$state := \text{active}$
$level := 0$
$leader :=$ **false**

Code:

for $i = 1$ **to** $N - 1$ **do**
  send $\langle$**capture**, $[level, ID]\rangle$ **to** $Neigh[i]$
  receive $\langle$**accept**$\rangle$ **from** $Neigh[i]$
  $level++$
$leader :=$ **true**
for $i = 1$ **to** $N - 1$ **do**
  send $\langle$**leader**, $ID\rangle$ **to** $Neigh[i]$

Dead:

loop forever

On receipt $\langle$**capture**, $[level_i, id_i]\rangle$ from $Neigh[i]$:

if $state \in \{\text{active}, \text{killled}\}$ **and** $[level_i, id_i] > [level, ID]$
  $state :=$ captured
  $parent := Neigh[i]$
  send $\langle$**accept**$\rangle$ **to** $parent$
  goto Dead
else if $state =$ captured
  send $\langle$**help**, $[level_i, id_i]\rangle$ **to** $parent$
  receive msg **from** $parent$
  if $msg =$ defeat
    send $\langle$**accept**$\rangle$ **to** $Neigh[i]$
    $parent := Neigh[i]$

On receipt $\langle$**help**, $[level_i, id_i]\rangle$ from $Neigh[i]$:

if $[level_i, id_i] < [level, ID]$
  send $\langle$victory$\rangle$ **to** $Neigh[i]$
else
  send $\langle$defeat$\rangle$ **to** $Neigh[i]$
  if $state =$ active
    $state :=$ killed
    goto Dead

On receipt $\langle$**leader**, $id_i\rangle$ from $Neigh[i]$:

Skonči algoritmus

voľba šéfa: úplné grafy II - analýza

Lema 1

V ľubovoľnom výpočte existuje pre každý level $l = 0, ..., N-1$ aspoň jeden proces, ktorý bol počas výpočtu na leveli l.

Lema 2

Nech v je aktívny proces ($state = \mathrm{active}$) s levelom l. Potom existuje l zajatých procesov ktoré patria v (t.j. ich premenná *parent* ukazuje na v).

$\Rightarrow$ práve jeden proces je šéf

Lemma 3

ľubovoľnom výpočte je najviac $N/(l+1)$ procesov, ktoré niekedy dosiahli level l.

$\Rightarrow$ maximálne $\sum_{l=1}^{N-1} \frac{N}{l+1} = N(\mathbf{H}_N - 1) \approx N \log N$ postupov o level (=správ)

dolný odhad

Treba $\Omega(n \log n)$ správ

- "globány" algoritmus
- graf indukovaný novými správami
- udržujem výpočet:
  - jeden súvislý komponent, ostatné vrcholy izolované
  - $e(k)$ – koľko viem vynútiť na komponente k
  - $e(2k+1) = 2e(k) + k + 1$

jednosmerné kruhy

Chang, Roberts

const: ID : integer
l_{in}, l_{out} : link
var: $leader$: integer

Init:

$leader := NULL$

Code:

send $\langle ID \rangle$
wait until $leader <> NULL$

On receipt $\langle i \rangle$:

if $i < ID$ **then send** $\langle i \rangle$
if $i = ID$ **then**
  $leader := ID$
  send $\langle \textbf{leader}, ID \rangle$

On receipt $\langle \textbf{leader}, x \rangle$:

$leader := x$
send $\langle \textbf{leader}, ID \rangle$

- $\Omega(n^2)$ správ v najhoršom prípade
- $O(n \log n)$ v priemernom

Chang Roberts - analýza

Koľkokrát sa pohla správa i?

$$P(i,k) = \frac{\binom{n-i}{k-1} \cdot (i-1)}{\binom{n-1}{k-1} \cdot (n-k)}$$

$$E[X_i] = \sum_{k=1}^{n-i+1} k \cdot P(i,k)$$

$$E[X] = n + \sum_{i=2}^{n} E[X_i] = n + \sum_{i=2}^{n} \sum_{k=1}^{n-i+1} k \cdot P(i,k)$$

Chang Roberts - analýza

lema

$$\sum_{j=k}^{n-1} \frac{j!}{(j-k)!} = k! \binom{n}{k+1}$$

dôkaz

$$\sum_{j=k}^{n-1} \frac{j!}{(j-k)!} = \sum_{j=k}^{n-1} \frac{k!j!}{k!(j-k)!} = k! \sum_{j=k}^{n-1} \binom{j}{k} = k! \binom{n}{k+1}$$

$$n+\sum_{i=2}^{n}\sum_{k=1}^{n-i+1} k\cdot\frac{\binom{n-i}{k-1}\cdot(i-1)}{\binom{n-1}{k-1}\cdot(n-k)} = n+\sum_{j=1}^{n-1}\sum_{k=1}^{j} k\cdot\frac{\binom{j-1}{k-1}\cdot(n-j)}{\binom{n-1}{k-1}\cdot(n-k)} =$$

$$= n+\sum_{j=1}^{n-1}\sum_{k=1}^{j}\frac{k(j-1)!(n-j)(k-1)!(n-k)!}{(k-1)!(j-k)!(n-1)!(n-k)} =$$

$$= n+\sum_{k=1}^{n-1}\left[\frac{k(n-k-1)!}{(n-1)!}\sum_{j=k}^{n-1}\frac{(j-1)!(n-j)}{(j-k)!}\right] =$$

$$= n+\sum_{k=1}^{n-1}\left[\frac{k(n-k-1)!}{(n-1)!}\left(\sum_{j=k}^{n-1}\frac{n(j-1)!}{(j-k)!}-\sum_{j=k}^{n-1}\frac{j!}{(j-k)!}\right)\right] =$$

$$\sum_{j=k}^{n-1}\frac{(j-1)!}{(j-k)!} = (k-1)!\binom{n-1}{k} \text{ a } \sum_{j=k}^{n-1}\frac{j!}{(j-k)!} = k!\binom{n}{k+1}$$

$$= n+\sum_{k=1}^{n-1}\frac{k(n-k-1)!}{(n-1)!}\left[n\cdot(k-1)!\binom{n-1}{k}-k!\binom{n}{k+1}\right] =$$

$$= n+\sum_{k=1}^{n-1}\frac{n}{k+1}$$

dvojsmerné kruhy

Hirschberg-Sinclair

- level l: dobýjať územie 2^l
- $\log n$ levelov
- $n/2^l$ vrcholov na leveli
- každý vrchol pošle 2^l správ

dvojsmerné kruhy

Hirschberg-Sinclair

- level l: dobýjať územie 2^l
- $\log n$ levelov
- $n/2^l$ vrcholov na leveli
- každý vrchol pošle 2^l správ

Franklin

- level l: poraziť susedov (na rovnakom leveli; "synchronizácia")
- $\log n$ levelov
- n správ na level

dvojsmerné kruhy

Hirschberg-Sinclair

- level l: dobýjať územie 2^l
- $\log n$ levelov
- $n/2^l$ vrcholov na leveli
- každý vrchol pošle 2^l správ

Franklin

- level l: poraziť susedov (na rovnakom leveli; "synchronizácia")
- $\log n$ levelov
- n správ na level

Dolev – simulácia na 1-smernom kruhu

- idea: presunúť identitu

dolný odhad

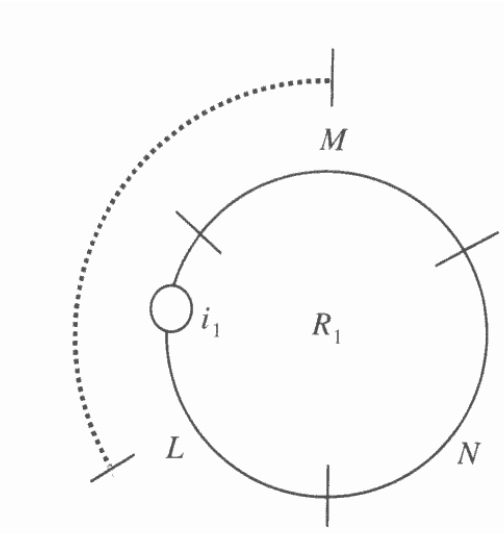
zapojiť do čiary L, vymení sa $C(L)$ správ

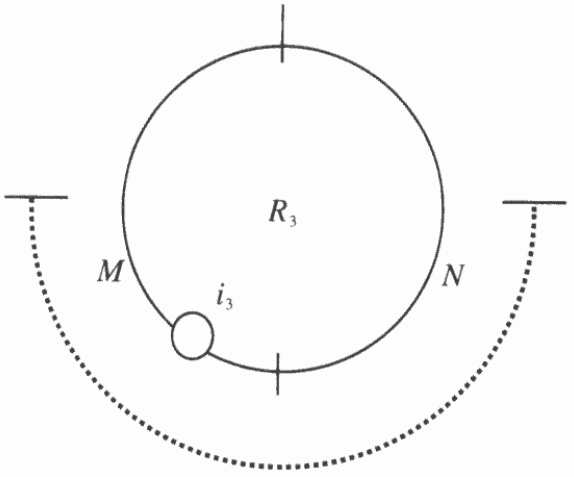
lema

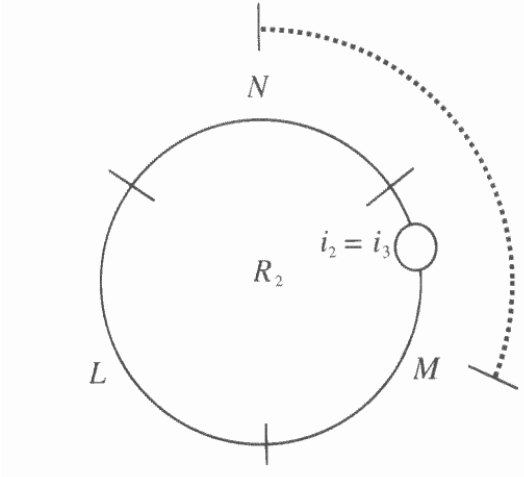
Pre každé r existuje nekonečne veľa čiar dĺžky 2^r, kde $C(L) \geq r2^{r-2}$

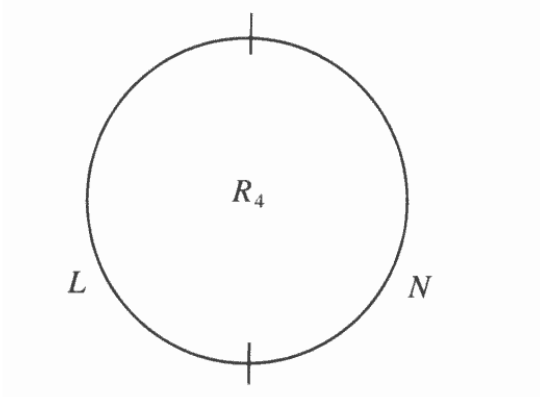
- indukcia
- dve vrecia: vyberám trojice, chcem dve spojiť item pri spojení: dĺžka 2^{r+1}, počet správ $\geq r2^{r-1}$
- ešte treba 2^{r-1} správ; sporom

dolný odhad









GHS

- ľubovoľná topológia
- buduje sa kostra
- "segmenty"
- spájanie po najlacnejšej odchádzajúcej hrane:
  - A menší sa pripojí k väčšiemu
  - B rovnakí sa spoja
  - C väčší čaká
- veľkosť $\Rightarrow$ level

GHS

var $state_p$: $(sleep, find, found)$;
$stach_p[q]$: $(basic, branch, reject)$ for each $q \in Neigh_p$;
$name_p$, $bestwt_p$: real ;
$level_p$: integer ;
$testch_p$, $bestch_p$, $father_p$: $Neigh_p$;
rec_p : integer ;

(1) As the first action of each process, the algorithm must be initialized:
begin let pq be the channel of p with smallest weight ;
$stach_p[q] := branch$; $level_p := 0$;
$state_p := found$; $rec_p := 0$;
send $\langle$ **connect**, 0 $\rangle$ to q
end

(2) Upon receipt of $\langle$ **connect**, L $\rangle$ from q:
begin if $L < level_p$ **then** (* Combine with Rule A *)
begin $stach_p[q] := branch$;
send $\langle$ **initiate**, $level_p$, $name_p$, $state_p$ $\rangle$ to q
end
else if $stach_p[q] = basic$
then (* Rule C *) process the message later
else (* Rule B *) send $\langle$ **initiate**, $level_p + 1, \omega(pq), find$ $\rangle$ to q
end

(3) Upon receipt of $\langle$ **initiate**, L, F, S $\rangle$ from q:
begin $level_p := L$; $name_p := F$; $state_p := S$; $father_p := q$;
$bestch_p := udef$; $bestwt_p := \infty$;
forall $r \in Neigh_p : stach_p[r] = branch \wedge r \neq q$ **do**
send $\langle$ **initiate**, L, F, S $\rangle$ to r ;
if $state_p = find$ **then begin** $rec_p := 0$; $test$ **end**
end

GHS

```
(4) procedure test:
    begin if ∃q ∈ Neigh_p : stach_p[q] = basic then
              begin testch_p := q with stach_p[q] = basic and ω(pq) minimal ;
                    send ⟨ test, level_p, name_p ⟩ to testch_p
              end
          else begin testch_p := udef ; report end
    end

(5) Upon receipt of ⟨ test, L, F ⟩ from q:
    begin if L > level_p then     (* Answer must wait! *)
              process the message later
          else if F = name_p then (* internal edge *)
                  begin if stach_p[q] = basic then stach_p[q] := reject ;
                        if q ≠ testch_p
                           then send ⟨ reject ⟩ to q
                           else test
                  end
               else send ⟨ accept ⟩ to q
    end

(6) Upon receipt of ⟨ accept ⟩ from q:
    begin testch_p := udef ;
          if ω(pq) < bestwt_p
             then begin bestwt_p := ω(pq) ; bestch_p := q end ;
          report
    end

(7) Upon receipt of ⟨ reject ⟩ from q:
    begin if stach_p[q] = basic then stach_p[q] := reject ;
          test
    end
```

GHS

(8) **procedure** *report*:
begin if $rec_p = \#\{q : stach_p[q] = branch \wedge q \neq father_p\}$
and $testch_p = udef$ **then**
begin $state_p := found$; send $\langle$ **report**, $bestwt_p$ $\rangle$ to $father_p$ **end**
end

(9) Upon receipt of $\langle$ **report**, ω $\rangle$ from q:
begin if $q \neq father_p$
then (* reply for **initiate** message *)
begin if $\omega < bestwt_p$ **then**
begin $bestwt_p := \omega$; $bestch_p := q$ **end** ;
$rec_p := rec_p + 1$; *report*
end
else (* pq is the core edge *)
if $state_p = find$
then process this message later
else if $\omega > bestwt_p$
then *changeroot*
else if $\omega = bestwt_p = \infty$ **then stop**
end

(10) **procedure** *changeroot*:
begin if $stach_p[bestch_p] = branch$
then send $\langle$ **changeroot** $\rangle$ to $bestch_p$
else begin send $\langle$ **connect**, $level_p$ $\rangle$ to $bestch_p$;
$stach_p[bestch_p] := branch$
end
end

(11) Upon receipt of $\langle$ **changeroot** $\rangle$:
begin *changeroot* **end**

analýza

správnosť

ukázať, že sa zvolí práve jeden šéf: nenastane deadlock

počet správ

- testovacie správy: jeden test po každej hrane
- kostrové správy: fragment s n_i vrcholmi pri postupe o level $O(n_i)$ správ
- postupy na level l: dizjunktné vrcholy

LE - summary

- pre kruhy, úplné grafy $\Theta(n \log n)$ správ
- bez promisu: $\Theta(n \log n + |E|)$ správ

ako vplývajú zmeny modelu na zložitosť?

- ("komprimovaná") znalosť mapy?
- synchrónnosť?

KKM

- $f(x)$-traverzovanie
- tokeny traverzujú/označujú územia
- levely: keď sa stretnú dva, vznikne nový
- naháňanie

KKM

```
var lev_p          : integer     init −1 ;
    cat_p, wait_p  : P           init udef ;
    last_p         : Neigh_p     init udef ;

begin if p is initiator then
        begin lev_p := lev_p + 1 ; last_p := trav(p, lev_p) ;
              cat_p := p ; send ⟨ annex, p, lev_p ⟩ to last_p
        end ;
      while ... (* Termination condition, see text *) do
        begin receive token (q, l) ;
              if token is annexing then t := A else t := C ;
              if l > lev_p then (* Case I *)
                begin lev_p := l ; cat_p := q ;
                      wait_p := udef ; last_p := trav(q, l) ;
                      send ⟨ annex, q, l ⟩ to last_p
                end
              else if l = lev_p and wait_p ≠ udef then (* Case II *)
                begin wait_p := udef ; lev_p := lev_p + 1 ;
                      last_p := trav(p, lev_p) ; cat_p := p ;
                      send ⟨ annex, p, lev_p ⟩ to last_p
                end
              else if l = lev_p and last_p = udef then (* Case III *)
                wait_p := q
              else if l = lev_p and t = A and q = cat_p then (* Case IV *)
                begin last_p := trav(q, l) ;
                      if last_p = decide
                        then p announces itself leader
                        else send ⟨ annex, q, l ⟩ to last_p
                end
              else if l = lev_p and ((t = A and
                   q > cat_p) or t = C) then (* Case V *)
                begin send ⟨ chase, q, l ⟩ to last_p ; last_p := udef end
              else if l = lev_p then (* Case VI *)
                wait_p := q
```

KKM

počet správ

- naháňacie: 1 na vrchol a level, spolu n za level
- objavovacie: $\sum_i f(n_i)$
- ak f je konvexná, t.j. $f(a) + f(b) \leq f(a + b)$, tak je $O(\log n(n + f(n)))$ správ

vplyv synchrónnosti

algoritmy založené na porovnaniach

- ekvivalentné okolia
- c-symetrický reťazec: pre každé $\sqrt{n} \leq l \leq n$ a pre každý segment S je $\left\lfloor \frac{cn}{l} \right\rfloor$ ekvivalentných
- bit-reversal je $1/2$-symetrický
- c-symetrický reťazec, alg. nemôže skončiť po k kolách pre $\left\lfloor \frac{cn}{2k+1} \right\rfloor \geq 2$
- počet správ: $k = \left\lfloor \frac{cn-2}{4} \right\rfloor$, je aspoň $k+1$ kôl
- aspoň $\left\lfloor \frac{cn}{2r-1} \right\rfloor$ aktívnych v r-tom kole pošle správu

vplyv synchrónnosti

algoritmy využívajúce integer ID

- rôzne rýchlosti
- v i-tom kroku test

cvičenie

V toruse rozmerov $n \times n$ (t.j. zacyklenej mriežke) sú na začiatku zobudené dva vrcholy. Napíšte algoritmus, pomocou ktorého sa každý z nich dozvie identifikátor druhého s použitím $O(n)$ správ.

Nájdite asymptoticky optimálny (čo do počtu správ) algoritmus na voľbu šéfa v úplnom bipartitnom grafe $K_{n,n}$. Dokážte jeho zložitosť a optimalitu.

$\star$

broadcasting a voľba šéfa na (ne?)orientovanej hyperkocke s lineárnym počtom správ

odolnosť voči chybám – strata správ

Problém dohody

- synchrónny systém
- známe identifikátory
- každý má na vstupe 0/1
- správy sa môžu strácať
- každý proces sa musí rozhodnúť
- treba zaručiť
  - **Dohoda:** všetky procesy sa rozhodnú na tú istú hodnotu
  - **Terminácia:** každý proces sa rozhodne v konečnom čase
  - **Netrivialita:**
    1. Ak všetci začnú s hodnotou 0, musia sa dohodnúť na 0.
    2. Ak všetci začnú s hodnotou 1 a správy sa nestrácajú, musia sa dohodnúť na 1.

neexistuje deterministické riešenie

- 2 vrcholy, 1 linka
- sporom, nech existuje a trvá r kôl
- výpočet, kde začnú obaja s hodnotou 1 a nestrácajú sa správy
- dohodnú sa na 1
- stratí sa posledná správa, jeden z nich to nezistí
- výpočet, kde neprejde ani jedna správa a dohodnú sa na 1
- jeden z nich dostane na vstup 0
- aj druhý

randomizované riešenie (úplný graf)

komunikačný pattern

zoznam trojíc (i, j, t): v čase t sa nestratí správa z $i \mapsto j$

(fixný) adversary = vstup a komunikačný pattern
Dohoda: $Pr[$ nejaké dva procesy sa rozhodnú na rôznu hodnotu$] \leq \varepsilon$

algoritmus s $\varepsilon = 1/r$

daný adversary γ: dvojice (i, t), kde i-procesor, t-čas majme usporiadanie:

1. $(i, t) \leq_\gamma (i, t')$, kde $t \leq t'$
2. ak $(i, j, t) \in \gamma$, tak $(i, t-1) \leq_\gamma (j, t)$
3. tranzitivita

úroveň informovanosti

1. $level_\gamma(i, 0) = 0$
2. ak $t > 0$ a existuje $j \neq i$ také, že $(j, 0) \not\leq_\gamma (i, t)$, tak $level_\gamma(i, t) = 0$
3. nech l_j je $\max\{level_\gamma(j, t') \mid (j, t') \leq_\gamma (i, t)\}$
   potom $level_\gamma(i, t) = 1 + \min\{l_j \mid j \neq i\}$

algoritmus

- prvý proces vygeneruje náhodný kľúč
- procesy si počítajú level
- rozhodnutie 1, ak všetci majú 1 a môj level je aspoň kľúč

$rounds := rounds + 1$
let (L_j, V_j, k_j) be the message from j, for each j from which a message arrives
if some $k_j \neq undefined$ then $key := k_j$
for all $j \neq i$ do
    if some $V_{i'}(j) \neq undefined$ then $val(j) := V_{i'}(j)$
    if some $L_{i'}(j) > level(j)$ then $level(j) := \max\{L_{i'}(j)\}$
$level(i) := 1 + \min\{level(j) : j \neq i\}$
if $rounds = r$ then
    if $key \neq undefined$ and $level(i) \geq key$ and $val(j) = 1$ for all j then
        $decision := 1$
    else $decision := 0$

dôkaz

- **Dohoda:** $Pr[$ nejaké dva procesy sa rozhodnú na rôznu hodnotu$] \leq \varepsilon$
- **Terminácia:** každý proces sa rozhodne v konečnom čase
- **Netrivialita:**
  1. Ak všetci začnú s hodnotou 0, musia sa dohodnúť na 0.
  2. Ak všetci začnú s hodnotou 1 a správy sa nestrácajú, musia sa dohodnúť na 1.

terminácia a netrivialita sú zrejmé
pre fixný pattern, aká je pravdepodobnosť nezhody?
levely sa líšia max o 1, preto jediný problém je ak $key = \max\{l_i\}$

dolný odhad

ľubovoľný r-kolový algoritmus má pravdepodobnosť nezhody aspoň $\frac{1}{r+1}$

orez

pre adversary B s patternom γ a proces i, $B' = \mathit{prune}(B, i)$

1. ak $(j, 0) \leq_\gamma (i, r)$ tak sa vstup j zachová, inak znuluje
2. trojica (j, j', t) je v kom. patterne B', akk je v γ a $(j', t) \leq_\gamma (i, r)$

$P^B[i \text{ sa rozhodne 1}] = P^{\mathit{prune}(B,i)}[i \text{ sa rozhodne 1}]$

lema

Ak majú na vstupe všetci 1, $P[i \text{ sa rozhodne 1}] \leq \varepsilon(\mathit{level}(i, r) + 1)$

lema

Ak majú na vstupe všetci 1, $P[i \text{ sa rozhodne 1}] \leq \varepsilon(level(i,r)+1)$

- indukcia na $level(i,r)$: nech $level(i,r) = 0$:
- $B' = prune(B,i) = prune(B',i)$
- $P^B[i \text{ sa rozhodne 1}] = P^{B'}[i \text{ sa rozhodne 1}]$
- od j-čka neprišla správa, $B'' = prune(B',j) = prune(B'',j)$ je triviálny adversary
- $P^{B'}[j \text{ sa rozhodne 1}] = P^{B''}[j \text{ sa rozhodne 1}]$
- lenže $P^{B''}[j \text{ sa rozhodne 1}] = 0$, takže $P^{B'}[j \text{ sa rozhodne 1}] = 0$
- pst' nezhody je ε, $\Rightarrow |P^{B'}[i \text{ sa rozhodne 1}] - P^{B'}[j \text{ sa rozhodne 1}]| \leq \varepsilon$
- preto $P^{B'}[i \text{ sa rozhodne 1}] \leq \varepsilon$ a $P^B[i \text{ sa rozhodne 1}] \leq \varepsilon$

lema

Ak majú na vstupe všetci 1, $P[i \text{ sa rozhodne 1}] \leq \varepsilon(\mathit{level}(i,r)+1)$

- indukcia na $\mathit{level}(i,r)$: nech $\mathit{level}(i,r) > 0$
- $B' = \mathit{prune}(B,i) = \mathit{prune}(B',i)$
- existuje j, že $\mathit{level}_{B'}(j,r) \leq l-1$
- podľa i.p. $P^{B'}[j \text{ sa rozhodne 1}] \leq \varepsilon(\mathit{level}(j,r)+1) \leq \varepsilon l$
- pst' nezhody je ε, $\Rightarrow |P^{B'}[i \text{ sa rozhodne 1}] - P^{B'}[j \text{ sa rozhodne 1}]| \leq \varepsilon$
- preto $P^{B'}[i \text{ sa rozhodne 1}] \leq \varepsilon(l+1)$ a $P^{B}[i \text{ sa rozhodne 1}] \leq \varepsilon(l+1)$

chyby procesov – stop chyby

Problém dohody

- synchrónny systém
- známe identifikátory
- každý má na vstupe 0/1
- proces môže havarovať (uprostred posielania správ)
- maximálne f havarovaných procesov
- každý proces sa musí rozhodnúť
- treba zaručiť
  - **Dohoda:** všetky procesy (ktoré nehavarovali) sa rozhodnú na tú istú hodnotu
  - **Terminácia:** každý proces (ktorý nehavaroval) sa rozhodne v konečnom čase
  - **Netrivialita:** ak všetci začnú s rovnakou hdonotou i, musia sa dohodnúť na i.

chyby procesov – stop chyby

algoritmus

flood počas $f + 1$ kôl; ak je iba jedna hodnota, rozhodni sa, inak (default) 0

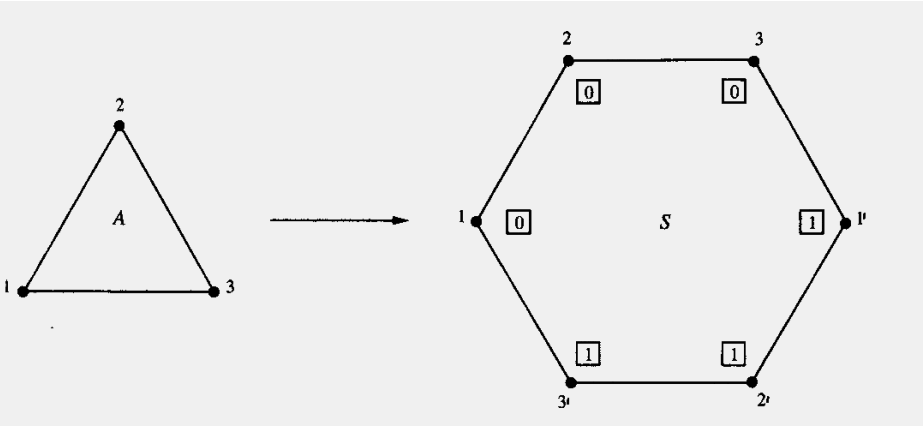
- existuje kolo, v ktorom nikto nehavaruje; potom sa udržiavajú rovnaké hodnoty
- $(f + 1)n^2$ správ
- zlepšenie: posielať iba keď sa zmení hodnota $\Rightarrow O(n^2)$ správ

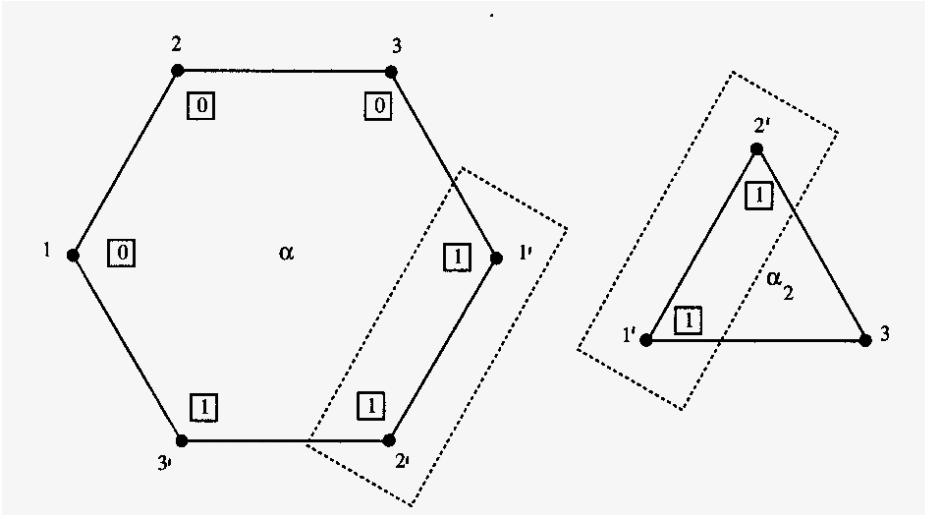
chyby procesov – byzantínske chyby

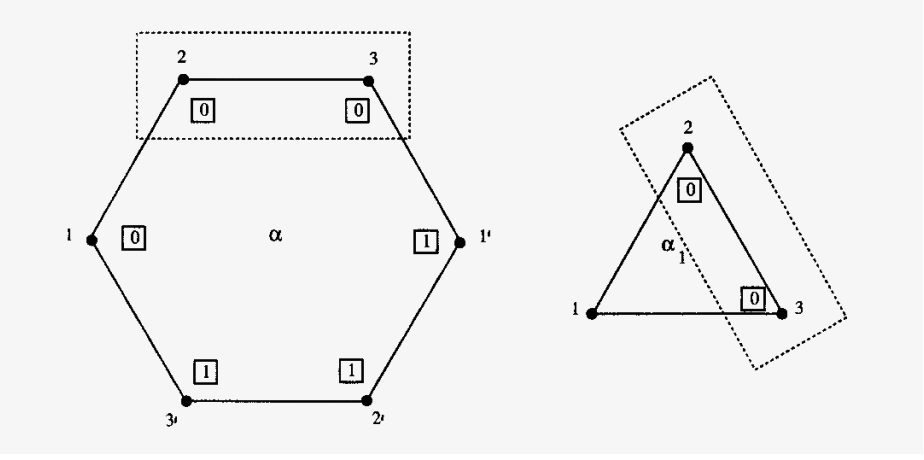
Problém dohody

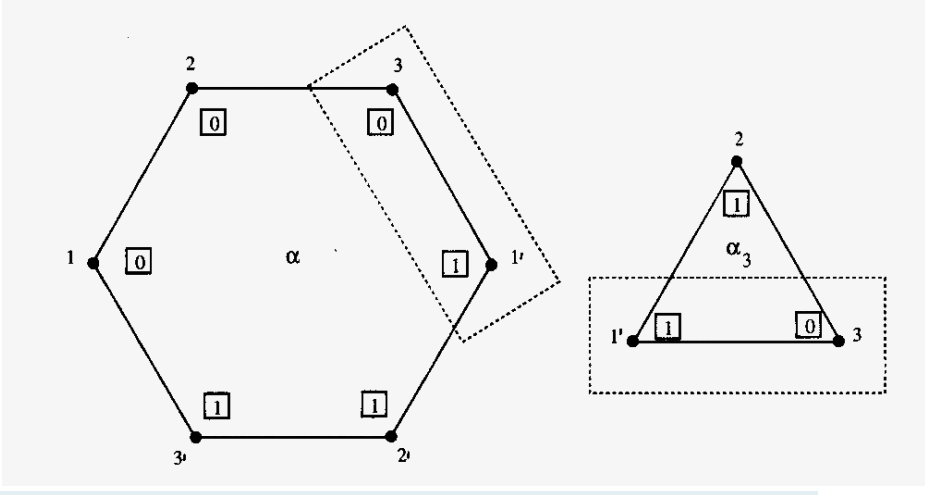
- synchrónny systém
- známe identifikátory
- každý má na vstupe 0/1
- niektoré procesy sú *zlé*
- maximálne f zlých procesov
- každý proces sa musí rozhodnúť
- treba zaručiť
  - **Dohoda:** všetky dobré procesy sa rozhodnú na tú istú hodnotu
  - **Terminácia:** každý dobrý proces sa rozhodne v konečnom čase
  - **Netrivialita:** ak všetci dobrí začnú s rovnakou hdonotou i, všetci dobrí sa musia dohodnúť na i.

dolný odhad na počet zlých: jeden zlý spomedzi troch



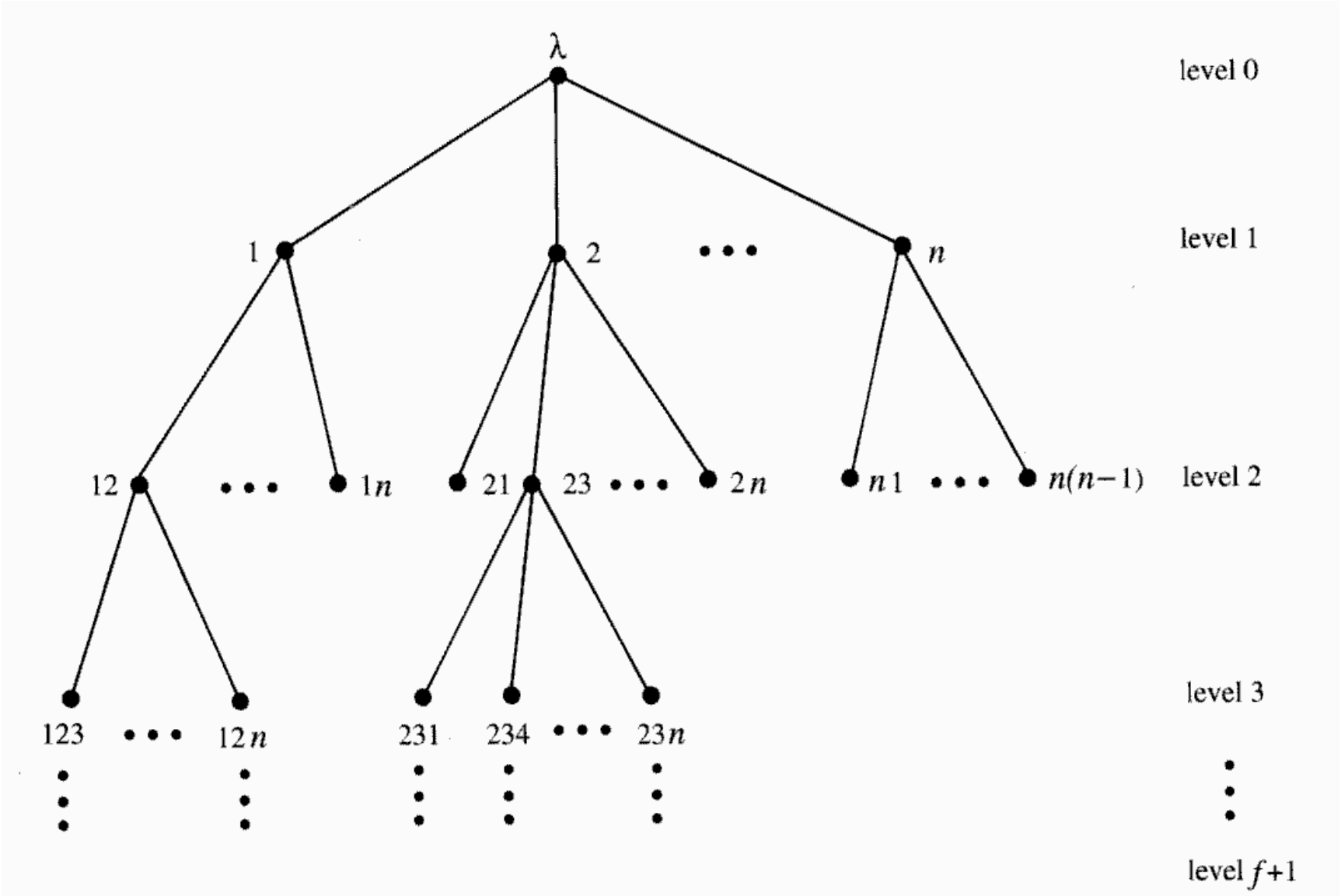






pre viac: simulácia

EIG algoritmus



newval(x): väčšina z *newval*(xj)

dôkaz

lema

Po $f + 1$ kolách platí: nech j, j, k, $i \neq j$ sú tri dobré procesy. Potom $val(xk)_i = val(xk)_j$ pre všetky x.

lema

Po $f + 1$ kolách platí: nech k je dobrý proces. Potom existuje v, že $val(xk)_i = newval(xk)_i = v$ pre všetky dobré procesy i

lema

keď všetci začnú s rovnakou hodnotou, musia sa na nej dohodnúť

dôkaz

vrchol x je *spoľahlivý*, ak všetky dobré procesy i majú po $f+1$ kolách $newval(x)_i = v$ pre nejaké v

lema

Po $f+1$ kolách je na každej ceste z koreňa do listu spoľahlivý vrchol

lema

Po $f+1$ kolách: ak existuje pokrytie podstromu vo vrchole x dobrými vrcholmi, potom x je dobrý.

polynomiálny počet správ

konzistentný broadcast

- ak dobrý proces i poslal (m, i, r) v kroku r, dobrí ju akceptujú najneskôr v $r + 1$
- ak dobrý proces i neposlal (m, i, r) v kroku r, nikto dobrý ju neakceptuje
- ak je správa (m, i, r) akceptovaná dobrým j v r', najneskôr v $r' + 1$ ju akc. všetci dobrí

polynomiálny počet správ

algoritmus

- i pošle $(init, m, i, r)$ v kole r
- ak dobrý dostane $(init, m, i, r)$ v kole r, pošle $(echo, m, i, r)$ všetkým dobrým v kole $r + 1$
- ak pred kolom $r' \geq r + 2$ dostane dobrý od $f + 1$ echo, pošle $(echo, m, i, r)$ v r'
- ak dostal echo od $n - f$, akceptuje

polynomiálny počet správ

dohoda

- dvojkrokové fázy
- v prvom kole bcastujú všetci s 1
- v kole $2s - 1$ pošlú tí, čo akceptovali od $f + s - 1$ a ešte nebcastovali
- ak po $2(f + 1)$ kolách i akceptoval od $2f + 1$ procesov, tak 1, inak 0

routovanie

cieľ: doručovať srávy medzi ľubovoľnou dvojicou

modely

- destination based
- splittable
- connections (wormhole)
- buffering
- selfish

ciele

- statické váhy (najkratšie cesty)
- dynamické váhy (hot potato)
- deadlock

najkratšie cesty

```
begin (* Initialize S to ∅ and D to ∅-distance *)
        S := ∅ ;
        forall u, v do
                if u = v then D[u, v] := 0
                else if uv ∈ E then D[u, v] := ω_uv
                else D[u, v] := ∞ ;
        (* Expand S by pivoting *)
        while S ≠ V do
           (* Loop invariant: ∀u, v : D[u, v] = d^S(u, v) *)
           begin pick w from V \ S ;
                   (* Execute a global w-pivot *)
                   forall u ∈ V do
                           (* Execute a local w-pivot at u *)
                           forall v ∈ V do
                                   D[u, v] := min ( D[u, v], D[u, w] + D[w, v] ) ;
                   S := S ∪ {w}
           end     (* ∀u, v : D[u, v] = d(u, v) *)
end
```

najkratšie cesty

```
var S_u   : set of nodes ;
    D_u   : array of weights ;
    Nb_u  : array of nodes ;

begin S_u := ∅ ;
      forall v ∈ V do
            if v = u
              then begin D_u[v] := 0 ; Nb_u[v] := udef end
            else if v ∈ Neigh_u
              then begin D_u[v] := ω_uv ; Nb_u[v] := v end
            else begin D_u[v] := ∞ ; Nb_u[v] := udef end ;
      while S_u ≠ V do
        begin pick w from V \ S_u ;
              (* All nodes must pick the same node w here *)
              if u = w
                then "broadcast the table D_w"
                else "receive the table D_w"
              forall v ∈ V do
                    if D_u[w] + D_w[v] < D_u[v] then
                      begin D_u[v] := D_u[w] + D_w[v] ;
                            Nb_u[v] := Nb_u[w]
                      end ;
              S_u := S_u ∪ {w}
        end
end
```

najkratšie cesty

```
var S_u   : set of nodes ;
    D_u   : array of weights ;
    Nb_u  : array of nodes ;

begin S_u := ∅ ;
      forall v ∈ V do
            if v = u
              then begin D_u[v] := 0 ; Nb_u[v] := udef end
            else if v ∈ Neigh_u
              then begin D_u[v] := ω_uv ; Nb_u[v] := v end
            else begin D_u[v] := ∞ ; Nb_u[v] := udef end ;
      while S_u ≠ V do
        begin pick w from V \ S_u ;
              (* Construct the tree T_w *)
              forall x ∈ Neigh_u do
                    if Nb_u[w] = x then send ⟨ ys, w ⟩ to x
                                   else send ⟨ nys, w ⟩ to x ;
              num_rec_u := 0 ; (* u must receive |Neigh_u| messages *)
              while num_rec_u < |Neigh_u| do
                    begin receive ⟨ ys, w ⟩ or ⟨ nys, w ⟩ message ;
                          num_rec_u := num_rec_u + 1
                    end ;
              if D_u[w] < ∞ then (* participate in pivot round *)
                begin if u ≠ w
                        then receive ⟨ dtab, w, D ⟩ fromthis Nb_u[w] ;
                      forall x ∈ Neigh_u do
                            if ⟨ ys, w ⟩ was received from x
                              then send ⟨ dtab, w, D ⟩ to x ;
                      forall v ∈ V do (* local w-pivot *)
                            if D_u[w] + D[v] < D_u[v] then
                              begin D_u[v] := D_u[w] + D[v] ;
                                    Nb_u[v] := Nb_u[w]
                              end
                end ;
              S_u := S_u ∪ {w}
        end
end
```

Netchange

```
var Neigh_u      : set of nodes ;     (* The neighbors of u *)
    D_u          : array of 0.. N ;   (* D_u[v] estimates d(u, v) *)
    Nb_u         : array of nodes ;   (* Nb_u[v] is preferred neighbor for v *)
    ndis_u       : array of 0.. N ;   (* ndis_u[w, v] estimates d(w, v) *)

Initialization:
    begin forall w ∈ Neigh_u, v ∈ V do ndis_u[w, v] := N ;
          forall v ∈ V do
                begin D_u[v] := N ; Nb_u[v] := udef end ;
          D_u[u] := 0 ; Nb_u[u] := local ;
          forall w ∈ Neigh_u do send ⟨ mydist, u, 0 ⟩ to w
    end

Procedure Recompute (v):
    begin if v = u
            then begin D_u[v] := 0 ; Nb_u[v] := local end
            else begin (* Estimate distance to v *)
                       d := 1 + min{ndis_u[w, v] : w ∈ Neigh_u} ;
                       if d < N then
                         begin D_u[v] := d ;
                               Nb_u[v] := w with 1 + ndis_u[w, v] = d
                         end
                       else begin D_u[v] := N ; Nb_u[v] := udef end
                 end ;
          if D_u[v] has changed then
            forall x ∈ Neigh_u do send ⟨ mydist, v, D_u[v] ⟩ to x
    end
```

```
Processing a ⟨ mydist, v, d ⟩ message from neighbor w:
    { A ⟨ mydist, v, d ⟩ is at the head of Q_wv }
    begin receive ⟨ mydist, v, d ⟩ from w ;
          ndis_u[w, v] := d ; Recompute (v)
    end

Upon failure of channel uw:
    begin receive ⟨ fail, w ⟩ ; Neigh_u := Neigh_u \ {w} ;
          forall v ∈ V do Recompute (v)
    end

Upon repair of channel uw:
    begin receive ⟨ repair, w ⟩ ; Neigh_u := Neigh_u ∪ {w} ;
          forall v ∈ V do
                begin ndis_u[w, v] := N ;
                      send ⟨ mydist, v, D_u[v] ⟩ to w
                end
    end
```

Netchange

```
var Neigh_u   : set of nodes ;   (* The neighbors of u *)
    D_u       : array of 0.. N ; (* D_u[v] estimates d(u, v) *)
    Nb_u      : array of nodes ; (* Nb_u[v] is preferred neighbor for v *)
    ndis_u    : array of 0.. N ; (* ndis_u[w, v] estimates d(w, v) *)

Initialization:
  begin forall w ∈ Neigh_u, v ∈ V do ndis_u[w, v] := N ;
        forall v ∈ V do
              begin D_u[v] := N ; Nb_u[v] := udef end ;
        D_u[u] := 0 ; Nb_u[u] := local ;
        forall w ∈ Neigh_u do send ⟨ mydist, u, 0 ⟩ to w
  end

Procedure Recompute (v):
  begin if v = u
          then begin D_u[v] := 0 ; Nb_u[v] := local end
          else begin (* Estimate distance to v *)
                     d := 1 + min{ndis_u[w, v] : w ∈ Neigh_u} ;
                     if d < N then
                       begin D_u[v] := d ;
                             Nb_u[v] := w with 1 + ndis_u[w, v] = d
                       end
                     else begin D_u[v] := N ; Nb_u[v] := udef end
               end ;
        if D_u[v] has changed then
          forall x ∈ Neigh_u do send ⟨ mydist, v, D_u[v] ⟩ to x
  end
```

```
Processing a ⟨ mydist, v, d ⟩ message from neighbor w:
    { A ⟨ mydist, v, d ⟩ is at the head of Q_wv }
    begin receive ⟨ mydist, v, d ⟩ from w ;
          ndis_u[w, v] := d ; Recompute (v)
    end

Upon failure of channel uw:
    begin receive ⟨ fail, w ⟩ ; Neigh_u := Neigh_u \ {w} ;
          forall v ∈ V do Recompute (v)
    end

Upon repair of channel uw:
    begin receive ⟨ repair, w ⟩ ; Neigh_u := Neigh_u ∪ {w} ;
          forall v ∈ V do
                begin ndis_u[w, v] := N ;
                      send ⟨ mydist, v, D_u[v] ⟩ to w
                end
    end
```

korektnosť

lexikograficky klesá hodnota $[t_0, t_1, \ldots, t_N]$
kde t_i je počet správ $\langle \textbf{mydist}, i \rangle$+ počet dvojíc u, v kde $D_u[v] = i$

hierarchické routovanie

cieľ: minimalizovať počet rozhodnutí

veta

Pre sieť s N vrcholmi stačí $O(\sqrt{N})$ rozhodnutí pri použití 3 farieb.

s-klastre:

- každý je súvislý, pokrývajú všetky vrcholy
- každý obsahuje aspoň s vrcholov a má polomer najviac $2s$

kostra spájajúca centrá klastrov: m listov $\Rightarrow$ $m - 2$ vetvení

hierarchické routovanie

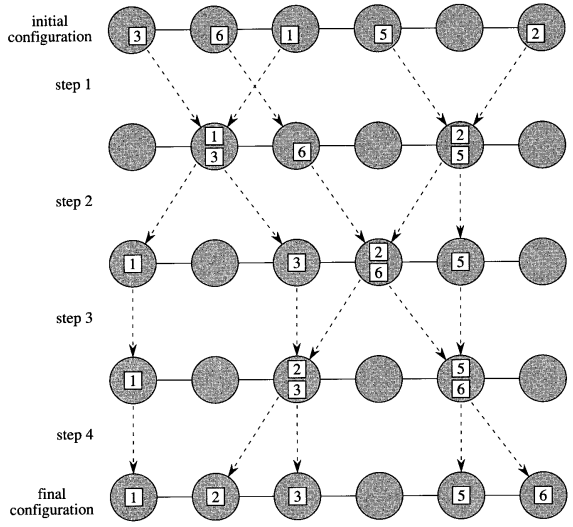
veta

Pre sieť s N a pre $f \leq \log N$ stačí $O(f \cdot N^{1/f})$ rozhodnutí a $2f + 1$ farieb

po i klastrovaniach s parametrom s: m_i listov, max. $m_i - 2$ vetvení $\Rightarrow$
$m_{i+1} = m_i(2/s)$
$m_f + fs$ rozhodnutí
$s \approx 2N^{1/f}$

packet routing

- synchrónny režim
- vrcholy majú pakety (uložené v bufferoch)
- v jednom kroku po jednej linke ide max. jeden paket
- algoritmus = odchádzajúce linky + priorita bufferov
- celkový čas



packet routing na mriežke $\sqrt{N} \times \sqrt{N}$

Každý vrchol má 1 paket, do každého smeruje 1 paket (permutation routing)

Najprv riadok, potom stĺpec. Prednosť má ten s najdlhšou cestou.

analýza: stačí $2\sqrt{N} - 2$ krokov

- po $\sqrt{N} - 1$ krokoch je každý v správnom stĺpci (nebrzdia sa)
- routovanie v stĺpci ide v $\sqrt{N} - 1$ krokoch
  - pre každé i platí: po $N - 1$ krokoch sú koncové pakety na koncových miestach
  - dôvod: zdržujú sa iba navzájom

veľkosť buffra v najhoršom prípade: $2/3\sqrt{N} - 3$

veľkosť buffra: priemerný prípad I

setting

Každý vrchol má jeden paket s **náhodným cieľom**

max. veľkosť buffra $\approx$ počet zahnutí vo vrchole

psť, že aspoň r zahne $\leq \binom{\sqrt{N}}{r} \left(\frac{1}{\sqrt{N}}\right)^r < \left(\frac{e}{r}\right)^r$

pre $r = \frac{e \log N}{\log \log N}$ je psť $o(N^{-2})$

veľkosť buffra: priemerný prípad II

wide-channel: nepredbiehajú sa

lema

psť, že vo wch prejde aspoň $\alpha\Delta/2$ paketov cez hranu e počas $t+1, t+2, \ldots, t+\Delta$ je najviac $e^{(\alpha-1-\alpha \ln \alpha)\Delta/2}$

očakávaný počet paketov na hrane $(i,j) \mapsto (i+1,j)$ je

$$\frac{2i(\sqrt{N}-i)\Delta}{N} \leq \frac{\Delta}{2}$$

chceme ukázať, že s veľkou psťou ich neprejde príliš viac

Černovov odhad

lema

Majme n nezávislých Bernoulihho náh. prem. $X_1, \ldots, X_n$, pričom $Pr[X_k = 1] \leq P_k$. Potom

$$Pr[X \geq \beta P] \leq e^{(1-\frac{1}{\beta}-\ln \beta)\beta P}$$

kde $X = \sum X_i$, $P = \sum P_i$

$$E[e^{\lambda X_k}] \leq 1 + P_k(e^{\lambda} - 1) \leq e^{P_k(e^{\lambda}-1)}$$

$$E[e^{\lambda X}] \leq e^{P(e^{\lambda}-1)}$$

$$P[e^{\lambda X} \geq e^{\lambda \beta P}] \leq \frac{E[e^{\lambda X}]}{e^{\lambda \beta P}} \leq e^{P(e^{\lambda}-1)-\lambda \beta P}$$

veľkosť buffra: priemerný prípad II

lema

Majme n nezávislých Bernoulihho náh. prem. $X_1, \ldots, X_n$, pričom $Pr[X_k = 1] \leq P_k$. Potom $Pr[X \geq \beta P] \leq e^{(1-\frac{1}{\beta}-\ln\beta)\beta P}$ kde $X = \sum X_i$, $P = \sum P_i$

lema

psť, že vo wch prejde aspoň $\alpha\Delta/2$ paketov cez hranu e počas $t+1, t+2, \ldots, t+\Delta$ je najviac $e^{(\alpha-1-\alpha\ln\alpha)\Delta/2}$

očakávaný počet paketov na hrane $(i,j) \mapsto (i+1,j)$ je

$$\frac{2i(\sqrt{N}-i)\Delta}{N} \leq \frac{\Delta}{2}$$

chceme ukázať, že s veľkou psťou ich neprejde príliš viac

$n = 2i\Delta$, $P_k = \frac{\sqrt{N}-i}{N}$, $P = \frac{2i(\sqrt{N}-i)\Delta}{N}$, $\beta = \frac{\alpha N}{4i(\sqrt{N}-i)}$

veľkosť buffra: priemerný prípad II

lema

ak je paket vo vzd. d od hrany e v čase T, a p prejde cez e v čase $T + d + \delta$, potom v každom kroku $[T + d, T + d + \delta]$ prejde paket cez e

lema

ak počas $[T + 1, T + \Delta]$ prejde cez e x paketov v št., tak pre nejaké t prejde $x + t$ paketov cez e v čase $[T + 1 - t, T + \Delta]$ vo wch.

lema

psť, že cez e prejde viac ako $\alpha\Delta/2$ paketov počas konkrétneho okna Δ krokov je najviac $O(e^{(\alpha-1-\alpha \ln \alpha)\Delta/2})$

dosledok

s psťou $1 - O(\frac{1}{N})$ neprejde po e viac ako $c \log N$ paketov v posebe idúcich krokoch, kde $c = \frac{5 \ln 2}{2 \ln 2 - 1} < 9$.

dynamické routovanie

model

V každom kroku sa v každom vrchole s psťou λ narodí paket s náhodným cieľom.

stabilita

Pre $\lambda \geq 4/\sqrt{N}$ je systém nestabilný

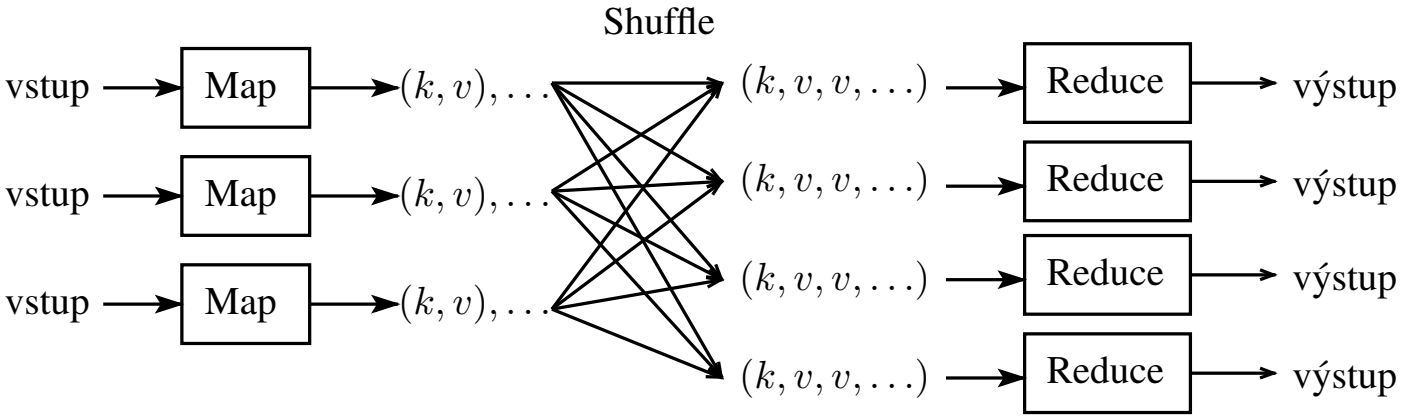
veta

Ak je $\lambda \leq 0.99\frac{4}{\sqrt{N}}$, tak psť zdržania konkrétneho paketu o Δ krokov je $e^{-O(\Delta)}$.
W.h.p. stačí buffer $O(1 + \frac{\log T}{\log N})$.

Apache Hadoop (Google MapReduce)

- odolnosť voči chybám, synchronizácia, scheduling
- menej flexibility: komunikácia Map $\rightarrow$ Shuffle $\rightarrow$ Reduce

- Map: vstup $\rightarrow (k_1, v_1), (k_2, v_2), \ldots$
- Shuffle: $\rightarrow (k_1, v, v, v, \ldots), (k_2, v, v, v, \ldots)$
- Reduce: $(k, v_1, v_2, \ldots) \rightarrow$ výstup



Príklad

	Map		Shuffle		Reduce	
škrtia sa krty	→	(škrtia, 1), (sa, 1), (krty, 1)	...	(škrtia, 1)	→	škrtia: 1
			...	(sa, 1)	→	sa: 1
krt krta škrtí	→	(krt, 1), (krta, 1), (škrtí, 1)	...	(krty, 1)	→	krty: 1
			...	(krt, 1, 1)	→	krt: 2
keď krt krta zaškrtí	→	(keď, 1), (krt, 1), (krta, 1), (zaškrtí, 1)	...	(krta, 1, 1)	→	krta: 2
			...	(škrtí, 1)	→	škrtí: 1
do rána ho zmaškrtí	→	(do, 1), (rána, 1), (ho, 1), (zmaškrtí, 1)	...	(keď, 1)	→	keď: 1
			...	(zaškrtí, 1)	→	zaškrtí: 1
			...	(do, 1)	→	do: 1
			...	(rána, 1)	→	rána: 1
			...	(ho, 1)	→	ho: 1
			...	(zmaškrtí, 1)	→	zmaškrtí: 1

```
Map(String input_line):
  for each word w in input_line: Emit(w, 1);

Reduce(String key, Iterator values):
  int result = 0;
  for each v in values: result += v;
  Emit(key + ": "+ result);
```

Keď treba viacero map-shuffle-reduce fáz

Zoznam slov s > 1000 výskytmi

- 1. MR: slovo $\rightarrow$ počet výskytov
- 2. MR: filter

- Správa pipeline: skript / iný nástroj
- Ošetrenie chýb (reštart)
- Komplikovaná optimalizácia
- $\Rightarrow$ MR je príliš nízkoúrovňová

Flume: Abstrakcia nad MR

- Knižnica: abstrakcia pre paralelizovanú kolekciu dát
- `PCollection<typ>`:
  veľmi veľká, distribuovaná kolekcia
  obmedzený spôsob manipulácie
- `PTable<typ_kľúča, typ_hodnoty>` =
  `PCollection< KV<typ_kľúča, typ_hodnoty> >`

Premenná typu PCollection:

- nedá sa meniť (immutable)
- nedá sa k nej priamo pristupovať (not materialized)
- malá interná reprezentácia

Operácie nad kolekciami

ParDo

- `PCollection<`t_1`> vstup`
- `PCollection<`t_2`> výstup =`
  `vstup.ParDo(new Funkcia())`

```
class Funkcia extends DoFn<t1, t2> {
  public void Do(t1 vstup, EmitFn<t2> emit) {
    t2 výstup = ...
    emit.Emit(výstup)
  }
}
```

- Funkcia beží paralelne $\Rightarrow$ nesmie komunikovať stavom
- Funkcia nesmie mať vedľajšie efekty

Operácie nad kolekciami

GroupByKey

- `PTable<` t_1`,` t_2 `> vstup`
- `PTable<` t_1`, list<`t_2`> > výstup =`
  `vstup.GroupByKey()`

Flatten

- `PCollection<`t`> vstup1, ..., vstupK`
- `PCollection<`t`> výstup =`
  `Flatten(vstup1, ..., vstupK)`

Operácie nad kolekciami

CombineValues

- `PTable<`t_1, t_2`> vstup`
- `PTable<`t_1, t_2`> výstup =`
  `vstup.CombineValues(new Funkcia())`

```
class Funkcia extends CombineFn<t1, t2> {
  public t2 Combine(t1 kľúč, list<t2> hodnoty) {
    t2 výstup = ...
    return výstup
  }
}
```

- Ako `GroupByKey + ParDo`
- Asociativita; nemusí vidieť všetky hodnoty naraz

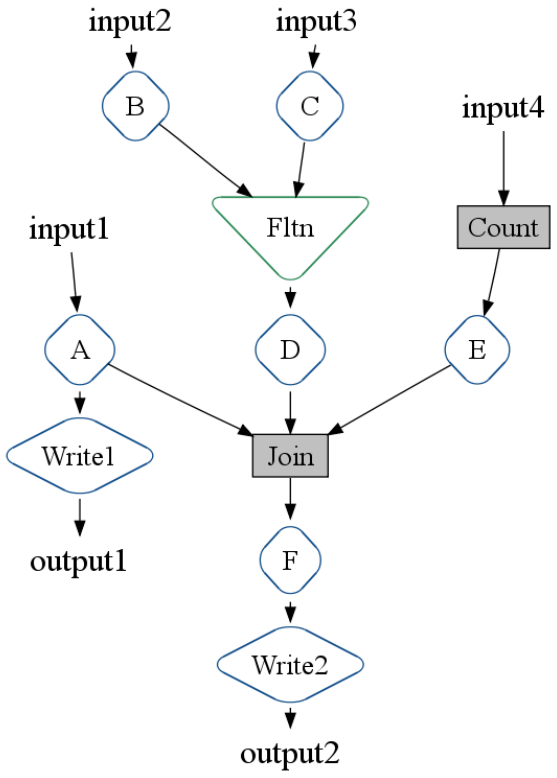
Ako prebieha výpočet

- `ReadFromFile()`, `ParDo()`, `GroupByKey()`, `...`, `WriteToFile()`
- `Flume::Run()`

Ako prebieha výpočet

- `ReadFromFile()`, `ParDo()`, `GroupByKey()`, ..., `WriteToFile()`
- `Flume::Run()`

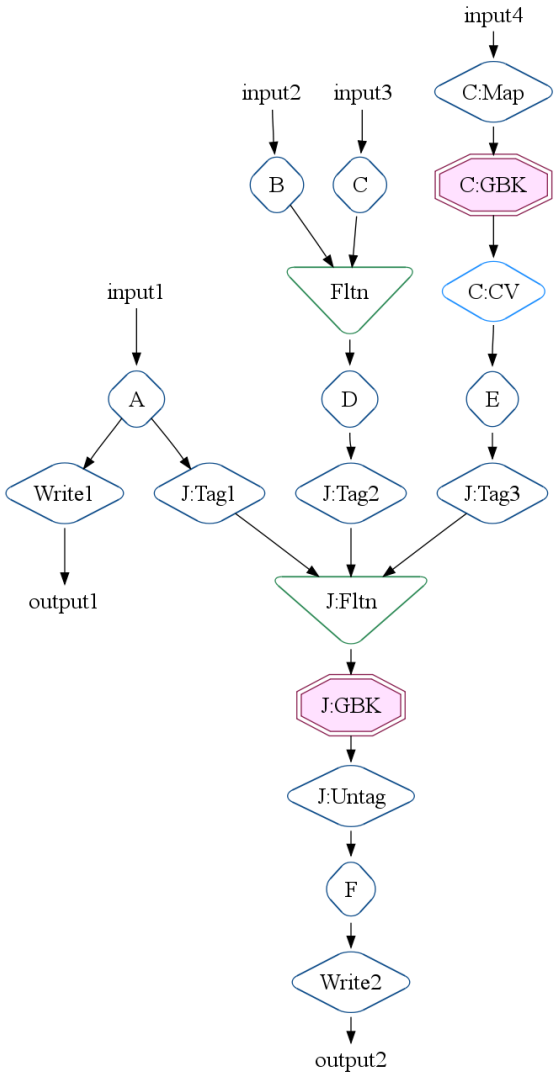
- Strom operácií
- Lazy evaluation



Ako prebieha výpočet

- `ReadFromFile()`, `ParDo()`, `GroupByKey()`, ..., `WriteToFile()`
- `Flume::Run()`

- Strom operácií
- Lazy evaluation



$\mathcal{MRC}$ teória

$\mathcal{MRC}^i$

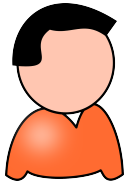
- algoritmus je postupnosť $\langle \mu_1, \rho_1, \ldots, \mu_R, \rho_R \rangle$
- μ_r je (randomizovaný) RAM s $O(n^{1-\varepsilon})$ pamäťou a poly časom
- ρ_r rovnako
- pamäť $\sum_{\langle k;v \rangle}(|k| + |v|)$ z každého reducera je $O(n^{2-2\varepsilon})$
- počet kôl je $O(\log^i n)$

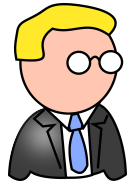
prefix sums

- $[1, 4, 0, 0, 3, 0, 2]$:
  in: $\{(1, 1), (2, 4), (5, 3), (7, 2)\}$,
  out: $\{(1, 1, 1), (2, 4, 5), (5, 3, 8), (7, 2, 10)\}$
- blok m^ε

zdieľané dáta

- HW vs. abstrakcia na sieťou
- úskalia

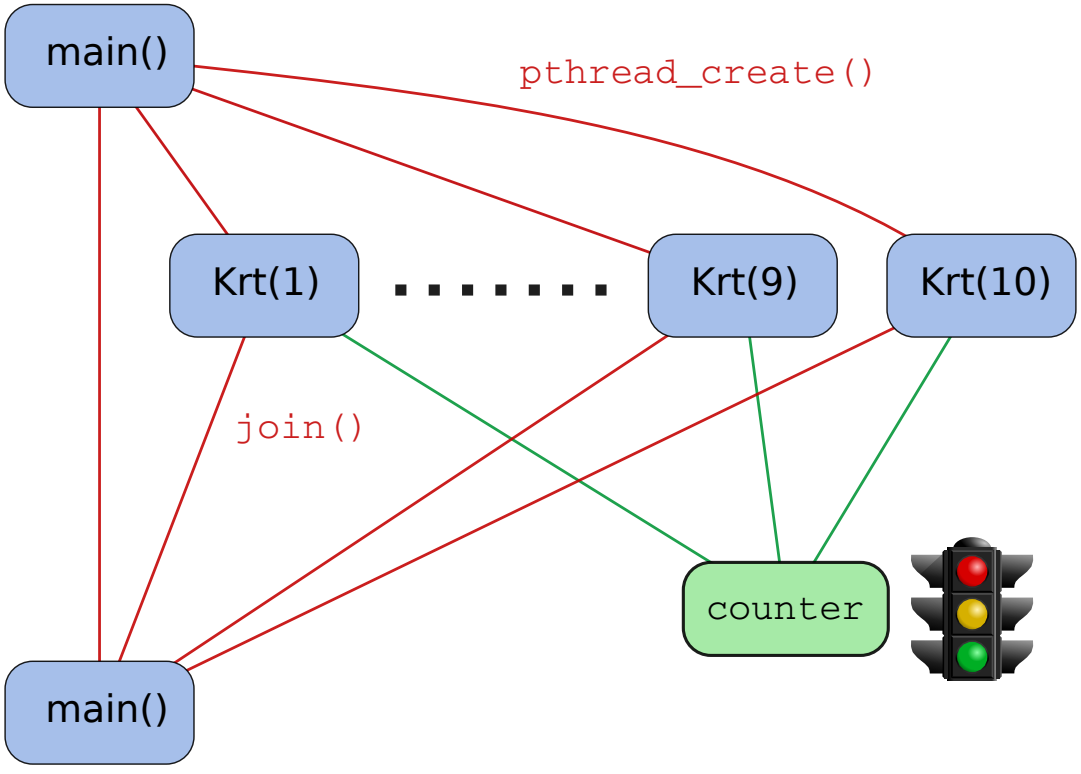




suma			suma	
1000	read(suma)	read(suma)	1000	1000
		suma := suma + 300	1300	
900	suma:=suma - 100			
		write(suma)	1300	1300
900	write(suma)			900

zdieľaná pamäť – thready

- dynamické vytváranie
- asynchrónne (`join`)
- riadenie prístupu (mutex/semafór/...)



thready (POSIX)

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex1 =
    PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;


void *Krt(void *p) {
  pthread_mutex_lock( &mutex1 );
  counter += *(int *)p;
  pthread_mutex_unlock( &mutex1 );
}
```

```
main()  {
  pthread_t thread_id[NTHREADS];
  int i, j, param[NTHREADS];

  for(i=0; i < NTHREADS; i++) {
    param[i]=i;
    pthread_create( &thread_id[i],
      NULL, Krt, param + i );
  }

  for(j=0; j < NTHREADS; j++)
    pthread_join( thread_id[j], NULL);
}
```

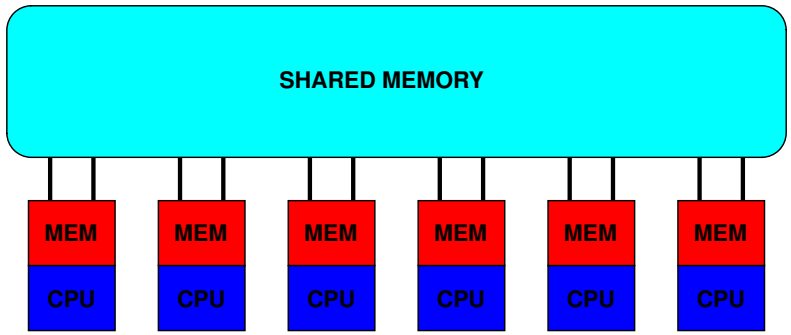
thready (POSIX)

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex1 =
    PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;


void *Krt(void *p) {
  pthread_mutex_lock( &mutex1 );
  counter += *(int *)p;
  pthread_mutex_unlock( &mutex1 );
}
```

```
main()  {
  pthread_t thread_id[NTHREADS];
  int i, j, param[NTHREADS];

  for(i=0; i < NTHREADS; i++) {
    param[i]=i;
    pthread_create( &thread_id[i],
      NULL, Krt, param + i );
  }

  for(j=0; j < NTHREADS; j++)
    pthread_join( thread_id[j], NULL);
}
```

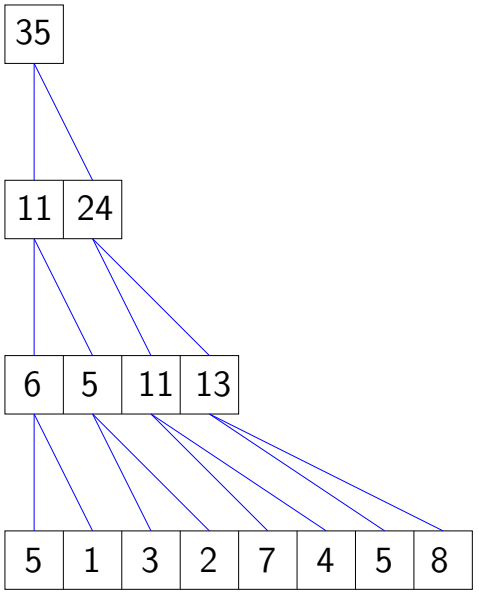
- deadlock
- synchronizácia
- debugovanie

PRAM



- synchrónne procesory
- kedy je problém paralelizovateľný?

```
global read( A(i), a )
global write( a, B(i) )
for h = 1 to log n do
    if (i ≤ n/2^h)
        global read( B(2i − 1),x )
        global read( B(2i),y )
        z := x + y
        global write(z, B(i))
if i = 1 global write( z,S )
```



WTF – Work-Time Framework

- písať algoritmy pre ľubovoľný počet procesorov
- **work** – počet použitých operácií

for $l \leq i \leq u$ **pardo** statement

```
for 1 ≤ i ≤ n pardo
    B(i) := A(i)
for h = 1 to log n do
    for 1 ≤ i ≤ n/2^h pardo
        B(i) := B(2i − 1) + B(2i)
S := B(1)
```

```
global read( A(i), a )
global write( a, B(i) )
for h = 1 to log n do
    if (i ≤ n/2^h)
        global read( B(2i − 1),x )
        global read( B(2i),y )
        z := x + y
        global write(z, B(i))
if i = 1 global write( z,S )
```

WT algoritmus s $T(n)$, $W(n)$ sa dá simulovať na PRAMe s p procesormi v čase $\left\lfloor \frac{W(n)}{p} \right\rfloor + T(n)$

PRAM

- EREW / CREW / CRCW
- príklad: pozícia prvej jednotky na common CRCW PRAM

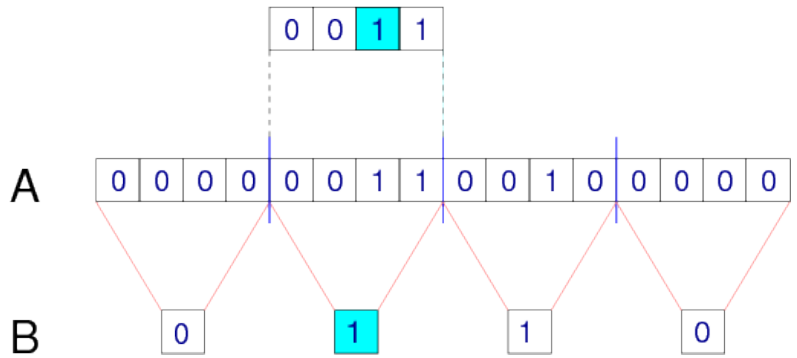
```
for i = 1 to n pardo B(i) := 0
for i = 1 to n^2 pardo
    if A(i) = 1 then B(⌈i/n⌉) := 1
```

```
for i = 1 to n pardo
    for j = 1 to i − 1 pardo
        if B(j) = 1 then B(i) := 0
```

```
for i = 1 to n pardo
    if B(i) = 1 then D := i − 1
```

```
for i = 1 to n pardo
    for j = 1 to i − 1 pardo
        if A(D + j) = 1 then
            A(D + i) := 0
```

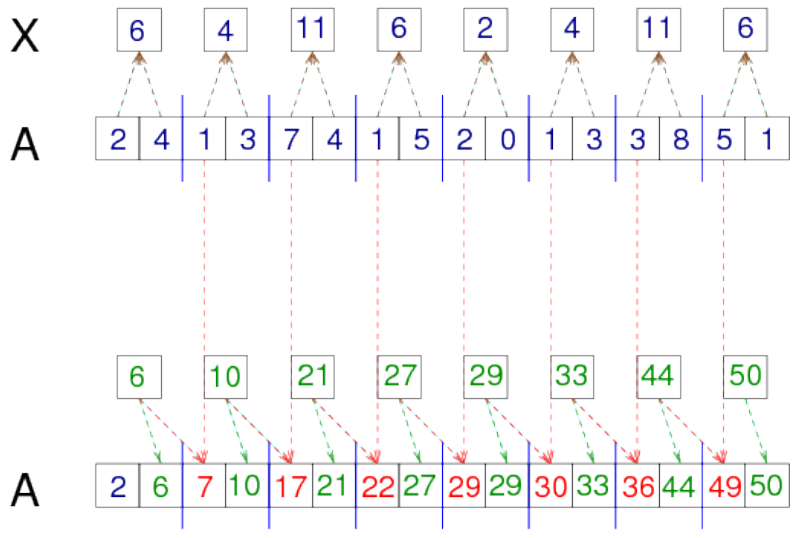
```
for i = 1 to n pardo
    if A(D + i) = 1 then X := D + i
```



Prefix Sum Problem

Dané pole A dĺžky n. Pre každý index i nájsť $\sum_{j=0}^{i} a_i$

```
for 1 ≤ i ≤ n/2 pardo
    x_i := a_{2i-1} + a_{2i}

z :=prefix_sums( X, n/2 )

for 1 ≤ i ≤ n/2 pardo
    a_i := { z_i/2          ak i = 2k
           { 1              ak i = 1
           { z_{(i-1)/2} + a_i   ak i = 2k+3
```



práca $= O(n)$
čas $= O(\log n)$

techniky - accelerated cascading

maximum: CRCW $W = O(n^2)$, $T = O(1)$

pre každé i paralelne zistiť, či a_i je maximum (viď. index prvej "1")

dvojlogaritmická rekurzia: $W = O(n \log\log n)$, $T = O(\log\log n)$

- deliť na $\sqrt{n}$ intervalov
- $T(n) = T(\sqrt{n}) + c$
- na spájanie $O(1)$-čas

kombinácia

- najprv sekvenčne spracovať úseky $O(\log\log n)$
- potom dvojlogaritmický algoritmus

techniky - pointer jumping

spájaný zoznam : implemenácia v poli

list ranking

$S(i) = S(S(i))$

Uvažujme základný model z prednášky, kde procesy majú identifikátory a komunikujú posielaním asynchrónnych spoľahlivých správ, pričom na začiatku sú zobudené všetky procesy. Procesy sú zapojené do kruhu, pričom každý má dva porty očíslované $0, 1$ (ľubovoľným spôsobom). Algoritmus Hirshberg-Sinclair z prednášky volil šéfa postupným dobýjaním väčších a väčších území takto (ID je identifikátor procesu):

1. `init: send` $\langle$`msg,`ID`,0,0`$\rangle$ `to both neighbors`
2. `upon receipt` $\langle$`msg,`$j, k, d$$\rangle$ `from port` p
3. `if` $j > ID$ `and` $d < 2^k$ `then send` $\langle$`msg,`$j, k, d+1$$\rangle$ `to port` $1-p$
4. `if` $j > ID$ `and` $d = 2^k$ `then send` $\langle$`reply,`$j, k$$\rangle$ `to port` p
5. `if` $j = ID$ `then announce self as leader`
6. `upon receipt` $\langle$`reply,`$j, k$$\rangle$ `from port` p
7. `if` $ID \neq j$ `then send` $\langle$`reply,`$j, k$$\rangle$ `to port` $1-p$
8. `else if already received` $\langle$`reply,`$j, k$$\rangle$
9. `then send` $\langle$`msg,`$ID, k+1, 0$$\rangle$ `to both neighbors`

Ostane algoritmus korektný, ak sa na riadkoch 3. a 4. namiesto 2^k použije $k+1$? Odvoďte (a dokážte) jeho asymptotickú zložitosť.

Graf G s $n > 4$ vrcholmi nazveme *takmer úplný*, ak každý vrchol grafu G má aspoň $n - 3$ susedov. Navrhnite asymptoticky optimálny algoritmus (hint: nie, GHS to nie je) na voľbu šéfa úplných grafoch (v základnom modeli, t.j. asynchrónne, bez zmyslu pre orientáciu, s identifikátormi, ...), ktorý používa $O(n \log n)$ správ. Určite zložitosť (počet správ) Vášho algoritmu.

Na prednáške bol popísaný algoritmus na riešenie problému dohody so stop-chybami (každý nezhavarovaný proces skončí, všetky nezhavarované procesy sa dohodnú, ak všetky procesy začínajú s rovnakou hodnotou, musia sa dohodnúť na nej), ktorý je odolný voči f výpadkom. Algoritmus pracoval $f + 1$ kôl. Ukážte, že žiaden algoritmus, ktorý by pracoval iba f kôl, nemôže byť korektný.

Uvažujme $n > 3$ procesov spojených obojsmernými linkami do kruhu. Procesy majú identifikátory, štartujú naraz a pracujú v asynchrónnom režime. V sieti je zvolený šéf, t.j. každý proces má logickú premennú `som_sef`, ktorá má hodnotu `true` práve v jednom procese.
V sieti je jedna chybná linka, ktorá nikdy neprepúšťa správy: ak ľubovoľný z koncových vrcholov chybnej linky po nej pošle správu, správa sa bez akejkoľvek notifikácie stratí (takže žiaden proces nevie, či sa správa stratila, alebo je iba pomalá). Cieľom šéfa je zistiť, medzi ktorými dvomi vrcholmi vedie chybná linka.
Navrhnite algoritmus, pomocou ktorého šéf lokalizuje chybnú linku (t.j. zistí identifikátory vrcholov susediacich s chybnou linkou). Váš algoritmus má v najhoršom prípade použiť $O(n \log n)$ správ (pričom najhorší prípad sa berie cez všetky rozdelenia identifikátorov, všetky pozície šéfa a chybnej linky a všetky časovania správ). Zdôvodnite správnosť a zložitosť (počet správ) Vášho algoritmu.