Distributed-Memory Programming Models III

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 368, Room 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 14/15

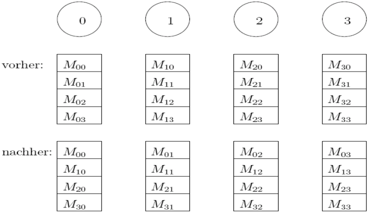
Distributed-Memory Programming Models III

Communication using message passing

- Global communication
- Local exchange
- Synchronisation with time stamps
- Distributed termination
- MPI standard

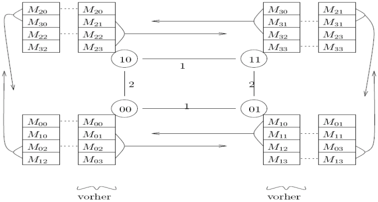
All-to-all with indiv. Messages: Principle

Here has *each* process $P - 1$ messages, one for *each other* process. There are thus $(P - 1)^2$ individual messages to send:



The figure shows already an application: Matrix transposition for column-wise subdivision.

As always, the hypercube (here d=2):



All-to-all with indiv. Messages: General Derivation I

- In general we have the following situation in step $i = 0, \ldots, d-1$:
- Process p communicates with $q = p \oplus 2^i$ and sends to him

$$\begin{array}{rccc} \text{all data of processes } p_{d-1} \ldots p_{i+1} & p_i & x_{i-1} \ldots x_0 \\ \text{for the processes } y_{d-1} \ldots y_{i+1} & \overline{p_i} & p_{i-1} \ldots p_0, \end{array}$$

  where the xe and $y$psilons represent all possible entries.
- $\overline{p_i}$ is negation of a bit.
- There are thus always $P/2$ messages sent in each communication.
- Process p stores at each point in time P data.
- An individual data is underway from process r to process s.
- Each data is identified by $(r, s) \in \{0, \ldots, P-1\} \times \{0, \ldots, P-1\}$.
- We write

$$\mathcal{M}_p^i \subset \{0, \ldots, P-1\} \times \{0, \ldots, P-1\}$$

  for the data, that stores process p *at the beginning of step i*, thus *before* communication.

All-to-all with indiv. Messages: General Derivation II

- *At the start of step 0* process p owns the data

$$\mathcal{M}_p^0 = \{(p_{d-1} \dots p_0, y_{d-1} \dots y_0) \mid y_{d-1}, \dots, y_0 \in \{0,1\}\}$$

- *After communication in step* $i = 0, \dots, d-1$ has p the data $\mathcal{M}_p^{i+1}$, that result from $\mathcal{M}_p^i$ and the following rule ($q = p_{d-1} \dots p_{i+1} \overline{p_i} p_{i-1} \dots p_0$):

$$\begin{aligned}
\mathcal{M}_p^{i+1} = \mathcal{M}_p^i & \\
\underbrace{\setminus}_{\text{sends } p \text{ to } q} & \{(p_{d-1} \dots p_{i+1} p_i x_{i-1} \dots x_0, y_{d-1} \dots y_{i+1} \overline{p_i} p_{i-1} \dots p_0) \mid x_j, y_j \in \{0,1\}\ \forall j\} \\
\underbrace{\cup}_{\text{receives } p \text{ from } q} & \{(p_{d-1} \dots p_{i+1} \overline{p_i} x_{i-1} \dots x_0, y_{d-1} \dots y_{i+1} p_i p_{i-1} \dots p_0) \mid x_j, y_j \in \{0,1\}\ \forall j\}
\end{aligned}$$

All-to-all with indiv. Messages: General Derivation III

- *By induction* applies therefore for *p after* communication in step *i*:

$$\mathcal{M}_p^{i+1} = \{(p_{d-1} \dots p_{i+1} x_i \dots x_0, y_{d-1} \dots y_{i+1} p_i \dots p_0) \mid x_j, y_j \in \{0,1\}\ \forall j\}$$

because of

$$\begin{array}{rlllllll} \mathcal{M}_p^{i+1} = & \{(p_{d-1} \dots p_{i+1} & p_i & x_{i-1} \dots x_0, & y_{d-1} \dots & y_i & p_{i-1} \dots p_0) & \mid \dots\} \\ & \cup\{(p_{d-1} \dots p_{i+1} & \overline{p_i} & x_{i-1} \dots x_0, & y_{d-1} \dots y_{i+1} & p_i & \dots p_0) & \mid \dots\} \\ \setminus & \underbrace{\{\dots\}}_{\text{what i do not need}} & & & & & & \\ = & \{(p_{d-1} \dots p_{i+1} & x_i & x_{i-1} \dots x_0, & y_{d-1} \dots y_{i+1} & p_i & \dots p_0) & \mid \dots\} \end{array}$$

All-to-all with indiv. Messages: Code

void *all_to_all_pers*(**msg** $m[P]$)
{
    int $i, x, y, q, index$;
    msg $sbuf[P/2], rbuf[P/2]$;
    for ($i = 0; i < d; i++$)
    {
        $q = p \oplus 2^i$;   // my partner

        // assemble send buffer:
        for ($y = 0; y < 2^{d-i-1}; y++$)
            for ($x = 0; x < 2^i; x++$)
                $sbuf[\underbrace{y \cdot 2^i + x}_{<P/2\ (!)}] = m[y \cdot 2^{i+1} + (q \& 2^i) + x]$;

        // exchange messages:
        if ($p < q$)
        { **send**(Π_q,$sbuf[0], \ldots, sbuf[P/2-1]$); **recv**($\Pi_q$,$rbuf[0], \ldots, rbuf[P/2-1]$); }
        else
        { **recv**(Π_q,$rbuf[0], \ldots, rbuf[P/2-1]$); **send**($\Pi_q$,$sbuf[0], \ldots, sbuf[P/2-1]$); }

        // disassemble receive buffer:
        for ($y = 0; y < 2^{d-i-1}; y++$)
            for ($x = 0; x < 2^i; x++$)
                $m[\underbrace{y \cdot 2^{i+1} + (q \& 2^i) + x}_{\text{exactly what has been sent is substituted}}] = sbuf[y \cdot 2^i + x]$;

    }
} // end *all_to_all_pers*

All-to-all with indiv. Messages: Code

Complexity analysis:

$$
\begin{aligned}
T_{all-to-all-pers} &= \sum_{i=0}^{\mathrm{ld}\,P-1} \underbrace{2}_{\substack{\text{send and}\\ \text{receive}}} (t_s + t_h + t_w \underbrace{\frac{P}{2}}_{\text{in every step}} n) = \\
&= 2(t_s + t_h)\,\mathrm{ld}\,P + t_w n P\,\mathrm{ld}\,P.
\end{aligned}
$$

MPI: Communicators and Topologies I

In all up to now considered MPI communication functions existed an argument of type `MPI_Comm`. Such a *communicator* contains the following abstractions:

- *Process group:* A communicator can be used to build a subset of all processes. Only these then take part in a global communcation. The pre-defined communicator `MPI_COMM_WORLD` consists of all started processes.
- *Context:* Each communicator defines an individal communication context. Messages can only be received within the same context, in which they have been sent. Such e.g. a library with numerical functions can use its own communicator. Messages of the library are then completely encapsulated from messages in the user program. Therefore messages of the library can not erroneously be received by the user programm and vice versa.
- *Virtual topology:* A communicator represents only a set of processes $\{0, \ldots, P-1\}$. Optionally this set can be enhanced by an additional structure, e.g. a multi-dimensional field or a general graph.

MPI: Communicators and Topologies II

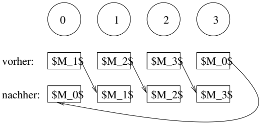
- *Additional attributes:* An application (e.g. a library) can associate with the communicator arbitrary static data. The communicator serves as medium to retain data from a call of the library to the next.
- This is an *intra-communicator*, that only enables communication *within* a process group.
- Furthermore there are *inter-communicators*, that support communication of *distinct* process groups. These are not considered further at the moment!
- As a possibility to create a new (intra-) communicator we have a look at the function

```
int MPI_Comm_split(MPI_Comm comm, int color,
                   int key, MPI_Comm *newcomm);
```

- `MPI_Comm_split` is a collective operation, that has to be called by *all* processes of the communicator `comm`. All processes with equal value for the argument `color` create each a new communicator. The sequence (rank) within the new communicator is managed by the argument `key`.

Local Exchange: Shifting in the Ring I

- Consider the following problem: Each process $p \in \{0, \dots, P-1\}$ has to send data to $(p+1)\%P$:



- Naive realisation with synchronous communication results in deadlock:

  ...
  send$(\Pi_{(p+1)\%P}, msg)$;
  recv$(\Pi_{(p+P-1)\%P}, msg)$;
  ...

- Avoiding the deadlock (e. g. exchanging of **send**/**recv** in one process) does not deliver maximal possible parallelism.
- Asynchronous communication is often not preferential because of efficiency reasons.

Local Exchange: Shifting in the Ring II

- Solution: *Coloring*. Be $G = (V, E)$ a graph with

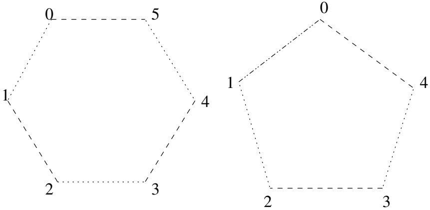
$$\begin{aligned} V &= \{0, \ldots, P-1\} \\ E &= \{e = (p, q) | \text{process } p \text{ has to communicate with process } q\} \end{aligned}$$

- There are the *edges* to color in such a way, that each node has only connections to edges with different colors. The assignment of colors is described by the mapping

$$c \colon E \rightarrow \{0, \ldots, C-1\}$$

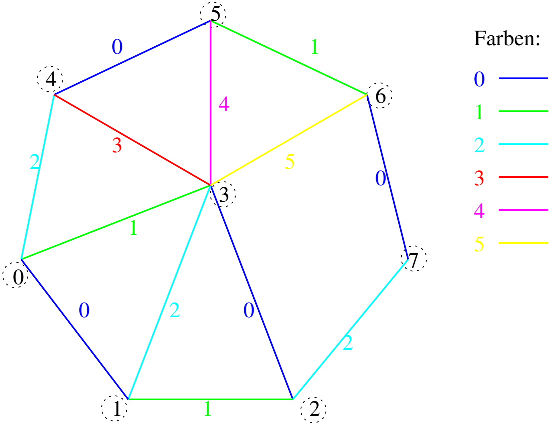
, where C is the count of necessary colors.

- Shifting in the *ring* needs two colors for P being even and three color for P being odd:



Local Exchange: General Graph I

Establish the communication relations a general graph, then the coloring is determined by an algorithm.



Here a more or less sequential heuristic:

Local Exchange: General Graph II

Program (Distributed Coloring)

```
parallel coloring
{
    const int P;
    process Π[int p ∈ {0, . . . , P − 1}]{
        int nbs;                                          // number of neighbors
        int nb[nbs];                                      // nb[i] < nb[i + 1] !
        int color[nbs];                                   // the result
        int index[MAXCOLORS];                             // free color management
        int i, c, d;
        for (i = 0; i < nbs; i + +) index[i]=-1;
        for (i = 0; i < nbs; i + +)                       // find color for connection to nb[i]
            c = 0;                                        // start with color 0
            while(1) {
                c=min{k ≥ c|index[k] < 0};                // next free color ≥ c
                if (p < nb[i]) { send(Π_nb[i],c); recv(Π_nb[i],d); }
                else { recv(Π_nb[i],c); send(Π_nb[i],d); }
                if (c == d){                              // the two have an agreement
                    index[c] = i; color[i] = c; break;
                } else c = max(c, d);
            }
        }
    }
}
```

Lamport Time Stamps I

- Goal: Ordering of events in distributed systems.
- Events: Execution of (marked) instructions.
- The ideal situation would be a global clock, but this is not available in distributed systems, since the sending of messages always is in conjunction with delays.
- *Logical clock*: Time points, that have been assigned to events, shall not be in obvious contradiction to a global clock.

Π_1:	Π_2:	Π_3:
$a = 5$;		
		$\vdots$
$\ldots$;	$\ldots$;	
$b = 3$;	$c = 4$;	
send(Π_2, a);	$\ldots$;	
	recv(Π_1, b);	$e = 7$;
	$d = 8$;	**send**(Π_2, e);
$\vdots$		
	recv(Π_3, e);	
	$f = bde$;	
	$\vdots$	
	send(Π_1, f);	
recv(Π_2, f);		

Lamport Time Stamps II

- Be a an event in process p and $C_p(a)$ the time stamp, p the associated process, e. g. $C_2(f = bde)$, then the time stamps should have the following properties:
  1. Be a and b two events in the same process p, where a occurs before b, then shall be $C_p(a) < C_p(b)$.
  2. Process p sends a message to q, then shall be $C_p(\textbf{send}) < C_q(\textbf{receive})$.
  3. For two arbitrary events a and b in arbitrary processes p resp. q be $C_p(a) \neq C_q(b)$.
- 1 and 2 represent the causality of events: If in a parallel program can surely be said, that a in p occurs *before* b in q, then applies $C_p(a) < C_q(b)$ too.
- Only with the properties 1 and 2 $a \leq_C b :\iff C_p(a) < C_q(b)$ would be a half ordering on the set of all events.
- Property 3 reusults then in a total ordering.

Lamport Time Stamps: Implementation

Program (Lamport time stamps)

parallel *Lamport time stamps*
{

const int P; // *whats this?*
int $d = \min\{i|2^i \geq P\}$; // *how many bit positions has P.*

process Π[**int** $p \in \{0, \ldots, P-1\}$]
{

int *C=0;* // *the clock*
int t, s, r; // *only for the example*
int *Lclock(***int** *c)* // *output of a new time stamp*
{

$C = \max(C, c/2^d)$; // *rule 2*
C++; // *rule 1*
return $C \cdot 2^d + p$; // *rule 3*
// *the last d bits contain p*

}

// **application:**
// *A local event happens*
t=Lclock(0);

s=Lclock(0); // *send*
send(Π_q,*message,s);* // *the time stamp is sent together!*

recv(Π_q,*message,r);* // *receivers also the time stamp of the reveiver!*
r=Lclock(r); // *thus applies* $C_p(r) > C_q(s)$*!*

}

}

Lamport Time Stamps: Implementation

- Management of the time stamps is in response of the user. Ordinarily one necessitates time stamps only for very specific events (see below).
- Overflow of the counter has not been considered.

Distributed Mutual Exclusion with Time Stamps I

- Problem: From a set of distributed processes exactly one shall do something (e. g. control a device, serve as server, . . .). Like in the case of a critical section the processes have to decide which is next.
- A possibility would be, that just one process decides who is next.
- We now present a distributed solution:
  - Does a process want to enter it sends a message to all others.
  - As soon as it has gotten an answer from all (there is no no!) it can enter.
  - A process confirms only, if it doesn't want to enter or if the time stamp of an entry query is larger than that of the others.
- Solution works with a local monitor process.

Distributed Mutual Exclusion with Time Stamps II

Program (Distributed mutual exclusion with Lamport time stamps)

```
parallel DME-timestamp // Distributed Mutual Exclusion
{
    int P; const int REQUEST=1, REPLY=2;                       // messages

    process Π[int p ∈ {0, ..., P − 1}]
    {
        int C=0, mytime;                                       // clock
        int is_requesting=0, reply_pending, reply_deferred[P]={0,...,0};  // deferred processes

        process M[int p‘ = p]                                  // the monitor
        {
            int msg, time;
            while(1) {
                recv_any(π,q,msg,time);                        // receive from q's monitor with time
                if (msg==REQUEST)                              // stamp of sender q wants to enter
                {
                    [Lclock(time);]                            // increase own clock for later request.
                                                               // critical section, since Π also increases.
                    if(is_requesting ∧ mytime < time)
                        reply_deferred[q]=1;                   // q shall wait
                    else
                        asend(M_q,p,REPLY,0);                  // q may enter
                }
                else reply_pending−−;                          // it has been a REPLY
            }
        }
        ...
}
```

Distributed Mutual Exclusion with Time Stamps II

Program (Distributed mutual exclusion with Lamport time stamps cont.)

```
parallel DME-timestamp // Distributed Mutual Exclusion cont.
{
    ...
        void enter_cs()                                  // to enter the critical section
        {
            int i;
            [ mytime=Lclock(0); is_requesting=1; ]
                                                         // critical section
            reply_pending=P − 1;                         // so many answers do I expect
            for (i=0; i < P; i++)
                if (i ≠ p) send(M_i,p,REQUEST,mytime);
            while (reply_pending> 0);                    // busy wait
        }
        void leave_cs()
        {
            int i;
            is_requesting=0;
            for (i=0; i < P; i++)                        // inform waiting processes
            if (reply_deferred[i]
            {
                send(M_i,p,REPLY,0);
                reply_deferred[i]=0;
            }
        }
        enter_cs(); /* critical section */ leave_cs();
    } // end process
}
```

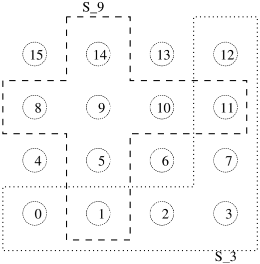
Distributed Mutual Exclusion with „Voting“ I

- The algorithm above needs $2P$ messages per process to enter the critical section. With voting we will only need $O(\sqrt{P})$.
- Especially a process doesn't need to ask *all* others before it may enter.
- Idea:
  - The related processes acquire for entry into the critical section. These are called *candidates*
  - All (or some, see below) vote who may enter. These are called *voters*. Each can be candidate or voter.
  - Instead of absolute majority we require only relative majority: A process may enter as soon as it knowns, that no other can have more votes than itself.
- Each process is assigned a voting district $S_p \subseteq \{0, \ldots, P-1\}$. It applies the coverage property:

$$S_p \cap S_q \neq \emptyset \quad \forall p, q \in \{0, \ldots, P-1\}.$$

Distributed Mutual Exclusion with „Voting“ II

- The voting districts for 16 processes look like this:



- A process p can enter, if it gets all votes of its voting district. Since no other process q can enter: According to prerequisite there exists $r \in S_p \cap S_q$ and r has decided to vote for p, thus q cannot have gotten all votes.
- Danger of deadlock: Is $|S_p \cap S_q| > 1$ thus one can decide for p and another for q, both never may enter. Solution of deadlocks with Lamport time stamps.

Optimality of Voting Districts I

- Question: How small can the voting districts be?
- Again: Each p has its voting district $S_p \subseteq \{0, \ldots, P-1\}$ and we require $S_p \cap S_q \neq \emptyset$.
- But this would allow e. g. $S_p = \{0\}$ for all p, what we do not want.
- Define D_p as the set of processes for which p has to vote:

$$D_p = \{q | p \in S_q\}\}$$

- We additionally require that for all p:

$$|S_p| = K, \qquad |D_p| = D.$$

  This excludes the trivial solution from above.
- With this assumption even holds $D = K$, since define the set of all pairs (p, q) with p chooses for q, d.h. :

$$A = \{(p, q) | 0 \leq p < P \wedge q \in D_p\}.$$

Optimality of Voting Districts II

- On the other side define the set of all pairs (p, q) where p has to be voted by q:
$$B = \{(p, q) | 0 \leq p < P \wedge q \in S_p\}.$$
Because of $q \in S_p \Leftrightarrow p \in D_q$ holds $(p, q) \in B \Leftrightarrow (q, p) \in A$ thus $|A| = |B|$. For the sizes applies $|A| = P \cdot D$ and $|B| = P \cdot K$ thus $D = K$.
- For fixed $K (= D)$ we maximize now the number of voting districts (processors) P:
  - Vote an arbitrary voting district S_p. This has K members.
  - Vote an arbitrary $r \in S_p$. This r is member in D voting districts (set D_r) where one is S_p (obviously is $p \in D_r$. Therefore we count $K(D-1)+1$ voting districts.
  - More cannot exist, since for arbitrary q applies: There is a r with $r \in S_p \cap S_q$ and thus $q \in D_r$. We have thus all gotten.

  Thus it holds that
$$\boxed{P \leq K(K-1)+1}$$
  or
$$K \geq \frac{1}{2} + \sqrt{P - \frac{3}{4}}.$$

Voting: Implementation I

Program (Distributed Mutual Exclusion with Voting)

```
parallel DME-Voting
{
    const int P = 7.962;
    const int REQUEST=1, YES=2, INQUIRE=3, RELINQUISH=4, RELEASE=5;
            // „inquire“ = „sich erkundigen“; „relinquish“ = „aufgeben“, „verzichten“
    process Π[int p ∈ {0, . . . , P − 1}]
    {
        int C=0, mytime;

        void enter_cs()                                   // wants to enter critical section
        {
            int i, msg, time, yes_votes=0;
            [ mytime=Lclock(0); ]                         // time of my request
            for (i ∈ S_p) asend(V_i,p, REQUEST,mytime);
                                                          // send request to voting districts
            while (yes_votes < |S_p|) {
                recv_any(π,q,msg,time);                   // receive from q
                if (msg==YES) yes_votes++;                // q choose
                if (msg==INQUIRE)                         // q wants vote back
                    if (mytime==time)                     // now current request
                    {                                     // there may be old on the way
                        asend(V_q,p,RELINQUISH,0);
                                                          // passes back
                        yes_votes−−;
                    }
            }
        }// end enter_cs
        . . .
```

Voting: Implementation II

Program (Distributed Mutual Exclusion with Voting cont. 1)

parallel *DME-Voting cont. 1*
{
    . . .
        void *leave_cs()*
        {
            int *i;*
            for $(i \in S_p)$ **asend**$(V_i,p,RELEASE,0)$;
            // There could be still not processed INQUIRE messages for this
            // critical section exist, that are now obsolete.
            // These are then ignored in enter_cs.
        }

        // Example:
        enter_cs();
        . . . ; // critical section
        leave_cs();

    }

Voting: Implementation III

Program (Distributed Mutual Exclusion with Voting cont. 2)

```
parallel DME-Voting cont. 2
{
    process V[int p′ = p]                                  // the voter for Π_p
    {
        int q, candidate, msg, time, have_voted=0, candidate_time, have_inquired=0;
        while(1)                                           // runs forever
        {
            recv_any(π,q,msg,time);                        // receive it with sender
            if (msg==REQUEST)                              // request of a candidate
            {
                [ Lclock(time); ]                          // increase clock for later requests
                if (¬have_voted) {                         // I have still to vote
                    asend(Π_q,p,YES,0);                    // back to candidate process
                    candidate_time=time;                   // remember whom I gave
                    candidate=q;                           // my vote.
                    have_voted=1;                          // yes, I have already voted
                }
                else{                                      // I have already voted
                    store (q, time) in list;
                    if (time < candidate_time ∧ ¬have_inquired)
                    {                                      // get back vote from candidate!
                        asend(Π_candidate,p,INQUIRE,candidate_time);
                                // with the candidate_time it recognizes which request
                                // it is: it could have happened, that it already entered.
                        have_inquired=1;
                    }
                }
            } . . .
```

Voting: Implementation IV

Program (Distributed Mutual Exclusion with Voting cont. 3)

```
parallel DME-Voting cont. 3
{
    . . .                                          // q is the candidate, that has
            else if (msg==RELINQUISH)              // passed back it vote.
            {
                store (candidate, candidate_time) in list;
                take away and delete
                    the entry with the smallest time from the list: (q, time)
                                                   // There could exist others
                asend(Π_q,p,YES,0);                // vote for q
                candidate_time=time;               // new candidate
                candidate=q;
                have_inquired=0;                   // no INQUIRE on the way
            }
            else if (msg==RELEASE)                 // q leaves the critical section
            {
                if (list is not empty)
                {                                  // vote new
                    take away and delete
                        the entry with the smallest time from list: (q, time)
                    asend(Π_q,p,YES,0);
                    candidate_time=time;           // new candidate
                    candidate=q;
                    have_inquired=0;               // forget all INQUIREs because obsolete
                }
                else
                    have_voted=0;                  // noone need to be voted
            }
```

Distributed Termination I

There are processes $\Pi_0, \ldots, \Pi_{P-1}$ defined, that communicate over a communication graph .

$$G = (V, E)$$
$$V = \{\Pi_0, \ldots, \Pi_{P-1}\}$$
$$E \subseteq V \times V$$

With that process Π_i sends messages to the processes

$$N_i = \{j \in \mathbb{N} \mid (\Pi_i, \Pi_j) \in E\}$$

```
process Πi [ int i ∈ {0, ..., P − 1}]
{
    while (1)
    {
        recv_any(who,msg),                    // Πi is idle
        compute(msg);
        for ( p ∈ N_msg ⊆ N_i )
        {
            msg_p = ...;
            asend(Π_p, msg_p);                // ignore buffer problems

        }
    }
}
```

Distributed Termination II

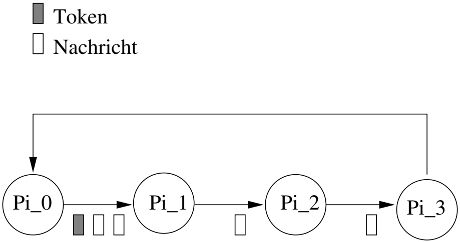
The termination problem consists of finalizing a program only if applies:

1. All wait for a message (are idle)
2. No messages are underway

Thereby the following assumption are applied regarding the messages:

1. Ignore problems with buffer overflow
2. The messages between two processes are processed in the sequence of sending

1. variant: termination in the ring



Distributed Termination III

Each process has one of two possible states: red (active) or blue (idle). For termination recognition a mark is sent around in the ring.
Suppose process Π_0 starts the termination process, thus turns first into blue.
Also suppose,

1. Π_0 is in state blue
2. mark has arrived at Π_i and Π_i has been recolored into blue

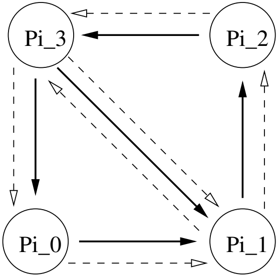
Then we can assume, that the processes $\Pi_0, \ldots, \Pi_i$ are idle and the channels $(\Pi_0, \Pi_1), \ldots, (\Pi_{i-1}, \Pi_i)$ are empty.
Is the mark again at Π_0 and is it still blue (what it can decide), then obvious applies:

1. $\Pi_0, \ldots, \Pi_{P-1}$ are idle
2. All channels are empty

Then the termination is recognized.

Distributed Termination IV

2. variant: general graph with directed edges



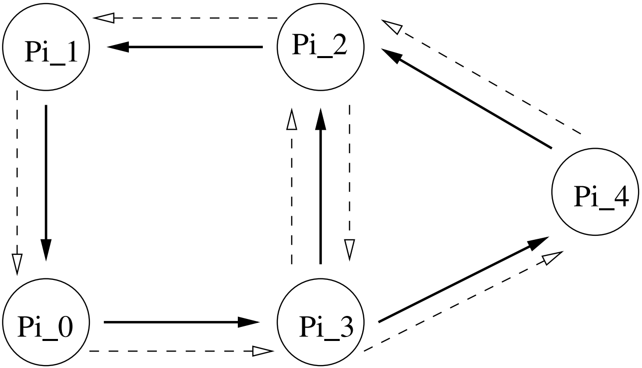
Idea: Over the graph a ring is formed, that includes all nodes, where a node also can be visited more than once.

Algorithm: Choose a path $\pi = (\Pi_{i_1}, \Pi_{i_2}, \dots, \Pi_{i_n})$ of length n of processes such that applies:

1. Each edge $(\Pi_p, \Pi_q) \in E$ exists at least in the path once
2. A sequence (Π_p, Π_q, Π_r) exists at most once in the path. Does one reach q from p, then is goes always further to r. r therefore depends on Π_p und Π_q ab: $r = r(\Pi_p, \Pi_q)$

Distributed Termination V

Example with $\pi = (\Pi_0, \Pi_3, \Pi_4, \Pi_2, \Pi_3, \Pi_2, \Pi_1, \Pi_0)$.



Distributed Termination VI

process Π [**int** $i \in \{0, \ldots, P-1\}$]
{
    int *color* = red , *token*;
    if ($\Pi_i == \Pi_{i_1}$)
    {   // initialisation of the token
        color = blue;
        token = 0 ,
        asend(Π_{i_2}, TOKEN, *token*)
    }
    while(1)
    {
        recv_any(*who*,*tag*,*msg*);
        if (*tag* != TOKEN) { *color* = red; calculate further }
        else   // msg = Token
        {
            if (*msg* == n) { break; „yeah, ready! “}
            if (*color* == red)
            {
                color = blue ;
                token = 0 ;
                rcvd = *who* ;
            }
            else

                if (*who* == *rcvd*) *token*++ ; // a full cycle

            asend($\Pi_{r(who,\Pi_i)}$, TOKEN , *token*);
        }
    }
}

Distributed Philosophers I

We consider the philosophers problem again, but now with message passing.

- Let a mark circle in the ring. Only how has the mark, may eventually eat.
- State transitions are told to the neighbors, **before** the mark is passed further.
- Each philosopher P_i is assigned a server W_i, that performs the state manipulation.
- We use only synchronous communication

process P_i [**int** $i \in \{0, \dots, P-1\}$]
{
    while (1) {
        think;
        send(W_i, HUNGRY);
        recv(W_i, *msg*);
        eat;
        send(W_i, THINK);

    }
}

Distributed Philosophers II

process W_i [**int** $i \in \{0, \dots, P-1\}$]
{
    int L = $(i+1)\%P$;
    int R = $(i+p-1)\%P$;
    int *state* = *stateL* = *stateR* = THINK ;
    int *stateTemp*;
    if (i == 0) **send**(W_L , TOKEN);
    while (1) {
        recv_any(*who*, *tag*);
        if (*who* == P_i) *stateTemp* = *tag* ; // my philosopher
        if (*who* == W_L & & *tag* $\neq$ TOKEN) *stateL* = *tag* ; // state change
        if (*who* == W_R & & *tag* $\neq$ TOKEN) *stateR* = *tag* ; // in neighbor
        if (*tag* == TOKEN){
            if (*state* $\neq$ EAT & & *stateTemp* == HUNGRY
                & & *stateL* == THINK & & *stateR* == THINK){
                    state = EAT;
                    send(W_l , EAT);
                    send(W_R , EAT);
                    send(P_i , EAT);

            }
            if (*state* == EAT & & *stateTemp* == THINK){
                state = THINK;
                send(W_L , THINK);
                send(W_R , THINK);
            }
            send(W_L , TOKEN);
        }
    }
}