Paralleles Höchstleistungsrechnen

—

Distributed-Memory Programmiermodelle III

Stefan Lang

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
INF 368, Raum 425
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

25. November 2009

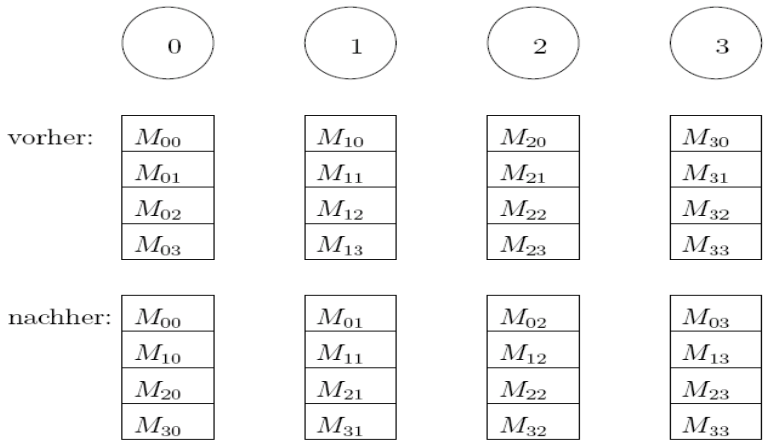
Distributed-Memory Programmiermodelle III

Kommunikation über Nachrichtenaustausch

- Globale Kommunikation
- Lokaler Austausch
- Synchronisation mit Zeitmarken
- Verteiltes Beenden
- MPI Standard

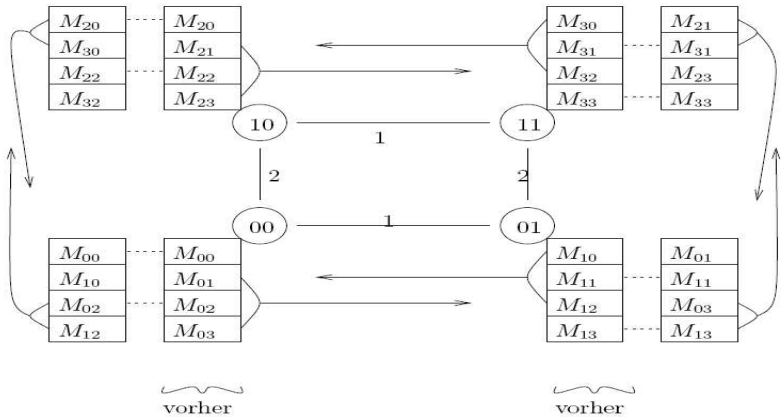
Alle-an-alle mit indiv. Nachrichten: Prinzip

Hier hat *jeder* Prozess $P-1$ Nachrichten, je eine für *jeden anderen* Prozess. Es sind also $(P-1)^2$ individuelle Nachrichten zu verschicken:



Das Bild zeigt auch schon eine Anwendung: Matrixtransposition bei spaltenweiser Aufteilung.

Wie immer, der Hypercube (hier d=2):



Alle-an-alle mit indiv. Nachrichten: Allgem. Herleitung

- Allgemein haben wir folgende Situation im Schritt $i = 0, \ldots, d-1$:
- Prozess p kommuniziert mit $q = p \oplus 2^i$ und sendet ihm

$$\begin{array}{rrrr} \text{alle Daten der Prozesse } & p_{d-1} \ldots p_{i+1} & p_i & x_{i-1} \ldots x_0 \\ \text{für die Prozesse } & y_{d-1} \ldots y_{i+1} & \overline{p_i} & p_{i-1} \ldots p_0, \end{array}$$

  wobei die xe und $y$psilons für alle möglichen Einträge stehen.
- $\overline{p_i}$ ist Negation eines Bits.
- Es werden also in jedem Kommunikationsschritt immer $P/2$ Nachrichten gesendet.
- Prozess p besitzt zu jedem Zeitpunkt P Daten.
- Ein individuelles Datum ist von Prozess r zu Prozess s unterwegs.
- Jedes Datum ist identifiziert durch $(r, s) \in \{0, \ldots, P-1\} \times \{0, \ldots, P-1\}$.
- Wir schreiben

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

  für die Daten, die Prozess p *zu Beginn von Schritt* i besitzt, d.h. *vor* der Kommunikation.

Alle-an-alle mit indiv. Nachrichten: Allgem. Herleitung

- *Zu Beginn von Schritt 0* besitzt Prozess p die Daten

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

- *Nach Kommunikation im Schritt* $i = 0, \dots, d-1$ hat p die Daten $\mathcal{M}_p^{i+1}$, die sich aus $\mathcal{M}_p^i$ und folgender Regel ergeben ($q = p_{d-1} \dots p_{i+1} \overline{p_i} p_{i-1} \dots p_0$):

$$\mathcal{M}_p^{i+1} = \mathcal{M}_p^i$$
$$\underbrace{\setminus}_{\text{schickt } p \text{ an } q} \{(p_{d-1} \dots p_{i+1} p_i x_{i-1} \dots x_0, y_{d-1} \dots y_{i+1} \overline{p_i} p_{i-1} \dots p_0) \mid x_j, y_j \in \{0,1\}\ \forall j\}$$
$$\underbrace{\cup}_{\text{kriegt } p \text{ von } q} \{(p_{d-1} \dots p_{i+1} \overline{p_i} x_{i-1} \dots x_0, y_{d-1} \dots y_{i+1} p_i p_{i-1} \dots p_0) \mid x_j, y_j \in \{0,1\}\ \forall j\}$$

Alle-an-alle mit indiv. Nachrichten: Allgem. Herleitung

- *Per Induktion* gilt damit für p *nach* Kommunikation in Schritt i:

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

denn

$$\begin{aligned}
\mathcal{M}_p^{i+1} = &\{(p_{d-1} \dots p_{i+1} \quad p_i \quad x_{i-1} \dots x_0, \quad y_{d-1} \dots \quad y_i \quad p_{i-1} \dots p_0) \quad \mid \dots\} \\
&\cup \{(p_{d-1} \dots p_{i+1} \quad \overline{p_i} \quad x_{i-1} \dots x_0, \quad y_{d-1} \dots y_{i+1} \quad p_i \quad \dots p_0) \quad \mid \dots\} \\
&\setminus \underbrace{\{\dots\}}_{\text{was ich nicht brauche}} \\
= &\{(p_{d-1} \dots p_{i+1} \quad x_i \quad x_{i-1} \dots x_0, \quad y_{d-1} \dots y_{i+1} \quad p_i \quad \dots p_0) \quad \mid \dots\}
\end{aligned}$$

Alle-an-alle mit persönlichen Nachrichten: Code

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

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

// Nachrichten austauschen:
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**($\Pi_q$,$sbuf[0], \ldots, sbuf[P/2-1]$); }

// Empfangpuffer disassemblieren:
for $(y = 0; y < 2^{d-i-1}; y++)$
for $(x = 0; x < 2^i; x++)$
$m[\underbrace{y \cdot 2^{i+1} + (q \& 2^i) + x}_{\text{genau das, was gesendet wurde, wird ersetzt}}] = sbuf[y \cdot 2^i + x]$;

}
} // ende *all_to_all_pers*

Alle-an-alle mit persönlichen Nachrichten: Code

Komplexitätsanalyse:

$$T_{all-to-all-pers} = \sum_{i=0}^{\mathrm{ld}\,P-1} \underbrace{2}_{\substack{\text{send u.}\\ \text{receive}}} (t_s + t_h + t_w \underbrace{\frac{P}{2}}_{\substack{\text{in jedem}\\ \text{Schritt}}} n) =$$

$$= 2(t_s + t_h)\,\mathrm{ld}\,P + t_w n P\,\mathrm{ld}\,P.$$

MPI: Communicators und Topologien I

In allen bisher betrachteten MPI Kommunikationsfunktionen trat ein Argument vom Typ `MPI_Comm` auf. Ein solcher *Communicator* beinhaltet die folgenden Abstraktionen:

- *Prozessgruppe:* Ein Communicator kann benutzt werden, um eine Teilmenge aller Prozesse zu bilden. Nur diese nehmen dann etwa an einer globalen Kommunikation teil. Der vordefinierte Communicator `MPI_COMM_WORLD` besteht aus allen gestarteten Prozessen.
- *Kontext:* Jeder Communicator definiert einen eigenen Kommunikationskontext. Nachrichten können nur innerhalb des selben Kontextes empfangen werden, in dem sie abgeschickt wurden. So kann etwa eine Bibliothek numerischer Funktionen einen eigenen Communicator verwenden. Nachrichten der Bibliothek sind dann vollkommen von Nachrichten im Benutzerprogramm abgeschottet, und Nachrichten der Bibliothek können nicht fälschlicherweise vom Benutzerprogramm empfangen werden und umgekehrt.
- *Virtuelle Topologien:* Ein Communicator steht nur für eine Menge von Prozessen $\{0, \ldots, P-1\}$. Optional kann man diese Menge mit einer zusätzlichen Struktur, etwa einem mehrdimensionalen Feld oder einem allgemeinen Graphen, versehen.

MPI: Communicators und Topologien II

- *Zusätzliche Attribute:* Eine Anwendung (z.B. eine Bibliothek) kann mit einem Communicator beliebige statische Daten assoziieren. Der Communicator dient dann als Vehikel, um diese Daten von einem Aufruf der Bibliothek zum nächsten hinüberzuretten.
- Dies ist ein *Intra-Communicator*, der nur Kommunikation *innerhalb* einer Prozessgruppe erlaubt.
- Darüberhinaus gibt es *Inter-Communicators*, die Kommunikation zwischen *verschiedenen* Prozessgruppen erlauben. Diese betrachten wir nicht weiter!
- Als eine Möglichkeit zur Bildung neuer (Intra-) Communicators stellen wir die Funktion

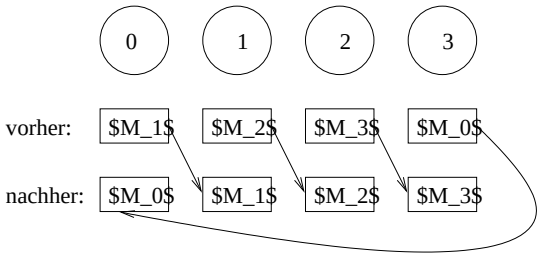
```
int MPI_Comm_split(MPI_Comm comm, int color,
                   int key, MPI_Comm *newcomm);
```

  vor.

- `MPI_Comm_split` ist eine kollektive Operation, die von *allen* Prozessen des Communicators `comm` gerufen werden muss. Alle Prozesse mit gleichem Wert für das Argument `color` bilden jeweils einen neuen Communicator. Die Reihenfolge (rank) innerhalb des neuen Communicator wird durch das Argument `key` geregelt.

Lokaler Austausch: Schieben im Ring I

- Betrachte folgendes Problem: Jeder Prozess $p \in \{0, \ldots, P-1\}$ muss ein Datum an $(p+1)\%P$ schicken:



- Naives Vorgehen mit synchroner Kommunikation liefert Deadlock:

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

- Aufbrechen des Deadlock (z. B. Vertauchen von **send**/**recv** in einem Prozess) liefert nicht die maximal mögliche Parallelität.
- Asynchrone Kommunikation möchte man aus Effizienzgründen oft nicht verwenden.

Lokaler Austausch: Schieben im Ring II

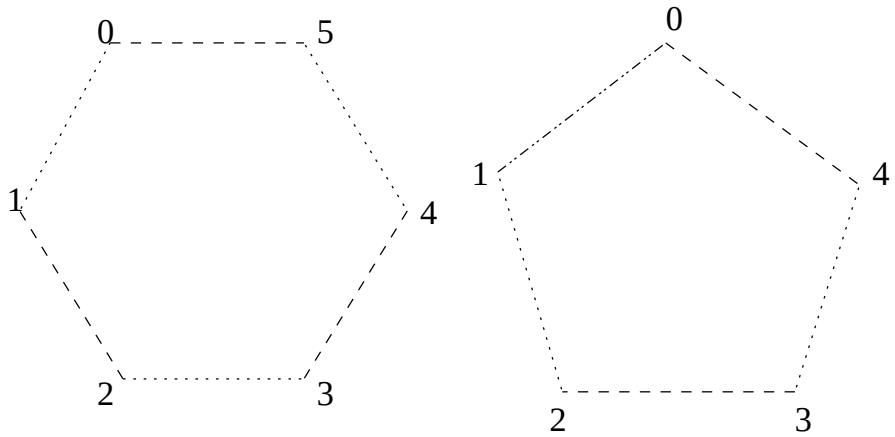
- Lösung: *Färben*. Sei $G = (V, E)$ ein Graph mit

$$\begin{aligned} V &= \{0, \ldots, P-1\} \\ E &= \{e = (p, q) | \text{Prozess } p \text{ muß mit Prozess } q \text{ kommunizieren}\} \end{aligned}$$

- Es sind die *Kanten* so einzufärben, dass an einem Knoten nur Kanten unterschiedlicher Farben anliegen. Die Zuordnung der Farben sei durch die Abbildung

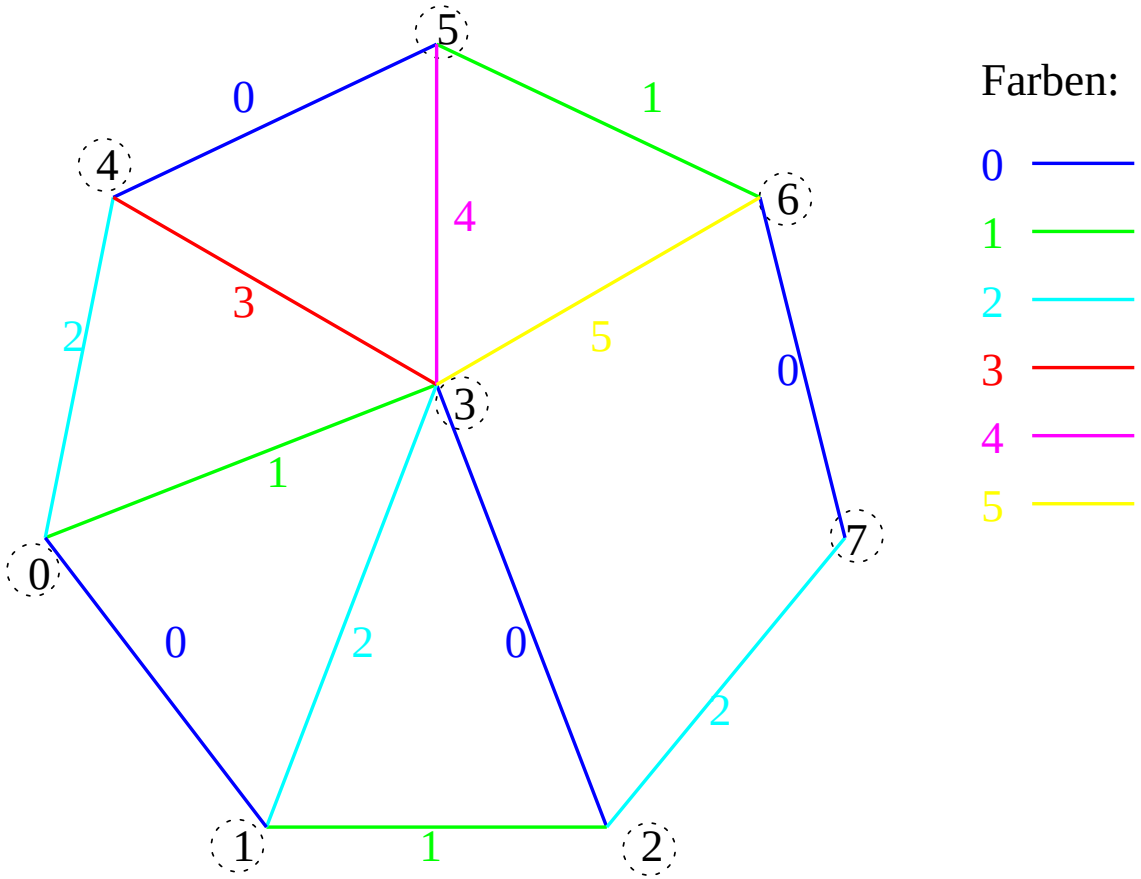
$$c : E \rightarrow \{0, \ldots, C-1\}$$

  gegeben, wobei C die Zahl der benötigten Farben ist.
- Schieben im *Ring* braucht zwei Farben für gerades P und drei Farben für ungerades P:



Lokaler Austausch: allgemeiner Graph I

Ergeben die Kommunikationsbeziehungen einen allgemeinen Graph, so ist die Färbung algorithmisch zu bestimmen.

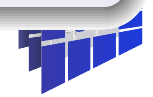


Hier eine mehr oder weniger sequentielle Heuristik:

Lokaler Austausch: allgemeiner Graph II

Programm (Verteiltes Färben)

```
parallel coloring
{
    const int P;
    process Π[int p ∈ {0, ..., P − 1}]{
        int nbs;                                          // Anzahl Nachbarn
        int nb[nbs];                                      // nb[i] < nb[i + 1] !
        int color[nbs];                                   // das Ergebnis
        int index[MAXCOLORS];                             // Verwaltung der freien Farben
        int i, c, d;
        for (i = 0; i < nbs; i++) index[i]=-1;
        for (i = 0; i < nbs; i++)                         // finde Farbe zur Verbindung zu nb[i]
            c = 0;                                        // beginne mit Farbe 0
            while(1) {
                c=min{k ≥ c|index[k] < 0};                          // nächste freie Farbe ≥ 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){                              // die beiden haben sich geeinigt
                    index[c] = i; color[i] = c; break;
                } else c = max(c, d);
            }
        }
    }
}
```



Lamport Zeitmarken I

- Ziel: Ordnen von Ereignissen in verteilten Systemen.
- Ereignisse: Ausführen von (gekennzeichneten) Anweisungen.
- Ideal wäre eine globale Uhr, die gibt es aber in verteilten Systemen nicht, da das Senden von Nachrichten immer mit Verzögerungen verbunden ist.
- *Logische Uhr*: Den Ereignissen zugeordnete Zeitpunkte sollen nicht in offensichtlichem Widerspruch zu einer globalen Uhr stehen.

Π_1:
$a = 5;$

$\ldots;$
$b = 3;$
send$(\Pi_2, a);$

$\vdots$

recv$(\Pi_2, f);$

Π_2:

$\ldots;$
$c = 4;$
$\ldots;$
recv$(\Pi_1, b);$
$d = 8;$

recv$(\Pi_3, e);$
$f = bde;$

$\vdots$
send$(\Pi_1, f);$

Π_3:

$\vdots$

$e = 7;$
send$(\Pi_2, e);$

Lamport Zeitmarken II

- Sei a ein Ereigniss in Prozess p und $C_p(a)$ die Zeitmarke, die Prozess p damit assoziiert, z. B. $C_2(f = bde)$. Dann sollen diese Zeitmarken folgende Eigenschaften haben:
  1. Seien a und b zwei Ereignisse im selben Prozess p, wobei a vor b stattfinde, so soll $C_p(a) < C_p(b)$ gelten.
  2. Es sende p eine Nachricht an q, so soll $C_p(\textbf{send}) < C_q(\textbf{receive})$ sein.
  3. Für zwei beliebige Ereignisse a und b in beliebigen Prozessen p bzw. q gelte $C_p(a) \neq C_q(b)$.
- 1 und 2 spiegeln die Kausalität von Ereignissen wieder: Wenn in einem parallelen Programm sicher gesagt werden kann, dass a in p *vor* b in q stattfindet, dann gilt auch $C_p(a) < C_q(b)$.
- Nur mit den Eigenschaften 1 und 2 wäre $a \leq_C b :\iff C_p(a) < C_q(b)$ eine Halbordnung auf der Menge aller Ereignisse.
- Die Bedingung 3 macht daraus dann eine totale Ordnung.

Lamport Zeitmarken: Implementierung

Programm (Lamport-Zeitmarken)

```
parallel Lamport-timestamps
{
    const int P;                                   // was wohl?
    int d = min{i|2^i ≥ P};                        // wieviele Bitstellen hat P.

    process Π[int p ∈ {0,...,P − 1}]
    {
        int C=0;                                   // die Uhr
        int t, s, r;                               // nur für das Beispiel
        int Lclock(int c)                          // Ausgabe einer neuen Zeitmarke
        {
            C=max(C, c/2^d);                       // Regel 2
            C++;                                   // Regel 1
            return C · 2^d + p;                    // Regel 3
                        // die letzten d Bits enthalten p
        }

        // Verwendung:
        // Ein lokales Ereignis passiert
        t=Lclock(0);

        s=Lclock(0);                               // send
        send(Π_q,message,s);                       // Die Zeitmarke wird mit verschickt!

        recv(Π_q,message,r);                       // empfängt auch die Zeitmarke des Senders!
        r=Lclock(r);                               // so gilt C_p(r) > C_q(s)!
    }
}
```

Lamport Zeitmarken: Implementierung

- Verwaltung der Zeitmarken obliegt dem Benutzer. Üblicherweise benötigt man Zeitmarken nur für ganz bestimmte Ereignisse (siehe unten).
- Überlauf des Zählers wurde nicht behandelt.

Verteilter wechselseitiger Ausschluss mit Zeitmarken I

- Problem: Von einer Menge verteilter Prozesse soll genau einer etwas tun (z. B. ein Gerät steuern, als Server dienen, . . .). Wie bei einem kritischen Abschnitt müssen sich die Prozesse einigen wer drankommt.
- Eine Möglichkeit wäre, daß genau ein Prozess entscheidet wer drankommt.
- Wir stellen jetzt eine verteilte Lösung vor:
  - Will ein Prozess eintreten schickt er eine Nachricht an alle anderen.
  - Sobald er Antwort von allen bekommen hat (es gibt kein Nein!) kann er eintreten.
  - Ein Prozess bestätigt nur, wenn er nicht rein will oder wenn die Zeitmarke seines Eintrittswunsches größer ist als die des anderen.
- Lösung arbeitet mit lokalem Monitorprozess.

Verteilter wechselseitiger Ausschluss mit Zeitmarken II

Programm (Verteilter wechselseitiger Ausschluss mit Lamport-Zeitmarken)

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

    process Π[int p ∈ {0,...,P − 1}]
    {
        int C=0, mytime;                                    // Uhr
        int is_requesting=0, reply_pending, reply_deferred[P]={0,...,0};  // abgewiesene Prozesse

        process M[int p‘ = p]                               // der Monitor
        {
            int msg, time;
            while(1) {
                recv_any(π,q,msg,time);                     // Empfange von q’s Monitor mit Zeitmarke
                if (msg==REQUEST)                           // q will eintreten
                {
                    [Lclock(time);]                         // Erhöhe eigene Uhr für spätere Anfragen.
                                                            // Kritischer Abschnitt, da Π auch erhöht.
                    if(is_requesting ∧ mytime < time)
                        reply_deferred[q]=1;                // q soll warten
                    else
                        asend(M_q,p,REPLY,0);               // q darf ’rein
                }
                else reply_pending−−;                       // es war ein REPLY
            }
        }
        ...

}
```

Verteilter wechselseitiger Ausschluss mit Zeitmarken III

Programm (Verteilter wechselseitiger Ausschluss mit Lamport-Zeitmarken cont.)

```
parallel DME-timestamp // Distributed Mutual Exclusion cont.
{
    . . .
        void enter_cs()                                   // zum Eintreten in den kritischen Abschnitt
        {
            int i;
            [ mytime=Lclock(0); is_requesting=1; ]
                                                          // kritischer Abschnitt
            reply_pending=P − 1;                          // so viele Antworten erwarte ich
            for (i=0; i < P; i++)
                if (i ≠ p) send(M_i,p,REQUEST,mytime);
            while (reply_pending> 0);                     // bisi wait
        }
        void leave_cs()
        {
            int i;
            is_requesting=0;
            for (i=0; i < P; i++)                         // benachrichtig wartende Prozesse
                if (reply_deferred[i]
                {
                    send(M_i,p,REPLY,0);
                    reply_deferred[i]=0;
                }
        }
        enter_cs(); /* critical section */ leave_cs();
    } // end process
```

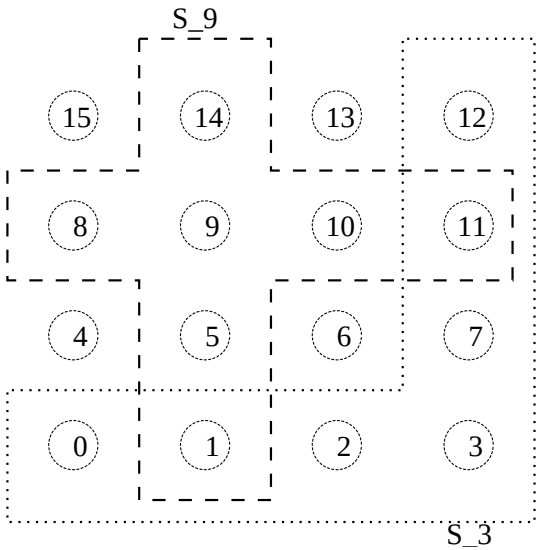
Verteilter wechselseitiger Ausschluss mit „Wählen“ I

- Obiger Algorithmus braucht $2P$ Nachrichten pro Prozess um in den kritischen Abschnitt einzutreten. Beim Wählen werden wir mit $O(\sqrt{P})$ auskommen.
- Insbesondere muss ein Prozess nicht *alle* anderen fragen bevor er rein darf.
- Idee:
  - Die entsprechenden Prozesse bewerben sich um das Eintreten in den kritischen Abschnitt. Diese heissen *Kandidaten*
  - Alle (oder einige, s. u.) stimmen darüber ab wer eintreten darf. Diese heissen *Wähler*. Jeder kann Kandidat und Wähler sein.
  - Statt absoluter Mehrheit verlangen wir nur relative Mehrheiten: Ein Prozess darf eintreten sobald er weiss, dass kein anderer mehr Stimmen haben kann als er.
- Jedem Prozess wird ein Wahlbezirk $S_p \subseteq \{0, \ldots, P-1\}$ zugeordnet. Es gelte die Überdeckungseigenschaft:

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

Verteilter wechselseitiger Ausschluss mit „Wählen“ II

- Die Wahlbezirke für 16 Prozesse sehen so aus:



- Ein Prozess p kann eintreten, wenn er alle Stimmen seines Wahlbezirkes bekommt. Denn kein anderer Prozess q kann eintreten: Nach Vor. existiert $r \in S_p \cap S_q$ und r hat sich für p entschieden, somit kann q nicht alle Stimmen haben.
- Deadlockgefahr: Ist $|S_p \cap S_q| > 1$ so kann sich einer für p und ein anderer für q entscheiden, beide kommen dann nie dran. Lösen des Deadlocks mit Lamport Zeitmarken.

Optimalität der Wahlbezirke I

- Frage: Wie klein können die Wahlbezirke sein?
- Nochmal: Jedes p hat seinen Wahlbezirk $S_p \subseteq \{0, \ldots, P-1\}$ und wir fordern $S_p \cap S_q \neq \emptyset$.
- Dies würde aber z. B. $S_p = \{0\}$ für alle p erlauben, was wir nicht wollen.
- Definiere D_p als die Menge aller Prozessoren für die p wählen muss:

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

- Wir fordern nun zusätzlich, dass für alle p:

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

  Dies schliesst obige Triviallösung aus.
- Unter dieser Annahme gilt sogar $D = K$, denn definiere die Menge aller Paare (p, q) mit p wählt für q, d.h. :

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

Optimalität der Wahlbezirke II

- Andererseits definiere die Menge aller Paare (p, q) wobei p von q gewählt werden muss:

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

  Wegen $q \in S_p \Leftrightarrow p \in D_q$ gilt $(p, q) \in B \Leftrightarrow (q, p) \in A$ also $|A| = |B|$. Für die Größen gilt $|A| = P \cdot D$ und $|B| = P \cdot K$ also $D = K$.
- Für festes $K (= D)$ maximieren wir nun die Zahl der Wahlbezirke (Prozessoren) P:
  - Wähle einen beliebigen Wahlbezirk S_p. Dieser hat K Mitglieder.
  - Wähle ein beliebiges $r \in S_p$. Dieses r ist Mitglied in D Wahlbezirken (Menge D_r) wovon einer S_p ist (offensichtlich ist $p \in D_r$. Somit zählen wir $K(D - 1) + 1$ Wahlbezirke.
  - Mehr kann es nicht geben, denn für beliebiges q gilt: Es gibt ein r mit $r \in S_p \cap S_q$ und somit $q \in D_r$. Wir haben also alle erfasst.

  Damit gilt also

$$\boxed{P \leq K(K - 1) + 1}$$

  oder

$$K \geq \frac{1}{2} + \sqrt{P - \frac{3}{4}}.$$

Wählen: Implementierung I

Programm (Verteilter wechselseitiger Ausschluss mit Wählen)

```
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()                                  // will in kritischen Abschnitt eintreten
        {
            int i, msg, time, yes_votes=0;
            [ mytime=Lclock(0); ]                        // Zeit meiner Anfrage
            for (i ∈ S_p) asend(V_i,p, REQUEST,mytime);
                                                         // sende Anfrage an Wahlbezirke
            while (yes_votes < |S_p|) {
                recv_any(π,q,msg,time);                  // empfange von q
                if (msg==YES) yes_votes++;               // q wählt mich
                if (msg==INQUIRE)                        // q will Stimme zurück
                    if (mytime==time)                    // nur aktuelle Anfrage
                    {                                    // es könnten noch alte unterwegs sein
                        asend(V_q,p,RELINQUISH,0);

                                                         // gib zurück
                        yes_votes−−;
                    }
            }
        }// end enter_cs
        . . .
```

Wählen: Implementierung II

Programm (Verteilter wechselseitiger Ausschluss mit Wählen cont. 1)

```
parallel DME-Voting cont. 1
{
    . . .
        void leave_cs()
        {
            int i;
            for (i ∈ S_p) asend(V_i,p,RELEASE,0);
            // Es könnten noch nicht bearbeitete INQUIRE-Messages für diesen
            // kritischen Abschnitt anstehen, die nun obsolet sind.
            // Diese werden dann in enter_cs ignoriert.
        }

        // Beispiel:
        enter_cs();
        . . . ; // kritischer Abschnitt
        leave_cs();


    }
```

Wählen: Implementierung III

Programm (Verteilter wechselseitiger Ausschluss mit Wählen cont. 2)

```
parallel DME-Voting cont. 2
{
    process V[int p' = p]                                   // der Voter zu Π_p
        {
            int q, candidate, msg, time, have_voted=0, candidate_time, have_inquired=0;
            while(1)                                        // läuft für immer
            {
                recv_any(π,q,msg,time);                     // empfange sie mit Absender
                if (msg==REQUEST)                           // Anfrage eines Kandidaten
                {
                    [ Lclock(time); ]                       // Uhr weiterschalten für spätere Anfragen
                    if (¬have_voted) {                      // ich habe meine Stimme noch zu vergebe
                        asend(Π_q,p,YES,0);                 // zurück an Kandidaten-Prozess
                        candidate_time=time;                // merke, wem ich meine
                        candidate=q;                        // Stimme gegeben habe.
                        have_voted=1;                       // ja, ich habe schon gewählt
                    }
                    else{                                   // ich habe schon gewählt
                        Speichere (q, time) in Liste;
                        if (time < candidate_time ∧ ¬have_inquired)
                        {                                   // hole Stimme vom Kandidaten zurück!
                            asend(Π_candidate,p,INQUIRE,candidate_time);
                                // an der candidate_time erkennt er, um welche Anfrage
                                // es geht: es könnte sein, dass er schon eingetreten ist.
                            have_inquired=1;
                        }
                    }
                } ...
```

Wählen: Implementierung IV

Programm (Verteilter wechselseitiger Ausschluss mit Wählen cont. 3)

```
parallel DME-Voting cont. 3
{
    . . .                                          // q ist der Kandidat, der mir meine
                else if (msg==RELINQUISH)          // Stimme zurück gibt.
                {
                    Speichere (candidate, candidate_time) in Liste;
                    Entnehme und Lösche
                        den Eintrag mit der kleinsten time aus der Liste: (q, time)
                                                   // Es könnte schon mehrere geben
                    asend(Π_q,p,YES,0);            // gebe dem q meine Stimme
                    candidate_time=time;           // neuer Kandidat
                    candidate=q;
                    have_inquired=0;               // kein INQUIRE mehr unterwegs
                }
                else if (msg==RELEASE)             // q verlässt den kritischen Abschnitt
                {
                    if (Liste ist nicht leer)
                    {                              // vergebe Stimme neu
                        Entnehme und Lösche
                            den Eintrag mit der kleinsten time aus der Liste: (q, time)
                        asend(Π_q,p,YES,0);
                        candidate_time=time;       // neuer Kandidat
                        candidate=q;
                        have_inquired=0;           // vergiss alle INQUIREs weil obsolet
                    }
                    else
                        have_voted=0;              // niemand mehr zu wählen
                }
```

Verteilte Terminierung I

Es seien die Prozesse $\Pi_0, \ldots, \Pi_{P-1}$ gegeben, welche über einen Kommunikationsgraphen kommunizieren.

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

Dabei schickt Prozess Π_i Nachrichten an die Prozesse

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

```
process Π_i [ int i ∈ {0, ..., P − 1}]
{
    while (1)
    {
        recv_any(who,msg),                    // Π_i ist idle
        compute(msg);
        for ( p ∈ N_msg ⊆ N_i )
        {
            msg_p = ... ;
            asend(Π_p, msg_p);                // vernachlässige Pufferproblem

        }
    }
}
```

Verteilte Terminierung II

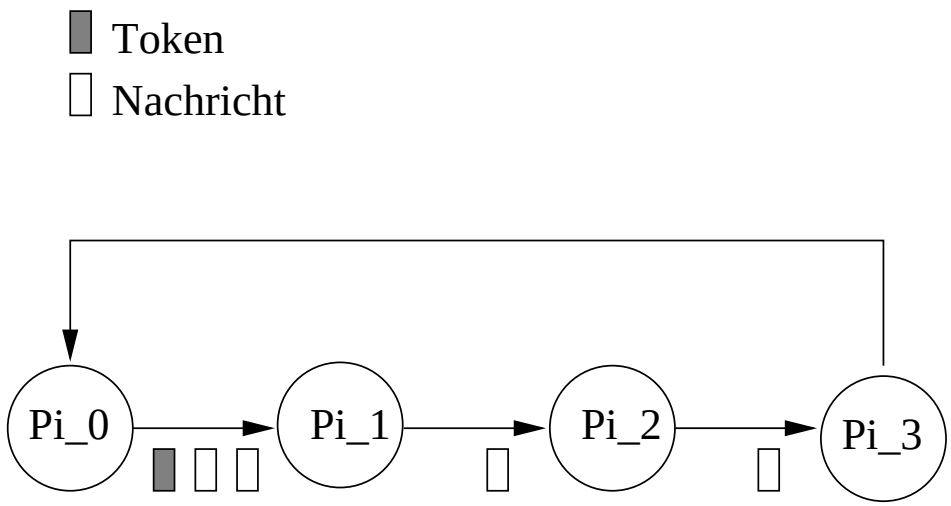
Das Terminierungsproblem besteht darin, daß das Programm beendet ist, falls gilt:

1. Alle warten auf eine Nachricht (sind idle)
2. Keine Nachrichten sind unterwegs

Dabei werden folgende Annahmen über die Nachrichten gemacht:

1. Vernachlässigung von Problemen mit Pufferüberlauf
2. Die Nachrichten zwischen zwei Prozessen werden in der Reihenfolge des Absendens bearbeitet

1.Variante: Terminierung im Ring



Verteilte Terminierung III

Jeder Prozess hat einen von zwei möglichen Zuständen: rot (aktiv) oder blau (idle). Zur Terminierungserkennung wird eine Marke im Ring herumgeschickt.
Angenommen, Prozess Π_0 startet den Terminierungsprozess, wird also als erster blau. Weiter angenommen,

1. Π_0 ist im Zustand blau
2. Marke ist bei Π_i angekommen und Π_i hat sich blau gefärbt

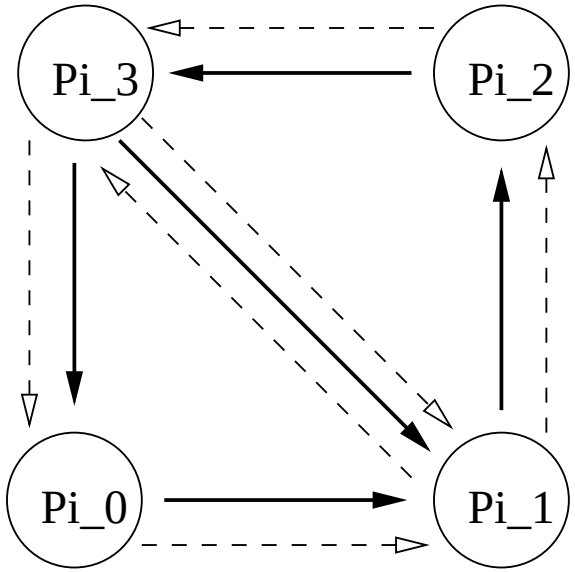
Dann kann gefolgert werden, daß die Prozesse $\Pi_0, \ldots, \Pi_i$ idle sind und die Kanäle $(\Pi_0, \Pi_1), \ldots, (\Pi_{i-1}, \Pi_i)$ leer sind.
Ist die Marke wieder bei Π_0 und ist dieser immer noch blau (was er ja feststellen kann), so gilt offenbar:

1. $\Pi_0, \ldots, \Pi_{P-1}$ sind idle
2. Alle Kanäle sind leer

Damit ist die Terminierung erkannt.

Verteilte Terminierung IV

2.Variante: Allgemeiner Graph mit gerichteten Kanten

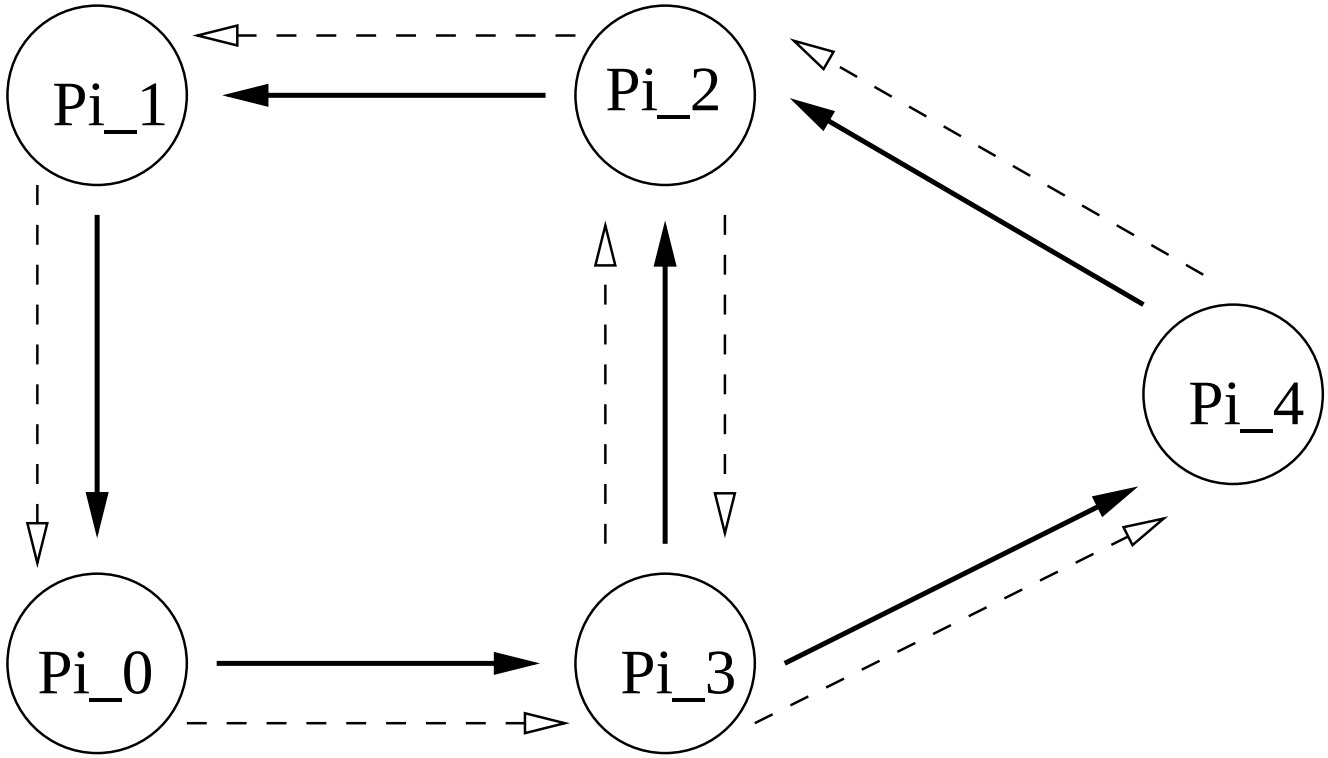


Idee: Über den Graph wird ein Ring gelegt, der alle Knoten erfasst, wobei ein Knoten auch mehrmals besucht werden kann.
Algorithmus: Wähle einen Pfad $\pi = (\Pi_{i_1}, \Pi_{i_2}, \ldots, \Pi_{i_n})$ der Länge n von Prozessoren derart aus, daß gelten:

1. Jede Kante $(\Pi_p, \Pi_q) \in E$ kommt mindestens einmal im Pfad vor
2. Eine Sequenz (Π_p, Π_q, Π_r) kommt höchstens einmal im Pfad vor. Erreicht man also q von p aus, so geht es immer nach r weiter. r hängt also von Π_p und Π_q ab: $r = r(\Pi_p, \Pi_q)$

Verteilte Terminierung V

Beispiel mit $\pi = (\Pi_0, \Pi_3, \Pi_4, \Pi_2, \Pi_3, \Pi_2, \Pi_1, \Pi_0)$.



Verteilte Terminierung VI

```
process Π [ int i ∈ {0, . . . , P − 1}]
{
    int color = red , token;
    if (Π_i == Π_{i_1} )
    {   // Initialisierung des Tokens
        color = blue;
        token = 0 ,
        asend(Π_{i_2} , TOKEN, token)
    }
    while(1)
    {
        recv_any(who,tag,msg);
        if ( tag != TOKEN ) { color = red; Rechne weiter }
        else        // msg = Token
        {
            if ( msg == n ) { break; „hurra, fertig! “}
            if ( color == red )
            {
                color = blue ;
                token = 0 ;
                rcvd = who ;
            }
            else

                if ( who == rcvd ) token++ ; // ein kompletter Zyklus

            asend(Π_{r(who,Π_i)}, TOKEN , token );
        }
    }
}
```

Verteilte Philosophen

Wir betrachten noch einmal das Philosophenproblem, diesmal jedoch mit message passing.

- Lasse eine Marke im Ring herumlaufen. Nur wer die Marke hat, darf eventuell essen.
- Zustandsänderungen werden dem Nachbarn mitgeteilt, **bevor** die Marke weitergeschickt wird.
- Jedem Philospohen P_i ist ein Diener W_i zugeordnet, der die Zustandsmanipulation vornimmt.
- Wir verwenden nur synchrone Kommunikation

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

    }
}

Verteilte Philosophen

```
process W_i [ int i ∈ {0, ..., 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 ; // Mein Philospoph
        if ( who == W_L & & tag ≠ TOKEN ) stateL = tag ; // state change
        if ( who == W_R & & tag ≠ TOKEN ) stateR = tag ; // in Nachbarn
        if ( tag == TOKEN){
            if ( state ≠ 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 ≠ EAT ){
                state = THINK;
                send( W_L , THINK );
                send( W_R , THINK );
            }
            send( W_L , TOKEN );
        }
    }
}
```