Seminar
Verteilte Simulation
Konservative Synchronisation und Time Warp *

Jan Wedekind †

21.1.1999

Inhaltsverzeichnis

*Prof. Wettstein, Institut für Dialogs- und Betriebssysteme, Universität Karlsruhe

†Ein Dank an den wiss. Mitarbeiter Gilles für die vielen Hinweise und an Herrn Liefländer für das nochmalige Lesen

1 Einführung

Die verteilte Simulation behandelt die Simulation eines *physikalischen Systems* auf Mehrprozessor-Systemen (Supercomputern). Die Gründe, eine Simulation verteilt auszuführen liegen auf der Hand:

- Das *Preis-/Leistungsverhältnis* von Mehrprozessor-Systemen ist günstiger als das eines einzelnen Großrechners.
- Die *begrenzte Rechenleistung eines einzelnen Großrechners* kann durch paralleles Rechnen um ein Vielfaches überboten werden. Eine verteilte Simulation ist damit viel leistungsfähiger.
- Einfache Implementation von Simulatoren für Finite-Elemente-Systeme.

In Kapitel 2 führe Ich zunächst einige Begriffe ein, die verteilte Simulation faßbar machen.

In der verteilten Simulation unterscheidet man zwischen *konservativer* und *optimistischer* Synchronisation. In Kapitel 3 geht es um konservative Synchronisation und in Kapitel 4 wird die gebräuchlichste optimistische Synchronisationsmethode - das Time-Warp-Verfahren - behandelt.

2 Begriffe

In einer verteilten Simulation wird die Prozessimulation *nicht sequentiell* ausgeführt, sondern in mehrere **logische Prozesse** (LP's[1]) mit eigener **lokaler Zeit**[2] aufgespaltet [FCE94].

Es gibt *keine globale Ereignisliste* mehr. Die Gleichzeitigkeit[3] muß also über das **logische System**[4], das die LP's und deren Verknüpfungen bilden, wiederhergestellt werden.

Zwecks Kapselung der einzelnen Prozesse versucht man heute immer mehr, ohne die Verwendung von *gemeinsamen Speicher*[5] auszukommen [Fuj90]. Die logischen Prozesse kommunizieren mit anderen logischen Prozessen über **Botschaften**[6]. Falls gemeinsame Variablen notwendig werden, wird dieses Problem durch Lese- und Schreib-Botschaften gelöst. Auf diese Art läßt sich auch das Problem, die richtige Version einer sich ändernden Variable anzusprechen, besser lösen. Allerdings muß man darauf achten, daß allzu exzessive Verwendung von Botschaften zu schlechter Performanz führen kann.

[1] engl.: logical processes
[2] engl.: local clock
[3] engl.: concurrency
[4] engl.: logical system
[5] engl.: shared-memory
[6] engl.: messages

Bei verteilter Simulation unterscheidet man zwischen *statischen* und *dynamischen* Systemen. Ein **statisches System** unterliegt gegenüber dem allgemeinen Fall folgender Einschränkung:

Die Zahl der Prozesse ist von Anfang an festgelegt. Es können also nicht (dynamisch) neue Prozesse gestartet werden. Ein dynamisches System mit bekannter Obergrenze von Prozessen läßt sich allerdings auf ein statisches System abbilden, indem man sich die hinzukommenden Prozesse als von Anfang an existierende Prozesse vorstellt, die nur einen Teil der Zeit aktiv sind.

Ein **dynamisches System** ist das Gegenteil eines statischen Systems (ein nicht-statisches System).

In einem **logischen Netzwerk** ist die Wahrscheinlichkeit, mit der eine Botschaft von Prozeß p_1 nach p_2 gesendet wird, unabhängig vom Zustand des Prozesses p_1 (und unabhängig vom Zustand/Inhalt der Botschaft).

Das heißt, es muß sich von vorneherein ein System von *Verbindungen*[7] zwischen den Prozessen angeben lassen, bei dem sich für jede Verbindung die Häufigkeit, mit der diese benutzt wird, statistisch beschreiben läßt.

Jede Botschaft, die auftritt, geht über eine dieser Verbindungen. Man nennt dieses System aus Prozessen und statischen Verbindungen auch *Logisches Netzwerk*.

Falls es in einer Implementierung eine Botschaft gibt, die je nach Zustand des sendenden Prozesses mit verschiedenen Wahrscheinlichkeiten auftritt, so kann man dieses System u.U. immer noch als statisch betrachten, wenn es möglich ist, durch Ergänzen mit *Null-Botschaften*, die Wahrscheinlichkeit wieder unabhängig von dem Zustand des Prozesses zu machen.

In einer **synchronen Simulation** werden nur die Ereignisse parallel berechnet, die auch im simulierten physikalischen System zur gleichen Zeit stattfinden. In einer **asynchronen** Simulation ist dies nicht mehr gegeben.

3 Konservative Synchronisation

Die konservative Synchronisation ist ein asynchrones Verfahren. D.h. wenn zwei Ereignisse parallel simuliert werden, heißt dies nicht, daß sie im simulierten Modell auch parallel "stattfinden". Daher spricht man von den *lokalen Zeiten* der Prozesse.

Um konservative Synchronisation einfacher charakterisieren zu können, verwende Ich im weiteren folgenden Formalismus:

3.1 Formalismus

- P ist die *Menge aller logischen Prozesse*. Der Einfachheit halber wird bei der konservativen Simulation angenommen, daß jeder Prozeß vom Anfang bis zum Ende der Simulation existiert (*statisches System*).
- Die Symbole $p, p1, p2 \in P$ bezeichnen einzelne logische Prozesse.
- Die Funktion $T : R \times P \rightarrow R$ ordnet allen Prozessen $p_i \in P$ zu jedem realen Zeitpunkt t_r ihre *lokalen Zeiten* $T(t_r, p_i)$ zu.

[7] engl.: links

- $(m; t_m)$ ist eine *zeitmarkierte Botschaft*[8]. Eine zeitmarkierte Botschaft führt den Simulationszeitpunkt t_m des Ereignisses, das m beschreibt mit.

3.2 Definition

Eine *konservative Synchronisation* läßt sich folgendermaßen charakterisieren:

Es gilt das *Ausführungsparadigma*[9] [Fuj90]: Wenn zwei (zeitmarkierte) Botschaften $(m_1; t_{m_1})$ und $(m_2; t_{m_2})$ einem Prozeß zugeteilt werden, und $t_{m_1} < t_{m_2}$ gilt, wird m_1 vor m_2 bearbeitet.

Es gilt das *Lokale Kausalitäts-Prinzip* [10] [Fuj90]: Wenn eine Botschaft m_1 für einen Prozeß p eine Botschaft m_2 verursachen wird, so wird m_1 von p bearbeitet, bevor m_2 abgesendet wird. Zusammenhängende Ereignisse werden also in der Reihenfolge ihrer Zeitmarkierungen bearbeitet.

Für die lokalen Zeiten der Prozesse ergeben sich dadurch folgende Randbedingungen [FCE94]:

- Ein logischer Prozeß p erneuert seine lokale Zeit nur, wenn alle relevanten Informationen verfügbar sind. Umgekehrt bedeutet dies, wenn p die lokale Zeit $t = T(t_r, p)$ angenommen hat, daß für jede danach empfangene Botschaft $(m; t_m)$ gelten muß: $t_m \geq t$
- Die lokalen Zeiten der einzelnen logischen Prozesse schreiten monoton fort: $\forall p \in P, \forall t_r, t'_r \in R : t'_r \geq t_r \Rightarrow T(t'_r, p) \geq T(t_r, p)$

Der Zustand des logischen Systems ist daher immer "korrekt" im Sinne einer sequentiellen Simulation. Das heißt, sobald man alle einzelnen Prozessdaten für eine bestimmte Simulationszeit zusammengenommen hat, kann man bei dem resultierenden Modell von dessen Korrektheit ausgehen (Dies kann man bei optimistischer Synchronisation im allgemeinen nicht!).

3.3 Konservativ optimale Simulation

Man kann den Vorgang einer Simulation mit speziellen Werten als (markierten) gerichteten Graph $G = (E, K, M_E)$ betrachten. Die Knoten E in dem Graphen stellen einzelne *Berechnungen* dar, die zwischen dem Empfangen und Aussenden von Botschaften ausgeführt werden. Die Knoten sind mit der Rechenzeit der Berechnung markiert.

Die Kanten K stellen den *Datenfluß* zwischen den Berechnungen dar. Eine Kante kann in einem Fall eine Variable sein, die ein Datum speichert; oder die Kante ist eine Botschaft, die ein Datum überträgt. Die Funktion $M_E : E \to R$ ordnet jeder Berechnung $e \in E$ eine Rechenzeit zu (Abb. 1).

Dieser Graph ist azyklisch und hat (mehrere) Start- und Endknoten. Den längsten Pfad von einem Start- zu einem Endknoten in diesem Graph (bez. der Rechenzeiten) nennt man den *kritischen Pfad* [Fuj90]. Die Summe der Rechenzeiten in diesem Pfad stellt eine untere Schranke für die Rechenzeit der gesamten Simulation dar. Wenn eine Simulation in dieser Zeit ausgeführt werden konnte, so nennt man diese Simulation *konservativ-optimal*.

[8] engl.: time-stamped message
[9] engl.: execution paradigm
[10] engl.: local causality constraint

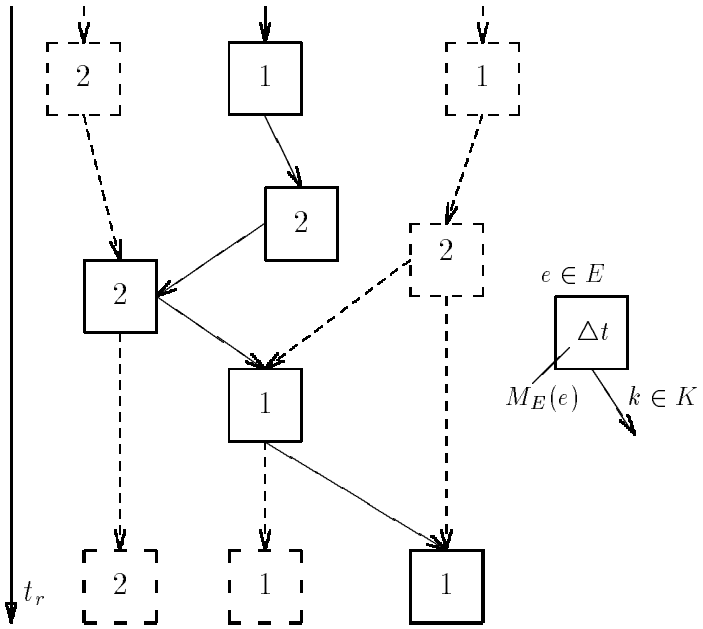

Abbildung 1: Kritischer Pfad einer Simulation

3.4 Aktualisierung der lokalen Zeit

Das Kernproblem bei konservativer Synchronisation ist also, herauszufinden, wann ein lokaler Prozeß sicher sein kann, keine Botschaft mehr zu empfangen, die es verbieten würde, weiterzurechnen.

3.4.1 Statischer Ansatz

Wenn es für die Simulation ein *logisches Netzwerk* (siehe Kap. 2) gibt, so weiß man für jeden Prozeß p_0, von welchen anderen Prozessen p_i dieser Botschaften zugeschickt bekommen kann. Da einer Botschaft $(m_a; t_{m_a})$ nie eine Botschaft $(m_b; t_{m_b})$ mit $t_{m_b} < t_{m_a}$ über dieselbe Verbindung folgen kann (vgl. Kap. 3.2), gibt das Minimum aller aus den einzelnen Verbindungen jeweils zuletzt empfangenen Zeitmarkierungen (der Botschaften) eine untere Schranke $T^+(t_r, p_0)$ für die Zeitmarkierungen aller Botschaften, die später empfangen werden, an [Fuj90] (z.B. in Abbildung 2 ist diese untere Schranke gleich 3). Die lokale Zeit des Prozesses p_0 kann also bis zu dieser Schranke erhöht werden.

Diese Technik alleine allerdings würde im besten Falle dazu führen, daß alle Prozesse die Lokalzeit des am weitesten fortgeschrittenen Prozesses übernehmen würden. Danach würde das System stehenbleiben.

3.4.2 Look-Ahead der Prozesse

Um also die Technik aus Kapitel 3.4.1 wirklich verwenden zu können, muß zusätzlich mindestens ein Prozeß die Möglichkeit haben, diese Schranke zu übertreten. Simulationen von physikalischen Systemen gehen oft in kleinen Zeitschritten vor. D.h. wenn ein Prozess p_1 eine Botschaft $(m_a; t_{m_a})$ an p_2 gesandt hat und in

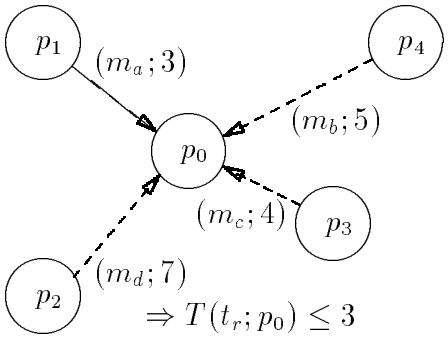

Abbildung 2: Hinr. Bed. für die lokale Zeit von p_0

p_2 bekannt ist, daß p_1 in Zeitschritten $\geq \triangle t_{min}$ rechnet, kann man in p_2 davon ausgehen, daß für die Zeitmarkierung der nächsten Botschaft $(m_b;t_{m_b})$ von p_1 gelten wird: $t_{m_b} \geq t_{m_a} + \triangle t_{min}$. Wenn man einen allgemein für alle Prozesse gültigen Wert $\triangle t_{min}$ angeben kann, so kann jeder Prozeß p_i seine lokale Zeit immer bis zu dem Wert $T^+(t_r, p_i) + \triangle t_{min}$ erhöhen.

Diese Technik ist eine spezielle Methode von *Look-Ahead*. Je nach dem Wissen über die physikalische Simulation werden auch andere Möglichkeiten verwendet, um Voraussagen über den Empfang von Botschaften zu machen.

3.4.3 Windowing

Beim *Windowing*[11] wird ein Zeitfenster definiert [Fuj90]:

- Die *kleinste Zeitmarkierung* t_m aller noch nicht bearbeiteten Botschaften $(m;t_m)$ legt den unteren Rand t_{start} des Fensters fest.
- Durch eine fest vorgegebene Fenstergröße und t_{start} wird dann der obere Rand t_{end} des Fensters definiert.

Beim *Windowing* werden nur Botschaften bearbeitet, deren Zeitmarkierungen im Fenster liegen. Wenn jetzt wie in Kapitel 3.4.2 ein minimales Zeitinkrement $\triangle t_{min}$ bekannt ist, kommen für das Aussenden von Botschaften, die einem Prozeß p verbieten würden, schon weiterzurechnen, nur noch Prozesse in Frage, die eine direkte logische Verbindung mit p haben oder eine indirekte Verbindung, deren Länge beschränkt ist.

Hinzu kommt: Je näher die lokale Zeit $T(t_r, p)$ von p am unteren Fensterrand t_{start} liegt, desto weniger Nachbarprozesse kommen in Frage.

Beispiel (siehe Abbildung 3) **:** In diesem Beispiel sieht man, daß Prozess p_0 bis zum Zeitpunkt t_1 nur Botschaften von direkt benachbarten Prozessen bekommen kann, bis zum Zeitpunkt t_2 nur Botschaften von Prozessen, die zwei Verbindungen entfernt sind.

3.4.4 Null-Botschaften

Man kann jeden Prozess nachdem die lokale Zeit erhöht wurde,über alle Verbindungen Botschaften senden lassen. Falls über eine Verbindung eigentlich kei-

[11] window, deutsch: Fenster

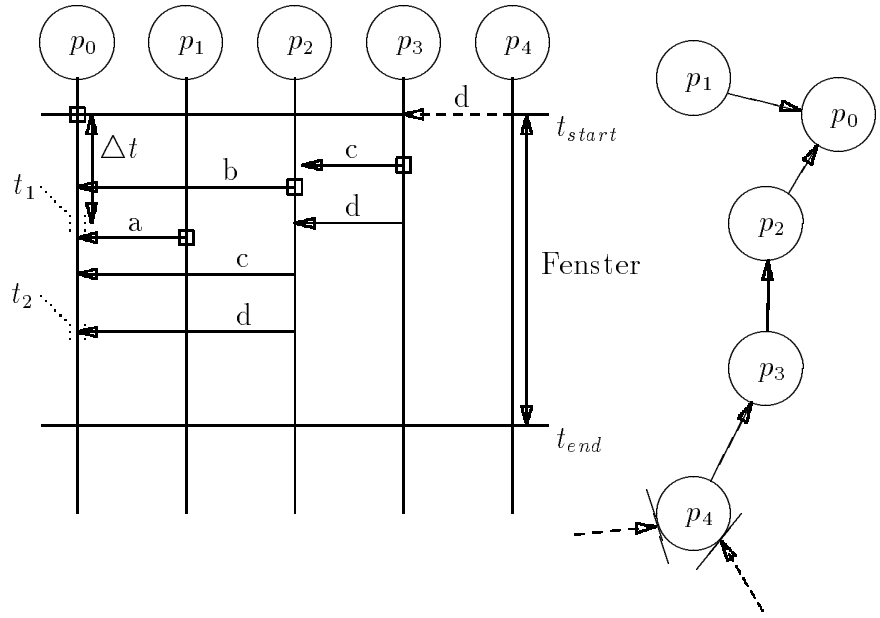

Abbildung 3: Beispiel für ein Zeitfenster

ne Botschaft gesendet werden müsste, sendet man dort stattdessen eine Null-Botschaft [Fuj90], die lediglich dazu dient, daß der Empfängerprozess seine lokale Zeit ebenfalls erhöhen kann.

Diese Technik hat allerdings den Nachteil, daß sehr viel Interprozess-Kommunikation geleistet werden muß. Dieser Ansatz führt in den meisten Fällen zu keinem guten Ergebnis.

3.5 Verklemmungsanalyse

3.5.1 Abhängigkeitsmenge

Wir definieren uns zunächst den Begriff der *Abhängigkeits-Menge*.

Zu jedem Prozess $p \in P$ gibt es zu jedem realen Zeitpunkt $t_r \in R$ die Abhängigkeitsmenge[12] D [CMH83]:

$$D(t_r, p_i) = \begin{cases} \emptyset \text{ wenn } p_i \textit{ ausführend}; \\ \{p_j \in P | p_i \text{ erw. Botsch. von } p_j\} \text{ sonst} \end{cases}$$

(Siehe auch Kapitel 3.5.2)

3.5.2 Prozesszustand

Zu jeder Zeit befindet sich ein Prozess in einem von zwei Zuständen: *wartend* oder *ausführend* [CMH83]. Nur Prozesse im ausführenden Zustand können Botschaften senden (in diesem Modell). Ein Prozess kann jederzeit in den Wartezustand treten. Ein Prozess verläßt den Wartezustand genau dann, wenn er eine Botschaft von *irgendeinen* Prozess in seiner Abhängigkeitsmenge empfängt!

[12] engl.: dependent set

Wenn ein Prozess also mehrere Botschaften braucht, um mit der Berechnung fortzufahren, so muß er nach dem Empfang einer Botschaft eben erneut in den Wartezustand eintreten. Der Grund für diese Art der Modellierung des Problems ist, daß sie den Algorithmus zur Verklemmungserkennung (siehe Kapitel 3.5.4) vereinfacht!

3.5.3 Verklemmung

An einer Verklemmung sind immer mehrere Prozesse beteiligt. Alle Prozesse p_i in einer Menge $S \subseteq P$ sind verklemmt, wenn gilt [CMH83]:

- Alle Prozesse in S sind im Wartezustand.
- Alle Abhängigkeits-Mengen $D(t_r, p_i)$ der Prozesse $p_i \in P$ sind Untermengen von S.
- Es werden keine Botschaften mehr ausgetauscht.

3.5.4 Algorithmus zur Verklemmungsanalyse

Der Algorithmus nach [CMH83] ist auf lokale Verklemmungsanalyse hin ausgelegt. Mit dem Algorithmus lassen sich Verklemmungen von wartenden Prozessen aus feststellen.

Der Algorithmus gewährleistet, daß in jeder "Verklemmungsmenge" S mindestens ein Prozess erfährt, daß er an einer Verklemmung beteiligt ist!

Die Voraussetzungen für den Algorithmus sind:

- Es ist immer für jeden wartenden Prozess die Abhängigkeitsmenge bekannt.
- Die Prozesse *terminieren nicht.* D.h. wenn ein Prozess wartend ist, so wird mindestens eine Botschaft von einem anderen Prozess erwartet. Formal bedeutet das: $\forall t_r \in R, p \in P : D(t_r, p_i) = \emptyset \rightarrow p_i$ ist nicht im Wartezustand.

Der **Algorithmus:** (siehe Abb. 4 und 5)

1. Wenn ein Prozess $p_i \in P$ in den Wartezustand geht, löst er eine Zustandsberechnung[13] aus. D.h. jedem Prozess $p_j \in D(t_r, p_i) \subseteq P$ wird eine Anfrage gesendet, ob von diesem Prozess wieder eine Botschaft kommen kann, die es p_i ermöglichen würde, den Wartezustand zu verlassen. Falls von keinem Prozess p_j eine Botschaft kommen kann, besteht eine Verklemmung!
2. Wenn in einem wartenden Prozess p_k eine Anfrage von p_i direkt oder indirekt ankommt, so wird diese wiederum an alle Prozesse in der Abhängigkeitsmenge $D(t_r, p_k)$ dieses Prozesses weitergegeben. Wenn alle diese weitergegebenen Anfragen (negativ) beantwortet wurden, so wird die Anfrage, die an p_k selber gestellt wurde, (negativ) beantwortet.
3. Wenn ein wartender Prozess eine zweite Anfrage bekommt, so wird diese sofort (negativ) beantwortet! Sonst würde dieser Algorithmus nämlich selber in eine Verklemmung laufen.

[13] engl.: query computation

Algorithmus zur Verklemmungsanalyse

- Zustandsberechnung von einem wartenden Prozess p_i aus beginnen:

```
begin
   latest(i):=latest(i)+1;
   wait(i):=true
   sende query(i,latest(i),i)
      an alle Prozesse p_j ∈ D(t_r, p_i);
   num(i):=Anzahl der Elemente in D(t_r, pi)
end
```

- Für einen ausführenden Prozess p_k: Nach dem Verlassen des Wartezustandes wird für alle `i wait(i)=false` gesetzt. Alle Anfragen und Antworten werden ignoriert während der Ausführungsphase.

- Für einen wartenden Prozess p_k, der `query(i,m,j)` empfängt:

```
if m>latest(i)
then begin
       latest(i):=m;
       engager(i):=j;
       wait(i):=true;
       für alle Prozesse p_r ∈ D(t_r, p_k)
          sende query(i,m,k);
       num(i):=Anzahl Prozesse in D(t_r, p_k)
     end
else if wait(i) and m=latest(i)
     then sende reply(i,m,k) an p_j
```

- Für einen wartenden Prozess p_k bei Empfang von `reply(i,m,r)`:

```
if m=latest(i) and wait(i)
then begin
       num(i):=num(i)-1;
       if num(i)=0
       then if i=k
         then erkläre p_k als verklemmt
         else sende reply(i,m,k) an p_j
                 wobei j=engager(i)
     end
```

Abbildung 4: Algorithmus zur Verklemmungsanalyse [CMH83]

Bedeutung der Variablen im Algorithmus
Jeder Prozess p_k hat folgende Variablen:

- `query(i,m,j)`: Anfrage von Prozess p_j aufgrund der m. von p_i ausgelösten Zustandsberechnung.
- `reply(i,m,j)`: (negative) Antwort von Prozess p_j auf Anfrage dieses Prozesses aufgrund der m. von p_i ausgelösten Zustandsberechnung.
- `latest(i)`: Aktuellste bekannte Nummer der von p_i ausgelösten Zustandsberechnung.
- `wait(i)`: `true` $\Leftrightarrow$ seit dem letzten Empfang von einer Anfrage `reply(i,latest(i),?)` hat dieser Prozess den Wartezustand nicht mehr verlassen.
- `num(i)`: Zahl der noch nicht (negativ) beantworteten Anfragen `query(i,latest(i),k)` dieses Prozesses.
- `engager(i)`: Das `j` der ersten Anfrage der Form `query(i,latest(i),j)` an diesen Prozess.

Abbildung 5: Variablen-Erklärung zum Algorithmus in Abb. 4

4. Jede Zustandsberechnung ist eindeutig dem initiierenden Prozess zugeordnet. Falls mehrere Prozesse Zustandsberechnungen initiiert haben, werden die zugehörigen Anfragen unabhängig voneinander behandelt!
5. Ein Prozess im ausführenden Zustand ignoriert alle Anfragen.
6. Ein Prozess der Anfragen gestellt hat und danach im ausführenden Zustand ist oder war, ignoriert alle zu diesen Anfragen gehörigen Antworten. Dadurch werden ungültige Zustandsberechnungen abgebrochen.
7. Jede Zustandsberechnung, die von einem Prozess p_i ausgelöst wurde, hat eine eigene Nummer, damit jeder Prozess weiß, daß dies eine neue Zustandsberechnung ist.

Dieser Algorithmus verlangt nicht, daß ein Prozess, der in den Wartezustand geht, sofort eine Zustandsberechnung auslöst [CMH83]. Man kann die Prozesse auch erst etwas warten lassen. Falls nämlich gar keine Verklemmung existiert, spart man auf diese Art unter Umständen sehr viel Kommunikationszeit.

Algorithmen zur Deadlock-Auflösung habe Ich hier nicht aufgeführt. Außerdem ist noch zu erwähnen, daß oft die Synchronisation so ausgelegt wird, daß gar keine Verklemmungen auftreten können.

3.6 Aktuelle Anwendungsbeispiele für konservative Synchronisation

3.6.1 Strömungsrechnung

Daimler-Benz Debis, Porsche, sowie die Universitäten Stuttgart und Karlsruhe betreiben in Stuttgart ein Rechenzentrum [Sch98b]:

In einem VR-Raum lassen sich Konstruktionsaufgaben virtuell erledigen. Es werden vor allem *Strömungs- und Verbrennungsvorgänge im Kolbenraum eines Motors* simuliert. Dabei müssen Differentialgleichungssysteme angenähert werden (*Navier-Stokes-Gleichungen*). Die verteilte Simulation hilft hier, Produktentwicklungszeiten zu verkürzen. Gerechnet wird vor allem auf einem NEC Cluster SX4/36 H2 und einer Silicon Graphic Onyx 2 mit 14 R10000 Prozessoren und 4 Gigabyte Hauptspeicher. Unter anderem rechnet in Karlsruhe noch ein IBM-Supercomputer mit.

3.6.2 Wettervorhersage

Der Deutsche Wetterdienst unterhält in Offenbach einen Cray-T3E-1200E Supercomputer von Silicon Graphics, der in der ersten Aufbaustufe mit 456 Prozessoren mit insgesamt 600 Gigaflops bestückt wurde [Sch98a] (zur Jahrtausendwende soll der Supercomputer mit 1024 Prozessoren arbeiten):

Aus 25000 Wetterbeobachtungen pro Stunde, die zum Großteil von Wetterstationen und Satelliten stammen, wird zweimal täglich ein Ausgangsmodell geschaffen und eine Vorausberechnung des Wetters auf bis zu eine Woche gemacht. Das simulierende Gittermodell hat über Deutschland eine Maschenweite von 14 km und 20 atmosphärische Schichten und wird mit internen Zeitschritten von 4 Minuten gerechnet. Ein Gitterpunkt braucht 3000-5000 Flops pro Zeitschritt. In den Modellen werden Wolken, Wasserdampf, Strahlung, Konvektion, Niederschlag, Wärmefluß, Verdunstung, Bodenfeuchte und Schneeschmelze berücksichtigt.

4 Das Time-Warp-Verfahren

Das Time-Warp[14]-Verfahren ist eine *optimistische Synchronisationsmethode*. Optimistische Methoden zeichnen sich durch folgende Eigenschaften aus:

- Alle Prozesse rechnen, ohne auf Botschaften zu warten. Es wird dabei darauf spekuliert, daß öfters kein Einfluß durch andere Prozesse stattfindet.
- Ein Prozeß p ist in der Lage ein *Roll-Back* auszuführen. Ein *Roll-Back* wird nötig, wenn eine Botschaft $(m;t)$ mit $t < T(p)$ eintrifft, d.h. die Berechnungen ab dem Zeitpunkt t falsch sind, weil sie m nicht berücksichtigt haben.

4.1 Formalismus

- Wenn ein Prozeß p nie mehr ein Roll-Back ausführen muß, so daß $T(t_r, p)$ wieder kleiner als t würde, so sagt man, daß zum Zeitpunkt t_r alle Berechnungen von p bis zum Simulationszeitpunkt t *wahr* sind [LL91]. Man spricht dann von einer *wahren Berechnung*[15].
- Wenn eine Botschaft nicht wahr ist, wird sie irgendwann durch eine Gegenbotschaft $\overline{m}$ gelöscht werden. $(\overline{m};t_m)$ sagt dem empfangenden Prozess, daß $(m;t_m)$ *falsch* war.

[14] deutsch: Zeit-Verzerrung
[15] engl.: true calculation

- Eine *wahre Botschaft* ist eine Botschaft, die nicht mehr durch eine Gegenbotschaft gelöscht wird [LL91].

Eine wahre Berechnung wird also keine falschen Botschaften auslösen.
Allerdings kann der *Inhalt* einer falschen Botschaft m durchaus korrekt sein. Die Bezeichnungen "falsche Botschaft" bzw. "wahre Botschaft" für eine Botschaft m sind hier nicht im intuitiven Sinne zu verstehen. Sie beziehen sich nicht auf den Inhalt von m, sondern lediglich darauf, ob m später widerrufen wird oder nicht.

4.2 Roll-Back

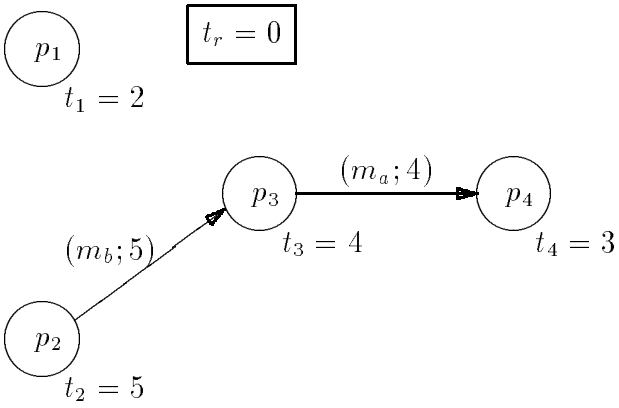

Abbildung 6: Roll-Back-Beispiel: $t_r = 0$

Roll-Back läßt sich am besten anhand eines Beispiels erklären:

$\mathbf{t_r = 0}$ (Abb. 6): Seien p_1 bis p_4 Prozesse. Zum (realen) Zeitpunkt $t_r = 0$ haben die Prozesse die in der Abbildung angegebenen lokalen Zeiten $t_i = T(t_r, p_i)$. Der Prozeß p_3 sendet die Botschaft m_a an p_4 und p_2 sendet m_b and p_3. p_3 speichert $(m_b; 5)$, da p_3 m_b erst berücksichtigen darf[16], wenn die lokale Zeit den Wert 5 erreicht hat (dasselbe gilt für p_4 und $(m_a; 4)$).

[16] Kausalitätsprinzip

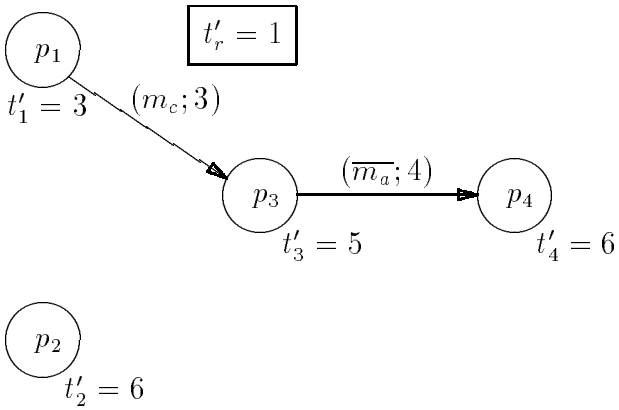

Abbildung 7: Roll-Back-Beispiel: $t'_r = 1$

$\mathbf{t'_r = 1}$ (Abb. 7): Zur Zeit $t'_r = 1$ entdeckt p_1 das Ereignis m_c, das p_3 betrifft, und sendet es. p_3 führt darauf hin ein Roll-Back aus: p_3 setzt seine lokale Zeit auf $t_{m_c} = 3$ zurück, und sendet die Gegenbotschaft $(\overline{m_a}; 4)$, da $t_{m_a} > t_{m_c}$.

Zu beachten ist:

- Bei einem Roll-Back wird nicht nur die lokale Zeit des betroffenen Prozesses zurückgesetzt, sondern auch dessen Zustand!
- p_3 muß die Botschaft $(m_a; 4)$ gespeichert haben, sonst kann von p_3 aus bei einem Roll-Back nicht mehr festgestellt werden, ob und an welchen Prozeß Gegenbotschaften gesendet werden müssen!
- Auch in p_4 muß ein Roll-Back ausgeführt werden, da die falsche Botschaft m_a berücksichtigt wurde. Unter Umständen kann dies bedeuten, daß p_4 auch wieder Gegenbotschaften senden muß!
- $\overline{m_a}$ löscht die Botschaft m_a aus dem Botschaften-Speicher von p_4.

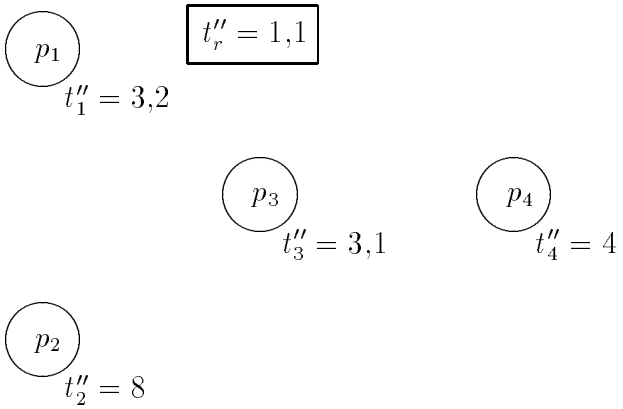

Abbildung 8: Roll-Back-Beispiel: $t''_r = 1{,}1$

$\mathbf{t''_r = 1{,}1}$ (Abb. 8): Zum Zeitpunkt $t''_r = 1{,}1$ hat p_3 seine Berechnungen ab der Simulationszeit 3,0 und p_4 ab der Sim.-Zeit 4,0 begonnen. p_1 und p_2 haben in diesem Beispiel ungestört weitergerechnet.

4.3 Abbrechen von Berechnungen

Das Abbrechen von Berechnungen[17] kann auf verschiedene Arten ausgeführt werden. In jedem Falle ist es von Vorteil, wenn man den Gegenbotschaften eine höhere Priorität als den Botschaften zuteilt, da diese sonst den Botschaften durch das ganze Netzwerk "hinterherlaufen".

4.3.1 Lazy Cancellation

Im Gegensatz zum intuitiven Verfahren (*direct/aggressive cancellation*[18]) gibt es das sogenannte *lazy cancellation*[19]. In diesem Falle sendet ein Prozess nach einem Roll-Back nicht sofort Gegenbotschaften aus, um vorher gesendete Botschaften aufzuheben [Fuj90]. Statt dessen wird gewartet, ob die neue Berechnung die *gleichen* Botschaften generiert; wenn die gleichen Botschaften generiert werden, ist es nämlich gar nicht nötig Gegenbotschaften auszulösen. Allerdings braucht dieses Verfahren Rechenzeit, um festzustellen ob die selben Botschaften generiert werden. Unter Umständen kann die Simulation mit diesem Verfahren wesentlich langsamer sein.

Lazy cancellation hat die interessante Eigenschaft, das damit die Simulationszeit einer *konservativ optimalen Simulation* unterschritten werden kann. Die Ausführungszeit des kritischen Pfades ist nämlich nicht mehr untere Grenze für die Rechenzeit, da das lokale Kausalitätsprinzip nicht mehr gilt (siehe Kapitel 3.2)!

4.3.2 Lazy Reevaluation

Wenn nach Bearbeitung einer Botschaft sich der Zustandsvektor eines Prozesses nicht geändert hat, und diese Botschaft durch eine Gegenbotschaft aufgehoben wird, so ist klar, daß bei einem Roll-Back die Wiederbearbeitung der darauffolgenden Botschaften genauso ausfallen wird [Fuj90]. Deshalb braucht man diese Botschaften nicht erneut zu bearbeiten. Stattdessen können diese übersprungen werden. Es wird allderdings ein Vergleich der Zustandsvektoren benötigt.

Wenn zum Beispiel "Lesebotschaften" häufiger vorkommen, so wird mit dieser Methode der Aufwand eingespart, den ein Roll-Back einer "Lesebotschaft" verursachen würde. Allerdings kann die Implementierung von Lazy Reevalutation den Programmcode deutlich komplexer machen [Fuj90]!

5 Statistik

Abbildung 9 zeigt, wie konservative Synchronisation bzw. Time Warp auf die Zahl von Botschaften reagieren: Der Beschleunigungsfaktor[20] bezeichnet den Geschwindigkeitszuwachs der parallelen Algorithmen gegenüber einer sequentiellen Ausführung.

Je mehr Botschaften ausgetauscht werden, desto weniger müssen die Prozesse bei der konservativen Simulation warten. Bei Time Warp führt eine höhere

[17] engl.: cancellation
[18] deutsch: direktes/aggressives Abbrechen
[19] deutsch: faules Abbrechen
[20] engl.: speedup-factor

Anzahl von Botschaften dazu, daß weniger Zeit für die Ausführung von Roll-Backs benötigt wird. In diesem Beispiel führt die Zerlegung von 16 in 64 Prozesse zu einer Verlangsamung beim konservativen Algorithmus!

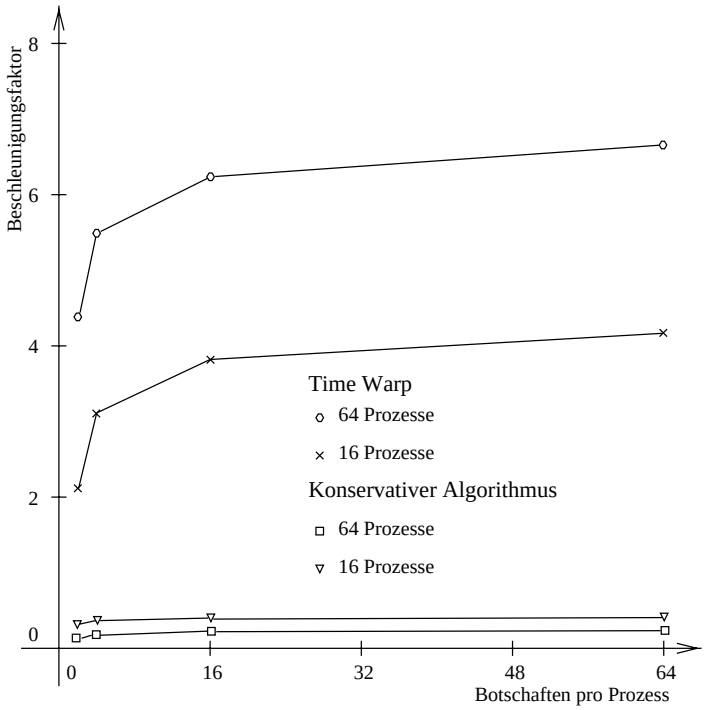

Abbildung 9: Experiment auf 8-Prozessor BBN Butterfly Multiprozessor [Fuj90]

Abbildung 10 zeigt ein Experiment, bei dem ein Beschleunigungsfaktor von 50 auf einem 64-Prozessor-System erreicht wurde. Man sieht, daß der Beschleunigungsfaktor nicht linear mit der Zahl der verwendeten Prozessoren ansteigt. Das liegt vor allem daran, daß die Zeit, die insgesamt für die Übertragung der Botschaften benötigt wird, mit der "Größe" des Netzwerkes zunimmt.

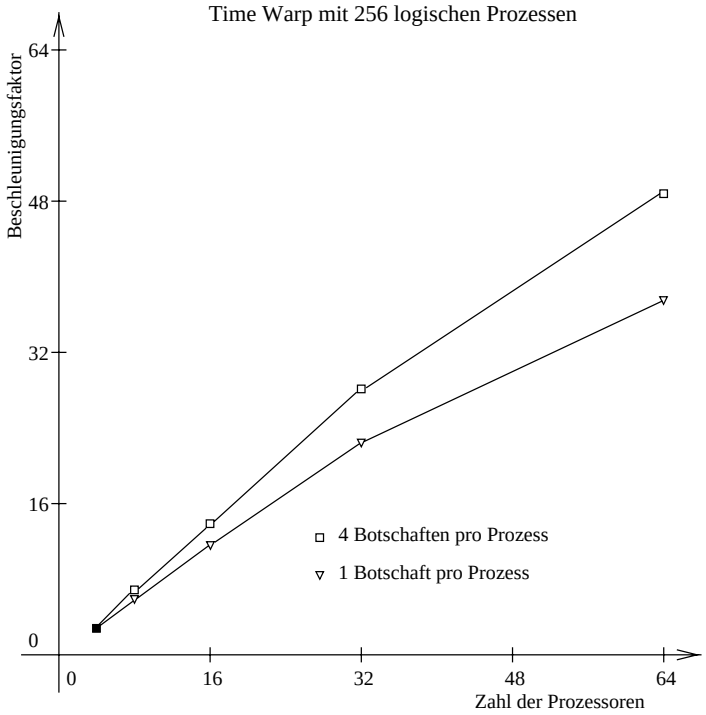

Abbildung 10: Experiment auf 64-Prozessor-System[Fuj90]

5.1 Anwendungen

5.1.1 Netzwerk-Simulation

Das Time-Warp-Verfahren wird in der Simulation von Rechner- und Kommunikations-Netzwerken verwendet. Vor dem Bau eines (teuren) Netzwerks wird zunächst simuliert.

Wenn man ein Netzwerk korrekt simuliert, kann man bessere Aussagen über die Leistung des Netzwerkes machen, als wenn man an einem existierendem Netzwerk Messungen ausführt. Man kann nämlich beliebig viel Beobachtungen ausführen lassen, ohne daß die hierfür benötigte Rechenzeit die Messergebnisse verfälscht.

5.1.2 VHSIC-Simulation

Auf der Web-Seite "http://www.ece.uc.edu/~paw/warped" findet sich ein Time-Warp-Kernel für Unix-Betriebssysteme. Mit diesem Kernel ist ein Simulator für VHSIC-Schaltungen entwickelt worden.

Der Simulator nimmt eine Schaltungs-Spezifikation in VHDL entgegen. Diese Spezifikation wird in C++ übersetzt. Der übersetzte Programmcode zusammen mit dem Time-Warp-Kernel simuliert dann die Schaltung.

Literatur

[CMH83] K. Mani Chandy, Jayadev Misra, and Laura M. Haas.
Distributed deadlock detection.
ACM Transactions on Computer Systems, 1(2):144–156, May 1983.

[FCE94] Richard M. Fujimoto, Christopher D. Carothers, and Paul England.
Effect of communication overheads on time warp performance: An experimental study.
In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, volume 24(1) of *ACM-Catalog*, pages 118–125, 6 July 1994.

[Fuj90] Richard M. Fujimoto.
Parallel discrete event simulation.
Communications of the ACM, 33(10):31–50, October 1990.

[LL91] Y-Bing Lin and Edward D. Lakowska.
A study of time warp rollback mechanisms.
ACM Transactions on Modeling and Computer Simulation, 1(1):51–72, January 1991.

[Sch98a] Gernot Schaermeli.
Das Supercomputing verleiht der lokalen Wettervorhersage endlich Präzision.
Computer-Zeitung, 29(41):6, 8 October 1998.

[Sch98b] Gernot Schaermeli.
Stuttgarter High-End-Simulationen machen komplexe Datensätze begreifbar.
Computer-Zeitung, 29(22):8, 28 May 1998.