# Parallele Programmierung mit C++
# SS 2017

Andreas F. Borchert
Universität Ulm

27. April 2017

Inhalte:

- Einführung in die Prozessalgebra CSP
- Architekturen paralleler Systeme
- Parallelisierungstechniken:
  - Threads
  - OpenMP
  - MPI
  - Koprozessoren mit CUDA, OpenCL und OpenACC
- Design-Pattern für die Organisation der Parallelisierung, der Algorithmen, der sie unterstützenden Verwaltung, der Kommunikation und Synchronisierung

# Warum parallelisieren? 3

Ohne Parallelisierung ist keine signifikante Leistungssteigerung mehr zu erwarten:

- Moore's law: Etwa alle zwei Jahre verdoppelt sich die Zahl der integrierten Transistoren, die sich auf einem integrierten Schaltkreis zu vertretbaren Kosten unterbringen lassen.
- Das bedeutet jedoch nicht, dass im gleichen Maße auch die Rechenleistung bzw. die Taktraten eines Prozessors steigen.
- Ein Teil der zusätzlichen Transistoren sind einem verbesserten Caching gewidmet und insbesondere der Unterstützung mehrerer Kerne (*multicore processors*).
- Dual-Core- und Quad-Core-Maschinen sind bereits weit verbreitet. Es gibt aber auch 16-Kern-Prozessoren mit 8 Threads per Kern (SPARC T5), Koprozessoren mit 72 Kernen (Intel Xeon Phi) und GPUs mit 4.992 Kernen (Nvidia Tesla K80).

# Warum parallelisieren? 4

Ohne Parallelisierung wird keine Skalierbarkeit erreicht:

- Google verteilt seine Webdienste über 15 Zentren. Eines dieser Zentren hatte einem älteren Bericht des *The Register* zufolge 45 Container mit jeweils 1.160 Rechnern.
- Der Supercomputer Sunway TaihuLight ist ausgestattet mit 160 sogenannten Supernodes, die jeweils aus 256 Sunway-RISC-Prozessoren umfassen, die jeweils aus vier Verwaltungsprozessoren mit je 64 Kernen bestehen. Insgesamt sind das 10.649.600 Kerne. Konkret verwendet werden hier C, C++ und Fortran in Kombination mit MPI, OpenMP und OpenACC. (Siehe Haohuan Fu et al, *The Sunway TaihuLight supercomputer: system and applications*.)
- Das bwUniCluster besteht im wesentlichen aus 512 Knoten mit jeweils zwei Intel-Xeon-Prozessoren mit je acht Kernen. Die Knoten sind miteinander über InfiniBand vernetzt (hier kommt wiederum MPI zum Einsatz).

- Traditionell ging häufig eine Parallelisierung von einer idealisierten Welt aus, bei der $n$ Prozesse getrennt arbeiteten und miteinander ohne nennenswerten Aufwand sich synchronisieren und miteinander kommunizieren konnten.
- Die Realität sieht jedoch inzwischen sehr viel komplexer aus:
  - Algorithmen können nicht mehr ohne tiefe Kenntnisse der zugrundeliegenden Architektur sinnvoll parallelisiert werden.
  - Der Zeitaufwand für Synchronisierung, Kommunikation und Speicherzugriffe ist von wesentlicher Bedeutung.
  - Parallelisierungen finden typischerweise auf drei Ebenen statt: Cluster, Prozessor-Kerne und Koprozessoren.
  - Bei einigen Koprozessoren wie insbesondere GPUs operieren die Kerne nicht unabhängig voneinander.

- Es gehört zu den Errungenschaften in der Informatik, dass Software-Anwendungen weitgehend plattform-unabhängig entwickelt werden können.
- Dies wird erreicht durch geeignete Programmiersprachen, Bibliotheken und Standards, die genügend weit von der konkreten Maschine und dem Betriebssystem abstrahieren.
- Leider lässt sich dieser Erfolg nicht ohne weiteres in den Bereich des *High Performance Computing* übertragen.
- Entsprechend muss die Gestaltung eines parallelen Algorithmus und die Wahl und Konfiguration einer geeigneten zugrundeliegenden Architektur Hand in Hand gehen.
- Ziel ist nicht mehr eine höchstmögliche Portabilität, sondern ein möglichst hoher Grad an Effizienz bei der Ausführung auf einer ausgewählten Plattform.

# Fokus der Vorlesung

- Der Schwerpunkt liegt bei den Techniken zur Parallelisierung von Anwendungen, bei denen Kommunikation und Synchronisierung eine wichtige Rolle spielen.
- Ziel ist es, die Grundlagen zu erlernen, die es erlauben, geeignete Architekturen für parallelisierbare Problemstellungen auszuwählen und dazu passende Algorithmen zu entwickeln.
- CSP dient im Rahmen der Vorlesung als Instrument zur formalen Beschreibung von Prozessen, die sich synchronisieren und miteinander kommunizieren.

- Für OpenMP, MPI, CUDA, OpenCL und OpenACC ist die Auswahl nicht sehr groß. Neben C++ kommen hier nur noch Fortran oder C in Frage und hier bietet C++ im Vergleich dann doch die moderneren Sprachkonzepte.
- Bei rechenintensiven Anwendungen spielen Cache-Optimierungen eine nicht zu unterschätzende Rolle. Diese können durchgeführt werden, wenn die Programmiersprache es erlaubt, das Layout von Datenstrukturen im Speicher genau festzulegen. Bei C++ ist dies problemlos möglich, bei der Mehrzahl der Alternativen ist das nicht vorgesehen.
- Ebenfalls von Bedeutung ist das Vorhandensein von Übersetzern mit fortgeschrittenen Optimierungstechniken (wie etwa *loop unrolling* und *instruction scheduling*), um das Pipelining moderner Architekturen auszureizen. Der Aufwand ist dafür polynomial und damit nur einmalig beim Übersetzen zumutbar – jedoch nicht zur Laufzeit (wie etwa mit einem JIT-Übersetzer bei Java).

- Grundkenntnisse in Informatik. Insbesondere sollte keine Scheu davor bestehen, etwas zu programmieren.
- Für diejenigen, die bislang weder C noch C++ kennen, gibt es in den Übungen zu Beginn eine kleine Einführung dazu.
- Freude daran, etwas auch an einem Rechner auszuprobieren und genügend Ausdauer, dass nicht beim ersten Fehlversuch aufgegeben wird.

# Syllabus: Struktur

- Jede Woche gibt es zwei Vorlesungsstunden an jedem Donnerstag von 10-12 Uhr in der Helmholtzstraße 22, Raum E.03.
- Die Übungen sind am Freitag von 14-16 Uhr in der Helmholtzstraße 18, Raum E.20.
- Da die Vorlesungstermine am 25. Mai und am 15. Juni wegen Feiertagen ausfallen, werden diese in anderen Wochen nachgeholt, indem der Übungstermin am Freitag auch für die Vorlesung genutzt wird.
- An den „Brückentagen“ am 26. Mai und 16. Juni finden keine Übungen statt.
- Webseite: `http://www.uni-ulm.de/mawi/mawi-numerik/lehre/sommersemester-2017/vorlesung-parallele-programmierung-mit-c/`

- Wir haben keinen Übungsleiter, keine Tutoren und auch keine Korrekteure.
- Lösungen zu Übungsaufgaben werden auf unseren Rechnern mit einem speziellen Werkzeug eingereicht. Details zu dem Verfahren werden zusammen mit der ersten Übungsaufgabe vorgestellt.
- Zu Beginn dienen die Übungen auch dazu, C++ einzuführen.

- Die Prüfungen erfolgen mündlich zu Terminen, die jeweils mit mir zu vereinbaren sind.
- Für die mündliche Prüfungen wird keine Vorleistung benötigt.
- Terminlich bin ich flexibel. Abgesehen von meinen Urlaubszeiten bin ich jederzeit bereit zu prüfen. Nennen Sie eine Woche, ich gebe Ihnen dann gerne einen Termin innerhalb dieser Woche.

- Die Vorlesungsfolien und einige zusätzliche Materialien werden auf der Webseite der Vorlesung zur Verfügung gestellt werden.
- Verschiedene Dokumente wie beispielsweise Tony Hoares Buch *Communicating Sequential Processes* oder Spezifikationen zu Threads, OpenMP und MPI stehen frei zum Herunterladen zur Verfügung.

- C. A. R. Hoare, *Communicating Sequential Processes*, ISBN 0131532898
- Timothy G. Mattson et al, *Patterns for Parallel Programming*, ISBN 0321228111
- Anthony Williams, *C++ Concurrency in Action*, ISBN 1-933988-77-0
- Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*, ISBN 0071232656
- Nicholas Wilt, *The CUDA Handbook*, ISBN 0-321-80946-7
- Matthew Scarpino, *OpenCL in Action*, ISBN 1-61729-017-3

# Syllabus: Sprechstunden

- Sie sind eingeladen, mich jederzeit per E-Mail zu kontaktieren:
  E-Mail: `andreas.borchert@uni-ulm.de`
- Meine reguläre Sprechstunde ist am Mittwoch 10:00-11:30 Uhr. Zu finden bin ich in der Helmholtzstraße 20, Zimmer 1.23.
- Zu anderen Zeiten können Sie auch gerne vorbeischauen, aber es ist dann nicht immer garantiert, daß ich Zeit habe. Gegebenenfalls lohnt sich vorher ein Telefonanruf: 23572.

# Syllabus: Nicht verzweifeln!

- Bevor Sie bei der Lösung einer Übungsaufgabe völlig verzweifeln, sollten Sie mir Ihren aktuellen Stand per E-Mail zukommen lassen. Dann werde ich versuchen, Ihnen zu helfen.
- Das kann auch am Wochenende funktionieren.

# Syllabus: Feedback

- Feedback ist ausdrücklich erwünscht.
- Noch ist die Vorlesung nicht fertig. Das bedeutet auch, dass ich auf Ihre Anregungen eingehen kann und auch Punkte mehr berücksichtigen kann, die Ihnen wichtig sind.

C. A. R. Hoare 1994

C. A. R. Hoare entwickelte von 1978-1985 die erste Prozessalgebra CSP (*Communicating Sequential Processes*). Obwohl es inzwischen einige alternative Prozessalgebren gibt, ist sie nach wie vor die bedeutendste geblieben.

Eine Prozessalgebra erlaubt die formale Beschreibung paralleler Systeme und die Ableitung ihrer innewohnenden Eigenschaften.

Die folgenden Ausführungen lehnen sich recht eng an das Buch von Hoare an.

# Ereignisklassen




Noebse, Wikimedia Commons

- Die Interaktionen zwischen Prozessen erfolgt über Ereignisse.
- Jedes Ereignis gehört einer Ereignisklasse an.
- Die für einen Prozess relevanten Ereignisklassen sind endlich und fest vorgegeben.
- Ereignisklassen bilden das Alphabet eines Prozesses und dienen der Abstrahierung.
- Der abgebildete Zeitungsautomat hat beispielsweise die Ereignisklassen *muenze5*, *muenze10*, *muenze20*, *muenze50*, *muenze100*, *zeitung_ausgeben* und *rueckgabe*.

- Ein konkretes für die Betrachtung relevantes Ereignis gehört einer Ereignisklasse an.
- Das Eintreten eines Ereignisses benötigt keine Zeit. Wenn längere Ereignisse modelliert werden sollen, kann dies erfolgen durch zwei verschiedene Ereignisklassen, die den Anfang und das Ende eines längeren Ereignisses markieren.
- Es gibt keinen Initiator eines Ereignisses und keine Kausalität.
- Prozesse werden dadurch charakterisiert, in welcher Reihenfolge sie an welchen Ereignissen teilnehmen.

- Ereignisklassen werden mit Kleinbuchstaben beschrieben. (Beispiele: *muenze5*, *a*, *b*, *c*.)
- Für die Namen der Prozesse werden Großbuchstaben verwendet. (Beispiele: *ZA*, *P*, *Q*, *R*.)
- Für Mengen von Ereignisklassen werden einzelne Großbuchstaben aus dem Anfang des Alphabets gewählt: *A*, *B*, *C*.
- Für Prozessvariablen werden die Großbuchstaben am Ende des Alphabets gewählt: *X*, *Y*, *Z*.
- Das Alphabet eines Prozesses *P* wird mit $\alpha P$ bezeichnet. Beispiel:

$$\begin{aligned} \alpha ZA \quad = \quad & \{\mathit{muenze5},\ \mathit{muenze10},\ \mathit{muenze20},\ \mathit{muenze50},\\ & \mathit{muenze100},\ \mathit{zeitung\_ausgeben},\ \mathit{rueckgabe}\} \end{aligned}$$

$$(x \to P)$$

- Gegeben sei eine Ereignisklasse $x$ und ein Prozess $P$. Dann beschreibt $(x \to P)$ einen Prozess, der zuerst an einem Ereignis der Klasse $x$ teilnimmt und sich dann wie $P$ verhält.
- Ein Zeitungsautomat, der eine Ein-Euro-Münze entgegennimmt und dann den Dienst einstellt, kann so beschrieben werden:

  $$(\mathit{muenze100} \to \mathit{STOP}_{\alpha ZM})$$

- $\mathit{STOP}_{\alpha P}$ repräsentiert einen Prozess mit dem Alphabet $\alpha P$, der an keinen Ereignissen aus diesem Alphabet mehr teilnimmt.
- Ein Zeitungsautomat, der eine Ein-Euro-Münze entgegennimmt, eine Zeitung ausgibt und dann den Dienst einstellt:

  $$(\mathit{muenze100} \to (\mathit{zeitung\_ausgeben} \to \mathit{STOP}_{\alpha ZM}))$$

- Der Operator $\to$ ist rechts-assoziativ und erwartet links eine Ereignisklasse und rechts einen Prozess. Entsprechend dürften in diesen Beispielen auch die Klammern wegfallen.

- Eine Uhr, bei der nur das Ticken relevant ist ($\alpha UHR = \{tick\}$), könnte so beschrieben werden:

$$UHR = (tick \rightarrow UHR)$$

- Der Prozess *UHR* ist dann eine Lösung dieser Gleichung.
- So eine Gleichung erlaubt eine iterative Expansion:

$$\begin{aligned} UHR &= (tick \rightarrow UHR) \\ &= (tick \rightarrow tick \rightarrow UHR) \\ &= (tick \rightarrow tick \rightarrow tick \rightarrow UHR) \\ &= \ldots \end{aligned}$$

- Dies gelingt jedoch nur, wenn die rechte Seite der Gleichung mit einer Präfixnotation beginnt. Hingegen würde $X = X$ nichts über $X$ aussagen.
- Eine Prozessbeschreibung, die mit einer Präfixnotation beginnt, wird *geschützt* genannt.

$$\mu X : A.F(X)$$

- Wenn $F(X)$ ein geschütze Prozessbeschreibung ist, die $X$ enthält, und $\alpha X = A$, dann hat $X = F(X)$ eine eindeutige Lösung für das Alphabet $A$.
- Dies kann auch kürzer als $\mu X : A.F(X)$ geschrieben werden, wobei in dieser Notation $X$ eine lokal gebundene Variable ist.
- Beispiel für die tickende Uhr:

  $$\mu X : \{tick\} . (tick \rightarrow X)$$

- Beispiel für einen Zeitungsautomaten, der nur Ein-Euro-Münzen akzeptiert:

  $$\begin{aligned} \mu X : \quad & \{muenze100, zeitung\_ausgeben\} . \\ & (muenze100 \rightarrow zeitung\_ausgeben \rightarrow X) \end{aligned}$$

- Die Angabe des Alphabets $A$ kann entfallen, wenn es offensichtlich ist.

$$(x \to P \mid y \to Q)$$

- Der Operator „|“ ermöglicht eine freie Auswahl. Je nachdem ob zuerst ein Ereignis der Klasse $x$ oder $y$ eintritt, geht es mit $P$ bzw. $Q$ weiter.
- Die Alphabete müssen bei diesem Konstrukt zueinander passen: $\alpha P = \alpha Q$. Und die genannten Ereignisklassen müssen in dem gemeinsamen Alphabet enthalten sein: $\{x, y\} \subseteq \alpha P$.
- Ein Zeitungsautomat, der entweder zwei 50-Cent-Stücke oder eine Ein-Euro-Münze akzeptiert, bevor er eine Zeitung ausgibt:

$$\begin{aligned} \mu X : \quad & \{\mathit{muenze50}, \mathit{muenze100}, \mathit{zeitung\_ausgeben}\} . \\ & (\mathit{muenze100} \to \mathit{zeitung\_ausgeben} \to X \\ \mid \quad & \mathit{muenze50} \to \mathit{muenze50} \to \mathit{zeitung\_ausgeben} \to X) \end{aligned}$$

- Es können beliebig viele Alternativen gegeben werden, aber all die angegebenen Präfixe müssen sich voneinander unterscheiden.

- Es sind auch mehrere wechselseitig rekursive Gleichungen möglich. Dann sollte aber die rechte Seite jeweils geschützt sein und jede linke Seite nur ein einziges Mal vorkommen.
- Ein Fahrzeug $FZ$ hat eine Automatik mit den Einstellungen $d$ (vorwärts), $r$ (rückwärts) und $p$ (Parken) und kann entsprechend vorwärts ($vf$) oder rückwärts ($rf$) fahren:

$$\begin{aligned} FZ &= (d \to FZV \mid r \to FZR) \\ FZV &= (vf \to FZV \mid p \to FZ \mid r \to FZR) \\ FZR &= (rf \to FZR \mid p \to FZ \mid d \to FZV) \end{aligned}$$

# Ablauf eines Prozesses

- Ein Ablauf (*trace*) eines Prozesses ist eine endliche Sequenz von Symbolen aus dem Alphabet, das die Ereignisklassen repräsentiert, an denen ein Prozess bis zu einem Zeitpunkt teilgenommen hat.
- Ein Ablauf wird in winkligen Klammern $\langle \ldots \rangle$ notiert.
- $\langle x, y \rangle$ besteht aus zwei Ereignisklassen: Zuerst $x$, dann $y$. $\langle x \rangle$ besteht nur aus dem Ereignis $x$. $\langle\rangle$ ist die leere Sequenz.
- Beispiel: $\langle d, vf, vf, vf, r, rf, p \rangle$ ist ein möglicher Ablauf von $FZ$.

$$traces(P)$$

- Sei $P$ ein Prozess, dann liefert $traces(P)$ die Menge aller möglichen Abläufe.
- Beispiel: $traces\,((muenze100 \rightarrow STOP)) = \{\langle\rangle, \langle muenze100\rangle\}$
- Beispiel:
  $traces\,(\mu X : (tick \rightarrow X)) = \{\langle\rangle, \langle tick\rangle, \langle tick, tick\rangle, \langle tick, tick, tick\rangle, \ldots\}$

# Operationen für Abläufe

- Seien $s$ und $t$ zwei Abläufe, dann ist $s \frown t$ eine zusammengesetzte Ablaufsequenz von $s$ und $t$. Beispiel: $\langle a, b \rangle \frown \langle c, a \rangle = \langle a, b, c, a \rangle$
- Die Relation $s \leq t$ gilt genau dann, wenn $\exists u : s \frown u = t$
- $n$ aufeinanderfolgende Kopien einer Sequenz sind so definiert:

$$\begin{aligned} t^0 &= \langle\rangle \\ t^{n+1} &= t \frown t^n \end{aligned}$$

- Sei $A$ ein Alphabet und $t$ ein Ablauf, dann ist $t \upharpoonright A$ der Ablauf, bei dem alle Elemente $a \notin A$ aus $t$ entfernt worden sind. Beispiel: $\langle a, b, c, a, c \rangle \upharpoonright \{a, b\} = \langle a, b, a \rangle$

- $STOP_A$ mit $\alpha STOP_A = A$ ist ein Prozess, der an keinem Ereignis teilnimmt.
- Entsprechend gilt: $traces(STOP_A) = \{\langle\rangle\}$.
- $STOP$ repräsentiert den Deadlock.
- $RUN_A$ mit $\alpha RUN_A = A$ ist ein Prozess, der an allen Ereignissen des Alphabets $A$ in beliebiger Reihenfolge teilnimmt.
- Entsprechend gilt: $traces(RUN_A) = A^*$, wobei $A^*$ die Menge aller endlichen Sequenzen aus dem Alphabet $A$ ist.
- Beispiel: $\mu X : \{tick\}.\,(tick \rightarrow X) = RUN_{\{tick\}}$
- $SKIP_A$ mit $\alpha SKIP_A = A \cup \{\checkmark\}$ verhält sich analog zu $STOP_A$ ist aber nicht als Deadlock zu verstehen, sondern als erfolgreiches Ende, das durch das Ereignis $\checkmark$ repräsentiert wird. Das Ende-Ereignis darf ansonsten nicht verwendet werden.

# $P \parallel Q$

- $P$ und $Q$ sind zwei Prozesse, bei denen wir zunächst aus Gründen der Einfachheit ausgehen, dass $\alpha P = \alpha Q = A$.
- Dann ergibt $P \parallel Q$ einen Prozess, bei dem Ereignisse aus dem gemeinsamen Alphabet $A$ genau dann stattfinden, wenn sie simultan bei den Prozessen $P$ und $Q$ stattfinden.
- Beide Prozesse laufen parallel, aber die Ereignisse finden synchron statt.
- Beispiel:

$$
\begin{aligned}
ZA &= (muenze50 \rightarrow zeitung\_ausgeben \rightarrow ZA \mid \\
&\quad\ muenze100 \rightarrow zeitung\_ausgeben \rightarrow ZA) \\
K &= (muenze20 \rightarrow zeitung\_ausgeben \rightarrow K \mid \\
&\quad\ muenze50 \rightarrow zeitung\_ausgeben \rightarrow K) \\
ZA \parallel K &= \mu X : (muenze50 \rightarrow zeitung\_ausgeben \rightarrow X)
\end{aligned}
$$

Unter der Voraussetzung, dass $\alpha P = \alpha Q$ gilt, gelten folgende Gesetze:

- Der Operator „||“ ist kommutativ: $P \parallel Q = Q \parallel P$
- Es gilt das Assoziativgesetz: $(P \parallel Q) \parallel R = P \parallel (Q \parallel R)$
- Ein Deadlock ist viral: $P \parallel STOP = STOP$
- Eine Kombination mit $RUN$ hat keinen Effekt: $P \parallel RUN = P$
- Wenn beide Prozesse kein gemeinsames Folge-Ereignis finden $(a \neq b)$, haben wir einen Deadlock: $(a \rightarrow P) \parallel (b \rightarrow Q) = STOP$
- $traces(P \parallel Q) = traces(P) \cap traces(Q)$

$P \parallel Q$ ist auch zulässig, falls $\alpha P \neq \alpha Q$. Dann gilt:

- $\alpha(P \parallel Q) = \alpha P \cup \alpha Q$
- Ereignisse aus $\alpha P \setminus \alpha Q$ werden nur von $P$ verfolgt und Ereignisse aus $\alpha Q \setminus \alpha P$ nur von $Q$.
- Nur Ereignisse aus $\alpha P \cap \alpha Q$ betreffen beide Prozesse.
- Es gelten weiterhin die Gesetze der Kommutativität und der Assoziativität.
-

$$\begin{aligned} \mathit{traces}(P \parallel Q) \;=\; \{t \mid\; & (t \upharpoonright \alpha P) \in \mathit{traces}(P) \;\wedge \\ & (t \upharpoonright \alpha Q) \in \mathit{traces}(Q) \;\wedge \\ & t \in (\alpha P \cup \alpha Q)^*\} \end{aligned}$$

Es gelte

$$\alpha ZA = \alpha K = \{muenze20, muenze50, muenze100, zeitung\_ausgeben\}$$

und es sei gegeben:

$$\begin{aligned} ZA &= (muenze50 \rightarrow zeitung\_ausgeben \rightarrow ZA \mid \\ &\quad\ muenze100 \rightarrow zeitung\_ausgeben \rightarrow ZA) \\ K &= (muenze20 \rightarrow zeitung\_ausgeben \rightarrow K \mid \\ &\quad\ muenze50 \rightarrow zeitung\_ausgeben \rightarrow K) \end{aligned}$$

Dann gilt: $traces(ZA \parallel K) = \{t | t \leq \langle muenze50, zeitung\_ausgeben \rangle^n\}$

Wenn jedoch *ZA* und *K* unterschiedliche Alphabete haben, dann werden Ereignisse wie *muenze20* oder *muenze100* nur von einem der beiden parallel laufenden Prozesse wahrgenommen:

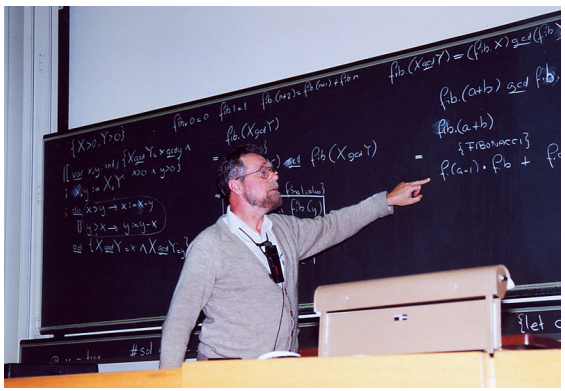
$$\begin{aligned} \alpha ZA &= \{muenze50, muenze100, zeitung\_ausgeben\} \\ \alpha K &= \{muenze20, muenze50, zeitung\_ausgeben\} \end{aligned}$$

Wenn dann eine Ablaufsequenz mit *muenze20* beginnt, dann muss zwingend *muenze100* folgen, damit beide Prozesse, *ZA* und *K* das Ereignis *zeitung_ausgeben* gemeinsam wahrnehmen können.
Entsprechend gilt

$$\begin{aligned} ZA \,||\, K = \; & \mu X : (muenze50 \rightarrow zeitung\_ausgeben \rightarrow X \mid \\ & muenze20 \rightarrow muenze100 \rightarrow zeitung\_ausgeben \rightarrow X \mid \\ & muenze100 \rightarrow muenze20 \rightarrow zeitung\_ausgeben \rightarrow X) \end{aligned}$$

# Beispiel: Philosophenproblem



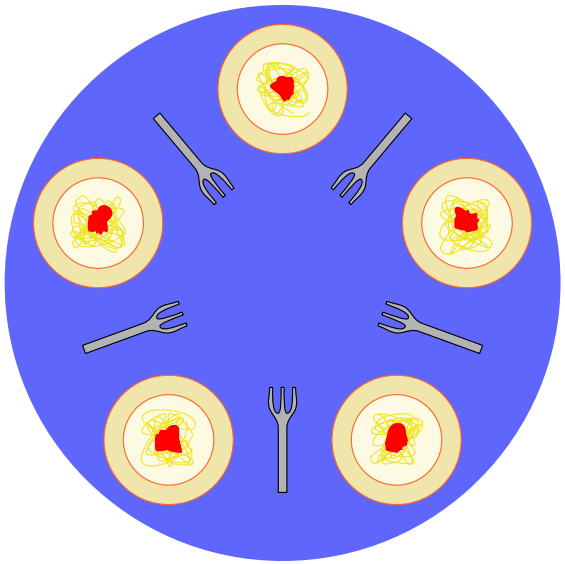


Edsger Dijkstra 1994

Edsger Dijkstra stellte in dem Aufsatz *Hierarchical ordering of sequential processes* in der *Acta Informatica* 1971 Semaphore als Mittel der Synchronisierung von Prozessen vor und demonstrierte diese Technik an dem von ihm aufgestellten Philosophenproblem, das die Gefahren eines Deadlocks und des Aushungerns sehr schön demonstrierte.

# Beispiel: Philosophenproblem





Gegeben seien fünf Philosophen $P_0, \ldots, P_4$, die an einem Tisch sitzen und die auf ihrem Teller befindlichen Spaghetti verzehren möchten. Zum Essen werden jeweils zwei Gabeln benötigt, die bereits neben den Tellern liegen. Jedoch stehen insgesamt nur fünf Gabeln $G_0, \ldots, G_4$ zur Verfügung.

Entsprechend kann die zwischen den Philosophen $P_i$ und $P_{i+1 \bmod 5}$ liegende Gabel $G_i$ nur von einem der beiden genommen werden. Und der Philosoph $P_i$ kann erst dann essen, wenn es ihm gelungen ist, die Gabeln $G_{i-1 \bmod 5}$ und $G_i$ zu ergattern.

# Beispiel: Philosophenproblem

Ein erster Lösungsansatz:

$$
\begin{aligned}
DP &= P_0 \parallel \ldots \parallel P_4 \parallel G_0 \parallel \ldots \parallel G_4 \\
G_i &= (G_i\_genommen\_von\_P_i \rightarrow \\
&\quad G_i\_hingelegt\_von\_P_i \rightarrow G_i \mid \\
&\quad G_i\_genommen\_von\_P_{i+1 \bmod 5} \rightarrow \\
&\quad G_i\_hingelegt\_von\_P_{i+1 \bmod 5} \rightarrow G_i) \\
P_i &= (G_{i-1 \bmod 5}\_genommen\_von\_P_i \rightarrow \\
&\quad G_i\_genommen\_von\_P_i \rightarrow \\
&\quad essen_i \rightarrow \\
&\quad G_{i-1 \bmod 5}\_hingelegt\_von\_P_i \rightarrow \\
&\quad G_i\_hingelegt\_von\_P_i \rightarrow P_i)
\end{aligned}
$$

$\alpha P_i$ und $\alpha G_i$ seien hier jeweils durch die in den jeweiligen Definitionen explizit genannten Ereignisse gegeben.

# Beispiel: Philosophenproblem

Wenn alle Philosophen gleichzeitig loslegen mit dem Aufnehmen der jeweils linken Gabel, haben wir einen Deadlock:

$$t = \langle G_4\_genommen\_von\_P_0, G_0\_genommen\_von\_P_1, G_1\_genommen\_von\_P_2, G_2\_genommen\_von\_P_3, G_3\_genommen\_von\_P_4 \rangle \in traces(DP)$$

$$\nexists\, u \in traces(DP) : u > t$$

Wenn $P_0$ und $P_2$ immer schneller als die anderen sind, werden $P_1, P_3$ und $P_4$ nie zum Essen kommen:

$$\langle \{ G_4\_genommen\_von\_P_0, G_1\_genommen\_von\_P_2, G_0\_genommen\_von\_P_0, G_2\_genommen\_von\_P_2, essen\_0, essen\_2, G_4\_hingelegt\_von\_P_0, G_1\_hingelegt\_von\_P_2, G_0\_hingelegt\_von\_P_0, G_2\_hingelegt\_von\_P_2 \}^n \rangle \subset traces(DP)$$

# Beispiel: Philosophenproblem



Das Deadlock-Problem kann mit einer Ergänzung von Carel S. Scholten vermieden werden. Diese Ergänzung sieht vor, dass sich die Philosophen nur mit Hilfe eines Dieners zu Tisch setzen und wieder aufstehen dürfen und dass dieser Diener die Anweisung erhält, nur maximal vier der fünf Philosophen sich gleichzeitig setzen zu lassen:

$$
\begin{aligned}
DP &= P_0 \parallel \ldots \parallel P_4 \parallel G_0 \parallel \ldots \parallel G_4 \parallel D_0 \\
P_i &= (\mathit{hinsetzen}_i \rightarrow G_{i-1 \bmod 5}\_\mathit{genommen\_von\_}P_i \rightarrow \\
&\quad\; G_i\_\mathit{genommen\_von\_}P_i \rightarrow \mathit{essen}_i \rightarrow \\
&\quad\; G_{i-1 \bmod 5}\_\mathit{hingelegt\_von\_}P_i \rightarrow \\
&\quad\; G_i\_\mathit{hingelegt\_von\_}P_i \rightarrow \mathit{aufstehen}_i \rightarrow P_i) \\
D_0 &= x : H \rightarrow D_1 \\
D_i &= (x : H \rightarrow D_{i+1} \mid y : A \rightarrow D_{i-1}) \text{ für } i \in \{1,2,3\} \\
D_4 &= y : A \rightarrow D_3
\end{aligned}
$$

Dabei seien $H = \bigcup_{i=0}^{4} \{\mathit{hinsetzen}_i\}$ und $A = \bigcup_{i=0}^{4} \{\mathit{aufstehen}_i\}$

$$P \sqcap Q$$

- $P$ und $Q$ mit $\alpha P = \alpha Q = \alpha P \sqcap Q$.
- Dann ergibt $P \sqcap Q$ einen Prozess, der sich nichtdeterministisch für einen der beiden Prozesse $P$ oder $Q$ entscheidet.
- Diese Notation ist kein Hinweis auf die Implementierung. Es bleibt vollkommen offen, ob sich $P \sqcap Q$ immer für $P$, immer für $Q$ oder nach unbekannten Kriterien oder Zufällen für $P$ oder $Q$ entscheidet.
- Beispiel eines Automaten, der sich nichtdeterministisch dafür entscheidet, ob er Tee oder Kaffee ausliefert:

$$\begin{aligned} GA &= (muenze \rightarrow tee \rightarrow GA) \sqcap (kaffee \rightarrow GA) \\ AT &= (muenze \rightarrow tee \rightarrow AT \mid kaffee \rightarrow AT) \\ KT &= (muenze \rightarrow kaffee \rightarrow KT) \end{aligned}$$

  Hier ist $AT$ flexibel genug, damit bei $GA \parallel AT$ ein Deadlock ausgeschlossen ist. Bei $GA \parallel KT$ besteht die Gefahr eines Deadlocks, wenn sich $GA$ nichtdeterministisch für die Ausgabe eines Tees entscheidet.

# $P \,[\!]\, Q$

- $P$ und $Q$ mit $\alpha P = \alpha Q = \alpha P \,[\!]\, Q$.
- Anders als bei $P \sqcap Q$ hat die Umgebung eine Einflussmöglichkeit. Wenn das nächste Ereignis aus $\alpha P$ nur $P$ oder nur $Q$ dies akzeptieren kann, dann fällt die Entscheidung für den entsprechenden Zweig.
- Wenn jedoch das nächste Ereignis sowohl von $P$ als auch $Q$ akzeptiert werden kann, dann fällt die Entscheidung genauso nichtdeterministisch wie bei $P \sqcap Q$.
- $c \to P \,[\!]\, d \to Q \;=\; (c \to P \mid d \to Q)$, falls $c \neq d$.
- $c \to P \,[\!]\, d \to Q \;=\; (c \to P) \sqcap (d \to Q)$, falls $c = d$.
- $\mathit{traces}(P \,[\!]\, Q) = \mathit{traces}(P \sqcap Q) = \mathit{traces}(P) \cup \mathit{traces}(Q)$

$$P \,|||\, Q$$

- $P$ und $Q$ mit $\alpha P = \alpha Q = \alpha P \,|||\, Q$.
- Sowohl $P$ als auch $Q$ werden parallel verfolgt, aber jedes Ereignis kann nur von einem der beiden Prozesse wahrgenommen werden. Wenn nur einer der beiden Prozesse das Ereignis akzeptieren kann, dann wird es von diesem akzeptiert. Andernfalls, wenn beide ein Ereignis akzeptieren können, dann fällt die Entscheidung nichtdeterministisch.

# Beispiel: Philosophenproblem

Damit vereinfacht sich die Umsetzung des Philosophenproblems:

$$
\begin{aligned}
DP &= (P_0 \,|||\, \ldots \,|||\, P_4) \,||\, (G_0 \,|||\, \ldots \,|||\, G_4) \\
G_i &= (G_i\_genommen \rightarrow G_i\_hingelegt \rightarrow G_i) \\
P_i &= (G_{i-1 \bmod 5}\_genommen \rightarrow \\
&\quad G_i\_genommen \rightarrow \\
&\quad essen \rightarrow \\
&\quad G_{i-1 \bmod 5}\_hingelegt \rightarrow \\
&\quad G_i\_hingelegt \rightarrow P_i)
\end{aligned}
$$

# Beispiel: Philosophenproblem

Ebenso vereinfacht sich die Fassung mit dem Diener:

$$
\begin{aligned}
DP &= (P_0 \,|||\, \ldots \,|||\, P_4) \,||\, (G_0 \,|||\, \ldots \,|||\, G_4) \,||\, (D \,|||\, D \,|||\, D \,|||\, D) \\
G_i &= (G_i\_genommen \rightarrow G_i\_hingelegt \rightarrow G_i) \\
P_i &= (hinsetzen \rightarrow G_{i-1 \bmod 5}\_genommen \rightarrow G_i\_genommen \rightarrow \\
&\quad\; essen \rightarrow \\
&\quad\; G_{i-1 \bmod 5}\_hingelegt \rightarrow G_i\_hingelegt \rightarrow aufstehen \rightarrow P_i) \\
D &= (hinsetzen \rightarrow aufstehen \rightarrow D)
\end{aligned}
$$

$$c.v$$

- Kommunikation wird mit Ereignissen der Form $c.v$ modelliert. Hierbei repräsentiert $c$ den Kommunikationskanal und $v$ den Inhalt der Nachricht über den Kanal.
- Zu jedem Kommunikationskanal $c$ und jedem Prozess $P$ gibt es das zugehörige Alphabet, das $P$ über den Kanal $c$ kommunizieren kann: $\alpha c(P) = \{v | c.v \in \alpha P\}$
- Ferner lässt sich definieren: $channel(c.v) = c, message(c.v) = v$

$$c!v \rightarrow P$$

- Sei $c$ ein Kommunikationskanal, $c.v \in \alpha c(P)$, dann kann das Einlesen mit dem Operator „!“ explizit notiert werden: $(c!v \rightarrow P) = (c.v \rightarrow P)$
- Beim Einlesen wird dann mit einer Variablen $x$ gearbeitet, die an die Übertragungsereignis gebunden wird:
  $(c?x \rightarrow P(x)) = (y : \{y | channel(y) = c\} \rightarrow P(message(y)))$
- Ein Prozess *CopyBit* mit zwei Kommunikationskanälen *in* und *out*, der die Bits $\alpha in(CopyBit) = \alpha out(CopyBit) = \{0, 1\}$ überträgt:
  $CopyBit = (in?x \rightarrow out!x \rightarrow CopyBit)$
- Normalerweise gilt für zwei Prozesse $P$ und $Q$, die parallel über einen Kanal kommunizieren $(P \parallel Q)$, dass $\alpha c(P) = \alpha c(Q)$.

- Ereignisse finden in CSP immer synchron statt – entsprechend erfolgt auch die Kommunikation synchron.
- Durch die Einführung von zwischenliegenden Puffern lässt sich eine asynchrone Kommunikation modellieren.
- Gegeben seien $P(c)$ und $Q(c)$, die über einen Kommunikationskanal $c$ miteinander verbunden werden können: $P(c) \parallel Q(c)$.
- Wenn jedoch die synchrone Kopplung zwischen $P(c)$ und $Q(c)$ nicht gewünscht wird, lässt sich dies durch das Einfügen eines Puffers vermeiden:

$$Puffer(in, out) = \mu X : (in?x \rightarrow out!x \rightarrow X)$$

Wir haben dann zwei Kommunikationskanäle $c_1$ (zwischen $P$ und dem *Puffer*) und $c_2$ (zwischen dem *Puffer* und $Q$):

$$P(c1) \parallel Puffer(c1, c2) \parallel Q(c2)$$

- Wenn Prozesse miteinander verknüpft werden, wird eine Abstrahierung möglich, die sich nur auf die Außenansicht beschränkt und die Betrachtung der Ereignisse weglässt, die nur intern stattfindet.
- Beispiel:
  $Puffer_3(in, out) = Puffer(in, c1) \parallel Puffer(c1, c2) \parallel Puffer(c2, out)$ ist ein FIFO-Puffer mit der Kapazität 3, bei dem in der weiteren Betrachtung die internen Kanäle $c1$ und $c2$ nicht weiter interessant sind und wir uns auf die äußeren Kanäle *in* und *out* beschränken können.
- Eine Abstraktion kann dann zu einer Divergenz führen, wenn es die Möglichkeit zu einer ungebundenen internen Sequenz von Ereignissen gibt.
- Beispiel: Folgende Pipeline besteht aus den beiden Prozessen $P$ und $Q$, die auch untereinander die Nachricht *Hallo* kommunizieren:

$$
\begin{aligned}
DivPuffer(in, out) &= P(in, c) \parallel Q(c, out) \\
P(in, out) &= \mu X : (in?x \rightarrow out!x \rightarrow X \mid out!Hallo \rightarrow X) \\
Q(in, out) &= \mu X : (in?x \rightarrow out!x \rightarrow X \mid in?Hallo \rightarrow X)
\end{aligned}
$$