Parallel Programming Concepts

Theory of Concurrency - Multicomputer

Peter Tröger
Von Neumann Model

- Processor executes a sequence of instructions, which specify
  - Arithmetic operation
  - Memory to be read / written
  - Address of next instruction

- Software layering tackles complexity of instruction stream

- Parallelism adds coordination problem between multiple instruction streams being executed

- Pipelining
- Super-scalar
- VLIW
- Branch prediction
- ...

Process

Processor

Memory
Terminology

**Concurrency**

- Supported to have two or more actions *in progress* at the same time
- Classical operating system responsibility (resource sharing for better utilization of CPU, memory, network, ...)
- Demands **scheduling** and **synchronization**

**Parallelism**

- Supported to have two or more actions executing *simultaneously*
- Demands **parallel hardware, concurrency support**, (and **communication**)
- Programming model relates to chosen hardware / communication approach
- Examples: Windows 3.1, threads, signal handlers, shared memory
History

• 1963: Co-Routines concept by Melvin Conway
  • Foundation for message-based concurrency concepts

• Late 1970’s
  • Parallel computing moved from shared memory towards multicomputers
  • Dijkstra / Hoare / Hansen worked on different according abstractions

• 1975, Concept of „recursive non-deterministic processes“ by Dijkstra
  • Generator concept, foundation for Hoare’s work on Communicating Sequential Processes (CSP)

• 1978, Distributed Processes: A Concurrent Programming Concept, B. Hansen
  • Synchronized procedure called by one process and executed by another
  • Foundation for RPC variations in Ada and other languages
Co-Routines


- Routines can suspend (yield) and resume in their execution

- Co-routines that always yield new results are also called **generators**

- Good for concurrent, not for parallel programming

- Foundation for theoretical and practical message passing concepts

- Broad language support today

```plaintext
var q := new queue

coroutine produce
    loop
        while q is not full
            create some new items
            add the items to q
        yield to consume

coroutine consume
    loop
        while q is not empty
            remove some items from q
            use the items
        yield to produce
```
Co-Routines

- Explicit language primitive to indicate transfer of control flow - **resume** primitive
- Example: Chess game - classical approach demands master procedure
- **detach** primitive: Return to control point that initially activated the co-routine
- Co-routines allow caller / callee model to be expressed in code

```plaintext
coroutine PLAYER1;
  initialize local variables
  detach
  while TRUE do
    make a move
    if game won then
      print message
      detach
    else
      resume(PLAYER2)

coroutine PLAYER2;
  initialize local variables
  detach
  while TRUE do
    make a move
    if game won then
      print message
      detach
    else
      resume(PLAYER1)
```
Communicating Sequential Processes

• Developed by Tony Hoare at University of Oxford from 1977

• Formal process algebra to describe concurrent systems

• Book: T. Hoare, Communicating Sequential Processes, 1985

• Basic idea
  
  • Computer **systems** act and interact with the environment continuously  
    • Decomposition in **subsystems** (**processes**) which operate concurrently  
    • **Interact** with other processes - subsystems or the environment  
      • Modular approach

• Based on mathematical theory, described with algebraic laws

• Direct mapping to Occam programming language
CSP: Processes

• Behavior of real-world objects can be described through their interaction with other objects, leaving out internal implementation details

• Interface of a process is described as set of atomic events

• Event examples for an ATM:
  • card – insertion of a credit card in an ATM card slot
  • money – extraction of money from the ATM dispenser

• Alphabet - set of relevant (!) events for the description of an object
  • Event may never happen in the interaction
  • Interaction is restricted to this set of events
  • \( \alpha_{ATM} = \{\text{card, money}\} \)

• A CSP process is the behavior of an object, described with its alphabet
CSP: Processes

- **Event** is an atomic action without duration
  - Time is expressed with start/stop events, can overlap
  - Timing of events is not relevant for logical correctness, but ordering
  - Makes reasoning independent of execution speed and performance

- No concept of simultaneous events
  - May be represented as single event, if synchronization is modeled

- **STOP** \_A
  - Process with alphabet A which never engages in any of the events of A
  - Expresses a non-working part of the system
CSP: Process Description through Prefix Notation

- \((x \rightarrow P)\)  "x then P"
  - \(x\): event, \(P\): process
  - Behavioral description of an object which first engages in \(x\) and than behaves as described with \(P\)
  - Prefix expression itself is a process (== behavior), chainable approach
  - \(\alpha(x \rightarrow P) = \alpha P\) - Processes must have the same alphabet
  - Example 1:
    \((\text{card} \rightarrow \text{STOP}_{\alpha\text{ATM}})\)
    "ATM which takes a credit card before breaking"
  - Quiz:
    "ATM which serves one customer and breaks while serving the second customer"  - \(\alpha\text{ATM}_Q=\{\text{card, money}\}\)
CSP: Recursion

- Prefix notation may lead to long chains of repetitive behavior for the complete lifetime of the object (until `STOP`)
  - Solution: Self-referential recursive definition for the object

- Example: An everlasting clock object
  \[ \alpha_{\text{CLOCK}} = \{\text{tick}\} \]
  \[ \text{CLOCK} = (\text{tick} \rightarrow \text{CLOCK}) \]

- Enables description of an object with one single stream of behavior through prefixing and recursion
CSP Process Description - Choice

- Object behavior may be influenced by the environment
  - Support for multiple ‘behavior streams’ triggered by the environment

- Externally-triggered choice between two or more events, leads to different subsequent behavior (== processes), forms a process by itself
  \[(x \rightarrow P | y \rightarrow Q)\]

- Example: Vending machine offers choice of slots for 1€ coin or 2€ coin
  \[VM = (\text{in1eur} \rightarrow (\text{cookie} \rightarrow VM) | \text{in2eur} \rightarrow (\text{cake} \rightarrow VM) | \text{crown} \rightarrow \text{STOP})\]

- | is an operator on prefix expression, not on the processes itself
Process Description: Pictures

- Single processes as circles, events as arrows
- Pictures may lead to problems - difficult to express equality, hard with large or infinite number of behaviors

$$\text{VM} = (\text{in1eur} \rightarrow (\text{cookie} \rightarrow \text{VM}) |$$
$$\text{in2eur} \rightarrow (\text{cake} \rightarrow \text{VM}) |$$
$$\text{crowncap} \rightarrow \text{STOP})$$
Concurrency in CSP

- Process = Description of possible behavior
- Set of occurring events depends on the environment, which may also be described as a process
- Allows to investigate a complete system, were the description is again a process
- Formal modelling of interacting processes
  - Formulate events that trigger simultaneous participation of multiple processes
- **Parallel combination**: Process which describes a system composed of the processes P and Q:

  \[ P \parallel Q \quad \alpha(P \parallel Q) = \alpha P \cup \alpha Q \]

- **Interleaving**: Parallel activity with different events
Graphical Representation

\[( \text{P} \parallel \text{Q} )\]
Communication in CSP

• Special class of event: communication

  • Modeled as uni-directional channel, only between two processes

  • Channel name is a member of the alphabets of both processes

  • Described by the events $c.v$ which are part of the processes alphabet
    $c$: name of a channel on which communication takes place
    $v$: value of the message being passed

• Set of all messages which $P$ can communicate on channel $c$:
  $\alpha_c(P) = \{v | c.v \in \alpha_P\}$

• $\text{channel}(c.v) = c$, $\text{message}(c.v) = v$

• Input choice: $(c?x \rightarrow P(x) | d?y \rightarrow Q(y))$
Communication (contd.)

• Process which first outputs v on the channel c and then behaves like P:
  \[(c!v \rightarrow P) = (c.v \rightarrow P)\]

• Process which is initially prepared to input any value x from the channel c and then behave like P(x):
  \[(c?x \rightarrow P(x)) = (y: \{y | \text{channel}(y) = c\} \rightarrow P(\text{message}(y)))\]
Communication (contd.)

- Channel approach assumes **rendezvous behavior**
  - Sender and receiver block on the channel operation until the message was transmitted
  - Meanwhile common concept in messaging-based concurrency approaches
- When two concurrent processes communicate with each other only over a single channel, they cannot deadlock (see book)
- Network of non-stopping processes which is free of cycles cannot deadlock
  - Acyclic graph can be decomposed into subgraphs connected only by a single arrow
- Pipes: Processes with only one input and one output channel
- Join of two pipes P and Q: $P >> Q$
The Dining Philosophers (E.W.Dijkstra)

- Five philosophers work in a college, each philosopher has a room for thinking.

- Common dining room, furnished with a circular table, surrounded by five labeled chairs.

- In the center stood a large bowl of spaghetti, which was constantly replenished.

- When a philosopher gets hungry:
  - Sits on his chair.
  - Picks up his own fork on the left and plunges it in the spaghetti, then picks up the right fork.
  - When finished he put down both forks and gets up.
  - May wait for the availability of the second fork.
Mathematical Model

• Philosophers: PHIL₀ ... PHIL₄

• αPHILᵢ = { i.sits down, i.gets up,
            i.picks up fork.ᵢ, i.picks up fork.(i⊕1),
            i.puts down fork.ᵢ, i.puts down fork.(i⊕1) }

• ⊕: Addition modulo 5 == i⊕1 is the right-hand neighbor of PHILᵢ

• Alphabets of the philosophers are mutually disjoint, no interaction between them

• αFORKᵢ = { i.picks up fork.ᵢ,
              (i⊕1).picks up fork.ᵢ,
              i.puts down fork.ᵢ,
              (i⊕1).puts down fork.ᵢ }
Behavior of the Philosophers

- **PHIL\textsubscript{i} = ( i.sits down ->
  i.picks up fork.i ->
  i.picks up fork.(i\oplus 1) ->
  i.puts down fork.i ->
  i.puts down fork.(i\oplus 1) ->
  i.gets up -> PHIL\textsubscript{i} )**

- **FORK\textsubscript{i} = ( i.picks up fork.i ->
  i.puts down fork.i -> FORK\textsubscript{i} |
  (i\oplus 1).picks up fork.i ->
  (i\oplus 1).puts down fork.i -> FORK\textsubscript{i} )**

- **PHILOS = (PHIL\textsubscript{0} | | PHIL\textsubscript{1} | | PHIL\textsubscript{2} | | PHIL\textsubscript{3} | | PHIL\textsubscript{4} )**

- **FORKS = (FORK\textsubscript{0} | | FORK\textsubscript{1} | | FORK\textsubscript{2} | | FORK\textsubscript{3} | | FORK\textsubscript{4} )**

- **COLLEGE = (PHILOS | | FORKS)**

We leave out the proof here ;-) ...
What's the Deal?

- Any possible system can be modeled through event chains
  - Enables mathematical proofs for deadlock freedom, based on the basic assumptions of the formalism (e.g. channel assumption)
- Some tools available (look at the CSP archive)
- CSP was the formal base for the Occam language
  - Language constructs follow the formalism, to keep proven properties
  - Mathematical reasoning about behavior of written code
- Still active research topics, channel concept adopted at several places
  - CSP channel implementation for Java, MPI design
  - Other formalism, e.g. Task / Channel model
Occam Example

PROC producer (CHAN INT out!)
  INT x:
  SEQ
    x := 0
    WHILE TRUE
      SEQ
        out ! x
        x := x + 1
  :

PROC consumer (CHAN INT in?)
  WHILE TRUE
    INT v:
    SEQ
      in ? v
      .. do something with `v`
    :

PROC network ()
  CHAN INT c:
  PAR
    producer (c!)
    consumer (c?)
  :
Task-Channel Model [Foster]

- **Computational model** for multi-computer case
- Parallel computation consists of one or more tasks
  - Tasks execute concurrently
  - Number of tasks can vary during execution
  - Task encapsulates **sequential program** with local memory
  - A task has **in-ports** and **outports** as interface to the environment
- **Basic actions**: read / write local memory, send message on outport, receive message on in-port, create new task, terminate
Task-Channel Model [Foster]

• Outport / in-port pairs are connect by message queues called **channels**
  • Channels can be created and deleted
  • Channels can be referenced as **ports**, which can be part of a message
  • **Send** operation is asynchronous
  • **Receive** operation is synchronous
  • Messages in a channel stay in order
• Tasks are **mapped** to physical processors
  • Multiple tasks can be mapped to one processor
• Data locality is explicit part of the model
• Channels can model **control** and **data dependencies**
Task-Channel Model [Foster]

• Effects from channel-only interaction model
  • Performance optimization does not influence semantics
    • Example: Shared-memory channels for multiple tasks on one machine
  • Task mapping does not influence semantics
    • Align number of tasks to problem, not to execution environment
    • Improves scalability of implementation
  • Modular design with well-defined interfaces
  • Determinism made easy
    • Verify that each channel has a single sender and receiver
Task-Channel Model [Foster]

- Model results in some algorithmic style
  - Task graph algorithms, data-parallel algorithms, master-slave algorithms
- Theoretical performance assessment
  - Execution time: Period of time where at least one task is active
  - Number of communications / messages per task
- Rules of thumb
  - Communication operations should be balanced between tasks
  - Each task should only communicate with a small group of neighbors
  - Task should perform computations concurrently (task parallelism)
  - Task should perform communication concurrently
Actor Model

• **Carl Hewitt, Peter Bishop and Richard Steiger. A Universal Modular Actor Formalism for Artificial Intelligence IJCAI 1973.**

• Another mathematical model for concurrent computation

• No global system state concept (relationship to physics)

• Actor as computation primitive, which can make local decisions, concurrently creates more actors, or concurrently sends / receives messages

• Asynchronous one-way messaging with changing topology (CSP communication graph is fixed), no order guarantees

• CSP relies on hierarchy of combined parallel processes, while actors rely only on message passing paradigm only

• Recipient is identified by *mailing address*, can be part of a message

• „Everything is an actor“
Actor Model

• Influenced the development of the Pi-Calculus

• Serves as theoretical base to reason about concurrency, and as underlying theory for some programming languages (Erlang, Scala)

• Influences by Lisp, Simula, and Smalltalk

• Behavior as mathematical function - describes activity on message processing
Other Formalisms

- Lambda calculus by Alonzo Church (1930s)
  - Concept of procedural abstraction, originally via variable substitution
  - Functions as first-class citizen
  - Inspiration for concurrency through functional programming languages
- Petri Nets by Carl Adam Petri (since 1960s)
  - Mathematical model for concurrent systems
  - Directed bipartite graph with places and transitions
  - Huge vibrant research community
- Process algebra, trace theory, ...