## Table of Contents

- Introduction and Motivation
- Theoretical Foundations
- **Distributed Programming Languages**
- Distributed Operating Systems
- Distributed Communication
- Distributed Data Management
- Reliability
- Applications
- Conclusions
- Appendix
Three Issues

- Use of multiple PEs
- Cooperation among the PEs
- Potential for survival to partial failure
Control Mechanisms

<table>
<thead>
<tr>
<th>Statement type \ Control type</th>
<th>Sequential control</th>
<th>Parallel Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential/parallel statement</td>
<td><strong>Begin</strong> ( S_1, S_2 ) <strong>end</strong></td>
<td><strong>Parbegin</strong> ( S_1, S_2 ) <strong>Parend</strong> <strong>Fork/join</strong></td>
</tr>
<tr>
<td>Alternative statement</td>
<td><strong>goto, case if</strong> ( C ) <strong>then</strong> ( S_1 ) <strong>else</strong> ( S_2 )</td>
<td>Guarded commands: ( G \rightarrow C )</td>
</tr>
<tr>
<td>Repetitive statement</td>
<td><strong>for ... do</strong></td>
<td><strong>doall, for all</strong></td>
</tr>
<tr>
<td>Subprogram</td>
<td><strong>procedure</strong> Subroutine</td>
<td><strong>procedure subroutine</strong></td>
</tr>
</tbody>
</table>

Four basic sequential control mechanisms with their parallel counterparts.
Focus 6: Expressing Parallelism

**parbegin/parend** statement

\[ S_1;[[S_2;[S_3||S_4];S_5;S_6]||S_7];S_8 \]

A precedence graph of eight statements.
Focus 6 (Cont’d.)

**fork/join** statement

\[
\begin{align*}
& s_1; \\
& c_1 := 2; \\
& \textbf{fork } L1; \\
& s_2; \\
& c_2 := 2; \\
& \textbf{fork } L2; \\
& s_4; \\
& \textbf{go to } L3; \\
& L1: s_3; \\
& L2: \textbf{join } c_1; \\
& s_5; \\
& L3: \textbf{join } c_2; \\
& s_6;
\end{align*}
\]

A precedence graph.
Dijkstra's Semaphore + Parbegin/Parend

$S(i)$: A sequence of $P$ operations; $S_i$; a sequence of $V$ operations

$s$: a binary semaphore initialized to 0.

$S(1): S_1; V(s_{12}); V(s_{13})$
$S(2): P(s_{12}); S_2; V(s_{24}); V(s_{25})$
$S(3): P(s_{13}); S_3; V(s_{35})$
$S(4): P(s_{24}); S_4; V(s_{46})$
$S(5): P(s_{25}); P(s_{35}); S_5; V(s_{56})$
$S(6): P(s_{46}); P(s_{56}); S_6$
Focus 7: Concurrent Execution

- $R(S_i)$, the **read set** for $S_i$, is the set of all variables whose values are referenced in $S_i$.
- $W(S_i)$, the **write set** for $S_i$, is the set of all variables whose values are changed in $S_i$.
- **Bernstein conditions**:  
  - $R(S_1) \cap W(S_2) = \phi$
  - $W(S_1) \cap R(S_2) = \phi$
  - $W(S_1) \cap W(S_2) = \phi$
Example 7

\( S_1 : a := x + y, \)
\( S_2 : b := x \times z, \)
\( S_3 : c := y - 1, \) and
\( S_4 : x := y + z. \)

\( S_1||S_2, S_1||S_3, S_2||S_3, \) and \( S_3||S_4. \)

Then, \( S_1||S_2||S_3 \) forms a largest complete subgraph.
Example 7 (Cont’d.)

A graph model for Bernstein's conditions.
Alternative Statement

*Alternative* statement in DCDL (CSP like distributed control description language)

\[
\left[ G_1 \rightarrow C_1 \square G_2 \rightarrow C_2 \square \ldots \square G_n \rightarrow C_n \right].
\]
Example 8

Calculate $m = \max \{x, y\}$:

$[x \geq y \rightarrow m := x \quad \square \quad y \geq x \rightarrow m := y]$
Repetitive Statement

\*\[ G_1 \to C_1 \to G_2 \to C_2 \to \ldots \to G_n \to C_n \].
Example 9

meeting-time-scheduling ::= $t := 0$;

* $t := a(t) \; \Box \; t := b(t) \; \Box \; t := c(t)$

Non-deterministic schedule brings the issue of fairness
Fairness

A scheduler that allows all possible schedules is *unfair*.

*Unconditional fairness*: every process (enabled or not) gets its turn infinitely often.

*\[true \rightarrow x := 0 \quad x = 0 \rightarrow x := 1 \quad x = 1 \rightarrow x := 2\]*

Unfair: $x = 0, 0, 0, 0, 0, 0, \ldots$

Unfair: $x = 0, 1, 0, 0, 0, 1, \ldots$

Fair: $x = 0, 1, 2, 0, 1, 0, \ldots$
Fairness (cond’t)

*Weak fairness*: every process that is continuously enabled from a certain time, gets its turn (infinitely) often.

*Strong fairness*: every process enabled infinitely often gets its turn (infinitely) often.

$x = T$

$\star \ [x \rightarrow y := T \quad x \rightarrow y := F \quad y \rightarrow x := F \quad y \rightarrow y := \text{not } y ]$

Weakly fair scheduler: termination is not guaranteed
Strongly fair scheduler: termination is guaranteed
Communication and Synchronization

- One-way communication: send and receive
- Two-way communication: RPC (Sun), RMI (Java and CORBA), and rendezvous (Ada)
- Several design decisions:
  - One-to one or one-to-many
  - Synchronous or asynchronous
  - One-way or two-way communication
  - Direct or indirect communication
  - Automatic or explicit buffering
  - Implicit or explicit receiving
<table>
<thead>
<tr>
<th>Primitives</th>
<th>Example Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARALLELISM</td>
<td></td>
</tr>
<tr>
<td>Expressing parallelism</td>
<td></td>
</tr>
<tr>
<td>Processes</td>
<td>Ada, Concurrent C, Lina, NIL Emerald, Concurrent Smalltalk</td>
</tr>
<tr>
<td>Objects</td>
<td>Occam</td>
</tr>
<tr>
<td>Statements</td>
<td></td>
</tr>
<tr>
<td>Expressions</td>
<td>Par Alfi, FX-87</td>
</tr>
<tr>
<td>Clauses</td>
<td>Concurrent PROLOG, PARLOG</td>
</tr>
<tr>
<td>Mapping</td>
<td></td>
</tr>
<tr>
<td>Static</td>
<td>Occam, Star Mod</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Concurrent PROLOG, ParAlfi</td>
</tr>
<tr>
<td>Migration</td>
<td>Emerald</td>
</tr>
<tr>
<td>COMMUNICATION</td>
<td></td>
</tr>
<tr>
<td>Message Passing</td>
<td></td>
</tr>
<tr>
<td>Point-to-point messages</td>
<td>CSP, Occam, NIL</td>
</tr>
<tr>
<td>Rendezvous</td>
<td>Ada, Concurrent C</td>
</tr>
<tr>
<td>Remote procedure call</td>
<td>DP, Concurrent CLU, LYNX</td>
</tr>
<tr>
<td>One-to-many messages</td>
<td>BSP, StarMod</td>
</tr>
<tr>
<td>Data Sharing</td>
<td></td>
</tr>
<tr>
<td>Distributed data Structures</td>
<td>Lina, Orca</td>
</tr>
<tr>
<td>Shared logical variables</td>
<td>Concurrent PROLOG, PARLOG</td>
</tr>
<tr>
<td>Nondeterminism</td>
<td></td>
</tr>
<tr>
<td>Select statement</td>
<td>CSP, Occam, Ada, Concurrent C, SR</td>
</tr>
<tr>
<td>Guarded Horn clauses</td>
<td>Concurrent PROLOG, PARLOG</td>
</tr>
<tr>
<td>PARTIAL FAILURES</td>
<td></td>
</tr>
<tr>
<td>Failure detection</td>
<td>Ada, SR</td>
</tr>
<tr>
<td>Atomic transactions</td>
<td>Argus, Aeolus, Avalon</td>
</tr>
<tr>
<td>NIL</td>
<td></td>
</tr>
</tbody>
</table>
Message-Passing Library for Cluster Machines (e.g., Beowulf clusters)

- **Parallel Virtual Machine (PVM):**
  [www.epm.orl/pvm/pvm_home.html](http://www.epm.orl/pvm/pvm_home.html)

- **Message Passing Interface (MPI):**
  [www.mpi.nd.edu/lam/](http://www.mpi.nd.edu/lam/)

- **Java multithread programming:**
  [www.mcs.drexel.edu/~shartley/ConcProjJava](http://www.mcs.drexel.edu/~shartley/ConcProjJava)
  [www.ora.com/catalog/jenut](http://www.ora.com/catalog/jenut)

- **Beowulf clusters:**
  [www.beowulf.org](http://www.beowulf.org)
Message-Passing (Cont’d.)

- **Asynchronous** point-to-point message passing:
  - send message list to destination
  - receive message list {from source}

- **Synchronous** point-to-point message passing:
  - send message list to destination
  - receive empty signal from destination
  - receive message list from sender
  - send empty signal to sender
Relationships Between Models

- Stronger model: a model with more constraints
  - Strong: synchronous systems, Weak: asynchronous systems
  - Strong: bounded-delay channels, Weak: unbounded-delay channels

- Can model A be simulated using model B: layers of abstraction
  - FIFO channel for Non-FIFO channel
    - sequence number
    - ack with bounded buffer
  - Asynchronous systems for synchronous systems
    - ABD synchronizer (for asynchronous bounded delay)
    - Awerbuch’s three synchronizers (for asynchronous unbounded delay)
Relationships Between Models (cont’d)

- Message Passing for Shared Memory: circular buffer

- Shared Memory for Message Passing: (total order) multicast
Example 10

The squash program replaces every pair of consecutive asterisks "**" by an upward arrow “↑”.

input::= * [ send c to squash ]
output::= * [ receive c from squash ]
Example 10 (Cont’d.)

\[
\text{squash::=}
\begin{align*}
&*[ \text{receive } c \text{ from input } \rightarrow \\
&[ \ c \neq * \rightarrow \text{send } c \text{ to output} \\
&\quad \quad [ \ c = * \rightarrow \text{receive } c \text{ from input}; \\
&\quad \quad \quad \quad \quad [ \ c \neq * \rightarrow \text{send } * \text{ to output}; \\
&\quad \quad \quad \quad \quad \quad \quad \text{send } c \text{ to output} \\
&\quad \quad \quad \quad \quad \quad \quad c = * \rightarrow \text{send } \uparrow \text{ to output} \\
&\quad \quad ] \ \square \\
&\quad ] \\
&\ ] \\
&\ ]
\end{align*}
\]
Partial Correctness

Neighbors are in different colors

Program produces correct results if the program terminates

Color set \{0, 1, 2, 3\}

Colorme::= *[c[i] = c[j] \land j \in N[i] \rightarrow c[i] := c[i] + 2 \mod 4]

Colorme is partial correct
Focus 8: Fibonacci Numbers

- $F(i) = F(i-1) + F(i - 2)$ for $i > 1$, with initial values $F(0) = 0$ and $F(1) = 1$.
- $F(i) = (\phi^i - \phi'^i)/(\phi - \phi')$, where $\phi = (1+5^{0.5})/2$ (golden ratio) and $\phi' = (1-5^{0.5})/2$.

0, 1, 2, 3, 5, 8, 13, 21, 35, 54, …
A solution for $F(n)$.
Focus 8 (Cont’d.)

- $f(0) ::=$
  
  send $n$ to $f(1)$;
  
  receive $p$ from $f(2)$;
  
  receive $q$ from $f(1)$;
  
  ans := q

- $f(-1) ::=$
  
  receive $p$ from $f(1)$
Focus 8 (Cont’d.)

- \( f(i) ::= \)
  
  ```
  receive n from f(i - 1);
  [ n > 1 → [ send n - 1 to f(i + 1);
               receive p from f(i + 2);
               receive q from f(i + 1);
               send p + q to f(i - 1);
               send p + q to f(i - 2) ]
  ]
  □ n = 1 → [ send 1 to f(i - 1);
              send 1 to f(i - 2) ]
  □ n = 0 → [ send 0 to f(i - 1);
              send 0 to f(i - 2) ]
  ```
Another solution for F (n).
Focus 8 (Cont’d.)

- \( f(0) ::= \)
  
  \[
  \begin{array}{l}
  \text{[ } n > 1 \rightarrow [ \text{send } n \text{ to } f(1) ; \\
  \text{receive } p \text{ from } f(1) ; \\
  \text{receive } q \text{ from } f(1) ; \\
  \text{ans } := p \\
  \text{]}
  \]
  \square n = 1 \rightarrow \text{ans } := 1 \\
  \square n = 0 \rightarrow \text{ans } := 0 \\
  \]
\]
Focus 8 (Cont’d.)

- \( f(i) ::= \)
  
  \[ \text{receive } n \text{ from } f(i - 1); \]
  \[ [ \, n > 1 \rightarrow [ \, \text{send } n - 1 \text{ to } f(i + 1); \]
    \text{receive } p \text{ from } f(i + 1); \]
  \[ \text{receive } q \text{ from } f(i + 1); \]
  \[ \text{send } p + q \text{ to } f(i - 1); \]
  \[ \text{send } p \text{ to } f(i - 1) \]
  \[ ] \]

- \( \Box n = 1 \rightarrow [ \, \text{send } 1 \text{ to } f(i - 1); \]
  \[ \text{send } 0 \text{ to } f(i - 1) \]
  \[ ] \]

]
Focus 9: Message-Passing Primitives of MPI

- MPI_Isend: asynchronous communication
- MPI_send: receipt-based synchronous communication
- MPI_ssend: delivery-based synchronous communication
- MPI_sendrecv: response-based synchronous communication
Message-passing primitives of MPI: Isend, send, ssend, sendrecv.
Focus 10: Interprocess Communication in UNIX

- **Socket**: `int socket (int domain, int type, int protocol).
  - domain: normally internet.
  - type: datagram or stream.
  - protocol: TCP (Transport Control Protocol) or UDP (User Datagram Protocol)

- **Socket address**: an Internet address and a local port number.
Focus 10 (Cont’d.)

Sender

```
 s = socket(AF_INET, SOCK_DGRAM, 0)
 ...
 bind(s, ClientSocketAddress)
 ...
 sendto(s, "message", ServerSocketAddress)
```

Receiver

```
 t = socket(AF_INET, SOCK_DGRAM, 0)
 ...
 bind(t, ServerSocketAddress)
 ...
 amount = recvfrom(t, buffer, from)
```

Sockets used for datagrams
High-Level (Middleware) Communication Services

- Achieve access transparency in distributed systems
  - Remote procedure call (RPC)
  - Remote method invocation (RMI)
Remote Procedure Call (RPC)

- Allow programs to call procedures located on other machines.
- Traditional (synchronous) RPC and asynchronous RPC.
Remove Method Invocation (RMI)

RMI is a generalization of RPC in an object-oriented system.
Robustness

- Exception handling in high level languages (Ada and PL/1)

- Four Types of Communication Faults
  - A message transmitted from a node does not reach its intended destinations
  - Messages are not received in the same order as they were sent
  - A message gets corrupted during its transmission
  - A message gets replicated during its transmission
Failures in RPC

If a remote procedure call terminates abnormally (the time out expires) there are four possibilities.

- The receiver did not receive the call message.
- The reply message did not reach the sender.
- The receiver crashed during the call execution and either has remained crashed or is not resuming the execution after crash recovery.
- The receiver is still executing the call, in which case the execution could interfere with subsequent activities of the client.
1. (The Welfare Crook by W. Feijen) Suppose we have three long magnetic tapes each containing a list of names in alphabetical order. The first list contains the names of people working at IBM Yorktown, the second the names of students at Columbia University and the third the names of all people on welfare in New York City. All three lists are endless so no upper bounds are given. It is known that at least one person is on all three lists. Write a program to locate the first such person (the one with the alphabetically smallest name). Your solution should use three processes, one for each tape.
Exercise 3 (Cont’d.)

2. Convert the following DCDL expression to a precedence graph.

\[
[ S_1 \parallel [ [ S_2 \parallel S_3 ]; S_4 ] ]
\]

Use fork and join to express this expression.

3. Convert the following program to a precedence graph:

\[
S_1;[[S_2;S_3\parallel S_4;S_5\parallel S_6]\parallel S_7];S_8
\]
Exercise 3 (Cont’d.)

4. $G$ is a sequence of integers defined by the recurrence $G_i = G_{i-1} + G_{i-3}$ for $i > 1$, with initial values $G_0 = 0$, $G_1 = 1$, and $G_2 = 1$. Provide a DCDL implementation of $G_i$ and use one process for each $G_i$.

5. Using DCDL to write a program that replaces $a*b$ by $a \uparrow b$ and $a**b$ by $a \downarrow b$, where $a$ and $b$ are any characters other than *. For example, if $a_1a_2*a_3**a_4***a_5$ is the input string then $a_1a_2 \uparrow a_3 \downarrow a_4***a_5$ will be the output string.