Course 02158

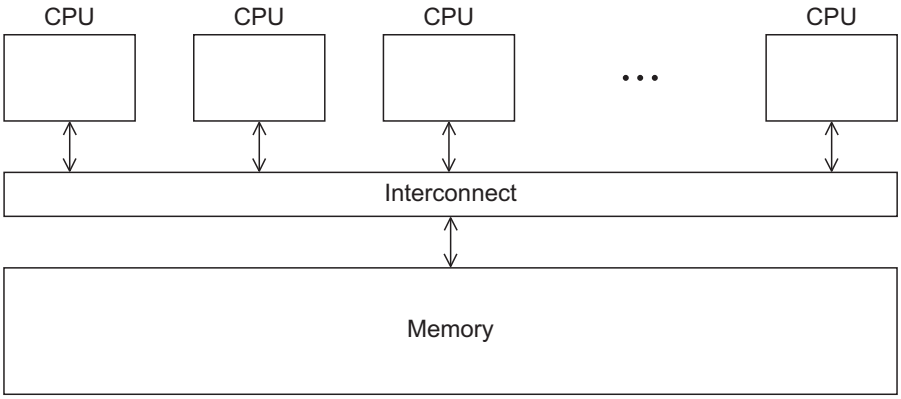
Message Passing

Hans Henrik Løvengreen

DTU Compute

Shared-Memory Systems — Multiprocessors

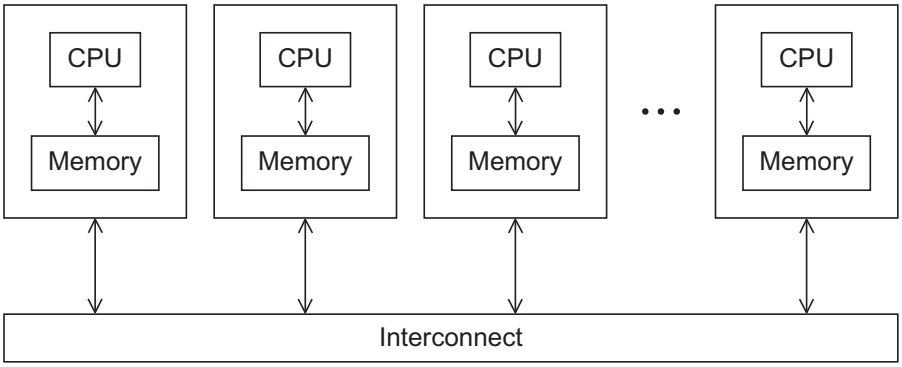
- Architecture



- Supports *shared state*

Distributed Memory Systems

- Architecture



- Calls for *message passsing* — no race conditions!
- But which kind?

Message Passing

- Abstract notions representing physical communication possibilities
- Many different names: Message queues, pipes, mailboxes, ...

Primitives

- **send** *msg* **to** *dest* **receive** *msg* [**from** *source*]

Aspects

- Direct or indirect addressing
- Synchronization (buffering)
- Message ordering
- Reliability
- Blocking of operations
- Selective communication
- Message types and encodings

Asynchronous Communication

- Uses *channels* buffering messages with given type
- Notation [Andrews]

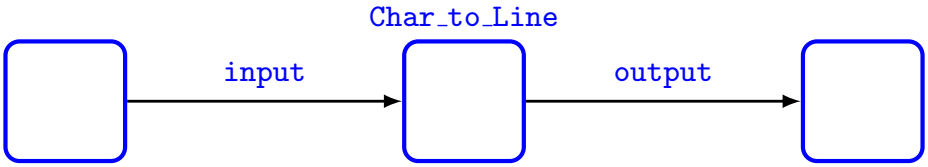
var *c* : **chan**< *T* >;

send *c*(*msg*) **receive** *c*(*msg*)

Properties

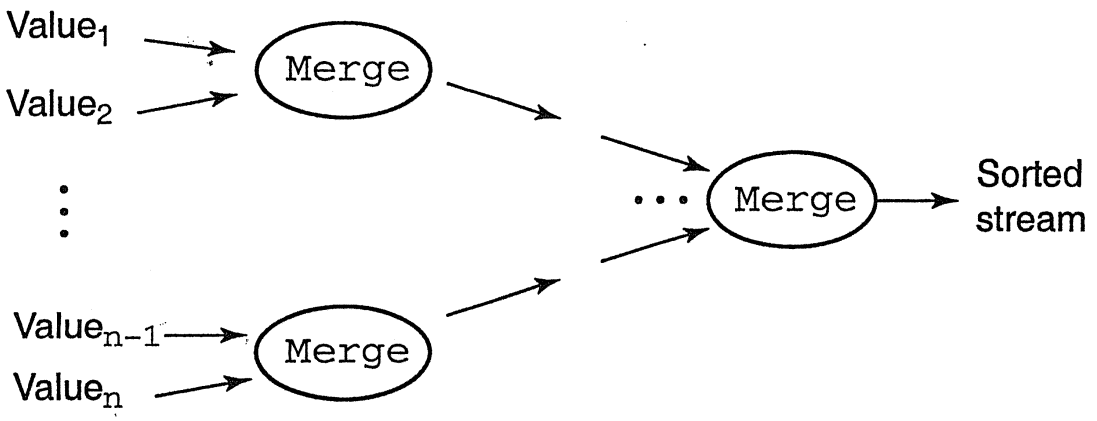
- Channels are typed
- No message loss
- Ordering is preserved (FIFO)
- The send operation never blocks
- The receive operation blocks on an empty channel
- There may be both several senders and several receivers

Pipelining: Example — line assembly




```
chan input(char), output(char [MAXLINE]);
process Char_to_Line {
  char line[MAXLINE]; int i = 0;
  while (true) {
    receive input(line[i]);
    while (line[i] != CR and i < MAXLINE) {
      # line[0:i-1] contains the last i input characters
      i = i+1;
      receive input(line[i]);
    }
    line[i] = EOL;
    send output(line);
    i = 0;
  }
}
```

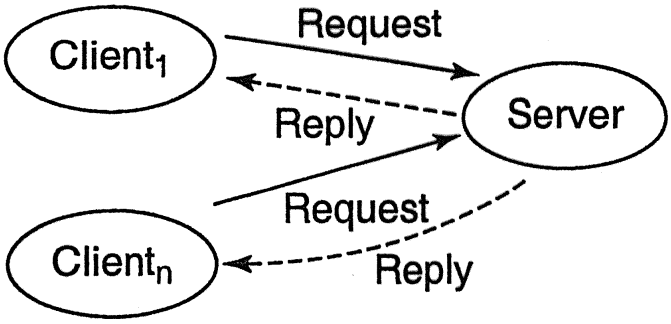
Parallel Merge Sort



Parallel Merge Sort

```
chan in1(int), in2(int), out(int);
process Merge {
  int v1, v2;
  receive in1(v1);  # get first two input values
  receive in2(v2);
  # send smaller value to output channel and repeat
  while (v1 != EOS and v2 != EOS) {
    if (v1 <= v2)
      { send out(v1); receive in1(v1); }
    else   # (v2 < v1)
      { send out(v2); receive in2(v2); }
  }
  # consume the rest of the non-empty input channel
  if (v1 == EOS)
    while (v2 != EOS)
      { send out(v2); receive in2(v2); }
  else   # (v2 == EOS)
    while (v1 != EOS)
      { send out(v1); receive in1(v1); }
  # append a sentinel to the output channel
  send out(EOS);
}
```

Client/Server Paradigm



Client/Server — one operation

- ```
  chan request(int clientID, types of input values);
  chan reply[n](types of results);
  ```

- ```
  process Server {
    int clientID;
    declarations of other permanent variables;
    initialization code;
    while (true) {   ## loop invariant MI
      receive request(clientID, input values);
      code from body of operation op;
      send reply[clientID](results);
  }
  ```

- ```
  process Client[i = 0 to n-1] {
    send request(i, value arguments);     # "call" op
    receive reply[i](result arguments);   # wait for reply
  }
  ```
  ```

Client/Server — multiple operations

```
type op_kind = enum(op_1, ..., op_n);
type arg_type = union(arg_1, ..., arg_n);
type result_type = union(res_1, ..., res_n);
chan request(int clientID, op_kind, arg_type)
chan reply[n](res_type);

process Server {
  int clientID; op_kind kind; arg_type args;
  res_type results; declarations of other variables;
  initialization code;
  while (true) {     ## loop invariant MI
    receive request(clientID, kind, args);
    if (kind == op_1)
      { body of op_1; }
    ...
    else if (kind == op_n)
      { body of op_n; }
    send reply[clientID](results);
  }
}
```

Client/Server — multiple operations

- ```
  process Client[i = 0 to n-1] {
    arg_type myargs; result_type myresults;
    place value arguments in myargs;
    send request(i, op_j, myargs);     # "call" op_j
    receive reply[i](myresults);       # wait for reply
  }
  ```
  ```

Resource Allocator: Monitor

```
monitor Resource_Allocator {
  int avail = MAXUNITS;
  set units = initial values;
  cond free;   # signaled when a process wants a unit
  procedure acquire(int &id) {
    if (avail == 0)
      wait(free);
    else
      avail = avail-1;
    remove(units, id);
  }
  procedure release(int id) {
    insert(units, id);
    if (empty(free))
      avail = avail+1;
    else
      signal(free);
  }
}
```

Resource Allocator: Types and Client

- ```
  type op_kind = enum(ACQUIRE, RELEASE);
  chan request(int clientID, op_kind kind, int unitid);
  chan reply[n](int unitID);
  ```

- ```
  process Client[i = 0 to n-1] {
    int unitID;
    send request(i, ACQUIRE, 0)    # "call" request
    receive reply[i](unitID);
    # use resource unitID, then release it
    send request(i, RELEASE, unitID);
    ...
  }
  ```

Resource Allocator: Server

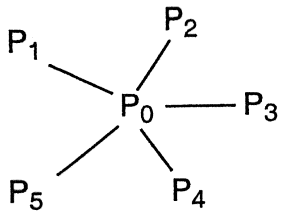
```
process Allocator {
  int avail = MAXUNITS; set units = initial values;
  queue pending;  # initially empty
  int clientID, unitID; op_kind kind;
  declarations of other local variables;
  while (true) {
    receive request(clientID, kind, unitID);
    if (kind == ACQUIRE) {
       if (avail > 0) {  # honor request now
          avail--; remove(units, unitID);
          send reply[clientID](unitID);
       } else   # remember request
          insert(pending, clientID);
    } else {   # kind == RELEASE
       if empty(pending) {  # return unitID to units
          avail++; insert(units, unitid);
       } else {  # allocate unitID to a waiting client
          remove(pending, clientID);
          send reply[clientID](unitID);
       }
    }
  }
}
```

Exchanging Values

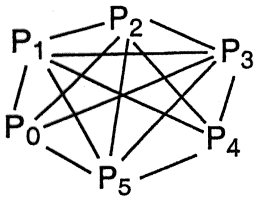
Problem

- A number of processes $P_0, P_1, \ldots, P_{n-1}$ have each found a *value* v_i
- All processes must learn the maximum and minimum of the values

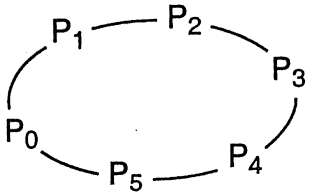
Patterns



(a) Centralized solution



(b) Symmetric solution



(c) Ring solution

Exchanging Values – centralized

```
chan values(int), results[n](int smallest, int largest);
process P[0] {    # coordinator process
  int v;    # assume v has been initialized
  int new, smallest = v, largest = v;  # initial state
  # gather values and save the smallest and largest
  for [i = 1 to n-1] {
    receive values(new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
  # send the results to the other processes
  for [i = 1 to n-1]
    send results[i](smallest, largest)
}
process P[i = 1 to n-1] {
  int v;    # assume v has been initialized
  int smallest, largest;
  send values(v);
  receive results[i](smallest, largest);
}
```

Exchanging Values – symmetric

```
chan values[n](int);
process P[i = 0 to n-1] {
  int v;    # assume v has been initialized
  int new, smallest = v, largest = v;  # initial state
  # send my value to the other processes
  for [j = 0 to n-1 st j != i]
    send values[j](v);
  # gather values and save the smallest and largest
  for [j = 1 to n-1] {
    receive values[i](new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
}
```

Exchanging Values – ring I

- ```
  chan values[n](int smallest, int largest);
  ```

- ```
  process P[0] {  # initiates the exchanges
    int v;   # assume v has been initialized
    int smallest = v, largest = v;  # initial state
    # send v to next process, P[1]
    send values[1](smallest, largest);
    # get global smallest and largest from P[n-1] and
    #   pass them on to P[1]
    receive values[0](smallest, largest);
    send values[1](smallest, largest);
  }
  ```

Exchanging Value – ring II

- ```
  process P[i = 1 to n-1] {
    int v;   # assume v has been initialized
    int smallest, largest;
    # receive smallest and largest so far, then update
    #   them by comparing their values to v
    receive values[i](smallest, largest)
    if (v < smallest)
        smallest = v;
    if (v > largest)
        largest = v;
    # send the result to the next processes, then wait
    # to get the global result
    send values[(i+1) mod n](smallest, largest);
    receive values[i](smallest, largest);
  }
  ```
  ```

Communication Sequential Processes (CSP)

- Language proposal by C.A.R. Hoare 1978

Idea

- A simple imperative langauge based on
  Dijkstra's Guarded Commands
- Plus a few simple concepts for concurrent programming
  - No shared variables
  - *Synchronous communication*
  - *Selection*

Impact

- Further eveloped into *Theoretical CSP* notation
- Realized in the Occam language
- Seminal for many theories of concurrency, esp. *process algebras*
- Inspiration for concurrency in Go

CSP: Basic Constructs

Dijkstra's Guarded Commands

- **if** $b_1 \rightarrow S_1 \;[]\; b_2 \rightarrow S_2 \;[]\; \ldots \;[]\; b_n \rightarrow S_n$ **fi**

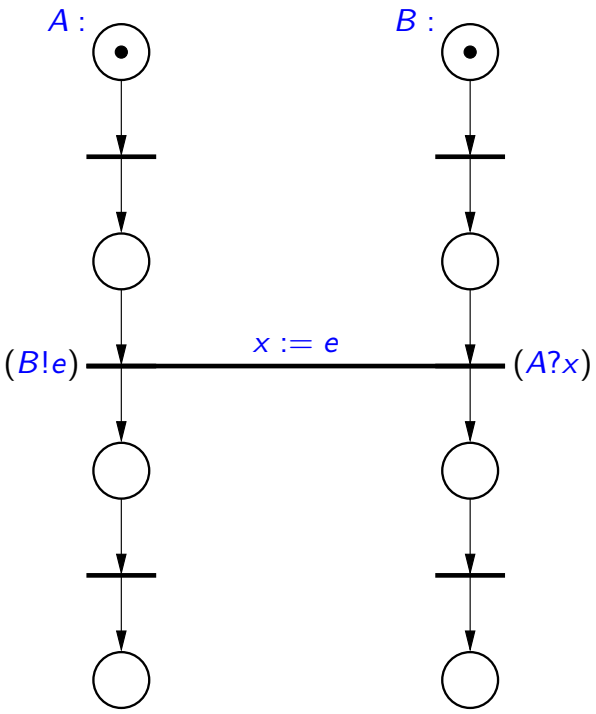
Synchronous Communication

- **process** A | **process** B

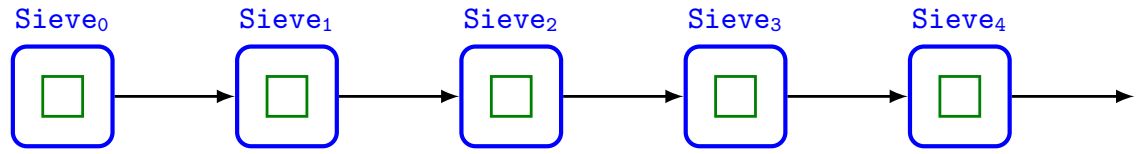
	$\vdots$	$\vdots$
Output	$B\,!\,e$	$A\,?\,x$ Input
	$\vdots$	$\vdots$

- A process has reached an output (!) operation
- Another process has reached an input (?) operation
- They name each other in the operations
- The output expression e matches the type of the input variable x
- No buffering needed!

CSP: Communication — Petri Net

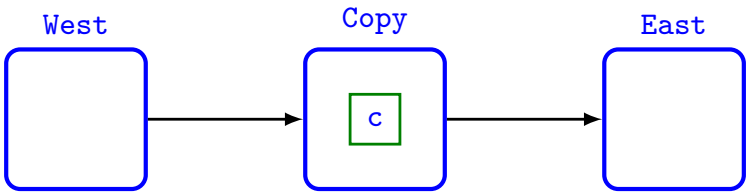


CSP: Sieve of Erathostenes



- **process** Sieve$_0$
  var p : integer := 2;
  for i **in** 3..n **by** 2 **do** Sieve$_1$! i;
- **process** Sieve[i **in** 1..L]
  var p, next : integer;
  Sieve$_{i-1}$? p;
  do true →
    Sieve$_{i-1}$? next;
    if (next mod p) ≠ 0 → Sieve$_{i+1}$! next
    [] (next mod p) = 0 → **skip**
    fi
  od

CSP: One element buffer



- **process** Copy
  var c : char;
  do true →
    West ? c;
    East ! c;
  od

CSP: Selection

Selective Communication

process A

$\vdots$

$C\,!\,e_1$

$\vdots$

process B

$\vdots$

$C\,!\,e_2$

$\vdots$

process C

$\vdots$

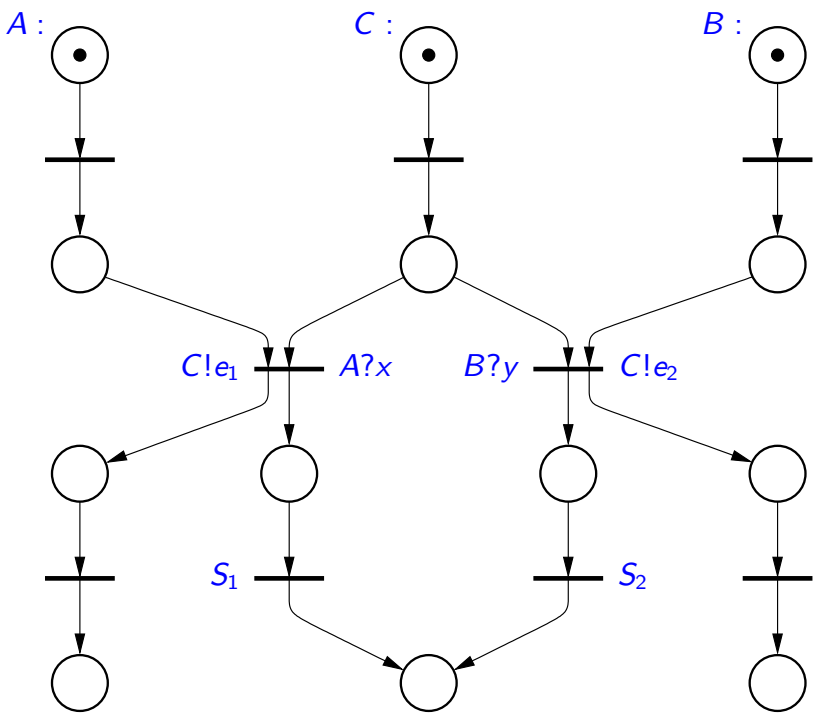
if $\quad A\,?\,x \rightarrow S_1$

[] $\quad B\,?\,y \rightarrow S_2$

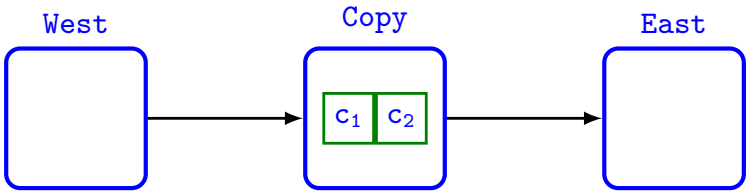
fi

$\vdots$

CSP: Selection — Petri Net



CSP: Two element buffer



- **process** Copy
  var c_1, c_2 : char;
  West ? c_1;
  do West ? c_2 $\rightarrow$ c_1; $c_1 := c_2$
  [] East ! c_1 $\rightarrow$ West ? c_1
  od

CSP: Selection

Selective Communication

process A

$\vdots$

C ! e_1

$\vdots$

process B

$\vdots$

C ! e_2

$\vdots$

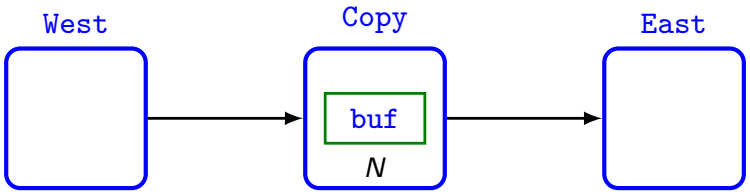
process C

$\vdots$

if b_1; A ? x $\rightarrow$ S_1
[] b_2; B ? y $\rightarrow$ S_2
fi

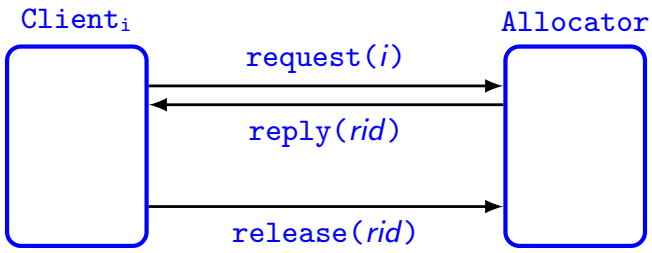
$\vdots$

CSP: N element buffer



- **process** Copy
  var buf[N] : char;
  in, out, count : integer := 0;
  do count < N; West ? buf[in] →
  count := count + 1; in := (in + 1) mod N
  [] count > 0; East ! buf[out] →
  count := count − 1; out := (out + 1) mod N
  od

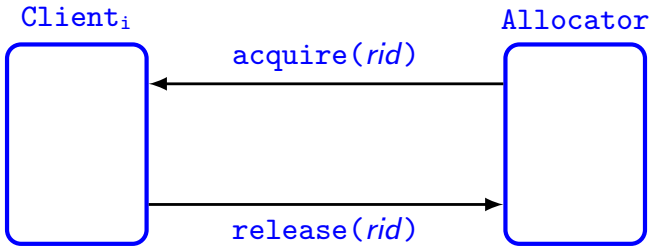
CSP: Resource Allocator



- Given a type of *resource identifiers*: `ResID`
- **process** `Client[i in 1..n]`
  var `rid : ResID;`
  ⋮
  `Allocator!request(i);`
  `Allocator?reply(rid);`
  Use resource `rid`
  `Allocator!release(rid);`
  ⋮

```
process Allocator
  var units : Set<ResID> := ... ;
      rid : ResID;
  do units.size() > 0;
       Client[*]?request(i) →
            rid := units.remove();
            Client[i]!reply(rid)
  []   Client[*]?release(id) →
            units.insert(id)
  od
```

CSP: Resource Allocator (alt.)



- Given a type of *resource identifiers*: `ResID`
- **process** `Client[i in 1..n]`
  var `rid : ResID;`
  ⋮
  `Allocator?acquire(rid);`
  Use resource `rid`
  `Allocator!release(rid);`
  ⋮

```
process Allocator
  var units : Set<ResID> := ... ;
  do units.size() > 0;
       Client[*]!acquire(units.next()) →
            units.remove();
  []   Client[*]?release(id) →
            units.insert(id)
  od
```

Theoretical CSP

- A formal notation for desribing processes (*process algebra*) based on CSP

Syntax

- Process expressions $P ::= \textbf{stop} \mid a \rightarrow P \mid P \,[]\, Q \mid P \,||\, Q \ldots$
- Example: $p \triangleq a \rightarrow (b \rightarrow p \,[]\, c \rightarrow \textbf{stop})$

Semantics

- Basic trasistion: $P \xrightarrow{a} Q$
- Axioms, e.g. $(a \rightarrow P) \xrightarrow{a} P$
- Inference rules, e.g.

$$\frac{P \xrightarrow{a} P'}{P \,[]\, Q \xrightarrow{a} P'} \qquad \frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{a} Q'}{P \,||\, Q \xrightarrow{a} P' \,||\, Q'}$$

- Forms the basis for defining process equivalence, refinement etc.

Occam

- Programming language developed in the 80'ies by the Inmos chip company
- Based directly on the CSP notions (albeit using typed channels). e.g.

```
PROC Copy (CHAN OF BYTE West, East)
  BYTE c:
  WHILE TRUE
    SEQ
      West ? c
      East ! c
```

- Named after *Ockham's Razor*

Entia non sunt mulitiplicanda praeter necessitatem

— William of Ockham (1287–1347)

Transputer

- Specialized processor supporting Occam with four *communication links*
- Transputers are readily connected in a *mesh* providing a parallel machine