Occam and Transputers

Occam: An Explicitly Parallel Language

- Occam based on Communicating Sequential Processes (CSP) formalism developed by Tony Hoare, Oxford, UK, and an experimental language by David May, Bristol, UK
- Designed to have a formal semantics suitable for automatic program transformations
- Many groups investigated direct translation of Occam into hardware
Transputers

- The transputer architecture was designed as an Occam engine
  - Transputer C compiler didn’t appear until much later, and initially produced inferior code compared with Occam compiler
- Original target for transputer was embedded control (robots) where interfacing to hardware directly was important
- Designed to allow large arrays of transputers to be connected easily
- Almost no glue logic required for minimal transputer node

Occam Basics

- Occam primitive is a process, five kinds:
  - Assignment \( x := y + 2 \)
  - Input \( \text{keyboard} ? \text{char} \)
  - Output \( \text{screen} ! \text{Char} \)
  - Skip \( \text{SKIP} -- \text{NOP that terminates} \)
  - Stop \( \text{STOP} -- \text{NOP that never terminates} \)
- Channels provide communication between processes
  - Unbuffered, point-to-point synchronous communication
  - Channels have declared protocol types

\[
\text{c ! x} \quad \text{Channel c} \quad \text{c ? y}
\]
Composing Sequential Processes

- SEQ executes sub-processes sequentially

  SEQ
  keyboard ? char -- read char from keyboard
  screen ! char -- write char to screen

- Can do replicated SEQ

  SEQ i = 0 FOR array.size
  stream ! data.array[i]
  -- equivalent to
  SEQ
  stream ! data.array[0]
  stream ! data.array[1]
  ...

Composing Parallel Processes

- PAR executes sub-processes in parallel

  PAR
  keyboard(kbd.to.ed)
  editor(kbd.to.ed, ed.to.screen)
  screen(ed.to.screen)
PAR for parallel execution

```plaintext
WHILE next <> EOF
    SEQ
    x := next
    PAR
    in ? next
    out ! x * x
```

Restrictions on parallel data access

- Variables modified in one arm of PAR cannot be read or written in other parts of PAR, e.g.,

```plaintext
PAR -- this PAR is invalid
SEQ
mice := 42 -- assigns to mice
c ! 42
c ? mice -- assigns to mice
```

Replicated PAR

- Can use replicated PAR to build array of parallel processes

```plaintext
PAR farmer()
    PAR i = 0 FOR 4 -- count must be constant
    worker(i)
```

![Diagram showing replicated PAR](image-url)
Alternation

- ALT combines a number of processes only one of which is executed
- Each process has a guard:
  - input on channel
  - wait on timer
  - can be predicated with boolean expression

\[
\text{ALT} \\
\text{left ? packet -- guard input statement} \\
\text{stream ! packet} \\
\text{right ? packet -- guard input statement} \\
\text{stream ! packet}
\]

- PRI ALT prioritizes sub-processes in textual order

Channel Protocols

- All channels have set of legal message types, the channel protocol. Compiler checks all uses of channels to ensure all communications are compatible with type of channel.
  - \text{CHAN OF [36]BYTE message : -- explicit array type}
  - \text{CHAN OF COMPLEX32 imp : -- named record (struct)}
  - \text{CHAN OF INT::[]BYTE link: -- length + vector}
  - \text{message ! “Hello, World!”}
  - \text{link ! len: [buffer FROM start]}
- Also supports tagged type channels, and sequential message channels
- Goal is type-safe communication
Configuration

- Occam application written as network of communicating processes
- Configuration step maps parallel process components onto available physical processors and maps channels to hardware links
- Configuration should not change correctness