Communicating Processes and Processors 1975 - 2025

David May

Background 1975-85

Ideas leading to CSP, occam and transputers originated in the UK around 1975.

1978: CSP published, Inmos founded
1983: occam launched
1984: transputer announced
1985: transputer launched and in volume production

This introduced the idea of a communicating computer - *transputer* - as a system component

Key idea was to provide a higher level of abstraction in system design - along with a design formalism and programming language

CSP, Occam and Concurrency

Sequence, Parallel, Alternative

Channels, communication using message passing, timers

Parallel processes, parallel assignments and message passing

Secure - disjointness checks and synchronised communication

Scheduling Invariance - *arbitrary interleaving* model

Initially used for software and programming transputers; later used for hardware synthesis of microcoded engines, FPGA designs and asynchronous systems

Transputers and occam

Idea of running multiple *processes* on each processor - enabling cost/performance tradeoff

Processes as *virtual processors*

Event-driven processing

Secure - runtime error containment

Language and Processor Architecture designed *together*

Distributed implementation designed *first*

Transputer overview

VLSI computer integrating 4K bytes of memory, processor and point-to-point communications links

First computer to integrate a large(!) memory with a processor

First computer to provide direct interprocessor communication

Integration of process scheduling and communication following CSP (occam) using microcode

What did we learn?

We found out how to

- support fast process scheduling (about 10 processor cycles)
- support fast interprocess and interprocessor communication
- make concurrent system design and programming easy - using lots of processes
- implement specialised concurrent applications (graphics, databases, real-time control, scientific computing)

and we made some progress towards general purpose concurrent computing using recongfigurablity and high-speed interconnects

What did we learn?

We also found that

- we needed more memory (4K bytes not enough!)
- we needed efficient system wide message passing
- we needed support for rapid generation of parallel computations
- 1980s embedded systems didn't need 32-bit processors or multiple processors
- most programmers didn't understand concurrency

General Purpose Concurrency

Need for general purpose concurrent processors

- in embedded designs, to emulate special purpose systems
- in general purpose computing, to execute many algorithms - even within a single application

Theoretical models for *Universal* parallel architectures emerged (as with sequential computing)

But they needed high performance interconnection networks

Also *excess parallelism* in programs to *hide* communication latency

Routers

We built the first VLSI *router* - a 32×32 fully connected packet switch

It was designed as a component for interconnection networks allowing latency and throughput to be matched to applications

Note that - for scaling - capacity grows as $p \times log(p)$; latency as $log(p)$

Low latency at low load is important for initiating processing; low (bounded) latency at high load is important for latency hiding

Network structure and routing algorithms must be designed together to minimise congestion (hypercubes, randomisation ...)

General purpose architecture

Key: ratio of executions/second to communications/second.This will be the lower of e/c (node executions/communications) and E/C (total executions/communications)

Bounded network latency l: hard bound for real-time; high expectancy for concurrent computing

Compiler: parallelise or serialise to match e/c; this produces p processes with interval i between communications

Loader: distribute the p processes to at most $p \times i/l$ processors

Open Microprocessor Initiative 1990

An architecture for multi-processor systems-on-chip

Interconnect protocol for memory access and message passing

Scalable interconnect

Processors, memories, input-output interfaces

Managing complexity of integrating and verifying components

Open ... but not open enough ...

Programmable platforms 2000-2010

Post 2000, divergence between emerging market requirements and trends in silicon design and manufacturing

Electronics becoming fashion-driven with shortening design cycles; but state-of-the-art chips becoming more expensive and taking longer to design ...

Concept of a single-chip tiled processor array as a programmable platform emerged

Importance of I/O - mobile computing, ubiquitous computing, robotics ...

XMOS 2005

Multiple processes and implemented in hardware

Process scheduling and synchronisation supported by instructions

Inter-process and inter-processor communication supported by instructions and switches - streamed or packetised communications

Input and output ports integrated into processor for low latency

Time-deterministic execution and input-output

Single-cycle instructions for scheduling and communications.

XMOS 2005

Event-based scheduling - a process can wait for an event from one of a set of channels, ports or timers

A compiler can optimise repeated event-handling in inner loops - the process is effectively operating as a programmable state machine

A process can be dedicated to handling an individual event or to responding to multiple events

Much more efficient than interrupts in which contexts must be saved and restored - to respond quickly a process must be *waiting*

Processes can replace hardware interfaces in many applications

Communicating processes 2015-2025

HPC, graphics, big-data, machine learning

- lots of communicating processors for performance; increasing need for energy-efficiency

Internet of things

- low energy, communicating, interfacing

Robotics (CPS)

- real-time - fusion of interfacing, communications, control, and machine learning

Programming and design

Focus on data, control and resource dependencies - *process structures and communication patterns*

Contrast:

- Conventional programming languages: over-specified sequencing
- Hardware design languages: over-specified parallelism

Need a single language to trade-off space and time (by designer or compiler); also need a semantics to do this automatically.

Expect to run *concurrent* applications on top of *concurrent* system software on top of *concurrent* hardware

Programming and design

CSP, occam and derivatives meet many of the requirements

In addition to being able to express the programs and designs

- verification is becoming more and more important
- error-containment is becoming essential - STOP is a starting point!

Transformations should be visible to programmers, not hidden inside compilers

Need to avoid hiding concurrency in libraries

Abstraction is for *managing* complexity, not *hiding* it!

Hardware

We can integrate thousands of processing components on a chip

We need to be able to design, verify and understand systems with lots of communicating processors

Hardware should support

- deterministic concurrent programming - and effective techniques for non-deterministic programming
- time-deterministic computing and communication
- error containment - it's very expensive unless the hardware does it

As far as possible, avoid heterogeneous hardware

Time-determinism

Many parallel programs rely on synchronisation (barriers, reductions)

Execution must be time-deterministic - but (eg) most caches aren't!

p: probability of no cache miss when executing program P

Suppose n copies of P in execute in parallel, then synchronise

Probability that the synchronisation will not be delayed = p^n

- For n = 100 and p = 0.99, p^n = 0.37
- For n = 1000 and p = 0.99, p^n = 0.00004

Contention in interconnection networks gives rise to similar problems

Universality

Turing: a Universal Machine can emulate any specialised machine

For Random Access Machines, the emulation overhead is constant

Is there an equivalent Universal Parallel Machine?

A key component is a Universal Network

Idea: A Universal Processor is an infinite network of finite processors

Another Idea: Use a non-blocking network

Universal Parallel Processors

Universal networks emulate specialised networks

Universal processors emulate specialised processors

Networks must have scalable throughput (bisection bandwidth)

Networks must have low latency ($\leq log(p)$) under continuous load

Use network pipelining for continuous (stream) processing: optimal

Use *latency hiding* otherwise: optimal with $log(p)$ excess parallelism

Program Structures

Parallel Random Access Machines

Data Parallelism; Systolic Arrays

Directed Dataflow Graphs

Task Farms and Server Farms

Sequential programs(!)

Recursive Embedding of any of the above

Communication Patterns

Communication and data access patterns are often known, especially in embedded processing (but also in HPC)

Communication can often be implemented as a series of permutation routing operations between known endpoints

Compilers can allocate processors and network routes

For unknown patterns, use randomisation

For many-to-one, use hashing and combining (or replication)

Composition

Patterns can be composed and embedded within each other

Sometimes the entire program evolution is visible to a compiler

Sometimes the evolution is data-sensitive

The issues in allocating processors and network routes mirror those of allocating memory in sequential processing

... local, global, stack, heap

How fast can a computation spread?

Non-Blocking Networks

Clos networks implement *permutations* on their inputs

A *strict-sense* network can always allocate a new route

A *re-arrangeable* network needs fewer routers but may require re-arrangement of existing routes

Known permutation + Re-arrangeable = Compile-time (or on-the-fly)

Unknown pattern + Re-arrangeable = Run-time using randomisation

Benes Networks

For parallel computers, use a *folded* network with two-way links

A network is built from switches with two *edge*-facing links and two *core*-facing links

At each switch, a single bit is needed to route each packet

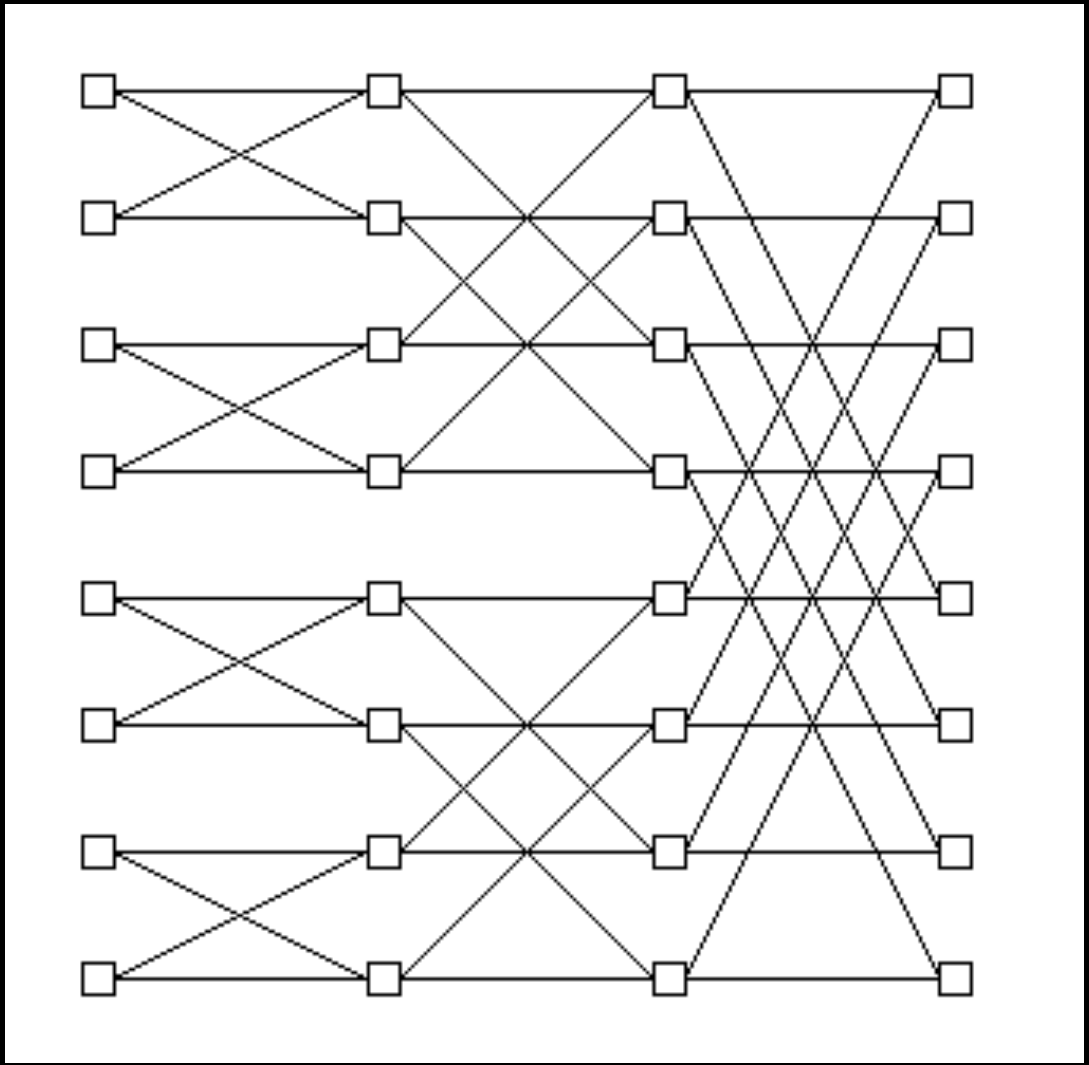
A network with l layers has 2^l edge facing links and 2^l core facing links

It connects 2^l processors together and provides 2^l *external* links

Route allocation is well understood; there are parallel algorithms

Folded Benes Network

edge core



Addressing and Partitions

Processor addresses are in the range 0 ... $2^n - 1$

A partition is of size 2^p and starts at base b: $(b \bmod 2^p) = 0$

Partitions are the unit of processor allocation, like pages for memory

Partitions can be used for (logical) synchronisation between processors

Within each partition, *synchronised messages* do not overtake those from a previous permutation

Routing

A message route starts with a *depth* that determines how far it progresses towards the core

The next part of the route determines the route towards the core

The final part routes the message to its edge destination

Each switch compares the depth part of each message from the edge with the depth of the switch

When they match, the switch routes the message back towards the edge

Partitions and Synchronisation

Within a switch, *synchronised messages* are forwarded from both inputs before a following synchronised message is forwarded

This enables an entire partition to perform a series of distinct permutations; it is *in-order pipelined*

This allows multiple channels per processor - compile-time communication scheduling is an extension of network path allocation

In-Order Pipelining = Time-Division Multiplexing (TDM)

2-phase TDM + re-arrangeable = strict-sense

Compiling communications

One-one communication: single permutation

Many-many communication: series of permutations

One-many (broadcast): series of permutations with forwarding via tree from root at source

Many-one (reducing): series of permutations with forwarding via tree to root at destination

All compilable communication patterns transformed to a series of permutations; no network contention

Emulating Sequential Processors

Distribute data structures across processors

Distribute procedures, functions and objects across processors

Accesses to data are less than 10% of instructions; calls are less than 5%

No contention - network is under-loaded

Optimisations: concurrent accesses and concurrent calls; moving program to data

An example network

Switch has 2 edge-facing links and 2 core-facing links.

There are d switch layers connecting 2^d processors

The interconnect contains $d \times 2^{d-1}$ switches

For $d = 16$, there will be 65536 processors and 524288 switches

A route will have a 4-bit depth, 15-bits for core routing and 16-bits for edge-routing

Another example network

Switch has 2^s edge-facing links and 2^s core-facing links.

There are d switch layers connecting 2^{sd} processors

The interconnect contains $d \times 2^{s(d-1)}$ switches

For $s = 4$ and $d = 4$, the routers have 16 edge-facing and 16 core-facing links

There will be 65536 processors and 16384 switches

A route will have a 4-bit depth, 15-bits for core routing and 16-bits for edge-routing

Technology and Packaging example

Network on processing chip has 256 links to on-chip processors, 256 to network core

Routing chips have 256 links to network edge; 256 to network core

Silicon Photonics would (massively) improve inter-device connections

Synchronisation is only needed between adjacent router layers

Processors handle input and output at the edge of the system

Processor Node Architecture

Almost any processor architecture can be used - provided that it can input and output messages concurrently

Conventional interrupt mechanisms can do this

Low latency processor-network interface is useful

Multi-threading / process scheduling is useful

Deterministic execution is useful

Simple architecture enabling on-the-fly compilation is useful

Transputers revisited

Original instruction set is compact and needs few registers

- program density compensates for additional memory operations
- no need for pipelining in (eg) low-energy applications
- could address many embedded applications

An alternative is a register-based instruction set exploiting wide access to memory to build a pipelined (or a superscalar) transputer

- all context registers written or read in one cycle
- scheduling instructions similar to the original
- could address high performance applications

The languages

Emphasis on *process structures* and *communication patterns* should replace emphasis on data structures and algorithms

A shift in thinking - a universal computer is an infinite array of finite processors, not a finite array of infinite processors

Our languages should support optimising transformations - and our compilers and tools should implement concurrency optimisations

It's time to educate a generation of *concurrent coders!*

The system components

Communication *links* supporting on-chip (wide) and inter-chip (narrow) communication - and easy conversion between them

Network switches with links

Computers with links and input-output ports

Interfaces between links and ‘standard’ communication protocols

Interfaces between links and external devices

Specialised accelerators with links

The Open Transputer

The development and enhancement of the processors, switches and links should be open - everyone can contribute and reap the benefits

Also the essential compilers and tools

The development of proprietary specialised interfaces and accelerators using the link and software tools should be permitted

This enables the collaborative development of the core technologies, whilst enabling commercial innovation

It shouldn’t be too difficult to achieve, using existing licenses ...