

University of Surrey
Department of Computing

Module CSM23 - Grid Computing

Lectures in Parallel Computing

Dr. Roger M.A. Peel

R.Peel @ surrey.ac.uk

<http://www.computing.surrey.ac.uk/personal/st/R.Peel/csm23/index.html>

Introduction

In my two lectures, I hope to introduce some of the important issues of any parallel computing activity - be it high-performance computing, multiprocessor embedded systems or Grid computing:

- Why use Parallel Computing?
- Definitions and Limitations of Parallelism
- Major Classifications of Parallelism
- Communication Schemes
- Deadlock and Livelock
- Applications and Examples

Why use Parallel Computing?

There are several reasons why we might choose to use parallel processing:

- Performance;
- Economy;
- Ease of programming.

This parallelism might come from:

- A pipelined or superscalar processor;
- A vector processor;
- A single-box multiprocessor;
- Multiple processors connected with communication lines [distributed processing]

Definitions and Limitations of Parallelism

The performance of a distributed parallel processor is influenced by:

- The *number* and *throughput* of the processing nodes.
- The *bandwidth* between the processing nodes. This is measured in Mbytes/sec (or Gbytes/sec) and determines the rate at which data can be sent from one node to another. It may vary for different pairs of nodes in a system.
- The *latency* in the node-to-node connections measures their delays in transmission - influenced by the bit rate, the time taken to route data, and the transmission protocol used. This is measured in seconds.
- The *Flow Control* strategy used influences performance - nodes must be prevented from flooding each other, or the routing fabric, with traffic.
- The *Speedup factor* for a parallel computer can be defined as

$$S(n) = \frac{\textit{Execution time using one processor}}{\textit{Execution time using a multiprocessor with } n \textit{ processors}}$$

Definitions and Limitations of Parallelism - Amdahl's Law

Any parallel code will also have sequential elements - at startup / shutdown, at the beginning and end of each loop, and so on. Consider how much processing is done sequentially and in parallel.

Gene Amdahl suggested the following law, relating to vector processors, but it is equally appropriate to VLIW machines, MIMD multiprocessors, networks of Grid computers and so on.

If the fraction of code in an application that cannot be parallelised is f , and the time taken for the whole computation on one processor is t , the time taken to perform the computation with n processors is given by

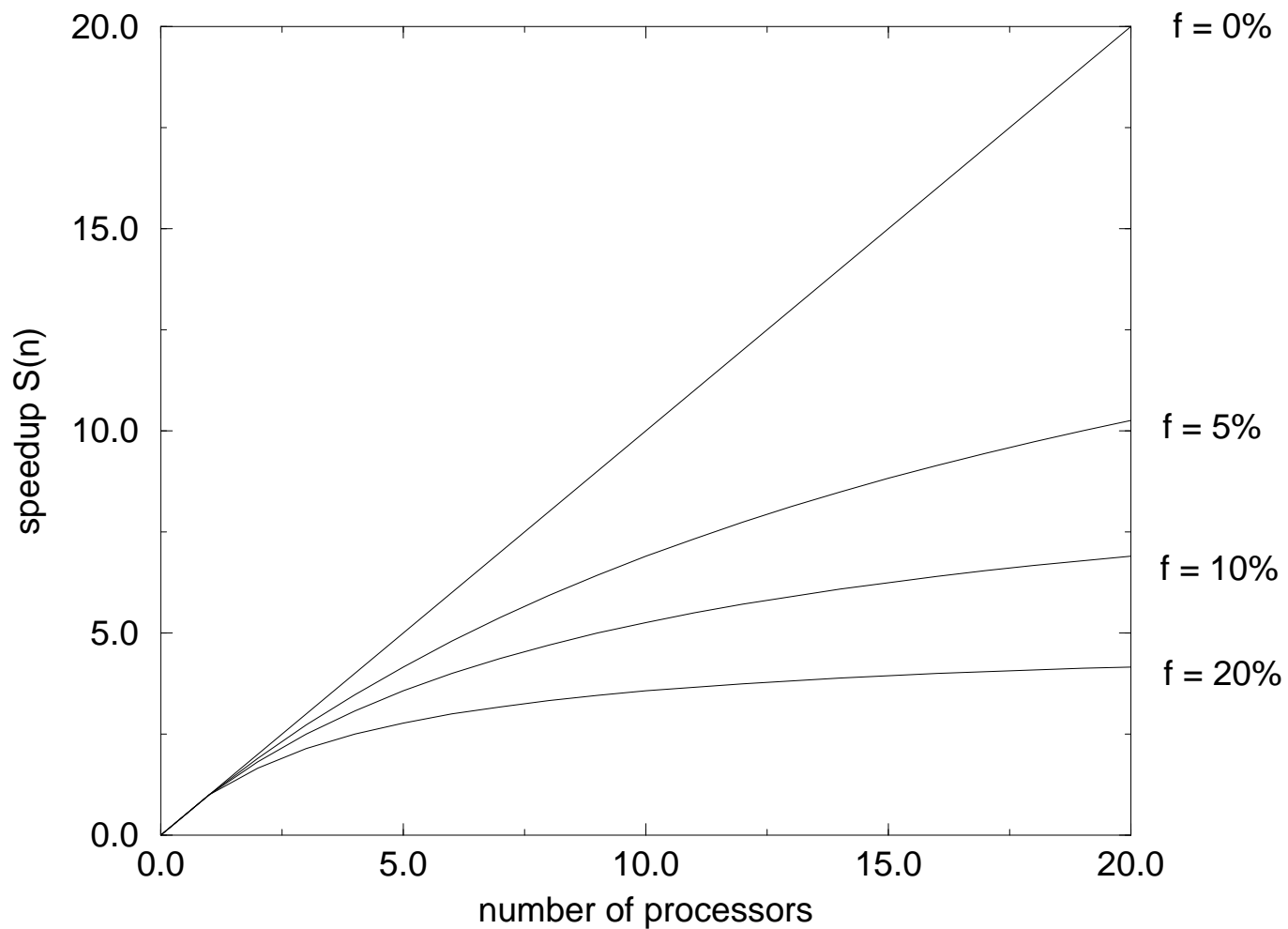
$$ft + (1 - f)t/n$$

and the speedup factor is

$$S(n) = \frac{t}{ft + (1 - f)t/n} = \frac{n}{1 + (n - 1)f}$$

This ignores the overhead that is usually added by parallelisation, as well as any communication costs between multiprocessors - often a significant extra cost.

Definitions and Limitations of Parallelism - Amdahl's Law



Speedup vs No. of Processors

Lessons from Amdahl's Law

From the above, we can see:

- * Even for a huge number of processors (or pipelined vector elements), the maximum speedup is $1/f$
- * Small reductions in the sequential overhead can make a huge difference in throughput.
- * Reducing the serial & communications overhead (e.g. by running communications and computation in parallel) is very beneficial.

Major Classifications of Parallelism

There are only three major classes of parallel processing:

Algorithmic

we split the algorithm into sections (e.g. pipelining)

Geometric

we split the static data space into sections (e.g. process an image on an array of processors)

Processor Farming

we pass the input data to many processors (e.g. pass ray-tracing coordinates to several processors, one ray at a time)

Some parallel applications might combine these techniques.

But first - Load Balancing

There are three forms of load balancing :

- Static load balancing

The choice of which processor to use for each part of the task is made at compile time. This is inflexible, but simple.

- Semi-dynamic load balancing

The choice of processor is made at run-time, but once started, each task must run to completion on the processor chosen. This is more efficient, but not always possible.

- Fully-dynamic load balancing

Here, tasks may be interrupted and moved between processors at will. This will enable processors with different capabilities always to be used to best advantage (e.g. FPU). The context-switching and communication cost of interrupting and moving processes (and their data) may outweigh the gains of optimal scheduling.

Algorithmic Parallelism

Many tasks can be split so that a stream of data may be processed in successive stages on a series of processors.

As the first stage finishes its processing, its result is passed to the second stage. The first stage may now accept more input data and process it. When the second stage finishes, it passes its result on, accepts the result from the first stage and so on. Thus the first task to be submitted passes out of the pipeline in n cycles, where n is the number of processing stages in the pipeline.

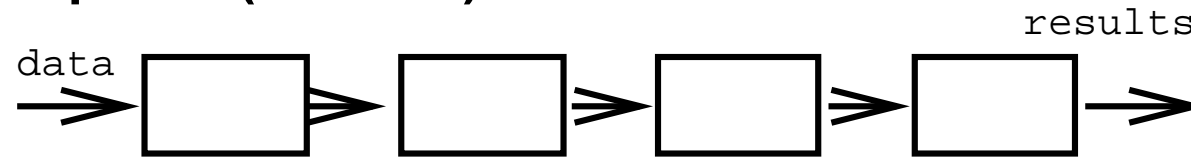
When the pipeline is full, one result is produced at every cycle

At the end of continuous operation, the early stages in the pipeline go idle before the last result is flushed out.

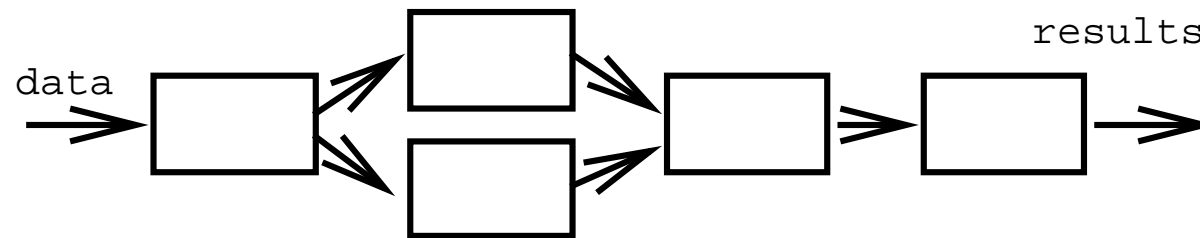
Load balancing is **STATIC** - the speed of the pipeline is determined by the speed of the slowest stage.

Algorithmic Parallelism

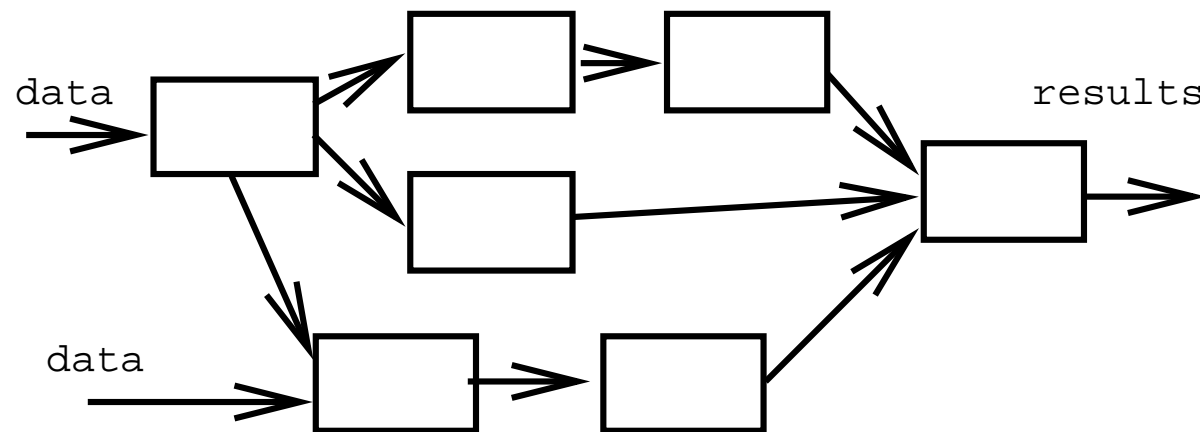
Linear Pipeline (or chain)



Pipeline with parallel section



Irregular network (general case)



Geometric Parallelism

Some regular-patterned tasks are suitable for processing by spreading their data across several processors and performing the same task on each section in parallel.

Many examples involve image processing - spread the pixels of an image across an array of transputers and perform local averaging, contrast enhancement etc. on each portion of the image.

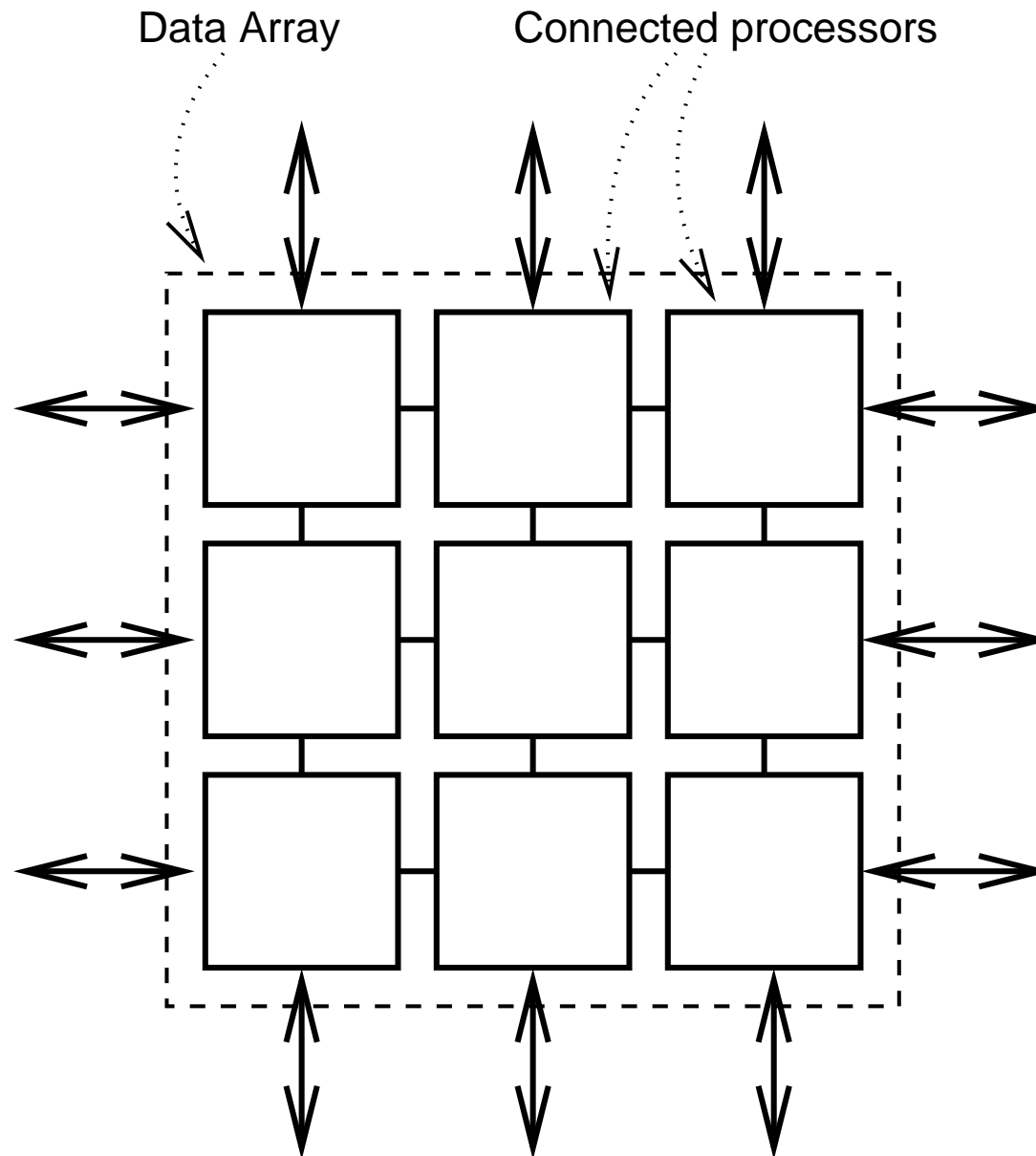
Many such tasks involve communication of boundary data from one portion of the image to another. (e.g. Conway's Game of Life). It may be possible to duplicate this edge data.

Load balancing is **STATIC** - the initial partitioning of the data determines the execution time to process each area. Rectangular blocks may not be the best choice - consider stripes, concentric squares, etc.

Initial loading of the data may prove to be a serious overhead.

Geometric Parallelism

(Data is distributed amongst the available processors)



Processor Farming

The third major form of parallelism involves sharing work out from a central controller process to several worker processes. The latter just accept packets of command data and return results. The controller splits up the task, sending work packets to free processors (i.e. ones which have just returned a result) and collating these results. Global data may be sent to all worker processes at the outset.

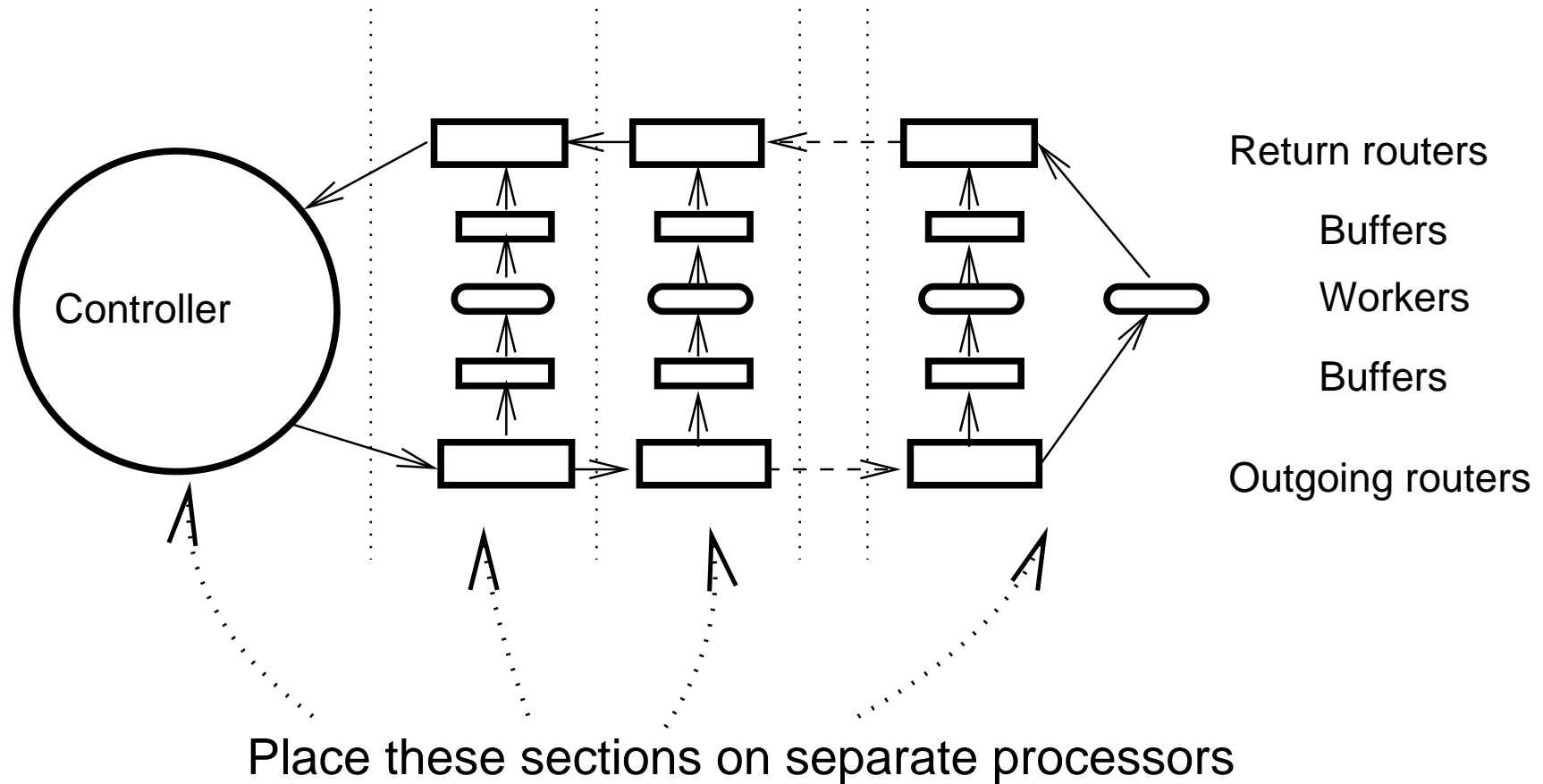
Processor farming is only appropriate if :

- * The task to be performed can be split up into many INDEPENDENT sections.
- * The amount of communication (commands + result) takes significantly less time than the processing itself.

To minimise latency, it may be better to keep two (or maybe three) packets in circulation for each worker - buffers are needed around it.

Load balancing is SEMI-DYNAMIC - the command packets are sent to processors which have just (or are about to) run out of work. Thus all processors are kept busy except for the closedown phase (when some finish before others)

A Processor Farm



Communication Schemes

Most inter-processor “communication” in a parallel computer can be categorised in one of three ways:

Shared Memory

- A shared memory multiprocessor provides a single memory map - for all operands or just for synchronisation.
- Processes may wait on *semaphore* variables until other processes are ready to share data.
- Once both processes synchronise, the data is copied and the processes release the semaphore.

Communication Schemes - 2

Synchronous Message Passing

- This typically uses blocking point-to-point channel communications.
- The first process of the pair to arrive at the channel communication instruction stops, and waits until the second arrives at its comms instruction.
- Then, data flows across the channel.
- Only when all of the data has been transmitted do the two processes continue separately again.

Communication Schemes - 3

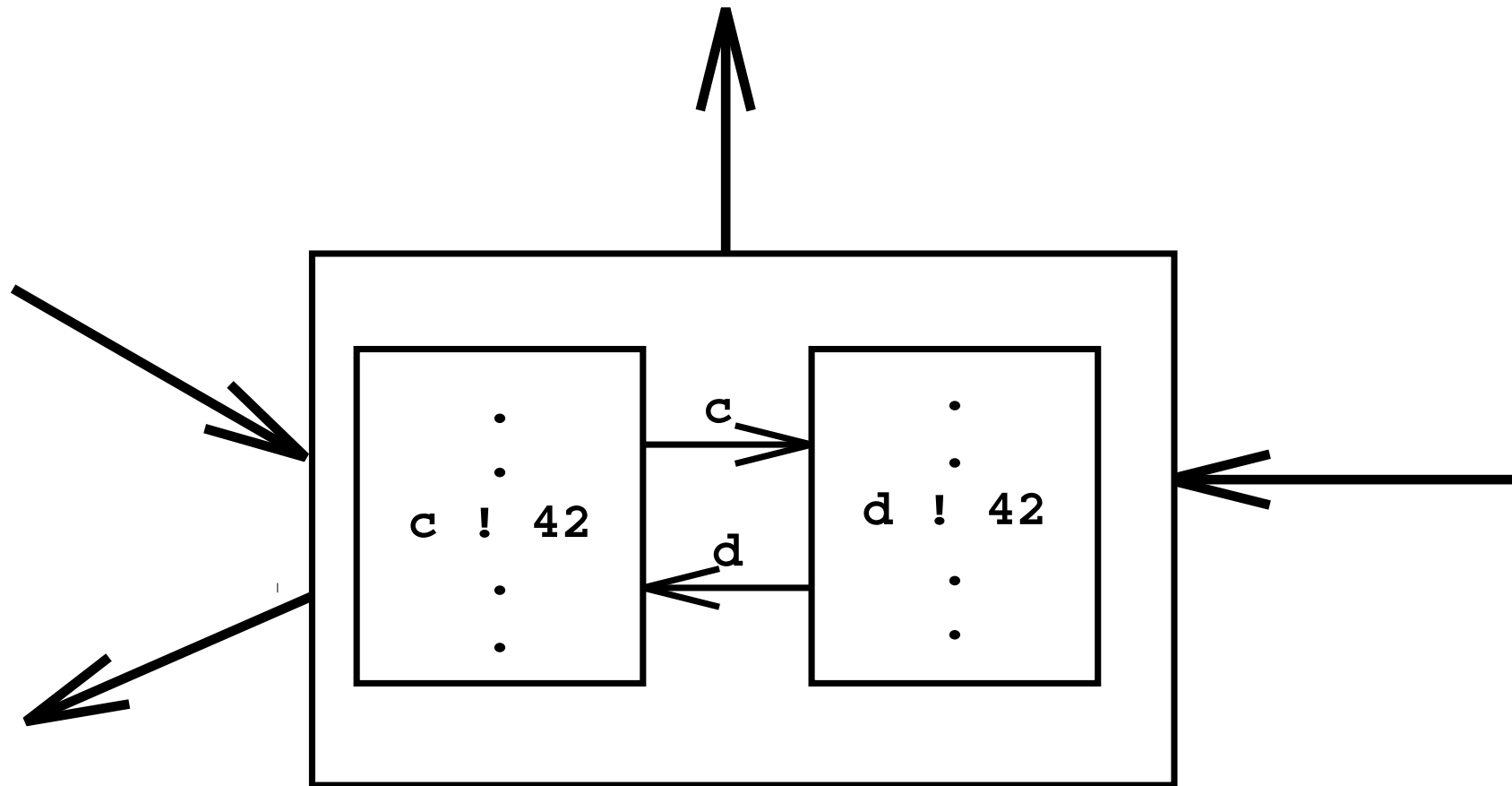
Asynchronous Message Passing

- An asynchronous transmitter sends its messages without blocking, thereby synchronising very loosely with the channel.
- The receiver necessarily blocks when it tries to read data from an empty channel.
- In effect, this method implements a channel with some degree of buffering.

These three schemes have strong similarities - it is possible to model each in terms of the initial two.

Deadlock and Livelock - 1

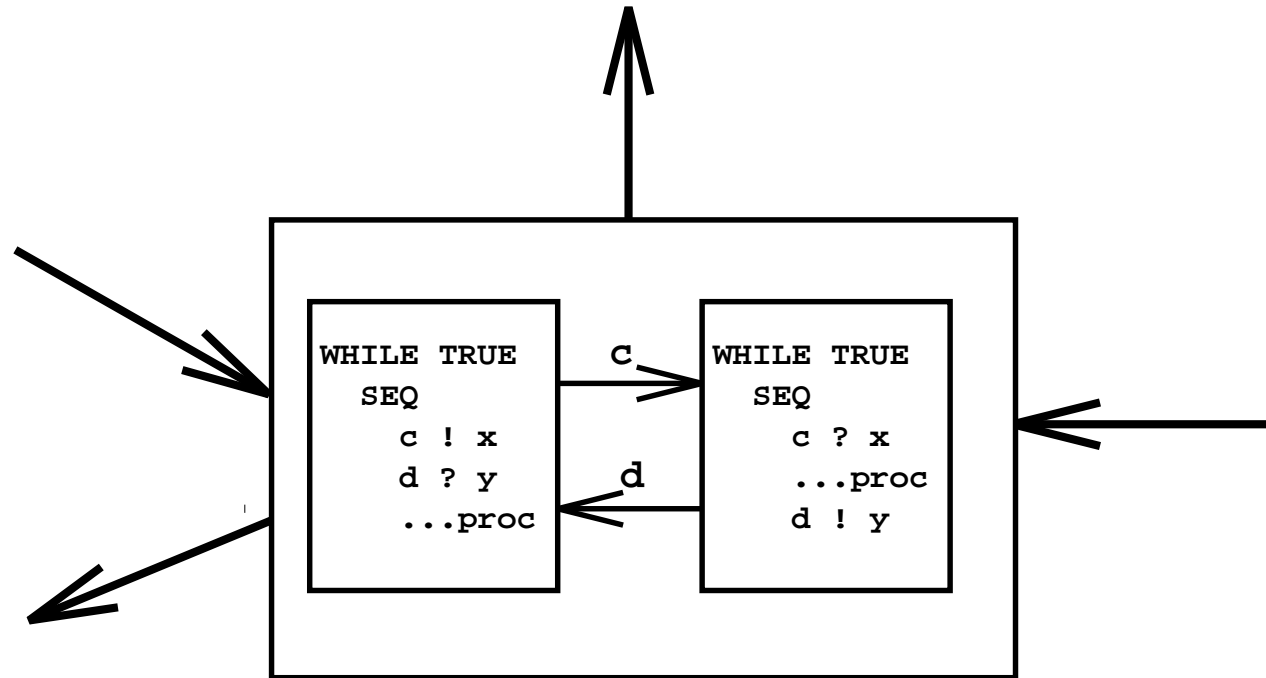
Consider a program in which two processes attempt to send synchronised messages to each other ... but in which neither process is prepared to receive the others' message.



This is a very simple example of a deadlocked system.

Deadlock and Livelock - 2

Consider another program in which two processes never communicate on their external channels, but remain busy internally.



This illustrates livelock. It appears identical to deadlock from the external viewpoint - but additionally consumes processing resources.

Deadlock - 3

- is a common design fault;
- its absence requires reasoning and / or verification
 - exhaustive testing cannot not work
 - verification requires us to check all the possible circumstances in which deadlock could occur.