University of BRISTOL

Department of Computer Science

Reducing the Number of Collisions in the Interconnection Network of a General-Purpose Parallel Architecture

Benjamin Tovar Matthews

May, 2015

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Benjamin Tovar Matthews, May 2015

Abstract

Contents

List of Figures

List of Tables

1 Introduction

1.1 Definitions

This thesis includes a number of terms that have been clarified below to avoid any potential ambiguity.

Processor: An integrated circuit which contains at least an arithmetic logic unit (ALU), registers, and a block of memory. Used to execute instructions outlined by programming code.

Network: A number of processors connected together by physical wires and intermediate nodes, allowing each processor to communicate with others in the network.

Link: A single interconnect in the network connecting either two processors, a processor and an intermediate node, or two intermediate nodes.

Routing algorithm: The algorithm that decides which links in the network are to be used when a communication is required between two processors on the network.

Collision: When two packets of communication data in the network try to use the same link at the same time, resulting in one of the packets having to wait until the link becomes free again before continuing its transmission.

1.2 Outline of the problem

The world of computer programming is currently undergoing a paradigm shift from sequential programming to parallel programming. Sequential programs execute their set of instructions one after another in order, which means they can be run using a single processor. Parallel programs on the other hand include sections of instructions that can be executed independently from one another, which means they can be run on more than one processor in parallel. Applying parallelism to a sequential program means that the execution time of the program is reduced – providing the program has independent sections suitable for parallel execution – because some of the computation is being carried out concurrently.

However parallel computing is not without its drawbacks. While parallel programs are generally faster than sequential programs, they require more hardware and suffer larger overhead costs in order for the program to be executed correctly. In other words, a fully parallel version of a sequential program executed on n processors will not run n times faster than the sequential program runs on one processor, because of the additional overheads that parallelism introduces.

The main source of overhead during execution of a parallel program is communication between processors. Communication is an essential part of parallel computing because it

allows processors to share their calculated data with the other processors they are working in collaboration with, which is necessary in almost all parallel programs. This communication is facilitated by a network of links which carry communication data from processor to processor. It is in these physical links where the majority of the overhead associated with parallel computing is sourced, resulting in slower execution time and reduced efficiency of parallel programs.

Ideally, a fully parallelised version of a sequential program would result in a linear speedup based on the number of parallel processors involved in the computation. However, due to the added overheads of parallelism, this is currently not the case. This thesis focuses on trying to reduce the significance of this overhead so that linear speedup due to added parallelism is more realistically achievable.

1.3 Possible optimisation of current solutions

One optimisation that has become a lot more accessible in the last few years is the replacement of traditional copper wires in the network with optic cables, making the network photonic. This means that instead of data being transmitted over the network in the form of differing voltages of electric current, it is transmitted as pulses of light, which makes the network latency a function of the speed of light (approximately 3.0×10^8 m/s), and eliminates inefficiencies such as friction between the electric current and the copper wire. However, while network latency of the speed of light sounds sufficient, in some cases it can still act as the computational bottleneck. An average modern CPU is manufactured with a clock speed of around 2.0 GHz, which equates to 2.0×10^9 clock cycles per second. This means that in 0.5 nanoseconds – the time taken for one clock cycle to be carried out on this CPU – light only travels around 15cm. This may be sufficient to carry out communication in one clock cycle in smaller networks, but for larger networks such as supercomputers which may have links spanning several metres, communication can take a lot longer. However, current research suggests that no advancement of the speed of light is possible at present, meaning that, for the time being at least, optimisations in the network must be sourced from elsewhere.

Another possible optimisation involves increasing the efficiency of the way in which data is transmitted across the network. When lots of communication is being carried out concurrently by multiple processors, links in the network will inevitably begin to fill up leading to a situation where more than one piece of data tries to use the same link at the same time to advance towards its destination. This is known as a collision in the network. Sending both pieces of data down the same link not physically possible, meaning that one of the pieces of data is forced to wait for the link to become vacant again before continuing towards its destination, resulting in increased network latency [Lam04]. This suggests that the fewer collisions that happen in the network, the lower the average network latency and the more efficiently communication data can be transmitted, making reducing the number of collisions in the network a possible strategy for optimising the execution time of parallel programs.

A third possible optimisation of current networks involves changing the way data is stored for each processor in the network. There is some variation between models and manufacturers, but a typical modern processor has access to four blocks of memory known as the memory hierarchy: CPU registers, cache memory (SRAM), main memory (DRAM) and the hard disk [Toy86]. The memory stores towards the top of the hierarchy allow the processor the quickest access to their data with limited storage capacity, whereas the memory stores towards the bottom of the hierarchy offer the largest stores of data but with slower access rates. This suggests that in terms of program efficiency, it is profitable to store as much data as high up the memory hierarchy as possible, meaning that the time taken by the processor to access data is reduced. It could also suggest that increasing the memory capacity of the stores towards the top of the hierarchy, the cache for example, might increase the efficiency of the processor. However, increasing the cache size would result in an increased number of memory addresses, which would subsequently mean that the processor would take longer to find and access a given address which reduces the effect of having cache memory in the first place. This means that a compromise between size and speed must be found, and further research into that area could result in more efficient processors and an increase in the efficiency of parallel programs which use these processors.

1.4 Motivation

Moore's law states that processing power should approximately double every two years [Moo65]. In terms of parallel processors, this performance increase can be acquired by either increasing the number of transistors in each processor, or by increasing the number of processors used for parallel computation. However both of these solutions are flawed; in recent years the performance of serial processors has plateaued (for reasons described in section 2.1), and while adding more processors to the network may increase the overall processing power, it also increases the overhead costs making the processing less efficient as well as bearing a significant financial cost. If a solution can be found that increases the efficiency of parallel computation without the need of large numbers of added processors, this could potentially save hardware manufacturers a lot of money while still allowing them to emulate the advancement of Moore's law.

1.5 Purpose

The main aims of the thesis are outlined below:

- Devise a solution to the problem of efficiency in general-purpose parallel architectures documented in section 1.2.
- Create an emulation of a network, including multiple processors which are able to carry out message passing instructions to each other. Parallelism in this network should be at least emulated.

- Deploy the solution on this emulated network. The improvement that the solution provides should be documented by the program as a quantitative, measurable output.
- Ensure that the network remains universal so that the solution can be applied to any network topology (explained in section 2.5).
- Ensure that the network remains scalable so that the solution can be applied to a network with any number of parallel processors in collaboration.

1.6 Outline of the following chapters

The remainder of the thesis is structured as follows:

Chapter 2: A background chapter to introduce to some of the fundamental concepts discussed in the thesis.

Chapter 3: An outline of the proposed solution and a detailed specification of requirements for this solution.

Chapter 4: The design of the network and the solution, and a description of how the solution will be deployed on the network to produce meaningful results.

Chapter 5: The implementation of the network and the solution.

Chapter 6: Testing of the network to ensure that it works correctly.

Chapter 7: Experimentation to show whether the solution produces the desired increase in network efficiency.

Chapter 8: A conclusion of the work done in the thesis and suggestions of further work that could be carried out in this field.

2 Background

This chapter describes some of the fundamental concepts needed to understand what is trying to be achieved with this project. It starts off with an introduction to the concept of parallelism. It then goes on to study parallel architecture in more depth, particularly the role that the interconnection network has in the system. Finally, it looks at some examples of modern general-purpose parallel architectures, and how the interconnection network provides communication between processors.

2.1 General-purpose processors: a need for parallelism

In the beginning processors were designed and built in a serial manner, using a single core, a single block of memory and a single set of instructions executed one after another. This setup is referred to as a SISD (single instruction, single data) architecture, a term defined in Flynn's Taxonomy as one of four classifications of processor architecture [Fly72]. This architecture suited early processors because it was easy to implement, and expectations of processing performance were not as demanding as they have become today because very few general-purpose machines existed. However, as more research was conducted and use of general-purpose machines became more common, it was clear that optimisation of this architecture was required to boost the performance of processors that were executing increasingly more complex programs and calculations.

At first, work was able to be done using the SISD architecture to increase processing performance. Techniques such as the pipelining of the fetch-execute cycle were introduced, which allowed different parts of the cycle to be carried out on different instructions in a concurrent manner, resulting in an instruction being executed on every clock cycle [Iba08]. This is an important increase compared to the whole fetch-execute cycle being carried out fully on each instruction before moving on to the next; assuming each step in the fetch-execute cycle takes one clock cycle, the resulting performance would be one instruction executed every three clock cycles. This suggests pipelining increased processor performance by a factor of three.

However, techniques such as pipelining were not the main source of increased performance for SISD architectures. It was clear that clock speed determined the speed at which instructions were executed, so the faster the clock speed, the better the resulting processor performance. To increase clock speed, the voltage of the power supply to the processor needed to be increased. This was not much of a problem, so the voltage was slowly increased to match performance increase stated by Moore's law, which observes that the number of transistors (therefore the processing power) in a processor should double every two years [Moo65]. Figure 1 below shows the rise of power input, clock speed and processor performance of Intel processors in the last 40 years or so.

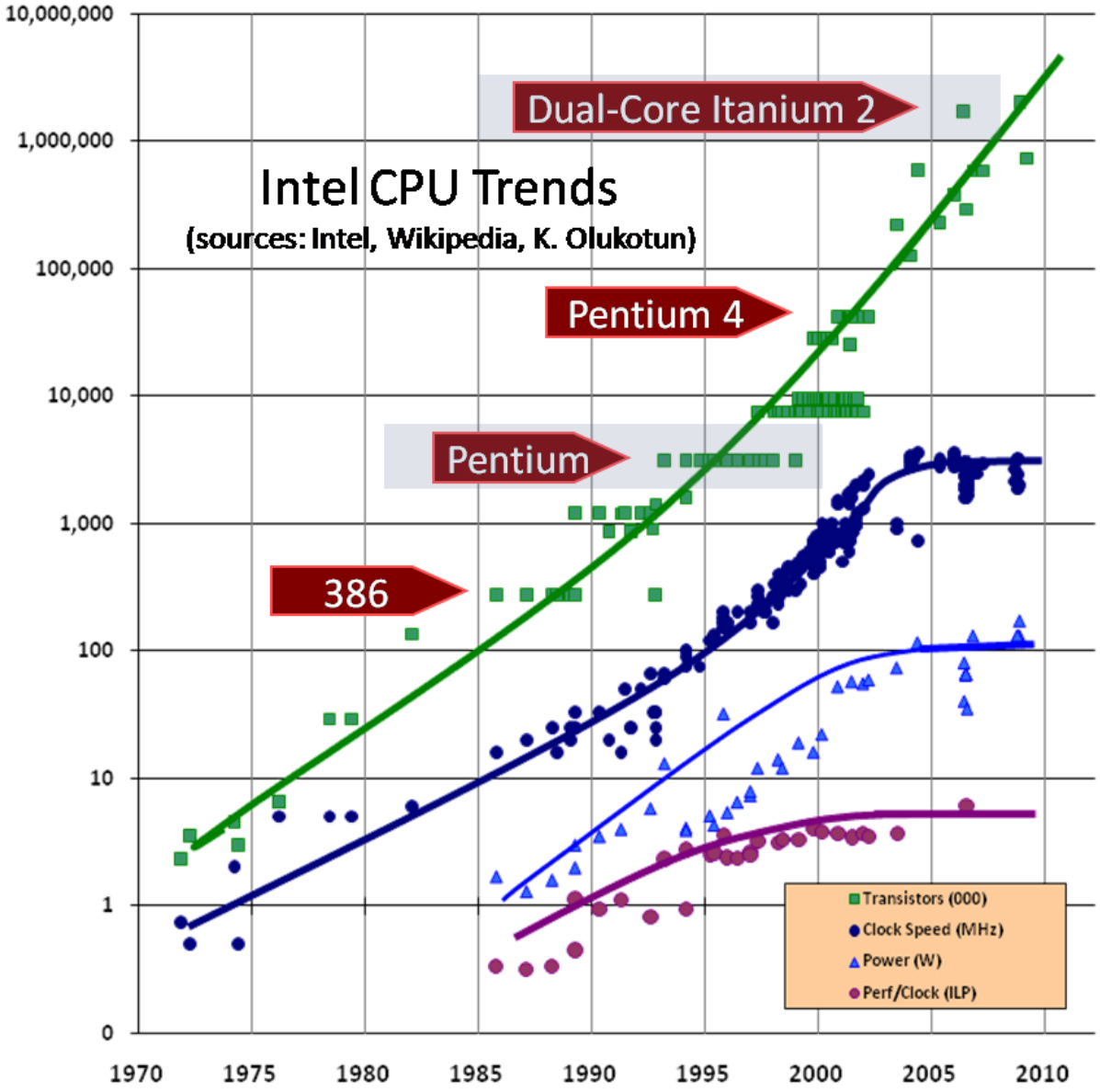


Figure 1: A subset of Intel CPU introductions between 1970 and 2010. [Sut05]

The number of transistors, shown by the green line, shows that processor performance is consistently growing at an exponential rate, which matches Moore's law. The growth of power input and clock speed are shown by the light and dark blue lines respectively. It can be seen that these values also rose exponentially, until about 2002 when they both seem to level out. This point indicates the beginning of the use of parallelism in general-purpose processors.

The problem with increasing the voltage input to the processor is that a by-product of power is heat. This was not a problem when the technique was originally applied, as cooling systems such as fans were sufficient to manage the heat output. However, as more and more power was required to increase clock speed further for each new build, the

processor was giving off more and more heat until a level of heat was reached that could not be sufficiently cooled and became a danger to the computer and its environment [Adv08]. This meant that processor performance matching Moore's law could not longer be attained by increasing clock speed; a new technique was required to continue growth of processing performance.

2.2 An introduction to parallel systems

This new technique was the use of parallelism within the processor. Operating on the principle that large problems can often be divided up into smaller, independent problems, parallel systems have more than one processing core that can each carry out an instruction on different pieces of data concurrently [Got89]. With each instruction executed in parallel to another, the execution time of the instruction is 'hidden' meaning the overall runtime of the program is reduced compared to serial execution. This is the basis on which parallel systems achieve improved performance, without relying on an increase in clock speed.

Of course, this technique was not particularly new. It was used prior to 2002 for many years in the high-performance computing industry, where it was used in massively parallel supercomputers which executed large-scale calculations such as molecular dynamics and climate modelling. These machines were very specialist and not general-purpose, but their parallelism worked very well. This meant that when the time came, the technique could be taken and applied to general-purpose systems to give increased processing performance on machines that were available for use by the general public.

Most general-purpose parallel systems use a SIMD (single instruction, multiple data) architecture, another classification of processor defined in Flynn's Taxonomy [Fly72]. This architecture works in a way that is best explained using a simple example of a calculation that it can perform. Imagine a 4×4 matrix of random integers, and a program that aims to find the product of these integers. A SISD architecture, which uses one core and one block of memory, would execute fifteen serial multiplications to find the product of the 4×4 matrix. However for a SIMD architecture with four cores, each with its own memory block, the 4×4 matrix can be split into four 4×1 matrices, one for each core in the processor. Parallel execution of three serial multiplications by each core can then be carried out, resulting in four partial product values which are then in turn multiplied by the master core to get the final value. Assuming each multiplication takes one clock cycle, the SISD architecture can complete the calculation in fifteen clock cycles, whereas the SIMD architecture can theoretically complete the calculation in six clock cycles. I say theoretically because parallel architectures are not as simple as serial architectures, and come with some overheads that have not yet been discussed. The main overhead associated with parallel architectures is communication between cores, which is the main focus of this paper.

2.3 An introduction to interconnects

Interconnects provide communication channels between cores working concurrently, and are an integral part of a parallel architecture. Let us look at the 4×4 matrix multiplication example again, this time in more detail.

The program starts off in the same way that a serial program would start, running on the master core only. Here, random integers are assigned to each position in the 4×4 matrix, arbitrarily in the case of this example. Once the matrix has been filled, the parallel section of the program can begin. The matrix is split up into rows, giving four 4×1 matrices. However for these matrices to be distributed between the four available cores, they need to be passed from the master core via an interconnect. The interconnect provides a path for data between cores, meaning the first 4×1 matrix can be passed from the master core to core[1], the second 4×1 matrix can be passed to core[2] and the third 4×1 matrix can be passed to core[3]. The fourth 4×1 matrix does not need to be passed to a different core, it remains on the master core (core[0]). Once all data has been distributed between cores, parallel processing begins as each core calculates the product of its section of the 4×4 matrix, resulting in four partial product values; one on each core. Again, interconnects are needed here to transport each partial product back to the master core, so that the final multiplication can take place. This is the end of the parallel section, and the final multiplication is done in serial resulting in the returned value of the program.

This is a very simplified example of the type of calculation parallel systems are used for, but it gives a good indication of how important interconnects are to the functionality of a SIMD architecture. However their use is not without its drawbacks, mainly the associated overheads that it adds to the system using it. For example in the program described above, extra computation is required by the system to specify a destination for each 4×1 matrix, to distribute the list of instructions to be executed by each core, and to specify the destination for each of the partial products. Another large overhead is the time taken to physically transport data across an interconnect wire. The speed of data transportation is restricted by the speed of light, roughly 3.0×10^8 m/s, and while this may seem more than sufficient for transportation rates of negligible time, let us consider the speed of modern processor clock speeds. An average clock speed in 2015 is around 2 GHz, which is equivalent to 2.0×10^9 clock cycles per second. So in one clock cycle, light travels roughly 15cm. This is not a very large distance, especially in larger scale supercomputers when interconnects may need to stretch tens or hundreds of metres. This suggests that, given the speed of modern processors, the majority of runtime in a parallel program is spent on data transfer rather than actual computation, which means that efforts to speed up the service that interconnects provide are essential for providing a further improvement of parallel processing performance.

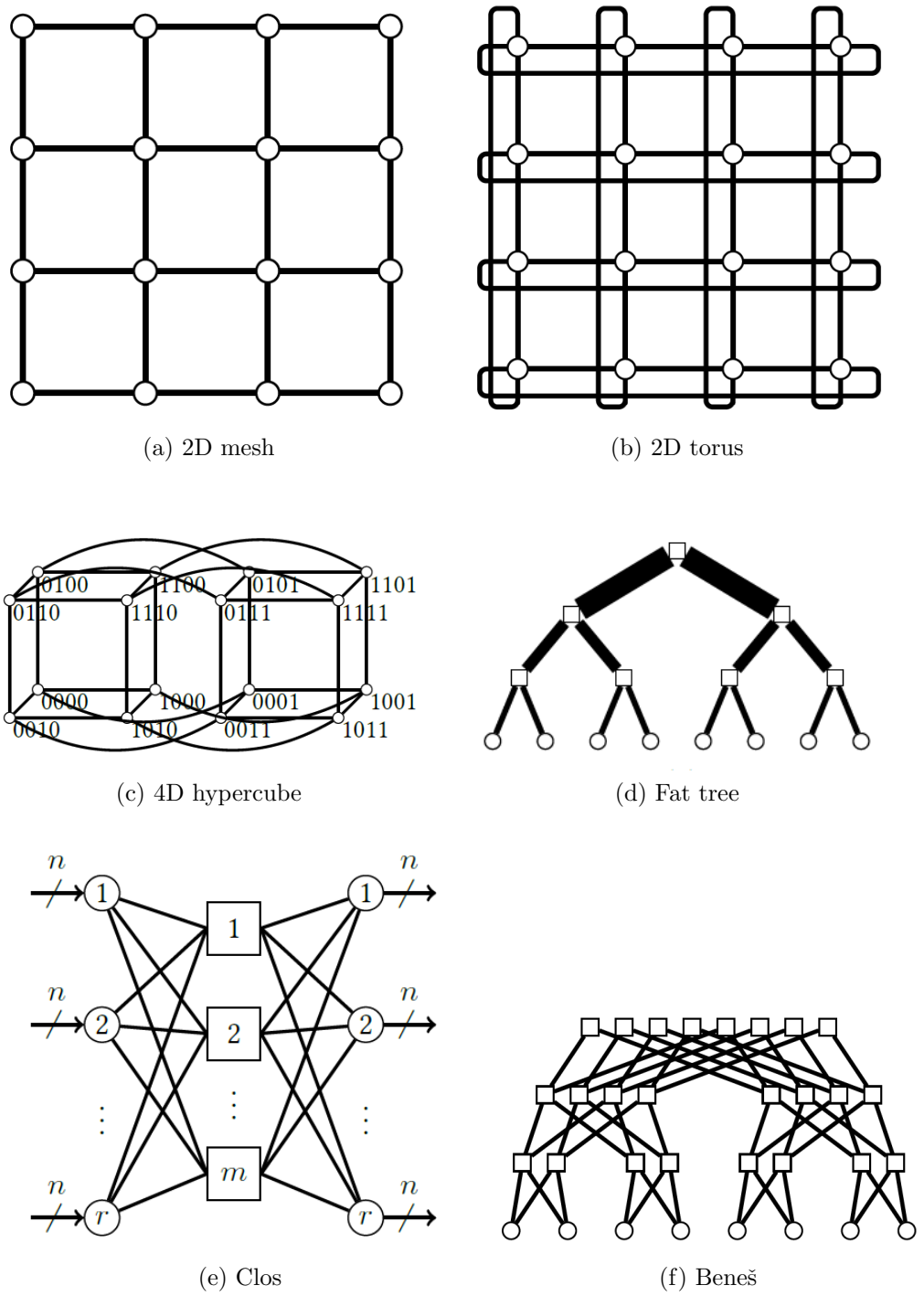
2.4 Interconnection networks

An interconnection network is built up when a number of interconnects are used in a parallel architecture. In an ideal world, each core would be connected to every other core by a single interconnect, giving direct communication between all cores in a system. However in practice this is not feasible, as for every nth core added to the network, $n - 1$ interconnects must also be added to maintain complete connection. This means that the number of interconnects scales by n^2 the number of cores, making this implementation non-scalable and not an option.

Instead, a sparsely-connected network must be used where each core is connected to a small number of other cores. As long as every core is reachable by every other core, which the network model requires, communication can then take place either directly between connected cores, or indirectly using more than one interconnect via intermediate cores. Switches can also be used in an interconnection network as intermediate nodes, which direct data down different paths in the network. These will be described in more detail later.

2.5 Network topologies

A topology represents an abstracted layout of a network, in terms of cores/switches (nodes) and interconnects (edges). There are many different topologies that can be used, with even modern supercomputer manufacturers disagreeing about which one gives best performance [Han14]. There are five main topologies: mesh, toroidal, hypercubic, tree and Clos.



(a) 2D mesh

(b) 2D torus

(c) 4D hypercube

(d) Fat tree

(e) Clos

(f) Beneš

Figure 2: Different network topologies used in parallel architectures. [Han14]

Mesh and toroidal networks have a regular layout similar to arrays. Cores in mesh networks are only connected to adjacent positions in the array, with the cores in the middle of the array having more connections than the cores at the edge. Toroidal networks remove this non-uniformity, allowing cores at the edge of the array to 'wrap around' the array to

connect to the adjacent node at the other end of the dimension.

Hypercubic networks have a symmetric and recursive structure. A $d + 1$ dimensional hypercubic network can be constructed by connecting two d dimensional hypercubes by their matching cores.

Tree networks have a root node connecting one or more child nodes, which in turn connect one or more child nodes until all nodes are connected. The fact that all nodes have the common root node means that every node is reachable, which is a requirement of a network topology. However because there is only one root node, traffic around this node can get quite busy when lots of communication is taking place. Fat trees address this problem, giving interconnects closer to the root of the tree more bandwidth so that more data can be transmitted on each clock cycle.

Clos networks were originally designed for use in telecommunications [Clo53]. They use non-blocking switches to allow continuous transfer of data over multiple connections. Each connection has three stages: an ingress stage where data is sent from a core, a middle stage where data passes through one or more switching nodes, and an egress stage where data is received by the receiving core. The middle stage can be made up of further Clos networks, making a chain structure which gives more paths for each connecting nodes. Beneš networks are a type of Clos network, where each node is connected to exactly two other nodes [Ben65]. Due to the symmetry of Clos and Beneš networks, they can be folded across the middle stage as shown in figure 2f. This merges the ingress and egress stages by using bidirectional communications links, allowing more efficient traversal of the network.

2.6 Routing

A routing algorithm decides which path of a network the data is sent along during communication between two nodes. Ideally the shortest path between the two nodes is selected, however consideration of the state of the network should be taken into account to try and minimise link contention within the network. There are a number of different routing algorithms used in parallel systems, a selection of which are described below.

2.6.1 Oblivious routing

Oblivious routing ignores the state of the network when deciding which path should be chosen for communication. This means that it typically chooses the shortest path between two nodes, which can result in very poor use of the network in the worst case. However the worst case is an unusual scenario, and in the average case this algorithm can perform quite well with minimal overheads slowing down communication.

2.6.2 Adaptive routing

Adaptive routing aims to distribute communication paths across the network more evenly [Dal03]. Each node uses the buffer occupancy in the nodes connected to it to choose the path with the least traffic, allowing data to be transmitted down the least congested path. However this technique is open to the risk of livelock, where the chosen path goes around in circles and the data does not progress towards its destination. This is obviously a problem, as data is not guaranteed to reach its destination in an acceptable time for efficient communication to take place.

2.6.3 Two-phase randomised routing

Two-phase randomised routing is another technique used to spread traffic uniformly over the network [Val90]. It selects a random intermediate node somewhere in the network, and then ensures that data passes through this node on its way to its intended destination. As the name suggests, communication is split into two phases: sending the data to the intermediate node, and forwarding the data on to the destination node. Both of these phases use oblivious routing, meaning that data generally travels twice the average path length, but the addition of the random element tends to distribute traffic evenly over the network and improves communication efficiency.

2.7 Switching

A switching scheme handles the way in which network resources are used during communication. Switches are included in a network as intermediate nodes that forward data along a specified route to its destination node. There are a number of different switching schemes used in parallel systems, a selection of which are described below.

2.7.1 Packet switching

Packet switching is a very commonly used switching scheme. It splits the piece of data to be transmitted up into fixed length parts called packets, and then each of these packets is transmitted independently of one another to their destination. Once all the packets have reached their destination, the data is then reassembled and reconstructed ready for use by the node. The fact that packets are send independently means that in some situations they can be sent concurrently down different paths of the network, improving communication efficiency.

2.7.2 Circuit switching

Circuit switching reserves a complete path from source node to destination node across the network before communication has begun, meaning that the links in the reserved path cannot be used by any other communication before the initial communication is complete. This technique reduces the number of decisions made during communication regarding the state of the network, however restricting certain links in the network from being used can result in a waste of available bandwidth, making communication sometimes inefficient.

2.7.3 Wormhole switching

Wormhole switching is a similar technique to packet switching, but goes a step further by dividing up packets again before transmission. Each packet is split up into a sequence of flits, which are then transmitted in the same manner as packet switching. However due to the smaller size of each flit, each packet is pipelined in the network reducing latency and buffer requirement. This can important in general-purpose systems where buffer size can be chosen independently of potentially differing packet sizes.

2.8 Flow control

Flow control algorithms deal with the allocation of network resources, such as buffer space and bandwidth used by each data packet [Jon97]. This allows the network to avoid buffer overflow, and increases the efficiency of communication. There are two main flow control schemes: credit-based flow control, and virtual channel flow control.

2.8.1 Credit-based control flow

During communication, credit-based control flow only allows transmission on a link if there is sufficient buffer space in the next node on the route. Each node has a counter of available buffer space on its input, and if the count value indicates that the next packet would cause a buffer flow, transmission is blocked. When there is a lot of traffic on the network, this technique can cause a backup of traffic where there are no suitable routes for a packet to be sent down, so the packet must wait until more buffer space in the next node becomes available. In very heavy traffic, this can halt communication altogether as no more packets can be injected into the network.

2.8.2 Virtual channel control flow

Virtual channels can be implemented in a network to avoid the problem of potential blockages which halt data transmission across a certain link [Dal92]. A virtual channel consists

of a buffer and associated state information, and multiple virtual channels can be allocated to a single link in the network [Dal87]. The virtual channels provide multiple buffer spaces for packets coming into the node, and when data in a channel is ready to progress, it can be assigned access to the physical link and communication can be continued. The addition of virtual channels to a link is analogous to adding new lanes to a motorway; if a car breaks down on one of the lanes, traffic can still pass on the other available lanes. In the same way, if data stored in the buffer of one of the virtual channels is unable to progress through the network for a certain reason, the other virtual channels can be used by data packets which are able to progress.

2.9 Modern interconnection networks

The final part of this section looks at two examples of modern interconnection networks used in general-purpose parallel systems. The two examples in question are InfiniBand and the IBM Blue Gene/L 3D torus network.

2.9.1 InfiniBand

InfiniBand is an industry-wide networking standard developed in 2000 by a consortium of companies belonging to the InfiniBand Trade Association [Hen11]. It is used predominantly in massively parallel supercomputers, where its flexibility regarding network topology and use of routing algorithms are popular with manufacturing companies which like to have control over the schemes used in their products. InfiniBand uses a switch-based interconnect that supports data transmission speeds of 2 to 120 Gbp/link, where each link can reach a length of up to 300m. It uses cut-through switching, a variation of packet switching, 16 virtual channels per link, and credit-based control flow. This allows data of up to 512KB to be transmitted between cores as a single message, however approximately 90% of messages are less than 512 bytes meaning the maximum message length is rarely transmitted.

2.9.2 IBM Blue Gene/L 3D Torus Network

In 2005, the IBM Blue Gene/L was the largest, most powerful supercomputer in the world, according to www.top500.org [Hen11]. It used a 32x32x64 3D torus interconnection network, which connected all of its 64000 nodes. When communication between two nodes was required, either node could be used for communication protocol processing while the other may have been carrying out other computation. Packet sizes ranged from 32 bytes to 256 bytes, with an 8 byte header containing routing, virtual channel and link-level flow control, as well as a 1 byte CRC for packet validation. Virtual cut-through switching and credit-based control flow were again used, along with various other schemes which provided the low latency and high bandwidth requirements of a machine designed to execute computation as quickly and efficiently as possible.

3 Specification

3.1 Outline of the solution

The chosen solutions builds on the premise outlined in the second paragraph of section 1.3, and focuses on the problem of collisions in the network during communication between parallel processors. The solution looks to reduce the number of collisions between pieces of communication data travelling through the network compared to the number of collisions that existing solutions such as two-phase randomised routing encounter during computation. As its primary use will be as a research and demonstration tool, only an emulation of each hardware components in the network is required meaning that the solution will be produced as a software product. The emulated hardware components can be split into three sections: the processors, the network, and the routing algorithm.

The emulated processors will be able to read in a parallel program, distribute the relevant sections of the program to their intended processors, and then execute the program until the intended computation described in the program has been carried out. There are two main aspects that the emulated processors must include. The first aspect is that the processors must be able to handle input and output instructions with regard to sending and receiving communication data over the network. The second aspect is that the processors must execute their instructions as if they are running concurrently. The fact that the solution is a software product means that this parallel execution can be emulated, but each processor must execute their instructions as if it was running in parallel with the others. In terms of the instruction set for the processors, only basic functionality is required for this solution. The network should remain universal, which means that any processor design should be able to be connected to it and still work correctly, so as long as network input and output can be handled correctly, the instruction set is not critical and can be kept relatively basic.

The emulated network will facilitate communication between the emulated processors. It will provide links between all processors in the network so that any processor and communicate with any other processor. However, due to the poor scalability of a completely connected network, the emulated network will contain a number of intermediate switch nodes which will direct communication data towards its intended destination in stages rather than using a direct link. There are two main qualities that the network must maintain. The first quality is that the network should be universal, which means it has the ability to emulate the behaviour of any other network topology using its own topology. The second quality is that the network must be scalable, which means that it should be able to include any number of processors and still carry out communication between them correctly.

The routing algorithm will decide which paths through the network the communication data will take in order to reach its intended destination. This is the part of the solution that aims to reduce the number of collisions within the network, and increase the efficiency of parallel architectures. When a processor executes an output instruction indicating that

communication across the network is required, the routing algorithm will be triggered. It will start off by planning a route through the network for the communication data aiming to avoid as many collisions with other data in the network as possible. Once a route has been decided, the algorithm will pass this route to the emulated network where the communication data will be sent on its calculated path towards its destination. The algorithm will carry out this process for all output instructions read by each processor until the parallel computation is complete.

3.2 Functional requirements

The solution must observe the following functional requirements:

1.1 A single parallel program must be taken as input.

1.2 The computation outlined in the input program must be spread across all of the processors in the network as intended by the input program.

1.3 Each serial processor must be able to execute basic instructions outlined in the input program including arithmetic and storing and editing variables. This basic functionality is required to allow for the possibility of communication in the network.

1.4 Parallel running of the processors must be emulated. This means, for example, that the first instruction that each of the processors is set to carry out must be executed before any of the processors can move onto their second instruction.

1.5 The processors must be able to recognise network output instructions included in the input program.

1.6 When a network output instruction is read by a processor, details of the communication – including the source address, the destination address and the data to be sent – must be passed to the routing algorithm.

1.7 The routing algorithm must find a path through the network for the provided communication data, ensuring that the path ends at the destination address.

1.8 Once a route has been found, the routing algorithm must pass the route path and the data to be transmitted to the network.

1.9 The network must pass communication data towards its destination according to the route path that was provided by the routing algorithm.

1.10 Parallel running of the network must be emulated. This means, for example, that each of the pieces of communication data in the network must complete one stage of their transmission before any of them can progress to another stage of their transmission. The exception to this rule is when a collision is detected, as described in requirement 1.11.

1.11 When a collision is detected between two pieces of communication data on the network, one of the pieces of data must wait for the other to vacate the link before it can continue its transmission through the network.

1.12 The network must keep a count of the number of collisions that take place on the network during execution of the input program.

1.13 The network must measure the emulated time that was taken by the network to execute the input program.

1.14 The emulation must be terminated gracefully once execution of the input program is complete.

3.3 Interface requirements

The interface of a software application is dependent on the users of the software. In this case, the software will primarily be used as a research and demonstration tool which means that the users are assumed to have at least a very basic understanding of the Linux command line. The nature of the solution suggests that users will be comfortable without a graphical user interface (GUI) – as long as a precise user manual is provided – in order to execute the software from the Linux command line and read the results it produces. With these user characteristics in mind, the solution must observe the following interface requirements:

2.1 The solution must be initialised and executed using exactly one command in the Linux command line, with the input program file being passed as a parameter.

2.2 The number of collisions in the network during the emulation must be printed to standard output once the input program has been fully executed.

2.3 The time taken for the emulation to fully execute the input program must be printed to standard output.

2.4 There must be an option that allows all of the routes calculated by the routing algorithm to be printed to standard output as the emulation is being carried out for testing and demonstration purposes.

2.5 There must be an option that allows a description of the full state of the network, including the locations of all of the communication data currently passing through the network, to be printed to standard output at a given time for vigorous testing and demonstration purposes.

2.6 Any intermediate files required by the emulation during execution of the input program must be deleted automatically in order to keep the user's file system tidy.

3.4 Performance requirements

The emulated parallelism present in the solution means that execution of the input program will be significantly slower than if it was executed on a real parallel architecture. This is not a serious problem as the solution is not performance critical, however there must be some sensible limits put in place which govern the speed at which the emulation executes the input program. These performance requirements are listed below:

3.1 The emulation must be able to execute basic input programs (less than one hundred lines of code) within one second.

3.2 The emulation must be able to execute medium-length input programs (less than one thousand lines of code) within ten seconds.

3.3 The emulation must be able to execute large input programs (more than one thousand lines of code) within one minute.

3.4 Allowances can be made for very large input programs (more than ten thousand lines of code), but they must still be fully executed by the emulation given sufficient time.

3.5 Software system requirements

All pieces of software require some general guidelines regarding reliability, availability, security, maintainability and portability to ensure that they can be used correctly on any suitable machine. Requirements for these five areas are listed below:

4.1 The emulation must compile cleanly without any errors or warnings, provided that a syntactically correct input program is entered as a valid parameter.

4.2 Regardless of any other output, the emulation must conclude the standard output with a clear indication of the number of collisions in the network during execution of the input program and the time taken for the network to execute the input program.

4.3 The source code of emulator must be open source, allowing universal access to its research and demonstration capabilities.

4.4 The instruction set of the emulated processors used in the network must not contain any instructions that make permanent changes (other than relevant file output) to the machine that the emulation is being run on, as these instructions could potentially be used in a malicious manner.

4.5 Any files output by the emulation must not be potentially harmful to the machine they are situated on, even if opened accidentally by the user with an incorrect program.

4.6 The process of changing the output of the emulation, to document all of the routes calculated by the routing algorithm for example, must be quick and easy taking less than 10 seconds.

4.7 The number of emulated processors that are included in the network must be quick and easy to change, taking less than 10 seconds.

4.8 Due to the academic nature of the solution, the source code of the emulator must be well structured and well commented so that it can be inspected and understood by any users curious about how the emulation has been created.

4.9 The emulator must compile and run using only software included in a base Linux operating system setup (eg. gcc, make). This ensures that no specialist or proprietary software is required to use the emulator, increasing portability between machines.

3.6 Constraints, assumptions and dependencies

In the requirements above, some assumptions have been made about the machine that the user will be using the solution on. These assumptions have been based on an average low-end laptop available in 2015, which includes an Intel Core i3 processor, 4GB RAM and at least 10GB available hard disk space. Assuming that the solution will be used on a low-end machine means that it can be used on both low-end and high-end machines, increasing portability. The machine is also assumed to be running a Linux operating system, Ubuntu 14.04 for example, which is a reasonable assumption as Linux operating systems are open source and freely available.

A possible constraint that the solution must negotiate is the amount of memory that it uses. Each of the emulated processors needs its own block of memory in order to carry out the instructions described by the input program. This may become a problem when a large number of processors are included in the network – each with their own block of memory – and there is not enough memory available for allocation. In this situation a compromise must be made: either the amount of memory allocated to each processor can be reduced, potentially limiting the size of the input program that the processor can execute, or the number of processors included in the network can be reduced. However, this scenario only has to be considered in extreme circumstances and will not affect the demonstration purposes of the emulation.

One area that has not been explained so far is how the input program code is compiled so that it can be read by the emulated processors. Creating a bespoke compiler is beyond the scope of the thesis, so the emulation is dependent on an open source compiler being available for use. In order to be compatible with the solution, the compiler must allow for basic serial instructions as well as indicating the distribution of the input program between the emulated processors in the network.

4 Design

4.1 Network topology design

The two main requirements of the interconnection network to be used in the emulation are that it should be universal and it should be scalable. Taking into account the characteristics of the different network topologies explored in section 2.5, it was decided that a variation of the Beneš network was the most suitable choice of topology to build the network with.

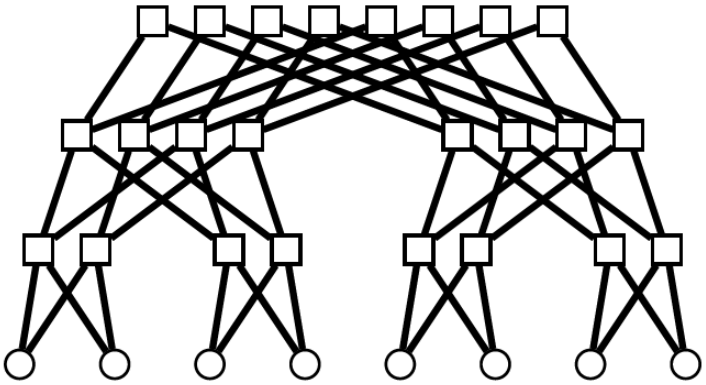


Figure 3: A folded Beneš network connecting eight processors. [Han14]

Figure 3 shows an example of the layout of nodes and edges in a folded Beneš network. The eight circular nodes at the bottom of the network represent eight processors which use the network for communication. Coming out from each of the processors are two interconnects which link the processors to the first layer of intermediate nodes in the network. These links – and all of the others in the network – are bidirectional which means that communication data can travel either towards the processors or away from them. Each of the intermediate square nodes in figure 3 represents a 2×2 switch, the design of which is described in more detail in section 4.2. Where p is the number of processors connected to the network, there are $\log p$ layers of p switches culminating in a partially connected network, which means that – using packet switching and credit-based control flow – any processor in the network can communicate with any other processor in the network.

Beneš networks are rearrangeable and provide non-blocking communication between processors. In terms of universality, the number of intermediate switch nodes in the network means that the network could take the physical form of the other topologies discusses in section 2.5, and emulate the behaviour of them. In terms of scalability, the number of switch layers included in the network is dependent on the number of connected processors. This means that, provided the number of switch layers is correctly updated, that an arbitrary number of processors can be connected to the network and it will still function correctly. In the situation where the number of processors is not a power of two, non-functioning placeholders can be used to bring the number of source nodes up to the next power of two

without affecting the behaviour of the network. This means that Beneš networks are both universal and scalable, making them ideal for use in the emulation.

4.2 Network switch design

In a folded Beneš network, each switch is connected to two core-facing links (which are located closer to the processors), and two edge-facing links (which are located further away from the processors). This means that the switches are referred to as 2×2 switches, or crossbar switches. A diagram of how communication data can pass through these switches when being transmitted through the network is displayed below.

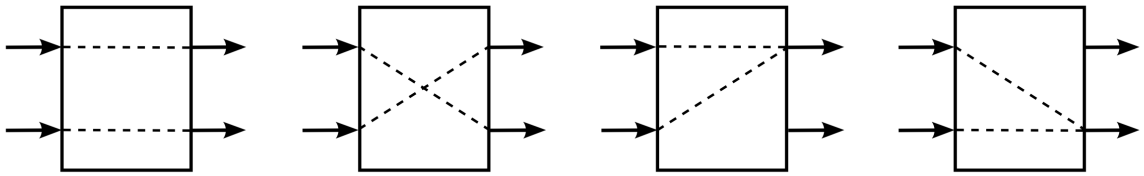


Figure 4: The possible permutations of data flow through a one-way crossbar switch.

Figure 4 shows the four possible scenarios of how concurrent communication data can be handled by crossbar switches which use unidirectional input and output links. However, in the case of folded Beneš networks, bidirectional links are used which means that each switch has four input/output opportunities at a given time. This could equate to four packets being sent from the switch, four packets being received by the switch, or a mixture of the two.

In the case of the two rightmost permutations in figure 4, the two input packets are shown to require the same output link to continue their transmission. This results in a collision in the network, as both packets cannot use the link at the same time. For this scenario, buffer space can be allocated to each output link of the switch so that communication data involved in the collision can be stored inside the switch until the link becomes free again and transmission of the stored communication data can continue. This means that data packets that collide do not have to be re-sent completely by their source processor, saving time and therefore increasing the efficiency of the network.

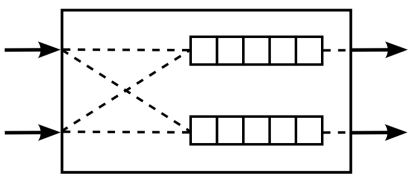


Figure 5: A one-way crossbar switch with a 5-space buffer for each output link.

Figure 5 shows how buffer space can be used in a one-way crossbar switch. In the case of folded Beneš networks, there are four output links meaning there are four buffers included inside the switch.

4.3 Routing algorithm design

The choice of routing algorithm used to determine which paths the communication data takes through the network is essential for reducing the number of collisions that occur. After considering many possibilities, an algorithm was chosen that allows parallel routing of both full and partial permutations through the network where, for a single permutation at least, it is *guaranteed* that no collisions will occur while the data is being transmitted.

To describe how this algorithm carries out collision-free permutation routing, its application on a slightly modified Beneš network, shown in figure 6, is discussed. The same principles of the algorithm can then be applied to the folded Beneš network described in section 4.1.

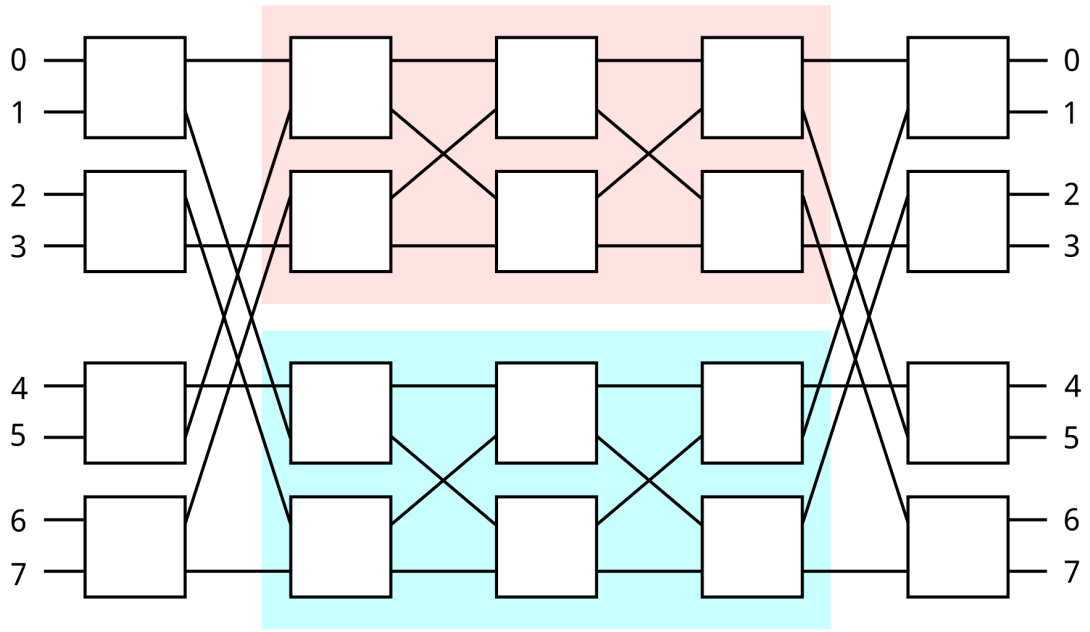


Figure 6: A slightly modified Beneš network with two sections highlighted.

The Beneš network shown in figure 6 can be thought of as the unidirectional equivalent of the folded Beneš network described in section 4.1 but with a slightly different link ordering. The numbers on the left of the diagram represent the source processors that send communication data packets, and the numbers on the right of the diagram represent the destination processors that the communication data is being sent to. An important feature

of Beneš networks is specifically highlighted by the pink and blue sections in figure 6. It can be seen that once a packet of communication data has entered one of these sections, it remains in that section until the end of the section is reached. This feature is important for the routing algorithm described below. It is also noteworthy that each of the pink and blue sub-networks is in fact its own Beneš network – a fact that can be extended to every layer of switches in the network – meaning that the network is simply a recursion of itself.

The routing algorithm that can be used to carry out collision-free permutation routing on the network shown in figure 6 is described by the following:

1. Take an empty, partial, or full permutation as input.

2. Take the i^{th} communication from the permutation and trace a front path from the source processor to the first switch it encounters in the network. If the switch has not previously handled any communication data for this permutation, set the packet route towards the first output link in the switch. If the switch has already handled a packet of communication data, set the packet route towards the second output link in the switch.

3. For the same communication, trace a back path from the destination processor to the first switch it encounters on the network (in reality, the last switch that the packet will pass through during communication). Once inside the switch, set the route of the back path to mirror the direction that the front path was sent towards. This will result in both paths being directed into the same closed sub-network as shown by the highlighted sections in figure 6.

4. Repeat steps 2-3 for every communication in the permutation to calculate the first pairs of front and back path routes. Because each switch takes a maximum of two input packets, no collisions can occur as each packet is routed a towards a different output link by definition.

5. Repeat steps 2-4, but this time continuing the front and back paths for each communication towards the centre of the network one layer of switches at a time.

6. Once the front and back paths reach the same switch in the centre layer of the network, the route is complete.

By following this iterative algorithm, routes through the network can be built up step by step until complete routes for the whole permutation are calculated. Not only does this ensure that each packet of communication data can be routed from the correct source processor to the correct destination processor, it also ensures that there are no collisions of communication data within the network. An example of the routes calculated from an arbitrary full permutation using this algorithm is shown below in figure 7.

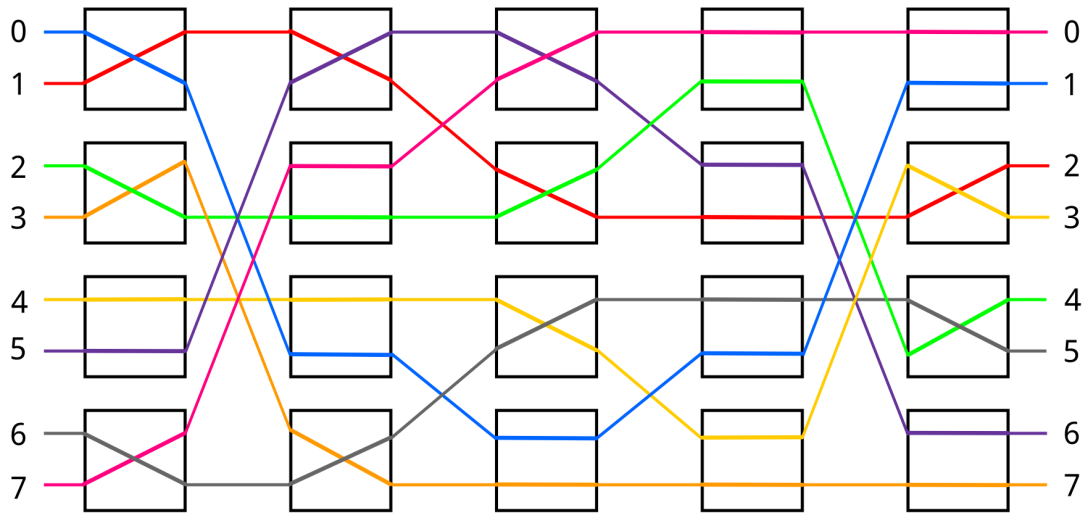


Figure 7: Routing a full permutation.

Until now, the workings of the algorithm have be described using a slightly modified Beneš network as a reference. This has been done for clarity. It is easier to demonstrate routing on unidirectional links in an unfolded Beneš network, but the same principles apply on a folded Beneš network with bidirectional links. In a single permutation, all of the packets of communication data are at the same switch layer at a time, meaning that no additional collisions can occur when the packets reach the outermost switch layer and the direction of transmission to changed to point towards the destination processors. Additionally, it is easier to demonstrate that the Beneš network is split into distinct sections after each layer of switches (as shown by the pink and blue highlighting in figure 6) using a slightly modified link layout between switch layers in the network. This modified layout is essentially a mirror image of the layout to be used in the emulation, which means that these distinct sections after each layer of switches still exist, but they are not as easy to split into clear physical groupings as those shown in figure 6. This difference in groupings has no affect on the algorithm.

All of this shows that the algorithm described above is able to calculate collision-free routes through a folded Beneš network, making it ideal for use in the emulation.

4.4 Optimising the routing algorithm

The algorithm described in section 4.3 is an optimal collision-free routing algorithm for the Beneš network described in figure 6. However, the fact that a folded Beneš network will be used in the emulation means that some optimisation techniques can applied to the routing algorithm that take advantage of this difference in network structure to make communication more efficient without affecting the lack of collisions in the network.

The first optimisation that can be applied to the routing algorithm involves the spread of data packets across the two sub-networks that follow the first layer of switches. As the folded Beneš network contains two output links for each processor, each sent packet has an initial choice of which of the sub-networks to travel across. This means that if the packets are sent to alternating sub-networks, spread of traffic over the network is more likely to be uniform and the possibility of collisions between packets from different permutations is reduced.

A second optimisation that can be applied to the routing algorithm takes into account the bidirectional links in the folded Beneš network. The routing shown in figure 7 implies that every pair of front and back paths must reach the centre layer of switches before the route is complete. However, in a folded Beneš network, the front and back paths may encounter the same switch before reaching the final layer of switches. In this situation, there is no need for the pair of paths to be continued to the final layer of switches, so a shorter completed route can be allocated. This shorter path means that the communication data travels quicker through the network, making communication more efficient.

4.5 Sourcing a suitable compiler

It was mentioned in section 3.6 that the functionality of the emulation is dependent on a compiler being sourced which can compile a parallel input program into instructions that are then distributed between the processors connected to the network and executed in parallel. During the network design phase, a solution to this problem was presented by David May, a Professor of Computer Science at the University of Bristol. David provided the source code of a compiler that is able to handle parallel code sections, and the source code of a serial processor that, with some modifications applied to it, could read and execute the instructions produced by the compiler. I am very grateful for David's contribution towards this thesis.

4.6 Output of the compiler

The compiler takes the source code of a parallel input program as input and produces binary file represented in hexadecimal as output, which is then available for the processors in the network to read and execute. In order to do this, the binary output file produced by the compiler must be understood so that the binary instructions it carries can be executed correctly by the processors in the network. An example of this binary output is given in figure 8 below.

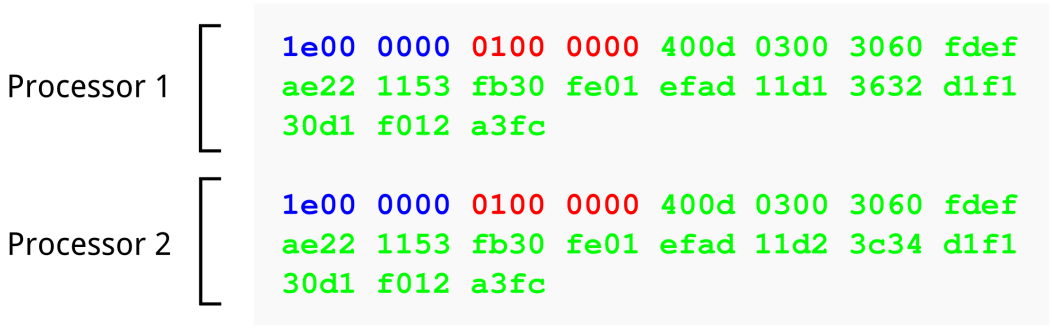


Figure 8: An example of binary output from the compiler, with added sectioning and colouring.

It can be seen that the binary for each processor is split into three sections. The first section, represented by the blue font colour in figure 8, indicates the length of the binary string relevant to each processor, measured in bytes. This gives an indication of the positions in the binary that instructions for one processor stop and instructions for another processor start, meaning it is vital for the distribution of the different parts of the parallel program to their correct processors ready for parallel execution. The second section in the binary for each processor, represented by the red font colour, indicates the initial value of the program counter that the processor uses during execution of the program. The third section, represented by the green font colour, indicates the instructions that the processor carries out in order to execute the program correctly. Each instruction is made up of 8-bits: a 4-bit operation code and 4-bits of immediate data. Larger data can be handled using inter-register operations.

This sequence of three binary sections is then repeated for every processor involved in the parallel input program, resulting in a full binary file ready to be executed by the emulation.

4.7 Processor modifications

The processor that was provided with the compiler source code is designed to execute most of the instructions described in the binary output file that the compiler produces. A summary of the instruction set of this processor is included as **Appendix B**. However, the processor was designed as a serial processor which means that it has no communication capabilities, and that instructions for the sending of communication data to the network and the receiving of communication data from the network needed to be added. Additionally, the provided source code for the processor only accounted for a single instance of the processor, whereas the network requires multiple processors to function correctly. Provided that these features are added, the processor source code is suitable for integration with the network and can be included in the emulation.

5 Implementation

The following chapter describes how the emulation was created. The programming language chosen to code the emulation was C due to its speed, efficiency and the ease at which memory address manipulation can be carried out. The code also uses only the standard C libraries, making it portable across virtually all machines running Linux.

5.1 Building the network

The network is made up of two main building blocks: 2×2 crossbar switches and interconnects which link these switches together. These entities were created as individual data structures that interact with each other to facilitate traversal through the network.

5.1.1 Link data structure

The links are used to carry packets of communication data across the network. Each link can contain a maximum of one packet of data at a time, which is the reason that collisions of data packets in the network occur. This means that the Link data structure consists of a pointer to the data packet that it is currently transmitting, and if the link is vacant the pointer is set to null. For reasons regarding the emulated parallelism in the network, a second pointer to a temporary data packet is also included in the Link data structure. The function of this additional pointer is described in section 5.5.

5.1.2 Switch data structure

The switches are used to direct the packets of communication data across the network towards their destination, following the route calculated for each packet by the routing algorithm. In a folded Beneš network, each switch connects four interconnects which can be used for both input and output of data packets to and from the switch. This means that the Switch data structure contains four instances of the Link data structure, which allow packets of communication data to move across the network.

In the situation where two packets of communication data try to use the same link at the same time during transmission across the network, a collision occurs. This means that one of the data packets must wait for the link to become free again before it can continue its transmission. To facilitate the queuing of data packets for each output link in the switch, four buffer spaces are needed, which means that four instances of the Buffer data structure are also included as part of the Switch data structure.

5.1.3 Buffer data structure

The buffer for each of the output links in a switch is implemented as a cyclic buffer. This

means that the Buffer data structure includes an array of data packets currently being stored in the buffer, a value indicating the index of the array that holds the next packet to be sent across the output link, a value indicating the index of the array where the next data packet to be input into the buffer should be placed, and a value that indicates the number of data packets currently being held in the buffer. The size of each buffer is variable, and is defined at the top of the code. The index values are variable, and change every time an input or output occurs. The count value can be used to indicate whether the buffer is empty or full; if the buffer is empty then it can simply be ignored, and if the buffer is full then it cannot accept any more data packets meaning that any additional data packets that require use of the output link that the buffer supplies must wait in the input link to the switch until space becomes available in the buffer. This functionality is referred to as credit-based flow control.

5.1.4 Forming the network

The main frame for the network is built up using these three data structures. The construction of this frame can be described in layers, where n is the number of processors that the network will connect. Firstly, a bottom layer consisting of $2n$ links is initialised. These links will be used to connect the processors to the first layer of switches. Next, a layer consisting of n switches is initialised and placed above the bottom layer of links. Each of the switches in this layer is then connected to two of the links in the layer below. Another layer consisting of a further $2n$ links is then initialised and placed above the first layer of switches, where each of the switches is connected to a further two links in the layer above. This layering process is continued until there are $\log(n)$ layers of switches connected to $\log(n)$ layers of links and a folded Beneš network as described in section 4.1 is formed.

5.2 Creating the processors

It was described in section 4.7 that an emulated processor was provided that can read and interpret the binary output of the provided compiler. A summary of the instruction set of this processor is included as **Appendix B**. However, to make this processor suitable for use in the emulation, some modifications to the processor were required. These modifications included adding input and output functionality which allows the processor to send and receive packets of communication data from the network, and adding the ability for multiple instances of the processor to be created so that more than one processor can be connected to the network.

5.2.1 Adding the output instruction

An output instruction is given by the compiler as the binary characters 0xF, which represents an operation, and 0x11 which represents an output operation. Directly before an output instruction, the processor takes the destination address of the packet of communication data to be sent, as well as the data itself, and stores this data in the processor's

registers. When the following output instruction is read, the processor informs the routing algorithm that a calculated route for a packet of communication data is required, and it provides the routing algorithm with the details of the destination address and the data to be sent stored in the registers. Once this information has been passed on, the processor can immediately carry on executing its next instructions because the network is a non-blocking network. This means that computation can be carried out while packets are being sent across the network, which is part of a technique for increasing program efficiency called latency hiding.

5.2.2 Adding the input instruction

An input instruction is given by the compiler as the binary characters 0xF, which represents an operation, and 0x10 which represents an input operation. Directly before an input instruction, the processor takes the port number where the received packet will be placed, and the memory address of the variable that will be set as the data provided by the received packet. When the following input instruction is read, the processor will check with the network adapter (described in section 5.3) whether a packet has been received at the specified port address. If the packet has been received, the processor takes the data provided by the packet and stores it in memory under the stated variable name, before continuing execution of its next instructions. If the packet has not been received at the specified port address, it is still being transmitted across the network. In this situation, the processor must wait until the packet is delivered before it can continue execution of its next instructions, which means that the input instruction is read over and over until the packet is delivered and the data it provides can be extracted and stored.

5.2.3 Creating multiple processors

Currently the processor works as a single entity. However, the network requires a variable number of processors to be connected to carry out parallel computation, which means that multiple instances of the processor are needed. To emulate this, arrays of the registers for each processor are initialised. An array of the memory blocks for each processor is also initialised, where each processor's individual set of instructions described in the binary file produced by the compiler are stored. This means that each processor has its own set of instructions stored in its own memory block, and can execute these instructions using its own registers, which indicates that the multiple processors are fully independent of one another and are suitable for connection to the network.

5.3 Connecting the processors to the network

Now that multiple emulated processors are available, they can be connected to the bottom layer of links described in section 5.1 which joins them to the network. To do this, a data structure is needed that acts as a network adapter. This allows packets to be added to the network when a processor needs to send communication data, and packets to be removed

from the network once the packet has been received by its destination processor.

5.3.1 Network Adapter data structure

The Network Adapter data structure consists of two Link data structures which connect the network adapter to the first layer of switches in the network, a port for next output packet to be added to the network, and an array of numbered ports where input packets can be stored until they are collected by the destination processor when an input instruction is read.

When an output instruction is read by a processor, the network adapter waits to receive a packet of communication data from the routing algorithm. Once this packet is received, it is added to the output port of the network adapter. The packet has now been introduced into the network, where it can travel along the path calculated by the routing algorithm towards its destination.

When a packet reaches its destination processor, it is removed from the network and stored in the specified port number in the network adapter. It remains stored in the port until an input instruction is read by the destination processor that requires the data inside the packet stored in that port. When this occurs, the data is removed from the port and used in the program currently being executed. If an input instruction is read by a processor which refers to an empty input port, the processor must wait until the required data arrives.

5.4 Integrating the routing algorithm

The routing algorithm is triggered when one of the processors reads an output instruction, which indicates that a calculated route though the network is required for a packet of communication data. Before calculating the route, the routing algorithm scans all of the other processors in the network for any other parallel output communications that are required. Using the source and destination addresses for all of the gathered communications, a permutation is formed which is then used by an implementation of the algorithm described in sections 4.3 and 4.4 to calculate collision-free routes through the network for each communication in the permutation.

Each calculated route is represented as three values: a count value, which indicates how many layers of switches the route traverses on its outward path before it changes direction and begins its inward path back towards the processors, a routing bit-string which indicates which of the output links connected to each switch encountered on the outward route are taken, and an addressing bit-string which indicates which of the output links connected to each switch encountered on the inward route are taken. For both bit-strings, a 0 represents a route though the leftmost output link, and a 1 represents a route though the rightmost output link. The least significant bit of both bit-strings represents the route taken at the first layer of switches, and the most significant bit represents the route taken at the last layer of switches before the direction of the route is reversed.

5.4.1 Packet data structure

Reference to the calculated route is required throughout each packet's traversal across the network, which means a Packet data structure is required. The Packet data structure consists of the count value and routing and addressing bit-strings created by the routing algorithm, as well as the source processor address, the destination processor address, the destination port number, the data being transmitted, and an indication of which direction the packet is currently travelling in.

For every output instruction read by the processors, a new packet is created. Once all of the values in the Packet data structure have been set by the routing algorithm, the packet is passed to the network adapter where it is added to the network ready for transmission.

5.5 Emulating the parallelism

The emulated hardware building blocks of the network are now fully connected, and can interact with each other. The final stage of the network implementation concerns the functionality of these hardware components when they are working together in parallel. The standard C libraries do not support parallel computation, so the parallelism is emulated using a timestep implementation where a single unit of time's worth of functionality is carried out by each component in the network in serial, before the next timestep is started and the process is repeated.

The execution of each timestep can be split into four chronological stages: a step of each processor, route calculation for any new packets, adding new packets to the network, and a step of packet traversal through the network.

In the first stage of the timestep, each of the processors connected to the network executes a single instruction and carries out any computation that this instruction involves. If any of the processors read a network output instruction, details of this communication are passed to the routing algorithm for use in the second and third stages. If no network output instructions are read by any of the processors in the timestep, the second and third stages can be skipped because there are no new packets to add into the network. The second stage of the timestep takes a permutation of all of the communications that were initialised previously by the processors, and calculates collision-free routes for this permutation using the routing algorithm. The run time of this algorithm is considered constant and can be executed in a single timestep. The third stage of the timestep passes the newly created packets to the network adapter to introduce into the network. This can also be carried out in a single timestep because a maximum of one packet can be added to each processor.

In the fourth stage of the timestep, each of the packets in the network is advanced one step along their calculated paths. For example, a packet currently passing though a link is progressed to the next switch on its journey, or a packet currently at the front of a switch

buffer is sent along the output link it has been waiting to use. Further emulated parallelism needs to be implemented during this stage to compensate for concurrent network traversal in a real hardware application. To carry out this emulation in a serial manner, traversal of the packets through the network is split into five further stages. Firstly, the network adapters are scanned for any newly added packets to the network. Any new packets found are passed along the first link of their route and placed in the temporary packet space in the Link data structure. This is because a data packet from the previous timestep may still be stored in the main packet space of the link, which will subsequently be passed on to the next switch on its journey later in the present timestep, vacating the link for the newly added packet. Once each of the newly added packets has embarked on their route, each of the packets positioned at the front of the output buffers in each switch are sent along their corresponding output links. Again, each of these packets is placed in the temporary packet space of the Link data structure they are travelling across. Next, each of the processors checks the main packet space of each of the links they are connected to for any packets that have reached their destination. If this is the case, the packet is placed in the specified port of the network adapter ready for collection by the processor, and the main packet space in the link it was taken from is set to null to indicate that the link has become vacant during the timestep. Once all of the arriving packets have been collected, the packets stored in the main packet space of the rest of the links in the network are forwarded to the next switch on their journey. If there is sufficient free buffer space in the switch for the packet, it is stored in the buffer and the main packet space in the input link is set to null. If there is not sufficient buffer space for the packet to be accepted into the switch, it must remain in the input link until the required buffer space becomes available. In the final stage of the emulated network traversal, each of the links in the network are checked for collisions. If the temporary packet space in the link is empty, there is no chance of a collision. If the temporary packet space is occupied but the main packet space is set to null, then the link has been vacated during the timestep and the temporary packet can be updated as the main packet currently travelling through the link. If the temporary packet space is occupied and the main packet space is also occupied, a collision has occurred where two packets are attempting to use the same link in the same timestep. In this situation, the packet stored in the temporary packet space is returned to the switch that it was sent by, and is required to wait until the link is vacated before it can attempt to travel along it again. This concludes the stages of emulated parallel network traversal, meaning that each of the packets in the network has attempted to advance exactly one step along their calculated paths towards their destination.

The four stages of each timestep are repeated until all of the processors have finished executing their section of instructions provided by the parallel input program. Once the end of the program has been reached, the emulation prints the gathered performance metrics of the program to standard output and terminates gracefully. The total number of timesteps that were required for full execution of the input program is included as a performance metric as it gives an indication of the emulated run time of the program.

6 Testing

6.1 Syntax of parallel input programs

The parallel input programs that are compiled and then executed by the emulation have been referenced throughout the thesis. Now that implementation of the network is complete and specific input programs are required to test the functionality of the emulation, they can be described in more detail. A partial description of the language used in the input programs is provided as **Appendix C**, however some instructions were added to this language to support parallel sections and communication between parallel processors. An example of a parallel input program, along with a description of the syntax added to the language to support parallelism, is given below.

```
proc main() is
    var in;
    var out;
    network
    { { out := 23; 1 ! out } &
      { 0 ? in             }
    }
```

Figure 9: An example of input program source code for syntax demonstration.

- `network { { } & { } }` – This structure supports the parallel sections in the program. The network keyword indicates the start of the parallel section, and it is followed by the sections of code to be executed by each parallel processor enclosed in curly brackets. The ampersand symbol is used to separate each block of parallel code, which means that multiple sections can be defined to incorporate all of the processors in the network. Figure 9 shows a parallel section that uses two processors.

- `dest ! data` – This is a communication output instruction, which informs the processor that a packet needs to be sent across the network to another processor. The leftmost value indicates the destination processor of the packet, the exclamation mark indicates that the packet is to be added to the network, and the rightmost value indicates the data to be included in the packet. Figure 9 shows an output instruction on the first processor, sending a value of 23 to the second processor.

- `port ? var` – This is a communication input instruction, which informs the processor that a packet from the network must be received before any further computation can be carried out. The leftmost value indicates the port of the network adapter that the received packet is stored in, the question mark indicates that the packet is to be removed from the network, and the rightmost value indicates the local variable name

that the data from the received packet will be stored under. Figure 9 shows an input instruction on the second processor, receiving a value of 23 from the first processor.

6.2 Testing the correctness of the network

Test program: The input program used to test whether the network sends data packets along their intended route to their intended destination is listed in **Appendix D.1**. The program is a relay-style message passing program using eight processors, where a value is initialised in the first processor and sent to the second processor, incremented, and passed on to the third processor. This process is repeated until the final processor is reached, where the final incrementation is applied and the value is returned to the first processor.

Result: The performance metrics returned by the emulation indicate that after a total of 137 timesteps, 8 packets were sent and 0 collisions were detected. This is the expected behaviour of the program as each processor sends one packet, and each packet was sent one after another so there was no possibility of any collisions occurring. On closer inspection of the verbose output of the emulation, it can be seen that all of the packets were sent and received in the correct order with the correct data. An initial value of 50 was set by the first processor and then incremented by each of the seven remaining processors. The correct value of 57 was returned to the first processor which suggests that the network functions as intended.

6.3 Testing the correctness of the routing algorithm

Test program: The input program used to test whether the routing algorithm calculates collision-free paths through the network for any permutation is listed in **Appendix D.2**. The program includes sixteen concurrent communications by processors on opposite sides of the network, which means that a full permutation of maximum-length routes must be calculated. Using other routing algorithms, this scenario would be most likely to cause at least one collision during network traversal.

Result: The performance metrics returned by the emulation indicate that after a total of 53 timesteps, 16 packets were sent and 0 collisions were detected. This is the expected behaviour of the program as each processor sends one packet, and the routes taken by these packets were calculated so that no collisions in the network would occur. The number of timesteps that the program takes to execute is also lower than the program used to test the correctness of the network because, although more packets are sent, they are sent in parallel so each processor does not have to wait to receive data in order to send its own data, resulting in a faster run time. On closer inspection of the verbose output of the emulation, it can be seen that all sixteen routes were of maximal length (four routing bits and four addressing bits), all packets were added to the network on the same timestep, and all packets reached their destination on the same timestep. This suggests that the routing algorithm calculates collision-free paths for full permutations as intended.

6.4 Testing against the specification

A measure of how well the emulation was implemented can be gauged by checking the specification of the solution to see whether all of the original requirements were met. The specification was split into four sections: functional requirements, interface requirements, performance requirements, and software system requirements.

The functional requirements described the purpose and functionality of the emulation. The design of the emulation was based firmly around these requirements, which means they are fulfilled completely by the final product. The emulation takes a single parallel program as input, and distributes the parallel sections of this program across the multiple processors. Each of these sections is then executed in emulated parallel by each processor. The network supports message passing between processors, and these messages are passed along collision-free routes in the network calculated by the routing algorithm. Once the emulation is complete, the number of packets sent, the execution time of the program, and the number of collisions detected in the network is displayed to the user, before the emulation is terminated. This brief description of the behaviour of the emulation covers all of the main functional requirements, and shows they have all been satisfied.

The interface requirements described how the user should interact with the solution. The emulation is packaged with a script file that takes an input program as a parameter, compiles the program and executes it on the emulation using a single Linux command. Once the emulation has finished, the performance metrics are printed to standard output and all of the intermediate files (eg. the program binary) are deleted. The user has the option of setting the emulation to verbose mode, where details of all of the communication packets are printed to standard output, however there was no sensible method of printing the whole state of the network to standard output in a readable manner. This means that all of the interface requirements have been satisfied apart from requirement 2.5.

The performance requirements described how quickly the emulation should be able to execute given input programs. The emulation was tested using examples of the input programs described, and the execution time noted. All of the performance requirements were met when the emulation was only required to print the performance metrics at the end, however the execution time of the larger programs suffered when verbose mode was activated due to the large increase in print statements.

The software system requirements provided some general guidelines that the emulation was required to follow. The emulation compiles cleanly and outputs clear performance metrics, which shows it is reliable. The source code of the emulation is well structured and open source, so it is readily available for inspection. No destructive behaviour can be carried out by the emulation, and it cannot be accessed externally, which means it is secure. Assuming the user has studied the user manual, the specified changes to the emulation and be carried out in less than 10 seconds which shows that it is easily maintainable. Finally, the source code of the emulation uses only the standard C libraries, making it extremely portable between Linux machines. These features show that all of the software system requirements have been satisfied.

7 Results

7.1 Performance compared to two-phase randomised routing

One way to measure the performance of the developed routing algorithm is to compare it to an existing algorithm used to route packets in modern parallel architectures. A commonly used algorithm is two-phase randomised routing, which selects a random switch in the network that the packet must pass through on the way to its destination in an attempt to spread traffic uniformly across the network. In order to compare the performance of these two routing algorithms against each other, an emulation of two-phase randomised routing was created which can be interchanged with the developed routing algorithm in the network. With both algorithms in place, their relative performance can be measured using a number of test programs.

7.1.1 Full permutations

Test program: The input program used to compare how efficiently the two algorithms can route multiple full permutations is listed in **Appendix D.3**. The program uses 32 processors, each of which sends a single packet to the opposite side of the network for each cycle of communication. There are 1000 cycles of communication in total, which will give a good indication of how each algorithm handles the routing of full permutations.

Result:

	Packets sent	**Timesteps completed**	**Collisions detected**
Routing algorithm	32000	36037	0
Two-phase randomised routing (avg. 10 runs)	32000	36576	7691

Table 1: The performance metrics for both algorithms routing full permutations.

The performance metrics show that the developed routing algorithm was able to avoid any collisions in the network, whereas the packets directed by two-phase randomised routing encountered a large number of collisions. These collisions caused the emulation using two-phase randomised routing to require more than 500 additional timesteps to execute the input program compared to the emulation using the developed algorithm, which is a 1.5% increase in run time. This increase suggests that not all of the collisions affected the overall run time of the emulation, and that the latency setbacks due to collisions on some routes were masked by larger latency setbacks on other routes. An explanation for why the

packets routed by the developed routing algorithm encountered no collisions in the network can be found on closer inspection of the input program. All of the processors require parallel communication with processors on the opposite side of the network, which means that every route has to pass through the outermost layer of switches in the network, making them all of equal length. The packets are all sent during the same timestep and, providing that no collisions occur in the network, are all delivered during the same timestep. This leads to guaranteed full permutations for each communication cycle in the program, which the routing algorithm has been proven to calculate collision-free routes for.

7.1.2 Regular permutations

Test program: The input program used to compare how efficiently the two algorithms can route multiple regular permutations is listed in **Appendix D.4**. The program connects 32 processors, each of which sends a single packet to the processor directly to the right of it for each cycle of communication (apart from the rightmost processor, which sends a packet back around to the leftmost processor). The linearity of the permutation means it is regarded as regular. There are a total of 1000 cycles of communication, which will give a good indication of how each algorithm handles the routing of regular permutations.

Result:

	Packets sent	**Timesteps completed**	**Collisions detected**
Routing algorithm	32000	23789	0
Two-phase randomised routing (avg. 10 runs)	32000	36972	9622

Table 2: The performance metrics for both algorithms routing regular permutations.

The performance metrics show that the developed algorithm was again able to avoid any collisions in the network, whereas the packets routed by two-phased randomised routing encountered a large number of collisions. The number of timesteps that the emulation using two-phase randomised routing required to execute the input program was approximately 55% more than the emulation using the developed algorithm. Some of this increase in run time can be attributed to the collisions in the network, however a large portion of it can also be credited to the fact that two-phase randomised routing directs all of the packets to the outermost layer of switches in the network, whereas the developed routing algorithm uses shorter routes when traversal to the outermost layer of switches is not required. An explanation for why the packets routed by the developed routing algorithm encountered no collisions in the network can be found on closer inspection of the input program. All of the processors require communication with processor directly to the right of them, mean-

ing that most of the routes are short and the paths rarely cross each other. This leads to collision-free paths being routed by the algorithm and causes the input program to be executed quickly and efficiently.

7.1.3 Irregular permutations

Test program: The input program used to compare how efficiently the two algorithms can route multiple irregular permutations is listed in **Appendix D.5**. The program uses 32 processors, each of which sends a single packet to an arbitrarily chosen processor in the network for each cycle of communication. As the permutation is made up of randomly chosen communications, it can be regarded as an irregular permutation. A total of 1000 communication cycles are carried out by the program, which will give a good indication of how each algorithm handles the routing of irregular permutations.

Result:

	Packets sent	**Timesteps completed**	**Collisions detected**
Routing algorithm	32000	36458	7416
Two-phase randomised routing (avg. 10 runs)	32000	36921	9472

Table 3: The performance metrics for both algorithms routing irregular permutations.

The performance metrics show that, while both emulations suffered a significant number of collisions in the network, the emulation using the developed routing algorithm suffered fewer collisions than the emulation using two-phase randomised routing. These extra collisions mean that the run time of the emulation using two-phase randomised routing is 1.3% slower than the emulation using the developed routing algorithm. However, it is immediately noticeable that the developed routing algorithm performed worse when routing irregular permutations than when it was routing full and regular permutations. An explanation for this behaviour can be found on closer inspection of the input program. The fact that random communications are required means that each route calculated by the routing algorithm for the first permutation is of a variable length, depending on the proximity of the processors that require the communication. This means that, although all of the packets are added to the network during the same timestep, the difference in path length means that some of the packets in the permutation arrive at their destination before others. Once an early arriving packet is collected by its processor, the processor can continue executing instructions while some of the processors are still waiting for their packets to arrive. This leads to a loss of synchronisation between the processors, and means that full permutations of communication are replaced with partial permutations that are

sent when there may still be data from the previous communication cycle being sent across the network. In these circumstances the routing algorithm cannot avoid collisions in the network between different permutations, so some of the added efficiency that the routing algorithm brings to the emulation is lost.

7.2 Performance compared to a serial alternative

routing algorithm: packets 1024, timesteps 4907, collisions 256

serial: packets 0, timesteps 76115, collisions 0

8 Conclusion

8.1 Summary of achievements

8.2 Possible future developments

9 References

[Adv08] Adve, S. V. et al. (November 2008). *'Parallel Computing Research at Illinois: The UPCRC Agenda'*. Parallel@Illinois, University of Illinois at Urbana-Champaign.

[Ben65] Beneš, V. E. (1965). *Mathematical Theory of Connecting Networks and Telephone Traffic*. Mathematics in Science and Engineering. Academic Press.

[Clo53] Clos, C. (March 1953). *'A study of non-blocking switching networks'*. Bell System Technical Journal 32 (2): 406-424. doi:10.1002/j.1538-7305.1953.tb01433.x.

[Dal87] Dally, W. J., Seitz, C. L. (May 1987). *Deadlock-free message routing in multiprocessor interconnection networks*. Computers, IEEE Transactions on, 100 (5): 547-553.

[Dal92] Dally, W. J. (March 1992). *Virtual-channel flow control*. IEEE Transactions on Parallel and Distributed Systems, 3 (2): 194-205.

[Dal03] Dally, W. J., Towles, B. (2003). *Principles and Practices of Interconnection Networks*. Morgan Kaufmann.

[Fly72] Flynn, M. J. (September 1972). *'Some Computer Organizations and Their Effectiveness'*. IEEE Trans. Comput. C-21 (9): 948-960. doi:10.1109/TC.1972.5009071.

[Got89] Gottlieb, A. Almasi, G. (1989). *Highly parallel computing*. Redwood City, Calif.: Benjamin/Cummings. ISBN 0-8053-0177-1.

[Han14] Hanlon, J. W. (March 2014). *Scalable abstractions for general-purpose parallel computation*. p. 44-48, 62.

[Hen11] Hennessy, J. L., Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach*. Online appendix F, retrieved from http://booksite.elsevier.com/9780123838728/references.php

[Iba08] Ibaroudene, Djaffer. (Spring 2008). *'Parallel Processing, EG6370G: Chapter 1, Motivation and History'*. St. Mary's University, San Antonio, TX.

[Jon97] Jones, A. M. et al. (1997). *The Network Designer's Handbook*. IOS Press, 1st edition.

[Lam04] Lammle, T. (2004). *CCNA Study Guide* (Fourth edition). Sybex Inc. ISBN 0-7821-4311-3.

[Moo65] Moore, G. E. (1965). *'Cramming more components onto integrated circuits'*. Electronics Magazine. p. 4.

[Sut05] Sutter, H. (March 2005). *'The free lunch is over'*. Dr Dobb's Journal, 30(3). Online version, updated August 2009.

[Toy86] Toy, W., Zee, B. (1986). *Computer Hardware/Software Architecture*. Prentice Hall. ISBN 0-13-163502-6.

[Val90] Valiant, L. G. (1990). *General purpose parallel architectures.* In Handbook of theoretical computer science (vol. A): algorithms and complexity, p. 943-973. MIT Press.

Appendix A: User Manual

Appendix B: Summary of Processor Instruction Set

This appendix describes a summary of the instruction set associated with the unmodified serial processor. The modifications carried out to to the processor to add parallelism to its capabilities are described in sections 4 and 5. This document was written by David May.

Background

The architecture described here is specifically designed as a very simple component processor for multiprocessors. In area, it would occupy less area on silicon than $16kbytes$ of memory, allowing a manufacturable chip to contain thousands of processors. Further, its instruction set requires a very small compiler to generate near-optimal code, so it is practical to compile on-the-fly as programs are input. This may be important as it makes it practical to broadcast programs to thousands of processors, each of which compiles the incoming program in the context of its local state. The main features of the instruction set are:

- Short instructions are provided to allow efficient access to the stack and other data regions allocated by compilers; these also provide efficient branching and subroutine calling. The short instructions have been chosen on the basis of extensive evaluation to meet the needs of modern compilers.
- The memory is word addressed; however the instructions are all single byte so instruction addresses refer to a specific byte position within a word.
- The same instruction set can be used for processors with different wordlengths; the only requirement is that the wordlength is a number of bytes.
- The processor has a small number of registers. Some registers are used for specific purposes such as accessing the stack or building large constants.
- Instructions are easy to decode.

All instructions are 8-bit; each instruction contains 4 bits representing an operation and 4 bits of immediate data. A special instruction, OPR causes its operand to be interpreted as an inter-register operation. Instruction prefixes are used to extend the range of immediate operands and to provide more inter-register operations.

The prefixes are:

- PFIX which concatenates its 4-bit immediate with the 4-bit immediate of the next 8-bit instruction.

- NFIX which complements its its 4-bit immediate and then concatenates the result with the 4-bit immediate of the next 8-bit instruction.

The prefixes are inserted automatically by compilers and assemblers.

The normal state of a processor is represented by 5 registers. Two of the registers are operated as a stack and used to hold the sources and destination of inter-register operations.

register	**use**
pc	the program counter
sp	the stack pointer
oreg	the operand register
areg	the first register in the operand stack
breg	the second register in the operand stack

Instruction Issue and Execution

The processor core is intended to be implementable without a pipeline to maximise responsiveness; this potentially allows a very simple design.

Note that the instructions are all 8-bit, so that on a 32-bit implementation four instructions are fetched every cycle. As typically less than 40% of instructions require a memory access, it is therefore practical to support the processor using a simple unified memory system.

Each processor has a short instruction buffer which is one word long. The rules for performing an instruction fetch are as follows:

- Any instruction which requires data-access performs it during the memory access stage.
- Branch instructions fetch their branch target instructions during the memory access stage.
- Any other instruction (such as ALU operations) performs an instruction fetch if it is the last instruction in the instruction buffer.
- If the instruction buffer is empty when an instruction should be issued, a special *no-op* is issued; this will load the instruction buffer.

Instruction set Notation and Definitions

In the following description

Bpw	is the number of bytes in a word
bpw	is the number of bits in a word
mem	represents the memory
pc	represents the program counter
sp	represents the stack pointer
$oreg$	represents the operand register
$areg$	represents the first stack register
$breg$	represents the second stack register
$u4$	is a 4-bit unsigned source operand in the range $[0:15]$

Data access

The data access instructions fall into several groups. One of these provides access via the stack pointer.

LDWSP	$areg, breg \leftarrow mem[sp + oreg], areg$	load word from stack
STWSP	$mem[sp + oreg], areg \leftarrow areg, breg$	store word to stack
LDAWSP	$areg, breg \leftarrow sp + oreg, areg$	load address of word in stack

Access to constants and program addresses is provided by instructions which either load values directly or enable them to be loaded from a location in the program:

LDC	$areg, breg \leftarrow oreg, areg$	load constant
LDAP	$areg, breg \leftarrow pc + oreg, areg$	load address in program

Access to data structures is provided by instructions which combine an address with an offset:

LDWI	$areg \leftarrow mem[areg + oreg]$	load word
STWI	$mem[areg + oreg] \leftarrow breg$	store word

Expression evaluation

ADDC	$areg \leftarrow areg + oreg$	add constant
ADD	$areg \leftarrow breg + areg$	add
SUB	$areg \leftarrow breg - areg$	subtract
EQC	$areg \leftarrow areg = oreg$	equal constant
EQ	$areg \leftarrow breg = areg$	equal
LSS	$areg \leftarrow breg <_{sgn} areg$	less than signed
AND	$areg \leftarrow breg \wedge areg$	and
OR	$areg \leftarrow breg \vee areg$	or
XOR	$areg \leftarrow breg \oplus areg$	exclusive or
NOT	$areg \leftarrow -1 \oplus areg$	not
SHL	$areg \leftarrow breg \ll areg$	logical shift left
SHR	$areg \leftarrow breg \gg areg$	logical shift right

Branching, jumping and calling

The branch instructions include conditional and unconditional relative branches. A branch using an offset in the stack is provided to support jump tables.

BR	$pc \leftarrow pc + oreg$	branch relative unconditional
BRT	if $areg$ then $pc \leftarrow pc + oreg$	branch relative true
BRX	$pc, areg \leftarrow areg, breg$	branch absolute

The procedure calling instruction uses a program address in the stack to determine a subroutine entry point, leaving the return address on the stack. The RET instruction can also be used simply to branch to a program address on the stack.

CALL	$mem[sp], pc, areg \leftarrow pc, areg, breg$	call subroutine
RET	$pc \leftarrow mem[sp]$	return from subroutine

Typically, the stack is initialised to a high address in memory and is extended on subroutine entry and contracted on exit. The instructions to support this are shown below.

SETSP	$sp, areg \leftarrow areg, breg$	set stack pointer
ADJ	$sp \leftarrow sp + oreg$	adjust stack

Instruction summary

LDWSP	$areg, breg \leftarrow mem[sp + oreg], areg$	load word from stack
STWSP	$mem[sp + oreg], areg \leftarrow areg, breg$	store word to stack
LDAWSP	$areg, breg \leftarrow sp + oreg, areg$	load address of word in stack
LDC	$areg, breg \leftarrow oreg, areg$	load constant
LDAP	$areg, breg \leftarrow pc + oreg, areg$	load address in program
LDWI	$areg \leftarrow mem[areg + oreg]$	load word
STWI	$mem[areg + oreg] \leftarrow breg$	store word
ADDC	$areg \leftarrow areg + oreg$	add constant
EQC	$areg \leftarrow areg = oreg$	equal constant
BR	$pc \leftarrow pc + oreg$	branch relative unconditional
BRT	if $areg$ then $pc \leftarrow pc + oreg$	branch relative true
ADJ	$sp \leftarrow sp - oreg$	adjust stack
ADD	$areg \leftarrow breg + areg$	add
SUB	$areg \leftarrow breg - areg$	subtract
EQ	$areg \leftarrow breg = areg$	equal
LSS	$areg \leftarrow breg <_{sgn} areg$	less than signed
AND	$areg \leftarrow breg \wedge areg$	and
OR	$areg \leftarrow breg \vee areg$	or
XOR	$areg \leftarrow breg \oplus areg$	exclusive or
NOT	$areg \leftarrow -1 \oplus areg$	not
SHL	$areg \leftarrow breg \ll areg$	logical shift left
SHR	$areg \leftarrow breg \gg areg$	logical shift right
BRX	$pc, areg \leftarrow areg, breg$	branch absolute
CALL	$mem[sp], pc, areg \leftarrow pc, areg, breg$	call subroutine
RET	$pc \leftarrow mem[sp]$	return from subroutine
SETSP	$sp, areg \leftarrow areg, breg$	set stack pointer
BOOT	$load()$	boot processor
	$pc, sp \leftarrow mem[0], memhigh$	

Appendix C: Description of Input Program Syntax

This appendix partially describes the syntax of the programming language that is used to write input programs for the emulation. Some modifications to the language described here have been carried out to support parallel sections and communication between parallel processors. These modifications are described in section 6.1. The following document was written by David May.

The X Language

X is a simple sequential programming language. It is easy to compile and an X compiler written in X is available to simplify porting between architectures. It is relatively easy to modify the compiler to target new architectures or to and extend the language.

Notation

The following examples illustrate the notation used in the definition of X.

The meaning of

assignment = *variable* `:=` *expression*

is "An *assignment* is a *variable* followed by `:=` followed by an *expression*"

The meaning of

literal = *integer* | *byte* | *string*

is "An *literal* is an *integer* or a *byte* or a *string*". This may also be written

literal = *integer*

literal = *byte*

literal = *string*

The notation { *process* } means "a list of zero or more *processes*".

The notation $\{_0$ `,` *expression*$\}$ means "a list of zero or more expressions separated from each other by `,`", and $\{_1$ `,` *expression*$\}$ means "a list of one or more expressions separated from each other by `,`".

The format of an X program is specified by the syntax. Space, tab and line breaks are ignored and can be inserted in text strings using the escape character `*`.

Comment

comment = | *text* |

text = {$_0$ *character*}

A comment is used to describe the operation of the program.

process = *comment process*

Let C be a comment and P be a process. Then C P behaves like P.

Statement

process = `skip`
| `stop`
| *assignment*
| *sequence*
| *conditional*
| *loop*
| *call*

`skip` starts, performs no action, and terminates.

`stop` starts but never proceeds and never terminates.

assignment = *variable* `:=` *expression*

An assignment evaluates the expression, assigns the result to the variable, and then terminates. All other variables are unchanged in value.

sequence = `{` {$_0$ `;` *process*} `}`

A sequence starts with the start of the first process. Each subsequent process starts if and when its predecessor terminates and the sequence terminates when the last process terminates. A sequence with no component processes behaves like `skip`.

Conditional

conditional = `if` *expression* `then` *process* `else` *process*

Let e be an expression and let P and Q be processes. Then

`if` e `then` P `else` Q

behaves like P if the initial value of e is *true*. Otherwise it behaves like Q.

Loop

loop = `while` *expression* `do` *process*

A loop is defined by

`while` e `do` P = `if` e `then { ` P `; while` e `do` P `} else skip`

Scope

process = *specification* `;` *process*

specification = *declaration*
| *abbreviation*
| *definition*

A block N : S behaves like its scope S; the specification N specifies a name which may be used with this specification only within S.

Let x and y be names and let $S(x)$ and $S(y)$ be scopes which are similar except that $S(x)$ contains x wherever $S(y)$ contains y, and vice versa. Let $N(x)$ and $N(y)$ be specifications which are similar except that $N(x)$ is a specification of x and $N(y)$ is a specification of y. Then

$N(x)$; $S(x) = N(y)$; $S(y)$

Using this rule it is possible to express a process in a canonical form in which no name is specified more than once.

Declaration

declaration = `var` *name*
| `array` *name* `[` *expression* `]`

A declaration declares a name as the name of a variable or of an array.

Abbreviation

abbreviation = `val` *name* `=` *expression*
| `array` *name* `=` *name*
| `proc` *name* `=` *name*
| `func` *name* `=` *name*

An abbreviation `val` n `=` e specifies n as an abbreviation for expression e. Let e be an expression and $P(e)$ be a process. Then

`val` n `=` e `;` $P(n) = P(e)$

Let T be `array`, `proc` or `func`. Then

T n `=` m `;` $P(n) = P(m)$

Procedure

definition = `proc` *name* `(` {$_0$ `,` *formal*} `)` `is` *body*

formal = `val` *name*
| `array` *name*
| `proc` *name*
| `func` *name*

body = *process*

The definition

`proc` n `(` {$_0$ `,` *formal*} `)` `is` B

defines n as the name of a procedure.

instance = *name* `(` {$_0$ `,` *actual*} `)`

actual = *expression*
| *name*

Let X be a program expressed in the canonical form in which no name is specified more than once. If X contains a procedure definition

P`(`F_0`,` F_1`,` ...`,` F_n`)` `is` B

then within the scope of P

P`(`A_0`,` A_1`,` ...`,` A_n`)` $= F_0$ `=` A_0 `;` F_1 `=` A_1 `;` ... F_n `=` A_n `;` B

provided that each abbreviation F_i `=` A_i is valid.

A procedure can always be compiled either by substitution of its body as described above or as a closed subroutine.

Element

Elements enable variables or arrays be selected from arrays.

element = *element*`[` *subscript* `]`
| *name*

subscript = *expression*

Let a be an array with n components and e an expression of value s. Then v`[`e`]` is valid only if $0 \leq s$ and $s < n$; it is the component of v selected by s.

Variable

variable = *element*

Every variable has a value that can be changed by assignment or input. The value of a variable is the value most recently assigned to it, or is arbitrary if no value has been assigned to it.

Let a be an array with n components, e be an expression of value s, and x be an expression. If $0 \leq s$ and $s < n$, then v`[`e`] :=` x assigns to v a new value in which the component of v selected by s is replaced by the value of x and all other components are unchanged. Otherwise the assignment is invalid.

Literal

literal = *integer* | *byte* | *string* | `true` | `false`

integer = *digits* | `#`*digits*

byte = `'`*character*`'`

An integer literal is a decimal number, or `#` followed by a hexadecimal number. A byte literal is an ASCII character enclosed in single quotation marks: `'`.

A string literal is represented by a sequence of ASCII characters enclosed by double quotation marks: `"`. Let s be a string of n characters, where $n < 256$. The value of s is an

array containing the value n, followed by ASCII values of the characters in the string. The string is packed into the array.

The literal `true` represents the logical value *true*; numerically `true` = 1. The literal `false` represents the logical value *false*; numerically `false` = 0.

Expression

An expression has a data type and a value. Expressions are constructed from operands, operators and parentheses.

operand = *element* | *literal*
| `(` *expression* `)`

The value of an operand is that of an element, literal or expression.

expression = *monadic.operator operand*
| *operand diadic.operator operand*
| *operand*

The arithmetic operators `+` and `-` produce the arithmetic sum and difference of their operands respectively. Both operands must be integer values and the result is an integer value. The arithmetic operators treat their operands as signed integer values and produce signed integer results. If n is an operand, then `-`n = `(0-`n`)`.

The logical operator `and` produces the logical and of its operands, both of which must have value `true` or `false`. If the value of the first operand is `false`, the result is `false`; otherwise the result is the value of the second operand.

The logical operator `or` produces the logical or of its operands, both of which must have value `true` or `false`. If the value of the first operand is `true`, the result is `true`; otherwise the result is the value of the second operand.

The logical operator `not` produces the logical not of its operand which must have value `true` or `false`:

`not false` = `true` `not true` = `false`

Let **O** be one of the associative operators `+`, `and`, `or`. Then

e_1 **O** e_2 **O** ... **O** e_n = `(`e_1 **O** `(`e_2 **O** `(` ... **O** e_n `)` ... `))`

The relational operators `=`, `<>`, `<`, `<=`, `>`, `>=` produce a result of `true` or `false`. The operands must both be integer values. The result of x `=` y is `true` if the value of x is equal to that of y. The result of x `<` y is `true` if the integer value of x is strictly less than that of y. The other operators obey the following rules:

(*x* <> *y*) = not (*x* = *y*) (*x* >= *y*) = not (*x* < *y*)
(*x* > *y*) = (*y* < *x*) (*x* <= *y*) = not (*x* > *y*)

where x and y are any values.

expression = valof *process*

process = return *expression*

A valof expression executes a process to produce a value. The final process executed in a valof must be a return. The return evaluates its expression and the resulting value is the value of the valof.

Function

definition = func *name* ({$_0$, *formal*}) is *body*

The definition

func *n* ({$_0$, *formal*}) is *B*

defines n as the name of a function with a body B that computes a value.

expression = *name* ({$_0$, *actual*})

Let X be a program expressed in the canonical form in which no name is specified more than once. If X contains a function definition

func F(F_0, F_1, ..., F_n) is B

then within the scope of F

F(A_0, A_1, ..., A_n) = valof F_0 = A_0 ; F_1 = A_1 ; ... F_n = A_n ; B

provided that each abbreviation F_i = A_i is valid.

A function can always be compiled either by substitution of its body as described above or as a closed subroutine.

Appendix D: Input Program Code Listings

D.1 A program for testing the correctness of the network

```
proc main() is
    var in;
    var out;
    network
    { { out := 50;  1 ! out; 7 ? in  } &
      { 0 ? in; out := in+1; 2 ! out } &
      { 1 ? in; out := in+1; 3 ! out } &
      { 2 ? in; out := in+1; 4 ! out } &
      { 3 ? in; out := in+1; 5 ! out } &
      { 4 ? in; out := in+1; 6 ! out } &
      { 5 ? in; out := in+1; 7 ! out } &
      { 6 ? in; out := in+1; 0 ! out }
    }
```

D.2 A program for testing the correctness of the routing algorithm

```
array in[16];
array out[16];

proc main() is
    network
    { {  out[0] := 50;   8 ! out[0];   8 ? in[0]  } &
      {  out[1] := 51;   9 ! out[1];   9 ? in[1]  } &
      {  out[2] := 52;  10 ! out[2];  10 ? in[2]  } &
      {  out[3] := 53;  11 ! out[3];  11 ? in[3]  } &
      {  out[4] := 54;  12 ! out[4];  12 ? in[4]  } &
      {  out[5] := 55;  13 ! out[5];  13 ? in[5]  } &
      {  out[6] := 56;  14 ! out[6];  14 ? in[6]  } &
      {  out[7] := 57;  15 ! out[7];  15 ? in[7]  } &
      {  out[8] := 58;   0 ! out[8];   0 ? in[8]  } &
      {  out[9] := 59;   1 ! out[9];   1 ? in[9]  } &
      { out[10] := 60;   2 ! out[10];  2 ? in[10] } &
      { out[11] := 61;   3 ! out[11];  3 ? in[11] } &
      { out[12] := 62;   4 ! out[12];  4 ? in[12] } &
      { out[13] := 63;   5 ! out[13];  5 ? in[13] } &
      { out[14] := 64;   6 ! out[14];  6 ? in[14] } &
      { out[15] := 65;   7 ! out[15];  7 ? in[15] }
    }
```

D.3 A program for comparing the performance between the routing algorithm and two-phase randomised routing for full permutations

```
proc main() is
    network
    { { addn(16, 16) }  &  { addn(17, 17) } &
      { addn(18, 18) }  &  { addn(19, 19) } &
      { addn(20, 20) }  &  { addn(21, 21) } &
      { addn(22, 22) }  &  { addn(23, 23) } &
      { addn(24, 24) }  &  { addn(25, 25) } &
      { addn(26, 26) }  &  { addn(27, 27) } &
      { addn(28, 28) }  &  { addn(29, 29) } &
      { addn(30, 30) }  &  { addn(31, 31) } &
      { addn(0, 0)   }  &  { addn(1, 1)   } &
      { addn(2, 2)   }  &  { addn(3, 3)   } &
      { addn(4, 4)   }  &  { addn(5, 5)   } &
      { addn(6, 6)   }  &  { addn(7, 7)   } &
      { addn(8, 8)   }  &  { addn(9, 9)   } &
      { addn(10, 10) }  &  { addn(11, 11) } &
      { addn(12, 12) }  &  { addn(13, 13) } &
      { addn(14, 14) }  &  { addn(15, 15) }
    }

proc addn(val send, val recv) is
    var count;
    var in;
    while count<1000 do
    { count:=in+1; send ! count; recv ? in }
```

D.4 A program for comparing the performance between the routing algorithm and two-phase randomised routing for regular permutations

```
proc main() is
    network
    { { addn(1, 31)  }  &  { addn(2, 0)   } &
      { addn(3, 1)   }  &  { addn(4, 2)   } &
      { addn(5, 3)   }  &  { addn(6, 4)   } &
      { addn(7, 5)   }  &  { addn(8, 6)   } &
      { addn(9, 7)   }  &  { addn(10, 8)  } &
      { addn(11, 9)  }  &  { addn(12, 10) } &
      { addn(13, 11) }  &  { addn(14, 12) } &
      { addn(15, 13) }  &  { addn(16, 14) } &
      { addn(17, 15) }  &  { addn(18, 16) } &
      { addn(19, 17) }  &  { addn(20, 18) } &
      { addn(21, 19) }  &  { addn(22, 20) } &
      { addn(23, 21) }  &  { addn(24, 22) } &
      { addn(25, 23) }  &  { addn(26, 24) } &
      { addn(27, 25) }  &  { addn(28, 26) } &
      { addn(29, 27) }  &  { addn(30, 28) } &
      { addn(31, 29) }  &  { addn(0, 30)  }
    }

proc addn(val send, val recv) is
    var count;
    var in;
    while count<1000 do
    { count:=in+1; send ! count; recv ? in }
```

D.5 A program for comparing the performance between the routing algorithm and two-phase randomised routing for irregular permutations

```
proc main() is
    network
    { { addn(25, 25) }  &  { addn(7, 7)   } &
      { addn(19, 19) }  &  { addn(16, 16) } &
      { addn(8, 8)   }  &  { addn(28, 28) } &
      { addn(21, 21) }  &  { addn(1, 1)   } &
      { addn(4, 4)   }  &  { addn(15, 15) } &
      { addn(29, 29) }  &  { addn(20, 20) } &
      { addn(14, 14) }  &  { addn(30, 30) } &
      { addn(12, 12) }  &  { addn(9, 9)   } &
      { addn(3, 3)   }  &  { addn(27, 27) } &
      { addn(26, 26) }  &  { addn(2, 2)   } &
      { addn(11, 11) }  &  { addn(6, 6)   } &
      { addn(31, 31) }  &  { addn(24, 24) } &
      { addn(23, 23) }  &  { addn(0, 0)   } &
      { addn(18, 18) }  &  { addn(17, 17) } &
      { addn(5, 5)   }  &  { addn(10, 10) } &
      { addn(13, 13) }  &  { addn(22, 22) }
    }

proc addn(val send, val recv) is
    var count;
    var in;
    while count<1000 do
    { count:=in+1; send ! count; recv ? in }
```

D.6 A parallel program that imitates the instructions of a program that blurs the pixels of a 256x256 image in parallel

```
array north_halo[16];
array south_halo[16];
array east_halo[16];
array west_halo[16];
array pixels[9];


proc main() is
    var in;
    var out;
    network
    { { send_halos(12,4,1,3);       blur_local_block();
        receive_halos(12,4,1,3);    blur_halo_blocks()  } &
      { send_halos(13,5,2,0);       blur_local_block();
        receive_halos(13,5,2,0);    blur_halo_blocks()  } &
      { send_halos(14,6,3,1);       blur_local_block();
        receive_halos(14,6,3,1);    blur_halo_blocks()  } &
      { send_halos(15,7,0,2);       blur_local_block();
        receive_halos(15,7,0,2);    blur_halo_blocks()  } &
      { send_halos(0,8,5,7);        blur_local_block();
        receive_halos(0,8,5,7);     blur_halo_blocks()  } &
      { send_halos(1,9,6,4);        blur_local_block();
        receive_halos(1,9,6,4);     blur_halo_blocks()  } &
      { send_halos(2,10,7,5);       blur_local_block();
        receive_halos(2,10,7,5);    blur_halo_blocks()  } &
      { send_halos(3,11,4,6);       blur_local_block();
        receive_halos(3,11,4,6);    blur_halo_blocks()  } &
      { send_halos(4,12,9,11);      blur_local_block();
        receive_halos(4,12,9,11);   blur_halo_blocks()  } &
      { send_halos(5,13,10,8);      blur_local_block();
        receive_halos(5,13,10,8);   blur_halo_blocks()  } &
      { send_halos(6,14,11,9);      blur_local_block();
        receive_halos(6,14,11,9);   blur_halo_blocks()  } &
      { send_halos(7,15,8,10);      blur_local_block();
        receive_halos(7,15,8,10);   blur_halo_blocks()  } &
      { send_halos(8,0,13,15);      blur_local_block();
        receive_halos(8,0,13,15);   blur_halo_blocks()  } &
      { send_halos(9,1,14,12);      blur_local_block();
        receive_halos(9,1,14,12);   blur_halo_blocks()  } &
      { send_halos(10,2,15,13);     blur_local_block();
        receive_halos(10,2,15,13);  blur_halo_blocks()  } &
      { send_halos(11,3,12,14);     blur_local_block();
        receive_halos(11,3,12,14);  blur_halo_blocks()  }
    }
```

```
proc send_halos(val north, val south, val east, val west) is
    var i;
    while i<16 do
    { north ! north_halo[i]; east ! east_halo[i];
      south ! south_halo[i]; west ! west_halo[i];
      i := i+1 }

proc blur_local_block() is
    var i;
    while i<196 do
    { blur_pixel(); i := i+1 }

proc blur_pixel() is
    var i;
    var pixel;
    while i<9 do
    { pixel:= pixel + pixels[i]; i := i+1 }

proc receive_halos(val north, val south, val east, val west) is
    var i;
    while i<16 do
    { south ? north_halo[i]; west ? east_halo[i];
      north ? south_halo[i]; east ? west_halo[i];
      i := i+1 }

proc blur_halo_blocks() is
    var i;
    while i<16 do
    { blur_halo_pixels(i); i := i+1 }

proc blur_halo_pixels(val num) is
    var i;
    while i<9 do
    { north_halo[num] := north_halo[num] + pixels[i];
      east_halo[num] := east_halo[num] + pixels[i];
      south_halo[num] := south_halo[num] + pixels[i];
      west_halo[num] := west_halo[num] + pixels[i];
      i := i+1 }
```

D.7 A serial program that imitates the instructions of a program that blurs the pixels of a 256x256 image in serial

```
array pixels[4096];
array local_pixels[9];

proc main() is
    var i;
    while i<4096 do
    { blur_pixel(i); i := i+1 }

proc blur_pixel(val num) is
    var i;
    while i<9 do
    { pixels[num] := pixels[num] + local_pixels[i];
      i := i+1 }
```