Creating a Predictable Interconnection Network for General-Purpose Parallel Architectures

Benjamin Tovar Matthews

April 28, 2015

Abstract

Contents

List of Figures

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 wire 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 optimisations to 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.

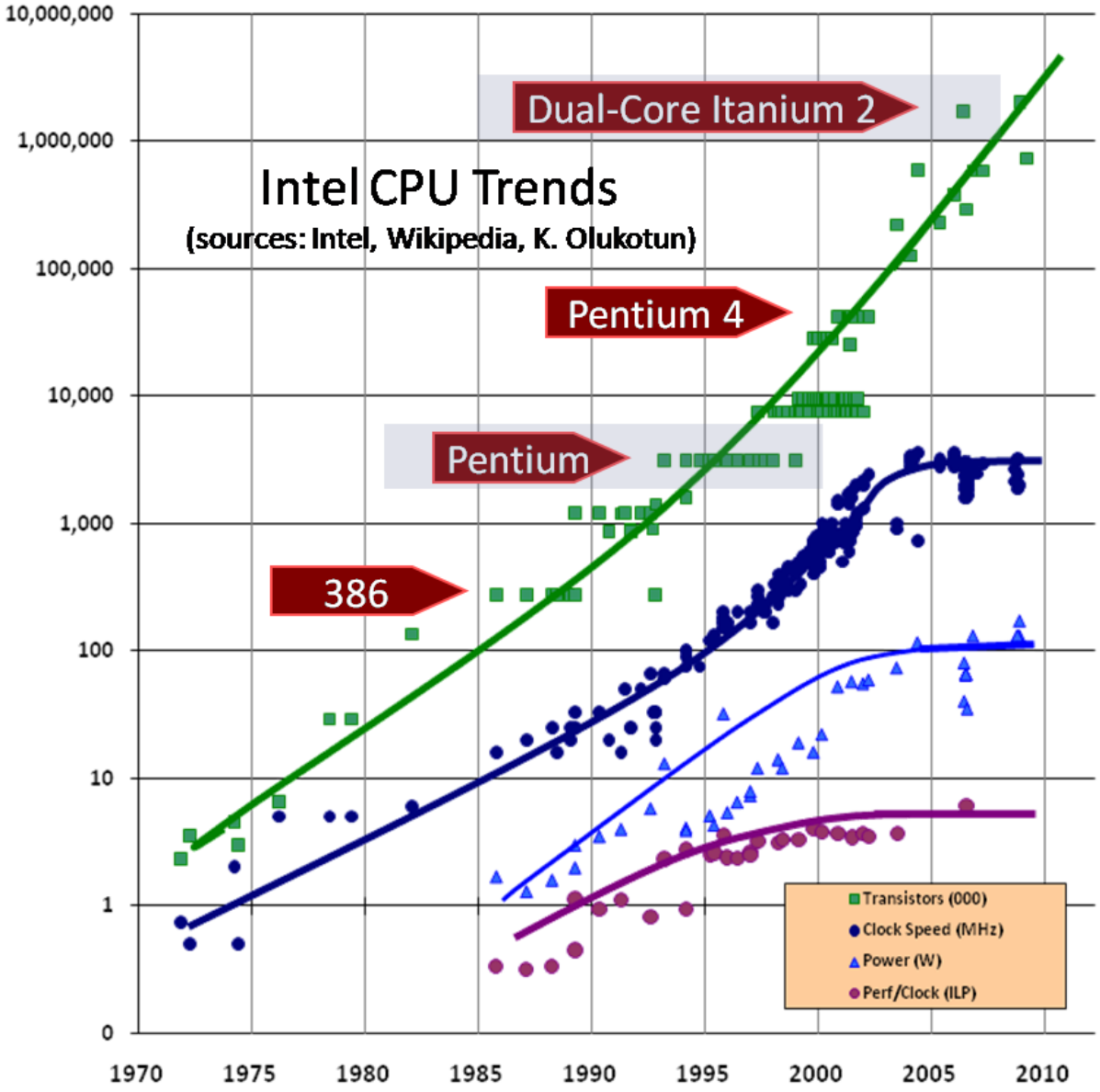


Figure 1: A subset of Intel CPU introductions between 1970 and 2010. [Sut05]

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.

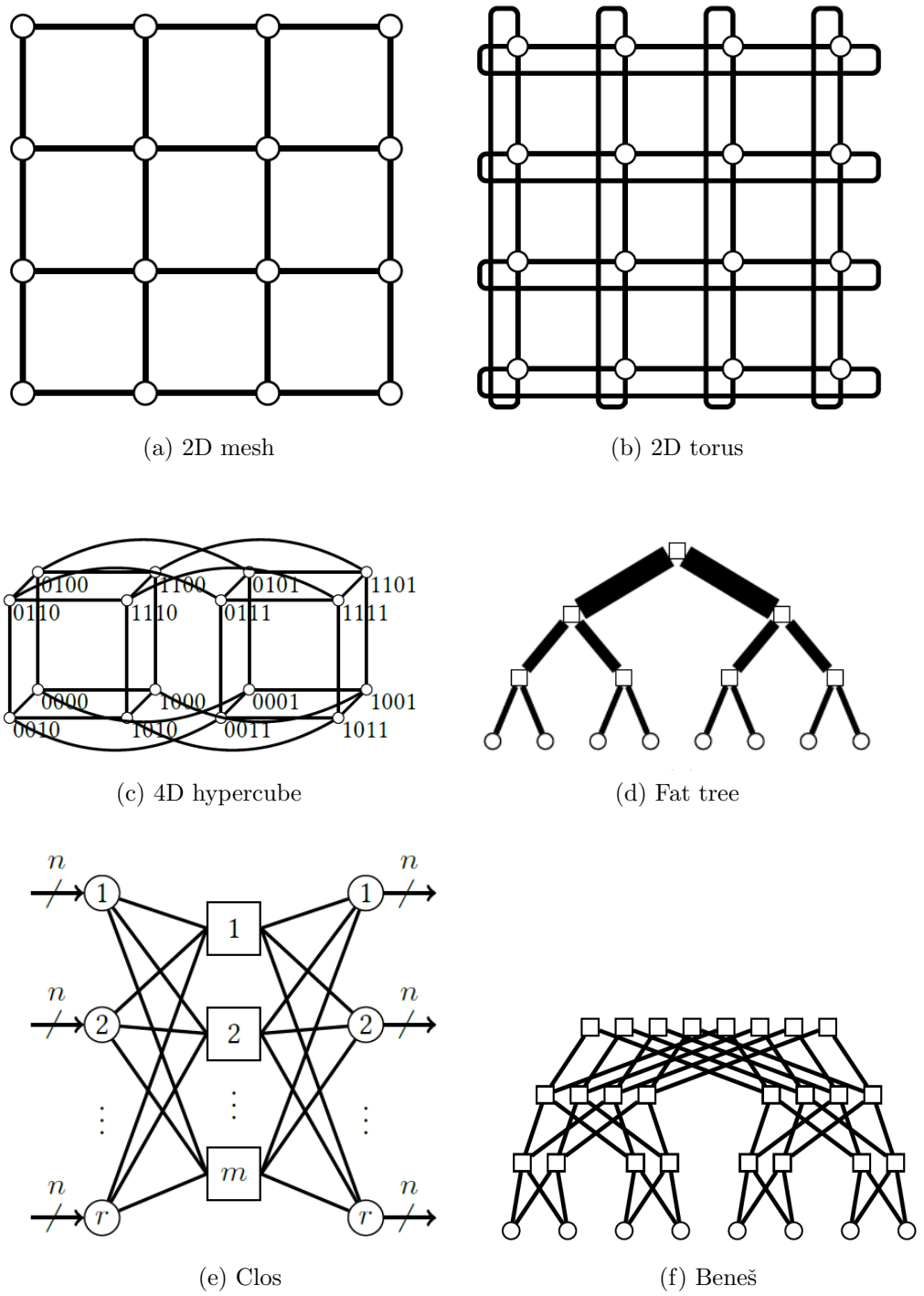
2.4 Interconnection networks

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 still must be fully executed by the emulation given sufficient time.

3.5 Software system requirements

3.5.1 Reliability

3.5.2 Availability

3.5.3 Security

3.5.4 Maintainability

3.5.5 Portability

3.6 Constraints, assumptions and dependencies

memory constraints design constraints hardware assumptions dependent on availability of program compiler

4 Design

5 Implementation

6 Testing

7 Results

8 Conclusion

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.