Creating a Predictable Interconnection Network for General-Purpose Parallel Architectures

Benjamin Tovar Matthews

April 20, 2015

Abstract

Contents

List of Figures

1 Introduction

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.

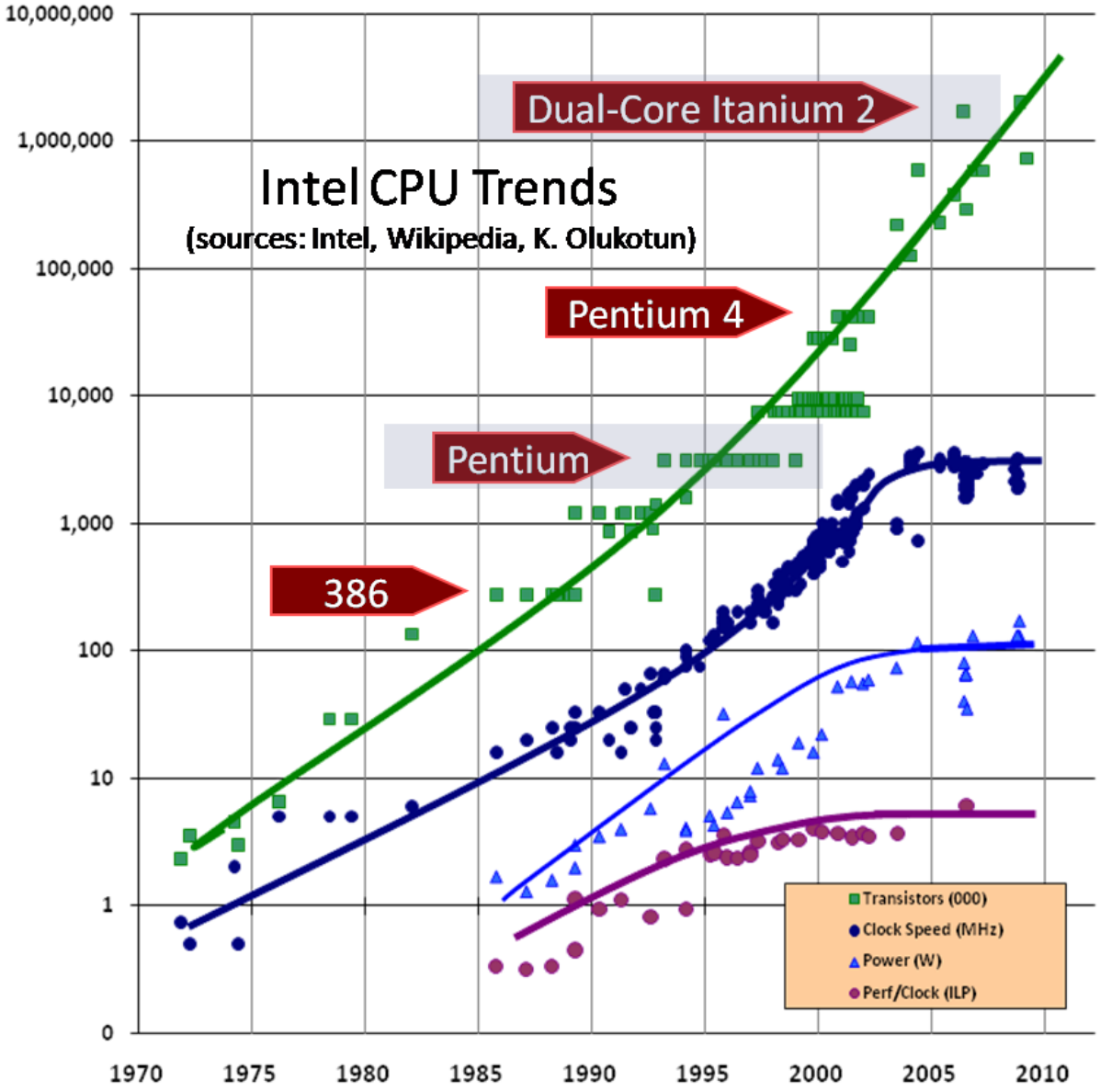


Figure 1: A subset of Intel CPU introductions between 1970 and 2010. [Sut05]

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 increased any further without being 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.

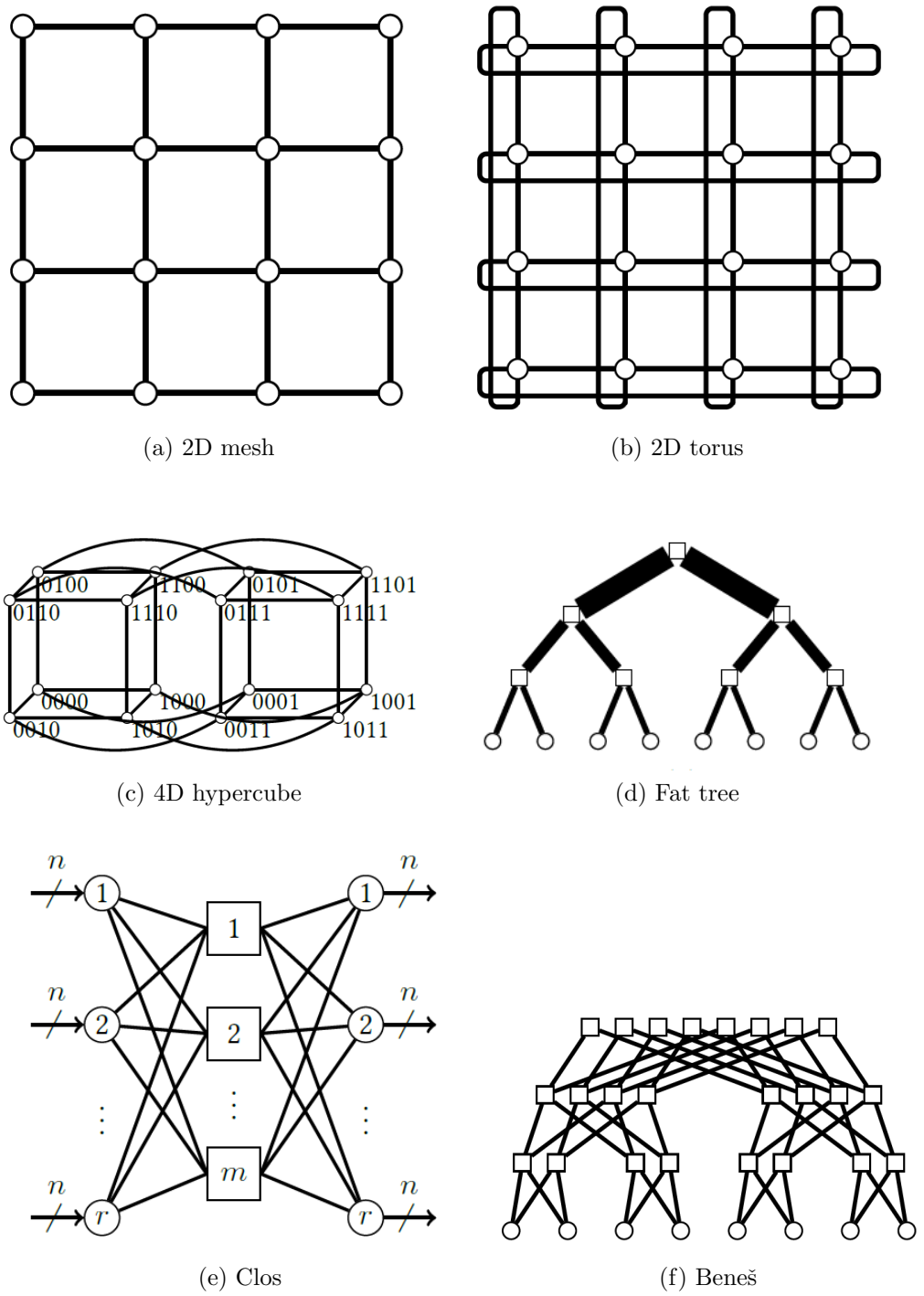
2.4 Interconnection networks

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 [Dal90]. 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 System Specification

4 System Design

5 Implementation

6 System Testing

7 Conclusion

8 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.

[Dal90] 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.

[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.

[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.