SMART: a Simulator of Massive ARchitectures and Topologies

Fabrizio Petrini and Marco Vanneschi

Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, 56125 Pisa, Italy,
tel +39 50 887228, fax +39 50 887226
e-mail: {petrini,vannesch}@di.unipi.it

Abstract

Many important results in the area of computer architecture have been achieved using simulators. In this paper we present SMART, a simulator of parallel architectures. SMART provides a flexible and efficient simulation environment that includes the most common interconnection networks and routing algorithms and gives the user basic mechanisms to define the internal structure of the processing nodes. To show the characteristics of SMART, we analyze the relations between the degree of overlapping of the transpose FFT algorithm and the presence of a communication processor on a fat tree and on a bi-dimensional cube that have the same normalized communication bandwidth.

Keywords: simulation tools, parallel architectures, wormhole routing, interconnection networks, all-to-all personalized broadcast.

1 Introduction

The availability of flexible, high-performance simulators is a key factor to make research in computer architecture.

The advantages of sequential simulators stem from three sources: they run on simulated hardware rather than directly on a host architecture, they are implemented as a single process and they can be executed on standard workstations [4].

Simulators are not limited to one target, nor even to existing machines. They can be used to test new interconnection networks, routing algorithms or cache coherency protocols. Unlike real parallel machines, simulators can measure everything. For example network contention, one-way message latency, cache behavior are all difficult to measure on a real machine. All these measurements can be done nonintrusively, without affecting the internal status.

A second set of advantages comes from running sequentially in a single address space. This implies that the simulation is repeatable and that users can exploit sequential debuggers as *gdb* [24]. The single address space provides instant access to all the internal status. Checking a global invariant on a massively parallel architecture is far more difficult and is difficult even on a shared memory multiprocessor.

Unlike supercomputers, workstations and personal computers can be used freely without sending batch jobs that may experience significant queuing delays. Networks of workstations are becoming a viable and cost-effective solution to high performance computing.

On the negative side, there are two main disadvantages: simulators can be slow and require small data sets. Although traditional simulators of parallel architectures are too slow to run real applications [11] [12], new simulation environments as Tangolite [15], the Rice Parallel Processing Testbed [6], Proteus [3] and the Trace Factory [23] are very fast [5].

The amount of main memory on a workstation is indeed a serious limiting factor. This space problem can also become a performance problem when the address space overflows in the swap space. As a consequence, simulators typically promote small data sets.

In this article we present a general-purpose simulator of parallel architectures, called SMART. We had the following goals for our simulator.*Flexibility*: we wanted the simulator to be extremely flexible to study network topologies, routing algorithms and the internal architecture of the processing nodes. It was intended to model the fundamental characteristics of a massively parallel architecture, not to study any particular one. *Speed*: we wanted the simulator to be fast enough to study large machines, with hundreds of processing nodes. *Portability*: the simulator should run on any computing resource available.

The rest of this paper is organized as follows. In Section 2 we briefly explain the salient features of the simulation kernel that is organized into two abstractions levels, the routing level and the node internal structure. Flow control and routing algorithms are specified at the routing level, while the node internal structure can be defined by the user with a collection of communicating lightweight processes. An example of processing node equipped with two processors, one dedicated to the computation of a parallel algorithm and the other to remote communication, is shown in Section 3. In Section 4 we we study the relations between a parallel algorithm, the transpose FFT algorithm, and the type of interconnection network. Some concluding remarks are given in Section 5.

2 The simulation kernel

The simulation kernel is designed following an object-oriented approach. We start from basic C++ classes and we incrementally add new features through inheritance. We make the assumption that a massively parallel architecture is made of a collection of processing elements or nodes, each one with a particular internal structure. The base class that describes a processing element is the class *node*.

These objects are then assembled together according to a given topology to form a parallel architecture. At the moment SMART supports the following families of topologies: k-ary n-trees [21], k-ary n-butterflies, k-ary n-cubes and the full crossbars [19].

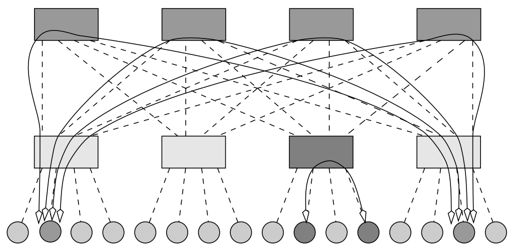

Figure 1: A 4-ary 2-tree. A packet can follow any minimal path passing through a nearest common ancestor of source and destination.

k-ary n-trees [21] are a particular subclass of the fat-trees and borrow from the k-ary n-butterflies [19] the topology of the internal switches. A k-ary n-tree has k^n leaf nodes and n levels of k^{n-1} switches. Each switch has $2k$ links. A 4-ary 2-tree is shown in Figure 1

From the simulation point of view, the processing node can be seen as a collection of communicating entities and can be divided into two main layers or abstraction levels. At the bottom there is the *message routing* level, where message routing and flow control take place. At the top the *node internal structure*, composed by the network interface, processors, memories, I/O devices and glue logic.

2.1 The router

The routing level is implemented using a class, *router*, that embeds all the data structures and implements the flow-control and message routing strategies. At this level the basic entity is the *flit* or flow control digit. When store and forward flow-control is used, the router moves single-flit packets. With wormhole flow control, several flits advance in a pipeline, worm-like fashion.

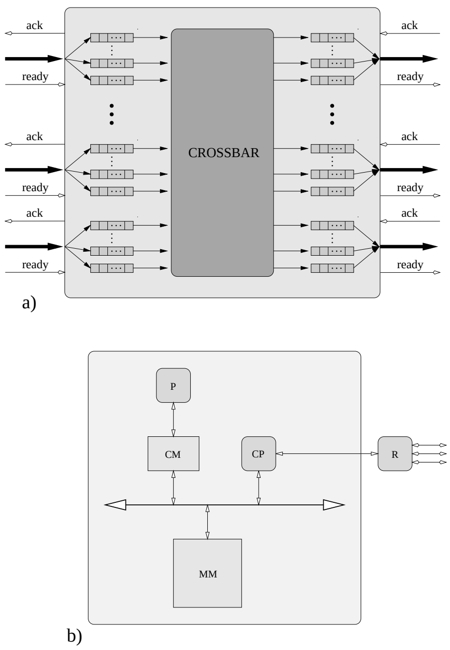

Figure 2: a) The logical structure of a router. b) The internal structure of the processing node and the router. P processor, CM cache memory, MM main memory, CP communication processor, R router.

Figure 2 a) outlines the logical structure of a router. We can distinguish the external channels or links, the input and the output buffers or lanes that implement the buffer space of the virtual channels and an internal crossbar. The switch has a set of bidirectional channels and each channel on the single direction is logically composed of three interfaces: a *data path* that transmits messages on a flit level, the *ready* lines that flag the presence of a flit on the data path and specify the virtual channel where the flit is to be stored and the *ack* lines in the reverse direction that send an acknowledgment every time buffer space is released in the input lanes.

SMART provides a rich library of routing algorithms. We have three distinct minimal algorithms, deterministic randomized and adaptive for the the k-ary n-trees and k-ary n-butterflies [21].

k-ary n-cubes are deadlock-prone and require sophisticated algorithms to route messages adaptively. We implemented some of the most popular ones. There are algorithms based on the Roscoe's ring protocol, several variants of wormhole and cut-through algorithms based on Duato's methodology [10], Cypher and Gravano's algorithm [8], the Chaos routing [2] and many other algorithms built using these basic solutions [20] [22].

In all these algorithms we can finely tune the flow control, buffer size, virtual channels and so on, potentially generating a wide set variants. Also, we can easily implement and analyze new routing algorithms by simply defining the routing function.

2.2 The node internal structure

The past few years have seen the introduction of many different types of parallel architectures. Many commercial machines, as the Cray T3D [7] and the Intel Paragon [16], are multicomputers that have nodes with multiple processors and adopt low-dimensional wormhole-routed cubes as their interconnection networks. Some nodes are specialized to handle parallel I/O. Other important research prototypes, as the Stanford FLASH, provide a cache-coherent shared address space and have communication processors optimized to implement the coherency protocols. COMA machines are another kind of shared memory multiprocessors with peculiar architectural requirements [17] [14].

At the moment there is no clear winner between all existing architectural approaches. For this reason we decided to provide basic primitives to model the desired machine rather than having a fixed internal structure. We didn't want to tie the design of our simulator to a peculiar architecture.

The internal structure is defined by a set of communicating processes that exchange information through message passing and accessing shared variables. These processes are used to model hardware devices, as cache or main memory, or user processes that run on a simulated processor. It is also possible to serialize the activity or these processes to simulate multithreading.

Process-driven simulation has the advantage of being very flexible and modular: we can easily add new elements in the simulation or change the existing ones. On the negative side, process-driven simulation is slower than event-driven simulation and requires more room to allocate the stacks of the processes. An efficient implementation of these mechanisms is crucial to achieve a good performance.

The heart of the simulation kernel is based on the QuickThreads package [18]. QuickThreads is a thread package written in assembler that achieves flexibility by stripping all but the essential operations, leaving just thread initialization and context switch. It runs on most existing platforms, including the Sparcs, Digital Alpha, x86, Mips, KSR1 and HPPA and on all parallel machines that are binary compatible, as the Cray T3D, the Connection Machine CM-5 and the Meiko CS-2. The typical speed of QuickThreads is in the order of the millions of context switches per minute and the basic memory requirement per thread is only a few hundred bytes.

Each lightweight process of the simulation is mapped on a QuickThreads thread. To reduce the simulation overhead of the interconnection network, all the activities at the routing level are managed by a single thread, called *simulation_engine*.

3 An example of internal structure: the dual processor

Starting from the simulation kernel, we have implemented a processing node, defined *dual processor*, with two processors, a compute processor and a communication processor that supports the remote communication primitives.

The internal structure of the dual processor is shown in Figure 2 b). We distinguish the following units: the *main processor*, its *cache memory*, the *main memory* and the *communication processor*. Bus, cache memory, main memory and the processes running on the two processors are implemented with threads.

The communication processor is dedicated to message handling and acts as a Message Driven Processor [9]. It fragments outgoing messages in packets that are enqueued in a FIFO directed to the router and receives incoming packets from another FIFO. The communication processor can perform DMA-like activities and store the content of incoming messages in proper memory locations. There is a simple coherency mechanism between the cache memory and the communication processor. The cache controller snoops on the main bus, and every time a memory location is overwritten by the communication processor, it invalidates the corresponding cache line, if present in the cache memory. Symmetrically, when the communication processor tries to read a memory location replicated on the cache memory, it gets the updated copy.

4 Experimental results

With a simulator as SMART it is possible to study the interactions between hardware and software, with the end goal of obtaining a better match between a parallel computing system and the program it executes.

The transpose algorithm is a well known approach to parallelize the computation of the FFT [13]. It is organized in two distinct phases, where data are locally available, separated by a global communication phase that can be implemented with an *all-to-all personalized broadcast* (AAPB). One of the most interesting features of the transpose algorithm is that, by properly scheduling the local computation, it is possible to overlap the first phase of the computation with of the AAPB.

The relations between the degree of overlapping and the presence of a communication processor can be investigated using the dual processor architecture. In our experiments we compared the execution of a 65536-input butterfly on two networks with 256 nodes, a 16-ary 2-cube and a 4-ary 4-tree.

A fair comparison of interconnection networks should take into account physical constraints as the pin count, wire delay and bisection width [1]. In our experiments we normalized the communication performance by setting the flit and the data path size on the fat-tree at one byte and at two bytes on the toroidal cube. In both cases the flit transmission delay is set to a single cycle. This normalization can be interpreted in the following two ways.

Technological constraints limit the number of pins on a given chip. In a quaternary fat-tree, the arity of the routing switches is eight, while the arity of the routing chip in a bi-dimensional cube is four, if we do not consider the connection with the local processing node. By doubling the data paths on the cube we have the same pin count on both routing chips.

The global communication bandwidth of an interconnection network is given by the product of the physical links for the link bandwidth. Both k-ary n-tree and the toroidal k-ary n-cubes have $n * k^n$ links. The quaternary fat-tree has got twice as many links as a bi-dimensional torus and our normalization equalizes the overall communication bandwidth.

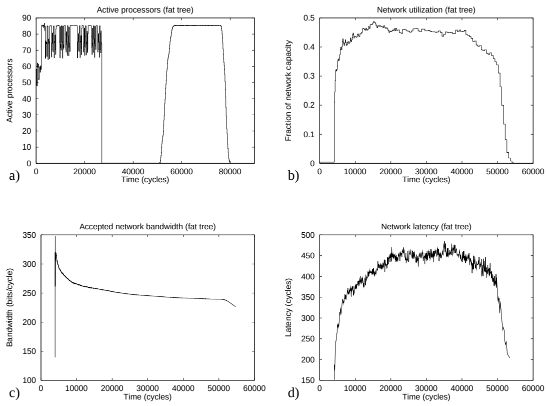

Figure 3: The FFT transpose algorithm on a 4-ary 4-tree

The behavior of the transpose algorithm on the 4-ary 4-tree using adaptive routing with two virtual channels [21] is shown in Figure 3. The transpose algorithm is executed in 80000 cycles, with a clean separation between the two computational phases of 23000 cycles. Computation and communication proceed in parallel for more than 20000 cycles.

We can see that the network throughput remains stable around 250 bits per cycle during the execution of the communication pattern. This corresponds to 50% of the asymptotic throughput, as it is confirmed by network utilization that is close to 50%. In the fat-tree the network utilization is proportional but slightly less than the network throughput, because the switches farther from the processing nodes are not fully utilized under uniform traffic.

From Figure 3 d) we can see that the network latency is stable at 450 cycles, only twice the base latency without contention.

The execution time on the bi-dimensional

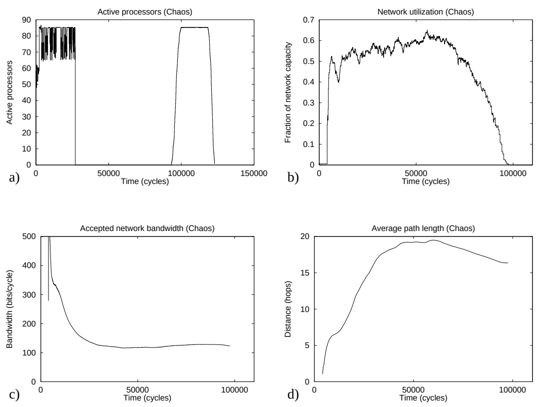

Figure 4: The FFT transpose algorithm on a toroidal 16-ary 2-cube with the Chaos routing.

cube is 123000 cycles, using the Chaos routing [2], a non minimal cut-through version of the hot potato routing. The network utilization reaches 63% of the network capacity, even if the throughput is only 125 bits/cycle, which amounts to 25% of the network capacity, as shown in Figure 4 b) and c). The gap between the network utilization and throughput in the Chaos routing can be explained looking at the graph in Figure 4 d). At saturation, many packets are derouted and temporally sent away from their minimal path. In fact the average path length reaches 20 hops, more than twice the topological average distance of 8 hops.

5 Conclusion and future work

In this paper we have described the main features of SMART, a simulator of massively parallel architectures. The main goal of the research reported here was to build a flexible tool to investigate the relationships between the characteristics of parallel algorithms and different parallel computer organizations. SMART can be used to study the effect on system performance of both major and minor changes in the structure of a parallel architecture. For a given architecture, it can be used to evaluate the relative performance of different parallel algorithms as well as the effect of minor changes in a given algorithm.

SMART is organized into two abstraction levels, the simulation kernel and the node internal structure. The simulation kernel models a parallel architecture as a collection of processing nodes connected in a given topology. It provides the flow control and routing policies and includes the most popular families of interconnection networks and routing algorithms.

The internal structure is defined by a set of communicating processes that exchange information through message passing and accessing shared variables. These lightweight processes are used to model hardware devices or user processes that run on a simulated processor. Rather than having a fixed structure, SMART provides basic primitives to build the desired architecture. We simulated on top of the kernel a node architecture with a compute and a communication processor interconnected by a shared bus.

In the experiments of Section 4 we mapped the transpose FFT algorithm on a fat tree and on a toroidal cube, both having 256 nodes. The dual processor simulation model has taken into account physical constraints as the global communication bandwidth and the router complexity to equalize the asymptotic communication performance of the two networks. The experimental results have shown that the fat tree provides a stable network throughput that is 50% of the asymptotic bound. With the Chaos routing, the toroidal cube cannot do better than 25%. This type of comparison is very difficult to make with real machines, given the widely different architectural characteristics of existing parallel architectures.

References

[1] A. Agarwal. Limits on Interconnection Net-

work Performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[2] K. Bolding. *Chaotic Routing: Design and Implementation of an Adaptive Multicomputer Network Router.* PhD thesis, University of Washington, Department of Computer Science and Engineering, Seattle, WA, July 1993.

[3] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, MIT, September 1991.

[4] E. A. Brewer and W. E. Weihl. Developing Parallel Applications Using High-Performance Simulation. In *1993 Workshop on Parallel and Distributed Debugging*, San Diego, California, 1993.

[5] S. Chittor, R. Enbody, and T.-S. Jou. High-Performance Simulator for Large Wormhole-Routed Networks. *International Journal in Computer Simulation*, 2:151–174, 1992.

[6] R. G. Covington, S. Dwarkadas, J. R. Jump, J. B. Sinclair, and S. Madala. Efficient Simulation of Parallel Computer Systems. *International Journal in Computer Simulation*, 1:31–58, 1991.

[7] Cray Research Inc. *Cray T3D System Architecture Overview*, 1^{th} edition, September 1993.

[8] R. Cypher and L. Gravano. Storage-Efficient, Deadlock-Free Packet Routing Algorithms for Torus Networks. *IEEE Transactions on Parallel and Distributed Systems*, 43(12):1376–1385, December 1994.

[9] W. J. Dally et al. The Message-Driven Processor. *IEEE Micro*, pages 23–39, April 1992.

[10] J. Duato. A Necessary and Sufficient Condition for Deadlock-Free Adaptive Routing in Wormhole Networks. *IEEE Transactions on Parallel and Distributed Systems*, 6(10):1055–1067, October 1995.

[11] S. R. Goldschmidt. *Simulation of Multiprocessors: Accuracy and Performance.* PhD thesis, Stanford University, 1993.

[12] S. R. Goldschmidt and H. Davis. Tango introduction and tutorial. Technical report, Stanford University, Computer Systems Laboratory, 1990.

[13] A. Gupta and V. Kumar. The Scalability of FFT on Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, August 1993.

[14] S. Haridi and E. Hagersten. The Cache Coherence Protocol of the Data Diffusion Machine. In *PARLE'89, Parallel Architectures and Languages Europe*, volume I, pages 1–18, June 1989.

[15] S. A. Herrod. Tango Lite: A Multiprocessor Simulation Environment. Stanford University, Computer Systems Laboratory, November 1993.

[16] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, Inc., 1993.

[17] Kendall Square Research. *Technical Summary*, 1^{th} edition, 1991.

[18] D. Keppel. Tools and Techniques for Building Fast Portable Threads Packages. Technical Report UW-CSE 93-05-06, University of Washington, Department of Computer Science and Engineering, May 1993.

[19] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, San Mateo, CA, USA, 1992.

[20] F. Petrini and M. Vanneschi. Minimal vs. non Minimal Adaptive Routing on k-ary n-cubes. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, volume I, pages 505–516, Sunnyvale, CA, August 1996.

[21] F. Petrini and M. Vanneschi. k-ary n-trees: High Performance Networks for Massively Parallel Architectures. In *Proceedings of the 11th International Parallel Processing Symposium, IPPS'97*, pages 87–93, Geneva, Switzerland, April 1997.

[22] F. Petrini and M. Vanneschi. Performance Analysis of Minimal Adaptive Wormhole Routing with Time-Dependent Deadlock Recovery. In *Proceedings of the 11th International Parallel Processing Symposium, IPPS'97*, pages 589–595, Geneva, Switzerland, April 1997.

[23] C. A. Prete, G. Prina, and L. Ricciardi. A Trace-Driven Simulator for Performance Evaluation of Cache-Based Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(9):915–929, September 1995.

[24] R. M. Stallman and R. H. Pesch. *Debugging with GDB.* Free Software Foundation, January 1994. version 4.14.