Predictive Block Dataflow Model

for Parallel Computation

Final Report

EE382C: Embedded Software Systems

Prof. B. L. Evans

Submitted by :

Vishal Mishra and Kursad Oney

May 8, 2002

Abstract

Dataflow architecture as a concept has been around since the 1970s for parallel computation. In dataflow machines, the hardware is optimized for fine-grain data-driven parallel computation. The architecture can tolerate long unpredictable communication delays and supports generation and coordination of parallelism directly in the hardware. We propose a new coarse-grained dataflow model of computation for parallel architectures named *Predictive Block Dataflow (PBD)* due to its predictive capacity and processing of instructions in blocks instead of the fine grain model of one machine instruction. We have simulated the high level machine architecture and implemented two algorithms, quicksort and Dijkstra's single source shortest path, to display the high degree of parallelism that our model can exploit in both highly recursive and irregular data structures.

1 Introduction

Dataflow is an intuitively appealing, simple yet powerful model of parallel computation. In dataflow architectures there is no concept of a program counter corresponding to conventional sequential program models. The dataflow model describes computations in terms of locally controlled events, where each event corresponds to the "firing" of a block. A block can be a single instruction or a sequence of instructions. A block fires when all of the inputs it requires are available.

Dennis [1] developed the model of dataflow schemas, building on work by Karp and Miller [2]. These dataflow graphs evolved from a method for designing and verifying operating systems to a base language for a new architecture. The first designs for such machines were made at MIT in the early 1970s by Dennis and Misunas [3] and Rambaugh [4]. These designs use dataflow graphs [5] to represent and exploit the parallelism in programs.

The failure of the original dataflow machines led to a new wave of hybrid multithreaded machines. It is argued that von Neumann and dataflow machines are not, in fact, orthogonal but rather sit on opposite ends of a spectrum of architectures. These hybrid architectures combine features of von Neumann and Dataflow.

We believe that the recent advances in chip fabrication have once again made the dataflow concept important for massively parallel architectures (MPA). The transistor size is becoming smaller and we can fit more transistors on a single chip. This allows us to have multiple processors with very low communication delays on one chip. Also, presently most programs have been implemented for sequential machines and do not exploit the inherent parallelism in programs. On the other hand, dataflow architectures expose parallelism in programs and are ideally suited for parallel programming models. We propose a model comprising of a 2-D multiprocessor array configuration on a single chip. In section 3 we describe our model of computation, which is a dynamic dataflow model with self scheduling nodes. Section 4 describes our simulation model along with the implementation details. We present the results of our simulation of quicksort and Dijkstra's single source shortest path *SSSP* in Section 5 and finally the conclusions are laid out in Section 6.

2 Background Research

Dataflow research made great strides since the seminal paper on dataflow graphs by Dennis [5]. In a Dennis dataflow graph, operations are specified by *actors* (or *nodes*) that are enabled only when all the actors that produce required data have completed their execution. The dependence relationships between pair of actors are defined by the arcs of a graph, which represent results forwarding by an actor to successor actors, and by the *firing rules*, which specify exactly what data is required for an actor to fire.

There are two forms of dataflow architecture; static and dynamic. In static dataflow the arc connecting one instruction to another can contain only a single result value (a token) from the source instruction. There can be only one instance of a dataflow actor in execution at any time. Thus concurrent reactivation of nodes (or actors) is not permitted, and hence, static systems are restricted to implementing loops and cannot accommodate recursion. In dynamic dataflow, code-copying or dynamic tagging is used to permit recursive reactivation. In dynamic tagging, tags are conceptually or actually associated with tokens so that tokens associated with different activations of an actor may be distinguished. This enables arcs to simultaneously carry multiple tokens, thereby exposing the existing parallelism.

Due to the inherently parallel nature of dataflow execution, dataflow computers provide an elegant solution to the two fundamental problems [6] of von Neumann computers, namely memory latency and synchronization overhead. Latency is tolerated by dynamic switching between ready computation threads, whereas synchronization is supported in hardware, which has low overhead. Our model employs some ideas from three well researched early dataflow machines. The MIT *Tagged Token Dataflow Architecture (TTDA)* [7] has *I-structure* storage units, which reside in global memory and can be addressed in *global address space*. The main characteristic of TTDA is that each token is tagged with a *context* identifier (format $< context, destination\_instr., data >$) that specifies the activation to which the token belongs. These tokens are matched at the destination node by *wait-match* units, which act as rendezvous points for pairs of arguments for dyadic operators. The *Manchester Prototype* Dataflow Computer [8] also employs dynamic tagging like TTDA for identifying *iteration level* and implementing recursion by matching tokens by an associative lookup. The *Monsoon* machine replaces the associative waiting-matching store of the above machines by an explicitly managed directly addressed token store [9]. It is an *Explicit Token Store (ETS)* model with the central idea that storage for tokens is dynamically allocated in sizable blocks. When a function is invoked, an *activation frame* is allocated explicitly to provide storage for all tokens generated by the invocation. A token now comprises of a value, a pointer to the instruction to execute (IP) and a pointer to an activation frame(FP).

It has been speculated that there exists some optimum point between the two extremes *i.e.* a new *hybrid model* which synergistically combines features of both von Neumann and Dataflow, as well as exposes parallelism at a desired level.

Based on his research on the MIT Dynamic TTDA and the experience gained by [10], Iannucci combined dataflow ideas with sequential thread execution to define a hybrid computation model [11]. *P-RISC* [12] explores the possibility of constructing a multithreaded architecture around a RISC processor. Nikhil and Arvind's P-RISC model split the *complex* dataflow instructions into separate synchronization, arithmetic and fork/control instructions, eliminating the necessity of presence bits on the token store (or frame memory) as proposed in the Monsoon machine [9]. *StarT* [13] is a successor of the Monsoon project and retained the latency-hiding features of the Monsoon split-phase global memory operations while being compatible with conventional von Neumann MPA's. Inter-node traffic consists of a *tag* (called *continuation*, a pair comprising a context and instruction pointer). All inter-node communications are performed using the *split-phase* transactions (request and response messages); processors never block when issuing a remote request, and the network interface for message handling is well integrated into the processor pipeline. A separate co-processor handles all the responses to remote requests.

3 PBD: Model of Computation

Our proposed model is basically a dynamic dataflow architecture with self scheduling nodes. It is a 2-D mesh of processors with local memory connected by a network architecture that has not been determined. This processor array executes a dataflow graph like other DDF systems; i.e., a node is enabled when all the input data arrives on the input arcs. This schedules firing of the node and a token is sent on the output arc. There are many differences with respect to the early dataflow systems covered in Section 2. It is a coarse-grained system as a block of instructions is broadcast for execution to every node instead of a single instruction. There is an asynchronous decoupled co-processor that pre-fetches the "block" of instructions based on a separate program produced by the compiler and then broadcasts or multicasts to the processor array. This program is based on the instruction dependence graph of the program. The block size is not fixed and block boundaries are defined by unpredictable latency operations. PBD is a message passing multiprocessor system, where *tokens* are passed. The fields of any token include:

$$< destination\_node\_id, instruction\_pointer, value, context\_id >$$

This resembles the approaches followed in TTDA and Manchester with respect to dynamic tagging for context identity. All the processors are data-driven like a dataflow architecture with another subtle variation called the *owner computes rule*, which means only those nodes will be activated that have the data in their local memory. Our model makes all data accesses local and excludes all remote data requests unless required. The operations are totally asynchronous and the need for synchronization is minimal. Synchronization can be performed on a need basis and the synchronization overhead is tolerated. This is not a bottleneck as dataflow models have low synchronization overhead as explained in section 2. The memory is addressed uniformly in a *global address space*. It is simulated using the processor ID as the most significant byte and the local memory in the node as the least significant byte. There is a block level locality in this model unlike the fine-grained models which only had instruction level locality. This can be effectively employed for pipelining within each node.

4 Simulation Model and Implementation

4.1 Work-Depth Model

In parallel processor-based simulation models, performance is measured in terms of the number of instruction cycles a computation takes and is usually expressed as a function of input size and number of processors. Work-Depth model is a virtual formal model that defines performance in more abstract measures than just running time on a particular machine. The model consists of a pair of measures, work and depth. Work is defined as the total number of operations executed by a computation, and depth is defined as the longest chain of sequential dependencies in the computation. In figure 1 the work required for summing n numbers on a balanced tree requires is 15 and the depth is 4.

Work is usually viewed as a measure of the total cost of a computation. The depth represents the best possible running time, assuming an ideal machine with an unlimited number of processors. A problem with using work and depth as cost measures is that they do not directly account for communication costs, which is not a bottleneck for our model.

4.2 Machine Simulation Model

We have implemented a message driven dataflow processor array in C++. The processors in the array communicate with each other via messages or *tokens* in our system. The implementation has used high-level features like STL, function objects, templates and callbacks provided by C++ for simulation. Figure 2 shows the high-level model. We briefly describe the design of the basic units and their implementation in our simulation.

• **Token:** It has been implemented as a function object in C++. The format is as explained

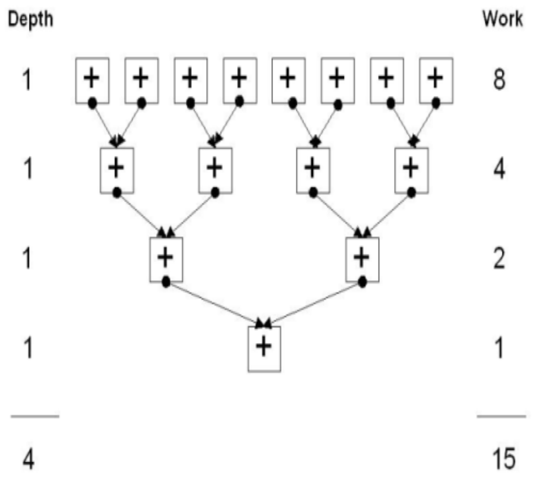

Figure 1: Work-Depth example

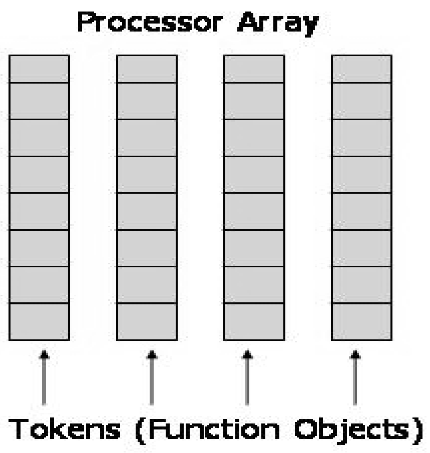

Figure 2: Processor Array Model

in Section 3. The instruction pointer is a function member pointer to a member function of the class to which the object belongs. Each token represents a unit of work. Tokens are queued up at processor and executed and represent the data in the dataflow model.

• **Processor:** A processor has been designed as a class with a FIFO queue of tokens. Received tokens are placed in the FIFO queue. Each token takes a unit time to execute on a processor. The instructions to be executed are referenced by a function pointer, which is part of the token. Like a dataflow system, once execution is complete, the generated tokens are sent to the next destination processor object.

• **Processor Array:** It has been implemented as a STL vector of processor objects. The number of processor is fixed along with the memory allocated to each for data storage, which in our simulation is the size of the data object e.g. tree, graph. The fixed address space concept for each processor has been modeled by dividing the address of the processor object with the memory allocated and applying the modulo over the number of processors. This effectively simulates global address space. The randomness of the communication process has been built in by randomly placing the tokens in the FIFO queue. The degree of parallelism is represented as the number of tokens that have been slated for execution at the same level (index) in the FIFO queues of the processors. The depth in the longest chain of execution any processor, whereas the sum of the tokens executed by all the processors represents the work done by the system.

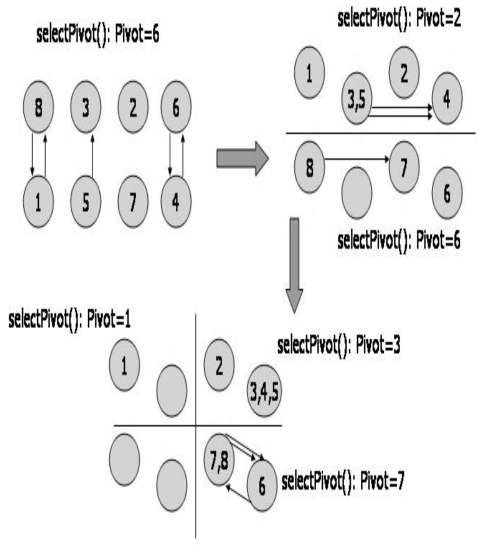

Figure 3: Quicksort Model

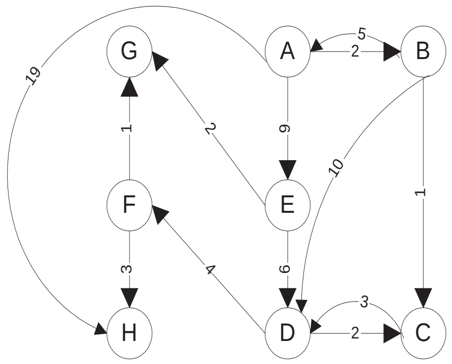

Figure 4: Graph used for Dijkstra's SSSP

5 Algorithm Simulation Results

We modeled quicksort [14] and Dijkstra's SSSP [14] and executed them to get analytical results in terms of the work-depth model.

5.1 Quicksort

Quicksort is a highly recursive algorithm that suits dataflow programming model. We divided the array uniformly over our processors and implemented parallel partitioning by simulating *pivot broadcast*. Figure 3 depicts the working of the model. Also, the standard parallel execution of the two recursive calls at each stage were simulated. The results have been presented in Table 5.1. We have compared our results to the analytical results for work, depth, maximum *degree of parallelism (DoP)* and average DoP. The analytical work for quicksort is $n * log_2 n$ and the depth is $log_2 n$. The maximum degree of parallelism is the maximum parallel operations existing at any level in the algorithm. The average degree is calculated as the $work/depth$ in our simulation. The depth value for the simulation is different as the longest chain of sequential dependencies will be higher due to communication work and non-uniform data division unlike the analytical model. The results show a higher work load due to the extra work done for broadcast and communication but they prove our main point by exploiting the inherent parallelism. The average and maximum degree of

	Analytical				**Simulation**			
No of Elements	Work	Depth	Max DoP	Avg DoP	Work	Depth	Max DoP	Avg DoP
8	24	3	8	4.7	35	6	8	5.8
16	64	4	16	7.5	76	10	16	7.6
32	160	5	32	12.4	187	13	8	14.3

Table 1: Quicksort Simulation Results

parallelism are very close to the analytical model. The average degree shows a higher value for the simulation for 16 and 32 elements as the work load increases. Thus the simulation shows that the computation scales with the data size. At 8 elements, the overhead work forms a bigger ratio of the total work but this ratio reduces with more data.

5.2 Dijkstra's SSSP

Theoretically the most efficient sequential algorithm to calculate single source shortest path (*SSSP*) on digraphs with non-negative edge weights is Dijkstra's algorithm. For a directed graph, $G = (V, E)$ with $|E| = m, |V| = n$, its running time is $O(nlog(n) + m)$. There is no parallel $O(nlog(n) + m)$ work *PRAM* (Parallel Random Access Machine) algorithm with sublinear running time. The best $O(nlog(n)+m)$ work solution has running time $O(nlog(n)$. We studied the PRAM algorithm presented in [15] and have implemented a highly modified version of it for our simulator. We used the graph shown in Figure 4 for our simulation runs. The number of nodes, $n = 8$ and the number of edges, $m = 13$, give us sequential $work = 37$ and $depth = 8$ as each node has to be extracted from the priority queue. We basically parallelized the *relaxation* [14] calls to the adjacent nodes of the node extracted from the priority queue at every step. Our values for the run were $work = 46$, $depth = 8$, $maxmimumDoP = 3$ and $averageDoP = 2.1$. The low value of average DoP is due to the small number of nodes and edges in the graph and not enough existing parallelism that can be exploited. We need to run the algorithm on more graphs as we cannot benchmark the results for runs on a single graph. Also, the algorithm design requires more fine-tuning to extract more parallelism.

6 Conclusions

The more recent superscalar microprocessors emulate dataflow machines. The control flow program is converted to a dataflow graph in the instruction window and then a reorder buffer retires the instructions in order. In a dataflow model like ours, the entire program is a dataflow model to exploit as much parallelism as possible. We believe it is more effective to expose the dataflow as a programming model than converting sequential instructions to and from dataflow during execution.

We have proposed the model of computation and implemented the basic parallel dataflow machine architecture simulation in C++. We also designed the algorithms for quicksort and Dijkstra's SSSP and implemented them on our simulator. Quicksort is a recursive and easily parallelizable algorithm and produced very good preliminary results. Dijkstra's SSSP is not recursive but represents a irregular data structure and is hard to parallelize. The present design, though not perfect, did produce reasonable results which show promise. More work needs to be done on the design and implementation of the algorithms before detailed analysis can be carried out. Other recursive algorithms need to be implemented and their performance need to be studied in detail.

This model exposes the parallelism in recursive programs and provides flexibility in terms of the data structures used. It is capable of handling irregular data structures like trees and graphs unlike the vector processors which only performed efficiently for vector data structures. The broadcast mechanism enables execution of multiple instances of the same function and multiple loop executions concurrently.

References

[1] J. B. Dennis, "Programming generality, parallelism and computer architecture," in *IFIP Congress*, vol. 1, pp. 484–492, 1968.

[2] R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal of Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, 1966.

[3] J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," in *Proc. IEEE/ACM International Symposium on Computer Architecture*, pp. 126–132, 1975.

[4] J. E. Rumbaugh, "A Data Flow Multiprocessor," *IEEE Computer*, vol. 26, no. 2, pp. 138–146, 1977.

[5] J. B. Dennis, "First version of a data-flow procedure language," in *Proc. of the Colloque sur la Programmation*, vol. 19, pp. 362–376, Springler-Verlag, 1975.

[6] Arvind and R. A. Ianucci, "Two Fundamental Issues in Multiprocessing," in *Proc. DFVLR Conference on Parallel Processing in Science and Engineering*, pp. 61–88, 1987.

[7] Arvind, D. Culler, R. Iannucci, V. Kathail, K. Pingali, and R. E. Thomas, "The Tagged Token Dataflow Architecture," *Technical Report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139*, 1984.

[8] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Communications of the ACM*, vol. 28, January 1985.

[9] G. M. Papadopoulos and D. E. Culler, "Monsoon: An explicit token store architecture," in *Proc. IEEE/ACM International Symposium on Computer Architecture*, pp. 82–91, 1990.

[10] Arvind and R. S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Computer*, vol. 39, no. 3, pp. 300–314, 1990.

[11] R. Ianucci, "Toward a Dataflow/von Neumann Hybrid Architecture," in *Proc. IEEE/ACM International Symposium on Computer Architecture*, pp. 131–140, 1988.

[12] R. S. Nikhil and Arvind, "Can dataflow subsume von Neumann computing?," in *Proc. IEEE/ACM International Symposium on Computer Architecture*, pp. 262–272, 1989.

[13] R. Nikhil, G. M. Papadopoulos, and Arvind, "*T: A multithreaded massively parallel architecture," in *Proc. IEEE/ACM International Symposium on Computer Architecture*, pp. 156–167, 1992.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms.* MIT Press and McGraw-Hill Book Company, 6th ed., 1992.

[15] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders, "A parallelization of Dijkstra's shortest path algorithm," in *Proc. Mathematical Foundations of Computer Science*, vol. 1450, pp. 722–731, Lecture Notes in Computer Science, Springer-Verlag, 1998.