Evaluation of Active Graph Applications on the Cray Eldorado Architecture

Shannon Kuntz, Peter Kogge, Jay Brockman, and Matthias Scheutz
University of Notre Dame

Gary Block and Mark James
NASA JPL

John Feo
Cray Inc.

Ed Upchurch
California Institute of Technology

May 5, 2006

Abstract

In this paper, we discuss an approach to such problems called an "active graph", where each node in the graph is tied to a distinct thread and where the flow of data over the edges is expressed by producer/consumer exchanges between threads. We will show how several cognitive applications, including a neural network and a production system, may be expressed as active graph problems and provide results on how active graph problems scale on both the MTA-2 and Eldorado architectures.

1 Introduction

As a recent National Academics report declared, "The peak performance of supercomputers has increased rapidly in the last decades, but their sustained performance has lagged, and the productivity of supercomputing users has lagged" ([1] page 5). As true as this may be for numeric, floating-point, problems, there are many emerging applications for very large, high end, ground-based systems with growing problems with irregularly structured and dynamically changing data, increasing high bandwidth and real-time I/O, 24x7 availability, and an increasing fragility in our ability to program, manage, and optimize their operation. Such applications will be found with web serving and searching, multi-sensor data collection fusion and distribution, intelligence analysis, data mining, knowledge discovery, catastrophe management, and homeland security. To solve such problems we must shift our thinking about architectures for such high end use from flops to memory, from complex human tuning to autonomy, from batch to real-time continuous operation, and from preplanned application scheduling to large amounts of self-management, learning, and planning. These are significantly different in behavior than the typical application studied by computer architects. The NRC report [1] gives several specific examples of such problems and the characteristics of the resulting processing. Signals Intelligence fall into two categories, intelligence processing and intelligence analysis, and place heavy emphasis on bit level processing, computing in nonstandard algebras, random access to extremely large data sets over large distributed memories, and processing of extremely large and dynamic graphs. Bioinformatics and Computational Biology is seeing the combination of equations-of-physics simulations with massive-data-driven computations, statistical data processing, data mining, and pattern recognition. Human/Organizational Systems Studies will need to simulate up to billions of independent agents for long time periods. Web searching, data mining, and knowledge discovery similarly involve the dynamic construction, searching, and updating of massive structures, often time-correlated. Virtually all of these involve graph problems of one form or another.

This paper addresses this emerging class of very high end applications that challenge even the most modern of conventional parallel systems designs. Such applications typically involve large, often randomly interconnected, graphlike data structures where processing is node-centric, potentially highly concurrent, and driven by the interchange of data between nodes. The class of architectures best suited to this set of applications are massively multi-threaded in nature, where each thread requires a bare minimum of machine state, the manipulation of thread states is a low-cost operation, and low-cost, fine-grain synchronization mechanisms are available. The MTA-2 and Eldorado systems provide just such features and would allow individual graph nodes to be mapped isomorphically to different threads, which then interact with each other as required by graph operations.

The remainder of this paper discusses the active graph model and how such applications would scale on the MTA-2 and Eldorado. Section 2 discusses the active graph model and how different applications can be mapped to this model. The experimental methodology used to evaluate the scaling of active graphs on the MTA-2 and Eldorado is presented in Section 3 and Section 4 discusses the results and observations of our experiments. Finally, in Section 5, we present our conclusions and future work.

2 Active Graphs

We define an active graph application as one based on a graph-like structure of nodes connected by (potentially directed) edges, where in implementation each node is tied to a distinct thread, and where the flow of data over edges is expressed by producer/consumer exchanges between the corresponding threads. We call nodes to which edges go from one node as downstream of that node; nodes with edges directed into a node are termed upstream of that node. Each thread has associated with it some set of dedicated memory locations, registers, or both, with subsets allocated to:

- Input values: containers for values that have been transmitted from other nodes who have edges pointing inwards to this node.
- Working locations: where intermediate computations are done leading to the computation of one or more output values that must be shared with downstream nodes.
- Downstream nodes: the addresses of the appropriate input value locations for downstream nodes, i.e. are to receive output values when they are computed by this node.
- Sync: for each distinct downstream node, a location that may be used by the downstream node to signal this node that the downstream node is ready to accept a new input.
- Upstream nodes: address of a sync location for each upstream node that provides a value.
- Control: a location used by other (usually upstream) nodes to signal a new input is available.
- Management chain: a pointer to the next node in order of creation.

Conceptually, operation is as follows. Some set of nodes are initially active, with the rest blocked. Each of these active nodes computes some value(s) and, when allowed to, proceeds to write the appropriate value to each of its downstream node input location(s), and signal that node that a new value is available. At some point each such active node also fills a sync location in the appropriate upstream node, indicating to that node that it is ready to accept a new input from that node. The active node then looks for an indication that a new computation cycle is already to start, and if not, it will block.

How a node thread unblocks to begin an active phase depends on the firing rules for the graph application. Some applications may require that there be new values deposited in all input locations. In such cases, the thread code can simply proceed down each input and attempt to do an atomic read and empty against that location. If there is no value available when the access is attempted, the thread blocks until the location is filled. On the other extreme, a node thread may wish to be activated whenever any input value is provided. In this case, the thread is programmed to empty the control location when it unblocks, and then interrogate the same location when it has completed its cycle, and wishes to see if any new inputs have arrived in the meantime.

During this time, an AMO to the thread's control location from any of the upstream nodes can signal the current thread that a new input has arrived. The use of an AMO is crucial here, since it is possible that multiple upstream nodes may signal updates before the thread is physically unblocked by the hardware from the first such update. Variations in the AMO from upstream nodes can range from a simple store and unblock, through a bit significant OR to memory that records which input has been provided (and perhaps which kind of processing is to be performed), to an increment/decrement that counts the number of changes. Finally, a practical consideration for such active graphs is the need to perform a variety of housekeeping functions against them, such as reset, read out status, or terminate execution and return all associate storage.

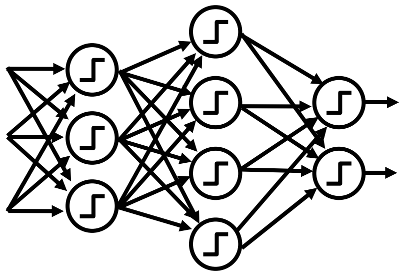
This active graph model can be applied to numerous applications including many in cognitive processing. Cognitive processing has become an increasingly important part of the workload for a number of government agencies but existing applications have typically not scaled well on parallel architectures. Using the active graph model to improve performance for these applications allows us to:

- analyze more complex problems
- perform more complete analysis
- generate results more quickly
- provide more accurate results.

It can also enable solutions to a class of problems that were previously not possible. The following subsections demonstrate how two cognitive processing applications, neural networks and production rule systems, can be implemented using the active graph model.

2.1 Neural Networks

We assume that the target neural net application is one constructed out of an acyclic graph of neurons, where each neuron accepts numeric inputs from some number of other neurons and generates outputs that go to some other set of neuron inputs. Operation of a neuron is as follows. Whenever one or more of the input values change, some function is applied to the current values, and used to compute a new output value. A change in the output value should cause the neuron to relay this value to its target children, where the cycle repeats itself. A typical function involves summing up all input values and comparing to a threshold. For simplicity we assume that the initial type of neuron described here is very asynchronous, that is it is only guaranteed to go through its evaluation cycle at some undetermined point after one of its inputs have changed. We assume no handshake with any of its connected partners; nor do we assume an ordering of evaluation that matches the time-ordering of the changes to the inputs. All of these additional features can be provided by extensions of the basic thread model discussed below.



..

Figure 1: Neural Network

Our goal for such networks is maximum useful concurrency. Individual neurons should consume processing resources only when they have a change in an input, and when multiple neurons see the same change in values that they should be capable of being executed concurrently. We assume an active graph implementation where each neuron is a separate lightweight thread with its own register frame. The neuron thread is established when the neural net is built, and lives until the net is to be shut down. In between, it will cycle between active and blocked, as inputs change.

We assume the following registers and initial values:

- inputs: empty
- signaling register: empty
- outputs: address of matching input to downstream neuron

Each thread then attempts to access its own signaling register with some instruction that blocks on empty, reads the value, and replaces the register value with an empty. If that register is empty, then no upstream neuron has sent a new value, and the thread simply blocks. When a non-empty value is found, the thread is re-awakened, and the instruction that blocked now automatically reempties the signaling register. Having been reawakened, the thread can recompute the output value based on current registers, compare the value against the stored value, go through the fan-out ritual again, and then block on its own signaling register. Note that the use of an AMO from all parents into a child neuron's signaling register means that the child will never miss a change notification, even if it happens for timing reasons to handle multiple changes with a single awakening.

2.2 Production Rule Systems

SHINE [2] is a knowledge base tool that uses dataflow representations of the production rule system to exploit parallelism and, as such, is well suited to an active graph model. We assume the knowledge bases are represented by a set of production rules that can be transformed into an acyclic dataflow graph as shown in Figure 2. Threads are created for each production rule in the system and use a producer/consumer style of interaction via synchronized memory locations to communicate data. Each thread monitors its input values, waiting for them to become full, then reading them and leaving them empty. Once all values have arrived, the rule begins processing and, when complete, the result is synchronously written to the downstream rules. This cycle then repeats for the downstream nodes. In this implementation we assume that a rule fires when all new values have arrived. However, modifications to this basic structure could support rule firing on the change of any input as discussed in the neural network application. This implementation makes significant use of the advantages provided by memory based synchronization support and lightweight multithreading to control the exchange of data between rules and provide data driven execution.

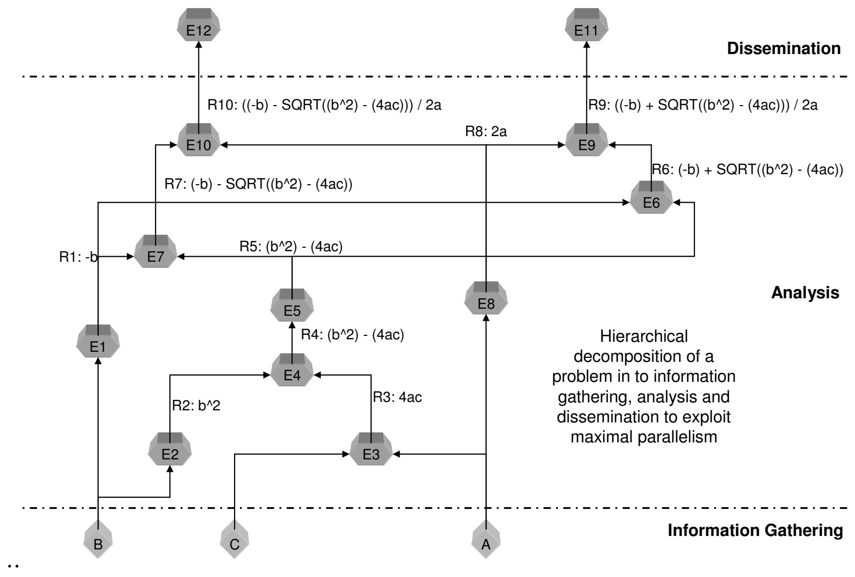

E12
E11
Dissemination
R10: ((-b) - SQRT((b^2) - (4ac))) / 2a
R9: ((-b) + SQRT((b^2) - (4ac))) / 2a
E10
R8: 2a
E9
R6: (-b) + SQRT((b^2) - (4ac))
R7: (-b) - SQRT((b^2) - (4ac))
E6
R1: -b
E7
R5: (b^2) - (4ac)
E5
E8
Analysis
E1
R4: (b^2) - (4ac)
E4
Hierarchical decomposition of a problem in to information gathering, analysis and dissemination to exploit maximal parallelism
R2: b^2
R3: 4ac
E2
E3
B
C
A
Information Gathering

Figure 2: Dataflow Representation of Knowledge Base

3 Experimental Methodology

In this set of experiments we investigate how active graphs scale with size and connectivity on the MTA-2 as well as examining both the productivity and performance provided by using full/empty bit synchronization as opposed to attempting to predetermine the execution ordering of the graph in the code. Active graph structures have many different characteristics depending on the application being explored. In order to evaluate the performance of lightweight multithreading and synchronization for a range of graph sizes and types, we synthetically generate graphs with the characteristics we wish to explore. These synthetic graphs and the associated dataflow code are based on the FFT butterfly, an example of which is shown in Figure 3.

The FFT graph is defined by the number of input nodes, N, and the radix, R. For a graph of radix R with N input nodes there are $S = log_R N$ stages with N nodes per stage for a total of $T = S * R + R$ nodes and $E = N * R * S$ edges in the graph. Graphs with different characteristics are generated for

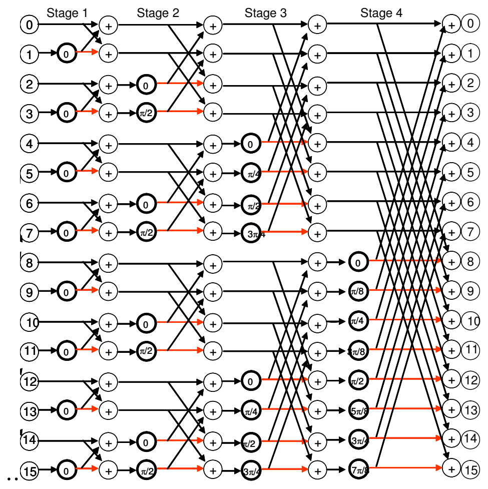


Figure 3: FFT Butterfly Structure

different values of N and R with N correlating to both total graph size and degree of parallelism while R specifies the number of edges incident to each node. Because our focus is on the memory accesses and synchronization aspects of execution, our dataflow code does not implement the complex computations of the FFT at each node but does a simple weighted sum. At each stage, each node in the stage reads the values from its in edges, computes the weighted sum, and writes the result to the out edges for the nodes in the next stage of the graph.

We have developed two versions of this FFT dataflow code to allow us to explore the advantages provided by full/empty bits on the MTA-2. In the first version the flow of data through the graph is controlled by executing each stage in succession while in the second version full/empty bits are used to control the flow of data between nodes for finer grained synchronization and increased parallelism. Figure 4 provides the pseudocode for each.

```
for each data set
  for each stage of the graph
    #pragma mta assert parallel
    for each node in that stage
      read()
      compute()
      write()

 VERSION 1: Staged Dataflow
```

```
for each data set
  #pragma mta assert parallel
  for each node in graph
    readfe()
    compute()
    writeef()

VERSION 2: Full/Empty Bits
```

Figure 4: Pseudocode of main loops for staged dataflow and dataflow with full/empty bits

The staged dataflow code controls the flow of data through the graph by executing one stage at a time assuring that each node has the correct data available from the previous stage when it begins execution. Each node in the stage reads its input data, computes the weighted sum, and writes the result to its out edges. This allows all nodes within a stage to execute in parallel but a node in the next stage cannot begin execution until ALL nodes in the previous stage have completed. The full/empty bit dataflow code uses fine-grained synchronization to control the flow of data through the system and all nodes in the graph are allowed to execute in parallel. Each node in the graph tries to read its input data. If it finds an input to be empty it blocks, waiting for the data to arrive. Once the all input data arrives the node computes the weighted sum and then does a synchronized write to the out edges, blocking if it finds an edge still full. This allows a node to begin processing as soon as it has its input values available as opposed to having to wait for all nodes in the previous stage to complete. Comparing these two implementations allows us to explore the characteristics of active graph applications that make them best suited to the fine-grain synchronization provided by full/empty bits.

The experiments were run on the MTA-2 using 10 processors. Graph sizes were swept from 1024 to 65536 N for both radix 2 and 4 FFT graphs as given in Table 1. Both the staged dataflow code and the full/empty bit dataflow code were run for each of the graphs. Section 5 discuses the results of these experiments.

	FFT radix-2			FFT radix-4		
N	S	T	E	S	T	E
1024	10	11264	20480	5	6144	20480
2048	11	24576	45056	—	—	—
4096	12	53248	98304	6	28672	98304
8192	13	114688	212992	—	—	—
16384	14	245760	458752	7	131072	458752
32768	15	524288	983040	—	—	—
65536	16	1114112	2097152	8	589824	2097152

Table 1: Graph Characteristics

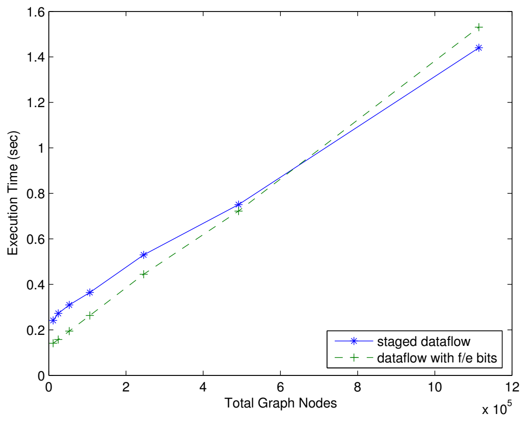
4 Results and Observations

The first set of experiments explored the performance characteristics for radix-2 FFT butterfly dataflow graphs of increasing size. The graphs were generated by doubling the number of inputs at each data point which approximately doubles the total number of nodes in the graph. Inputs ranged from 1024 to 65536 nodes corresponding to 11,264 to 1,114,112 total nodes in the graph as given in Table 1. Figure 5 shows the performance characteristics for these graphs by both the staged dataflow version of the application and the full/empty bit version.

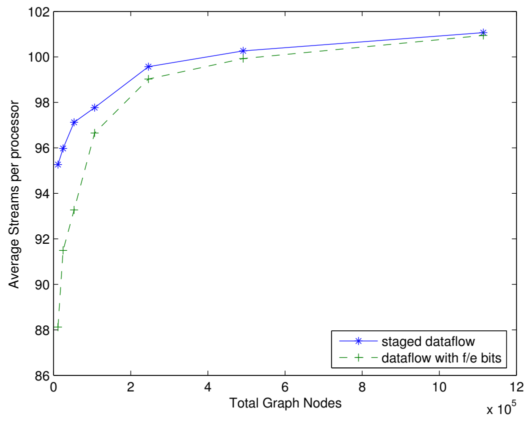
We would expect the execution time to double as the size of the graph doubles. However Table 2 demonstrates that, for the smaller graphs, execution time increases only minimally as the graph size increases. For example, increasing the graph from 11264 to 106496 total nodes increases the execution time for the staged dataflow from 0.242 to 0.364 seconds while the full/empty bit version increases from 0.141 to 0.263 seconds corresponding to less than a 2X increase in execution time for almost a 10X increase in graph size. This occurs because the smaller graphs do not have enough parallelism to saturate the processor and mask the memory latencies and so, as the graph size increases the processor is able to execute more efficiently. This effect is reflected in both the average number of streams per processor and average issue rate per processor presented in Figures 5(b) and (c) respectively. As the number of streams increases from 88 to 100 streams, there is a corresponding increase in issue rate from 0.2 to 0.75. So, as the size of the graph increases more parallelism is gained and execution is interleaved to more fully utilize the processor. Once graph sizes of approximately 500,000 nodes are reached there is a corresponding plateau in both the number of streams and the issue rate. At this point the processor begins to saturate with over 100 streams and reaches approximately 85% utilization and the expected doubling of execution time with doubled graph sizes occurs. So as the size of the active graph and the corresponding available parallelism increases, the performance increases as well until the processor begins to saturate.

N	Staged (sec)	Full/Empty (sec)
11264	0.141	0.242
24576	0.157	0.272
53248	0.194	0.31
106496	0.263	0.364
245760	0.445	0.53
491520	0.722	0.75
1114112	1.531	1.44

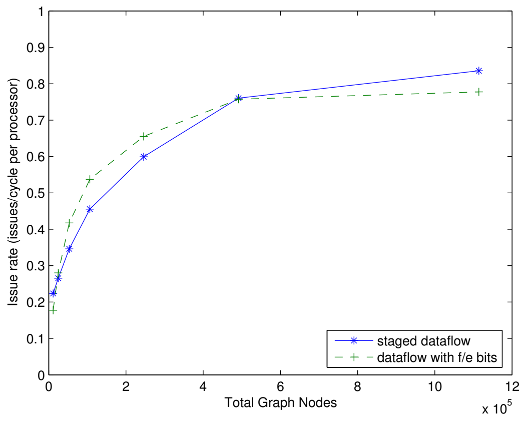
Table 2: Execution Time for FFT Radix-2 Graphs



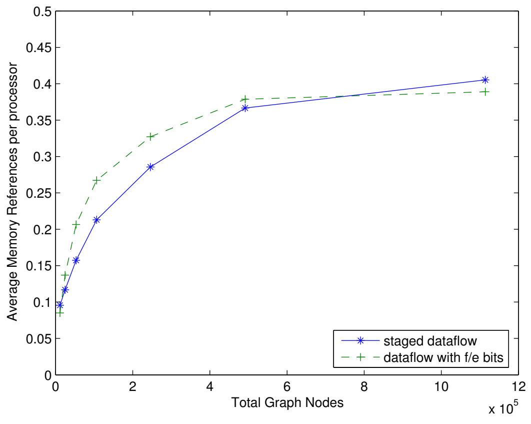
(a) Execution Time



(b) Average Streams



(c) Issue Rate



(d) Average Memory References

Figure 5: Performance characteristics for staged dataflow and dataflow with full/empty bits on radix-2 FFT butterfly graphs

In addition to exploring how multithreaded active graphs problems scale, we investigate the use of full/empty bits in programming such problems. Overall the performance characteristics of the two versions are very similar. For smaller graphs the full/empty bit version provides better performance; as the graphs increase in size the staged dataflow version begins to perform slightly better. This is due to two factors: the overhead involved in going parallel and the amount of parallelism available. For smaller graphs the full/empty bits provide an advantage because they provide lower overhead for going parallel, once for all nodes rather than once for each stage of nodes. The full/empty bits also allow nodes from later stages to begin execution sooner. Thus, when the graphs are smaller and the amount of available parallelism per stage is less, the combination of increased parallelism and lower overhead provided by the full/empty bits results in better performance. However, for larger graphs the number of nodes in each stage, and thus available parallelism, has grown to the point that the staged version almost fully saturates the processor and the full/empty bits no longer provide that edge.

The second set of experiments explored the effects of increasing the number of edges in the graph by generating FFT butterfly graphs of radix 4. This increases the number of edges per node from 2 to 4 but it also has the effect of increasing the number of nodes per stage, and thus the available parallelism, as compared to radix-2 graphs of approximately the same total size. If we consider our dataflow algorithm, doubling the number of edges between nodes doubles both the number of inputs and the number of outputs. For each node this increases the memory operations by a factor of 4 but increases the computation per node by only a factor of 2 thereby increasing the number of streams needed to mask the memory latency. As a consequence, our performance results show that the increase in parallelism seems to approximately balance the increase in memory reference and associated latency leading to performance very similar to that of the radix-2 examples.

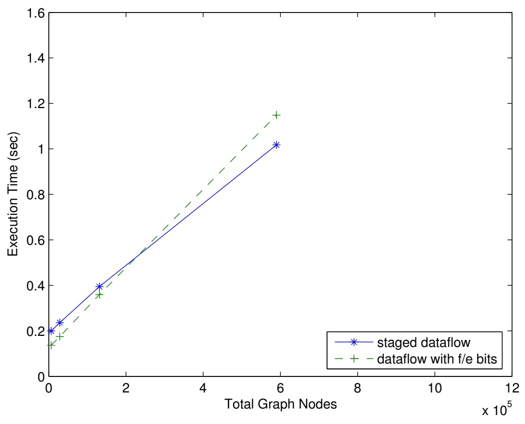
Figure 6 gives the performance characteristics for the radix-4 FFT dataflow experiments. The graphs were generated with inputs of 1024 to 65536 nodes corresponding to total graph sizes of 6144 to 589,824 nodes. Figure 6 (a) presents execution times very close to that of the radix-2 experiments even though the number of edges at each node has doubled. The average streams (b) and issue rate (c) are also very close to those of the radix-2 graphs. However, a significant difference is seen in the number of memory references, presented in subfigures (d). For the radix-4 example, the average memory references per processor per cycle peaks at approximately 0.45 for 600,000 nodes whereas the radix-2 graphs plateau at approximately 0.4. As the number of edges per node increases, the number of memory references increases as well. Therefore, active graphs with larger numbers of edges need an associated increase in arithmetic operations and/or parallelism to offset the increased memory latency.

Comparing the performance of the staged dataflow and full/empty bit dataflow on the radix-4 graphs shows similar trends to those found with the radix-2 graphs. Increasing the size of the graph from 6144 to 28763 total nodes gives a corresponding change in execution time of only 0.199 to 0.236 seconds for the staged dataflow code and 0.137 to 0.175 seconds for the full/empty bit dataflow code. However, increasing the graph size from 131072 to 589824 total nodes provides enough parallelism to more fully saturate the processor and corresponds to an increase in execution time from 0.395 to 1.017 for the staged dataflow and from 0.359 to 1.14 for the full/empty bit dataflow. Again we see that for small graphs the staged dataflow cannot provide enough parallelism and the full/empty bit version provides better performance. Large graphs provide enough parallelism in both the staged and full/empty bit dataflow codes to begin to saturate the processor and, at this point, they begin to provide similar performance.

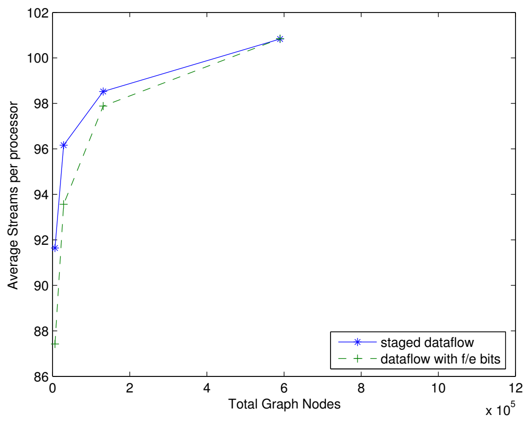
In summary, for active graphs that can be efficiently scheduled in the code to provide sufficient parallelism without using the full/empty bits there is some performance benefit to doing so. However, for many graphs it is either impossible or time prohibitive to determine a scheduling of loops in the code that will provide sufficient parallelism. The code may be highly irregular, difficult to stage, or the available staging may not provide enough parallelism. In these cases the use of full/empty bits allows the programmer to parallelize over all nodes, spending much less time in analyzing and tuning the code for equivalent, if not improved, performance.

5 Conclusions and Future Work

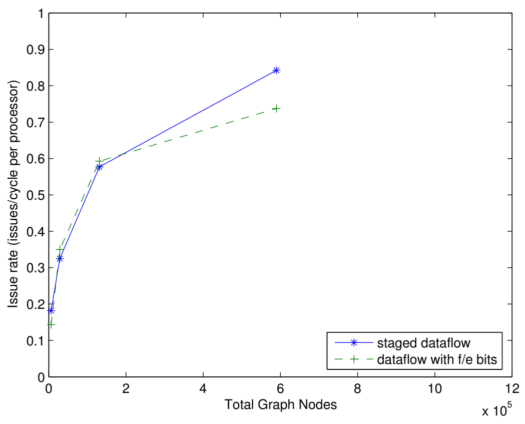
Even as the performance for many computationally intensive applications has improved, there is a large class of applications that continue to challenge conventional parallel architectures. These applications frequently involve irregular graph-like structures, dataflow-type execution models, and memory intensive computations. These applications can be efficiently represented with an active graph model in which components of the system are nodes in a graph and, in implementation, each node is represented by a thread with the flow of data between nodes expressed as producer/consumer exchanges between threads.



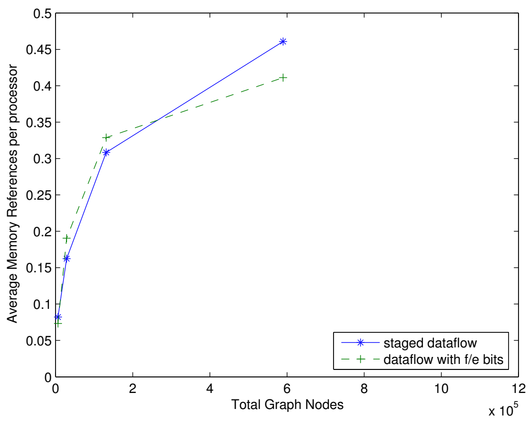
(a) Execution Time



(b) Average Streams



(c) Issue Rate



(d) Average Memory References

Figure 6: Performance characteristics for staged dataflow and dataflow with full/empty bits on radix-4 FFT butterfly graphs

This execution model makes them better suited to and provides better performance on massively multithreaded architectures with low-cost, fine-grain synchronization such as the MTA-2 and, ultimately, Eldorado.

Our experiments demonstrate that active graph applications scale well on the MTA-2. Memory intensive applications such as the FFT dataflow application achieve performance by interleaving the execution of multiple threads to mask memory latencies. For applications in which it is difficult or even impossible to explicitly schedule the code in a way that provides sufficient parallelism, full/empty bit synchronization provides a productive way to increase both available parallelism and performance. The use of full/empty bits allows the programmer to open up the parallelism found in the entire graph and allow the flow of data through the graph to control scheduling. This results in much less time spent in analyzing graph dependencies and tuning the code for parallelism while it provides performance that is similar to that of explicitly scheduling the code.

In future work we plan to explore how a broader range of applications can be represented as active graphs as well as how these different types of graphs scale on the MTA-2 and Eldorado. Neural networks, production rule systems, and decision trees are just a few of the applications with which we are currently experimenting. We also plan to explore the use of futures for more direct, task-level implementation of threads in the graph model.

References

[1] Susan L. Graham, Marc Snir, and Cynthia A. Patterson, editors. *Getting Up to Speed: The Future of Supercomputing.* The National Academies Press, Washington, D.C., Feb. 2005.

[2] Mark James. Spacecraft health inference engine: An ultra high-speed inference engine for the monitoring and diagnosis of real-time systems. April 2000.