Evaluation of Active Graph Applications on the MTA-2

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

Mark James, Ed Upchurch
NASA JPL

John Feo
Cray Inc.

UNIVERSITY OF NOTRE DAME
JPL Jet Propulsion Laboratory California Institute of Technology
CRAY

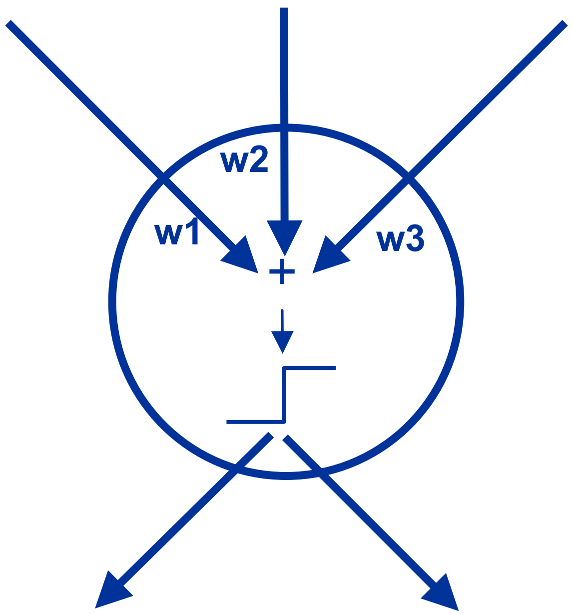
Summary

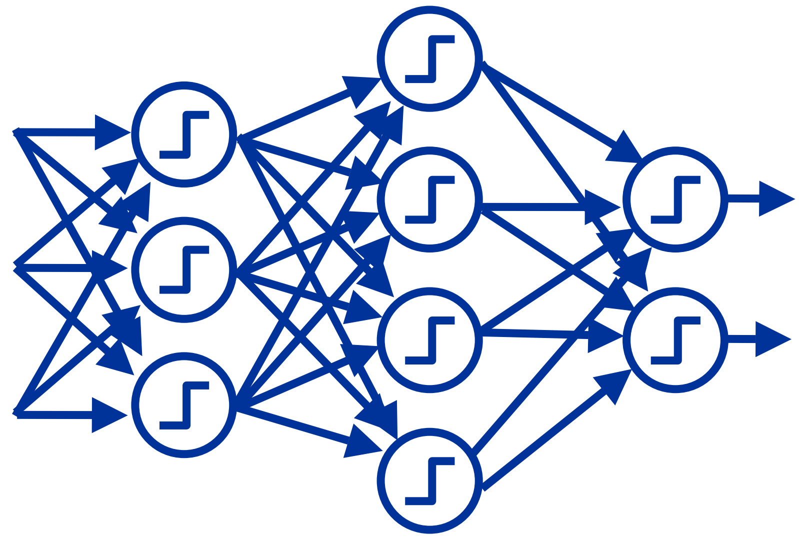
- Goals
  - Examine massively-parallel lightweight multithreading for modeling systems of intercommunicating agents
  - Look at tradeoffs between programming productivity and execution efficiency for active graphs on MTA-2
- Outline
  - Definitions and example applications
  - Representation of active graphs
  - Experiments and results
  - Conclusions and future work

Active Graphs

- Definition
  - Each node (possibly edge) tied to a distinct thread
  - Flow of data through producer/consumer relationships between threads
- Thread state
  - input mailboxes
  - working registers
  - pointers to upstream/downstream nodes
- Key issues
  - exploiting parallelism
  - managing synchronization

Neural Net



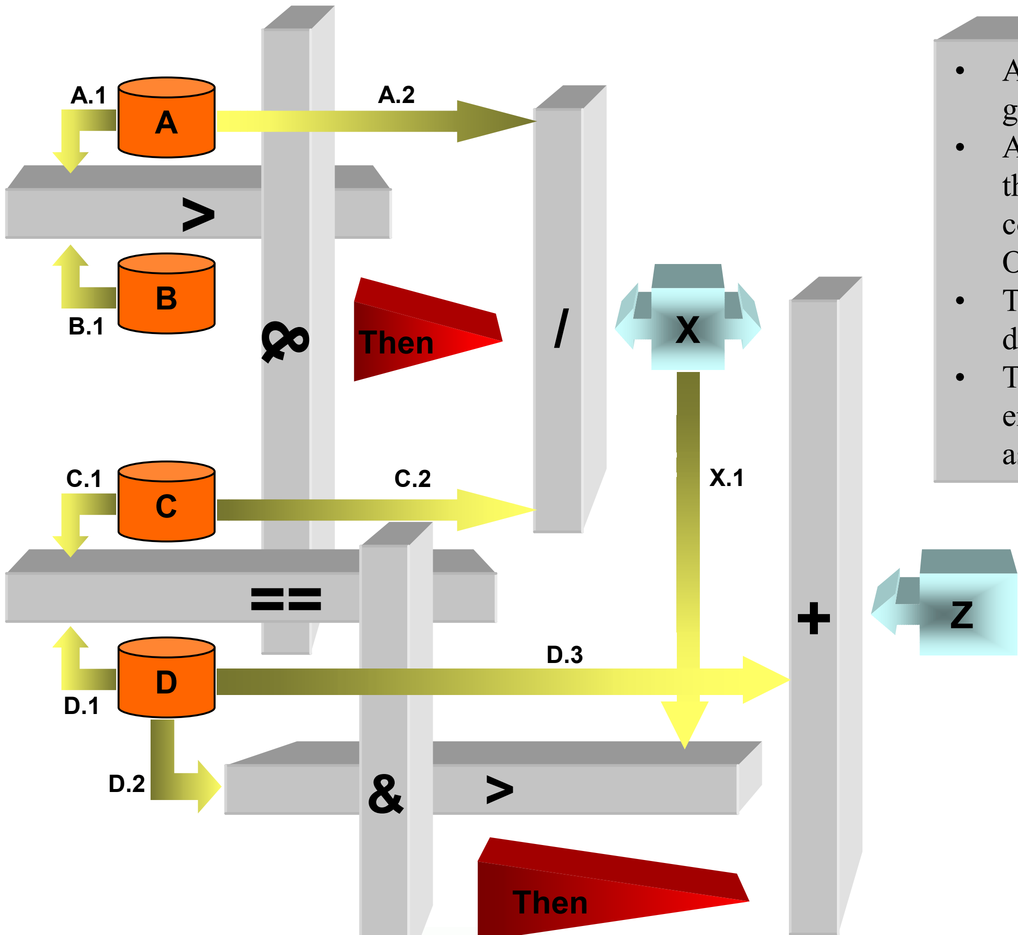


Neural Net

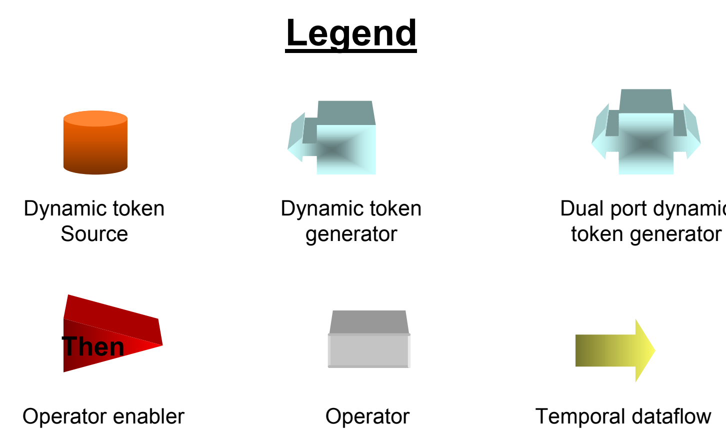
Shine: Spacecraft Health Inference Engine

Rule 1:
If Sensor A > Sensor B & Sensor C == Sensor D
then Var X = Sensor A / Sensor C

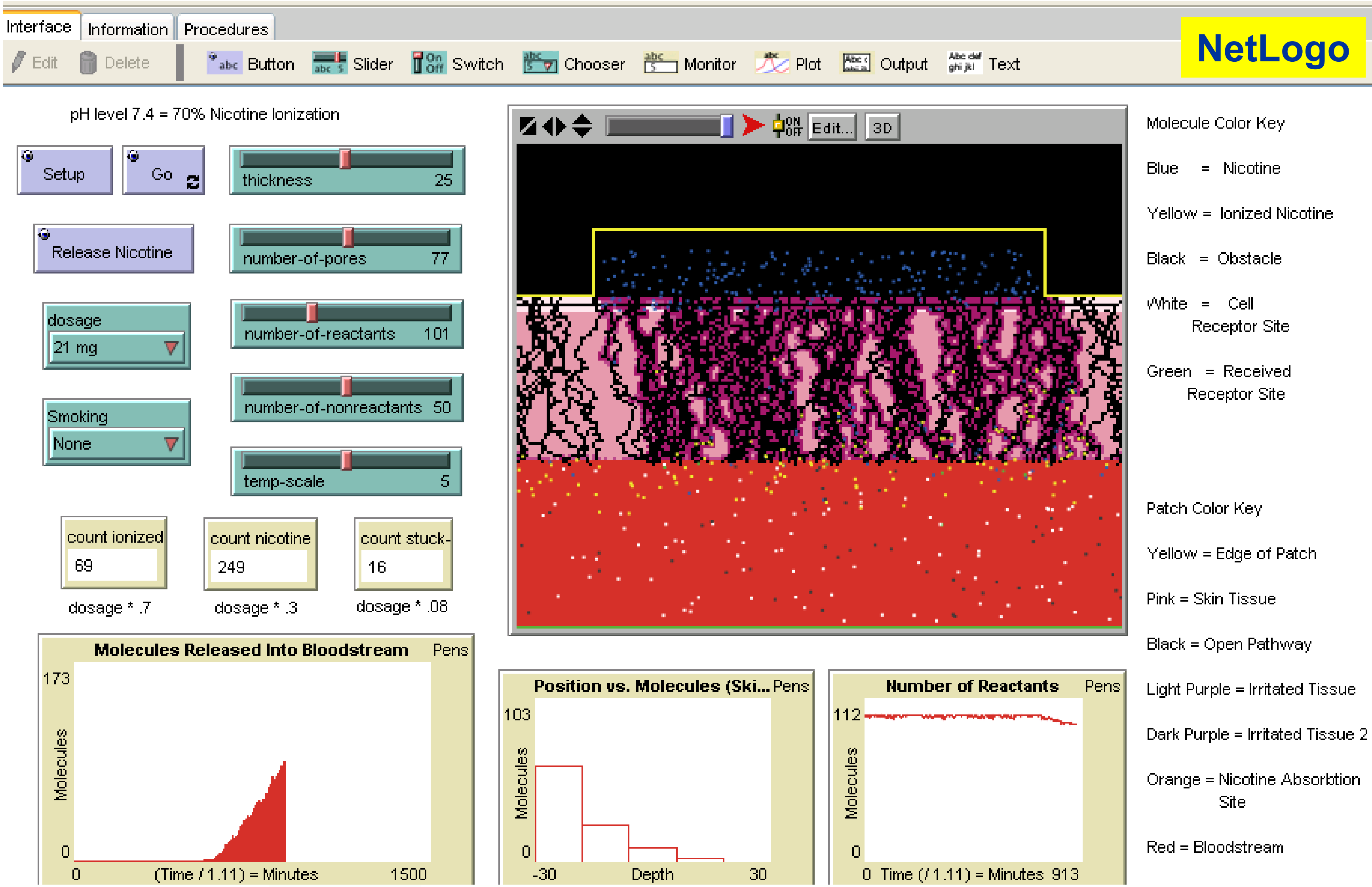
Rule 2:
If X > Sensor D & Sensor C == Sensor D
then Var Z = Var X + Sensor D



- All rules are analyzed respect to one another for all possible global interactions with maximal sharing.
- A sophisticated mathematical transformation, based on graph-theoretic data flow-analysis is introduced, that reduces the complexity of conflict-resolution during the match cycle from $O(n^2)$ to $O(n)$.
- The underlying structure is mapped into temporally invariant dataflow elements.
- The final representation is either executed in a debugging environment or translated to a variety of target languages such as ADA, C and C++.



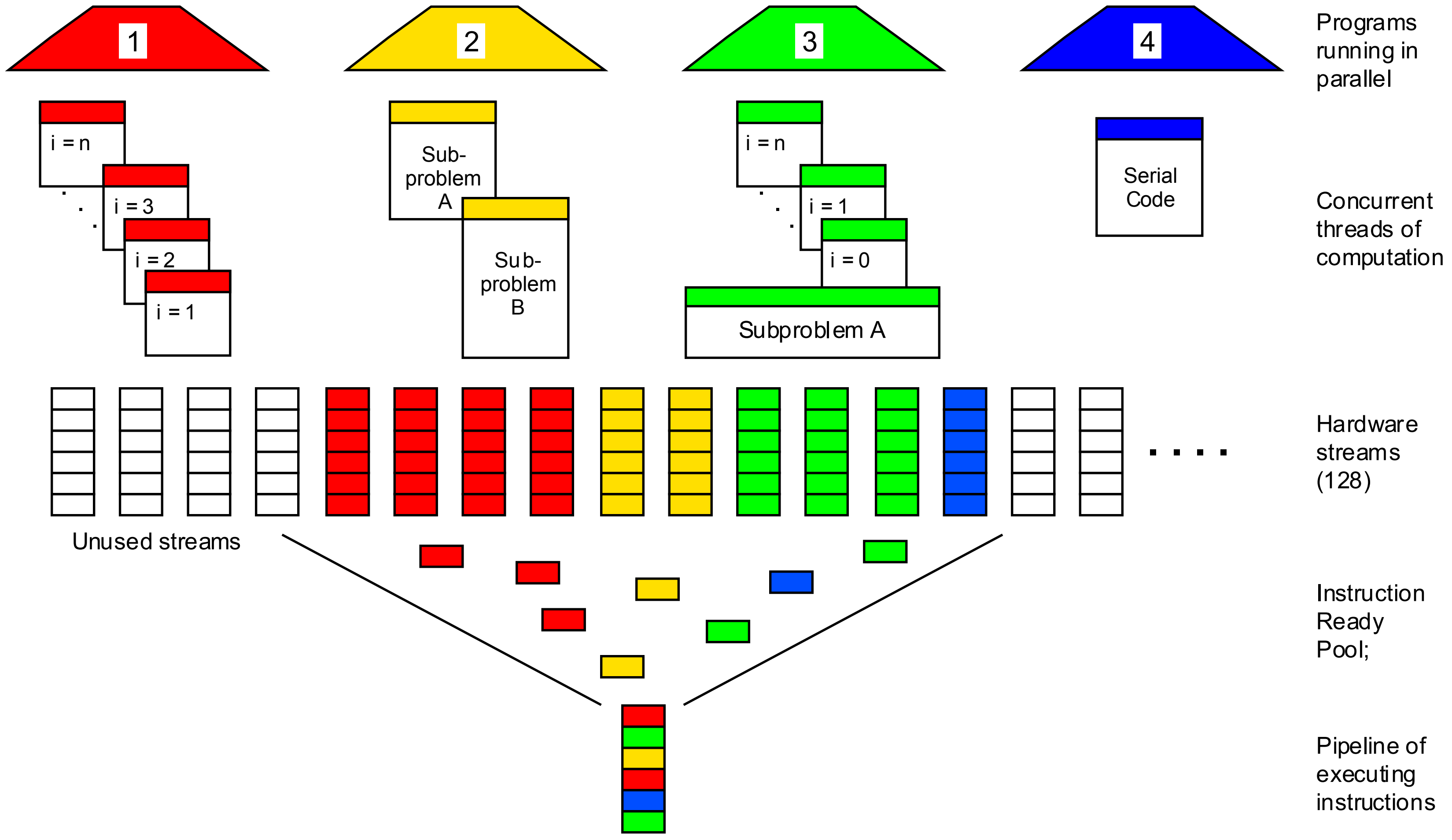
Agent-Based Simulation



MTA Threads

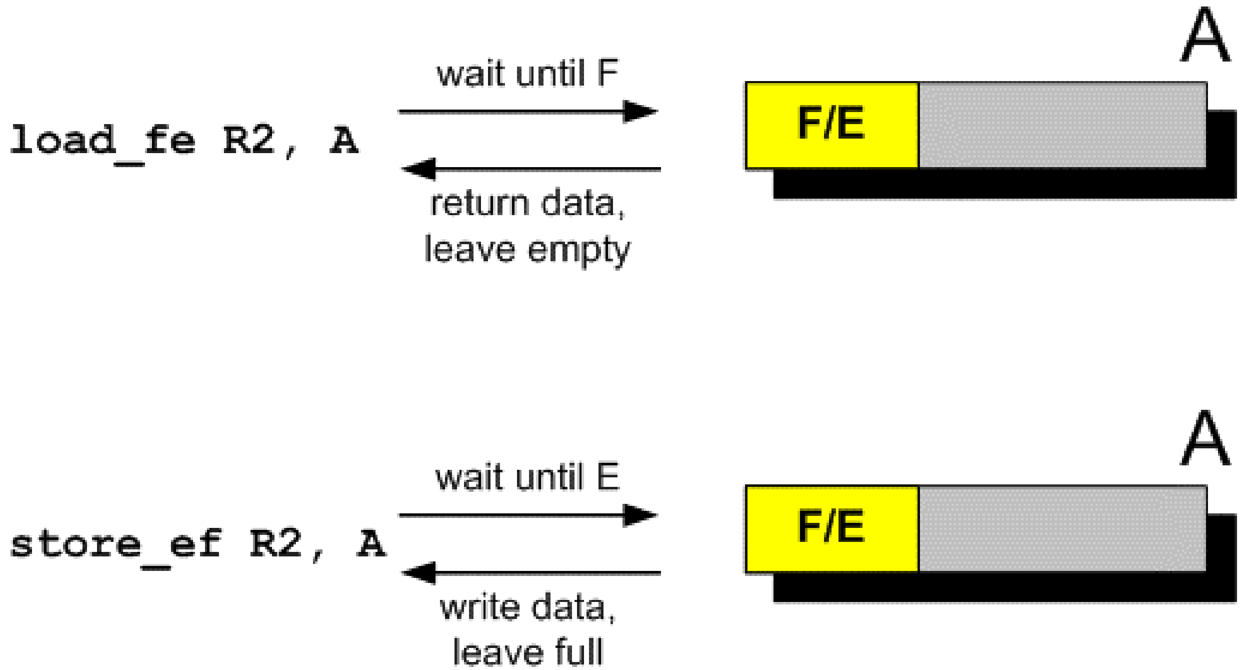
- Very lightweight; managed by compiler and runtime
  - Thread creation, termination, synchronization, and scheduling are performed by user space library code
- Runtime requests OS to add processors
- Thread virtualization
  - Application is coded without knowledge of stream count
  - Runtime migrates thread state between memory and stream
- Implicit creation by complier and user directives
- Explicit creation using future statement

Threads and streams



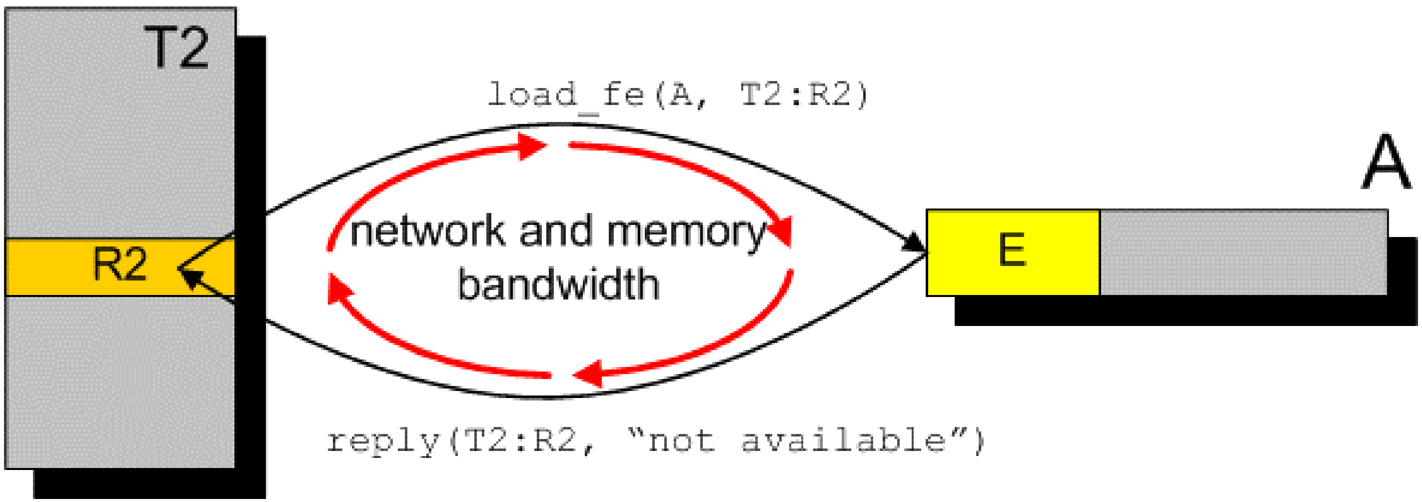
Full/Empty Semantics

- F/E status of a location influences load/store behavior and loads/stores can modify F/E status.
  - Loads/stores can synchronize on F/E.
  - Cheap and abundant synchronization operations.
  - Prior MTA experience implementing f/e semantics.



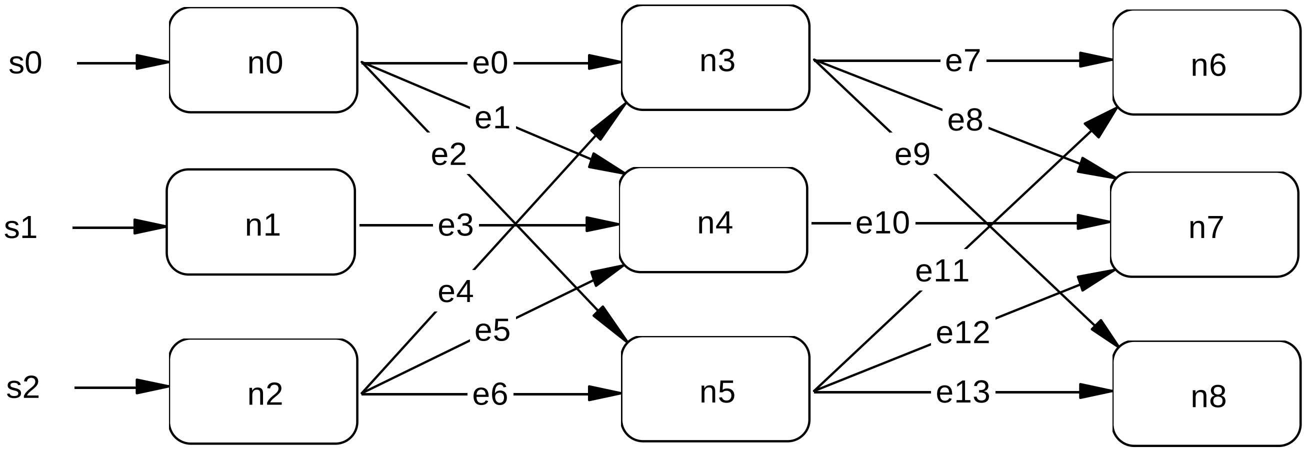
MTA F/E Implementation

- What happens if A is not in the required state when the `load_fe` or `store_ef` arrives?
  - Cray MTA f/e behavior for `load_fe` with A empty.



  - Eventually a timeout occurs, the thread traps to a software handler; the thread state is saved to memory and the thread sleeps until A is marked full.
  - When A becomes full, thread state is reloaded from memory and the `load_fe` is restarted.

Active Graph Representation




```
typedef struct {
  int value;
  int n_edges_in;
  union {
    int *edges_in;
    int streamid;
  } u;
  int n_edges_out;
  int *edges_out;
} Node;

typedef struct {
  int from;
  int to;
  int weight;
  int value;
} Edge;
```

- Dataflow execution of input correlated streams
  - weighted sum at each node
- Nodes, edges, streams are arrays
  - MTA randomly scatters data in physical memory

Scheduling Alternatives

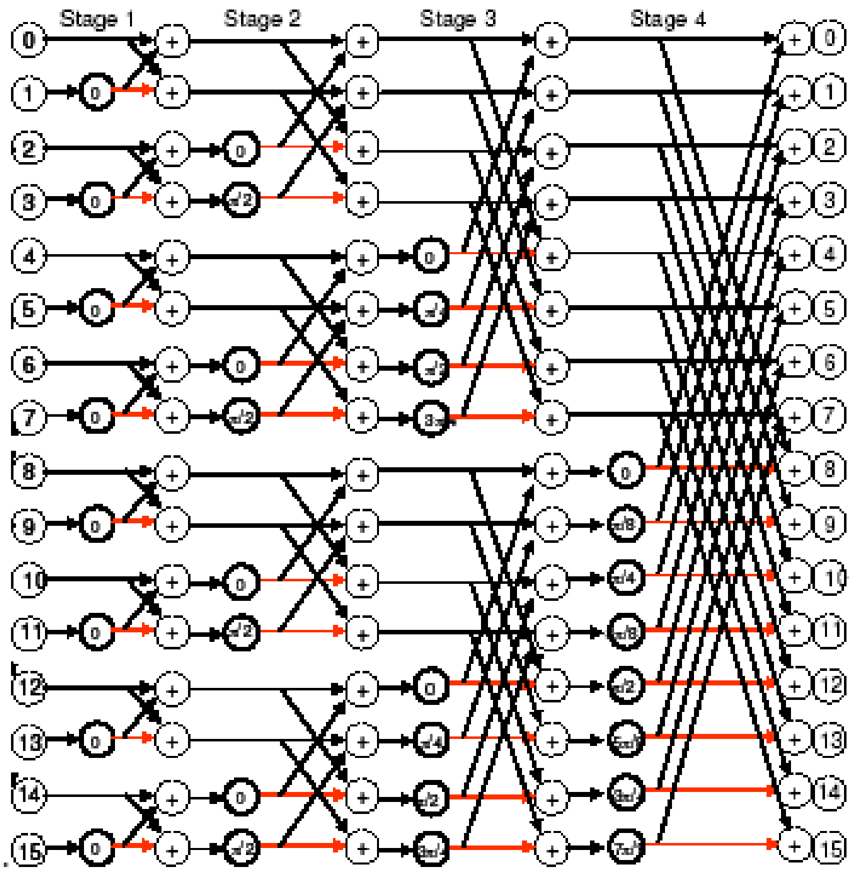
```
for each data set
   for each stage of the graph
      #pragma mta assert parallel
      for each node in that stage
         read()
         compute()
         write()
```

- Synchronous stages
  - stages can be hard to identify in random graphs

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

- Asynchronous using full/empty bits
  - no need to identify stages

Experiment: Parameterized Butterfly Graphs

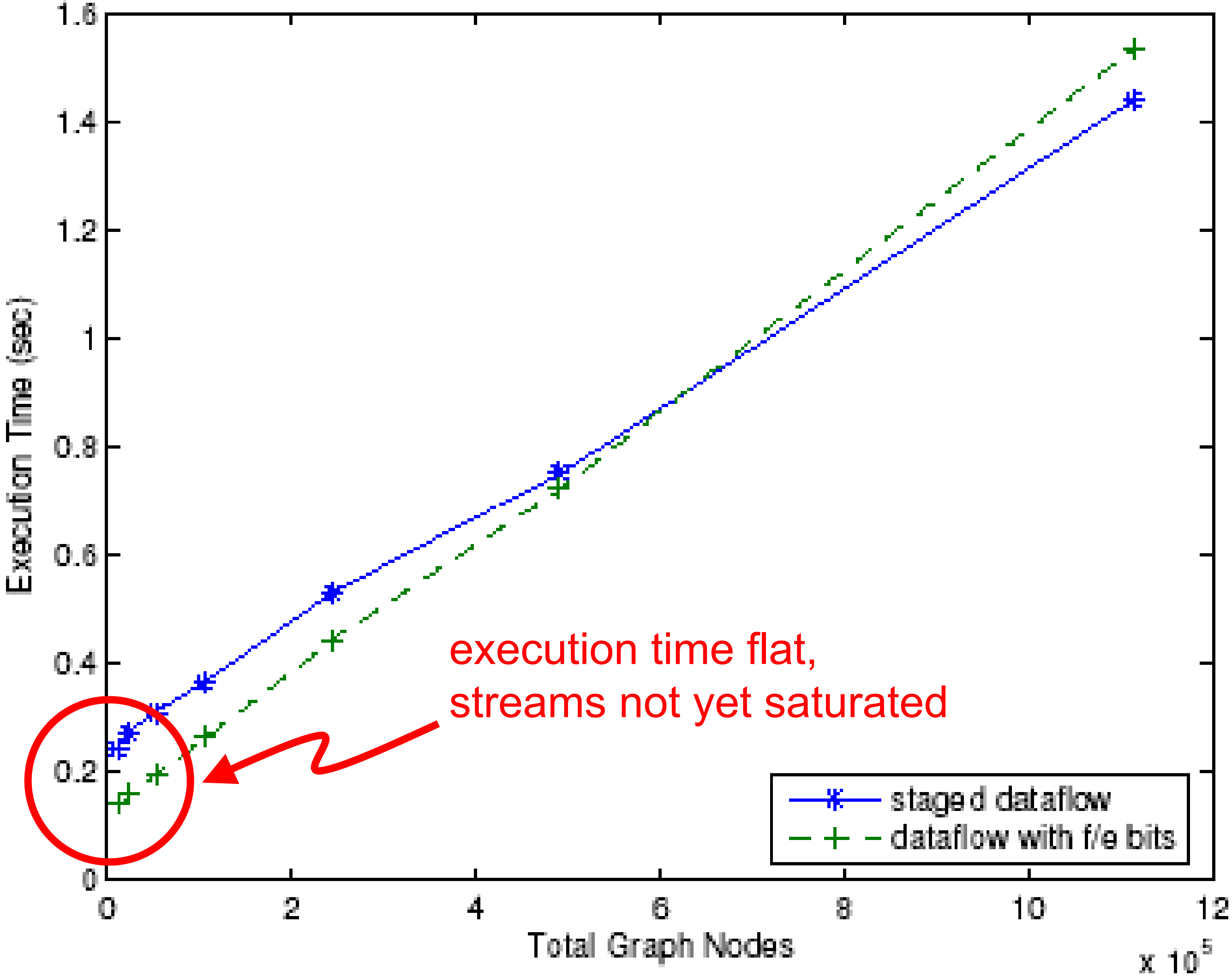


- Varied
  - Number of nodes
  - Radix
  - Randomly deleted edges
- Run on 10 MTA-2 processor nodes

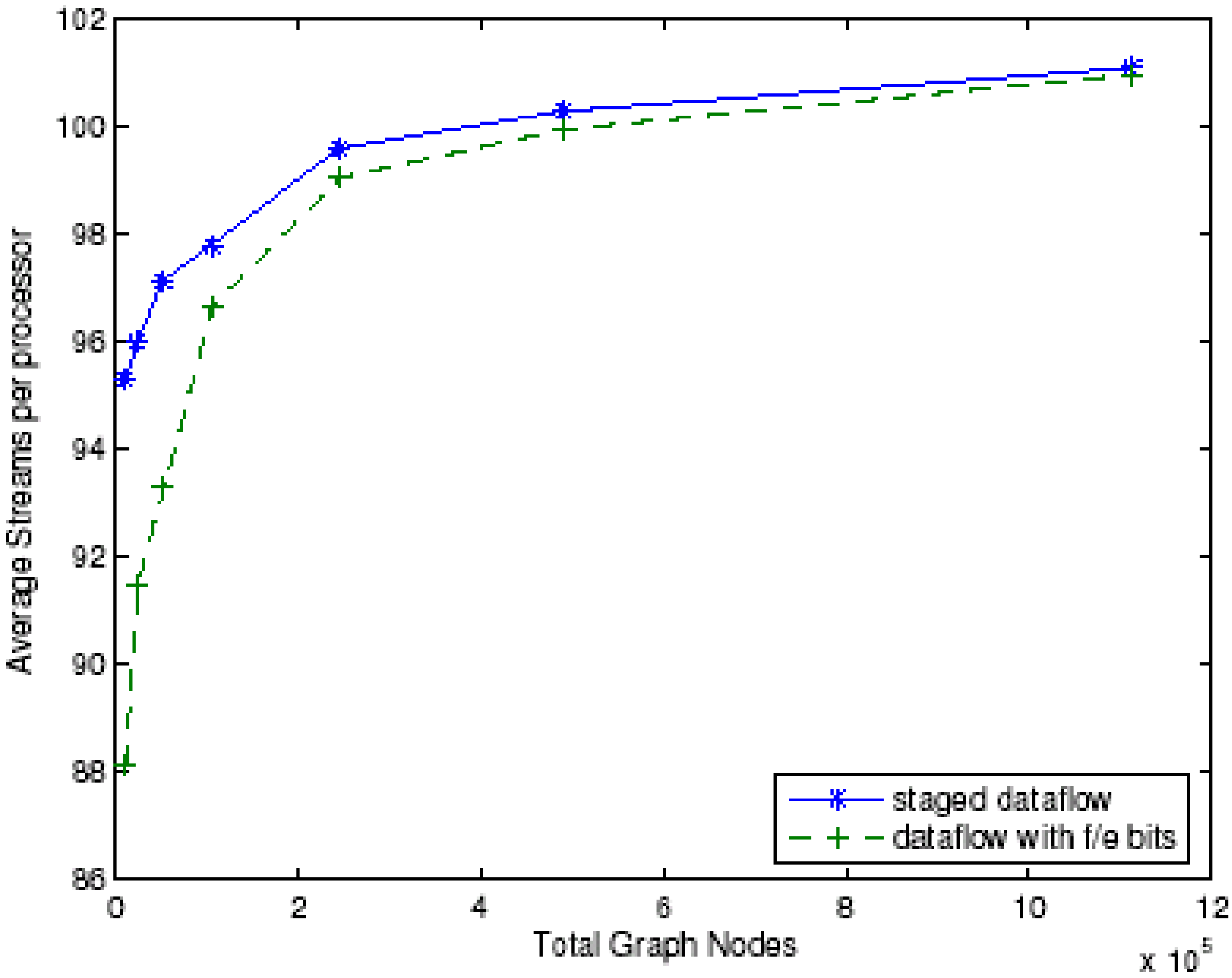
UNIVERSITY OF NOTRE DAME

Jet Propulsion Laboratory
California Institute of Technology

Execution Time

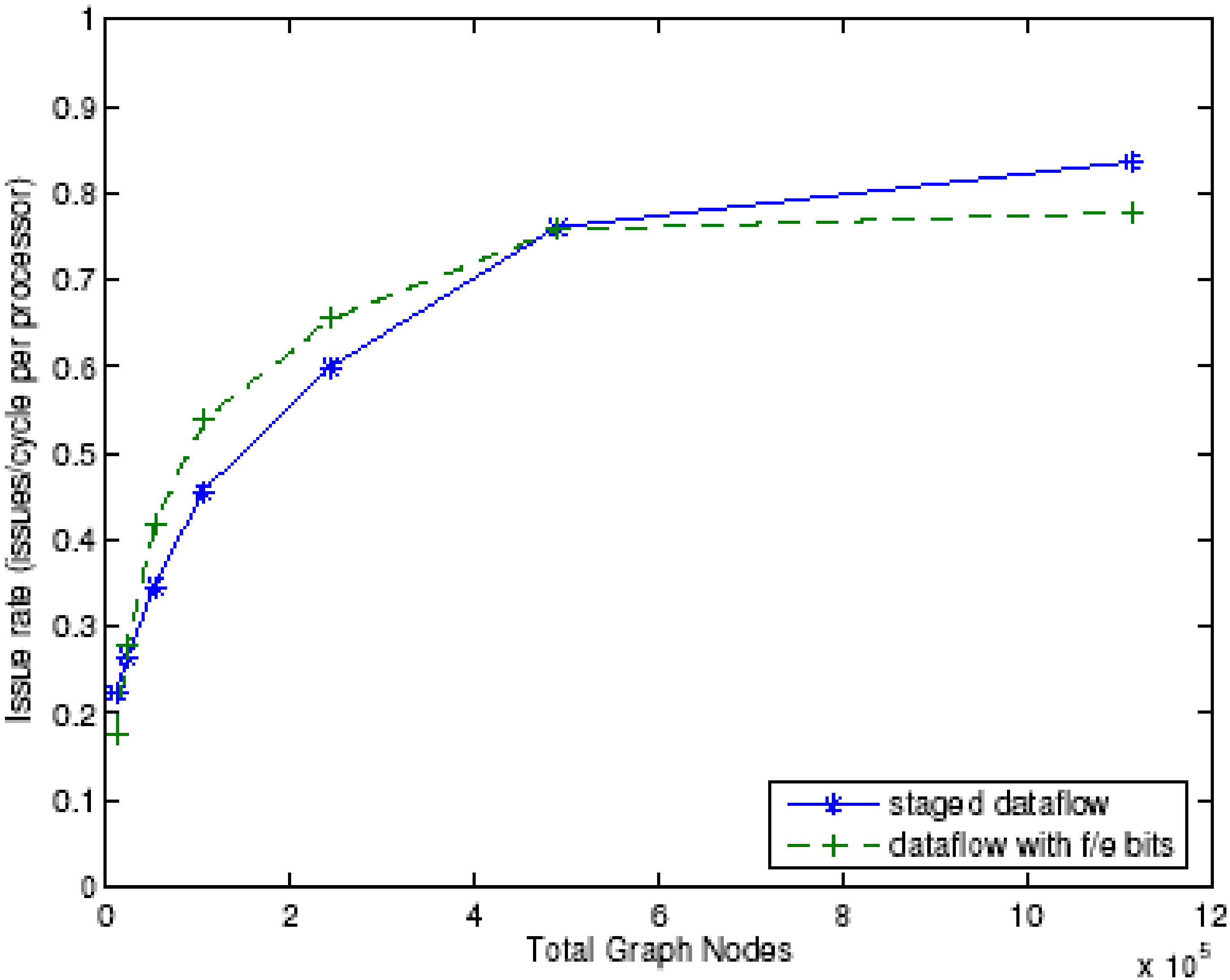


Streams



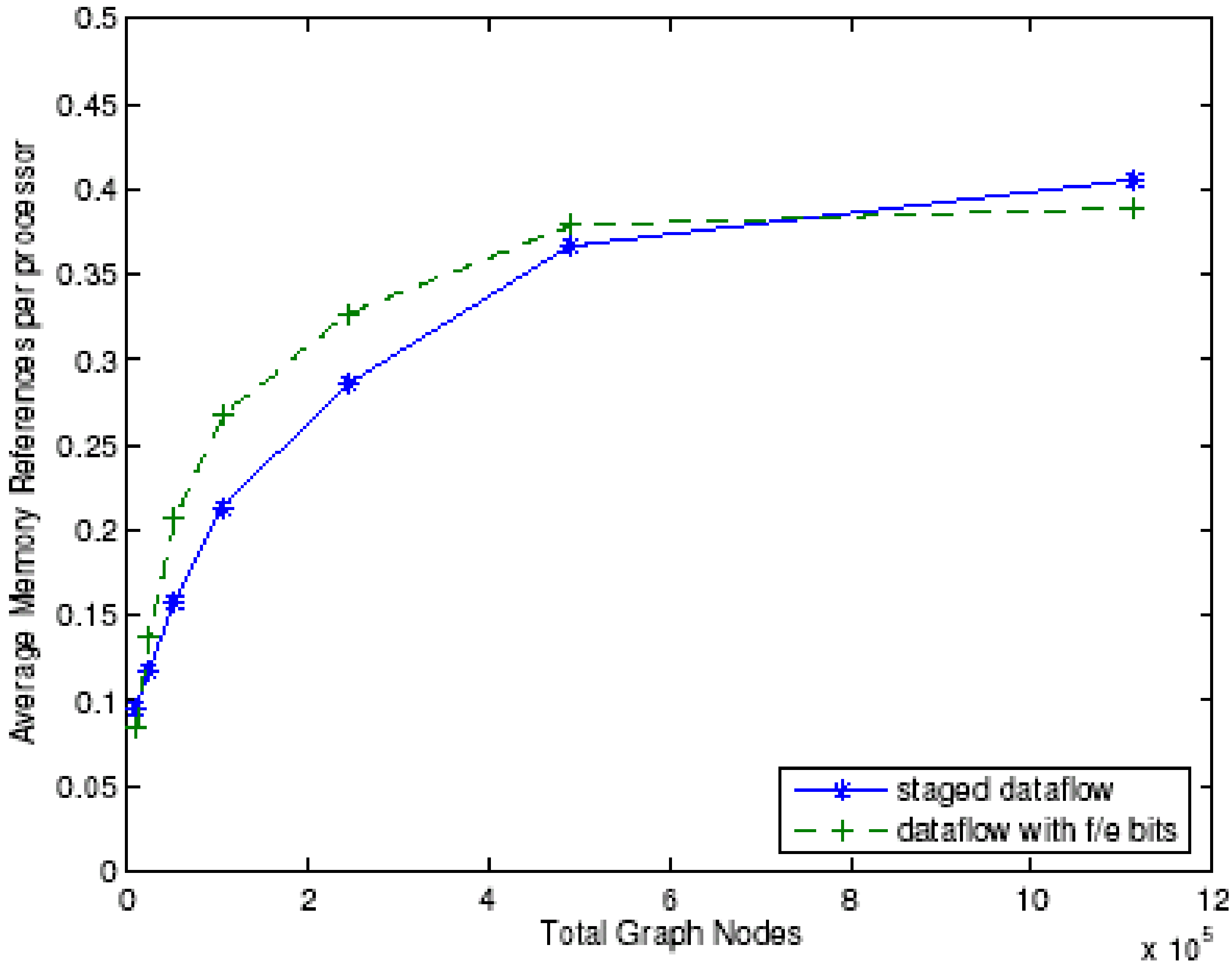
102
100
98
96
94
92
90
88
86
0
2
4
6
8
10
12
Average Streams per processor
Total Graph Nodes
x 10^5
staged dataflow
dataflow with f/e bits

Issue Rate



1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Issue rate (issues/cycle per processor)
0
2
4
6
8
10
12
Total Graph Nodes
x 10^5
staged dataflow
dataflow with f/e bits

Memory References



Trying for More Parallelism

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

- Deadlock after 512 nodes!!!
- Why?
  - Outstripped number of available streams
  - Runtime scheduling becomes an issue

Conclusions and Future Work

- Active graphs are a natural way to formalize systems of interacting agents
  - Easy implementation on MTA
- Very low overhead synchronization using full/empty bits
  - Avoids need for explicit scheduling—to a point
- What's next?
  - More applications, including ZCHAFF SAT solver
  - Investigate PGAS programming models and performance on Eldorado—path to Cascade