Characterizing Applications on the MTA2 Multithreading Architecture

Cray User Group 2006, Lugano, Switzerland
May 10, 2006

Richard Barrett, Jeffrey Vetter, Sadaf Alam, Collin McCurdy, and Philip Roth

Oak Ridge National Laboratory
Oak Ridge, TN 37831

http://www.csm.ornl.gov/ft
http://www.nccs.gov

OAK RIDGE
National Laboratory

MTA Motivation

- Memory access latency

- Common approach: cache

  Con:

  - Leads to code transformations to increase likelihood of accessing data in cache
  - Not all code can be made “cache friendly”
  - Transformations may limit performance on other architectures (e.g., vector processors)

MTA Philosophy

- *Tolerate* memory access latency
- Instead of data caches to reduce latency of *some* accesses, use computation to hide "communication" (data transfer between memory and processor registers) for *all* accesses

- Problem: available overlap within one thread of execution is often too small to hide the entire memory access latency
- MTA solution: support enough concurrent threads of execution to hide the worst case memory access latency
  - When one thread issues a load instruction, execute instructions from other threads until load completes
  - Low-overhead switching between threads

MTA-2 Processor

- Compute nodes based around MTA processor
  - Support for 128 concurrent instruction streams
  - Switch between streams on each cycle
  - 64-bit VLIW instruction
    - One fused multiply-add
    - One add or control
    - One memory load or store
  - 220 MHz

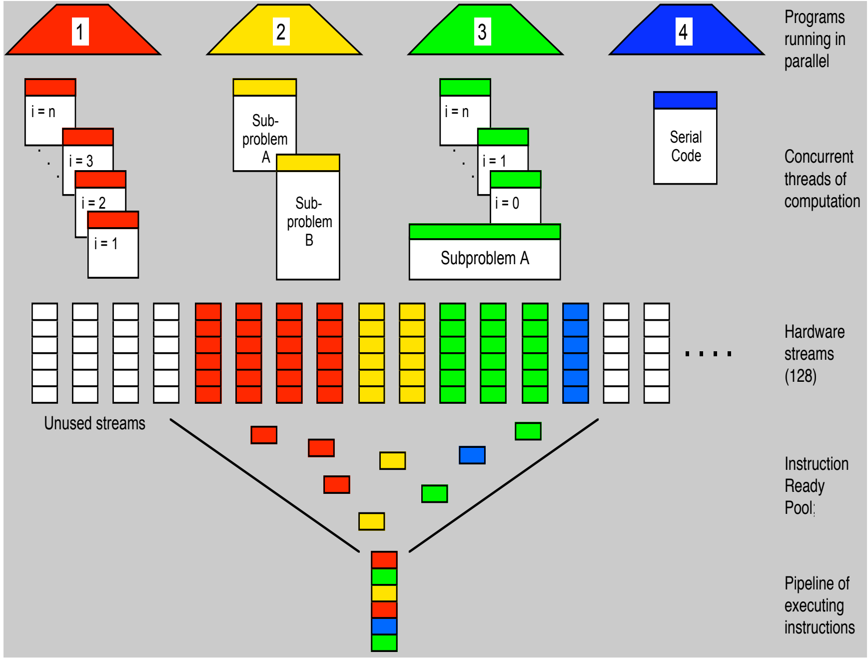


Image courtesy of Cray Inc.

MTA-2 System Organization

- Compute nodes connected with interconnect network
  - "Modified Cayley" topology
  - Also described as 3D torus with some links removed
- Memory units distinct from compute nodes
  - "Dance hall" organization
  - Every memory access goes across the interconnect
  - Memory locations have associated "full/empty" bit
- SPARC Solaris front-end system

Programming the MTA-2

- Global shared memory model
  - Programs are collections of threads that access shared data
  - Synchronize using full-empty bits on memory locations

- Implicit and explicit expressions of parallelism
  - Loops (implicit)
    - Compiler automatically splits loop iterations across multiple threads
    - May require directives to specify absence of dependencies or best number of threads to use
  - Futures (explicit)
    - Somewhat like a function call, with code body and return value
    - Executed in a separate thread, can synchronize on return value
    - For task parallelism and recursion
    - Can use generic functions like readfe() for explicit synchronization between threads

MTA-2 Tools

- Traditional toolchain on front-end node
  - Compiler, assembler, linker
  - C, C++, Fortran (F77 and F90)
  - Cross-compilation, since front-end is SPARC Solaris

- Traceview provides insight into program's dynamic behavior
  - Graphical user interface showing program timeline with observed and theoretical maximum parallelism
  - Can provide detailed information (e.g., source code) for points along the timeline
- Canal (Compiler Analysis) provides insight into compiler transformations

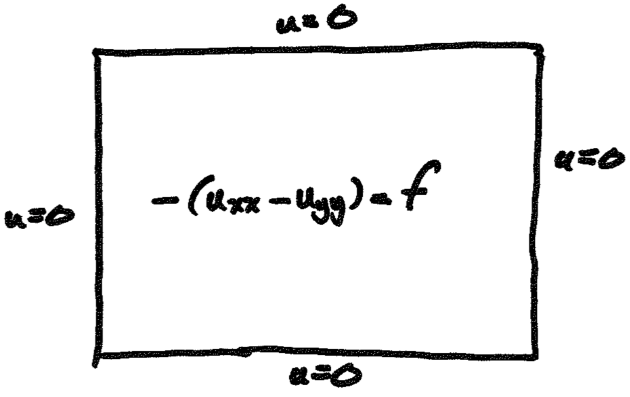
  - Exposes whether compiler has parallelized a loop and how many threads it will request to execute it
  - Also explains why compiler didn't parallelize a loop

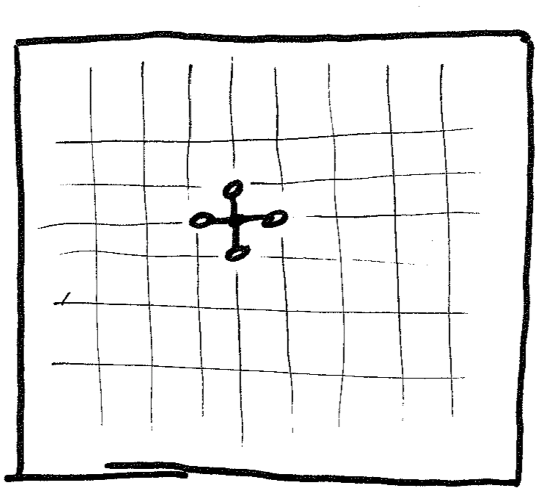
Programming MTA-2 for Performance

- Key to good performance is keeping processors saturated (I.e., each processor always has a thread whose next instruction can be executed)

- Potential usage scenario
  1. Compile
  2. Use canal tool to check that important loops were parallelized
     - If loops weren't parallelized, add directives or modify code to enable compiler to parallelize loops
     - Back to step 1.
  3. Run instrumented code to produce program trace
  4. Use traceview to identify situations where processors are under-utilized
     - If there are any ☺, add directives or modify code to expose more parallelism
     - Back to step 1

Continuous PDE to discrete form for Finite Difference Stencils



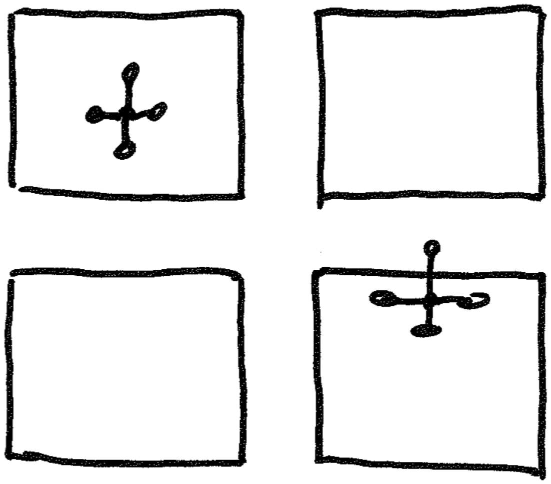


```
DO J = 2, LCOLS+1
    DO I = 2, LROWS+1

         GRID2(I,J) = (
                                             GRID(I-1,J) +
                          GRID(I,J-1) + GRID(I,J) + GRID(I,J+1 ) +
                                             GRID(I+1,J)  ) / 5

    END DO
END DO
```

Parallel Processing

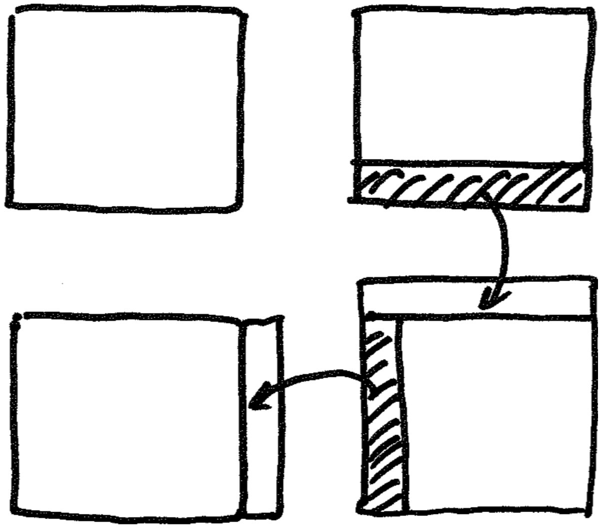


Parallel Processing

MPI Code:

! Exchange $\partial\Omega$

! Compute



MTA-2 implementation

```
DO J = 2, LCOLS+1
   DO I = 2, LROWS+1

      GRID2(I,J) = (
                                 GRID(I-1,J) +
                GRID(I,J-1) + GRID(I,J) + GRID(I,J+1 ) +
                                 GRID(I+1,J)  ) / 5

   END DO
END DO
```

What is peak?

- Performance expectation:

  F(mach capability for our problem)

- Flops/MemRef * 220[MHz]
- Tools:
  - Traceview: shows where to look.
  - Canal (Compiler ANALysis) tool. Shows effects of work.
- *Feo's Rule*: Expect ~90+% of peak.

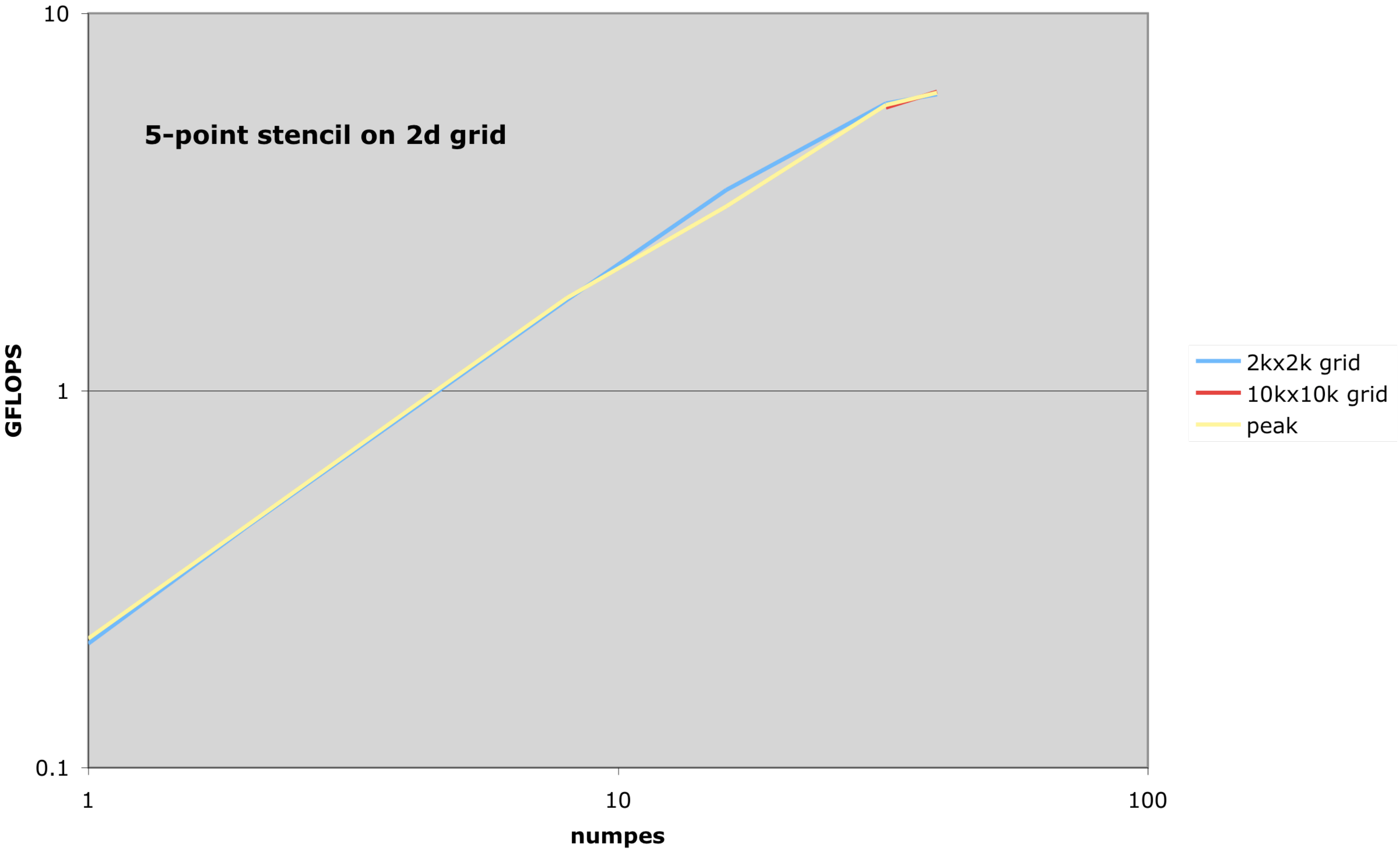
Expectation: CAnal

```
        |     DO I = 2, LROWS+1
        |       DO J = 2, LCOLS+1
26 SSPP |         GRID2(I,J) =                            &
        |              (          GRID1(I-1,J)+           &
        |               GRID1(I,  J-1)+GRID1(I  ,J)+GRID1(I,  J+1) +  &
        |                         GRID1(I+1,J)          )  &
        |              * FIFTH
        |       END DO
        |     END DO
```

```
Loop  26 in MAIN__ at line 197 in loop 25
    Parallel section of loop from level 4
    Loop summary: 6 memory operations, 5 floating point operations
         8 instructions, needs 30 streams for full utilization
         pipelined
```

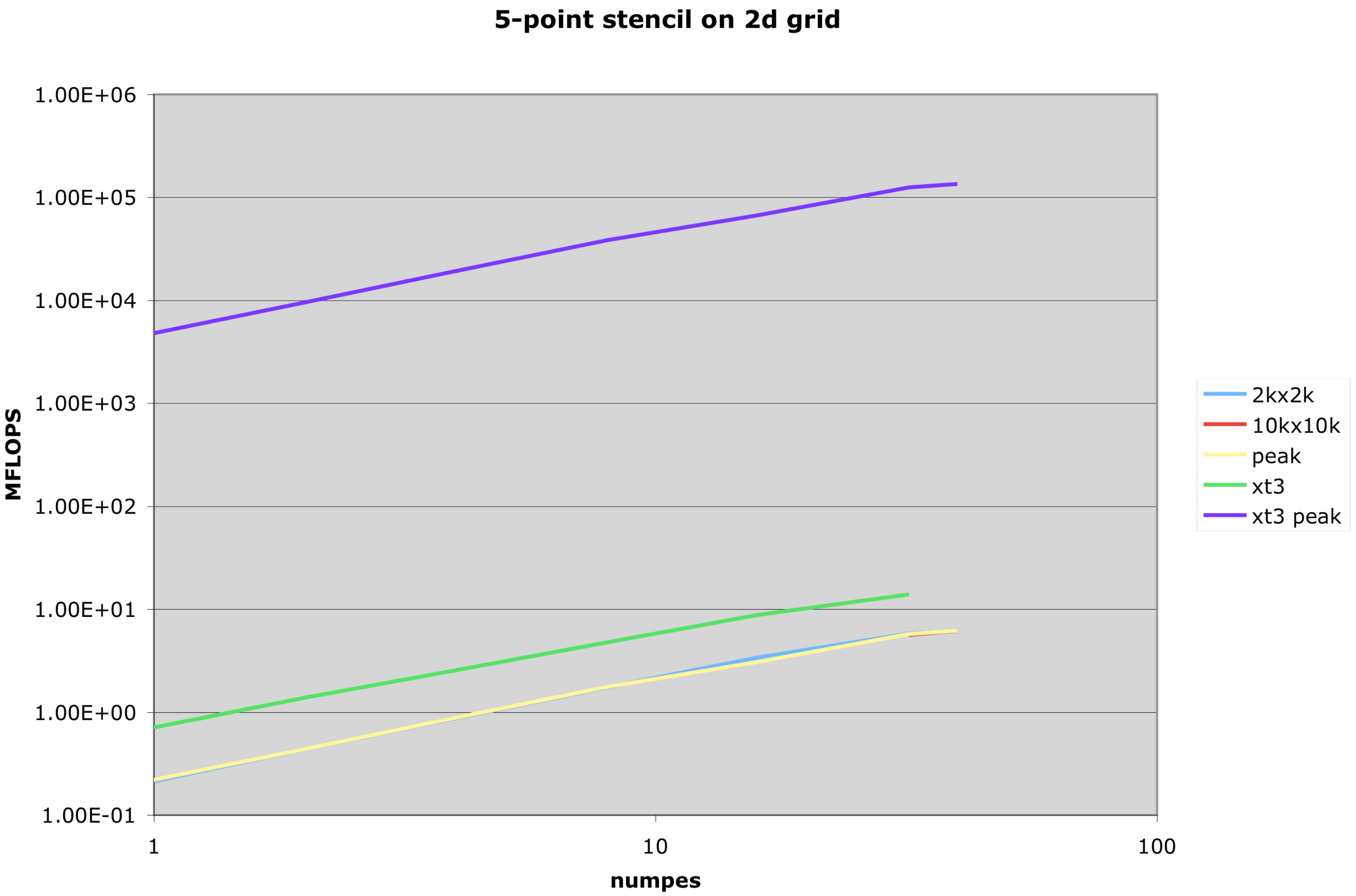
```
!$mta use 60 streams
```

Performance:
5-pt difference stencil



Serial code!

Comparison with XT-3

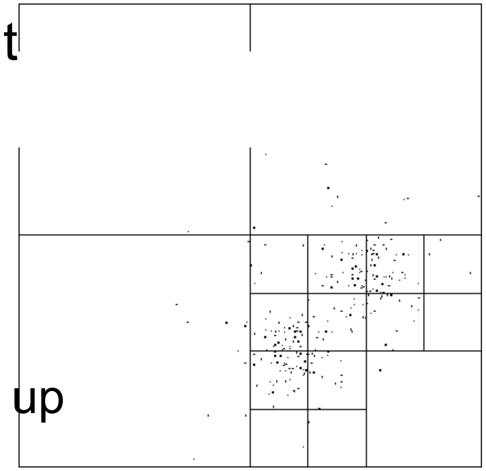


Applications

- Fast Multi-pole
- Molecular dynamics
- Discrete even simulation

Fast Multipole Method

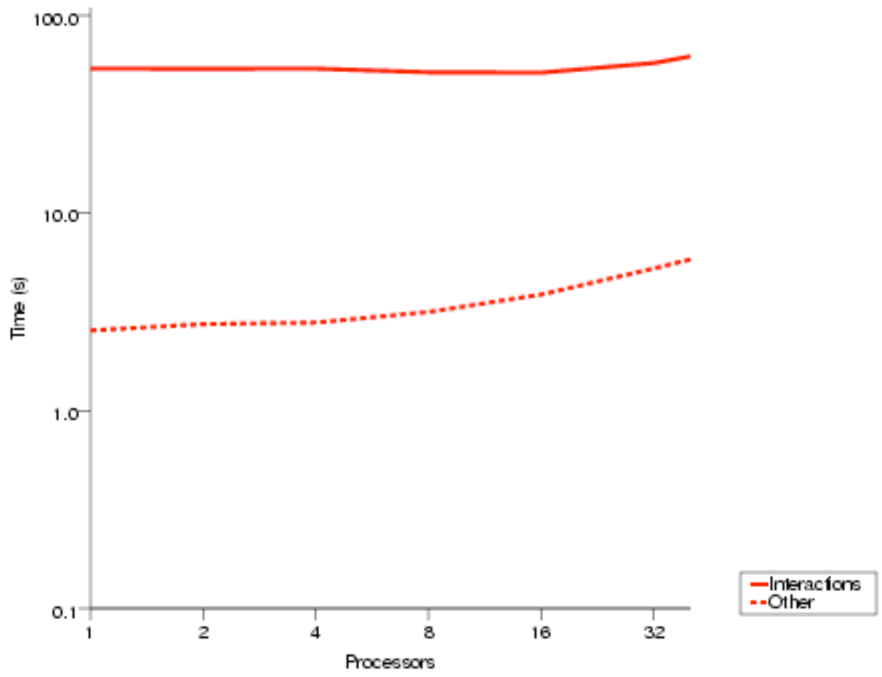
- Adaptive tree-code: solves $O(n^2)$ N-body problem in ~O(n) time
- Attractive candidate for MTA:
  - Irregular references to global data structure
    - Tree has a single root…
  - Adaptive nature makes load-balancing difficult
- Algorithm:
  - Insert particles into adaptive tree
  - Tree traversals:
    - Create interaction lists
    - Upward pass, propagate summary information up
    - Interactions
    - Downward pass, propagate potentials down to particles



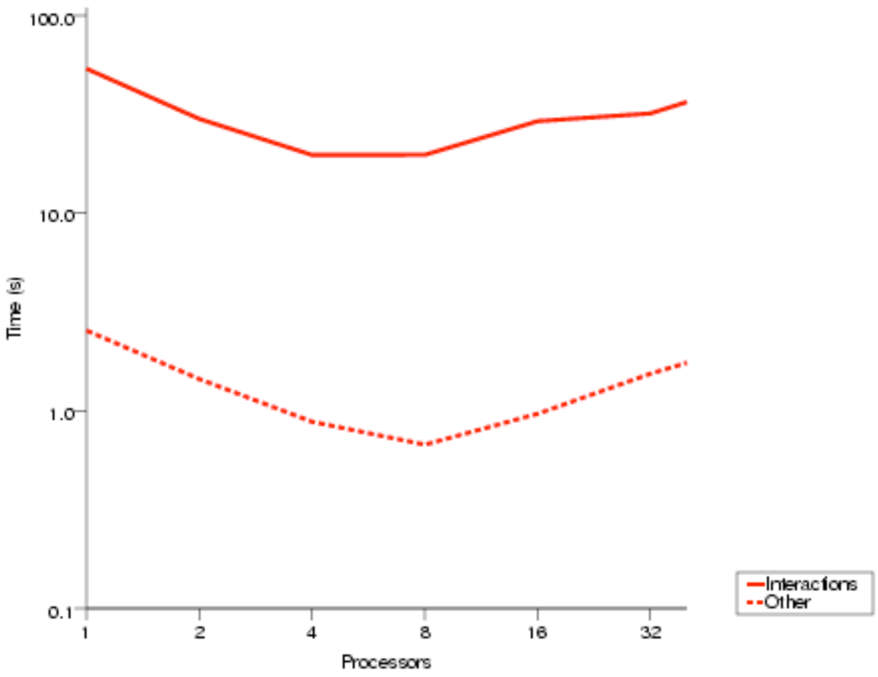
MTA Implementation

- Tree Traversals
  - Significant parallelism obtained simply by parallelizing tree traversals
  - Initial cut: use **future** construct for recursive traversals
    - Proved unnecessarily expensive
  - More efficient solution: forall loop over nodes w/ additional synchronization when required
- Tree Construction
  - Parallelize loop that inserts particles in tree
    - Substantial sync required to ensure nodes uniquely created
    - Final implementation likely only possible on MTA:
      - Use synchronizing reads rather than locks to get to leaf, *then* lock leaf; retry if leaf modified before locked

Initial Results



64k bodies / proc



64k bodies total

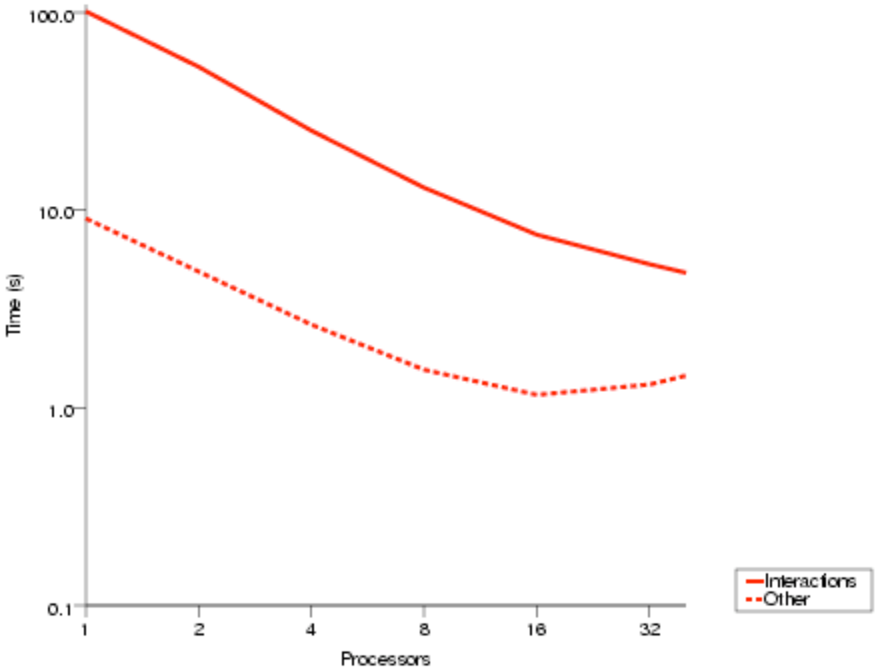
Decent weak scaling, but strong scaling needs work…

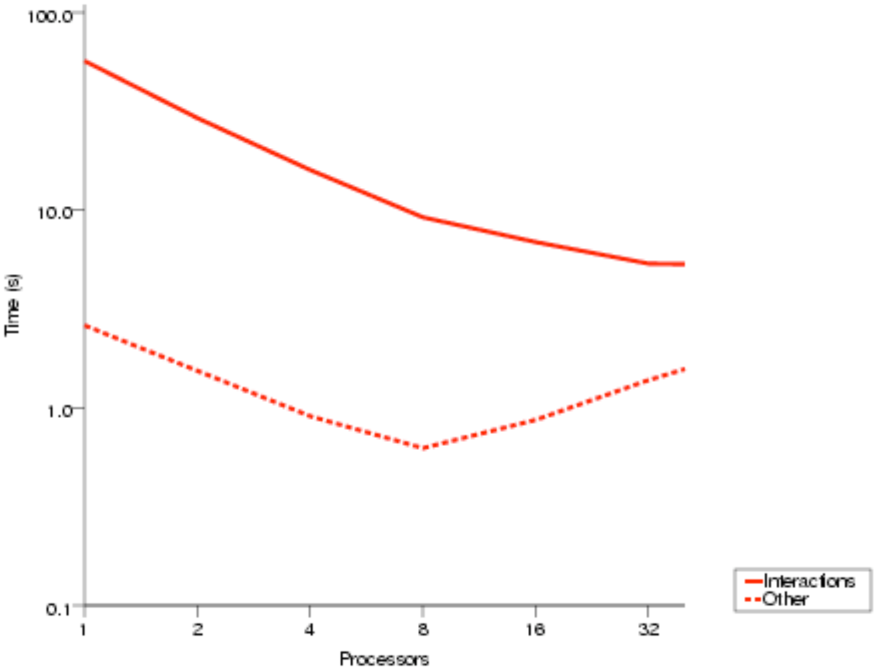
Improving Strong Scaling

- Two related problems:
  1. Not enough work – proportional to # nodes…
  2. Variance in amount of work per node

- Two potential solutions:
  - ↔↘ Reduce “Maximum Bodies Per Node”
    - Runtime parameter, determines depth of tree
    - Fewer bodies/node implies deeper tree, more nodes, more work, less variance in amount of work
  - ↕↘ “Crack open” Interaction computation
    - Allow multiple threads to compute one node’s interactions
    - Implies significantly more synchronization: lock for every update of field being computed

Improved Strong Scaling



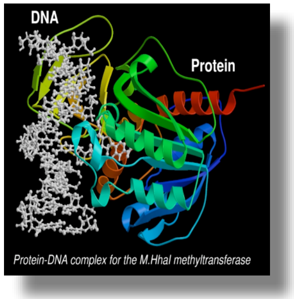


Reduced bodies/node:
- from 128 to 2
- increases runtime, scales better

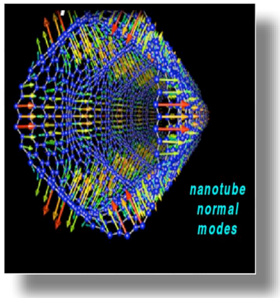
Cracked Interactions:
- back to 128 bodies/box
- better than initial, but tails off (contention?)

Molecular Dynamics (MD) Simulation

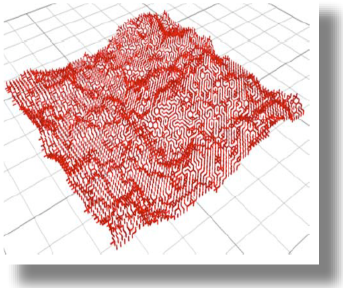
Application areas of MD Simulations



Biology



Chemistry



Materials/Nanotech

- Time evolution—integration of Newtonian Equation of Motion: $F_i = m_i * a_i$. Force (F), mass (m) and acceleration (a) of a particle *i*.
- Computational complexity: N^2 (N—number of atoms) or $N*N_c$ (N_c—number of atoms within cutoff limit)
- Characteristics:
  - Computationally intensive calculations
  - Random memory access patterns
  - Dynamic runtime behavior

Implementation & Optimization of an MD Kernel on MTA2

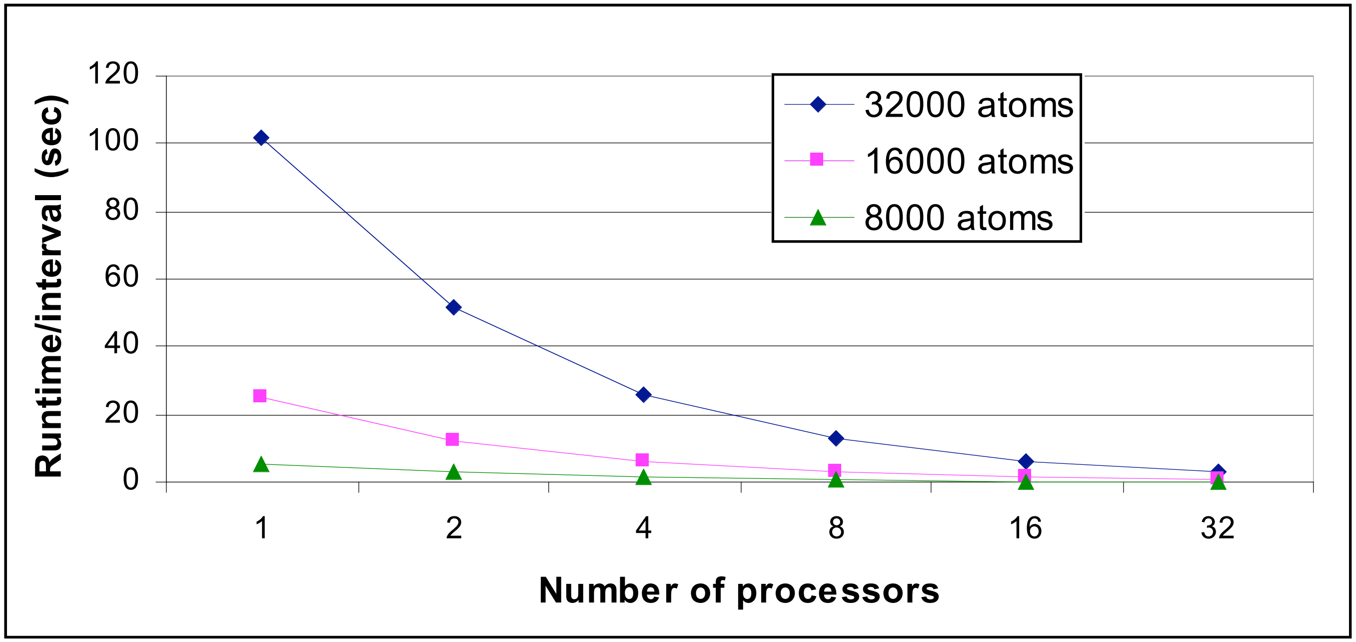
- Our MD kernel contains force evaluation and integration routines
- Bonded forces are deterministic—straightforward to compute
- Simulation targets:
  - Longer time-scale simulations (strong-scaling mode)
  - Larger systems simulations (weak-scaling mode)
- Non-bonded forces modeled by LJ model

$$V(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$$

```
1.  advance velocities
2.  calculate potential energy and forces
      for i=1 to N atoms
        for j=1 to N-1 atoms
          if (i & j in cutoff limits)
             compute force
3.  complete velocities update
4.  calculate new kinetic and total energies
```

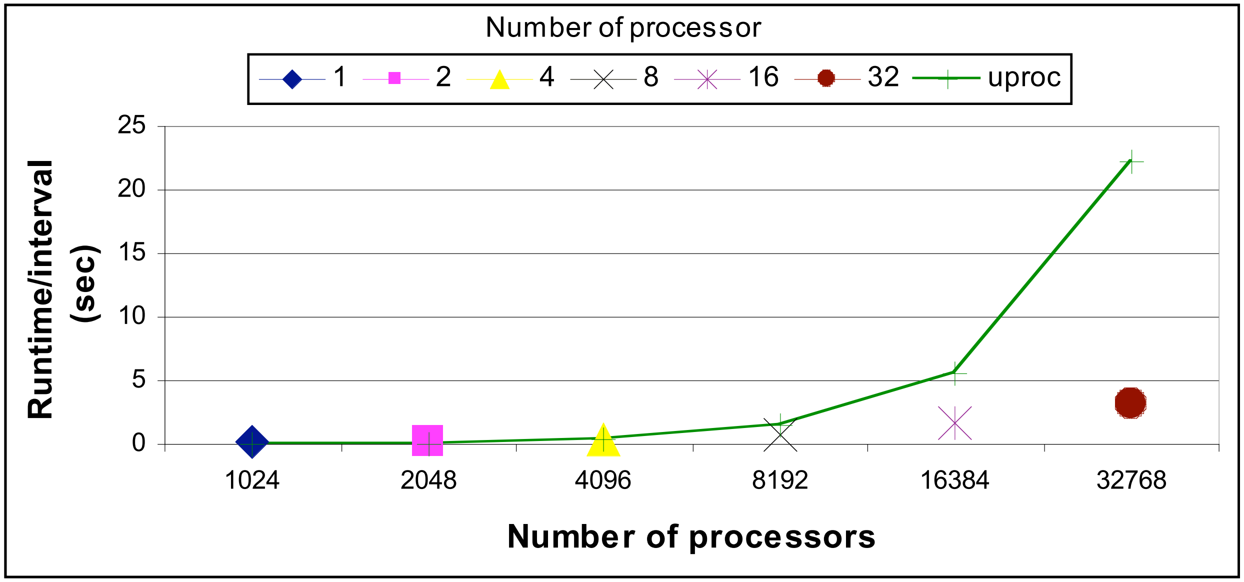
MTA2 compiler parallelized the main loops by moving a scalar calculation outside of the loop—very low implementation overhead

Performance Evaluation (MD Kernel)



- Strong scaling mode results—overall problem size fixed
- Ideal speedup (speedup = $time_{oneMTA2}/time_{nMTA2}$) for all three test cases (8000, 16000 and 32000 atoms) on up to 32 MTA2 processors

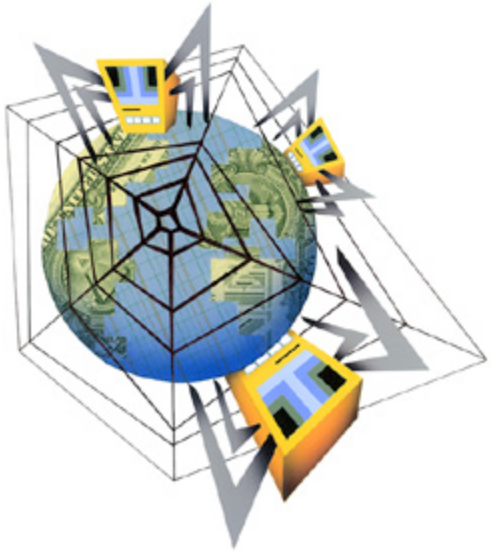
Performance Evaluation (MD Kernel)



- Weak Scaling mode—by increasing the problem size and number of MTA2 processors *2
- Not ideal—compute time increase with problem size due to load imbalances
- Significantly better than a microprocessor—computational complexity: N^2 (N—number of atoms) or $N*N_c$ (N_c—number of atoms in cutoff limit)

Discrete-event Simulation (DES)

- Modeling of time dependents systems
- Asynchronous system
- Time-stamped events (do not model a single time step)
- Inherently sequential—event queue is updated after processing an event
- Applications:
  - Internet modeling
  - Computer & telecommunication network modeling
  - Service systems modeling
  - Security networks
  - Real-time decision making



A Simplified DES Kernel

- Basically, a tree-based priority queue and two loops:
  - Loop 1: Insert N elements
  - Loop 2: Remove all N elements
- A straightforward, but *inefficient*, parallelization strategy:
  - Only permit one thread to insert/remove at a time

```
For 1 to MAX_ELEMENTS in Parallel
   Create an event with a random timestamp
   lock()
   Insert event in Priority Queue
   unlock()

For 1 to MAX_ELEMENTS in Parallel
   lock()
   Remove the event with minimum timestamp
   unlock()
```

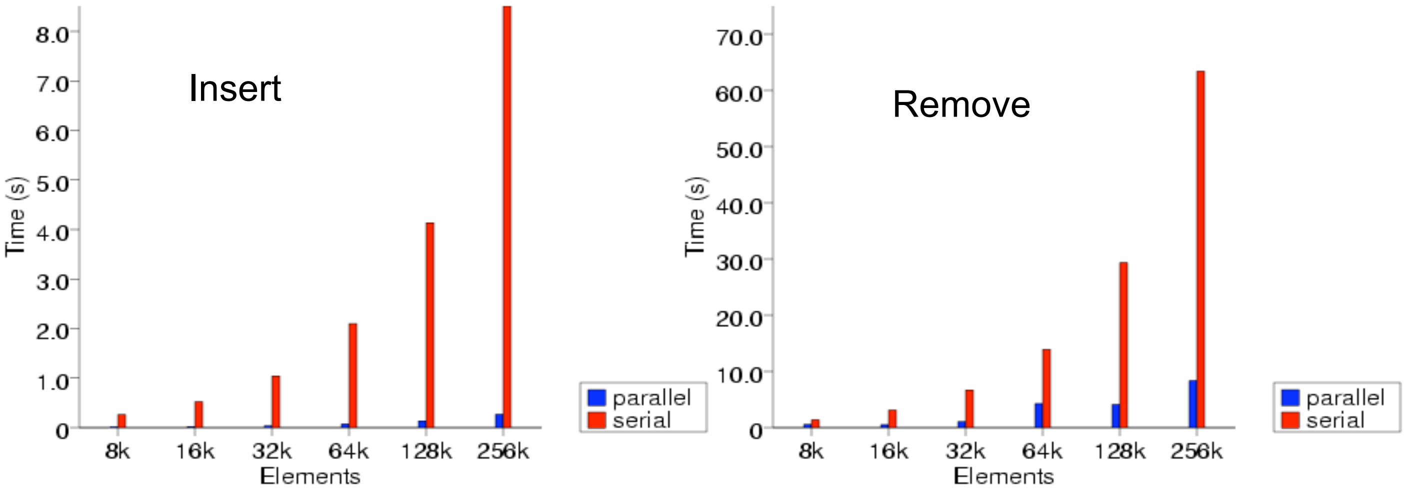
- Ques ithin priority queue enable parallel insertions/removals? *Profitably*??

MTA PQ Implementation

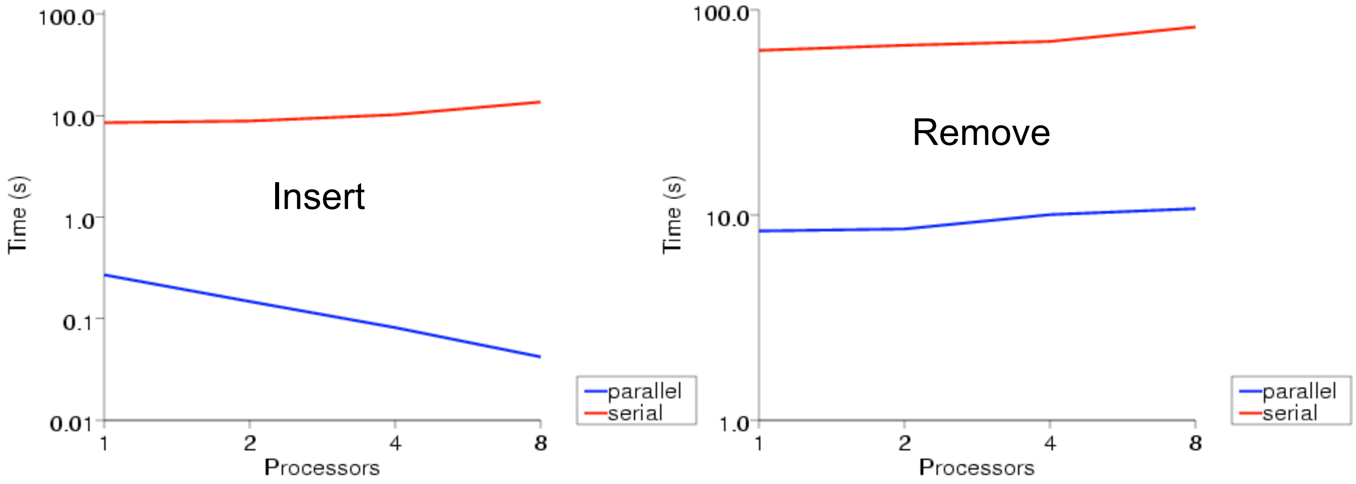
- Priority Queue Insert
  - Sequential:
    - Add element as binary tree leaf
    - Move up tree, SWAP()'ing w/ parent, until > parent
  - Parallel:
    - Atomic fetch_add_int() to find leaf in which to add element
    - Lock child and parent before SWAP()…
- Priority Queue Remove
  - Sequential:
    - Remove root, move leaf to root
    - Move down tree, SWAP()'ing w/ smallest child, until both children >
  - Parallel:
    - Atomic fetch_add_int() to find leaf to move
    - Lock root and leaf before removal/move
    - Lock parent and each child before moving down

Parallel Performance (vs Serial)

Single processor, multiple element counts:



Multiple processors, single element count (256K):



Conclusions

- Answer to our question:
  - YES, PQ insertions and removals can be done in parallel
  - Insert - surprisingly large amount of parallelism available
  - Remove - definite benefit for 1p, but currently too much synchronization to be scalable
    - More scalable as number of elts increases?
    - More efficient use of locks possible?

- Other areas for investigation:
  - More difficult proposition: can Inserts and Removes occur at the same time?
  - Priority queue might not be the best choice of data structure for DES on the MTA…others?

Acknowledgements

- This research was sponsored by the Office of Mathematical, Information, and Computational Sciences, Office of Science, U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.
- Cray, including Supercomputing Center of Excellence
  - John Feo