Experiments Using IBM’s Software Transactional Memory Compiler

Barna L. Bihari
ICON Consulting, Inc.
Representing IBM at LLNL
Livermore, California

John Gyllenhaal, Tom Spelce, Scott Futral
Lawrence Livermore National Laboratory
Acknowledgements

- Dong Ahn
- Bronis de Supinski
- Peng Wu (IBM)
Transactional memory: what is it?

**Motivation:** Increased number of cores per node for the foreseeable future – shared memory parallel programming or threading

**Problem:** Multiple threads in a shared-memory environment can lead to race conditions, hence memory conflicts and incorrect execution

**Old Solution:** Traditionally users had to resort to synchronization techniques such as barriers or mutex type locks

- Locks are expensive and can result in dead-locks

**New Solution:** Transactions allow operations to go through, but check for memory violations and *unroll* it if necessary

- Transactional memory (TM) ensures atomicity and isolation
- An abstract construct that hides its details from the programmers
- It is intended to make parallel threading easier and more efficient
- Suggested use: when conflict probability is relatively low

**Transactional memory enables efficient high level threading**
The IBM solution

• IBM has a freely available software implementation of transactional memory
• Currently available for AIX: http://www.alphaworks.ibm.com/tech/xlcstm
• It uses OpenMP constructs
• IBM STM features:
  - runtime library has standard and debugging versions
  - ensures weak isolation of atomic regions (as opposed to “strong isolation”)
  - TM languages extensions: #pragma tm atomic default(trans)
  - TM function attributes
  - intrinsic functions: tm_trans_read/write, tm_notrans_read/write
  - runtime statistics dumped via stm_print_stats
  - compiling and linking via stmxlc and stmxlc++ commands
  - all other XL options valid for C/C++ Enterprise Edition for AIX, v9.0
  - need to link via –Istm (standard) or -Istmstat (debug)
  - compiler instrumentation report available via the –qreport link option

IBM STM is a freely available state-of-the-art compiler
Current stage of experimentation

- Installed and tested the IBM STM on LLNL’s AIX systems
- Current focus: identify algorithms that would benefit from STM
- Present: actual runtime and parallel speed-up is secondary
- Current goal: learn about potential fertile grounds for TM using STM, hoping for eventual HTM (hardware transactional memory)
- Future goal: actual runtime is ultimate bottom line - via HTM
- Expect much of current work to carry over – OpenMP standard?
- Algorithms with low but nonzero chance of conflicts are prime candidates
- To date: identified two such algorithms (frequent in simulations)

TM is not for every code: round up the (usual?) suspects
Designing the Toy Code/Benchmark

- Spawn multiple threads
- Have at least one “parallel for” loop
- Should be relatively load balanced
- Should mimic the loop-structure of an actual simulation code or code segment
- Should generate race conditions between threads (i.e. conflicts)
- Some number crunching in between memory updates
- Error checking should be simple
- Upon request, be able to get the “right” answer
- Should sometimes be able to get the “wrong” answer

Goal: simulate thread conflicts in a multiphysics code without the actual code
The simplest toy code one could think of

- The code has one key variable for each thread
- Conflicts on this variable occur in a random way
- Threads increment their own or other’s counter
- Outer loop increments conflict probability at each pass (0 to \((n\_threads-1)/n\_threads\))
- The update is \textit{atomic}
- Checksum over all threads is independent of:
  - conflict probability
  - number of threads
- We always get the correct checksum with STM
- No number-crunching yet

\textbf{We can confirm that STM corrects unresolved conflicts}
The Syntax

Specifying a transactional memory section

#pragma tm atomic default(trans)
{
    count_array[threadid]++;
}

Specifying the parallel (threaded) region

#pragma omp parallel for
    for (i=0; i < iter_count; i++){
        Main body of threaded computation
    }

Calling the STM diagnostic routine

stm_print_stats();
Actual conflicts vs. conflict probabilities

- 2 Threads
- 4 Threads
- 8 Threads
- 16 Threads
Benchmark code design

• Goal: quick experimentation without production code changes
• Be able to use realistic unstructured mesh connectivity
• Have both deterministic and random run modes
• Chance of conflict low or very low, but cannot be ruled out
• Easy parallelization: embarrassingly parallel loops?
• Current name: **BUSTM** (**Benchmark for Unstructured-mesh Software Transactional Memory**)
• Have at least *some* resemblance to *some* real algorithms
• Two targets so far:
  - deterministic CFD
  - Monte Carlo transport

**Benchmark codes allow for fast feasibility studies**
Unstructured-mesh infrastructure

- Code can handle cells with arbitrary number of faces
- Three (hierarchical) basic element types are used for implementation:
  1. nodes or grid points (physical xyz-points in 3-D space)
  2. faces (boundaries between cells) - contain nodes
  3. cells - contain faces and nodes
- Practical cell types:
  - Tetrahedron (4 faces, 4 nodes)
  - Hexahedron (6 faces, 8 nodes)
  - Triangular prism (5 faces, 6 nodes)
  - Pyramid (5 faces, 5 nodes)
- Connections between the three basic elements:
  - each face knows which cells are on either side (2 per face)
  - each face knows which nodes are part of it (3 or 4 per face)
  - each cell knows which nodes are part of it (4 to 8 per cell)
  - each cell knows which cells are part of it (4 to 6 per cell)
1. “Deterministic” Conflicts

E.g. **Conservative finite volume schemes** on unstructured meshes

- Compute-intensive loops are face-based
- Each face has 2 cells on either side
- Cells are updated based on face-based loops
- Face (flux) computations are compute-heavy
- Probability of conflicts is very low, but nonzero
- Traditional thread-safety can be expensive
- Potential debugging nightmare
- Transactional memory can have a huge payoff
Test of STM applicability

- The set of connections between the different element types forms a *graph*
- Two of the three basic elements form the nodes and edges of the graph
- *Indirect indexing* is pervasive throughout such unstructured-mesh codes
- Example of a triangular prism mesh and its corresponding graph might be:

  ![Mesh and graph example](image)

- Heart of any conservative CFD code: *flux computation* – done face-by-face
- Question: How to simulate flux computation without a full CFD-solver?

  **Answer:** Compute the *numerical gradient instead*
Unstructured-mesh algorithm

• Goal: compute the gradient of a function on an unstructured mesh
• Use well-known formula

\[ \nabla f = \frac{1}{|V|} \int_{\partial V} \vec{n} f dS \]

• Approximated at cell \( j \) by:
  - where \( i \) runs over the local faces of cell \( j \)
  - \( \vec{n} \) is the cell face normal (precomputed)

Similar loops compute the numerical divergence essential in conservative CFD codes on unstructured meshes
Relevant code section

Gradients computed by accumulation within TM section of `compute_cell`

```c
#pragma tm atomic default(trans)
{
    gradient[cell_no_1] += incr;
    gradient[cell_no_2] -= incr;
}
```

The general parallel (threaded) region is face-based

```c
#pragma omp parallel for
for (i=0; i < max_face; i++){
    left_neighbor = left_cells[i];
    right_neighbor = right_cells[i];
    compute_cell(incr, left_neighbor);   (face increments cell)
    compute_cell(incr, right_neighbor);  (face increments cell)
}
```

Error checking: assume \( f = \text{constant} \), then \( \text{grad } f = 0 \)
Actual cases: 2 meshes used

- Used two 3-D meshes for 2-D testing:
  - 1. **Prism mesh** – 2-D layer of 3-D cells (medium):
    - 119893 cells, 420060 faces (240655 BC +179405 interior), 123132 nodes
  - 2. **Hex mesh** – 2-D layer of 3-D cells (small):
    - 3000 cells, 12110 faces (6220 BC + 5890 interior), 6324 nodes

Test cases are borrowed from CFD research simulations
- Conflict occurrence (detected by STM) is between 0% and 0.00042%
- No conflicts on 1 or 2 threads
- Number of STM-retries is much less than that of “bad results” (non-STM)

STM fixed the conflicts – much fewer “retries” than “wrong answers”
Hex mesh results

- Conflict occurrence (detected by STM) is between 0% and 0.1%
- No conflicts on 1 or 2 threads
- Number of STM-retries is comparable to number of “bad results” (non-STM)

STM fixed the conflicts – “retries” comparable to “wrong answers”
2. Probabilistic Conflicts

- Imagine a (large) number of randomly released particles travelling through a mesh composed of cells.
- Each particle operates on many mesh cells as it hits them.
- Parallelized (i.e. threaded) loop is over particles.
- Conflicts can occur as more than one particle hits the same cell.
- This is a simplistic view of a Monte-Carlo simulation.
- Embarrassingly parallel loops, except for the conflicts.
Test of STM applicability

• There is no one-to-one correspondence between particles and cells
• Question: How to simulate particles without the physics?
• Answer: Use unstructured mesh connectivity to “guide” particles
• Add probabilistic flavor by:
  1. randomly selecting in which cell the particle is born
  2. randomly selecting which cell-neighbor is next on the particle’s path

Conflict simulation: Multiple particles hit same cell
Relevant code section

Cell counter is incremented within TM section of routine mark_cell

```c
#pragma tm atomic default(trans){
    cell_counter[cell_no] ++;
}
```

The parallel (threaded) loop is over the particles

```c
#pragma omp parallel for
for (i=0; i < max_particles; i++){
    next_cell = rand();
    while(inside){
        mark_cell(next_cell); (particle increments cell)
        next_face = rand();
        next_cell = neighbor(next_face);
        if(next_cell < 0)inside = 0;
    }
}
```

Error checking: total cell hits = total path lengths
- Input: number of particles = 12000 (10% of total number of cells)
- Conflict occurrence (detected by STM) is between 0% to 0.0099%
- Number of STM-retries is quite a bit higher than number of “bad results” (non-STM)

**STM fixed the conflicts – much fewer “wrong answers” than “retries”**
- Input: number of particles = 15000 (5 times the total number of cells)
- Conflict occurrence (detected by STM) is between 0% to 0.26%
- Number of STM-retries is quite a bit higher than number of “bad results” (non-STM)

**STM fixed the conflicts – much fewer “wrong answers” than “retries”**
Early results on timings

- We know that STM has large overheads, but…
  … should we time it anyways?
- Make sure statistics/diagnostic option is turned off
- It would give us an idea about parallel behavior
- It would create a minimum expectation for HTM
- Compare to:
  1. non-thread-safe code (unfair, but informative)
  2. thread-safe directives, such as “critical” (fair)

Results on timings are tentative and approximate
Run-times: deterministic case
Prism mesh

- Problem scales well (constant user time in “unsafe” mode)
- STM is slightly faster than “critical” on multiple threads
- Single thread overhead for STM is very high
Run-times: deterministic case

Hexahedral mesh

- Problem scales reasonably up to 4 threads
- STM is slightly faster than “critical” on multiple threads
- Single thread overhead for STM is very high
Run-times: probabilistic case
Prism mesh

- Problem does not scale (uneven particle path lengths)
- STM is slightly faster than “critical” on multiple threads
- Single thread overhead for STM is very high
Run-times: probabilistic case
Hexahedral mesh

- Problem does not scale (uneven particle path lengths)
- STM is slightly faster than “critical” on multiple threads
- Single thread overhead for STM is very high
Summary and Current/Future Work

• Transactional Memory promises to make thread-safe programming easier while keeping efficiency
• IBM has STM available now, ready to be tried out
• STMXLC has a useful statistics/diagnostics tool
• Demonstrated two algorithms which may benefit
• Pay-off is highly algorithm- and problem-dependent
• In compute-heavy codes STM over-predicts conflicts
• It is already competitive with OpenMP “critical”
• Need more accurate timing routines (CLOMP code)
• Will try fine-tuning of STM sections/routines
• Need to demonstrate on real simulations