Experiments Using BG/Q’s Hardware Transactional Memory

Barna L. Bihari

Lawrence Livermore National Laboratory
Livermore, California
Acknowledgements

Technical:
• Scott Futral
• Amy Wang (IBM)
• Bronis de Supinski
• John Gyllenhaal

• Don Lipari
• Martin Schulz
• Dave Fox
• Martin Schindewolf

Financial support (present and past):
• IBM
• ICON Consulting, Inc.
• LLNL

• Boeing
• Rockwell International
• HyPerComp, Inc.

Some of the material here has been presented at previous LLNL meetings and various conferences.
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 them 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

**Goal:** enable efficient high level threading
The IBM Hardware Solution

- IBM has Hardware Transactional Memory (HTM) on Blue Gene/Q
- It uses OpenMP constructs – syntax is trivial
- Synchronized in L2 cache and implemented via multiversioned cache
- “Short transactions” also available, as well as “omp atomic” (single instruction)
- IBM HTM features:
  - ensures strong isolation, atomicity and consistency of atomic regions (as opposed to “weak isolation” with STM)
  - TM OpenMP extension: #pragma tm_atomic
  - runtime intrinsic query functions: tm_get_stats/tm_get_all_stats
  - runtime statistics dumped via tm_print_stats/tm_print_all_stats
  - compiling and linking via the cross compilers on BG/Q using the -qtm option
  - all other XL options valid for C/C++
  - various environmental variables control specific behavior

BG/Q’s HTM is the first production quality transactional memory system
Previous stage of experimentation: STM and HTM on Simulators

- Installed and tested the IBM STM on LLNL’s AIX systems
- Previous focus: identify algorithms that would benefit from TM
- In the past actual runtime and parallel speed-up was secondary
- Previous goal: learn about potential fertile grounds for TM using STM, hoping for eventual HTM (hardware transactional memory)
- Eventual goal: runtime/performance is ultimate bottom line, via HTM
- Expect much of current work to carry over to other platforms (such as Intel) – OpenMP standard eventually?
- 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**
Current stage: HTM on Early Access Blue Gene/Q Systems and Sequoia

- Access to BG/Q systems at Rochester, as “non-paid IBM contractors”
- Current focus: transition from functionality to performance of TM
- Actual runtime and parallel speed-up are being tested for small codes
- Current tasks: study the
  1. correctness
  2. retry patterns as a function of probabilities and number of threads
  3. TM performance issues
- Future goal: performance testing/tuning for more benchmarks/codes
- Algorithms with low but nonzero chance of conflicts and which are non-L1-cache–friendly are prime candidates currently
- To date: (i) used tm_rand and BUSTM, for testing functionality
  (ii) used CLOMP for performance testing (runtime/scaling)

**TM is not for every code:** round up the (usual?) suspects
Specific Hardware Features

- We are currently working on performance issues: trying to quantify TM overhead/efficiency using CLOMP-TM
- Multi-versioned cache
- Short transactions vs. long transactions vs. OMP atomic
- TM synchronizes in L2-cache, with latency of about 75-80 cycles
- L1 cache size is 16K/core, with 64 byte cache lines
- L2 cache size is 32M/node, with 128 byte cache lines
- BG/Q also has an L1P cache for prefetch streaming with 4K/core
The STM vs. HTM syntax

1. Specifying a software/hardware transactional memory section

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

```c
#pragma tm_atomic
{
    count_array[threadid]++;  
}
```

2. Specifying the parallel (threaded) region

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

3. Calling the TM diagnostic routine

```c
stm_print_stats();
```

```c
export TMREPORT_ENABLE=YES

tm_print_stats();
```
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 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**
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:

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

• Task: compute the gradient of a function on an unstructured mesh

• Use the well-known formula:

\[ \nabla f = \frac{1}{|V|} \oint_{\partial V} \bar{n} f dS \]

• Approximated at cell \( j \) by:

\[ \nabla f_j \approx \frac{1}{|C_j|} \sum_i n_i f_i \]

- where \( i \) runs over the local faces of cell \( j \)
- \( n \) 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 \texttt{update\_cells}

\begin{verbatim}
#pragma tm atomic default(trans)
{
    gradient[cell\_no\_1] += incr;
    gradient[cell\_no\_2] -= incr;
}
\end{verbatim}

The general parallel (threaded) region is face-based

\begin{verbatim}
#pragma omp parallel for
    for (i=0; i < max\_face; i++){
        left\_neighbor = left\_cells[i];
        right\_neighbor = right\_cells[i];
        update\_cells(incr, // face increments cells
                      left\_neighbor, right\_neighbor);
    }
\end{verbatim}

Error checking: assume $f = \text{constant}$, then grad $f = 0$
Actual case: realistic unstructured mesh

- Used a 3-D tetrahedral mesh
  - 362,429 cells
  - 739,480 faces
  - 67,938 nodes

Test cases are borrowed from CFD research simulations
Results for Tetrahedral Mesh – Deterministic Mode

- Very few rollbacks: < 100 during most runs
- Conflict probability \( \approx 0.00013\% \) even at the maximum
- This is a small mesh – larger meshes would probably have even fewer conflicts
- TM is about 10% faster than OMP atomic
- TM is scalable up to about 16 threads
- OMP atomic scalable up to 64 threads
- OMP atomic catches up at 16 threads
- OMP critical does not scale, fast on 1 thread
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
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
Results for Tetrahedal Mesh – Deterministic Mode

- Most rollbacks on 16 threads
- Conflict probability ~ 0.045% at the maximum
- Unusually high error rate at 2 threads
- TM is about 15% slower than OMP atomic
- TM is scalable on up to 16 threads
- OMP atomic scalable up to 64 threads
- Single instruction TM – same as OMP atomic
- OMP critical does not scale, fast on 1 thread
Performance Analysis – CLOMP-TM

- Parameter set defines:
  - Parts, Zones, scatterZones,
  - ScatterMode defines contention
    - None, Adjacent, Random, firstParts, and mixed modes
  - Configurable amount of computation per zone update
    - none, divide, complex
    - Scaled by a factor

- Compare different synchronization mechanisms
  - Quantify overheads
  - Determine best length
- Without any synchronization get very high scalability (but wrong answer…)
- “Large TM” performs well – recently we have “Huge TM” (not shown)
- “(Small) Atomic” is best when it is usable (i.e. single instruction)
- Critical does not scale at all – neither “Small Critical” not “Large Critical”
Experiments with CLOMP-TM-MPI

- Add MPI_Barrier construct
  - All MPI tasks execute one thread synchronization mechanisms at the same time
  - Increases pressure on architecture
- Determine best task-to-thread ratio

```c
MPI_Barrier(MPI_COMM_WORLD);
get_timestamp (&bestcase_start_ts);
do_bestcase_version();
get_timestamp (&bestcase_end_ts);
MPI_Barrier(MPI_COMM_WORLD);
```
- Total of 64 “generalized tasks” are allowed per node – MPI tasks * threads = 64
- “Large TM” performs well in low contention case, acceptable for high contention
- “(Small) Atomic” performs best in high contention cases
- Surprise: “Critical” does better at higher MPI task count than atomic
- OpenMP version with highest possible thread count performs the best in general
Summary and Current/Future Work

• For select algorithms, transactional memory promises thread-safety, easier programming, and performance simultaneously
• IBM BG/Q has HTM available using hardware implementation
• Omp atomic scales well on up to 64 threads
• TM scales on up to 16 threads
• TM pay-off is expected to be highly code- and problem-dependent
• For HTM, we study both functionality and performance
• SWL performance issues are being worked on with IBM
• Target algorithms and codes must be carefully chosen
• Need to demonstrate on more benchmarks and eventually real simulations/codes
• Looking for collaborators in our quest for new candidates for TM