Asynchronous Programming in UPC: A Case Study and Potential for Improvement

Aniruddha G. Shet, Vinod Tipparaju, Robert J. Harrison

Oak Ridge National Laboratory

Research sponsored by the Laboratory Directed Research and Development Program and Post Masters Research Participation Program of Oak Ridge National Laboratory (ORNL), managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725
Today: Exceeding a Peak Petaflop

- **Hardware characteristics (~1.3-1.5 petaflops peak)**
  - 10,000-100,000 processors
  - 2-4 cores per processor
  - Homogeneous processor environment

- **Application characteristics**
  - Scaling up problem size/resolution
  - Increasing physical fidelity/model complexity
  - Early explorations of coupled simulation

- **Dominant programming model**
  - Sequential language (Fortran/C/C++)
  - 2-sided messaging library (MPI)
  - threads (OpenMP)
  - Age: ~30 years
Tomorrow: Sustained Petaflops and Beyond

- Hardware characteristics (10-100 petaflops peak)
  - 100,000-1,000,000 processors
  - 100-1,000 cores per processor
  - Heterogeneous processor environment may be common
    - Example: LANL Roadrunner: Opteron, PowerPC, Cell
    - Also GP-GPUs, integrated GPUs, FPGAs, etc.

- Application characteristics
  - Scaling up problem size/resolution leveling off
  - Increasing physical fidelity/model complexity
  - Serious coupled simulation
  - Serious algorithmic scaling challenges
    - Increase in multi-level parallelism
    - Increase in adaptive representations, irregular computations

- Dominant programming model
  - ???
Asynchronous Programming: What and Why?

- Application developer’s working definition – express parallelism in application algorithm to be managed by programming model implementation

- Benefits
  - Encompasses many different kinds of parallelism, which is going to be increasingly important given hardware and application trends
  - Decouples logical from physical parallelism for easier user code composition
  - Efficient portable execution from smarter runtimes handling the expressed parallelism to perform dynamic load balancing, fault handling, resource management etc.
  - Natural style of writing irregular, tree-based codes
Why PGAS? Why UPC?

• Two-sided message-passing communication was not found to be suited for adaptive, recursive concurrency

• PGAS is a convenient model in which to develop an irregular, distributed application
  – Shared view with locality-awareness, direct remote memory access

• HPCS languages are an attractive option due to integrated support for PGAS and asynchronous concepts, but are works-in-progress

• More importantly, we were out to explore asynchronous programming in the more traditional SPMD context

• Starting point was serial code in C
What is (the) MADNESS (about)?

- **Multiresolution Adaptive Numerical Environment for Scientific Simulation (MADNESS)**
  - Programming environment for the solution of integral and differential equations based on Multiresolution Analysis (MRA)

- Fast algorithms with guaranteed precision
  - Trade precision for speed

- High-level composition of numerical codes
  - Work with functions and operators

- Target applications
  - Quantum chemistry, atomic and molecular physics, material science, nuclear structure
MADNESS: Adaptive Refinement

- Refinement starts at one node and is applied recursively from parent to children nodes.
- The degree of parallelism is determined by the desired numerical accuracy.
- A key feature is the provision to refine select portions of the tree as need be.

Refinement is a recursive, top-down, adaptive tree algorithm.
MADNESS: Tree distribution

- Multiresolution adaptive properties produce unbalanced coefficient trees
  - binary tree in 1-d, quadtree in 2-d, octree in 3-d etc.
- Tree structure evolves in unscheduled ways due to very flexible adaptive refinement
- Need a scheme to partition the complete tree as an entire tree, and not just leaf nodes, is utilized in some algorithms

Binary tree numerical form of a 1-d analytical function. Note that some intervals are not sub-divided due to the adaptive nature of refinement.
Serial Refinement – C

Serial data structures

- Hash table of (node, tensor) pairs
  ```c
  typedef struct {
    size_t order;
    GHashTable *tree;
  } SubTree;
  typedef SubTree *subtree_t;
  typedef struct {
    subtree_t sumC;
    /* Math data structures */
  } Func;
  typedef Func *func_t;
  ```

Serial program

- MRA function, target node being refined
  ```c
  void refine(func_t f, node_t node) {
    /* Math logic to calculate normf */
    for (int c=0;c<children(node);c++) {
      node_t child = get_child(node,c);
      if (normf > threshold)
        refine(f,child);
      else
        insert_coeffs(f,child);
    }
  }
  ```

- Recursive call to refine child
- Add a (node, tensor) pair to function tree
Parallelization in Standard UPC: Asynchronous Refinement

Global data structures

typedef shared Func *GFunc_t;
typedef shared GFunc_t*gfunc_t;

gfunc_t gf = upc_all_alloc(THREADS, sizeof(GFunc_t));
gf[MYTHREAD] = upc_alloc(sizeof(Func));

Parallel program

global MRA function

void refine(gfunc_t gf, node_t node) {
    /* Math logic to calculate normf */
    for (int c=0; c<children(node); c++) {
        node_t child = get_child(node, c);
        if (normf > threshold)
            gtaskq_launch(gtaskq, thread_id, refine_task);
        else
            gtaskq_launch(gtaskq, thread_id, insert_coeffs_task);
    }
}

if (MYTHREAD == 0)
    gtaskq_launch(gtaskq, thread_id, refine_task);

gtaskq_execute(gtaskq);
Parallelization in Standard UPC: Tree distribution

• A node is mapped to the UPC thread given by:
  \[ \text{hash}(\text{node}) \mod \text{THREADS} \]

• Not optimized for locality

• Owner-computes policy gives an effective distribution of the workload
Parallelization in Standard UPC: Global Task Queue

Structure

```
<table>
<thead>
<tr>
<th>lock_dir</th>
</tr>
</thead>
<tbody>
<tr>
<td>tail_dir</td>
</tr>
<tr>
<td>task_dir</td>
</tr>
</tbody>
</table>
```

```
| task buffer |
| head |
| 0 |
| PGAS |
| THREADS-1 |
| task buffer |
| head |
```

API

```
gtaskq_t gtaskq_init(size_t taskq_size, task_func_t *func_table, size_t ftable_size);

void gtaskq_launch(gtaskq_t gtaskq, size_t thread_id, task_t *task);

void gtaskq_execute(gtaskq_t gtaskq);

void gtaskq_destroy(gtaskq_t gtaskq);
```
Parallelization in Standard UPC: Global Task Queue

• Each UPC thread locally allocates a shared task buffer; these are linked together in the PGAS to form the global queue

• A tail variable indicates the next slot in a thread’s task buffer to insert a task; a head variable indicates the next task to execute

• A UPC lock synchronizes access to a thread’s task buffer

• `gtaskq_launch` adds a task to another thread’s task buffer

• A task may create child tasks using `gtaskq_launch`; a task always runs to completion and does not return control back

• All threads invoke `gtaskq_execute` to perform execution and termination* of the dynamically unfolding tree of tasks

*Francez, N., Rodeh, M.: Achieving distributed termination without freezing
Experimental Results: Setup

- **Machine**: Smoky @ National Center for Computational Sciences
  - 80-node Linux cluster, each node with 4 quad-core 2.0GHz AMD Opterons and 32GB memory
  - Nodes are connected by InfiniBand network

- **Compiler**: Berkeley UPC v2.8.0 on OpenIB InfiniBand Verbs conduit

- **Libraries**: GLib 2.18.4 for hash table structure

- 1 UPC thread per core, 2 pthreads per UPC thread
Experimental Results:
Wall time in Standard UPC

**3D**

**1D**
Experimental Results: Micro-benchmarking

Computing time of MADNESS methods in microseconds

<table>
<thead>
<tr>
<th>Code</th>
<th>refine</th>
<th>insert_coeffs</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D</td>
<td>3099</td>
<td>153</td>
</tr>
<tr>
<td>1D</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Task launch and execute times in microseconds

<table>
<thead>
<tr>
<th>threads</th>
<th>launch</th>
<th>lock</th>
<th>execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>81</td>
<td>20</td>
<td>0.43</td>
</tr>
<tr>
<td>2</td>
<td>91</td>
<td>25</td>
<td>91</td>
</tr>
<tr>
<td>4</td>
<td>313.99</td>
<td>200.33</td>
<td>106</td>
</tr>
<tr>
<td>8</td>
<td>676.43</td>
<td>567.86</td>
<td>97</td>
</tr>
<tr>
<td>16</td>
<td>1450.07</td>
<td>1336.07</td>
<td>97</td>
</tr>
</tbody>
</table>

Current programming models do not provide the right tools for easy & efficient manipulation of irregular, adaptive, distributed data structures

- Standard solution of shared queues protected by locks hurts performance
- Alternative is to build your own solution using asynchronous puts (bupc_memput_async); not the most programmable or scalable way
Parallelization in UPC with Asynchronous Remote Methods (ARM)

• Design of ARM
  – Layered directly on top of GASNet’s Active Message (AM) interface; added ARM entries to GASNet’s AM handler table
  – Provided a new construct `bupc_arm` in Berkeley UPC to call GASNet’s AM layer with ARM entries
    • `void bupc_arm(size_t arm_handler_index, size_t thread_id, void *buf, size_t nbytes)`
    • Local completion semantics, potential for overlap
  – `gtaskq_launch` now invokes `bupc_arm` to perform remote task insertions as ARM
Experimental Results:
Wall time in UPC with ARM

3D

1D
Parallelization in UPC with ARM

• Limitations of current ARM design
  – \texttt{bupc\_arm} has semantics of GASNet’s AM interface
    • AM concept exposed at the application level
    • Cannot return a result
  – Termination detection relies on ordered message delivery
    • Unordered delivery of ARM messages is not an issue on the single-rail InfiniBand network used in our experiments
  – Only a static number of ARM entries are permitted
Conclusions

- PGAS simplified the parallelization of our dynamic and irregular application, MADNESS

- Asynchronous remote methods achieved within 7\% of ideal performance and 20-fold improvement over the Standard UPC implementation in some cases

- Asynchronous remote methods can provide substantial programmability and performance benefits to application developers
  - We hope this work motivates their inclusion in the UPC standard
Future Work

• Extend the design of asynchronous remote methods to be more general, portable, and adaptable to different UPC applications
  – Require more than active message semantics
• Demonstrate scaling of asynchronous remote methods on bigger machines
• Implement other MADNESS algorithms in UPC to derive a general solution for benchmarking of UPC implementations and systems suitable for the APGAS model