Los Alamos
NATIONAL LABORATORY
EST. 1943

LA-UR-15-20294
Approved for public release; distribution is unlimited.

Title: (U) A Comparison of Threading Models in an Unstructured Mesh Mini-App

Author(s): Ferenbaugh, Charles Roger

Intended for: Nuclear Explosives Code Development Conference (NECDC), 2014-10-20
(Los Alamos, New Mexico, United States)

Issued: 2015-01-16

Disclaimer:
Los Alamos National Laboratory, an affirmative action/equal opportunity employer,is operated by the Los Alamos National Security, LLC for the National NuclearSecurity Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Departmentof Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

(U) A Comparison of Threading Models in an Unstructured Mesh Mini-App

Charles R. Ferenbaugh
Los Alamos National Laboratory

LA-UR-15-XXXXX

Abstract

In many large physics codes that have historically run using MPI-only parallelism, developers are beginning to introduce various forms of threading. But it's not completely clear which threading strategies will be most effective. In this paper we compare the performance of several types of threading using the unstructured mesh mini-app PENNANT, adapted from parts of the LANL rad-hydro code FLAG. Results include different strategies for using MPI+OpenMP on multi-core CPUs and CUDA on GPUs, and are given for both conventional and advanced architectures.

Introduction

Large multi-physics code projects at NNSA laboratories and elsewhere are starting to grapple with the challenges of adapting to advanced computer architectures such as graphics processing units (GPUs), Intel Many-Integrated-Cores (MIC), and IBM Blue Gene/Q. These architectures, as diverse as they are, nevertheless share several characteristics distinguishing them from the more traditional architectures that are commonly used. One prominent feature of these architectures is a heavy reliance on threading, in both hardware and software, to achieve best performance.

Many programming mechanisms are available to implement various forms of threading, including OpenMP, Cilk, and Thread Building Blocks (TBB) for CPU-like architectures, and CUDA, OpenCL, and OpenACC for GPUs. NNSA labs are implementing abstraction layers on top of these mechanisms, such as Livermore's RAJA [1] or Sandia's KokkosArray [2]. And also, work is being done to develop domain-specific languages (DSLs) that support threading, such as Liszt [3]. However, much study is still needed to evaluate the various mechanisms and strategies for using them in codes. The purpose of this paper is to provide an initial performance comparison of various strategies for using OpenMP and CUDA, on both current and future architectures. The comparison will be done using the PENNANT unstructured mesh mini-app [4, 5], described in more detail below.

Three basic threading models

Here are three basic models for implementing threading in an application code. These models are not mutually exclusive: multiple models can be combined in a single code.

Data-parallel, loop level. The most straightforward way to add threading to an existing code is to add OpenMP loop annotations:

```
#pragma omp parallel for
for (int n = 0; n < num_points; ++n) {
    z[n] = a * x[n] + y[n];
    w[n] = z[n] * x[n];
    ...
}
```

This approach is relatively easy to implement in legacy codes, since it can be done in ways that are minimally invasive to the code. However, it has the potential downside that the relatively large number of parallel regions could impose a performance penalty for context switches between different regions, and between parallel and serial regions.

Instead of using OpenMP directly, this model can also be applied using an abstraction layer such as RAJA or KokkosArray. It can also be implemented on GPUs using CUDA, OpenCL, OpenACC, or NVIDIA's Thrust library.

Data-parallel, higher level. An alternate threading model was used in the baseline OpenMP version of PENNANT. The input mesh data is divided into (nearly) independent chunks. The hydro cycle code flow is then divided into phases, which can be executed in parallel over mesh chunks. The result looks something like this:

```
#pragma omp parallel for
for (int c = 0; c < num_chunks; ++c) {
    run_step1(nbegin[c], nend[c]);
    run_step2(nbegin[c], nend[c]);
    run_step3(pbegin[c], pend[c]);
    ...
}
void run_step1(int nbegin, int nend) {
    for (int n = nbegin; n < nend; ++n) {
        // do work on n
    }
}
// similarly for run_step2, etc.
```

(Note that, in the very simplest examples, this could be reduced to the loop-level model by loop fusion; but in general that's not possible.) This leads to a coarse-grained, chunk-level strategy with

much more work in each parallel region, and less context-switching than in a loop-level strategy. However, it requires the code and data to be refactored and specifically structured for chunk-based processing, in a way that would take some effort to apply to a legacy code not designed for threading.

This model could also be implemented using CUDA, OpenCL, or other similar mechanisms.

Task-parallel. A third possible model would be to execute multiple independent tasks concurrently. For example:

```
for (int cycle = 0; cycle < num_cycles; ++cycle) {
    #pragma omp task
    run_step1();
    #pragma omp task
    run_step2();
    ...
    #pragma omp taskwait
}
```

This approach has the potential to include the largest amount of work in each parallel region, and to minimize the amount of context-switching. However it would also require major refactoring.

This model could be implemented using other mechanisms such as CUDA or OpenCL. It could also utilize libraries and runtimes specifically designed for task parallelism, such as Legion [6], Uintah [7], or extensions of KokkosArray that are planned for future development.

The PENNANT mini-app

The tests shown here were performed using the PENNANT unstructured mesh mini-app. PENNANT contains a small subset of the basic physics algorithms adapted from the hydrodynamics packages in the LANL ASC code FLAG, specifically:

- Staggered-grid Lagrangian hydrodynamics [8]
- TTS subzonal pressure treatment [9, 10]
- Caramana-Shashkov tensor artificial viscosity [11]

PENNANT operates on general unstructured meshes (arbitrary polygons) in 2-D, and gives a representative sample of the data structures and memory access patterns of an unstructured physics code such as FLAG. It contains about 3300 lines of C++ source code, and has complete baseline implementations for multicore CPUs (MPI + OpenMP) and GPUs (CUDA).

For the tests shown here, four variants of the MPI+OpenMP version of PENNANT were tested:

- **coarse**: baseline version, has 5 fairly large chunk-level parallel for loops per hydro cycle

Table I: Platforms used for testing

platform	cores	threads/core	freq. (MHz)	peak Gflops	power (W)
Sandy Bridge (TLCC2)	16		2600	332.8	230
Intel MIC (KNC)	60	4	1050	1065.0	225
IBM Blue Gene/Q	16	4	1600	204.8	63
NVIDIA M2090 (Fermi)	512		1300	665.0	225
NVIDIA K20Xm (Kepler)	2688		732	1310.0	235

Table II: Test problems used

test name	test type	# zones	# cycles	mesh type
nohsquare	spherical implosion	129600	7677	structured, all square zones
nohpoly	spherical implosion	63001	9876	unstructured, hexagon zones
sedov	point explosion	291600	3882	structured, all square zones
leblanc	shock tube	230400	3775	structured, all square zones

- **medium**: splits up some loops, has 13 chunk loops per cycle
- **fine**: puts every function call in its own loop, has 30 chunk loops per cycle
- **loop**: instead of chunk-level threading, adds loop-level pragmas to all loops

Since the CUDA version of PENNANT uses a threading strategy very similar to that of the MPI+OpenMP version, it was straightforward to implement three CUDA variants for testing. In each of the **coarse, medium,** and **fine** variants described above, each OpenMP parallel for loop was translated into a CUDA kernel. Since the CUDA version contains no explicit loops, there is no CUDA counterpart to the **loop** OpenMP version of PENNANT.

Performance Results

The following timing results were obtained on three different CPUs and two GPUs, described in Table I. Tests were done using a single node/card for each platform. Four standard hydro test problems were run, described in Table II.

Results are shown in Figs. 1-3. Across all platforms, the **coarse** variant is the fastest of all chunked variants, while the **fine** variant is the slowest. The OpenMP **loop** variant is the slowest of all. For the **fine** and **loop** OpenMP variants, the performance penalty relative to the **coarse** variant becomes larger as the problem size increases.

As expected, the **medium** and **fine** variants were slower than the **coarse** variant, presumably due to greater context switching and higher turnover of data in memory. The **loop** variant is slower still, especially on the BG/Q platform. It, too, has more context switching and memory turnover. In addition, it contains many atomic operations that are needed to make side-to-zone reductions work properly; these are not present in the chunked variants.

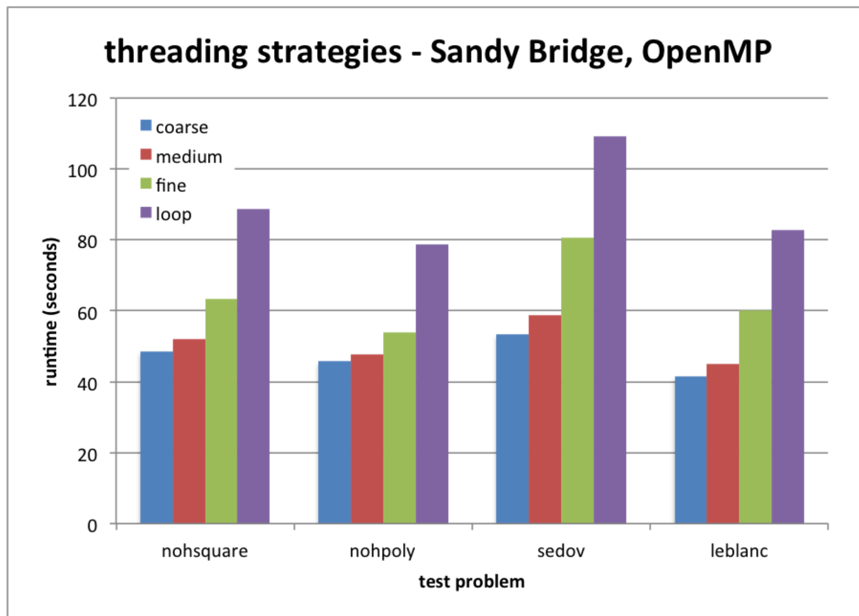

Figure 1: Timings for OpenMP PENNANT on Sandy Bridge CPU

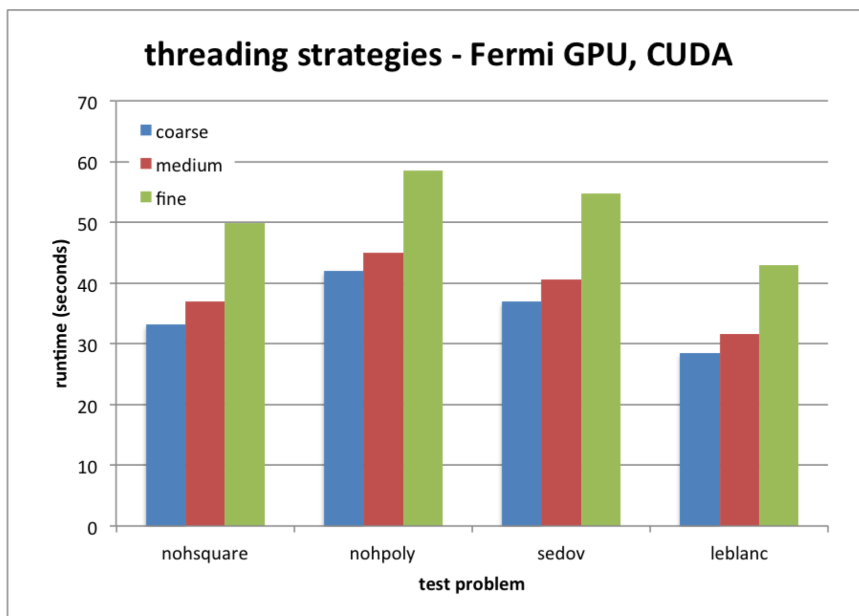

Figure 2: Timings for CUDA PENNANT on NVIDIA Fermi GPU

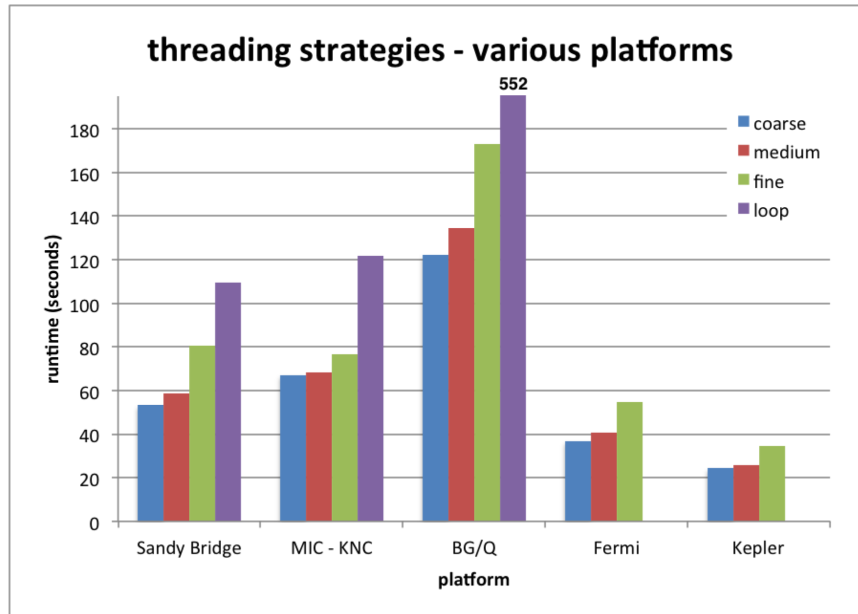

Figure 3: Timings for OpenMP and CUDA PENNANT on various platforms (Sedov problem only; other tests give similar results)

What about task-level parallelism?

Task-parallel models were not considered in the timing study shown above. We discuss some possibilities for task parallelism here.

In the **fine** and **loop** variants of PENNANT, some of the parallel regions are independent, so they could in principle be run as parallel tasks. One way to do this would be to hand-code the task parallelism in OpenMP, CUDA, or similar models. But in this case, the developer would have to analyze the dependencies between tasks, and hard-code a launch order for tasks with appropriate synchronization. This would be feasible for a mini-app such as PENNANT, but would be prohibitively difficult for most full-sized applications.

A more promising way to implement task-level parallelism would be to use a programming model designed to support it, such as Legion or Uintah. In such a model, the runtime would automatically analyze the dependencies, create a task graph, and execute tasks in an appropriate order. To explore this, a Legion implementation of PENNANT is in progress. As of this writing, PENNANT in Legion is code-complete, and performance optimizations are ongoing. Because this version is not yet fully optimized, it was not included in the timing study shown here.

It should be noted that PENNANT is probably not the best choice for testing task-parallel models, because PENNANT is very well load-balanced: the method is explicit, the data distribution does not change during the run, and only one type of physics is being run. Other applications that are not as well-balanced would likely be better candidates for evaluating task-parallel approaches.

Conclusions

In the tests shown here, coarse-grained threading over chunks consistently performed better than finer-grained chunk threading or loop-level threading. This suggests that, when developing new codes for advanced architectures, a coarse-grained chunking strategy is likely to be the most efficient threading strategy of those considered here. For legacy codes, the consideration becomes more difficult, since a significant amount of refactoring might be needed to support chunk-level threading. In this case loop-level threading could be a reasonable compromise for achieving some threading performance with limited development resources.

In some applications, it is likely that task-level parallelism will also be needed, in conjunction with the data parallelism investigated here. Further study in this area is needed.

Acknowledgements

This work was supported by:

- Mikhail Shashkov and the ASCR Mimetic Methods project
- ASC Hydrodynamics project
- ASC Programming Models project

Thanks also to:

- ASC Lagrangian Applications project
- Intel EPOCH workshop, June 2012

References

[1] Hornung, R. and Keasler, J., "RAJA Programming Model For Performance and Resilience," presentation for LLNL/LANL Next-Generation Platforms Working Group, July 2013. Also available as technical report LLNL-PRES-639392.

[2] Edwards, H. and Sunderland, D., "Kokkos Array performance-portable manycore programming model," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, New Orleans, LA (2012).

[3] DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., *et al.*, "Liszt: A domain specific language for building portable mesh-based PDE solvers," in *Proceedings of the 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC2011)*, Seattle, WA (2011).

[4] Ferenbaugh, C., "PENNANT: An unstructured mesh mini-app for advanced architecture research," *Concurrency Computat.: Pract. Exper.*, 2014. DOI: 10.1002/cpe.3422. Also available as technical report LA-UR-13-22550.

[5] Ferenbaugh, C., PENNANT mini-app, `https://github.com/losalamos/PENNANT`, version 0.6 (2014).

[6] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A., "Legion: Expressing locality and independence with logical regions," in *Proceedings of the 2012 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC2012)*, Salt Lake City, UT (2012).

[7] Berzins, M., Luitjens, J., Meng, Q., Harman, T., Wight, C. A. and Peterson, J. R., "Uintah: A Scalable Framework for Hazard Analysis," in *Proceedings of the 2010 TeraGrid Conference*, Pittsburgh, PA (2010), pp. 3:1-3:8.

[8] Caramana, E., Burton, D., Shashkov, M., and Whalen, P., "The construction of compatible hydrodynamics algorithms utilizing conservation of total energy," *J. Comput. Phys*, **146**:227–262 (1998).

[9] Caramana, E., and Shashkov, M., "Elimination of artificial grid distortion and hourglass-type motions by means of Lagarangian subzonal masses and pressures," *J. Comput. Phys*, **142**:521–561 (1998).

[10] Wallick, K.B., "Temporary triangular subzoning (TTS)," in *REZONE: A method for automatic rezoning in two-dimensional Lagrangian hydrodynamics problems*, *Technical Report LA-10829-MS*, Los Alamos National Laboratory, Los Alamos, NM 1987.

[11] Campbell, J., and Shashkov, M., "A tensor artificial viscosity using a mimetic finite difference algorithm," *J. Comput. Phys*, **172**:739–765 (2001).