LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-661692

Summary Report for ASC L2 Milestone #4782: Assess Newly Emerging Programming and Memory Models for Advanced Architectures on Integrated Codes

J. R. Neely, R. Hornung, A. Black, P. Robinson

September 30, 2014

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Summary Report for ASC L2 Milestone #4782

Assess Newly Emerging Programming and Memory Models for Advanced Architectures on Integrated Codes

Introduction

This document serves as a detailed companion to the powerpoint slides presented as part of the ASC L2 milestone review for Integrated Codes milestone #4782 titled "*Assess Newly Emerging Programming and Memory Models for Advanced Architectures on Integrated Codes*", due on 9/30/2014, and presented for formal program review on 9/12/2014. The program review committee is represented by Mike Zika (A Program Project Lead for Kull), Brian Pudliner (B Program Project Lead for Ares), Scott Futral (DEG Group Lead in LC), and Mike Glass (Sierra Project Lead at Sandia). This document, along with the presentation materials, and a letter of completion signed by the review committee will act as proof of completion for this milestone.

The definition of the milestone as it appears in the 2014 ASC IP for Integrated Codes under "Next Generation Architectures" is as follows:

> *Developing and understanding techniques to address current and future architectures is critical to the success of IC. Three key areas on which to focus efforts in FY14 have been identified:*
>
> 1) *Fine-grained concurrency and programmability*
> 2) *Effective use of complex network interconnects, and*
> 3) *Efficient use of decreasing memory capacity.*
>
> *Success in these areas will bring benefit to current advanced technology system platforms as well as provide direction for next-generation platforms in existing and future codes.*
>
> *Specifically, LLNL will undertake at least two of the following:*
>
> 1) *Demonstrate the use of the RAJA programming model in one or more applications. Extend existing SIMD model to include abstractions for low-level threading, resilience, and other loop-level constructs*
> 2) *Develop a task-mapping API to allow parallel codes at scale to call a network-aware library of algorithms designed to optimize layout of MPI tasks on complex interconnect networks. Test and work with algorithm developers to improve the performance of an unstructured mesh application using the Sequoia 5D torus.*
> 3) *Develop a common method for sharing large data tables (for example, EOS and nuclear data) across multiple MPI tasks on a node. Evaluate memory contention and other potential overheads, and the potential impact of using non-cache-coherent shared memory for read-only tables. Understand and recommend system software and hardware requirements to influence co-design of advanced architectures.*
>
> ***Completion Criteria**: Publication of a report to document the results of this assessment*

> ***Certification Method**: Program review is conducted and its results are documented. Professional documentation, such as a report or set of viewgraphs with a written summary, is prepared as a record of milestone completion.*

While the milestone only required reporting out on two of the three tasks described above, the team made significant progress across all three areas, the results of which are reported here.

Motivation and Goals

The milestone criteria were defined after LLNL had approximately one year of experience on the Sequoia BlueGene/Q Advanced Technology System (ATS) platform. Up until that time, many of the codes had focused on scaling out to large numbers of MPI tasks (> 1million) and were shifting focus to address other architecture features of the platform that were challenging them, while also beginning to get glimpse of what the next several ATS platform acquisitions might be in the Trinity and CORAL timeframes.

In all cases, the focus was on

a) bringing researched concepts into a production code environment, and extending research beyond simple proxy applications, and
b) building general solutions that could be utilized by multiple ASC codes

Task 1 – RAJA in production codes

The importance of using hardware threads on the BG/Q processors to get good per-core performance, and the lack of effective SIMD instruction generation by the compilers led to the necessary approach of using fine-grained threading (namely OpenMP) in the codes where parallelism was most available – within domains in loops over elements, nodes, faces, and other mesh-length field arrays. At the end of FY2013, RAJA was emerging as a promising approach to deal with both of these issues in a way that would also "future proof" our codes against alternate programming models (e.g. GPUs), but RAJA had as of yet only been tested in several proxy applications.

The primary goal of the first task in this milestone was to evaluate the usefulness of RAJA in one or more production applications, with a secondary goal of continuing to develop the RAJA model as additional requirements emerged.

Task 2 – Topology mapping and Chizu development

Likewise, LDRD research in topology mapping was demonstrating value for several structured codes (primarily through the use of the Rubik tool), and several algorithms existed that showed potential promise for unstructured applications whose communication could be represented as a general graph. Because task mapping assigns MPI tasks to physical processors on the interconnect at simulation

launch time and thus requires virtually no changes to application source code or binaries, this is a low-overhead method for codes to optimize performance at scale.

The primary goal of this second task in the milestone was to incorporate a number of diverse algorithms into a single API and standalone tool called *Chizu* that will abstract many of the details underlying topology mapping, allowing new hardware interconnects and new mapping algorithms to be added in a reusable library without detailed knowledge required by the code developer.

Task 3 – Support for shared memory in MPI

Finally, the reduced amount of memory capacity on Sequoia (1Gb per core) is straining the limits of codes to strong scale. In particular, data tables used by large multiphysics codes in support of Equation of State (EOS) and nuclear cross-section tables are taking a significant portion of memory per task. The combination of reduced memory and ever-growing size of data tables put additional strain on the ability for codes to utilize MPI in ways that were most computationally efficient. While the concepts of using shared memory space across multiple MPI tasks that shared a common memory space (e.g. a node) were known, solutions were ad-hoc, often platform specific, and could require significant changes to the code.

The primary goal of this third task was to develop a standardized way in which codes could utilize shared memory, evaluate any performance impacts (e.g. degradation due to memory contention, or improvements due to increased L2 cache usage), and provide a platform for future exploration of increasingly complex memory hierarchies on-node, such as in-package memory or NVRAM.

Summary of Results and Findings

Detailed results are presented in Powerpoint format as part of the completion criteria for this milestone, with an overview description provided here.

RAJA

Overview

RAJA has been under development for a little over two years, and was initially conceived as a way to unify loop structures involving regular unit-stride and indirection-based data access, and to abstract parallel programming model constructs in multiphysics codes. The concept of a “hybrid index set” formed the core ideas, which allowed for unstructured codes based on index sets (or indirection arrays) to take advantage of compiler optimizations in regular unit-stride portions, such as SIMD vectorization that historically could not be done without expensive gather/scatter operations. Achieving that goal promoted the separation of the loop body from the iteration, which in turn opened up a series of additional concepts which could be abstracted – including, but not necessarily limited to:

- SIMD instruction-level parallelism
- Loop level threading and GPU device dispatch
- Asynchronous task-level parallelism and lock-free array ordering
- Fine-grained rollback for resilience
- Exploration of data permutations to increase locality and/or reduce synchronization dependencies

RAJA is integrated into a code by first converting loops into the RAJA format (based on C++ templates and lambda functions), and hiding the implementation of platform or problem-specific details through parameterization of loops via iteration methods defined in a core set of header files.

As noted in the introduction, the focus of this L2 was on evaluation of RAJA in a production code environment. Hydrodynamics packages from Ares and Kull were evaluated as part of this effort. More advanced RAJA concepts were explored in LULESH, a proxy-app derived from the Lagrange hydrodynamics algorithm in ALE3D. It was in the implementation of RAJA within Ares that demonstrated that RAJA was more of a parallel programming idiom, than it was a general library. While the RAJA distribution (available at https://rzlc.llnl.gov/stash/projects/RAJA) is intended to provide a set of headers for codes to start with, ultimately is expected that code-specific function templates will be developed, as was done by the Ares developer who implemented the work in this milestone. Custom templates were also developed for Kull to enable integration with Kull C++ iterators.

The Ares code uses a block-structured mesh, allowing for `(i,j,k)` nested loops over logically structured blocks, and reduced and enhanced connectivity points connecting blocks. The result is that some coding is written with nested loops, while other code is written using indirection arrays to handle material regions, etc.. RAJA-style loops were able to unify those loop structures using code-specific and natural notation (e.g. `forEachRealZone`, `forEachZoneWithRegion`, etc.). Beneath this thin Ares-specific loop API layer, RAJA iteration templates are invoked.

Performance-wise, the introduction of RAJA for nominal runs of Ares and Kull on BG/Q of 1 MPI task/core (e.g. 1Gb per task) led to a 50% speedup by introducing 4-way OpenMP threading on the inner loops. On Kull, the performance impact was studied on a loop-by-loop basis, and showed that some loops saw no benefit, while others saw close to perfect 4x speedup. The speedup is due to the use of the hardware threads, and since running multiple MPI tasks per core is generally not possible due to memory capacity limitations, these gains are a pure performance win over the status quo. And while similar performance gains could be obtained from putting straight OpenMP directives into the code, by using RAJA – we're ensuring that we have a platform for further improvements and/or programming model options without application code disruption.

On the flip side, the addition of threading on the Intel-based TLCC2 machines showed no advantage. This is not unexpected, because fine-grained threading on standard multi-core CPUs with out-of-order instructions do not see significant benefit . More importantly, the current OpenMP runtimes have too much overhead to support fine-grained inner loop threading. The RAJA model allows us to easily specify at compile time how those inner loops should be executed, and future work will be to standardize how loops are parameterized in such a way to provide sufficient flexibility to run optimally on different architectures.

A detailed report on the RAJA portion of this L2 milestone is available as LLNL report `LLNL-TR-661403` (attached as part of this L2 completion packet) which will also serve as a comprehensive guide to the RAJA programming model and use cases.

Key takeaways from RAJA study

1) RAJA has been demonstrated to be sufficiently flexible to address the needs of LLNL production codes. As a result, we recommend continuing to develop RAJA (see caveat in #3 below), and more importantly – continue the migration of our production codes to use it.
2) While many programming models (e.g. directives) can "clutter" code, RAJA can enhance readability and maintainability. In the production codes studied in this milestone, the introduction of RAJA helped enhance readability by providing a unified iteration mechanism. In some cases, conditional statements on mask arrays inside of loops can be eliminated by using index sets, which in turn can assist in compiler optimizations and readability.
3) As part of this milestone, continued work evaluating RAJA in the context of the LCALS (Livermore Compiler Analysis Loop Suite, or "Livermore Loops") demonstrated that not all compilers were ready to accept the lambda notation preferred by RAJA. Functors can be used as a well-supported alternative, but adds significant clutter to the code, and is deemed as not a viable path forward for a production environment. In addition, certain optimizations that compilers make in standard normal form C-style loops were lost with the introduction of the lambda notation. Discussions with several key vendor compiler teams has led to some resolution of this deficiency, but as of the time of this milestone – there were a few cases (namely the Intel v14 compiler) where the RAJA lambda notation caused noticeable slowdown relative to the C-style for loops. The GNU C++ compiler generally had the best support for lambdas with optimizations.
4) The use of RAJA will allow for easy exploration of new programming models and data layouts. For example, switching between OpenMP, CilkPlus, and CUDA is simply a matter of recompiling, and different data layouts can be easily tried to reduce data motion and increase locality without any changes to the way loops are written in a code.

Future work

1) The work in converting production codes done thus far was performed primarily by hand. To ease the tedium of converting codes to RAJA with thousands of target loops, we have begun to explore ways in which the ROSE compiler can assist with the one-time conversion of loops to the RAJA format, which should significant ease the burden of this tedious translation while doing so with static analysis that can help avoid potential bugs introduced by human error.
2) Expand the exploration of RAJA to include other architectures, such as GPU and Intel MIC, and other parallel programming models, such as CUDA. This work is dependent upon the availability of compiler support.
3) Compiler feature and optimization issues continue to be actively worked with the vendors through co-design, open discussions with the open source community, and the emerging CORAL "centers of excellence".
4) Continue migrating codes to RAJA as compiler support becomes available.
5) Continue researching RAJA in the context of next-gen code toolkit infrastructure.

Topology Mapping and Chizu

Overview

Prior to, and during this milestone, a series of research papers had been released by LLNL staff (Bhatele, et al) demonstrating the value of topology mapping, primarily on structured meshes using the Rubik tool as an optimization engine. The ASQ working groups had identified topology mapping as a potential area for collaboration across multiple code teams, and conceived of the idea of developing a library and standalone code called Chizu to simplify the introduction of mapping into our current code base.

The Chizu library supports several different mapping options, which were implemented under, or integrated into, the Chizu framework and evaluated as part of this milestone task:

- Zoltan – a mapper available as part of the Trilinos library developed at Sandia
- Scotch – a standalone partitioner and mapper library
- An internal mapper that utilizes plug-and-playable third party partitioners such as those available in Scotch or Metis, and can be optimized by bandwidth or weighted-hops.
- RR – a Round Robin mapping
- Random - a Random mapping

The current capabilities of these mappers were initially tested using the AMG2013 mini-app. A study was performed of all of the above mappers at scales ranging up to 132k tasks, using the metric of max and average hops (ie. "manhattan distance" on the 5D torus). This data was then correlated to runtime.

The first big surprise of this study was that the correlation between hop count and runtime was the opposite of what was expected. The naïve round-robin and random mappings performed better than the default mapping, which in turn performed better than the mappings that optimized for hop count. Upon deeper inspection, it was hypothesized that our codes were not limited by bandwidth, but instead by message injection rates. The default algorithm for queuing outgoing messages on the BG/Q network interface is to place messages in one of 10 available queues based on a simple "hop-count modulo 10" algorithm. Thus, by optimizing on this metric, we are disproportionately favoring a subset of the injection queues, causing contention that is relieved by the more naïve random or round-robin mappings. Additional data on detailed hop counts is necessary to prove this theory, but the LDRD team is currently working with IBM on developing an alternate queuing mechanism that will better distribute the message injection queues. Early results with this mechanism have improved ping-pong networking benchmarks by a factor of 5.

The second surprise of this study was that topology mapping can break assumptions about the layout of tasks to I/O nodes which in turns can severely degrade performance of file writes for non-collective I/O – the default for all of our major codes. It is expected that if Chizu were used in an "inline" mode where domains were renumbered instead of physically remapping tasks to processors via the SLURM srun command, that this issue would go away. Using Chizu as the "inline" option requires more significant changes to the source codes, which we did not have time to fully implement in either ALE3D or Kull.

Finally, the third big surprise of this study was that mappings that favored node-local communication (e.g. communication between MPI tasks on the same node) were much slower than off-node communication. In particular, MPI_Wait times were markedly slower. Several environment variables were identified (setting PAMID_ASYNC_PROGRESS and PAMID_CONTEXT_POST to 1) which activates MPI helper threads on the node, which in turn offloads operations related to both one-sided MPI as well as nonblocking point-to-point messages to the 17th core on the BG/Q node. This simple change led to a 30% speedup in the radiation transport package available in Kull.

In summary, this effort uncovered the fact that our intuition about topology mapping and the behavior of interconnects was dramatically undercut by the complexities of modern interconnect technology. While the results were not what were expected, this research uncovered a wealth of previously undiscovered knowledge about how to best approach mapping, and shined a light on a number of issues that otherwise would have gone unresolved.

Key Takeaways from Topology Mapping study

1) Topology mapping, especially in the context of unstructured codes, is not a completely solved problem. The intuitive metric of reducing "hop count" by placing communicating domains preferentially physically closer on the network can actually harm performance. While topology mapping is one valuable approach to optimizing use of interconnects (esp. the Sequoia 5D torus), an understanding of environment variables, message injection algorithms, and impact of offloaded functionality on the network interface (NIC) are all important as well, and can potentially have even greater performance benefits.
2) The 5D torus on Sequoia is a very high performing interconnect that is hard to optimize for with codes that don't stress bandwidth and/or injection rates. As a result, the use of Chizu and a variety of mappers often demonstrated results that were non-intuitive, and it was difficult to improve performance over the default configurations
3) I/O must also be considered fully when performing topology mapping. In particular, the ALE3D model of "multiple independent files" (sometimes referred to as "poor mans parallel I/O") can be severely degraded when a mapping is defined through SLURM, versus an inline renumbering of domains. This is because there is an underlying assumption that contiguously numbered ranks all shared a common I/O service node, and when this assumption is broken – I/O performance suffers likewise.
4) The naïve metric of reducing hop count can have a negative correlation with performance. For example, after optimizing on hop count (e.g. network distance between arbitrary sets of communicating nodes in nearest neighbor calculations), it was discovered that the underlying system software for queueing outgoing messages was based on a naïve algorithm that filled 10 available queues based on hop count modulo 10. This resulted in contention at the network interface, which degraded performance. A fix has been proposed and is being tested on Sequoia.
5) Likewise, the intuition that communicating ranks should optimally be on the same node is incorrect, this strategy can in fact degrade performance. This is likely due to the fact that messages between ranks on the same node are not offloaded to a network co-processor, and can steal work from computation on the cores. A pair of non-default environment variables, *PAMID_ASYNC_PROGRESS and PAMID_CONTEXT_POST,* were discovered which can direct the 17th core on the node to handle these on-node communications, and resulted in a dramatic performance improvement when on-node communication was preferred.
6) We expect topology mapping to potentially become a bigger issue as hierarchical interconnects (e.g. dragonfly) emerge as standards. This effort puts us in a good position to evaluate emerging interconnects (e.g. StormLake), and study optimal interconnects in the context of our production codes.
7) Chizu will continue to be developed for current and next-gen codes. Additional research required to address complexities of unstructured nearest-neighbor and sweep algorithms.

8) Additional metrics beyond hop count will be developed and tested on our ASC production codes. This work will likely be done in conjunction with LDRD, using our codes to help evaluate new mapping algorithms developed there.

Shared Memory between MPI tasks on node

Overview

It has been well understood for many years that data tables used for rapid interpolation and retrieval of EOS and nuclear cross-section data were taking up significant amounts of memory in our multiphysics codes. While the exact amount is problem dependent, it is not uncommon for these tables to occupy 25-50% of available heap memory in a task. In the past, when memory-per-core was in the 2-4GB range, this was not an issue worth confronting. But the 1GB/core on Sequoia means that our standard model of using one MPI-task per core is putting considerable strain on the memory capacity of these codes.

Because these tables have the properties that a) they are the same for all MPI tasks, and b) they are read-only once initialized, sharing of these tables across MPI tasks that reside on the same node could provide significant memory savings. In the simplest back-of-the-envelope calculations, the effective space required per task would by reduced by the inverse of the number of tasks sharing the table. However, the MPI semantics are such that no data is assumed to be shared, and as such – portable ways to implement this were heretofore not available, or at best, ad hoc solutions.

Developing this library required several issues be addressed:

1) We must be able to identify all of the tasks that reside in a common memory space, and in turn – assign one of those tasks as the "owner" of setting up the tables in a shared memory segment.
2) We must take into account that the virtual address to the shared memory segment (e.g. via an `mmap()` call) may be different for each task on a node, necessitating the concept of "offset pointers" that perform the base address offset at each lookup.
3) The handling of shared memory segments can be different on different platforms. While most Unix© based OSes have a standard set of calls, the details can differ – necessitating an abstraction layer for optimal productivity.

For this milestone report, we are reporting on one of two solutions that were attempted. In this case, the LEOS library and it's underlying support library LIP were used to store bulk data arrays associated with Equation of State lookup tables in shared memory.

A C++ library (tentatively called `ShmemASC`) was developed to abstract away the details of #1 above, and implement multiple concrete subclasses behind a common high level interface to explore multiple platform-specific implementations to interact with shared memory. The main focus was using an initial implementation based on the Boost Interprocess library, a set of C++ classes , to manage the shared memory segments.

A new version of the LEOS/LIP library was updated to use this new ShmemASC library, and required very few changes. At its core, it simply required the use of an "alternate malloc" (already hidden behind macros) to specify which bulk data was to be stored in the shared memory segment. Thus on each process, the normal C style pointers for the bulk data were set to point to locations in the shared memory block on its node by the "alternate malloc".

A series of tests were then performed using one of our production codes to verify results, confirm memory savings, and evaluate any performance impacts. In short, the results were as expected – significant memory savings on Sequoia and TLCC2 nodes, with no measurable performance impacts.

Shared memory through type modifiers

An alternate method was also developed that used the shared type qualifier to denote pointers that will live in the shared memory segment. This work was based on 20-year old technology originally developed in Eugene Brooks' "massively parallel project" in the early 1990's, but updated the preprocessor support to use the ROSE tool. This method was implemented in the MCAPM library and has demonstrated positive results, but will not be reported in detail for this milestone. But a set of slides and documentation is available upon request. We should note that this alternate method was developed because the preferred method developed for the LEOS/LIP library was deemed too intrusive for existing legacy codes, but was a solution appropriate for future development. This second approach will be supported in MCAPM going forward, and is available for legacy codes that don't want to require C++ support, or are otherwise too onerous to modify to use this method.

Key Takeaways from Shared Memory study

1) A reusable library now exists that provides an easy path for C++ (and C/Fortran) codes to effectively use shared memory on multiple platforms. Allocations into shared memory are specified with an "alternate `malloc()`" and can be easily hidden behind standard macros, allowing the programmer to direct data that should live in shared memory.
2) As expected, memory savings are significant when shared memory is used in an application that uses multiple MPI tasks on a single node. This in turn frees up significant resources for application codes, and in some cases should prevent the need for either idling cores, or implementing thread models. In cases where codes do implement threading, the additional memory savings can have a side-

benefit of allowing larger domains, which in turn can amortize the cost of thread overheads.

3) Performance impacts (e.g. from memory hotspots) are minimal to non-existent on both TLCC2 and Sequoia. We expected to see some performance degradation due to memory "hot spots" on Sequoia, and NUMA effects across sockets on TLCC2. Conversely, more effective use of shared L2 cache may offset any negative performance. This issue needs to be studied further to better understand the performance. But as was demonstrated in the accompanying presentation, the performance impacts were negligible for EOS lookups.
4) Some legacy libraries are not amenable to this approach of modifying existing code, due to the face that memory allocation is decentralized. An alternate approach was implemented for the MCAPM nuclear cross-section library which used a type-qualifier based approach with a compiler pre-processor to automatically add offset pointers at each reference. While this approach was not reported in detail for this milestone, the ROSE compiler was modified to allow it to parse the type qualifier statements, and with additional funding, will find its way into production use as needed.

Future work

1) Investigate the incorporation of the MPISM library developed as part of the MCAPM effort into ShmemASC. This will allow for a common solution to identifying groups of MPI tasks that live on the same node and thus can shared memory
2) Complex memory hierarchies are going to be a fact of life in future advanced architectures. This library could expand to support multi-level memory management in future architectures, including placing memory in a shared memory segment of scarce on-package memory, placing large data tables in NVRAM (and allowing system software to cache in DRAM), and allocating memory in device memory (e.g. GPUs). It is our hope that as these architectures come available, we can utilize this library to expand into these critically important research areas.

Conclusions

All three of these efforts have significantly advanced our ability to more effectively use Sequoia, and have generated initial implementations of software products that will be widely applicable across multiple code efforts. While the results were sometimes surprising, the lessons learned were universally valuable, and we expect all of these products to make their way into production use, and in turn provide a smoother path to addressing similar issues on future ATS and CTS platforms.