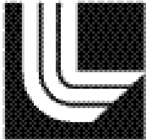
LLNL-TR-515971



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

SURVEY OF NOVEL PROGRAMMING MODELS FOR PARALLELIZING APPLICATIONS AT EXASCALE

*Rich Cook, Evi Dube, Ian Lee, Lee Nau,
Charles Shereda, and Felix Wang*

November 17, 2011

Disclaimer
This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

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

Table of Contents

1. EXECUTIVE SUMMARY 5
2. INTRODUCTION: OVERVIEW OF THIS STUDY AND ITS OBJECTIVES 6
2.1. RESOURCES 7
3. PARALLEL PROGRAMMING MODELS OVERVIEW 8
3.1. CHAPEL (CASCADE HIGH-PRODUCTIVITY LANGUAGE) 10
3.1.1. Overview 10
3.1.2. Present State of the Model 11
3.1.2.1. Robustness and Known Issues 12
3.1.3. Tool Availability 13
3.1.4. Performance 13
3.1.5. Suitability to LLNL Application Codes 14
3.1.6. Resources and Additional Information 14
3.2. X10 PROGRAMMING LANGUAGE 16
3.2.1. Overview 16
3.2.2. Present State of the Model 17
3.2.3. Tool Availability 18
3.2.3.1. Debugger 18
3.2.3.2. Performance analysis tools 18
3.2.3.3. Other Tools 18
3.2.4. Performance 18
3.2.5. Suitability to LLNL Application Codes 19
3.2.6. Bibliography 20
3.3. FORTRESS 21
3.3.1. Overview 21
3.3.2. Present State of the Model 22
3.3.2.1. Robustness and Known Issues 22
3.3.3. Tool Availability 22
3.3.4. Performance 22
3.3.5. Suitability to LLNL Application Codes 23
3.3.6. Resources and Additional Information 23
3.3.7. Bibliography 23
3.4. CILK PLUS LANGUAGE EXTENSIONS 24
3.4.1. Overview 24
3.4.2. Present State of the Model 25
3.4.2.1. Robustness and Known Issues 26
3.4.3. Tool Availability 26
3.4.4. Performance 26
3.4.5. Suitability to LLNL Application Codes 27
3.4.6. Resources and Additional Information 27
3.5. INTEL PARALLEL BUILDING BLOCKS: ARBB & TBB 29
3.5.1. Overview: TBB and ArBB Common Characteristics 29
3.5.1.1. Threading Building Blocks 29
3.5.1.2. Array Building Blocks 31

3.5.2. Present State of Model ... *32*
3.5.2.1. Robustness and Known Issues ... 32
3.5.3. Tool Availability ... *32*
3.5.4. Performance ... *32*
3.5.5. Suitability to LLNL Application Codes ... *33*
3.5.6. Resources and Additional Information ... *33*
3.5.7. Bibliography ... *33*
3.6. UPC ... 35
3.6.1. Overview ... *35*
3.6.2. Present State of the Model ... *36*
3.6.2.1. Robustness and Known Issues ... 36
3.6.3. Tool Availability ... *36*
3.6.4. Performance ... *37*
3.6.5. Suitability to LLNL Application Codes ... *40*
3.6.6. Resources and Additional Information ... *40*
3.7. AMPI & CHARM++ ... 42
3.7.1. Overview ... *42*
3.7.2. Present State of Model ... *43*
3.7.2.1. Robustness and Known Issues ... 45
3.7.3. Tool Availability ... *45*
3.7.4. Performance ... *46*
3.7.5. Suitability to LLNL Application Codes ... *46*
3.7.6. Bibliography ... *47*
3.8. OPENCL ... 48
3.8.1. Overview ... *48*
3.8.2. Present State of Model ... *48*
3.8.3. Robustness and Known Issues ... *49*
3.8.4. Tool Availability ... *49*
3.8.5. Performance ... *49*
3.8.6. Suitability to LLNL Application Codes ... *49*
3.8.7. Resources and Additional Information ... *50*
3.9. CUDA ... 52
3.9.1. Overview ... *52*
3.9.2. Present State of Model ... *53*
3.9.3. Robustness and Known Issues ... *53*
3.9.4. Tool Availability ... *53*
3.9.5. Performance ... *53*
3.9.6. Suitability to LLNL Application Codes ... *54*
3.9.7. Resources and Additional Information ... *54*

4. RECOMMENDATIONS ... **56**

APPENDIX A. ADDITIONAL EXASCALE PROGRAMMING MODELS ... **58**

1. Executive Summary

A range of programming models (and the languages that constitute the practical implementation of these models) exist to address parallel programming challenges. At present, however, one model dominates LLNL application codes: SPMD message-passing using MPI for internode communication, and occasionally, OpenMP for intranode parallelism.

In this survey, we characterize the following novel programming models that may be of use to LLNL teams as supercomputing moves toward exascale.

- Chapel
- X10
- Fortress
- Cilk Plus
- Intel Parallel Building Blocks
- UPC
- AMPI and Charm++
- OpenCL
- CUDA

Some of these models make better use of intra-node resources than the present model, or are targeted at GPUs. Others present an entirely new way of addressing large-scale simulation problems, or one that promises more compact and comprehensible code. None are as robust or stable as the present model. For each model, we provide an overview and discuss the model's present state, robustness, and suitability to LLNL application codes. Several models are not discussed in detail and are mentioned in the appendix.

Our recommendations for action consist of the following:

- We recommend a further study of Chapel – specifically, an application port.
- We recommend devoting resources to monitoring X10 and Intel PBB.
- We suggest maintaining an Intel PBB implementation.
- We recommend that MPI support staff at LLNL familiarize themselves with AMPI and Charm++ to see if some of its innovations can be applied to issues such as fault tolerance at large scale.
- We recommend maintaining in-house expertise in OpenCL and CUDA but caution against developing a significant codebase, especially in CUDA, which is proprietary.

2. Introduction: Overview of This Study and Its Objectives

We conducted this study in response to the need of application teams to determine how to make effective use of the future exascale supercomputing environment at Livermore. The exascale generation of supercomputers will have massive on-node parallelism, much more than current hardware. Making use of this on-node parallelism will certainly require code modification and creative approaches, and may necessitate the adoption of new language paradigms. We investigated some of these new paradigms to determine:

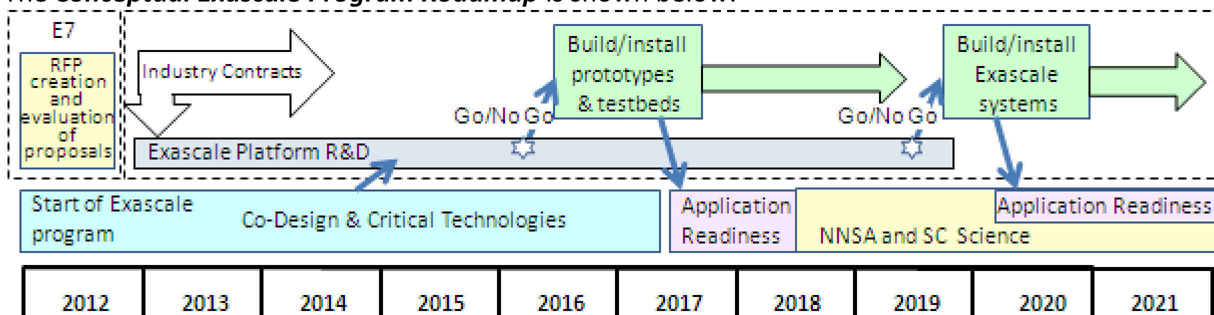
a) The ease in learning and adopting these languages.
b) The specific benefits to switching to the new language paradigm. For instance, the language may be far more comprehensible than current models, and mask explicit parallelism to greatly reduce the amount of code required for the same amount of work. Or, it may allow programmers to make use of a form of parallelism not available through other paradigms, such as CUDA or OpenCL for GPUs.
c) The robustness of the model.
d) The potential of this model to meet programming needs in the future, regardless of its present state.

Background:
It is widely expected that the computer systems anticipated in the 2015 – 2020 timeframe will be qualitatively different from current and past computer systems. They will be built using massive multi-core processors with 100's of cores per chip, their performance will be driven by parallelism, constrained by energy, and with all of their parts, they will be subject to frequent faults and failures. (Trivelpiece, 2010)

Coined *Exascale Computing*, the changes will occur in many areas. There will be multiple memory types, including programmable (scratchpad) memory along with generally more heterogeneous/hierarchical systems than today, and the memory:FLOPS ratio is expected to worsen. (Dubrow, 2011)

For *Exascale Computing*, the main programming environment challenges are expected to be within the new node rather than across nodes, since that is where the biggest changes appear to be headed. The total number of nodes is not changing dramatically, so current practices of MPI between nodes to this scale provides one option of utilizing the exascale systems. Another option is to utilize unified programming models at the global level (Chapel, X10, UPC, Co-array Fortran, etc.).

The ***Conceptual Exascale Program Roadmap*** is shown below:



(Duke, 2011)

Objective:
Our objective in preparing this survey was to determine and document the characteristics of the most promising next-generation parallel programming models that may have applicability at exascale. Targeted at development teams and support staff, it provides a basic overview of each model – enough for a developer to determine if further investigation is warranted for their development effort. Section references and bibliographies can be used as pointers for conducting further work.

2.1. Resources

1. http://science.energy.gov/ascr/news-and-resources/workshops-and-conferences/grand-challenges/
2. *The International Exascale Software Project Roadmap*,; http://www.exascale.org/mediawiki/images/2/20/IESP-roadmap.pdf
3. Chamberlain, Brad *(*February 2010) *Programming Models (and Programming environments) at the Exascale*; http://chapel.cray.com/presentations/Chamberlain-ProgEnv-CrossCut.pdf
4. Dubrow, Aaron (October 5, 2011) *IEEE Conference Keynoters Lay Out Path to Exascale Computing* ; HPC in the Cloud, HPC Wire; http://www.hpcinthecloud.com/hpcwire/2011-10-05/ieee_conference_keynoters_lay_out_path_to_exascale_computing.html
5. Duke, Karl (July 2011) *Exascale RFI*; https://www.fbo.gov/download/8d0/8d04ef8edaca4638063a4a0b7afe9a1d/ExascaleRFI.pdf
6. Stevens, Rick etal. (March 2011) *A DOE Laboratory plan for providing Exascale applications and technologies for critical DOE mission needs*;; http://computing.ornl.gov/workshops/scidac2010/presentations/r_stevens.pdf
7. Trivelpiece, Alvin *(*January 19-20, 2010) *Exascale Workshop Panel Meeting Report*; http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Trivel_piece_exascale_workshop.pdf
8. Yelick, Katherine ; *Programming Models for Exascale*; https://hpcrd.lbl.gov/E3SGS/Yelick-Pmodels.pdf

3. Parallel Programming Models Overview

In the course of preparing this survey we discovered that we were using the term *parallel programming model* to refer to different concepts. The term can be defined in several different ways. For example:

a) Informally, a "parallel programming model" can be thought of as a loose, generic term encompassing languages, language extensions, libraries, and other 'things' that are used to accomplish a particular programming objective that requires parallelism. Each unique system such as Chapel or Cilk Plus or MPI or OpenMP, regardless of whether it is a language or a library or language extension, constitutes a programming model in this definition.

b) One might more formally define "parallel programming model" as being a particular combination of data visibility (local, global), control strategy (local, SPMD, etc), and programming capabilities that describe a method for solving a computational problem, specifically one that requires computational parallelism to solve. An example of this definition of programming model is "Asynchronous Partitioned Global Address Space (PGAS)."

In this survey, we use "programming model" primarily to refer to the systems that are practical implementations of the underlying models. This corresponds with definition (a).

The following chart shows the models that we examined for this survey and connects definition (a) with definition (b) for each of these models.

System (a)	Programming Model (b)	Data Model	Control Model
Chapel	Partitioned Global Address Space (PGAS)	Global memory view	Global view
X10	Asynchronous PGAS	Global memory view	Global view
Fortress	PGAS	Global memory view	Global view
Cilk Plus	Multithreaded	Global memory view (single node only)	Global view (single node)
Intel Parallel Building Blocks	Multithreaded	Global memory view (single node only)	Global view (single node)
UPC	PGAS	Global memory view	Global view
Charm++	Object-oriented	Local memory view	?
AMPI	Message passing	Local memory view	Local view
OpenCL	GPU language	GPU memory view (data is transferred to and from GPU memory)	Global view (single node)
CUDA	GPU language	GPU memory view (data is transferred to and from GPU memory)	Global view (single node)

Three of these languages, X10, Chapel, and Fortress, began their lives through the DARPA-funded HPCS effort. X10 is from IBM, Chapel from Cray, and Fortress from Sun (now Oracle). Phase II of the project lasted from 2003 to 2006 and funded all three projects; Phase III began in July 2006; it funded work on X10 and Chapel and dropped funding for Fortress, although Sun continued work on the project for some time in the absence of DARPA money. The original goal of the DARPA effort was to increase programmer

productivity by a factor of 10 by the year 2010. Given that it is now late 2011 and none of these three languages are quite ready to be used for real production codes, that original goal was apparently overly aggressive and optimistic.

The next two languages, Cilk Plus and Intel Parallel Building Blocks, are Intel-driven languages. UPC is a PGAS language developed at LBNL, while Charm++/AMPI is a UIUC project that advances the message passing model. The last two languages, OpenCL and CUDA, allow for programming Graphical Processor Units.

Listed below are additional parallel programming models, presented for comparison purposes. OpenMP, Coarray Fortran, Titanium, and Global Arrays are discussed briefly in Appendix A.

System (a)	Programming Model (b)	Data Model	Control Model
MPI	Message passing	Local memory view	Local (for library – global control resides elsewhere)
OpenMP	Multithreaded	Global memory view (single node only)	Global view (single node)
Coarray Fortran	PGAS	Local memory view	SPMD
Titanium	PGAS	Local memory view	SPMD
Global Arrays	PGAS	Global memory view	SPMD

Descriptions of the individual languages appear in the following subsections.

3.1. Chapel (Cascade High-Productivity Language)

Owner / Development Location	Cray Inc. (head of team is based in Seattle, WA)
Project Website	http://chapel.cray.com/index.html
Download Page	http://chapel.cray.com/download.html
Platforms Available	Most UNIX-based systems, Mac OS X, Windows. Works in conjunction with the GASNet library which works with various interconnects.

3.1.1. Overview

Chapel (Cascade High-Productivity Language) is a general-purpose parallel programming language being developed by Cray, Inc. under the DARPA High Productivity Computing Systems (HPCS) program. Chapel was developed from first principles, and the principles guiding the design are:

- general parallel programming,
- locality-aware programming,
- object-oriented programming, and
- generic programming.

As per the language specification documentation, the first two principles were motivated by a desire to support general, performance-oriented parallel programming through high-level abstractions. The second two principles were motivated by a desire to narrow the gulf between high-performance parallel programming languages and mainstream programming and scripting languages.

Taken from its name, Chapel's purpose is to increase programmer productivity while enhancing code robustness. The Chapel development team has four stated productivity goals (Chamberlain B. , Chapel tutorial, 2011)

- *Programmability*. All aspects of programming (reading, modifying, maintaining, etc.) should be dramatically improved over current programming models.
- *Performance*. Performance should be competitive with MPI-parallel codes, and better on advanced architectures.
- *Portability*. As universal as MPI and more portable than OpenMP.
- *Robustness*. Semantics and better abstractions eliminate common coding errors.

Several aspects of the language exist to achieve these productivity goals. Based on the Partitioned Global Address Space (PGAS) model, Chapel provides a global view for expressing both data structures and the control flow of the program. Global-view data structures allow programmers to express data aggregates globally even though their implementations may distribute them across the locales of a parallel system. The programmer therefore codes as if a single process is running across many processors, rather than coding for a many-process model. Work is shared across threads. This provides for a more general expression of parallelism than more common contemporary parallel languages that use the Single Program Multiple Data (SPMD) approach. Chapel also operates under a multiresolution philosophy, meaning that programmers can initially write very abstract code and subsequently add more detail to tune for their target architecture. Object-oriented design, type inference, and other features allow for rapid prototyping and code reuse (Chamberlain B. , Chapel presentations, 2009)

Chapel has several general and parallel language constructs that are meant to reduce the amount of code necessary to express a concept or perform work. Among many others, these include constructs to

deal with the distributed nature of a global-view data structure, concurrency constructs, and data and task parallelism constructs.

Domains in Chapel are a first-class representation of an index set. They provide the programmer with a way to express operations that more closely mimics the original problem without much of the overhead that comes from checking boundaries or complex indexing. A domain's indices may also be distributed across multiple locales on a platform, thus supporting global-view data structures. Additionally, domains may be modified in size fairly easily and arrays associated with those domains will be reallocated accordingly. This is handled by the runtime through the use of domain maps and it provides a good deal of flexibility to potential algorithms without requiring otherwise extraneous portions of code on the part of the programmer. Domains come in three types, base domains, subdomains, and sparse subdomains, and may also be classified as being regular (e.g. rectangular) or irregular (e.g. associative).

Locales are described by the Chapel Language Specification as a portion of the target parallel architecture that has processing and storage capabilities (Cray Inc., 2011). In a clustered system, this would be equivalent to a node with multiple processors sharing a memory space. Locales may be referenced within program code, and they provide a way for programmers to reason about and exploit locality.

Data parallelism is generally invoked by the *forall* keyword where iterations of an otherwise serial loop may be calculated independently of other iterations. The number of threads that are used for this all-way parallelism depends on how many cores exist on a processor, but may be changed through configuration variables. Other methods for managing data parallelism include reductions, scans, and shorthand forms in dealing with arrays as a whole.

Task parallelism is invoked using the *begin* and *cobegin* keywords for unstructured task parallelism and the *coforall* keyword for structured task parallelism. In general, there is a distinct thread for each task spawned through a task parallelism construct, and there may be many more threads than there are cores on a processor. In the case of structured task parallelism, each iteration is processed as a separate task from other iterations, and generally, the serial code inside an iteration is more complex than its data parallel counterpart. Because the threads of a task parallel region of code may exceed the number of cores in a processor, the kernel may switch among threads leading to possible issues with concurrency for poorly written codes.

Chapel introduces synchronization variables that provide a way for the programmer to perform more complex types of parallel control beyond the constructs already presented. Synchronization variables possess *full* and *empty* states that methods reading and writing to them can use to prevent data races as well as to determine blocking behavior. Although synchronization variables allow more fine-grained control of the program, the use of too many may be costly both in terms of programming productivity and performance.

3.1.2. Present State of the Model

Chapel is currently an evolving language. Many of the core features of the language have been implemented functionally, however these features are not fully optimized. Additionally, there are several features in the language specification that have yet to be developed. As the language matures, many of these features are expected to be integrated and optimized.

Chapel is available for several platforms with many platform-specific options handled by the Chapel compiler. Inter-node parallelism is based on the PGAS model with communication handled primarily through a Global-Address Space Networking (GASNet) layer. Future plans include additional communication layer options such as Pacific Northwest National Laboratory's (PNNL) Aggregate Remote Memory Copy Interface (ARMCI) library. Intra-node parallelism uses a multithreaded execution model using pthreads as the standard for the threading mechanism. Support for other mechanisms exist, including more lightweight implementations such as Qthreads developed at Sandia National Laboratory (SNL) as well as platform specific options such as MTA when targeting the Cray XMT platform. Support has recently emerged for using GPUs as accelerators in Chapel, in particular using Nvidia GPUs with CUDA. As with the other levels of parallelism, the support at the GPU level is expected to expand, probably with OpenCL as a next target. Several papers can be found describing these options under Section 3.1.6.

Perhaps the main limitation we found of the current state of the Chapel language is the lack of task synchronization within a parallel section of code. Task synchronization would allow the interleaving of serial and parallel code with less overhead than entering and exiting from a structured parallel section. It would also improve ease of programmability, as per the productivity goals. Although cross-task synchronization was achieved using synchronization variables, the resulting mechanism took a great deal of programming effort – more than should be required for such a task. In the literature, there is mention about introducing task *colors* for grouping purposes, and it is likely that this would include constructs for cross-task synchronization. (Chamberlain, Deitz, & Navarro, pgas 11, 2011)

Another current limitation for the Chapel language is the lack of an API to the runtime system. Although the language exposes several parameters that are important to the runtime configuration of a program (e.g. dataParTasksPerLocale), there is no built in way to modify these constants once they are set at execution time. Moreover, there are no methods that expose information or modification of specific threads or cores, meaning that there is no built-in way of exploiting NUMA effects. It is possible that these details of the language will be resolved in a later revision.

3.1.2.1. Robustness and Known Issues

Chapel is designed to provide the programmer more ease in coding than are other languages. This results in generally more readable code and with respect to robustness, a lessened chance for syntactical coding errors leading to more reliable code from the beginning. As mentioned previously, the *domain* construct provides a more natural way of expressing the subtleties of indexing a data structure of an algorithm, and this absolves the programmer of much of the effort required in complex bound checking and corner cases. Some language features have not been fully developed yet, with garbage collecting projected to be a significant addition. The features that are currently functional do not seem to have many issues besides being less than optimal in terms of performance.

We noticed an issue with passing a variable by reference multiple times before modifying it in a child process within a parallel section. Serially, this does not pose an issue, but within a parallel section, such as a *coforall* statement, if the variable is modified as though it were serial (i.e. updated by a single task in the collection), an immediate read in parallel may cause unpredictable results. The same algorithm behaves correctly if the multiply referenced variable is replaced by a global variable, however.

The interoperability with other languages is presently not fully functional, and there are definitely issues with calling Chapel generated code from an external language. Moreover, as Chapel does not support explicit pointers, it will require a great deal of effort to successfully glue Chapel together with languages

such as C. There is currently a project, BRAID, which is being developed at LLNL to interface PGAS based languages to traditional HPC languages. (Prantl, Epperly, Imam, & Sarkar, 2011) The BRAID system for Rewriting Abstract Intermediate Descriptions provides glue code in places where languages such as Chapel make references outside of themselves; BRAID then goes through another, older and very robust project, Babel, which is used to integrate with a multitude of other languages. The project is also targeted to handle the distributed arrays that are present in Chapel, introducing a new Scientific Interface Description Language (SIDL) datatype: *DistributedArray*. The robustness of BRAID for Chapel is good and many data structures and types are supported to communicate with other languages. The report from the Chapel compiler team indicates that a new keyword is in development that would allow Chapel modules to be called from an external language more easily. This development would enable support for creating libraries in Chapel for external use.

3.1.3. Tool Availability

At presently there is no Chapel specific debugger available. However, the Chapel compiler allows the programmer to enable a debugging flag (i.e. *-g*) and compile to the C programming language. The resulting source code may then be stepped through and run through any generic debugger (e.g. gdb) that is able to debug C code. With respect to parallel debugging, it is unclear whether tools exist that work with Chapel, although there are a few compiler flags (i.e. *--blockreport* and *--taskreport*) that may be used on a single locale to debug deadlock conditions. Native support for debugging Chapel that includes language specific knowledge is planned.

Tools for profiling are being added to the priority queue along with performance optimizations and scalability work. There has been success in profiling non-trivial Chapel code using HPCToolKit. The expectation is that as the language grows, more and more of these tools will become available to Chapel. Currently, there are a few Chapel tools for communication diagnostics as well as some memory tracking features to aid the programmer in determining memory leaks. External tools such as HPCToolKit will require more effort on the part of the programmer to work correctly with Chapel.

3.1.4. Performance

Performance is not a defining feature of the Chapel language in its present state. In conversations with the Chapel team, we have learned that there are several low-hanging fruit remaining with respect to Chapel performance that they wish to address. The Chapel team notes that performance and scalability are high priorities for them.

Following a port of the ASC Sequoia benchmark, CLOMP, to the current Chapel version at time of writing (1.3.0), it was discovered that there were speedups in many cases when executing in parallel with Chapel over the serial version of the code. However, for smaller amounts of work per iteration in the parallel loop section, there was greater relative influence of the overheads present in Chapel and slowdowns occurred for the benchmark's "target input" case relative to the serial version. Because Chapel does not yet possess a native barrier call, a manual barrier using synchronization variables had to be implemented for the CLOMP port, and as expected the performance was severely lacking (an order of magnitude) behind the OpenMP barrier call. Moreover, it was discovered that the synchronization constructs of Chapel do not scale with the number of processors as well as expected for a language targeted at multiprocessor platforms. As per our conversations with the Chapel team on the scalability focus, this critical aspect of the language is expected to improve.

Performance measurements with respect to several of the HPC Challenge benchmarks have been performed. (Chamberlain, Choi, Deitz, & Iten, 2009) The most current published results from the

challenge are from 2009 by the Chapel team; their entry for class 2 (most productivity) was awarded "most elegant implementation". Although the language has evolved since then, the performance results show that Chapel was comparable to an MPI version of the code for the Global and EP STREAM Triad benchmarks. Performance on the Random Access benchmark was not competitive with MPI. While a productivity rather than performance measure, the paper also noted that Chapel consistently beat MPI in terms of lines of source code (Chapel had far fewer lines for the same program).

3.1.5. Suitability to LLNL Application Codes

Switching to Chapel would require a dramatic shift in thinking for LLNL application teams coming from an MPI background, distributed memory model, and explicit parallelism framework. Although such a shift may be necessary for the next generation of parallel computing, ideally, this shift would be incremental. This would require that Chapel objects be capable of being directly linked against non-Chapel objects, and against objects that link against MPI. Currently, application teams would be required to rewrite a good portion of their code base due to Chapel's inability to exist as a secondary language. However, work is in progress to create libraries in Chapel, and furthermore, the BRAID project is expected to provide the necessary glue for Chapel to communicate with existing languages.

Chapel provides several constructs that are compatible with the types of codes in use at LLNL. For example, the domain construct supports the creation of adaptive meshes and sparse matrices, and Chapel provides an easy way for programmers to index into the associated data structure. Moreover, the mapping between the global view of these data structures and the locales where they reside is provided through domain maps via the runtime, allowing for less programming overhead than with message passing. Domains may also be distributed across the locales of a clustered system using the *distributions* construct. Standard distributions in Chapel are the block and cyclic layouts, but users may also define their own distributions to better fit their problems. These distributions do not affect codes semantics, but rather are used for implementation and performance effects.

The primary advantages Chapel has over MPI are in programming ease and elegance. Because it is a multiresolution language, it allows for quicker prototyping of algorithms, with architecture specific optimizations added in later. This provides greater code robustness due to a decreased chance for communication errors. Finally, because Chapel is multi-level, it is able to provide homogeneous coding semantics, whereas one may have to add OpenMP to MPI code to fully utilize processor capabilities. With respect to performance, Chapel currently lags behind OpenMP and MPI in general, although it is comparable in some cases and is expected to improve overall as the language progresses.

3.1.6. Resources and Additional Information

The information on Chapel is internet based, and primarily written from the perspective of the Chapel development team or Chapel proponents.

Many resources including several papers, presentations, and tutorials are available through the Chapel homepage: http://chapel.cray.com/

Chapel Language Specifications: http://chapel.cray.com/spec/spec-0.82.pdf

Presentations and videos: http://chapel.cray.com/presentations.html

Tutorials:

- Chapel Overview for Hands-On Session; Discovery 2015; http://chapel.cray.com/tutorials.html
- An Overview of the Chapel Parallel Programming Language; CUG 2011 Tutorial; http://chapel.cray.com/tutorials.html
- Parallel Programming in Chapel: The Cascade High-Productivity Language, SC10 Tutorial; http://chapel.cray.com/tutorials.html

Download: http://chapel.cray.com/download.html

Sourceforge: http://sourceforge.net/projects/chapel/

Programmer Assistance: The chapel-users email archive found on the Chapel Sourceforge page contains email archives that a developer may find useful. A development version of Chapel may also be downloaded, which contains more information on the workings of the Chapel compiler and its features.

http://sourceforge.net/mailarchive/forum.php?forum_name=chapel-users

Additionally, Colorado State has a wiki and quick reference worth perusing.

https://www.cs.colostate.edu/wiki/Chapel_language
https://www.cs.colostate.edu/wiki/mediawiki/images/b/b2/QuickReference.pdf

Wikipedia page: http://en.wikipedia.org/wiki/Chapel_%28programming_language%29

Babel homepage: https://computation.llnl.gov/casc/components/#page=homePapers:

1. Chamberlain, Bradford L.; Choi, Sung-Eun; Deitz, Steven J.; Iten, David (2009) *HPC Challenge Benchmarks in Chapel*; HPC Challenge; http://chapel.cray.com/hpcc/hpcc09.pdf
2. Chamberlain, Brad; Deitz, Steven; Navarro, Angeles (September 2011); *User-Defined Parallel Zippered Iterators in Chapel*; http://pgas11.rice.edu/papers/ChamberlainEtAl-Chapel-Iterators-PGAS11.pdf
3. Epperly, Thomas; Prantl, Adrian; Chamberlain, Bradford L (September 2011) *Composite Parallelism:Creating Interoperability between PGAS Languages, HPCS Languages and Message Passing Libraries*; http://chapel.cray.com/papers/CompParallelismProgress.pdf
4. Prantl, Adrian; Epperly, Thomas; Imam, Shams; Sarkar, Vivek (September 2011) *Interfacing Chapel with traditional HPC programming languages*; PGAS 2011; http://pgas11.rice.edu/papers/PrantlEtAl-Chapel-Interoperability-PGAS11.pdf
5. Sidelnik, Albert; Garzar´an, Mar´ıa J.; David Padua; Chamberlain, Bradford L. (April 2011) *Using the High Productivity Language Chapel to Target GPGPU* Architectures;; https://www.ideals.illinois.edu/bitstream/handle/2142/18874/techreport_chplgpu.pdf?sequence=2
6. Sridharan, S; etal (5/7/2011) *A Scalable Implementation of Language-based software Transactional Memory fpr Transactional Memory for Distributed Memory Systems*; Future Technologies Group Technical Report Series; ORNL; http://ft.ornl.gov/pubs-archive/chplstm1-2011-tr.pdf
7. Weiland, Michèle ; Haddow, Thom (2009) ; *Performance Evaluation of Chapel's Task Parallel Features*; CUG 2009; http://cug.org/5-publications/proceedings_attendee_lists/CUG09CD/S09_Proceedings/pages/authors/01-5Monday/4A-Weiland/weiland-paper.pdf
8. Wheeler, Kyle B. etal (June 2011) *The Chapel Tasking Layer Over Qthreads*; CUG 2011; http://chapel.cray.com/publications/CUG11-wheeler.pdf

3.2. X10 programming language

Owner / Development Location	IBM Research
Project Website	http://x10-lang.org/
Download Page	http://x10-lang.org/software/download-x10/release-list.html
Plaforms Available	AIX/Power, Linux/Power, Cygwin/x86, Linux/x86 and x86_64, MacOS/x86 and x86_64

3.2.1. Overview

X10 is a parallel asynchronous Partitioned Global Address Space language based on Java and developed by IBM. Like Chapel and Fortress, the X10 project started in 2006 as a response to the DARPA HPCS program.

Because X10 is an extension to Java rather than an entirely new language, it is ostensibly easy to learn for Java developers. It uses Java math functions and garbage collection, and, like Java, uses regions and points for array manipulation. X10 incorporates ideas from object-oriented languages such as classes, objects, inheritance, and generic types; like functional languages, it also uses type inference, anonymous functions, closures, and pattern matching (Hudak, 2010).

From Chapel, Fortress, and X10: Novel Languages for HPC:

> *The name X10 is derived from the developers' aim to create, by 2010, a language that is ten times more productive than those languages and libraries that are commonly used for current HPC codes… The design goals for X10 are to create a language which enables the development of safe, scalable, analysable and flexible code. Java was chosen as a basis for X10 because it supports three out of these four goals. It is also a widespread and accepted modern OO language that is accompanied by reliable tools, libraries and documentation* (Weiland, 2007).

X10 does not support Java concurrency, arrays, or built-in types, but adds several key concepts in order to support parallelism: *places* and *activities* (Weiland, 2007).

Places

A place is the X10 term for a unique memory location. X10 is designed so that the number of memory locations is distinct from the number of threads. Places are containers for activities and objects, and their number is fixed at launch. If an application is normally launched with mpirun, the number of places is equivalent to the number of MPI processes launched.

Activities

Activities are PGAS threads in X10. Activities can be dynamically created during program execution, unlike UPC or MPI, which maintain a fixed number of threads or processes for an entire run.

Every X10 program begins with a single activity in Place 0. X10 provides several keywords to control activities:

- The *at* keyword evaluates an expression at a particular place. Using at blocks the parent process until the evaluation completes. At can be used to read or write remote values (similar to MPI_Put and MPI_Get) or to invoke a remote method.
- The *async* statement causes an expression to be spawned in parallel as a child activity and evaluated asynchronously. Control proceeds immediately to the subsequent statement.
- The *finish* statement (finish{ async N; async N+1; …} S) waits to perform statement S until all async statements N, N+1,… have completed. This is useful for identifying dependencies and implying a barrier.
- The *atomic* statement evaluates an expression atomically.

X10 is called an asynchronous PGAS language because of the ability to explicitly control concurrency through constructs such as async and finish. The independence of memory locations (places) and threads (activities) is a necessary precondition for this capability (Saraswat, 2008).

The stated motivation behind X10 is to provide a language that addresses the inherent complexities of the increasingly popular many-core architectures (of which nVidia Fermi and the Intel MIC are examples) in a single, unified programming model. The goals of the project are to create a language that is simple (hence its Java base), safe (from design errors, and through static checking), powerful (capable of expressing typical HPC codes), scalable, and universal (can be used and deployed on a host of architectures) (Hudak, 2010).

Two runtime environments are available for X10: A Java-based one, in which all places are in a single Java Virtual Machine, and a multi-process C++ runtime (Hudak, 2010).

3.2.2. Present State of the Model

We conducted several interviews with developers of the X10 language. According to a former developer (Sayantan Sur), X10 was not, at the time of the interview, robust enough to reliably support the more complex features of the language or to port a meaningful application (Sur, 2011). A current developer (David Grove) stated that there are X10 codes in existence with up to about 10,000 lines. He believes that the present state of the language is that it is 'more than a toy,' meaning that it is beginning to reach beyond the prototype phase. Grove acknowledged that a specific problem exists with sparse non-negative factorization, but said that X10 can perform at scale, and performs well at load balancing irregular applications. Grove noted that the 2.2 release of the software (released May 2011) has many improvements such as a finalized language specification, decent error messages, and a usable IDE. There is an eclipse-based debugger that interacts with gdb. He stated that this release should be a reasonable candidate for experimenting with HPC applications (Grove, 2011).

Sur noted that funding for X10 comes primarily from IBM Business Analytics, which is focused on cloud computing and a loosely coupled parallelism model. He believed that until there is strong funding from the HPC community, tightly coupled parallelism will not be well supported or tested (Sur, 2011). Grove noted that a collaboration with the Australian National University ANU Chem team was providing insight into what needed improvement in the language (Grove, 2011).

As of interview time, Grove estimated the X10 community at somewhere between 50 and 100 active participants (Grove, 2011).

Finally, according to the X10 website, "X10 is in active development, so the language and libraries are still changing, so coding to X10 is still a bit of a moving target. Notable current gaps include library

support, interoperability, IDE and compiler performance, and runtime performance of certain X10 idioms. Of course, we are actively working on all these issues" (IBM).

3.2.3. Tool Availability

3.2.3.1. Debugger

IBM developerWorks provides an X10 debugger called 'The IBM Parallel Debugger for X10 Programming.' The debugger consists of a client-side component and a host system component. The components talk to one another over a TCP/IP connection. Using the debugger on the client side requires the use of X10DT, the Eclipse-based IDE for X10 (IBM developerWorks).

The X10 Parallel Debugger is a fairly new product, so bugs are to be expected. However, the goal of the project is to provide a complete parallel debugger for X10 code, so in the long-term, it should be robust.

3.2.3.2. Performance analysis tools

At present no performance tools specifically designed for native X10 code exist. However, a prototype performance tool based on the Parallel Performance Wizard (PPW) and the GASP interface has been proposed by Alan George of the University of Florida (IBM).

Additionally, there are several existing methods for doing performance analysis on X10 codes. The first method is to profile the C++ executables produced by X10; the X10 website suggests using Google profiling tools and Valgrind-based tools on x10c++[1] executables (IBM).

The C++ X10 runtime also contains tracing options that can be set through environment variables. Allocations, serialization points, and messaging between places can all be traced; other tracing options are also available (IBM).

There are a number of suggestions and best practices for achieving optimal performance from your X10 application on the X10 Performance Tuning page (IBM) (see bibliography for link).

3.2.3.3. Other Tools

Additional tools that are either proposed or in current development for X10 are described on the X10 Tool Development page (IBM) (see bibliography for link). Besides the novel performance toolset, tools in development include a consistency checker and a deterministic replay tool.

3.2.4. Performance

We did not find published studies of the performance of X10 code compiled to C++.

Some of the X10 team has, however, conducted research on the performance of X10 compiled to Java. In the X10-to-Java case, KmeansSequential benchmark results lagged native Java by a range of factors depending on the X10 release; the latest release in the comparison was 2.1.2, released in February of 2011. That release took 1.8x as long to complete as the native Java implementation. KMeansSPMD performance was tested up to 12 places, with a maximum speed up of 7.7x using X10 release 2.1.2 (Takeuchi, 2011).

[1] x10c++ executables are X10 executables that have been compiled to C++ code.

According to Grove, X10 code scales fairly well and is capable of running up to a few hundred or as many as 1000 cores – the HPC Challenge Benchmarks have been run at this scale (Grove, 2011). Sur stated that running at Terascale is not presently possible (Sur, 2011).

Sur also stated that the language is not yet performance competitive with C or C++ for serial applications. He estimated that this would take a number of years before it attains competitive equilibrium with C++ (Sur, 2011). Grove admitted that the X10 implementations of some codes tested underperforms, but he believed that this relates partly to the application code design and not strictly to the X10 language (Grove, 2011).

3.2.5. Suitability to LLNL Application Codes

Like Chapel, programming in X10 would be a radical shift for LLNL application code teams. However, this is not necessarily a negative if the shift provides benefit in the form of reduced time to code, large reductions in lines of code, and greater maintainability. Also like Chapel, at present X10 cannot exist as a secondary language; while C or C++ object files can be linked into an X10 application, the reverse is not possible. We are not aware of any BRAID-like projects to provide the capability to link against standalone X10 objects from a non-X10 program.

According to Sur, who developed the MPI runtime for X10, the present focus of X10 development is on making it suitable for use in distributed computing applications (Sur, 2011). Because these are more loosely coupled application codes than MPI parallel application codes, they do not stress the parallel runtime as much as a scientific application code would. This means that presently the language is not being heavily tested with tightly coupled scientific applications. Sur speculated that it would be a relatively long amount of time before X10 would be ready for parallel scientific application code development, partly because serial performance still significantly lags that of native C or C++ compiled code.

We asked Grove about the state of object compatibility between X10 objects and non-X10 objects. He said that it was possible to make calls to non-X10 functions from within X10, but that calling back into X10 from an external object was 'rough' and while perhaps feasible, not very elegant (Grove, 2011).

X10's asynchronous PGAS framework provides the programmer with greater flexibility than a standard PGAS model by allowing threads (activities) to be created dynamically and by making it possible for dynamic load balancing to occur. This means heterogeneous systems and applications that require load balancing should both be supported, increasing the potential for adoption. This may have value for some LLNL applications that exhibit workload heterogeneity.

Multidimensional arrays in X10 are declared as multidimensional, rather than as arrays of arrays as in C or C++. This can be beneficial for identifying programming errors and allowing for the memory layout of the implementation to be independent of the array reference notation (Brezin, 2010). Currently X10 does not appear to provide features equivalent to Chapel's domain maps or the distributions construct.

Like Chapel, X10 should greatly reduce the amount of code required for a particular parallel application, relative to the amount of code in an MPI-based or MPI/OpenMP code. This is due to the absence of message passing code, as global memory locations can be addressed as if they are local.

The fact that X10 is an extension to Java makes it a language that should be well understood by a large number of programmers. Like Java or C++, however, there are characteristics of the language syntax

that may make it more difficult for some programmers to maintain or work with. A solid understanding of existing classes will be necessary to optimally program and maintain application code, and in many cases there may be a nontrivial learning curve associated with acquiring this understanding.

3.2.6. Bibliography

1. Brezin, J. e. (2010, December 2). An Introduction to Programming with X10.
2. Grove, D. (2011, January). (C. Shereda, Interviewer)
3. Hudak, D. (2010, October). The X10 Language and Methods for Advanced HPC Programming. Retrieved September 2011, from Ohio Supercomputer Center: www.osc.edu/~dhudak/Site/About_Me_files/x10-tutorial.ppt
4. IBM developerWorks. (n.d.). IBM Parallel Debugger for X10 Programming. Retrieved September 2011, from IBM developerWorks: https://www.ibm.com/developerworks/mydeveloperworks/groups/service/html/communityview?communityUuid=e3fb8100-3ee3-4402-bf67-bf66b29797ea
5. IBM. (n.d.). Frameworks and Libraries. Retrieved 10 2011, from X10: Performance and Productivity at Scale: http://x10-lang.org/x10-community/x10-in-use/frameworks-a-libraries.html
6. IBM. (n.d.). Tool Development. Retrieved 10 2011, from X10: Performance and Productivity at Scale: http://x10-lang.org/x10-community/x10-in-use/tool-development.html
7. IBM. (n.d.). X10 Frequently Asked Questions. Retrieved September 2011, from X10: Performance and Productivity at Scale: http://x10-lang.org/home/faq-list.html#FAQ-WhatisthepurposeofX10%3F
8. IBM. (n.d.). X10 Performance Tuning. Retrieved September 30, 2011, from X10: Performance and Productivity at Scale: http://x10-lang.org/documentation/performance-tuning.html
9. Saraswat, V. (2008, May 21). APGAS: Programming for concurrency and distribution. Retrieved September 2011, from IBM TJ Watson Research Center: http://www.cs.waseda.ac.jp/gcoe/eng/whatsnew/img/080428VSaraswat1-APGAS.pdf
10. Sur, S. (2011, January 15). (C. Shereda, Interviewer)
11. Takeuchi, M. e. (2011). Compiling X10 to Java. X10 '11 (p. 10). San Jose, CA: http://dist.codehaus.org/x10/documentation/papers/X10Workshop2011/CompilingX10ToJava.pdf.
12. Weiland, M. (2007, October 10). Chapel, Fortress, and X10: Novel Languages for HPC. Retrieved September 2011, from HPCx, The University of Edinburgh: http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0706.pdf

3.3. Fortress

Owner / Development Location	Sun Microsystems (Oracle), now an OpenSource project
Project Website	http://labs.oracle.com/projects/plrg/index.html http://projectfortress.sun.com/Projects/Community (Old page, broken link)
Download Page	http://java.net/projects/projectfortress/sources/sources/show
Platforms Available	Windows, Linux, Solaris, Mac OS X (Java Virtual Machine)

3.3.1. Overview

The name Fortress is derived from the developer's goal of creating a "secure FORTRAN," i.e. "a language for high-performance computation that provides abstraction and type safety on par with modern programming language principles" (Allen, et al., 2008). Fortress began as Sun Microsystem's entry into the DARPA HPCS program that began in 2002 with the goal of "enhancing human productivity as well as improving computer performance, scalability and availability" (Acocella, 2006). Fortress was not funded for Phase III of the project, which began in July 2006, losing out to competitors X10 (IBM) and Chapel (Cray). The project was then converted to an open-source project, which has an uncertain future, especially following the acquisition of Sun Microsystems by Oracle in 2009/10. A project page still exists for Fortress, now located on the Java.net website and there is apparently active work being done on the language, although it is not being promoted and seems to be a hobby limited to some of the initial team from Sun.

The core problems that Fortress seeks to address include working on:

- Mathematical Notation:
  - Provide a way to better realize mathematical notation within high-performance programming language.
- Dimension Checking:
  - Provide support for static checking of units and physical dimensions in an OO-type system that would allow polymorphic programs.
- Communication Efficiency:
  - How to effectively move large amounts of data around while continuing computation?
- Memory Consistency:
  - Large Global Shared Memory Space: "we must trade away the simplicity of cache coherent shared memory and manage consistency more explicitly" (Oracle, Project Fortress Overview)
- Synchronization:
  - Absence of cache coherency requires major changes to the shared-memory synchronization mechanisms (compare and swap, load locked/store conditional) making certain operations more difficult (asynchronous message sends, barrier synchronization, and parallel prefix).

Use of mathematical notation is a key component of the Fortress language specification. The idea is that mathematical notation is a framework which has been developed over thousands of years, and is readily taught and understood by scientists and engineers alike. In fact, even the concept of dimensions and units of measure are built into the language allowing for equations and functions to carry this

information with them.

Fortress uses a large global shared memory space to get around problems involved with message passing frameworks. Currently there are few details about the specifics of how this will work in particular, though it's possible this information currently still resides on the Sun website pages, which have not completely been moved over to the Oracle servers.

Fortress aims to provide *inherent parallelization.* For instance, loops in Fortress are parallel unless they are explicitly performed using sequential generators. Individual iterations of a loop are treated by default as being completely independent, so that the compiler can parallelize them to take advantage of the hardware available.

3.3.2. Present State of the Model

The first stable release (Version 1.0) of the Fortress language specification was released in April 2008, along with a compliant implementation targeting the Java Virtual Machine (Wikipedia). Currently there is a download link from the Java.net site which provides information about how to download the source code for the language and the compiler; however, at present there are no real programming examples and much of the documentation is missing or outdated.

The compiler is still actively being developed by Guy Steele and others. Currently only a subset of the language specification has been implemented. Guy Steele has created web entries in 2011 concerning performance and implementation of the compiler, and the team has recent publications in top tier programming language conferences. However, most of these publications highlight problems that affect many programming languages and the Fortress team's solutions to these problems, rather than focusing solely on the Fortress language.

3.3.2.1. Robustness and Known Issues

Fortress was developed to run on the Java Virtual Machine which allows a very widespread audience for the language. Some estimates are that there are over 4.5 billion JVM-enabled devices and 6.5 million Java developers worldwide (Oracle, Learn about Java Technology). Such familiarity helps to alleviate some of the ramp-up required to adopt new programming languages or programming models.

Development and progress on Fortress beyond the language specification itself has been slow. There are a number of factors contributing to this, including Oracle's acquisition of Sun Microsystems in 2009/10 and the cancellation of funding under the DARPA HPCS program. Currently the development is handled primarily by the original Sun Programming Languages Team, now part of the Oracle Programming Languages Group, and their development has focused on adding functionality to the Fortress compiler.

3.3.3. Tool Availability

Currently there is very little information regarding the tools available to Fortress developers. This is primarily due to the fact that the compiler is the primary development goal of the Fortress team, and currently does not fully implement the language specification. Due to the compiler being incomplete, there are no currently available tools for debugging or measuring the performance benefits of Fortress relative to other parallel models.

3.3.4. Performance

Currently there are no performance numbers or porting examples available to compare Fortress to other prominent HPC languages.

3.3.5. Suitability to LLNL Application Codes

Due to Fortress's distinction as being a "secure FORTRAN," the language is designed such that it would be familiar for a FORTRAN developer to pick up easily. Language syntax and an emphasis on units, dimensions, and mathematical notation are aimed at increasing the readability of the code by the physicists, chemists, engineers, and mathematicians that are the source of the content of an application (Steele & Maessen, 2006).

3.3.6. Resources and Additional Information

Oracle Project Page: http://labs.oracle.com/projects/plrg/index.html
Java.net Page: http://java.net/projects/projectfortress/pages/Home
Wikipedia Page: http://en.wikipedia.org/wiki/Fortress_%28programming_language%29

Parallel Programming and Parallel Abstractions in Fortress:
http://www.brics.dk/pilambda/old/docs/Aarhus-Fortress-Parallelism-2006public.pdf

3.3.7. Bibliography

1. Acocella, K. (2006, 11 21). IBM Press room - 2006-11-21 DARPA Selects IBM for Supercomputing Grand Challenge. Retrieved 09 20, 2011, from IBM : http://www-03.ibm.com/press/us/en/pressrelease/20671.wss
2. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., et al. (2008). The Fortress Language Specification. Sun Microsystems, Inc.
3. Oracle. (n.d.). Learn about Java Technology. Retrieved 08 25, 2011, from Java: http://www.java.com/en/about/
4. Oracle. (n.d.). Project Fortress Overview. Retrieved 08 25, 2011, from Oracle Labs: http://labs.oracle.com/projects/plrg/Fortress/overview.html
5. Steele, G., & Maessen, J.-W. (2006). Fortress Programming Language Tutorial. Ottawa, Canada: PLDI.

3.4. Cilk Plus Language Extensions

Owner / Development Location	Intel
Project Website	http://software.intel.com/en-us/articles/intel-cilk-plus/
Download Page	http://software.intel.com/en-us/articles/intel-software-evaluation-center/
Platforms Available	Windows, Linux, Intel based parallel machines

3.4.1. Overview

Cilk Plus, the concept developed originally at MIT, has been made a part of Intel Parallel Building Blocks. It is fairly straightforward as an extension of C/C++. The language is meant to be processor oblivious, which provides the programmer scalability without the need to rewrite code in order to utilize new architectures with additional cores. As such it is targeted mainly at legacy code that may be easily parallelized through the aid of the Cilk Plus keywords and parallel constructs. The language does not contain constructs for multiple nodes and an existing framework such as MPI would be required for communication across nodes of a platform.

The main concept behind the Cilk Plus programming model is work-stealing. That is, when a *worker* (roughly equivalent to a thread) has completed all of its assigned work, it attempts to steal *strands*[2] of work from other workers. This allows for good load balancing across the cores running worker threads. Each worker is assigned a worker id, which is an integer ranging from *0* to *n-1*, where *n* is the total number of workers allocated (the default is the number of cores available on a node). Here, worker *0* is referred to as the user thread, which is where the serial sections of code are executed. The remaining workers are referred to as Cilk threads, and these are where the parallel sections of code are executed after work is load balanced. The user thread is also utilized during these parallel sections for a total of *n* executing threads. Work-stealing is handled by the Cilk Plus runtime system. (Frigo, 2011)

Cilk Plus contains only a handful of keywords that control the bulk of the parallel related tasks that a programmer would use. These are the *cilk_spawn*, *cilk_sync*, and *cilk_for* keywords. The language provides additional features such as elemental functions that take advantage of vector operations available on the hardware. Definitions for reductions are present to simplify certain codes.

The most commonly used keyword in Cilk Plus is *cilk_spawn*, which is used to branch parallel sections of code. Each spawn creates a strand of work that is scheduled for a worker to process. Both the user thread and Cilk threads may be used to perform spawned work, and the default thread that runs the strand is the parent thread that spawned it. If and only if there is a larger quantity of work to do, multiple strands are broken up to be processed by other workers.

Following a *cilk_spawn* in which there are dependencies on the return from the spawn, the Cilk Plus program must provide a matching *cilk_sync* to ensure that the spawned threads have finished their strands of work before moving forward. This synchronization point is local, meaning a parent thread may continue once its children threads have all finished, as opposed to a global synchronization, where all threads must finish. This allows for more fluid workflow control in a program, especially with respect to task-based parallelism.

[2] A **strand** describes a serial section of the program. More precisely, the definition of a strand is "any sequence of instructions without any parallel control structures."

The *cilk_for* keyword is used primarily where there is a higher level of data parallelism, where the many iterations in a standard for loop are independent of one another, allowing for parallel execution. One of the key advantages to using the *cilk_for* keyword, as opposed to a standard loop with *cilk_spawn* inside the iteration, is that the *cilk_for* splits up the work using a divide-and-conquer approach. This decreases the overall execution time because it counters some of the overhead associated with the otherwise serialized spawning.

There is an optional compiler directive that allows the programmer to choose the granularity of *cilk_for* parallelization. This is *#pragma cilk grainsize* where the value of the parameter may be an expression. The default grainsize chosen by the compiler if none is explicitly specified is:

$$min(512,N/(8*p))$$

where N is the number of iterations, and p is the number of workers.

The Cilk Plus runtime provides a handful of API functions to obtain information such as the worker a strand is on as well as the modification of certain parameters. Through the runtime calls, programs may manually shutdown or initialize the runtime scheduler. This allows for the number of workers to be changed between parallel sections of code, for instance. This manual starting and stopping of the runtime may only be performed outside of *spawning functions*, that is, a function containing any one of the Cilk Plus keywords.

3.4.2. Present State of the Model

Development for the predecessor of Cilk Plus began as simply Cilk at MIT around 1994 and has since been refined and adopted by Intel, with the release of Cilk Plus in 2010. In spite of the relatively young age of the language, because of its minimal size and background, essentially all of the functionality of the language is implemented. Furthermore, Intel has provided its own additions to the language to utilize certain features of target architectures, such as vector operations through elemental functions. The Cilk Plus language extension specification is currently only supported by the Intel C++ Compiler. Although Cilk Plus is meant to be, as per Intel, a language specification with the implementation dependent on the platform, at present it only runs on a select set of compatible processors. It is expected that the Cilk Plus implementation for the Intel C++ Compiler will continue to be refined.

Because Cilk Plus is an extension to C/C++, its compatibility with existing languages is excellent, especially considering the Babel project at LLNL provides the necessary glue to communicate between several of the HPC languages currently used (Epperly et al. 2011). Cilk Plus code is also compatible with OpenMP, and both OpenMP directives and Cilk Plus keywords may be present in the same code file. Although both of these languages use pthreads currently, they use separate thread pools. However, the program must not interweave the two thread pools, or it will break. A downside to using both Cilk Plus and OpenMP in the same program is that the Cilk Plus runtime negatively affects the performance of OpenMP – Cilk Plus will steal processing time from OpenMP in order to execute (Intel Documentation, 2010). This issue may be preventable, although we did not determine a method for doing so.

Cilk Plus currently operates in a manner similar to the specifications of the MIT Cilk releases (MIT Supertech 1998). As such it both benefits from the advantages, such as load balancing, and suffers from the same pitfalls as its predecessor. Perhaps the most critical pitfall of Cilk Plus is its high overhead. When compiling Cilk Plus code, both fast and slow versions are created for nano- and micro-scheduling,

respectively. A function call to even the fast version requires the allocation of the procedure's frame, saving of the procedure state, checking for stolen frames, freeing frames, and a frame pointer. The slow version induces extra overhead by extracting arguments through the frame pointer and by managing synchronization with potential child procedures on neighboring cores. All of the excess work-stealing tasks not present in a serial implementation are also unpreventable. This is because when a Cilk Plus spawning function spawns work, the thread that it is located on is assigned the entirety of that work. It is then the responsibility of the other Cilk threads to steal the excess work from the original thread. This means that work-stealing in a parallel algorithm using Cilk Plus is inevitable. It is unclear how much more the current mechanism may be optimized, given the same framework. (Ganesh, 2011)

3.4.2.1. Robustness and Known Issues

T he functionality of the language is implemented, and these features are fairly robust, albeit less than optimal in terms of performance. Given the simplicity of the language extension there is not much chance for a programmer to make errors when coding. The tests we ran regarding local synchronization indicate that the language appears to scale well with increased core counts. The work-stealing mechanism performs load balancing fairly reliably and a load imbalanced scenario performs similarly to that of a load balanced scenario. We believe this can be attributed to the way that Cilk threads steal from the main worker thread.

There appears to be an underutilization of processor cores for relatively small amounts of work. Although related to the mechanism behind the work-stealing, Cilk Plus does not give all of the available processors work at the start of a parallel section, so worker threads incrementally steal work. In some instances, not all of the cores are utilized, with a larger bulk of the work being performed sequentially on only a few cores. When all the processors were utilized, the runtime scheduler occasionally hangs prior to the utilization of the last remaining core, often equivalent to the same amount of time required for a single iteration. It is possible that these runtime issues have been resolved with the newest version of the Intel C++ Compiler; the issue was seen with version 12.0.191.

3.4.3. Tool Availability

There are several tools available for Cilk Plus, both for debugging and for profiling. Specific to Cilk Plus are *cilkscreen* and *cilkview*, which are bundled with the SDK. *CilkScreen* may be used for race detection across the parallel tasks in a program, while *CilkView* generates performance data and calculates performance estimates, which it then uses to visualize the application performance graphically. Intel also provides extensions to Microsoft Visual Studio in its Intel Parallel Debugger for both viewing the execution flow of a Cilk Plus parallelized code and for viewing the Cilk Plus thread stack for further analysis. Finally, the Intel C++ Compiler can turn a Cilk Plus program into its equivalent "C-elision," where all the keywords and parallel constructs are replaced or removed, serializing the code. The resulting executable may then be debugged with any number of compatible debuggers (e.g. gdb) for correctness.

3.4.4. Performance

In comparing a port of the ASC Sequoia benchmark, CLOMP, to the original OpenMP implementation, we found that Cilk Plus underperforms in a rather diverse parameter space, even though the best-case implementation (without synchronization) is comparable to that of OpenMP. The exception is the test for synchronization where Cilk Plus, likely due to the local nature of the synchronization, not only outperforms, but also appears to scale better than the standard barrier. In discussions with the Cilk Plus development team, the lack in performance was attributed to the overhead of the work-stealing mechanism, which, in this case, is rather severe. For the "target input" of the CLOMP benchmark, it was found that Cilk Plus experienced slowdowns relative to the serial version. When increasing the amount

of work per iteration, the ported program performed comparably with OpenMP dynamic scheduling, and although this experienced some moderate speedups, it was still slow. (Hewitt & Ganesh, 2011)

Due to the dynamic nature of Cilk Plus' distribution of work to threads, we found that Cilk Plus performance suffered much less from a load imbalanced problem than did a statically scheduled process. However, while this behavior of Cilk Plus is preferable, it comes at a large cost, especially to parallel sections that are more regular. Although Cilk Plus does well in these imbalanced scenarios and it conforms nicely for recursive algorithms, it appears that the largest barrier to the performance of Cilk Plus is also the most critical element of the programming model: work-stealing. Much of the performance issues with work-stealing have been mentioned above, but it is worth stating again that the present implementation does not seem to allow much room to optimize. It is unclear how well this model will scale to larger platforms, in spite of the local synchronization performing well.

3.4.5. Suitability to LLNL Application Codes

Because Cilk Plus is a language extension to C/C++, LLNL application teams would not need to rewrite much code in order to use it. Interoperability with existing code is essentially a nonissue considering the Babel project at LLNL is already able to communicate well between several HPC languages. However, Cilk Plus is mainly ideal for legacy code that has not yet been parallelized. Considering the performance numbers currently seen through the CLOMP port, it may be more practical from a performance standpoint to simply use OpenMP. The cases where Cilk Plus would be most applicable are algorithms that use recursion, or where the workload is sharply imbalanced. However, this does not seem to be the case with the majority of existing application codes.

Although the Cilk Plus language does not provide any support for sparse matrices, the elemental functions, reductions, and scans may make programming certain portions of an application easier. Apart from these additions, the language does not provide much advantage over OpenMP. It is restricted to a single node and thus existing message passing implementations (e.g. MPI) will need to provide the necessary communication across nodes, resulting in a heterogeneous programming model. Moreover, in comparing the keywords in Cilk Plus and the directives in OpenMP, it is clear that the ease of programming is not a concern for either, with OpenMP providing additional options in scheduling, allowing for NUMA effects in some variations. In this sense, Cilk Plus is considerably more limited than OpenMP. While the language certainly has its advantages, it is unclear that these advantages are applicable to LLNL application codes.

3.4.6. Resources and Additional Information

Many of the resources available for Cilk Plus are provided solely through Intel.

The main Cilk Plus project website provides links to the current distribution as an SDK, the language, runtime ABI specifications, and additional whitepapers. It is located at:
    http://software.intel.com/en-us/articles/intel-cilk-plus/

Tutorials, learning library (filter on Intel Cilk Plus for "All Products")
    http://software.intel.com/en-us/articles/intel-learning-lab/

Blog: http://software.intel.com/en-us/blogs/tag/cilk-plus/

Intel also provides a Cilk Plus reference in their Intel C++ Compiler User and Reference Guides:

http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/cpp/win/cref_cls/common/cilk_bk_using_cilk.htm

The Intel Learning Lab has some basic information about Cilk Plus:
http://software.intel.com/en-us/articles/intel-learning-lab/

Intel Software Network support forum: http://software.intel.com/en-us/forums/intel-cilk-plus/

Information on Cilk, including a reasonably in depth manual, may be found at the MIT Supertech page: http://supertech.csail.mit.edu/cilk/

Wikipedia page: http://en.wikipedia.org/wiki/Intel_Cilk_Plus

A few features of Cilk Plus have not been documented, primarily information relating to the runtime system. These may be found by downloading the current distribution and viewing some of the header files.

Papers:

1. Epperly, Tom etal (2011) ; Babel Project ; https://computation.llnl.gov/casc/components/#page=home
2. Feldman Michael (August, 2011) *Intel Opens Up Cilk Plus*; HPC Wire; http://www.hpcwire.com/hpcwire/2011-08-17/intel_opens_up_cilk_plus.html
3. Frigo, Matteo (June, 2011); *Cilk Plus: Multicore extensions for C and C++;* MIT; http://developer.amd.com/afds/assets/presentations/2080_final.pdf
4. Ganesh, Kittur (July 2011); *CILK: no conceivable way to have 0 steals*; Wang, Felix interviewer; https://docman.llnl.gov/ICCD/SMS/Lists/TeamDiscussion
5. Geva, Robert (August 2011); L*anguage of the Month: Intel's Cilk Plus* ; Dr. Dobb's; http://drdobbs.com/architecture-and-design/231400279
6. Hewitt, Brandon; Ganesh, Kittur (July 2011) *Cilk Plus Performance and Mechanics Assistance*; Wang, Felix interviewer; https://docman.llnl.gov/ICCD/SMS/Lists/TeamDiscussion
7. Intel Documentation (2010); *Nested use of OpenMP* and Intel(R) Cilk(TM) Plus in same program*; http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/ptref/pt_reference/references/sc_omp_different_parallel_kind.htm
8. Larabel, Michael (August 2011) *Easy Parallel Programming: Cilk Plus Ported To GCC*; Phoronix; http://www.phoronix.com/scan.php?page=news_item&px=OTc5NQ
9. Leiserson, William ; Luk, Chi-Keung (CK) ; Tu, Peng (2010); *Parallel Programming in Intel®CilkTMPlus with Autotuning; Intel Software*; http://www.ckluk.org/ck/talks/cilkplus-tutorial-ppopp11.pdf
10. Moore, John (2011) ; *Tech Trends: Language Lessons: Where New Parallel Developments Fit Into Your Toolkit; Intel Software Dispatch*; http://www.intelligenceinsoftware.com/feature/tech_trends/language_lessons/index.html
11. Ramkumar, Krishna (Fall 2010); *Intel Cilk Plus*; The Parallel Universe, Intel; http://img.delivery.net/cm50content/intel/ProductLibrary/4093_IN_ParallelMag_Issue4_090310.pdf
12. Wang, Felix (July 2011); *INTEL CILK PLUS PROGRAMMING MODEL Performance Comparisons with CLOMP*; https://docman.llnl.gov/ICCD/SMS/Shared%20Documents/Cilk/cilkplusreport.pdf

3.5. Intel Parallel Building Blocks: Array Building Blocks and Threading Building Blocks

Owner / Development Location	Proprietary effort by Intel Corporation, Santa Clara, CA
Project Websites	http://software.intel.com/en-us/articles/intel-parallel-building-blocks/ http://software.intel.com/en-us/articles/intel-array-building-blocks/ http://software.intel.com/en-us/articles/intel-tbb/ http://threadingbuildingblocks.org/
Download Page	http://software.intel.com/en-us/articles/intel-parallel-building-blocks/ http://threadingbuildingblocks.org/download.php
Plaforms Available	

3.5.1. Overview: TBB and ArBB Common Characteristics

Intel Parallel Building Blocks (PBB) is a three-part HPC toolkit targeted at single-node multi-core programming. Elements of PBB are open source but it is mostly supported and developed by Intel Corporation. PBB consists of three elements: Array Building Blocks (ArBB), Threading Building Blocks (TBB), and CilkPlus (Cilk+ or simply Cilk). This section will focus on TBB and ArBB.

PBB's design goals are to enable programmer productivity, code portability, and code performance. These design goals are sometimes in tension with each other, and each piece of PBB uses different approaches to attain them. Each PBB component has its strengths relative to the others, and all three PBB components are compatible and interoperable.

TBB and ArBB are C++ template libraries that do not extend the C++ language per se, nor do they require a library to link with or a special compiler. Because they do not rely on language extensions, TBB and ArBB are potentially more portable than Cilk+.

With respect to the current HPC standards of OpenMP and MPI, PBB is more aptly compared with OpenMP than with MPI as it concerns itself with intra-node rather than inter-node parallelism. PBB is very similar to OpenMP in that it abstracts some of the "bookkeeping" and other thread logistics away. Unlike OpenMP, however, PBB is based on C++ templates, and this will either tempt or horrify programmers, depending on their stylistic preferences. For developers used to C++ Standard Template Library constructs, PBB will seem to have a more natural and expressive vocabulary than OpenMP.

3.5.1.1. Threading Building Blocks

Intel Threading Building Blocks (TBB) is a C++ template library intended to support task parallelism without explicitly managing threads. TBB does not implement any vectorization. That is left for the programmer to manage, either with pragmas, compiler flags, platform-specific vectorization API calls such as SSE, or using a vectorizing template system such as ArBB. TBB's existence and task scheduling approach were inspired by the MIT Cilk Art project, which later became part of PBB and is now called Cilk+. There are substantial differences between TBB's scheduling implementation and Cilk+, however, as Cilk+ narrowly concerns itself with fork-join thread scheduling, while TBB supports a very broad range of scheduling constructs.

TBB bases its scheduling on a task graph, which is a directed acyclic graph of tasks, ordered by their execution dependence on other task. The core implementation concept for TBB is that it implements a

“work stealing” algorithm to traverse this task graph in both breadth-first and depth-first manner simultaneously, responsively balancing workloads while deferring tasks as little as possible.

Work stealing is implemented as follows. Each thread has its own task queue, which is a list of task lists. Task lists get added to the work queue of a particular thread at the head of the queue. The local thread works off the head of the queue, maximizing locality on the local thread, and other worker threads can steal task lists from the tail of the queue, allowing load balancing.

Looking at this in more detail, taking tasks from the head of the queue is a depth-first traversal of the task graph corresponding to a task stack. As a local thread descends in this manner into the task graph, other threads steal work from the tail. This operation corresponds to a FIFO, and is a breadth-first operation at a high level in the graph. The depth-first traversal enables data locality while the breadth-first work stealing partitions the problem into broad chunks, which is a load balancing operation.

The task stealing scheduler allows nested parallelism to be easily and naturally expressed without worrying about contention for resources. An example of such a tool for avoiding contention is the TBB wait() call, which crucially does not block the calling thread; it instead only blocks the current task and leaves the thread immediately free to begin another task.

TBB contains lower level and higher level components. The lower level components of TBB include: Task Scheduler, Thread Local Storage, Synchronization Primitives, Memory Allocation, Threads and others. The higher level domain of TBB has two elements: Generic Parallel Algorithms, which are analogous to C++ STL algorithms, and Concurrent Containers, which are analogous to C++ STL containers.

The lower-level TBB constructs are aimed at allowing TBB programmers flexibility in making lower level parallelism choices to optimize performance while preserving a serial coding style. As an example of the lower level flexibility in the Task Scheduler, creating a custom partitioner can allow the Task Scheduler to make a better partitioning of the task graph without the programmer having to worry about explicitly creating or managing the graph. Synchronization Primitives, instead of managing all aspects of critical code section access, allow the programmer to use class accessor objects, a.k.a. “local locks,” to protect specific data regions instead of just execution blocks. This reduces contention, a frequent cause of performance problems in threads, while adding little code complexity.

The higher-level TBB constructs conveniently express and automate many common tasks. Parallel Algorithms such as parallel_for() and parallel_reduce() allow programmers to express normal serial programming constructs in a natural way, yet still take advantage of the TBB core task stealing. Similarly, Concurrent Containers include C++ constructs which present the proper interface for use with Parallel Algorithms, such as concurrent_vector. The programmer thinks in terms of for loops and reductions, while TBB and ArBB manage the load balancing and task graph.

Other examples of the high-level TBB toolkit include the parallel_pipeline template, which encourages data locality by passing tasks through a pipeline in chunks rather than executing each pipeline stage; the spin_mutex, which implements an exponential backoff non-blocking mutex; and a novel concept called “continuation passing” that allows threads to not only steal work, but steal work dependencies from each other. (Kukanov & Voss, 2007)

TBB includes a scalable memory allocator that avoids the need for global lock acquisition in memory allocations by managing its own memory space in a threadsafe manner. You can also just interpose a

special libmalloc to get this behavior without any code changes, assuming you are using TBB in your code.

Since TBB is a C++ template library, it can take advantage of lambda expressions in the new C++0x standard to greatly simplify the syntax and help envelop existing code blocks into TBB calls. Lambda expressions are a simple way to group code into code blocks and then manipulate those code blocks in a manner similar to function pointers. This avoids having to package code sections into functions with defined signatures, and then passing function pointers as one might do in a library such as POSIX Pthreads. The result is somewhat similar in style to OpenMP pragmas, but does not require special compiler support. (Reinders, 2009)

3.5.1.2. Array Building Blocks

Intel Array Building Blocks (ArBB) is the newest component of PBB. ArBB is the merger of Intel's 2007 product, Intel Ct, with technologies that Intel received when they acquired RapidMind, Inc. ArBB is a C++ template library whose main purpose is to enable programmers to more easily exploit multiple cores and GPUs using data parallelism, primarily using vectors and SIMD, without the need to manage task parallelism. ArBB seeks to strike an appropriate balance between performance, productivity and portability. The target community for ArBB is all HPC thread programmers, particularly those who want to focus more on their code structure and algorithmic expression without delving deeply into hand-tuning their code.

ArBB makes use of special C++ templated data types and a linked-in JIT compiler in a virtual machine to take ordinary user-defined functions and optimize them for the architecture as detected at runtime. ArBB has no parallel execution features, but does use TBB for threading as needed to implement its vectorizing features. Vectorizing the data and using machine hardware effectively produce all of the ArBB acceleration.

ArBB performs runtime analysis to produce architecture-specific optimizations such as taking advantage of available vector hardware units. The first time a given ArBB function call is made, it incurs a performance penalty as it is compiled and optimized by the runtime library, but on subsequent invocations it can be reused without this penalty.

ArBB is distinguished from the data parallelism of Array Notations included with Cilk+ by its template basis. ArBB can vectorize arbitrary data elements, while Array Notations only operate on native C/C++ types. However, ArBB may require more code to be rewritten than Array Notations, and Array Notations allow greater control over threading than ArBB.

With respect to exascale computing, perhaps the biggest drawback of ArBB is its "copy in, copy out" semantic foundation, which will increase memory requirements for ArBB programs. This is especially problematic given the low memory per core expected at exascale. It is unavoidable, though, as without these data copies between C++ and ArBB memory partitions, the ArBB dynamic engine cannot optimize the data for multiple cores. When this memory overhead is acceptable, its semantics allow ArBB to give programmers an expressive and readable way to manipulate and process data structures.

Creating execution regions in the code that are purely ArBB can minimize the need for memory copies. This is analogous to minimizing memory transfers when programming with CUDA or OpenCL. However, ArBB does not support cluster programming, in the sense of distributed memory, nor does it provide

facilities for I/O or network communication. Therefore, memory copies would be inevitable at all synchronization and I/O points, and would incur memory overhead with most existing LLNL codes.

Like TBB, ArBB will likely require code rewrites for existing codes. Since ArBB uses TBB as its threading model, using ArBB and OpenMP together might result in a performance penalty as TBB and OpenMP could compete for thread resources.

ArBB uses C++ exceptions internally, which could be a consideration when linking. Also, C++ exception handling can result in a performance and size penalty regardless of whether an exception is thrown or not.

3.5.2. Present State of Model

PBB is a recent technology. During a seminar in September of 2010, Noah Clemons of Intel referred to PBB as a "new initiative in parallelism within Intel." (Clemons, 2010)

ArBB is currently in beta release, currently beta 6. The current release is not considered to be a performance-optimized release (Intel teleconference with LLNL, September 2011).

TBB is the most mature element of PBB, having been released circa 2006. TBB is currently at version 3.0, and has been released on Linux, Windows, and Mac OS. Intel also provides open source code releases which enable ports to other platforms such as XBox360 and Solaris.

PBB is well supported and documented by Intel with written materials, white papers, online videos, and user communities. It is included as a basic part of their toolkits on their newer MIC architectures as well as more mainstream offerings.

PBB appears to be part of a strategy by Intel to ensure developers on their hardware have a robust, high-performance, productive software toolkit to develop with and there is no expectation that this toolset will disappear in the next few years. Rather, it should get stronger and the feature set should grow. In particular, the ArBB developers are actively looking for feedback on how their code can be improved. It remains to be seen whether they will implement critical features such as internodal communication, I/O support, and zero-memcopy or memcopy-minimizing semantics.

3.5.2.1. Robustness and Known Issues

No users of TBB or ArBB were found to exist at LLNL, making any analysis of robustness difficult.

3.5.3. Tool Availability

ArBB and TBB are entirely C++ template-based tools, so there is no special debugger needed.

Similarly, while specialized performance tools are not specifically available for these tools, ordinary instrumentation should enable adequate analysis when desired.

3.5.4. Performance

No performance numbers are available from Intel. Early research by Cilk Art indicates that the work-stealing scheduler is a good general-purpose algorithm that gives locality and load balancing advantages (see http://supertech.csail.mit.edu/papers.html), but recent direct performance comparisons are not available. Since ArBB is in beta release, performance is not likely to approach that of tuned SSE or other vector code (Kukanov & Voss, 2007).

Without the benefit of performance numbers it is safe to assume that for static scheduling, OpenMP outperforms TBB. This also seems to be confirmed by performance and algorithmic analysis (Faxen, 2010).

3.5.5. Suitability to LLNL Application Codes

While heterogeneous programming models are definitely an important direction to look at as a possibility for exascale computing, ArBB and TBB might be too narrowly focused on single-node performance and code simplification to meet the needs of exascale computing. Intel is working on "Inter-Array Building Blocks" to scale to internodal dimensionality but this technology is currently speculative. As mentioned, the memory footprint of ArBB is projected to be very high, which is highly problematic for programming on anticipated exascale architectures.

For existing software, both TBB and ArBB require code rewrites without necessarily providing compelling benefits for the rewrite. For example, they will not mix well with existing OpenMP threading since OpenMP wants to own the threading model on a CPU, making an incremental rewrite problematic in terms of performance of mixed codes.

However, the rewrite itself might not be terribly difficult in terms of code organization, compared to other languages like OpenCL or CUDA, which often require a rethinking of the entire data flow in an application. Thus, an initial port from OpenMP to TBB might well be straightforward. The Intel port of the OpenMP SPEC benchmarks to TBB is available to illustrate this process. (Murashov, 2008)

Some code groups might prefer to avoid the complexities of templating. In this case they might consider using Cilk+, keeping in mind that Cilk+ uses TBB for threading.

The high level nature of TBB is probably not a feature that is sufficient to attract HPC code groups looking to extend or improve existing codes. Most likely it is the lower level ideas that might be important. However, in a situation where a code rewrite is necessitated by new restrictions or paradigms evolving out of a move to exascale, the high level nature of TBB and expressivity of the algorithmic components could elevate it as a choice worth considering.

3.5.6. Resources and Additional Information

TBB documentation is available at (Intel Corporation and open source community)
More information about Array Building Blocks can be found in these two videos: (Clemons, Intel® Array Building Blocks Technical Presentation: Introduction and Q&A, 2010) and (Intel Corporation, 2010)

3.5.7. Bibliography

1. Clemons, N. (2010, Oct 14). Intel® Array Building Blocks Technical Presentation: Introduction and Q&A. Retrieved Oct 18, 2011, from Intel Software Network: http://software.intel.com/file/31293
2. Clemons, N. (2010, Sep 23). Topic: Introducing Intel® Parallel Building Blocks. Retrieved Oct 18, 2011, from Intel Software Network: http://software.intel.com/file/31246
3. Faxen, K.-F. (2010). Efficient Work Stealing for Fine Grained Parallelism. IEEE 2010 39th International Conference on Parallel Processing (p. 313). San Diego, CA: IEEE.
4. Intel Corporation and open source community. (n.d.). TBB Documentation. Retrieved Oct 18, 2011, from Intel Threading Building Blocks for Open Source: http://threadingbuildingblocks.org/documentation.php
5. Intel Corporation. (2010, Dec 9). Intel Software Network. Retrieved Oct 18, 2011, from Intel ArBB Code Tips: http://software.intel.com/en-us/articles/arbb-webinar-dec-9-2010/

6. Kukanov, A., & Voss, M. J. (2007). The Foundations for Scalable Multi-Core Software in Intel Threaded Building Blocks. Intel Technology Journal , 11 (04).
7. Murashov, A. (2008, May 8). Porting OpenMP SPEC benchmarks to TBB. (Intel Corporation) Retrieved Oct 18, 2011, from Intel Software Network: http://software.intel.com/en-us/blogs/2008/05/08/porting-openmp-spec-benchmarks-to-tbb/.
8. Reinders, J. (2009, Aug 3). parallel_for is easier with lambdas, Intel Threading Building Blocks. (Intel Corporation) Retrieved Oct 18, 2011, from Intel Software Network: http://software.intel.com/en-us/blogs/2009/08/03/parallel_for-is-easier-with-lambdas-intel-threading-building-blocks/

3.6. UPC

Owner / Development Location	Lawrence Berkeley National Laboratory and UC Berkeley
Project Website	http://upc.lbl.gov/ upc-users@lbl.gov users email list http://upc.gwu.edu/
Download Page	http://upc.lbl.gov/download/
Plaforms Available	Check download page for current status

3.6.1. Overview

UPC, or Unified Parallel C, is a parallel programming language that is an extension of ISO C [ISO99]. Additionally, UPC is a PGAS (Partitioned Global Address Space) language using a globally-shared memory programming model which exploits data locality combined with a distributed-memory model for its underlying implementation. The computer scientist is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. UPC uses a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time, typically with a single thread of execution per processor. Parallelism is achieved through the use of shared memory and work sharing across independent *UPC threads* of execution, hereafter referred to as simply "threads." Each thread has its own private memory space, as well as an associated shared memory region of the global address space that can be accessed by other threads.

The implementation of UPC threads is not restricted to actual user-space threads and the two are distinct concepts. In the case of Berkeley UPC, UPC applications run on top of a layer called the GASNet (Global Address Space Network), and the GASNet determines the actual thread implementation. Possible GASNet layers include pthreads, PSHM (process shared memory), MPI (in which the UPC shared memory is actually implemented via MPI calls), and various network APIs such as Infiniband verbs.

The number of UPC 'threads'[3] is fixed at program startup, and does not change during the code's execution. This attribute of UPC makes it similar to MPI in that each process or 'thread' is alive from inception through exit.

Shared memory variables in UPC form the foundation of UPC's parallelism. Rather than exchanging data across threads through explicit communication as in MPI, information is exchanged primarily through the use of shared memory.

Shared memory variables are declared through the use of the *shared* qualifier. In UPC, shared variables are always of global scope and must be declared globally; there is no provision for local shared variable declarations. This limitation is discussed further in Section 3.6.2.

Work sharing is done primarily through the use of the *upc_forall* construct. This statement is used in place of the *for* statement on loops for which work sharing is to occur. The *upc_forall* statement is of the form:

[3] - the term *thread* in UPC refers to an independent execution of the code. The underlying implementation of a UPC thread can be either a thread or a process depending on the underlying network layer.

```
upc_forall(expression1; expression2; expression3; affinity)
```

The first three expressions are equivalent to those of a normal C *for* loop. The affinity component indicates to a particular thread of execution which subset of the total loop iterations it should execute. In the simple case, if affinity is an integer expression, a thread will execute all iterations in which (*affinity* modulo *number of threads*) equates to the current thread number. The simple case of using the loop counter variable as the affinity expression usually results in best performance.

3.6.2. Present State of the Model

UPC has been in development for over 10 years, primarily in the academic community. As more people increase their interest in multicore architectures, people are giving more consideration to the possibilities of PGAS languages and UPC. Section3.6.6 lists several notable papers along this vein. The most active UPC development occurs at Lawrence Berkeley National Laboratory, with a small group of developers that work on several areas of the Berkeley UPC project. This group maintains a project mailing list, upc-users@lbl.gov, and a Berkeley BugZilla database.

3.6.2.1. Robustness and Known Issues

It is fairly easy to acquire a basic knowledge of UPC. Unlike MPI, UPC does not contain a large library of function calls to learn. However, understanding the idiosyncrasies of UPC, especially allocation and manipulation of shared variables and the concept of work sharing, takes practice and skill.

There are several major limitations we discovered in our investigation of UPC. UPC lends itself well to computational problems that can be modeled using a single global address space. If you were to diverge from this problem space, and, say, introduce pointer arithmetic, you would incur a fairly high performance penalty - the UPC translator inserts two function calls for every pointer dereference and modification. Those functions are inlined but each calls other functions that interact with the underlying GASNet layer.

Another limitation involves shared variables. Shared variables are always global in UPC and declared at the beginning of a program. The global nature of shared variables is an impediment to porting a large code to UPC. It will require computer scientists to forgo modularity in their code design for any variables that reside in shared memory.

The third limitation deals with porting code that is already threaded. Because all threads are alive from beginning of program, it is nontrivial to port code that is already threaded. Every piece of the code must be examined to determine if it should only be executed by thread 0, or if it should be executed in parallel.

Finally, porting existing MPI applications to a mixed MPI/UPC model will be especially tricky. All UPC threads and all MPI tasks are simultaneously alive throughout the life of the program. Fortunately, an excellent paper has been written outlining a successful process of MPI+UPC. We reference this paper in Section 3.6.6 below.

3.6.3. Tool Availability

There are versions of the Totalview debugger (7.0.1 or greater) that will debug UPC programs on x86 architectures. Otherwise, you can use gdb with partial debugging capability, or printf with best results.

UPC comes with *upc_trace,* a utility to help analyze communication behavior of your UPC programs. You must compiler your code with the "-g" option and then run your code with "–trace" to generate the trace file. You can then use *upc_trace* to post-analyze the results.

3.6.4. Performance

We compared UPC to OpenMP via a port to an LLNL OpenMP code called CLOMP. CLOMP is a benchmark code that mimics the computational kernels of a key laboratory code. John Gyllenhaal and Greg Bronevetsky developed the CLOMP code to measure OpenMP overhead. The code is relatively small, highly configurable via run-time parameters, and ideal for modeling overheads associated with thread parallelism. CLOMP was ported to UPC, and measurements were taken to quantify porting effort along with performance results for UPC-CLOMP.

The charts in this section are comparisons of OpenMP and UPC performance of the CLOMP code on the LLNL *hera* compute cluster for different input problems. Each node of *hera* has 4 Quad-core Opteron 8356 2.3 GHz CPUs for a total of 16 cores per node, and 32 GB of memory. All runs were performed on a dedicated, 'quiet' node.

OpenMP cases were compiled with the Intel 10.0 compilers with–O3 optimization. UPC cases were translated to C using the Berkeley 2.10.2 compiler/translator, and the Intel 10.0 compilers were then used to compile the translated code. All UPC cases used the PSHM (Process Shared Memory) GASNet layer for parallelization.

Speedup figures provided are relative to the OpenMP serial reference case for the given input problem. In Figure 3.6-1 andFigure 3.6-2, all OpenMP threads allocate their own memory. This 'intelligent allocation' strategy optimizes NUMA memory access patterns and mimics the performance of the OpenMP memory affinity patch being developed by B. deSupinski, M. Schulz, and A. Baker at LLNL.

Over a variety of problem sizes, the raw time to run parallel regions of the UPC port of CLOMP was faster than dynamic OpenMP scheduled loops but slower than manual or statically scheduled OpenMP loops when shared memory for the OpenMP cases was allocated by each worker thread. Combined with the generally slower serial regions of the code under UPC, the present UPC port of CLOMP is significantly slower than an OpenMP port with an intelligent allocation strategy.

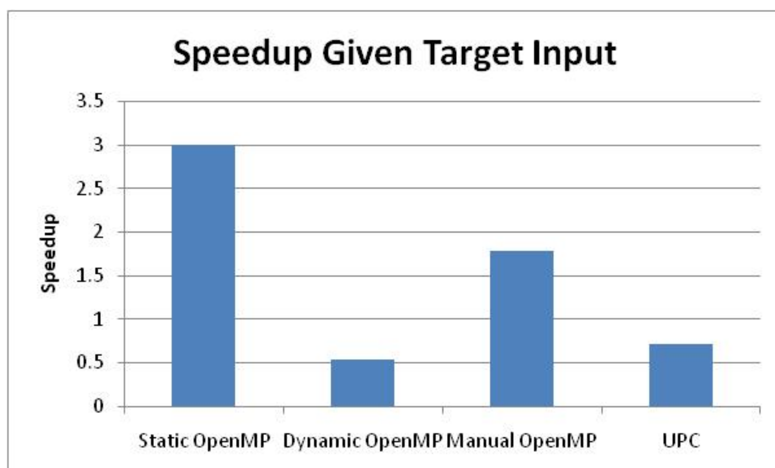


Figure 3.6-1 Speedup on 16-way hera node relative to OpenMP serial reference case for 'Target' input.

The ‘Target’ input case that was examined in Figure 3.6-1 is a small memory footprint problem (209k). There are 64 partitions and 100 zones per partition. This translates to 64 independent linked lists with 100 elements in each list.

Performance of all cases is low relative to peak speedup of 16 and the ‘Bestcase’ OpenMP speedup of 11.5. Bestcase speedup provides an upper bound on speedup but does not have adequate barriers to ensure correct answers. For this case and the cache-friendly input case (Figure 3.6-2), several OpenMP environment variables were set in order to increase performance of the Static and Dynamic OpenMP cases, at the expense of the Manual case (and the Bestcase)[4].

Note that UPC performance is significantly worse than the serial reference case, as seen by a speedup value less than 1.

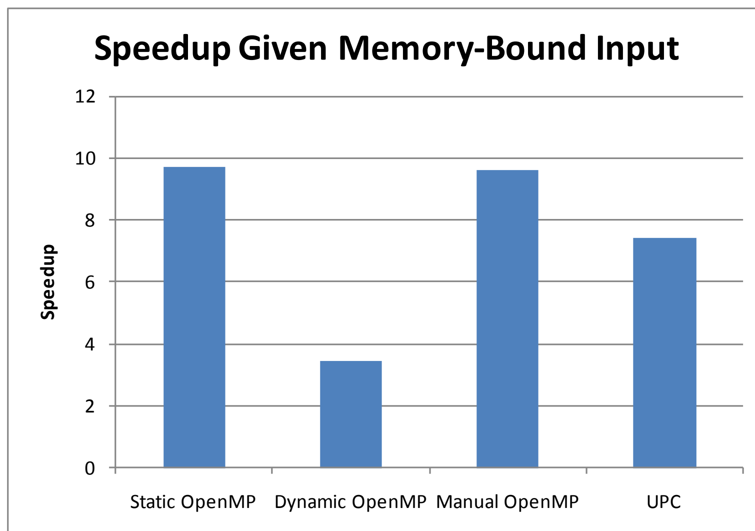


Figure 3.6-2 Speedup on 16-way hera node relative to serial reference case for memory-bound input.

Best results under UPC are seen with large-memory cases. This can be attributed to the lower relative impact of the code that is inserted to do pointer arithmetic on shared variables, which has a greater relative impact on small-memory cases or cases with a high flop/memory access ratio.

Figure 3.6-2 shows speedup for an input problem that is 328 MB in size. The larger memory footprint is achieved by increasing the length of each linked list to 10,000 zones. Speedup of all cases is closer to the Bestcase speedup of 14.8 than for smaller memory inputs. Despite performing better, the UPC case still lags in performance relative to the Static and Manual OpenMP cases.

[4] These environment variables settings are: KMP_BLOCKTIME=infinite, KMP_LIBRARY=turnaround, and KMP_AFFINITY=compact,0

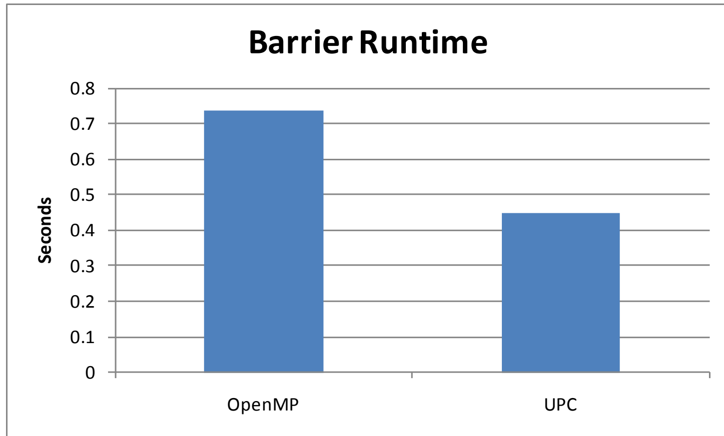


Figure 3.6-3 Runtime for Barrier loop.

As seen in Figure 3.6-3, UPC barriers are notably faster than OpenMP barriers for the compilers and platform tested.

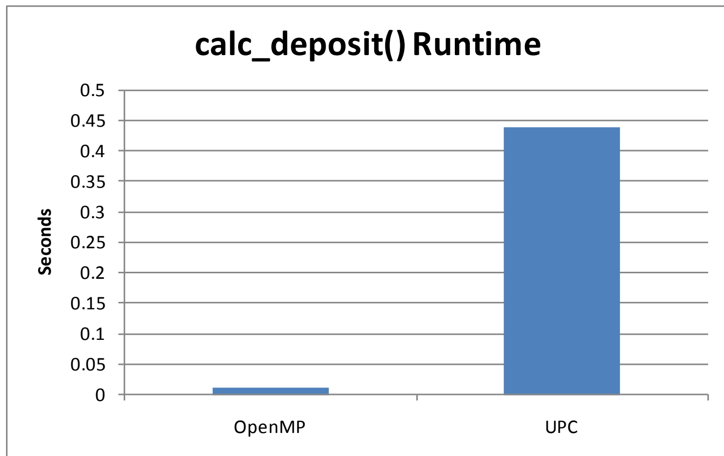


Figure 3.6-4 Runtime for calc_deposit() routine calls.

The calc_deposit() function emulates an MPI data exchange but does not do any actual communication. There are a large number of shared memory accesses that occur in this routine. Each shared memory access causes the UPC compiler to insert code. As a result, this function takes 35 to 36 times more time in UPC code than the same routine coded in straight C/OpenMP, regardless of the problem dimensions.

There is a small amount of actual work in the original calc_deposit routine, and there are many function calls inserted. The actual number of top-level functions that are inlined into the code is equivalent to:

13 + (3 * numParts)

where numParts is the number of partitions, which is equivalent to the number of independent linked lists. This accounts for the high number of code insertions and the poor performance under UPC.

For more results for the UPC- CLOMP port, please refer to *the 2010 Sept 29 Exascale Programming Model White Paper Final Report*, LLNL-TR 457671.

3.6.5. Suitability to LLNL Application Codes

A lack of tools such as stable debuggers made accurate effort estimation for the UPC port difficult. We believe that the time we spent porting CLOMP to UPC was greater than the time an application developer would have spent on the same code, especially if he or she was already familiar with it. However, the present nature of the UPC language requires that an entire application be modified if one small region is to be parallelized. This implies that the time to parallelize a large code with UPC will always be longer than the time to parallelize with OpenMP.

We recommend against the use of UPC as an alternative to OpenMP for intranode parallelism in LLNL scientific applications that use MPI for internode parallelism. If an application has constructs that make the use of global arrays preferable to the existing code design, application teams may still choose to port to UPC. However, if an application design would not benefit from a switch to global arrays, we believe that the complexity involved in the port and subsequent code maintenance does not justify the performance increase relative to OpenMP.

For more thorough analysis and conclusions, please refer to *the 2010 Sept 29 Exascale Programming Model White Paper Final Report*, LLNL-TR 457671.

3.6.6. Resources and Additional Information

The resources for UPC consist of a book written to explain the overall principles of the language, a number of tutorials given at Supercomputing, various academic papers, and a user group.

1. *UPC: Distributed Shared Memory Programming* by Tarek El-Ghazawi, William Carlson, Thomas Sterling and Katherine Yelick, 2005, John Wiley and Sons, Hoboken, New Jersey.
2. http://www.cug.org/1-conferences/CUG2010/pages/1-program/final_program/CUG10CD/CUG10_Proceedings/pages/authors/06-10Tuesday/9B-Alam-slides.pdf - *Evaluation of Productivity and Performance Characteristics of CCE, CAF and UPC Compilers*, by Sadaf Alam, William Sawyer, Tim Stitt, Neil Stringfellow, and Adrian Tineo. Excellent, current article given at CUG 2010.
3. http://www.prace-project.eu/documents/13_pgas_sa.pdf - *Productivity Analysis of Integrated Compilers for PGAS Languages* by Sadaf Alam at the PRACE (Partnership for Advanced Computing in Europe) Workshop “New Languages & Future Technology Prototypes”, March 1-2, 2010.
4. http://gac.udc.es/~gltaboada/papers/mallon_pvmmpi09.pdf - *Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures*, published Euro PVM/MPI 2009, Dami´an A. Mall´on1, Guillermo L. Taboada2, Carlos Teijeiro2, Juan Touri˜no, Basilio B. Fraguela2, Andr´es G´omez1, Ram´on Doallo2, and J. Carlos Mouri˜no. Excellent recent article on timings between these different approaches
5. http://www.mcs.anl.gov/uploads/cels/papers/hybrid.pdf - *Hybrid Parallel Programming with MPI and Unified Parallel C*, James Dinan, P. Sadayappan (Ohio State); Pavan Balaji, Ewing Lusk, Rajeev Thakur (Argonne). First real hybrid of MPI+UPC application with good results.
6. http://www.deepdyve.com/lp/ios-press/execution-model-of-three-parallel-languages-openmp-upc-and-caf-d765EZwe0J - *Execution Model of three parallel languages: OpenMP, UPC, CAF*, published ISO press 2005, Arni Marowka. Good article describing these three approaches, and the pros and cons.
7. http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0709.pdf - *Unified Parallel C - UPC on HPCx*, Ian Kirker and Adrian Jackson, January 14, 2008, HPCx Capability Computing. This document outlines the basic concepts of UPC, and explores what functionality is available on

HPCx. It then goes on to analyze the performance of UPC against IBM's MPI and LAPI on HPCx. Both IBM's UPC offering, and an open-source (Berkeley) UPC compiler are evaluated.

8. http://www.prace-project.eu/documents/d8-3-2.pdf - *Technical Report on the Evaluation of Promising Architectures* for. *Future multi-Petaflop/s Systems*. Authors: Ramnath Sai Sagar (BSC), Jesus Labarta(BSC), Aad van der Steen (NCF), Iris Christadler (LRZ), Herbert Huber (LRZ)
9. *Getting to Exascale: Applying Novel Parallel Programming Models to Lab Applications for the Next Generation Supercomputers,* E. Dube, L. Nau, C. Shereda, L. Harris, 2010, LLNL, LLNL-TR-457671

3.7. AMPI & Charm++

Owner / Development Location	Parallel Programming Laboratory, University of Illinois – Urbana-Champaign
Project Website	http://charm.cs.uiuc.edu/research/ampi http://charm.cs.uiuc.edu/research/charm
Download Page	http://charm.cs.uiuc.edu/software
Platforms Available	Windows, Linux, Solaris, Mac OS X

3.7.1. Overview

Charm++ and Adaptive MPI (AMPI) are two technologies originating from the Parallel Programming Laboratory (PPL) at the University of Illinois Urbana-Champaign. These two technologies have been in active development by the PPL group since the late 1980's. AMPI is an implementation of the MPI standard that virtualizes each processor rank, allowing for a migratable threads view of MPI. Charm++ is the underlying runtime system upon which AMPI is built. Charm++ allows for the virtualized threads to be migrated among unique physical processors, enabling improvements such as overlapping computation and communication within a program, as well as load balancing among the processors.

Adaptive MPI, or AMPI, is an implementation of the MPI standard that virtualizes the MPI processors in an application, where several virtual processors (VPs) may exist on a single physical processor (P). There are several benefits to this virtualization of the MPI ranks, including:

- Natural problem decomposition, unrestricted by the number of cores of the physical machine (Lawlor, Bhandarkar, & Kale, 2002) (Huang, Zheng, Kale, & Kumar, 2006).
- Load balancing across physical cores of the machine (Rodrigues, Navaux, Panetta, Fazenda, Mendes, & Kale, A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model, 2010) (Sarood & Kale, 2011).
- Improved debugger capabilities for specific problem sizes (Huang, Zheng, Kale, & Kumar, 2006).
- Checkpoint and Recovery features for fault tolerance (Lawlor, Bhandarkar, & Kale, 2002).
- Improved cache performance and spatial locality (Huang, Zheng, Kale, & Kumar, 2006).

There are several prominent limitations and barriers to adopting AMPI into even existing MPI code. Most notable are concerns related to memory usage per process, process migration overhead, and the determination of an ideal ratio between the number of virtual processes and the number of physical processors.

Charm++ is the underlying framework that provides the communication facilities, load balancing strategies, and threading model that AMPI relies on. "Charm++ is a parallel programming system based on a message-driven migratable-objects programming model" (Mei, et al., 2011). The Charm++ runtime is in many ways similar to an MPI implementation in which the focus is shifted to highly decomposed objects, driven by asynchronous communications. This system has enabled the development of several highly scalable parallel applications including:

- NAMD (bio-molecular simulation) (Mei, et al., 2011)
- ChaNGa (cosmological simulation) (Jetley, Gioachin, Mendes, Kale, & Quinn, 2008)
- BRAMS (weather and climate modeling) (Rodrigues, Navaux, Panetta, Fazenda, Mendes, & Kale, A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model,

2010) (Rodrigues, Navaux, Panetta, Mendes, & Kale, Optimizing an MPI Weather Forecasting Model via Processor Virtualization, 2010)
- Rocstar (rocket combustion simulation) (Jiao, et al., 2006)

As part of its design objectives, Charm++ provides built-in static and dynamic load balancing, node-aware communication optimizations, high portability to the vast majority of existing parallel platforms (Blue Gene L/P, Roadrunner, Cray XT, etc.), and scalability up to hundreds of thousands of processors (Mei, et al., 2011). More features are continually added, as requested by users.

3.7.2. Present State of Model

Charm++ provides a number of communications optimizations that directly benefit scalability for large-scale applications (Mei, et al., 2011) (Huang, Zheng, Kale, & Kumar, 2006). When an application is launched, the Charm++ runtime examines the topology of the hardware on which it is running in order to determine the optimal arrangement of the MPI processes onto nodes. This allows the Charm++ runtime to minimize the topological distance between nodes that communicate frequently, thus improving the performance of the application by reducing communication latency. Another Charm++ optimization is "application-guided node-aware multicast spanning tree" construction (Mei, et al., 2011). Such a tree allows the Charm++ runtime to optimally balance multicast messages across nodes, such that no single path becomes the bottleneck. Application level knowledge is used to build a spanning tree which places heavily loaded processors in the leaves of the tree, keeping them off of the critical path through the tree.

MPI Process Virtualization is at the heart of AMPI. This virtualization enables many features which are either difficult or impossible to implement in a standard MPI model. Load balancing across many thousands of cores can be a daunting task for an MPI programmer, as there are a limited number of methods for tuning the performance on a core. Processor and OS load balancing techniques commonly use Dynamic Voltage and Frequency Scaling (DVFS) controls to either speed up or slow down individual cores based on how they are performing relative to the system. The Intel Nehelem architecture, for example, introduced the TurboBoost feature which allows the processor to disable half of the processor cores in order to run the remaining cores at an overclocked rate. These controls come at the cost of either increased energy consumption and heat generation or decreased total system performance relative to some theoretical limits for speeding up or down respectively.

The AMPI/Charm++ approach is to over-decompose the problem space and assign several virtual MPI ranks to each processing core. For example, if your application performs 1 million calculations and you are allocated 10 physical processing cores, in a standard MPI code you might decompose the problem such that each core performs 100,000 calculations. However, if one core slows down, due to OS interactions, I/O processing, or any number of other concerns, the entire application will be limited by that slowest thread. In the AMPI case, with the same application and hardware allocations, you could over-decompose the problem space, creating 10 VPs (virtual processor threads) per physical core, each responsible for 10,000 of the calculations. Now if any VP on a physical core is stalled, waiting for I/O data or waiting at a synchronization barrier for instance, then the Charm++ runtime can perform a context switch to a new VP on the same core and continue processing. Another benefit provided by this virtualization is the ability to migrate VPs to other physical cores that are idling, allowing for a greater utilization of the processor cores. Context switching and process migration are good strategies if and only if the cost of the migration is small in comparison to the performance boost provided.

As an example, the cost of statically load balancing the application BRAMS every 600 timesteps is on the order of 10's of seconds (~15 seconds), while the execution time for each individual timestep is on the order of seconds. If we assume load balancing improves the execution time per timestep by 10% (~0.1 second) then the corresponding reduction in execution time per 600 timesteps is ~60 seconds minus the time to perform the balance, or ~45 seconds (Rodrigues, Navaux, Panetta, Mendes, & Kale, Optimizing an MPI Weather Forecasting Model via Processor Virtualization, 2010). While this static method can be used to provide some benefit, the Charm++ runtime provides dynamic load balancing techniques that make use of performance metrics to decide whether a balance invocation is worth the associated cost (Mei, et al., 2011).

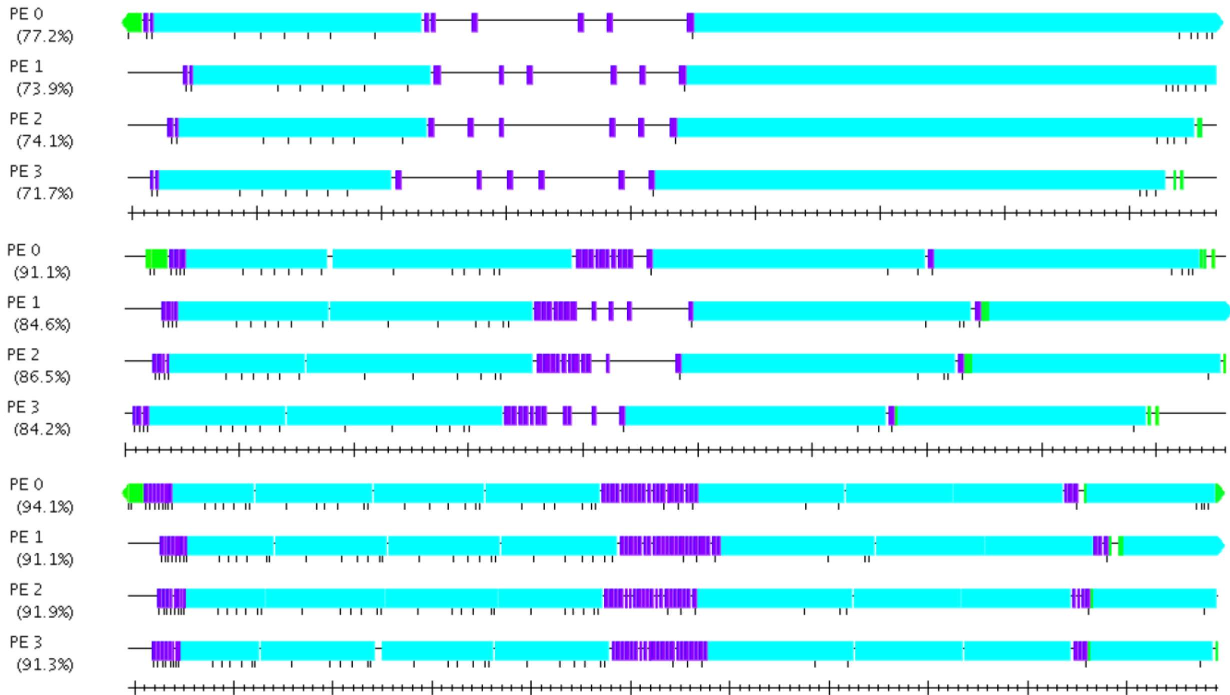


Figure 3.6-1: Load Balancing of Charm++ for 1, 2, and 4 VPs per processor

Figure 3.6-1 above shows the effect on utilization of multiple VPs per physical processor (Huang, Zheng, Kale, & Kumar, 2006). The light blue sections are the computational portions of the code, while the dark blue are areas that require communication blocking. When there is only a single VP on the processor (top), the communication blocks the processor while waiting for the response, leading to a decrease in the processor utilization (~74%). As the number of VPs is increased, the communication and computation sections of independent VPs can be interleaved, producing higher processor utilization (~92%).

AMPI is an implementation of the MPI standard which is MPI-1.1 compliant and partially supports MPI-2 (Mendes & Huang, 2007) (Zheng, 2011). This implies that it could be used in place of other currently available and utilized MPI implementations, provided they do not make use of the non-supported features in the language. This has enabled the PPL group to create an automatic Fortran 90 MPI to AMPI conversion tool, Photran (Negara, et al., 2010). This tool provides source-to-source transformation, allowing for the automatic handling of global variables as well as the packing and unpacking interfaces used during thread migration.

Currently, AMPI only supports a FORTRAN language binding, due solely to a lack of a request for more language bindings by the user base, though according to Gengbin Zheng, the project lead for AMPI, a C++ interface "should not be hard to implement" (Zheng, 2011).

3.7.2.1. Robustness and Known Issues

Process Virtualization is the core feature that Charm++ and AMPI provide to users. This Virtualization enables several key benefits related to debugging, load balancing, and the overlapping of computation and communication. Virtualization does, however, have a number of limitations associated with it.

The primary limitation is memory usage per core. Because several virtual processors reside on a single physical processor, the physical memory per physical core will be divided up among each of the VPs. Special care must therefore be taken when decomposing the application space to the virtual processors to avoid exceeding the physical memory limitations. If care is not taken, then the application will be required to spend a great deal of time swapping data between main memory and disk during the context switches which are at the core of the load balancing and virtualization methodologies employed by AMPI and Charm++.

Another limitation of the virtual processors methodology is the runtime overhead associated with managing all of the VPs and performing computations to decide if and when VPs should be migrated to other physical processors. The process of performing a thread migration between physical processors requires a set of Pack/Unpack subroutines be provided. These subroutines manage the transmission of the data structures associated with a migrating thread. AMPI provides a basic API for handling the migration of primitive data types, such as INTEGER and REAL as well as fixed-size one-dimensional arrays. However, AMPI does not automatically handle multi-dimensional arrays or FORTRAN allocatable arrays, and these must be manually handled in the Pack/Unpack subroutines (Negara, et al., 2010).

Photran provides source-to-source transformation from FORTRAN 90 MPI code and AMPI code. Due to the fact that AMPI is an implementation of the MPI standard, this is a fairly straightforward process; however, it requires some care be taken to correctly handle global and static variables. The process involves the privatization of global variables, giving each MPI process its own copy of the variables. Typically this is handled by the MPI implementation, as each MPI process is a separate OS-level process. Because of AMPI's multithreading approach, the user is responsible for handling this privatization. One technique suggested by the AMPI developers is to place all global variables into a single data structure and pass this object around between the subprograms (Negara, et al., 2010).

Finally, while AMPI is an implementation of the MPI standard, it is not yet fully compliant with MPI-2. New features are continually added to the implementation following user suggestions and specific requests for features. Dynamic process creation is not supported by the AMPI API because AMPI is able to migrate the VPs dynamically and create new AMPI ranks as needed (Zheng, 2011). In addition, whereas typical MPI implementations have several language bindings (FORTRAN, C, C++, etc), AMPI currently supports only a FORTRAN API.

3.7.3. Tool Availability

Photran is a tool developed by the PPL group to automatically transform FORTRAN 90 MPI programs to AMPI. This tool provides analysis features which allow for streamlining the privatization of global and static variables, as well as the creation of the Pack/Unpack subroutines required for the thread migration calls. Photran performance has been analyzed in detail, showing that the AMPI code produced performs similar to native MPI code on benchmarks (Negara, et al., 2010).

The BigSim Emulator was developed by the PPL group in order to facilitate the execution of any Charm++ or AMPI program on a specified number of processors while using some smaller number of physical processors. For example, the emulator allows users to run an MPI program on 100,000 emulated processors while using it on 2,000 physical processors. This supports testing and debugging capabilities which may arise from scaling up the size of the execution.

Due to the fact that Charm++ and AMPI are closely related to traditional MPI standards, the debugging features are very similar to what you would find for MPI debugging. Gdb can be used directly with Charm++ programs through use of the '++debug' or '++debug-nopause' when launching a Charm++ program with charmrun. This option works best only for local desktop debugging and would not be the preferred method for debugging on supercomputer clusters (Bhatale, 2011). Ideally the programmer would simplify execution to one or a few chares or VPs running on a single processor and attempt to recreate the error. When this is insufficient, parallel debuggers such as TotalView or charmdebug can be used to attach to the nodes of a running Charm application.

3.7.4. Performance

Most of the performance results that are currently published and available for Charm++ and NAMD apply to specific features that are being examined. Publications have been presented to highlight the load balancing, automatic source-to-source transformation, and scalability of programs written in Charm++ and/or AMPI. Scalability tests on the NAMD benchmark have shown efficient scaling to the full 224,076 cores of the Jaguar PF Cray XT5 at Oak Ridge National Laboratory, in addition to excellent scaling on the 65,536 cores of the Intrepid Blue Gene/P at Argonne National Laboratory (Mei, et al., 2011). This same test demonstrated the load balancing capabilities of the Charm++ runtime to minimize the difference between the maximum and average processor load during execution, thus resulting in higher utilization of the supercomputing resources.

Benchmark tests show that virtualization and load balancing techniques provided by Charm++ provide as much as a 32.5% reduction in execution time in the BRAMS weather modeling application (Rodrigues, Navaux, Panetta, Fazenda, Mendes, & Kale, A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model, 2010). These results are largely due to the natural imbalance that occurs in these types of simulations and may not be typical of all applications.

The results of the automatic MPI to AMPI transformation tool, Photran, show that the performance of the application code resulting from the transformation has similar performance characteristics to the results seen from traditional MPI implementations (Negara, et al., 2010). This transformation would be suitable as the starting point for converting an application from legacy MPI to the virtualized and asynchronous framework that is Charm++. As such a conversion would require a reworking of the parallel structure of the application, a transition through AMPI would provide the means to perform the conversion incrementally rather than requiring a complete rewrite.

3.7.5. Suitability to LLNL Application Codes

A transition from current FORTRAN MPI codes to AMPI would be a straightforward affair, particularly if the codes are limited in the number of global variables that are employed throughout the program. FORTRAN programs in particular have the opportunity to take advantage of Photran in order to assist in the analysis and transformation of code to AMPI.

A full conversion to Charm++ would be a more challenging endeavor. This would require a complete rewrite of the parallel programming structure and would be akin to similar rewrites for CUDA, OpenCL, Cilk+, or other parallel frameworks. Because Charm++ operates on a very asynchronous methodology, the conversion would require a different approach from those found in MPI programs, which typically require blocking the execution in order to perform message passing operations. In particular, there is the potential for applications that require frequent synchronizations to perform better on traditional MPI implementations due to overheads in the Charm++ runtime system.

One particular concern for LLNL code groups, which will become more prominent as we approach Exascale computing, is the limitation on the size of the memory footprint allowed per virtual processor. Due to the fact that a number of virtual processors may reside on a single physical processor, the memory on each physical processor will need to be divided up among each of the virtual processors in order to avoid costly (performance, energy, etc) accesses to disk. Another directly related concern is the trend towards decreasing physical memory per processing node in current and future supercomputers. Several techniques can be applied to mitigate this trend, allowing for the scalability results demonstrated in the NAMD benchmark tests (Mei, et al., 2011).

3.7.6. Bibliography

1. Bhatale, A. (2011, September 28). Personal Communication. (I. Lee, Interviewer)
2. Huang, C., Zheng, G., Kale, L., & Kumar, S. (2006). Performance Evaluation of Adaptive MPI. *PPoPP.* New York, New York, USA.
3. Jetley, P., Gioachin, F., Mendes, C., Kale, L. V., & Quinn, T. R. (2008). Massively parallel cosmological simulations with ChaNGa. *Proceedings of IEEE International Parallel and Distributed Processing Symposium.* Miami, FL, USA.
4. Jiao, X., Zheng, G., Alexander, P. A., Campbell, M. T., Lawlor, O. S., Norris, J., et al. (2006). A system integration framework for coupled multiphysics simulations. *Engineering with Computers*.
5. Lawlor, O., Bhandarkar, M., & Kale, L. (2002). *Adaptive MPI.* Urbana-Champaign: University of Illinois.
6. Mei, C., Sun, Y., Zheng, G., Bohm, E. J., Kale, L. V., Phillips, J. C., et al. (2011). Enabling and Scaling Biomolecular Simulations of 100 Million Atoms on Petascale Machines with a Multicore-optimized Message-driven Runtime. *SC.* Seattle, Washington, USA.
7. Mendes, C., & Huang, C. (2007). *AMPI: Adaptive MPI Tutorial.* University of Illinois Urbana-Champaign: Parallel Programming Laboratory.
8. Negara, S., Zheng, G., Pan, K.-C., Negara, N., Johnson, R. E., Kale, L. V., et al. (2010). Automatic MPI to AMPI Program Transformation using Photran. *PROPER.* Ischia, Naples, Italy.
9. Rodrigues, E. R., Navaux, P. O., Panetta, J., Fazenda, A., Mendes, C. L., & Kale, L. V. (2010). A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model. *SBAC-PAD.* Itaipava, Brazil.
10. Rodrigues, E. R., Navaux, P. O., Panetta, J., Mendes, C. L., & Kale, L. V. (2010). Optimizing an MPI Weather Forecasting Model via Processor Virtualization. *HiPC.*
11. Sarood, O., & Kale, L. V. (2011). A 'Cool' Load Balancer for Parallel Applications. *SC.* Seattle, Washington, USA.
12. Zheng, G. (2011 9-September). Personal Communication. (I. Lee, Interviewer)

3.8. OpenCL

Owner / Development Location	Khronos Group
Project Website	http://www.khronos.org/opencl/
Download Page	Different for each vendor – see resources section
Plaforms Available	Intel CPU; NVIDIA GPU; AMD CPU,GPU; IBM Cell,Power6,Power7

3.8.1. Overview

OpenCL (Open Computing Language) is a parallel programming open standard intended for use with heterogeneous computing systems. It is similar to CUDA, mentioned in Section 3.9, in that it is able to target graphics processing units (GPUs). However, OpenCL is more general-purpose than CUDA, with a goal to provide a standard language for computer scientists to write efficient, portable code for multi-core CPUs, GPUs, Cell-type architectures and other parallel processors.

Programs that utilize OpenCL consist of two parts, the traditional code (C/C++), and the OpenCL API, which enables the setup and control of execution *kernels* performing the computationally intensive work requiring parallelization. Kernels are written in a subset of the ISO C99 language that is compiled to target a particular computing device.

OpenCL supports both task and data parallel execution models, while CUDA is primarily focused on data parallelism. A kernel applies a single stream of instructions to vast quantities of data. Each piece of data is known as a work-item, and kernels can have a practically unlimited number of work-items. The more data crunched in a kernel, the better the performance. Kernels form the parallel unit of OpenCL, and they can be composed into a task graph[5] via asynchronous command queues. The computer scientist indicates dependencies between kernels, and what conditions must be met for a kernel to start execution. The OpenCL run-time layer can simultaneously execute independent kernels, thus extracting task parallelism within an application.

OpenCL treats all memory as being one of four distinct types. Global memory is available and fully read and write accessible to both the host CPU and all work-items on the OpenCL device. Constant Memory is read-only for work-items on the OpenCL device, but the host CPU has full read and write access. The next two memory types are not available to the host CPU. Private memory is accessible to a single work item for the OpenCL device for reads and writes and is similar to a register file. Local memory is accessible to a single work group for reads and writes, and is intended for shared variables and communication between work-items on the OpenCL device.

3.8.2. Present State of Model

While OpenCL is a relatively new standard, it is slowly maturing and being adopted in industry, providing some standardization between vendors and continues to be a desirable target for new parallel hardware. OpenCL was initially developed by Apple Inc., with collaboration from AMD, IBM, Intel, and NVIDIA (Kanter, 2010). The Khronos Group was pulled onto the team since they had successful experience working as the standards body with OpenGL. In June, 2008, the Khronos Compute Working Group was formed with representatives from CPU, GPU, embedded-processor, and software companies.

[5] A task graph is a graph in which each node represents a task to be performed. A directed arc from task 1 – to task 2 indicates that task 1 must complete before task 2 begins.

OpenCL 1.0 was released in November 2008, and the OpenCL 1.1 specification was released in June 2010. Implementations have been released by a variety of industry figures, including Apple, IBM, Intel, AMD, and NVIDIA. Each vendor supplies its own implementation conformant with the standard.

3.8.3. Robustness and Known Issues

Given that each vendor supplies its own implementation, it is difficult to gauge the overall robustness of OpenCL, since this varies from implementation to implementation. Those implementations which have been released are compliant, but some suffer from performance issues. For instance, the NVIDIA OpenCL implementation tends to underperform when compared to their CUDA software, for kernels solving the same problem. In general, the newness of the specification and releases leads to uncertainty regarding performance characteristics. However, the disparate releases could also be looked at as strength of the standard, since it is controlled by a neutral body and not solely reliant upon one vendor for support and performance. As OpenCL becomes more widely adopted, performance of individual implementations should improve.

Since OpenCL is built on a subset of ISO C99 there are a number of restrictions such as no recursion and limited pointers. These restrictions in the kernels can be limiting. Expanding OpenCL to handle these advanced features (similar to the way CUDA functionality has grown) would be beneficial.

3.8.4. Tool Availability

Debuggers exist for OpenCL, but tend to be vendor-specific and non-standard. AMD, for instance, has a tool, OpenCL Emulator-Debugger, which is used to compile and debug OpenCL kernels as C++ functions. NVIDIA has the CUDA-GDB debugger, but its support for OpenCL is unclear.

Additionally, NVIDIA provides the Visual Profiler performance analysis tool, which is suitable for use with OpenCL.

In general, besides built-in OpenCL timing and event functions, robust debugging and performance analysis tools are not yet present, especially for HPC Linux environments.

3.8.5. Performance

Remember that the main goal for OpenCL is programming portability, which does not necessarily equate with performance. Additionally, OpenCL implementations are at various stages of maturity, and therefore performance is dependent upon the hardware version, software version, and level of optimization conducted. In general, optimization techniques for a particular hardware architecture may not benefit other hardware architectures. That said, as implementations mature, performance should increase.

As use of OpenCL grows, more papers will be published discussing performance of the language. (Hamze, 2011)

3.8.6. Suitability to LLNL Application Codes

The main appeal for OpenCL is the fact that it is a parallel programming open standard intended for use with heterogeneous computing systems. LLNL uses diverse computing systems, and application codes are required to port to, and then perform on, these diverse systems, so OpenCL can be part of the parallel programming environment solution.

OpenCL is a shared-memory programming model, and therefore must be used in conjunction with another library such as MPI for inter-node parallelism. In some cases it may be a suitable substitute for

tasks currently parallelized using OpenMP work sharing. Data movement with respect to the various hardware devices is a chief concern, and one must be conscious of where in physical memory each buffer resides. Future hardware implementations may reduce the latency penalty of data movement as accelerator chips and memory are integrated directly into motherboards rather than as PCI cards, but at present, latency is a limiting factor. Codes must have a sufficient amount of floating-point work between data transfers to amortize the performance penalty associated with the transfers.

3.8.7. Resources and Additional Information

Main project page: http://www.khronos.org/opencl/
Wikipedia page: http://en.wikipedia.org/wiki/OpenCL
OpenCL 1.1 Specification: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf
NVIDIA OpenCL page: http://developer.nvidia.com/opencl
AMD OpenCL site: http://developer.amd.com/zones/openclzone/pages/default.aspx
OpenCL Blog: http://www.openclblog.com/
Intel OpenCL SDK: http://software.intel.com/en-us/articles/opencl-sdk/ (Intel will support OpenCL in future revisions of its Many Integrated Core (MIC) chips)
Additional information:

1. http://www.realworldtech.com/page.cfm?ArticleID=RWT120710035639&p=1 *Introduction to OpenCL, By: David Kanter, 12-07-2010*
2. http://www.gpucomputing.net/ - Excellent GPU Computing web site:
3. http://www.gpucomputing.net/?q=node/128 - OpenCL tutorial from this site
4. http://www.khronos.org/news/events/detail/amd-opencl-programming-webinars/ - OpenCL Programming Webinar Series
5. http://www.nvidia.com/content/GTC/documents/1409_GTC09.pdf - OpenCL introductory material, slides
6. http://www.nvidia.com/content/GTC/videos/GTC09-1409.mp4 - OpenCL introductory material, video
7. http://www.multicoreinfo.com/2009/08/parprog-part-9/ - Parallel Programming Tutorials Series, Part 9
8. http://developer.apple.com/mac/library/documentation/Performance/Conceptual/OpenCL_MacProgGuide/Introduction/Introduction.html - Apple Developer Resources (Reference Library)
9. http://developer.nvidia.com/object/gpu_computing_online.html - NVIDIA GPU Computing Resources – occasionally there will be OpenCL sessions
10. http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx - AMD Stream SDK/OpenCL Resources
11. http://developer.amd.com/documentation/videos/OpenCLTechnicalOverviewVideoSeries/Pages/default.aspx - AMD Stream SDK/OpenCL Resources Videos
12. http://www.khronos.org/registry/cl/ - Khronos Group OpenCL Web Site
13. http://www.youtube.com/user/khronosgroup - Khronos Group YouTube Videos (SIGGRAPH 2010)
14. http://enja.org/ - ENJ Tutorials
15. http://sa09.idav.ucdavis.edu/ - SIGGRAPH Asia 2009 Tutorial
16. http://arxiv.org/ftp/arxiv/papers/1005/1005.2581.pdf - *A Performance Comparison of CUDA and OpenCL*, Kamran Karimi, Neil G. Dickson, Firas Hamze, Cornell University Library, May 2011

Books:

1. *OpenCL Programming Guide*. Aaftab Mushi et. al: http://www.amazon.com/OpenCL-Programming-Guide-Aaftab-Munshi/dp/0321749642
2. *Heterogeneous Computing with OpenCL*. Benedict Gaster et. al: http://www.amazon.com/Heterogeneous-Computing-OpenCL-Benedict-Gaster/dp/0123877660/ref=sr_1_1?s=books&ie=UTF8&qid=1313520873&sr=1-1
3. *The OpenCL Programming Book*, Fixstars Corporation (Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, Satoshi Miki), 3/31/2010 http://www.fixstars.com/en/opencl/book/index.html

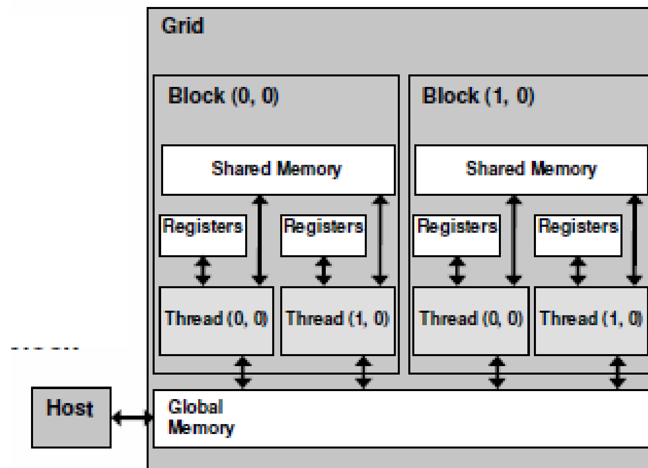
3.9. CUDA

Owner / Development Location	NVidia
Project Website	http://developer.nvidia.com/category/zone/cuda-zone
Download Page	http://developer.nvidia.com/cuda-downloads
Plaforms Available	NVIDIA (Windows, Linux, Mac OS)

3.9.1. Overview

CUDA, *Compute Unified Device Architecture*, is a programming model and instruction set architecture initially released in November 2006 by NVIDIA to allow for application developers to access the GPUs (Graphical Processing Units) without having to use the graphics application programming interface. CUDA comes with a software environment that supports C and C++, along with Fortran, OpenCL, and DirectCompute. Additionally, you can get compilers for Python (PyCUDA) and Java (JCUDA).

The core concepts for CUDA revolve around three key abstractions: a hierarchy of thread groups (think tree structure), shared memories, and barrier synchronization. These abstractions are accessible to the programmer through a set of language constructs. The result is fine-grained data-parallelism/thread parallelism nested within coarse-grained data parallelism/task parallelism – and this coarse-grained data parallelism/task parallelism is solved independently by blocks of threads. The programmer must think about data parallelism and the effects of threading when considering how to partition his problem. At present, data cannot be shared between GPUs, so a task is limited in size by the amount of memory on a single GPU.



[6]Figure 3.9-1 Diagram of CUDA GPU Memory layout

Programming in CUDA requires that the programmer pay careful attention to where entities are going to reside or need to be accessible from – i.e., will this data/function reside solely on the Host (CPU) or will it go/be accessible to the Device (GPU), and if the Device, which memory will be used (global, shared, private, etc.).

[6]*VolumeI_CUDA_Intro.pdf*, http://developer.nvidia.com/object/cuda_training.html, pg 21

3.9.2. Present State of Model

CUDA has been in existence since 2006 and is supported by a single vendor, NVIDIA, so the system has had time to mature. The learning resources available are thorough and robust. The CUDA developer web site is well maintained and contains useful examples and documentation. Some examples are already partially coded and only requiring filling in specific parts to get started. The examples progress from easy to more challenging and so guide the learner as his skill increases. The annual GPU Technology Conference is an excellent venue for developers to collaborate with others using CUDA, and explore new potential tools, APIs, and hardware configurations.

The current software release of CUDA is version 4.0. The software system has reached a point of reasonable maturity and continues to be widely adopted. Version 4.0 included easier support for multiple GPUs, as well as a Unified Virtual Address space between CPUs and GPUs on the same node. CUDA is mature enough for production codes to explore its feasibility since many of the previous difficulties have been alleviated.

3.9.3. Robustness and Known Issues

As mentioned, CUDA continues to see widespread adoption by a growing scientific community and continued development from NVIDIA. It is very effective when using NVIDIA GPUs, but cannot be used for other hardware architectures at the moment. That is, it is a vendor-specific standard and useful only for NVIDIA products.

Another challenge is in getting code to work in an HPC environment where jobs must be submitted to a queue, such as the edgelet system at LLNL. Most of the learning examples assume a user has a single workstation personally available to him, and that code will be run locally. Working in an HPC environment generally has two differences: loading the CUDA module into the working environment, and submitting jobs to a queue to be executed.

3.9.4. Tool Availability

NVIDIA now supports the CUDA-GDB debugger for Linux, and its own proprietary Visual Studio plugin for Windows. The former tool is very similar to traditional gdb and can be used in a similar fashion.

Rogue Wave Software offers Totalview for CUDA as an add-on feature and supports CUDA running directly on Tesla or Fermi hardware.

The Visual Profiler from NVIDIA is an excellent performance analysis tool which is supported on Linux, Mac OS, and Windows. It uses hardware counters on the GPU to assess performance and possible bottlenecks encountered during execution.

3.9.5. Performance

For well-behaved problems, CUDA performance is promising and facilitates easy exploitation of the underlying GPU hardware. Optimizations are performed by the *nvcc* compiler which attempts to make the CUDA code result in as high performance as possible. In this sense the software architecture is very mature for delivering high performance, assuming the problem can be effectively expressed in CUDA. Many advanced optimizations can be taken, and indeed are required to obtain peak performance. These optimizations can differ with new hardware architectures and therefore might not be effective in the long-term. The optimizations require knowledge of the underlying hardware being used for development.

The performance of CUDA code is further limited by the high latency penalty of data transfers to and from the GPU, which is connected to the motherboard through the PCI bus. As the technology matures, GPUs are expected to integrate directly into the motherboard and this latency penalty will decrease.

Another ongoing issue with heterogeneous programming is the challenge in effectively utilizing all of the available hardware. For instance, in a dual-socket, hex-core system with 2 GPUs, ideally an application would keep all 12 CPU cores busy while providing a full workload to both GPUs. In practice with a typical HPC code, this is extremely difficult to implement. As the number of cores in traditional CPUs increases, designs that directly integrate multiple GPUs onto the motherboard will be necessary to provide performance competitive with many-core CPUs.

Section 3.9.7 references several papers and talks on performance studies where one can learn more about CUDA performance.

3.9.6. Suitability to LLNL Application Codes

LLNL's IRS benchmark was ported to the GPU using CUDA as a part of a project. The process was reasonably straightforward, since several compute-intensive functions were obvious choices for GPU kernels. Once the proper data was identified to perform the computation, it was a matter of deciding when and how to move that data to/from the GPU. With CUDA 4.0, unified virtual addressing allows CPU-resident pointers to be dereferenced on the GPU. However, the performance implications of this are unknown and undocumented. Empirically, explicit memory copies are faster than automatic transfers.

A major hurdle to high performance was the need to move data back to CPU physical memory multiple times per iteration in order to facilitate ghost cell communication. The paradigm of keeping data on the GPU as long as possible is not easy to implement in this sense. Additionally, the use of doubly indirect pointers in the CPU code is problematic for porting to the GPU. Expressing such complex pointers in GPU space is cumbersome and error prone. It is possible on the GPU, but cannot be expressed elegantly. A preferable system would be flat memory arrays with specific or easily computed offsets.

Additionally, the standard of assuming one processing core per domain is not necessarily easily translated to the GPU. The GPU is more easily exploited with one large domain, rather than using complex memory layouts. In short, depending on the memory layout and inter-node communication requirements of the code, GPU acceleration may or may not be low hanging fruit.

3.9.7. Resources and Additional Information

Main website: http://developer.nvidia.com/category/zone/cuda-zone
Wikipedia page: http://en.wikipedia.org/wiki/CUDA
CUDA Training website: http://developer.nvidia.com/cuda-education-training
CUDA webinars: http://developer.nvidia.com/gpu-computing-webinars
http://developer.download.nvidia.com/CUDA/training/CUDA_webinars_multi_gpu.pdf

Totalview for CUDA page: http://www.roguewave.com/products/totalview-family/totalview/cuda.aspx

Papers:

1. http://www.oerc.ox.ac.uk/personal-pages/gihan/publications/p23-pennycook.pdf ; *Performance Analysis of a Hybrid MPI/CUDA Implementation of the NASLU Benchmark*, S.J.

Pennycook, S.D. Hammond, S.A. Jarvis, G.R. Mudalige, General-Purpose Computation on Graphics Hardware, November 16th, 2010

2. http://www.oerc.ox.ac.uk/research/hpc-na/workshop-3-programme/HPC-NA%203%20Fatica.pdf ; *Cuda for High Performance Computing*, Massimiliano Fatica, HPC-NA Workshop, January 2009
3. http://www.macresearch.org/cuda-quick-look-and-comparison-fft-performance ; *CUDA: A Quick Look and Comparison of FFT Performance*; dgohara, Mac Research, May 23 2008
4. http://www.cs.wisc.edu/techreports/2011/TR1693.pdf ; *Porting CMP Benchmarks to GPUs*; Matthew D. Sinclair, Henry Duwe, Karthikeyan Sankaralingam; The University of Wisconsin-Madison; June 2011
5. http://www.hpcadvisorycouncil.com/pdf/NAMD_GPU_Analysis_and_Profiling_NVIDIA.pdf ; *NAMD GPU Performance Benchmark*; HPC Advisory Council; March 2011
6. http://www.pgroup.com/lit/articles/insider/v2n3a2.htm ; Porting the SPEC Benchmark BWAVES to GPUs with CUDA Fortran; Greg Ruetsch and Massimiliano Fatica; PGI Insider; September 2010

Books

1. *Programming Massively Parallel Processors: A Hands-on Approach*. David B. Kirk Wen-mei W. Hwu: http://www.amazon.com/dp/0123814723/ref=asc_df_01238147231668764?smid=ATVPDKIKX0DER&tag=hyprod-20&linkCode=asn&creative=395093&creativeASIN=0123814723
2. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Jason Sanders, Edward Kandrot: http://www.amazon.com/CUDA-Example-Introduction-General-Purpose-Programming/dp/0131387685/ref=sr_1_1?s=books&ie=UTF8&qid=1313532228&sr=1-1
3. *GPU Computing Gems Emerald Edition*. Wen-mei W. Hwu: http://www.amazon.com/GPU-Computing-Gems-Emerald-Applications/dp/0123849888/ref=sr_1_1?s=books&ie=UTF8&qid=1313532269&sr=1-1

4. Recommendations

We recommend the following regarding the programming models in this survey:

- **Chapel and X10:** Monitor and further investigate both of these as PGAS alternatives to the present de facto programming model using MPI. Of the two, focus more resources on Chapel. However, do not neglect developments in X10.

  We recommend at least one LLNL team port a nontrivial benchmark application to Chapel and then share the results of this port in a paper and presentation.

- **Fortress and UPC:** Monitor ongoing developments but do not explicity dedicate resources to these languages. Fortress is likely defunct unless Oracle decides to revive it or another group picks it up. See our 2010 report, 'Getting to Exascale: Applying Novel Parallel Programming Models to Lab Applications for the Next Generation of Supercomputers' for further information on why we do not believe UPC in its present form is a candidate PGAS language for LLNL (Dube, Nau, & Shereda, 2010).

- **Cilk Plus:** Cilk Plus is more relevant for first-time parallel programmers and is unlikely to be of interest to LLNL code teams because of its limitations and performance issues. The same person responsible for PBB should be capable of staying abreast of Cilk Plus developments without expending much additional effort.

- **Intel Parallel Building Blocks:** PBB is a viable candidate for multicore heterogeneous computing with solid prospects for improvement, especially on Intel machines. While current utilization of existing PBB libraries on LLNL clusters seems to be low and the data duplication requirement of ArBB is problematic for exascale, it is nonetheless our recommendation that LLNL build and maintain expertise in this toolset and keep recent versions of Threaded Building Blocks and Array Building Blocks installed and available to users.

- **OpenCL and CUDA:** Participate in CUDA and OpenCL advancement where possible. Monitor closely and maintain onsite expertise. Code teams should be cautious of developing a codebase, however, especially in CUDA. Processor and model advancement may make the CUDA language obsolete, so using OpenCL is preferable. OpenCL's performance lag with CUDA may require the use of CUDA in some instances, though.

- **AMPI and Charm++**: Charm++ and AMPI are already in use by at least one application code at LLNL – NAMD. Charm++'s process virtualization might provide a component to a strategy for dealing with very low MTBF (Mean Times Between Failure) on exascale systems – namely, the need for process migration during hardware failure in order to continue a run. It may also be of value for load balancing, although significant gains obtained from AMPI over standard MPI for regular, balanced LLNL applications would be somewhat surprising and would warrant an examination of the system software stack. We recommend that LLNL's MPI support staff become more familiar with Charm++ and AMPI to understand whether some of its architectural components could be of benefit in helping standard MPI make the transition to exascale. We do not recommend that application teams take any immediate action related to AMPI or Charm++.

- **Models mentioned in the appendix:** Monitor these models for any increase in adoption and acceptance. In the case of OpenMP advancement, several key LLNL staff (Bronis de Supinski, Martin Shulz) are closely involved in the OpenMP Architecture Review Board and will stay abreast of ongoing developments. de Supinski is the Chair of the Language Committee and represents LLNL as an auxiliary member of the ARB.

Appendix A. Additional Exascale Programming Models

The following models were not examined in detail. In the case of Titanium and Global Arrays, development and papers appear to have tapered off by 2007, although the Global Arrays website lists a 2009 Gordon Bell Nomination. ParalleX appears to be a newcomer and thus little is written about this neophyte. Perhaps this parallel execution model is worth watching.

Advances in OpenMP and domain-specific languages (DSLs) may provide opportunities for LLNL application development teams to enhance codes for exascale development and should be monitored. This is especially true for OpenMP advancement.

Coarray Fortran

Owner / Development Location	ISO Fortran Standards Committee (one version); Rice University (Coarray Fortran 2.0)
Project Website	http://gcc.gnu.org/wiki/Coarray (Best Found) http://www.nesc.ac.uk/talks/892/09-45_John_Reid_hpcx.pdf (Viewgraphs Overview) ftp://ftp.nag.co.uk/sc22wg5/N1751-N1800/N1787.pdf (Document Overview) ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1830.pdf (International Standard) http://caf.rice.edu (Rice version)
Download Page	ISO version downloads available with Fortran 2008 compilers. Rice version at http://www.hipersoft.rice.edu/caf/download.html and requires installation of GASNet.
Platforms Available	Any platform that has a Fortran 2008-compliant compiler will include Coarray Fortran extensions.
Description	Coarray Fortran (CAF) was created by Robert Numrich and John Reid and is an extension of Fortran 95/2003. A CAF program is SPMD – single program, multiple data. Each task runs its own set of data, and this task is called an *image*. The array syntax of Fortran is extended with additional subscripts to provide a representation of references to data that is spread across images. A group at Rice University is pursuing an alternate version of coarray extensions for the Fortran language. They don't believe that the set of extensions agreed upon by the committee are the right ones. In their view, both Numrich and Reid's original design and the coarray extensions proposed for Fortran 2008, suffer from a number of shortcomings. A user of coarray Fortran needs to be aware of these two versions when deciding on which path to pursue.

Titanium

Owner / Development	Computer Science Division, University of California at Berkeley

Location	
Project Website	http://titanium.cs.berkeley.edu/
Download Page	http://titanium.cs.berkeley.edu/ - click on Software button
Platforms Available	UC Berkeley EECS systems, Linux x86 systems
Description	Titanium is an explicitly parallel dialect of Java developed at UC Berkeley to support high-performance scientific computing on massively parallel supercomputers and distributed-memory clusters with one or more processors per node. The language provides a global memory space abstraction, using a distributed memory back-ends implemented with GASNet. Note that most research looks like it ends in 2007 time frame.

Global Arrays

Owner / Development Location	Pacific Northwest National Laboratory
Project Website	http://www.emsl.pnl.gov/docs/global/ (Main Website) http://acts.nersc.gov/ga/index.html
Download Page	http://www.emsl.pnl.gov/docs/global/download.shtml
Platforms Available	BlueGene, Cray, Linux Clusters, Mac, SGI Altix, Solaris, Windows, NEC, Fujitsu
Description	The Global Arrays (GA) toolkit is a library for writing parallel programs that use large arrays distributed across processing nodes. This library is a portable "shared-memory" programming interface for distributed-memory computers, with Fortran, C and C++, and python wrapper interfaces. Originally developed to support arrays as vectors and matrices (one or two dimensions), it now supports up to seven dimensions in Fortran and even more in C. GA offers two types of operations: collective operations and local operations.

ParalleX and High Performance ParalleX (HPX)

Owner / Development Location	Louisiana State University – Stellar Group
Project Website	http://stellar.cct.lsu.edu/ (Main Website) http://stellar.cct.lsu.edu/info/publications/ (Publications) http://www.cs.virginia.edu/~skadron/cs793_s07/paralleX.pdf (Overview Document) http://arxiv.org/PS_cache/arxiv/pdf/1109/1109.5201v1.pdf (Example Document)
Download Page	http://stellar.cct.lsu.edu/downloads/
Platforms Available	x86-64 Linux and x86-64 Windows 7+. Gilgamesh II is mentioned in the Overview paper cited above.
Description	The ParalleX execution model is based on message-driven flow control in a global address space coordinated by lightweight synchronization semantic constructs. The key features of ParalleX that are advertised to provide advantages over other programming models include message driven computation based on light weight synchronization using local control objects, data flow, and fine grain multithreading.

Domain-Specific Languages

Owner / Development Location	Various
Project Website	Below are a collection of websites that explore writing domain specific languages, along with the Wikipedia website which lists a number of actual domain specific websites to check out. http://en.wikipedia.org/wiki/Domain-specific_language http://en.wikipedia.org/wiki/Language-oriented_programming http://hrcak.srce.hr/cit_ojs/index.php/CIT/article/view/1465/1169 http://c2.com/cgi/wiki?DomainSpecificLanguage http://groovy.codehaus.org/Writing+Domain-Specific+Languages http://dsl-engineering.org/
Download Page	
Platforms Available	
Description	In software development and domain engineering, a domain-specific language (DSL) is a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique.

OpenMP advancement

Owner / Development Location	OpenMP Architecture Review Board (or just “ARB”) is a non-profit corporation that owns the OpenMP Brand
Project Website	http://openmp.org/wp/
Download Page	The website below will help you figure out where to go to from the vendors perspective. http://openmp.org/wp/openmp-compilers/
Platforms Available	All Unix platforms and Windows NT platforms
Description	The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran. OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.