ORNL/TM-2012/85

COMPOSE-HPC: A TRANSFORMATIONAL APPROACH TO EXASCALE

March 5, 2012

Prepared by
David E. Bernholdt
Benjamin A. Allan
Robert C. Armstrong
Daniel Chavarría-Miranda
Tamara L. Dahlgren
Wael R. Elwasif
Tom Epperly
Samantha S. Foley
Geoffrey C. Hulette
Sriram Krishnamoorthy
Adrian Prantl
Ajay Panyala
Matthew Sottile

DOCUMENT AVAILABILITY

Reports produced after January 1, 1996, are generally available free via the U.S. Department of Energy (DOE) Information Bridge:

Web Site: http://www.osti.gov/bridge

Reports produced before January 1, 1996, may be purchased by members of the public from the following source:

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Telephone: 703-605-6000 (1-800-553-6847)
TDD: 703-487-4639
Fax: 703-605-6900
E-mail: info@ntis.fedworld.gov
Web site: http://www.ntis.gov/support/ordernowabout.htm

Reoports are avilable to DOE employees, DOE contractors, Energy Technology Data Exchange (ETDE), and International Nuclear Inforamtion System (INIS) representatives from the following sources:

Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831
Telephone: 865-576-8401
Fax: 865-576-5728
E-mail: reports@adonis.osti.gov
Web site: http://www.osti.gov/contact.html

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States nor any agency thereof, nor any of their employees, makes any warranty, express 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 any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL/TM-2012/85

COMPOSE-HPC: A TRANSFORMATIONAL APPROACH TO EXASCALE

David E. Bernholdt[1]
Benjamin A. Allan[2]
Robert C. Armstrong[2]
Daniel Chavarría-Miranda[3]
Tamara L. Dahlgren[4]
Wael R. Elwasif[1]
Tom Epperly[4]
Samantha S. Foley[1]
Geoffrey C. Hulette[2]
Sriram Krishnamoorthy[3]
Adrian Prantl[4]
Ajay Panyala[5,3,1]
Matthew Sottile[6]

[1] Oak Ridge National Laboratory
[2] Sandia National Laboratories
[3] Pacific Northwest National Laboratory
[4] Lawrence Livermore National Laboratory
[5] Louisiana State University
[6] Galois, Inc.

Date Published: March 5, 2012

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831
managed by
UT-Battelle, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

ABSTRACT

The goal of the COMPOSE-HPC project is to “democratize” tools for automatic transformation of program source code so that it becomes tractable for the developers of scientific applications to create and use their own transformations reliably and safely. This paper describes our approach to this challenge, the creation of the KNOT tool chain, which includes tools for the creation of annotation languages to control the transformations (PAUL), to perform the transformations (ROTE), and optimization and code generation (BRAID), which can be used individually and in combination. We also provide examples of current and future uses of the KNOT tools, which include transforming code to use different programming models and environments, providing tests that can be used to detect errors in software or its execution, as well as composition of software written in different programming languages, or with different threading patterns.

Keywords: Productivity Tools, Evolutionary Programming Models

1. INTRODUCTION

High-performance computational science and engineering (CSE) developers are accustomed to transforming and refactoring their software to maintain and enhance it. Reasons include:

- Porting codes to new programming models, libraries, or platforms;
- Updating calls to a library or utility routine used pervasively throughout a code;
- Changing data structures to improve performance, such as switching between an array of structures (derived types) and a structure containing arrays as members; and
- Expressing platform-specific optimizations as transformations of code, applied at build time, as a more sustainable alternative to trying to maintain multiple distinct versions of the code.

Such transformations are typically done manually, in a text editor. On occasion, tools such as sed or scripts written in perl or python might be used to help automate the process. However, these approaches are tedious and error-prone. The possibilities for error or oversight by a person working in an editor are obvious. In addition, text-based transformation tools are not aware of programming language syntax; can easily be "fooled" by code that violates (often in trivial ways) the programmer's assumptions; and are hard to apply selectively. On the other hand, source-to-source transformation tools, such as ROSE [1], are syntax-aware so can perform such transformations rigorously, but require a deep understanding of compiler operations and programming experience beyond the majority of CSE software developers.

The COMPOSE-HPC project seeks to address these very real and pervasive challenges through a tool chain for simplifying the development and use of source-to-source (s2s) transformation, facilitating its adoption as part of the CSE software developer's tool box. The rest of this paper is organized as follows: We provide a motivating example of what we're trying to accomplish, in the context of the NWChem parallel computational chemistry package [2] (Sec. 2), and then describe the KNOT tool chain and its tools: PAUL, ROTE, and BRAID (Sec. 3). We then summarize a number of other applications of the KNOT tool chain we are working on (Sec. 4) and close with a discussion of our experience to date and future plans (Sec. 5).

2. A MOTIVATING EXAMPLE

The NWChem parallel computational chemistry package [2] makes extensive use of the Global Array Toolkit (GA) [3, 4], a library-based partitioned global address space programming model that was co-designed with NWChem. A common idiom in GA programming is dynamic load balancing of irregular tasks across available processes through the use of a shared counter. However, as process counts increase, so does contention for the shared counter, which adversely impacts scalability. Moreover, the irregularity of the tasks made it simpler to ignore locality considerations (which tend to *increase* the imbalance). This was generally considered acceptable as long as the cost of the computation scaled significantly higher than communication (i.e., n^4 computation vs. n^2 communication). However this is clearly not going to be satisfactory for exascale, where data movement costs, in terms of both time and power, will increasingly dominate computation costs.

The recent development of the TASCEL [5, 6] task pool environment offers an alternative approach to the dynamic load balancing problem through work stealing, as illustrated in Fig. 1. Conversion of a code from the GA shared counter idiom to the TASCEL work pool idiom is a significant change, requiring a series of transformations which must be implemented carefully and correctly throughout a fairly significant chunk of code. Though the semantics of TASCEL are relatively straightforward, its application in the original SCF code is both integral and ubiquitous, making it ideal for annotation-based code transformations. Added benefits of expressing annotations as structured comments within the original code include:

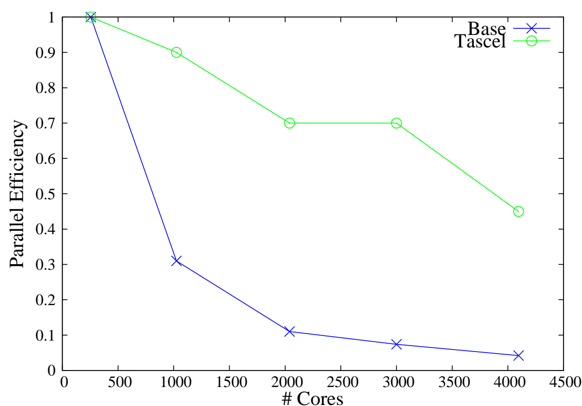

Figure 1. Parallel efficiency for the base (GA) and transformed (TASCEL) versions of the SCF mini-application simulating an array of beryllium atoms.

- exposing and making transformation points explicit;
- preserving the original code for production use; and
- facilitating comparisons against the new code.

To illustrate the application more concretely, we focus on the Self-Consistent Field (SCF) [7] module of the NWChem package, and more specifically on the contribution of the two-electron integrals to the Fock matrix, which is the most computationally intensive portion of the SCF algorithm. Because of the current capabilities of the KNOT tools, we piloted the transformations using a somewhat simplified, stand alone SCF mini-application implemented in C. We are now beginning to work with the full Fortran code within the actual NWChem package.

The two-electron contribution to the Fock matrix involves a computationally sparse n^4 calculation over an n^2 data space. Most of the n^4 tasks do not make any significant contribution to the matrix ($<1\%$ for larger problems), but all must be enumerated and at least partly evaluated to know which are significant. In the GA idiom, a shared counter is used to track the task identifiers from the n^4 space as they are handed out to processes ready for work regardless of the locality of the associated data. Each task gets the (non-local) data required to evaluate the significance of the contribution (two tiles each from the "Schwarz" and density matrices). Exceeding the threshold results in a task proceeding with the four-deep loop nest to evaluate the relevant block of two-electron integrals and accumulate their contributions to the Fock matrix.

A number of transformations are required to shift the code to use the TASCEL programming model and optimize it:

1. Change the task enumeration loop so that all processes enumerate all n^4 tasks.
2. Filter out, on a process basis, all tasks in which the first Schwarz tile is non-local (to enhance locality).
3. Evaluate the significance of the remaining "local" tasks.
4. Insert significant tasks into a TASCEL task pool, which will be executed with dynamic load balancing via work stealing.
5. Record which process tasks, from the original n^4 space, are actually executed. This can be used to seed the task pools for subsequent iterations of the SCF algorithm, thus avoiding the expense of global task enumeration and filtering.

The clearly systematic transformations span a significant code base (over 2100 lines spread over 7 files), resulting in about a 40% increase in size. A similar set of transformations could be applied to other occurrences of the GA shared counter idiom elsewhere in the approximately 4.5M lines of NWChem code to achieve similar effects.

3. THE KNOT TOOL CHAIN

KNOT is the overarching name given to the suite of tools we are developing. These tools are intended to provide selected, general, cross-cutting transformation capabilities, such as language interoperability and automated contract enforcement, as well as provide a basis for CSE software developers to create their own transformation tools. Custom tools can be developed to answer software engineering problems related to the increasing rate-of-change of HPC architectures, allowing scientific programmers to modify, rearrange and otherwise rewrite their code in an automated, componentized, and deterministic way. This approach enables a lighter-weight, "pay for what you use" strategy, facilitating the development of small tools addressing specific needs of applications and their developers, while providing sufficient structure to allow the tools to be combined and shared with others.

As depicted in Fig. 2, the three main components of the KNOT tool chain are PAUL, ROTE and BRAID. PAUL is a parser front-end that understands and renders annotations in the scientist's code. Annotations identify particular contexts and blocks of transformable code. The scientist is free to use existing annotations drawn from other sources or create new annotations. Annotations may be bound to particular rewrite rules that ROTE uses to perform the actual transformations. BRAID enables rewrite rules to generate code in multiple languages. It uses both PAUL and ROTE to create a language-independent intermediate representation for complete two-way language interoperability.

3.1 PAUL: USING SOURCE CODE ANNOTATIONS TO CONTROL TRANSFORMATIONS

One of the key challenges to developing automated transformation tools for large software projects is targeted, end-user control since transformations often require information about their context. The PAUL (Parser for Annotations in a User-defined Language) component of KNOT is based on the use of comments, embedded in the original code, to explicitly identify transformation locations and provide context. Preserving information about transformations directly in the code also provides documentation and mitigates the risk that transformation information will diverge from the source code over time.

Annotations in PAUL take the form of structured comments. For example,[1]

```
/* %TAG formatted-string */
```

[1] We use multi-line C comments here. PAUL also supports single line C and C++ comments, as well as F77 and F90 comments in Fortran.

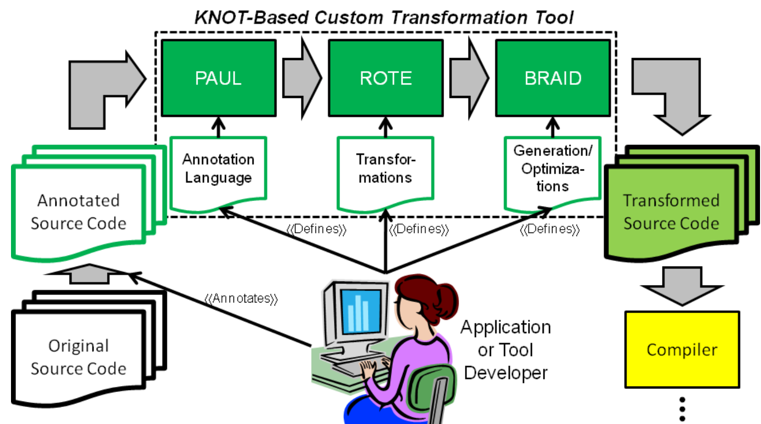

Figure 2. A schematic depiction of the process through which custom transformation tools are created and used within the KNOT tool chain.

The % introduces the annotation, TAG identifies the particular annotation language to be used, and the remaining formatted-string is parsed according to the definition of the indicated annotation language.

Transformation annotations can serve many purposes. They can be used to allow a programmer to identify specific data structures to be transformed, leaving other data structures within the program alone. Annotations can also be used to indicate which representation is most appropriate at different program locations to allow code to be generated to map between two representations. For example, one of the transformations investigated early in the project involved the conversion of data structures and associated access code from an array-of-structs to a struct-of-arrays form, which is a common change to improve cache utilization of code that has an impact on locality of access.

PAUL is implemented as a thin layer over the existing ROSE compiler infrastructure. PAUL provides the ROSE traversals that can be invoked to search for structured comments, parse them, and decorate the appropriate nodes of the abstract syntax tree (AST) representing the program via the ROSE AST attribute mechanism. The result of applying PAUL to an annotated source code is a ROSE AST decorated with the information contained in the structured comments, ready to be further processed by other KNOT tools.

PAUL also facilitates developing custom annotation languages utilizing new structured comment TAGs. By default PAUL supports annotations formatted as "key=value" pairs and LISP-style symbolic expressions. Future plans include adding support for user-supplied parsers. By relying on structured comments and providing a parser infrastructure, the process of extending PAUL to support the specification and parsing of new annotations is simplified.

3.2 ROTE: PROGRAM TRANSFORMATION VIA TERM REWRITING

The Retargetable Open Transformation Engine (ROTE) is intended to provide programmers control over how programs are automatically transformed at the source level in order to implement refactorings and code adaptations that are necessary in porting code between HPC platforms. While tools like Eclipse or Netbeans provide refactoring capabilities, they offer only a fixed set of pre-defined transformations over which users have little control. By contrast, an *open* engine for supporting transformations will allow users with specialized knowledge of both the code and platforms to make informed decisions about the precise manner by which a transformation is implemented.

Our goal with ROTE is to allow transformations to be specified with annotated program code as input, allowing information not reflected in the language syntax and semantics to be available to the transformation tool. Source code annotations specified and parsed by PAUL are used to drive the ROTE process, telling the tool where and how transformations should be applied.

Code transformations in ROTE are based on the formal theory of *term rewriting* [8]. In the theory, *terms* are any tree-structured data, which can be iteratively transformed by the repeated application of *rules*. A set of properly constructed rules induces a particular transformation on a set of terms. Although term rewriting provides us with a solid formal foundation, the theory is quite general; part of the challenge in ROTE is to adapt the general theory to the specific problem of transforming *programs*.

There are several existing term rewriting systems, languages, and tools. We evaluated a number of these, including Maude [9], CiME [10], Stratego [11], and ASF/SDF [12]. Some of these tools are already specialized to work with source code, but only at a very low level. Since our transformations are intended to be directed by end-users, ROTE must be able to mediate between users' high-level intentions and the low-level details of term rewriting tools. We use a tool called *minitermite*, developed as part of the SATIrE project [13, 14], to bridge between a high-level representation, similar to an AST, and a lower-level term representation used at the level of the rewriting system. We have extended minitermite to support Fortran-specific AST nodes in addition to the original C language support.

We have investigated two approaches to the specification and implementation of transformations. First,

```
foo(...,
- struct s *k
+ struct s k
,...) {
<...
- k[E1].x
+ k.x[E1]
...>
}
```

Figure 3. A snippet of Coccinelle "semantic patch" syntax showing the before and after code in transforming part of a C program from using an array-of-structs to a struct-of-arrays. Similar to a unix "unified diff" patch, the overall pattern of the patch code snippet is matched to the source code. Lines prefixed with "-" specify patterns to be replaced with the matching "+" line. Different from unix diffs, the matching is language aware and pattern-based rather than literal text.

we have used the Coccinelle [15] tool to implement transformations as *semantic patches*. This diff-like format allows a transformation to be specified with a before-and-after template of the code to be modified. Coccinelle is convenient because it provides an interface layer between ROTE and the underlying term rewriting engine, but unfortunately it currently only supports the C language. Second, we have used Stratego/XT, a full-featured term rewriting system, to implement transformations directly as rewrite rules on terms. Stratego does provide a facility for transforming program source code to and from the "ATerm" term format, but is otherwise a much lower level tool than Coccinelle. In particular, rules can specified directly on the term encoding of the program, and this can be useful, at times, when very precise or subtle transformations are required.

The choice to approach ROTE using both higher-level semantic patches and lower-level term rewriting rules allows us to study the problem of user-tunable transformations at two different levels of abstraction. On the one hand, the semantic patches in Coccinelle allow us to specify complex transformations with relative concision and clarity, since the patch is expressed directly in terms of C syntax. For example, a semantic patch that matches and transforms access to a C `struct` is done by writing normal C syntax for the structure accessors, and using diff-like "+/-" prefixes to indicate the pre- and post-transformation code (an example is shown in Figure 3). Rewriting systems like Stratego instead require such program elements to be denoted by AST nodes; this syntax can be quite verbose, and consequently may obscure the transformation logic. On the other hand, semantic patches provide the user with only a before-and-after view of transformations, and may make complex, multi-step transformations cumbersome to describe. In term rewriting systems, transformations are built up from smaller transformations, allowing for precise control over their composition.

PAUL annotations are used by ROTE in two ways. First, in Coccinelle's semantic patch format, we have observed that *targeted* transformations (i.e., transformations that single out a specific data structure or instance) can induce rather convoluted patches. To mitigate this issue, we have used PAUL annotations to embed the targeting information within the program code itself. The annotations are then used to generate the complex parts of the semantic patch automatically. Second, in Stratego annotations allow us to embed metadata that will appear as sub-terms in the AST representation. This allows for rules which are "keyed" to, or triggered by, particular annotations. We have used this facility in order to target specific regions of a program, as well as to embed semantic information not inferable from the AST itself.

We are using PAUL and ROTE to apply the locality, communication, and scalability optimizations in the Self-Consistent Field application discussed in Sec. 2, and also in the BLAS to CUBLAS transformation for GPU-based accelerators discussed in Sec. 4.

3.3 BRAID: LANGUAGE-INDEPENDENT CODE GENERATION

The BRAID rewrite system for abstract intermediate descriptions combines a term-based intermediate representation with code generators for the languages C, C++, Fortran 77–2008, Java, Python, and Chapel. BRAID is intended to be complementary to ROTE, operating at a higher level of abstraction, and in a language-independent fashion. But BRAID's powerful capabilities can also be used separately from ROTE, which is the case for our initial applications.

While ROTE inherits the detailed intermediate representation (IR) used by ROSE, which captures all of the details of the input source code down to the spacing of tokens within a line of code, BRAID operates at a higher level. It uses a specially designed IR that is more abstract, language independent, and easy to generate. While most programming languages offer multiple ways to express an operation, the BRAID IR provides just one way represent each source language construct which is identical for all supported programming languages. Switching output languages is simply a matter of switching code generators. It is straightforward to define semantic-preserving conversions between the ROTE and BRAID IRs.

To facilitate authoring tools that work with the BRAID IR we provide a pattern-matching mechanism, which we used extensively in the implementation of the BRAID code generators. The pattern-matcher supports terms as first-class data types and provides full unification semantics. For example, we can write a code generation rule for variable increment operations that matches against each `assignment(Var,bin_op(plus, Var, 1)` where both occurrences of `Var` are identical. We implemented the pattern-matcher as an extension to the Python language inside of a decorator that expands all pattern-matching operations to more low-level Python code on the fly. For each node type in the IR, we automatically generate several helper functions from the formal grammar specification of the BRAID IR, including constructor functions that dynamically type-check each of the node's children. If these functions are used, it is impossible to create an inconsistent IR tree. Due to the dynamic nature of the Python language it is also possible to experiment with *ad hoc* node types by bypassing the type-checking constructors.

Initial applications of BRAID have been focused on language interoperability and interface contracts, both of which are discussed in Sec. 4.

4. (OTHER) APPLICATIONS OF THE KNOT TOOL CHAIN

Since the primary purpose of the KNOT tool chain is to allow developers to create custom transformation tools to address their software challenges, we briefly summarize some of the other applications we are pursuing, besides the SCF example presented in Sec. 2, to help drive our overall research forward.

4.1 LANGUAGE INTEROPERABILITY

While most modern large-scale CSE applications involve the use of at least two programming languages, and sometimes more, the connections between them, which can be subtly tricky and frequently compiler-dependent are most commonly done manually.

With BRAID, we are building on prior work developing the Babel [16, 17] tool, which takes a more general approach to language interoperability. BRAID allows the generation of glue code which is tailored to the specific target application, allowing many of the overheads (both in terms of performance and code size) previously associated with the Babel approach to be eliminated.

To illustrate these capabilities, we used BRAID to develop an interoperability tool that supports the

Chapel language [18] in addition to C, C++, Fortran, Java, and Python[2]. The resulting tool takes formal definitions of the desired interfaces, expressed in the Scientific Interface Definition Language (SIDL), previously developed in conjunction with Babel, and generates tailored glue code (partly in Chapel, partly in C) that translates function arguments as needed. We also extended the Chapel runtime to allow Chapel's distributed arrays to be passed to other programming languages without copying the actual data [19, 20].

4.2 CONTRACTS FOR CORRECTNESS AND RESILIENCE

BRAID is not limited to language interoperability applications. It is also being used to support the specification and enforcement of contracts for software interfaces. Executable interface contracts are a software verification mechanism involving the runtime monitoring of correctness properties. Interface contracts identify those properties that must be satisfied by the caller to use the software correctly and those that must be satisfied by a proper implementation of the interface.

The specification and automated runtime checking of software contracts are not typical features of traditional scientific programming languages. These capabilities, supported in Babel [21], are being integrated into KNOT through BRAID and PAUL. PAUL structured comments will be used to specify the contracts, and BRAID will generate the verification code.

In addition to supporting software verification, contracts can also aid fault tolerance. Silent data corruption is an area of increasing concern as we approach exascale. Contracts specifying "sanity" and consistency checks on outputs (and inputs, to be thorough) can be used as detectors for data corruption. In many cases, such contracts would be based on the programmer's knowledge of the domain and the mathematics behind the simulation, making them quite complementary to other techniques to detect data corruption.

4.3 SIMPLIFYING THE USE OF ACCELERATOR-BASED SYSTEMS

Accelerators, such as GPUs, are currently a popular route to increasing hardware performance while limiting cost and power consumption. As they are commonly used today, CSE applications must put in a good deal of additional effort to take advantage of them, porting and maintaining (portions of) their code to a new language (i.e. CUDA or OpenCL). However, as accelerators gain support from common numerical libraries, it will become increasingly feasible to take advantage of accelerators through calls to libraries maintained by others, thus reducing or eliminating the need to maintain two code bases in parallel.

On the other hand, the interfaces for GPU libraries are generally not the same as the corresponding CPU versions – data must be marshaled to and from the accelerator, etc. Thus, the programmer is still left with a code maintenance challenge if they want to be able to use their application on both accelerator and non-accelerator systems.

We have developed a proof of principle demonstration which uses ROTE to transform BLAS library calls into the CUDA BLAS (or CUBLAS) library developed by NVIDIA, including the necessary memory management and data marshaling code. PAUL annotations are used to control the application of the transformations so that (for example) small operations, which are not cost effective to execute on the accelerator, can be kept on the CPU. With such tools, the application developer could write and maintain the CPU-only version of the code, along with annotations for where CUBLAS substitutions are appropriate. The transformations could then be applied at compile time, depending on the target platform.

Right now, the ROTE term rewriting engine is limited to the languages supported by the ROSE compiler. BRAID will enable us to automatically generate code for an unsupported accelerator or a highly domain-

[2]This synergistic work is funded under the Composite Parallelism project supported by DOE ASCR.

specific language (DSL). In this scenario, a transformer written with ROTE removes compute kernels from the original program and inserts the calls and data transfers to the accelerator in their place. The (ROTE) term representation of the compute kernels is then raised into the BRAID representation and handed over to a code generator that compiles it into for the accelerator code. BRAID was designed to make it easy to write customized code generators for new languages. Supporting a new accelerator architecture is then mostly a work of writing a new BRAID code generator.

4.4 COMPOSITION OF THREADED SOFTWARE

A more forward-looking application we are beginning to explore involves multi-threaded applications. Although threaded applications have been around for quite some time, the rise of multi-core processors has lead to increasing interest in multi-threading. One of the challenging problems that comes with multi-threaded applications is how to deploy them onto the computer – what combination of MPI processes and threads to use – in order to get the best performance? Experience shows that frequently, different portions of an application will perform better with different threading models. However with today's tools, applications are generally forced to compromise on a single threading model which is used throughout the application.

The goal of our work in this area is to allow different portions of an application to run with different threading models, using tools built with the KNOT tool chain to insert the code necessary to glue the different portions together. We are currently working with an OpenMP multi-threaded version of the LAMMPS molecular dynamics application [22–24] to identify an appropriate target for a proof of principle implementation of the threaded composition capability which we will subsequently automate with KNOT.

5. DISCUSSION AND FUTURE PLANS

We have described how the COMPOSE-HPC project is developing a suite of tools, collectively known as the KNOT tool chain, aimed at democratizing source-to-source transformation technology in order to make it accessible to knowledgeable developers of computational science and engineering applications. Our goal is a "pay for what you use" environment, in which users can develop small, tailored solutions to their programming challenges with the confidence that they can be shared and composed. We have also described a number of diverse applications we are pursuing, both as demonstrations of the possibilities and to drive our research and development forward.

Our work to date has focused primarily on developing the core capabilities needed in the KNOT tools, leveraging existing tools wherever possible. As capabilities mature, we will increase our focus on the user interface to the tools, which is crucial to our research objective of demonstrating s2s transformation technology which is tractable for those who are not experienced in writing compilers. Given the scope and duration of the project, our goal is to deliver research-grade, proof of principle implementations of the technology – accompanied by compelling demonstrations of its use – which can be turned into robust production-quality tools as a follow-on activity.

ACKNOWLEDGMENTS

We gratefully acknowledge the contributions of Manojkumar Krishnan and Dietmar Ebner to the project.

This work has been supported by the U. S. Department of Energy, Office of Science, Office of Advanced

Scientific Computing Research (ASCR), as part of the X-Stack Software Research program. It has also been supported by by the ORNL Postmasters and Postdoctoral Research Participation Programs and the ORNL Higher Education Research Experiences Program which are sponsored by ORNL and administered jointly by ORNL and by the Oak Ridge Institute for Science and Education (ORISE). ORNL is managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725. ORISE is managed by Oak Ridge Associated Universities for the U. S. Department of Energy under Contract No. DE-AC05-00OR22750. This work performed, in part, under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Pacific Northwest National Laboratory is operated by Battelle under contract DE-AC06-76RLO 1830. Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, under contract DE-AC04-94-AL85000.

REFERENCES

[1] ROSE Compiler Framework, http://rosecompiler.org/.

[2] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong, NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations, *Computer Physics Communications* **181**, 1477 (2010).

[3] Jarek Nieplocha, Ponuswamy Sadayappan, and Robert Harrison, *Global Arrays: Scientific Programming for Scalable Parallel Computers*, CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2011.

[4] Jarek Nieplocha, Bruce Palmer, Manojkumar Krishnan, and P. Saddayappan, Overview of the global arrays parallel software development toolkit, in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006, ACM.

[5] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha, Scalable work stealing, in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, New York, NY, USA, 2009, ACM.

[6] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy, Lifeline-based global load balancing, in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 201–212, New York, NY, USA, 2011, ACM.

[7] Tracy P. Hamilton and Henry F. Schaefer, New variations in two-electron integral evaluation in the context of direct SCF procedures, *Chemical Physics* **150**, 163 (1991).

[8] Franz Baader and Tobias Nipkow, *Term Rewriting and All That*, Cambridge University Press, New York, NY, USA, 1998.

[9] The Maude System, `http://maude.cs.uiuc.edu/`.

[10] The CiME Rewrite Tool, `http://cime.lri.fr/`.

[11] Stratego/XT home page, `http://strategoxt.org/`.

[12] The Meta-Environment: ASF+SDF, `http://www.meta-environment.org/Meta-Environment/ASF%2BSDF`.

[13] Gergö Barany and Adrian Prantl, Source-Level Support for Timing Analysis, in Tiziana Margaria and Bernhard Steffen, editors, *ISoLA (2)*, volume 6416 of *Lecture Notes in Computer Science*, pages 434–448, Springer, 2010.

[14] SATIrE: Static Analysis Tool Integration Engine, `http://www.complang.tuwien.ac.at/satire`.

[15] Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller, Documenting and Automating Collateral Evolutions in Linux Device Drivers, in *EuroSys 2008*, pages 247–260, Glasgow, Scotland, 2008.

[16] Tamara Dahlgren, Dietmar Ebner, Thomas Epperly, Gary Kumfert, James Leek, and Adrian Prantl, *Babel User's Guide*, Lawrence Livermore National Laboratory, 2012, version 2.0.0.

[17] Thomas G. W. Epperly, Gary Kumfert, Tamara Dahlgren, Dietmar Ebner, Jim Leek, Adrian Prantl, and Scott Kohn, High-performance language interoperability for scientific computing through Babel, *IJHPCA* (2011).

[18] B.L. Chamberlain, D. Callahan, and H.P. Zima, Parallel Programmability and the Chapel Language, *International Journal of High Performance Computing Applications* **21**, 291 (2007).

[19] Adrian Prantl, Thomas G. W. Epperly, Shams Imam, and Vivek Sarkar, Interfacing Chapel with traditional HPC languages, in *Fifth Partitioned Global Address Space Conference (PGAS 2011)*.

[20] Adrian Prantl, Thomas G. W. Epperly, and Shams Imam, Poster: Connecting PGAS and traditional HPC languages, SC'11 Research Poster Session, 2011.

[21] Tamara Dahlgren, David Bernholdt, and Lois Curfman McInnes, Gaining Confidence in Scientific Applications Through Executable Interface Contracts, in Rick Stevens, editor, *SciDAC 2008, 14-17 July 2008, Seattle, Washington, USA*, volume 125 of *Journal of Physics: Conference Series*, page 012086, Institute of Physics, 2008.

[22] Steve Plimpton, Fast parallel algorithms for short-range molecular dynamics, *J. Comput. Phys.* **117**, 1 (1995).

[23] LAMMPS Website, `http://lammps.sandia.gov`.

[24] Axel Kohlmeyer, LAMMPS-ICMS, `https://sites.google.com/site/akohlmey/software/lammps-icms`.

ORNL/TM-2012/85

INTERNAL DISTRIBUTION

1. Central Research Library
2. Laboratory Records, record copy (RC)

3–4. DOE Office of Scientific and Technical Information (OSTI)