Reinventing Explicit Parallel Programming for Improved Engineering of High Performance Computing Software

Anthony Skjellum, Purushotham Bangalore, Jeff Gray, and Barrett Bryant

University of Alabama at Birmingham
Department of Computer and Information Sciences
Birmingham, Alabama 35294-1170, U.S.A.
{tony, puri, gray, bryant} @ cis.uab.edu
http://www.cis.uab.edu

Abstract

From on-going practices of those who develop high performance software, there is an evident, pressing need for a highly focused effort to "reinvent" the explicit methodologies that support parallel programming. To enable a new methodology, orthogonalization of features and separation of concerns within the feature sets of existing parallel models (e.g., MPI-1, MPI-2, DRI, MPI/RT, BSP) are required from the viewpoint of creating notational instantiations sufficiently simple to achieve the requirements of actual parallel programs. Additionally, the new notational representations for expressing parallel programs must not be burdensome in terms of the accidental complexities for expressing an implementation (either in middleware or in compiler-assisted syntactical form). In support of this vision, this paper identifies specifically the need for application of modern software engineering and design concepts to a refactoring of the Message Passing Interface's set of capabilities. Concepts and methodologies from modern software engineering – Model Driven Architecture and Aspect-Oriented Software Design – inform this effort.

1. Introduction

This position paper asserts the need for a highly focused effort to "reinvent" how explicit parallel programming is accomplished practically. It takes the pragmatic view of applying modern software engineering and design concepts to a refactoring of the Message Passing Interface's (MPI) [15] set of capabilities (rather than its API, *per se*). MPI has been widely adopted by developers of high-performance computing (HPC) systems [21], yet there is a pressing need to update the models, notations, and general methodology in order to ensure that MPI will continue to meet the demands and requirements of future HPC systems. This complements the many efforts in automatic parallelization/compilers/techniques.

As observed at a recent CRA meeting [5], naively replacing a 10-year old object-based standard with yet another object-based standard would prove insufficient toward increasing programmer productivity. A minor incremental revision to HPC standards would also fail to raise the level of abstraction where possible, as consistent with high performance and scale.

A reinvented methodology for parallel programming, as applied to high-performance computing, should embrace recent concepts from software engineering. Our vision of a new approach to HPC considers the adoption of the following four ideas from software engineering:

1. The use of object-oriented (vs. object-based) notations, together with ideas from generative programming [6] and other post-object-oriented programming techniques need to be employed at a practical level. Aspect-oriented software development (AOSD) [13] can assist as a means both to transform implementations of the notation to meet programmer requirements, and to transform user programs [3] to achieve optimizations, fault enhancements, specializations, and/or instrumentation for performance understanding.
2. Model-driven architecture (MDA) [8] and Model-Integrated Computing [23] for the definition of the new prototype notations should be available to assist domain experts in the construction of HPC systems. Model-driven approaches, when combined with a domain-specific visual language, enable the specification of the essential properties of a system in a notation that is closer to the problem space, rather than the solution space [14]. From such models, the underlying representation of the solution for a specific platform can be generated. This has been shown to be helpful in isolating the accidental complexities of middleware [9], and the benefits

of the approach, as applied to HPC and parallel programming, should be studied.

3. The use of design patterns for high-performance middleware (similar to descriptions in [20]) are needed to express domain-specific parallel data and computation kernels. Such patterns form the basis for a notation that is rich in asynchrony, supports planned transfers, emphasizes overlapping of computation, communication, and I/O, and leaves open the removal of abstraction barriers by compilers, translators, and middleware. The description and classification of patterns for HPC can provide a catalog of experiential reuse to assist new developers of HPC systems.
4. Many of the more recent approaches to software engineering have been investigated and applied to modern languages like Java. Yet, many HPC and parallel programming solutions are coded in legacy languages like FORTRAN. As a bow to the particular needs of real HPC parallel programmers, there is a strong need to bring the advantages of new software engineering approaches into the purview of legacy languages. In particular, tool support for refactoring [7, 17, 19] and aspect-orientation [13] is needed for legacy languages [11]. It is expected that with advances in aspect-oriented programming in the next five years, that FORTRAN will be supportable using program transformation approaches [3].

The next section will highlight some of the problems of current practice in HPC and parallel programming, particularly when MPI is used. Section 3 offers future directions for applying recent software engineering techniques to the identified problems. A summary of potential impact of the research directions proposed in this paper is provided in the conclusion.

2. State of Current Practice

HPC applications are rarely designed and implemented from scratch. Rather, they are developed most often by reengineering existing sequential applications using ad-hoc approaches. The most common method used to develop HPC applications is a data parallel approach wherein a given dataset is distributed among multiple processors and each processor is assigned to work on that dataset while exchanging data through explicit message passing. With the data parallel approach, data distribution plays an important role in the overall performance of the application because it has impact on load balancing and message passing time. Often, data distribution is determined by the specific application and performed manually by the domain-expert using automated tools such as Metis [16], Chaco [12], and others that could be used with unstructured grids.

Several key problems can be identified in the current practice of HPC software development:

Ad hoc design decisions: Once data distribution is performed either manually or using automated tools, the next step is to establish communication between different processors. There are several communication options available (e.g., synchronous/ asynchronous, static/dynamic, point-to-point/collective), and there are trade-offs with each choice depending on the specific application. There are no well-established rules, guidelines or patterns; thus, users rely on ad-hoc approaches. Software engineering practices and principles are typically not used to evaluate the different design choices. Instead, users try out an option and, if it works with some reasonable improvement in performance, they will use that solution. Alternative choices are considered only when there is significant performance degradation. This typical ad-hoc scenario is predominated by the non-scientific approach of trial and error.

Lack of support for adaptivity and evolution: HPC applications developed using MPI-1 used a fixed number of processors once an application started execution. Even though MPI-2 provides dynamic process management capabilities, many applications are currently not capable of exploiting this feature because the runtime systems do not provide the capability to add more resources once a job is scheduled for execution.

Poor capabilities for fault-tolerance and recovery from exceptions: When an HPC application is running on multiple processors and there arises an error in one of the processes, the application aborts. The sudden termination of the process leaves the entire job hanging, mainly because of the lack of support for fault-tolerance in message-passing. An additional problem is caused because there is no state associated with message-passing that could be saved and used to restart that process. Application level check pointing has to be built-in to the parallel programs to address faults and support restarts. Even though several parallel check pointing and fault-tolerance additions to MPI are underway, there is no clear programming paradigm to support the development of fault-aware adaptive parallel applications.

3. An Outline for a New Approach

A longstanding goal of software engineering is to construct software that is easily modified and extended. A desired result is to achieve modularization such that a

change in a design decision is isolated to one location [18]. As demands for HPC software increase, future requirements will necessitate new strategies to support the requisite adaptations across different software artifacts (e.g., models, source code, test cases, documentation) [2].

Although MPI is a common, effective, and powerful programming model for multicomputers and clusters, it has been repeatedly "bashed" for its low-level abstractions that are too hard to use. It is the basis for "legacy parallel codes" of the 1990's, whereas vectorized Cray FORTRAN produced the 1980's equivalent. (Because MPI is the only parallel notation to work effectively at scale, its importance cannot be understated, *vis a vis* other projects, which have no significant "market penetration" as yet to compare with MPI at scale.) A systematic study by the first author of this paper and his students has revealed both obvious and subtle opportunities for enhancements to MPI [22]. However, retrofitting MPI incrementally is not interesting enough to constitute a key research contribution, and there may be insufficient demand among legacy code users either to exploit new features by manual recoding, nor to accept amendments to the MPI standard for the foreseeable future. This position paper emphasizes a refactoring of the MPI specification to demonstrate how to represent the features offered by MPI in a collective, non-blocking, object-oriented framework that emphasizes composition, and deemphasizes blocking, polling, single-threaded computation per process (or processor).

The refactoring of MPI is required to compare the features provided by existing standards MPI/RT 1.1, DRI 1.1, and other message passing systems. Experience and intuition about limitations justified in terms of use cases and scenarios will certainly motivate changes and consolidate functionality. Patterns not captured by MPI but emerging in it (like rings, data transpositions) will be utilized in the refactoring process. However, no new MPI implementation will be required. Rather, the goal is to gather semantic information on the major features and emerging/minable patterns of parallel programming that can be obtained from looking at MPI globally, because small local changes could drastically affect the global performance and behavior of HPC applications. The refactored MPI will emphasize more functions that do less, but compose better, and orthogonality and scalability that exceed MPI from a specification point of view, rather than a quality of implementation point of view (the best MPI implementation will nonetheless be less scalable than a comparable quality refactored MPI implementation for areas of specific improvement.) The remainder of this section will outline the software engineering principles that will aid in this refactoring.

Model-driven Configuration: Refactoring and customization of MPI will be accomplished through the use of Model Integrated Computing (in particular, the GME meta-modeling tool [14]) to build a meta-model that captures the features of MPI (plus added features, minus removed features) and the particular usage scenario for a specific application. From the high-level models, the footprint of the MPI middleware will be minimized to provide concise, orthogonal, and complete features that superset what traditional MPI provides, while supporting specialization for semantically simpler cases that are not accessible in MPI because of the fat interfaces uses for the object-based API. The combined use of GME, the meta-model for MPI-1, and MPI-2, and the stub generation capabilities of this system will be used to restructure object-oriented APIs that could emerge from refactoring. Principles from generative programming [6], such as generation of code artifacts from high-level domain models, will be a key technique. The model-driven configuration will allow HPC developers to explore design alternatives more readily, compared to the ad-hoc approach that is now prevalent in the tools and methodologies available to developers of scientific applications.

Grid relevance will be built into the modeling notation by providing process management functionality that allows groups to be formed and reformed, and these groups to be used with sets of operations defined for them in an object-oriented manner. This kind of group instantiation will not be as "safe" and "opaque" as MPI's communicator model, but will be much more flexible and scalable. They are therefore not equivalent to PVM's open groups, but rather more like generalized MPI intercommunicators, including multipartitite relationships. State transition–based event notifications will be considered for groups as they transform. Lower safety in the communication model will be overcome by providing constraint satisfying compositions that help compose code safely in a generative programming style [6] (checking of the uses will be globalized either through a pre-compilation tool, or enforced through use of parallel design patterns, but low-level operations will be more optimistic to allow for higher performance when composed).

An example of the dynamic process management model that is needed, which is nearly impossible in MPI-2, is the merger of K groups of processes (whether independent singletons or groups). The new notation's parallel programming model will emphasize ease of build up and build down of groups. However, it will also keep in mind the need to be able to use underlying scalable startup schemes, and not effectively defeat these. We have long argued about correct MPI-2 programs for making even three groups merge, and so this has to be fixed in a next-generation system.

Another aspect of the new modeling notation will be the ability to do reversibility of creating and destroying process groups. In MPI, it is not quite clear how to shrink the process management task, so effectively, dynamic process management is used in bi-partite settings, in settings where two groups merge and then stay merged for the rest of a computation, but not when where there is a lot of dynamicism.

Improved Separation of Concerns in HPC: Much of the ad-hoc practice in HPC development stems from the inability to properly separate specific concerns. This severely hampers the ability to adapt and evolve software when changes are needed to the source code in order to address new requirements driven by a specific scientific domain. Aspect-Oriented Software Development (AOSD) provides a capability to modularize concerns that are crosscutting in nature [13]. With respect to the lack of flexibility for fault-tolerance, error recovery, and exception handling in HPC applications, an AOSD approach has the potential to offer significant benefit. With AOSD, the fault tolerance concerns can be weaved into the proper locations, eliminating the need for manual adaptation, which can often lead to an error-prone process.

This approach does not emphasize the "salvation" of all legacy MPI codes; however, support to transitioning the computational applications to the new notation is part of our future investigation, and experiential data about this transition will be part of the computer science outcomes of this work. Furthermore, a performance monitoring interface will be provided inside the new notation and its implementation prototypes so that a) profile guided optimization is possible, and b) hooks to program visualization and existing performance code is immediately possible. These will all be driven by aspects. In previous efforts, the authors have combined model-driven techniques with aspects in order to accomplish large-scale evolution of a legacy system by generating aspect-oriented transformations from high-level models [10].

Componentization of HPC: The Common Component Architecture (CCA) [1] provides a standard comprising component framework to deal with the complexity of developing interdisciplinary high performance computing applications, while allowing integration of multiple components written in different programming languages using multiple paradigms. For example, a visualization module written in one language using a shared memory paradigm can be integrated with a computational application that may be written in another language that uses MPI for interprocessor communication by a collective port. It is interesting to study the implications of CCA to Grid applications in order to integrate an even more diverse range of applications and tools.

CCA has been widely used in "plug-and-play" type application environments such as climate, weather, and ocean modeling simulations [1]. In this model, components are units of software functionality that can be plugged together to form applications. The components interact via interfaces, known as ports. The components are enclosed in a framework that provides ports for components to interface, and a set of standard services available to all components.

The types of components implied by CCA are relatively coarse grain, and therefore appear relevant to grid computing, as well as process-level computing. Components of finer granularity, which compose at light weight at compile time are also of interest. Importantly, this effort will avoid making least common denominator decisions driven by language particulars, as MPI-1 was forced to do with supporting Fortran-77 and C simultaneously. (Some of these actually impact the scalability of the MPI notation, which limits MPI's usefulness in describing communication in certain irregular parallel codes). Language-independent aspect weaving and transformation tools can be utilized to assist in such composition [3, 4, 11]. Model-driven techniques can also be integrated with a CCA approach such that a domain-specific modeling environment is used to connect and generate CCA-based applications.

The combined use of aspect-oriented programming, model driven architecture, and a more sensible, orthogonal design for non-blocking collective operations suitable either for data parallel or irregular programs will make next-generation parallel programming far better than MPI.

4. Conclusion

As asserted in this position paper, improved methodologies and respective tool support are needed to better enable developers of HPC systems in dealing with the increasing complexities of parallel programming. Our vision is to integrate modern software engineering techniques into the practice of HPC development. The outcome would offer designs and prototypes of a new parallel programming notation that remains explicit in the sense that parallelism is not automatically obtained, derived, or inferred. Instead, the notation will make it much easier to express certain patterns of parallel programs, compose these, make them work correctly, and study/refine performance. Our work in this area is just beginning, but we are convinced that the topics described in this paper would generate fruitful discussion at the workshop.

References

1. R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, "Toward a Common Component Architecture for High-Performance Scientific Computing", in *Proceedings of the High-Performance Distributed Computing Conference*, August 1999, pp. 115-124.
2. D. Batory, J.N. Sarvela, and A. Rauschmeyer, "Scaling Step-Wise Refinement," *International Conference on Software Engineering*, Portland, Oregon, May 2003, pp. 187-197.
3. I. Baxter, C. Pidgeon, and M. Mehlich, "DMS: Program Transformation for Practical Scalable Software Evolution," *International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, May 2004.
4. F. Cao, B. Bryant, R. Raje, M. Auguston, A. Olson, and C. Burt, "A Component Assembly Approach Based On Aspect-Oriented Generative Domain Modeling," to appear in *Electronic Notes in Theoretical Computer Science*, 2004.
5. Computing Research Association, "Report of Workshop on The Roadmap for the Revitalization of High-End Computing," organized by Computing Research Association, Edited by Daniel A. Reed, Washington D.C., June 16-18, 2003.
6. K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison Wesley, 2000.
7. M. Fowler, *Refactoring: Improving the Design of Existing Programs*, Addison-Wesley, 1999.
8. D. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, Wiley Publishing, 2003.
9. A. Gokhale, D. Schmidt, B. Natarajan, J. Gray, and N. Wang, "Model-Driven Middleware," in *Middleware for Communications*, (Qusay Mahmoud, ed.), John Wiley & Sons, 2003.
10. J. Gray, J. Sztipanovits, D. C. Schmidt, T. Bapty, S. Neema, and A. Gokhale, "Two-level Aspect Weaving to Support Evolution of Model-Based Software," in *Aspect-Oriented Software Development*, (Robert Filman, Tzilla Elrad, Mehmet Aksit, and Siobhán Clarke, eds.), Addison-Wesley, 2004, Chapter 29 .
11. J. Gray and S. Roychoudhury, "A Technique for Constructing Aspect Weavers Using a Program Transformation System," *International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, March 22-27, 2004, pp. 36-45.
12. Bruce Hendrickson and Robert Leland, "An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations," *SIAM Journal on Scientific and Statistical Computing*, 16(2) – 1995, pp. 452-469.
13. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241, Springer-Verlag, Jyväskylä, Finland, June 1997, pp. 220-242.
14. Á. Lédeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, November 2001, pp. 44-51.
15. Message Passing Interface Forum, "MPI2: A Message-Passing Interface Standard," *International Journal of Supercomputer Applications and High Performance Computing*, Special Issue, 12(1/2), pp. 1-299, 1998 [Specifically: Chapter 7, pp. 139-157]
16. METIS. http://www-users.cs.umn.edu/~karypis/metis/
17. W.F. Opdyke, *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992, http://citeseer.nj.nec.com/opdyke92refactoring.html
18. D. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, December 1972, pp. 1053-1058.
19. D.B. Roberts, *Practical Analysis for Refactoring*, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1999. (http://st-www.cs.uiuc.edu/~droberts/thesis.pdf)
20. D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, John Wiley and Sons, 2000.
21. A. Skjellum, E. Lusk, W. Gropp, "Early Applications in the Message-Passing Interface," Invited Paper, *International Journal of Supercomputing Applications*, June 1995.
22. Anthony Skjellum, "High Performance MPI: Extending the Message Passing Interface for Higher Performance and Higher Predictability," *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, Las Vegas, Nevada, July 1998.
23. J. Sztipanovits and G. Karsai, "Model-Integrated Computing," *IEEE Computer*, April 1997, pp. 10-12.