A Case Study in Tightly Coupled Multi-paradigm Parallel Programming

Sayantan Chakravorty[1], Aaron Becker[1], Terry Wilmarth[2] and Laxmikant Kalé[1]

[1] Department of Computer Science, University of Illinois Urbana-Champaign
[2] Center for Simulation of Advanced Rockets, Univ. of Illinois Urbana-Champaign

Abstract. Programming paradigms are designed to express algorithms elegantly and efficiently. There are many parallel programming paradigms, each suited to a certain class of problems. Selecting the best parallel programming paradigm for a problem minimizes programming effort and maximizes performance. Given the increasing complexity of parallel applications, no one paradigm may be suitable for all components of an application. Today, most parallel scientific applications are programmed with a single paradigm and the challenge of multi-paradigm parallel programming remains unmet in the broader community.
We believe that each component of a parallel program should be programmed using the most suitable paradigm. Furthermore, it is not sufficient to simply bolt modules together: programmers should be able to switch between paradigms easily, and resource management across paradigms should be automatic. We present a pre-existing adaptive run-time system (ARTS) and show how it can be used to meet these challenges by allowing the simultaneous use of multiple parallel programming paradigms and supporting resource management across all of them. We discuss the implementation of some common paradigms within the ARTS and demonstrate the use of multiple paradigms within our feature-rich unstructured mesh framework. We show how this approach boosts performance and productivity for an application developed using this framework.

1 Introduction

A parallel programming paradigm defines how concurrent tasks in a parallel program access data and interact with each other and how those interactions are expressed by the programmer. Such paradigms are created to address the needs of specific classes of problems in parallel computing, making implementations easier to develop. A programmer often develops a parallel algorithm with a certain parallel programming paradigm in mind. Programmers have a wide range of paradigms to choose from, such as message passing (e.g. MPI), shared address spaces (e.g. Global Arrays[28], UPC[9], OpenMP[7], HPF[11]), message-driven (active messages, actors, Charm++) and stream processing. This variety exists because not all paradigms are suitable for all problems. Choosing the correct paradigm for an application yields benefits in terms of lower programming effort, elegant, easily maintained code, and better performance.

As parallel applications become increasingly complex with multiple constituent algorithms, no single paradigm is suitable for all the different parts of an application. Most programmers currently choose a paradigm that is suitable for the bulk of the application and force the rest into the same paradigm. This results in reduced programmer productivity via inelegant code, potentially more errors, longer development time and lower maintainability. It can also result in poor performance when the inherent parallelism in the problem could be more naturally and fully expressed in some other paradigm.

Developing each application component with the most suitable paradigm would require a multi-paradigm parallel programming system in which different paradigms can be tightly coupled. In such a system, *the application would allow different paradigms to be used concurrently*. In addition, *the components of an application would not be restricted to a single paradigm*, nor would components of a certain paradigm be restricted to a subset of the physical processor space. Without a tightly coupled multi-paradigm environment, performance and productivity are adversely affected when software components cannot be cleanly expressed within one paradigm. Moreover, this multi-paradigm system should enable *resource management across all the paradigms on all the processors*. This is important for scaling applications to even moderately sized machines, since resource management issues like computational load imbalance and communication bottlenecks are often the biggest roadblocks to scaling.

A programmer faces further challenges when selecting a non-mainstream paradigm that may be ideally suited to her problem. There is a huge barrier to entry for alternative paradigms caused by the relative dominance of MPI. In the absence of decisive performance benefits, the best way for new models to gain traction is to interoperate cleanly with existing models, both to allow the use of existing libraries and to facilitate reuse of new code.

2 Background

Approaches to multi-paradigm parallel programming roughly fall into four categories: 1) multi-paradigm parallel languages; 2) extensions to existing parallel programming models; 3) interoperability libraries; and 4) run-time systems.

Multi-paradigm parallel programming languages have primitives that let a user take advantage of different paradigms in a tightly coupled fashion. One major disadvantage of this approach is that existing software implemented in various paradigms is difficult or impossible to reuse, and conversely, software developed in such languages may not be reusable in other applications developed with other languages. As an example of a multi-paradigm language, mpC[23] is a C superset that provides *network objects* to describe data and communication layouts in a parallel environment. It supports both task and data parallelism and enables both computation and communication optimizations.

Extensions to parallel programming models enhance existing models to allow the use of additional paradigms. Efforts to merge OpenMP and MPI such as Extended OpenMP (EOMP) [30] and MPI+OpenMP [6] fall into this category. Codes developed using mixed-mode techniques have been quite successful in some cases [29] Most extensions involve only two paradigms without a general framework for adding more. Moreover, different paradigms are often executed as different processes or kernel threads, increasing the cost of switching between paradigms. Although MPI+OpenMP attempts dynamic load balancing by moving OpenMP threads among processes [6], it is restricted to moving them among processors on the same node and cannot do truly global resource management.

Interoperability libraries provide interfaces between systems that implement different parallel models. Fortran M uses MPCL (message-passing compatibility library) to interface with other message passing libraries and HPF [12]. Here, the freedom to use multiple paradigms is marred by the additional complexity of the interface. Fortan M tackles the resource management issues by allowing access to its resource management facilities through MPCL. Other attempts to bolt multiple paradigms together, such as PVM with Solaris Threads, resulted in poor performance and undue code complexity [25].

Parallel Adaptive Run-time Systems (ARTS) attempt to provide the set of tools that are required to implement different parallel programming models. Existing models and languages can be implemented on top of such a run-time system and can interoperate using a common substrate. ARTS provide resource management capabilities common to all models, thereby relieving the programmer of this task. TPVM[10] was an early extension to PVM that implemented a thread-oriented, event-driven run-time and the notion of virtual processors. It consistently outperformed PVM in several experiments.

Our approach relies on CONVERSE [19], an ARTS which fully realizes the tightly coupled interoperability of multiple paradigms. CONVERSE meets many of the goals mentioned in a recent report from Berkeley [3], including the need for models to be independent of the number of processors, and it alleviates cognitive load on programmers by performing automatic resource management.

3 A Multi-paradigm Runtime System

Object-based virtualization [18] is a flexible approach to implementing a variety of interoperable programming paradigms. It encourages the decomposition of a computation into a large number of interacting objects called *virtual processors* (VPs). The task of mapping VPs to physical processors is handled by the ARTS, which can change the mapping at run-time by migrating VPs between processors. The ARTS is also responsible for message delivery between VPs. A scheduler on each processor selects which local VP executes next. The scheduler is message-driven and only schedules VPs that have pending messages.

Object-based virtualization does not dictate the paradigm used within a VP, so different VPs may use different paradigms. CONVERSE [19] provides the tools for implementing different paradigms in a message-driven system with object-based VPs. Apart from providing a scheduler on each processor, CONVERSE also provides a messaging framework with primitives for point-to-point communication and multicasts, and methods for handling a message on the receiving processor. CONVERSE also offers user-level threads[32] with low context switch overhead and the ability to migrate threads between processors. Local CONVERSE threads share the scheduler with incoming messages.

Programming paradigms developed using the CONVERSE ARTS give programmers the ability to efficiently compose independently-developed components into a single application or higher-level component. Components developed using separate paradigms can overlap their execution in time and over processors. Since multi-paradigm VPs share the same address space on a processor, a flow of control can switch paradigms cheaply via function calls or local messages within a processor. Therefore different paradigms implemented on CONVERSE can be tightly coupled within a single application.

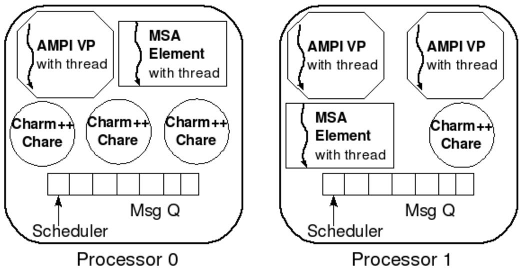


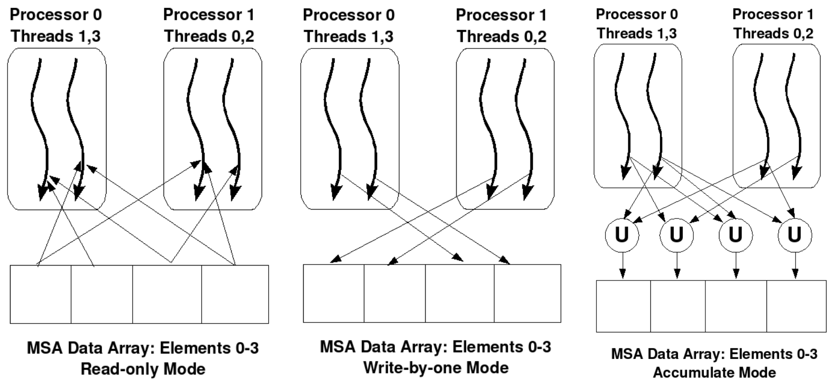
Fig. 1. The CONVERSE ARTS allows multiple VPs with different paradigms on the same processor.

Numerous models have already been implemented on the CONVERSE ARTS. These include the *message-passing* model via Adaptive MPI, the *message-driven* model via CHARM++, and a phase-based *distributed shared memory* model called *Multi-phase shared arrays* (MSA). Figure 1 shows VPs using these models while sharing the same processor. Similarly, *global address space* languages are supported via an implementation of ARMCI. An *orchestration* model called Charisma allows for clear expression of control and data flows between serial components.

Object-based virtualization enables a number of performance benefits such as adaptive overlap of computation and communication [18], dynamic measurement-based runtime load balancing [31] and dynamic communication optimizations [22]. Since all paradigms in an application use CONVERSE, resource management need not be limited to one paradigm. For example, 1) the load balancer takes into account work loads of the VPs belonging to all paradigms while trying to balance load; 2) if two VPs using different paradigms send each other many small messages, the communication optimization library can merge these into fewer larger messages.

CHARM++ and MPI have both been described extensively, and their relative merits and deficiencies have been explored [20, 13, 26, 2, 14, 4]. Therefore we will not describe them further except to say that both have proven suitable for developing complex parallel applications. CHARM++ is implemented on top of CONVERSE, and AMPI [17] is an implementation of MPI on CONVERSE. Together with Multiphase Shared Array (MSA), a simple shared memory model which we will now describe, these models make up the bulk of our unstructured meshing framework.

MSA consists of shared arrays that can change between three possible access modes. An MSA array can be constructed from any user-specified type. The lifetime of an MSA array is divided into phases, with all threads accessing the array in the same access mode during each phase. Phase boundaries are marked by synchronization. The phase based nature of MSA programs means that they can never have deadlocks or race conditions. The three possible access modes (shown in Figure 2) are:



(a) Read only mode (b) Write by one mode (c) Accumulate mode

Fig. 2. The three different access modes of a MSA array

Read-only mode: All threads can only read from the MSA array during this phase. Each element can be read by multiple threads.

Write-by-one mode: In this mode, all threads are permitted to write to the MSA array, but no element can be written to by multiple threads.

Accumulate mode: All threads can update the MSA array and multiple threads can update a single element. For each element, the data provided by different threads is accumulated using a user defined associative commutative operation such as addition, multiplication, max, set union and set intersection.

MSA represents a compromise between the convenience of a global address space and the performance and correctness problems associated with unfettered access to shared data [5]. The restrictions imposed by the access phases allow for more efficient communication while also preventing common shared memory programming hazards. MSA has proven useful in codes varying from matrix multiplication to distributed hashtables to molecular dynamics and has provided significant advantages to parallel programmers across many problem domains.

The CONVERSE runtime has also been used to implement a variety of other parallel programming paradigms. One example is Charisma [15], an orchestration language that lets the programmer specify the control and macro data flows of a parallel program separately from the sequential portions. Charisma is built on top of CHARM++. The user expresses the global message flow in the orchestration code without fragmenting it among all the different types of objects in a complicated parallel application. The orchestration code can express all commonly used communication patterns, including point-to-point, broadcast, multicast, reductions, scatter and gather. The sequential portions are normal C++ code. A

Charisma program can be combined with any library written using one of the parallel programming paradigms supported by the CONVERSE runtime system.

Aggregate Remote Memory Copy Interface (*ARMCI*) [27] supports high performance remote memory copy on multiple platforms. It offers blocking and non-blocking versions of data transfer operations, synchronization operations and memory allocation and deallocation routines. ARMCI is used as the foundation for a number of global address space languages such as Global Arrays [28] and Co-Array Fortran [8]. ARMCI is implemented on the ARTS by encapsulating each ARMCI process within a threaded Charm object [16]. The ARTS can perform intelligent resource management for any application using ARMCI.

4 ParFUM: An example of multi-paradigm programming

PARFUM [24] is a framework for the parallelization of unstructured mesh applications. It provides the programmer with a rich set of features such as mesh partitioning, communication between mesh partitions, mesh adaptivity, mesh locking, collision detection and data transfer. Due to the complexity of these features, each was implemented in the parallel paradigm most suited to it. These differences in paradigm are largely hidden from the user, to whom the application appears to be completely within the message-passing style. We describe some of these features and our programming model choices for their implementation below. We also discuss situations in which multiple parallel programming paradigms were used to provide a single feature.

In PARFUM, each mesh partition is associated with a single VP, and a driver routine is invoked by each of these VPs. In most applications, mesh nodes and elements (collectively, *entities*) along partition boundaries require data from entities on neighboring partitions to compute local solutions. PARFUM provides functionality for adding local read-only copies of remote entities, or *ghosts*, to the partition boundary. A single collective PARFUM call updates all ghost entities with data from the original entities on neighboring partitions.

PARFUM also provides synchronization primitives to update the values of shared nodes during a simulation. The user code in the driver routine is typically written in a message passing style with blocking ghost update calls and synchronization routines such as barriers and reductions, and makes use of adaptivity and locking mechanisms which use the message-driven style in a manner transparent to the programmer. As a result, both serial codes and pre-existing MPI codes can be easily modified to use the PARFUM library and its features.

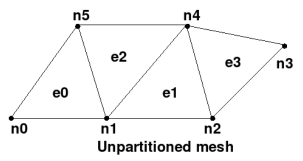


Fig. 3. An unpartitioned mesh.

4.1 Mesh Partitioning

Step 1: Compute a mapping of elements to partitions that produces approximately balanced partitions and minimizes the number of boundary elements. We start with an arbitrary mapping as an input to PARMETIS [21], a third-party MPI library for parallel partitioning that we use without modification via AMPI. As shown in Step 1 in Figure 4, PARMETIS takes in the connectivity of the mesh elements and produces a mapping of elements to partitions. Here multi-paradigm programming enables us to use a library developed by subject matter experts without having to re-implement it in another paradigm.

ParFUM divides the entities of a serial mesh into one partition per VP, using a memory-efficient parallel partitioner to handle large meshes with large numbers of partitions. Figure 3 shows a simple 2D mesh with triangular elements that is to be partitioned between two VPs. We use this simple mesh to illustrate the parallel partition algorithm:

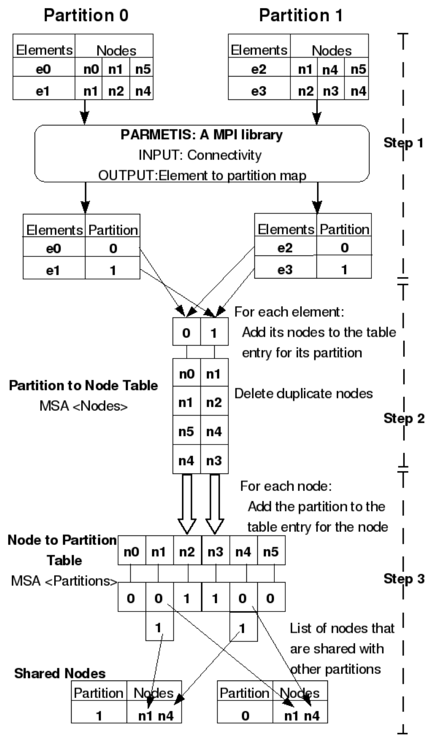


Fig. 4. Steps to partition a mesh

Step 2: Create partitions and send them their entity data. This is easy for elements because the mapping tells us exactly which partition each element belongs to. However, a node belongs to all partitions with an element adjacent to that node (for e.g.. $n1$ and $n4$ belong to both partitions in the example in Figure 4), so a node's ownership information is scattered across an unknown number of VPs on different processors. Collecting this information is simplified with a global table indexed by VP which stores all nodes owned by that VP's partition. This *Partition-to-Node* table has a list of nodes for each VP. Step 2 in Figure 4 shows that for each element, its nodes are added to the *Partition-to-Node* table at the entry for the element's partition (the partition to which PARMETIS has mapped this element in Step 1), with duplicate nodes being deleted.

Thus, in the first phase of this step the *Partition-to-Node* table is populated by all the VPs and in the second phase it is read by each VP. MSA, with its separate accumulate and read modes, is ideally suited for this. The *Partition-to-Node* table is implemented as a MSA array of node lists in accumulate mode. A message passing implementation would have been more complex since a VP does not know how many nodes to expect from other VPs. Before passing the nodes, each VP would have to tell all the others how many to expect. This would lead to less readable code, higher communication, and poorer performance.

Step 3: Find the nodes shared between different pairs of partitions. This can be calculated by creating a global table (called the *Node-to-Partition* table) that maps each node to all the partitions to which it belongs. As shown in Step 3 of Figure 4, for every node owned by a VP, the VP adds itself to the node's entry. After all the VPs have finished writing to the table, each VP looks at the entry each of its nodes to determine the other partitions sharing that node. This lets ParFUM build up a list of nodes shared by every pair of neighboring partitions. MSA is an ideal fit for Step 3 for the same reasons as in Step 2.

4.2 Mesh Adaptivity

ParFUM implements two types of mesh adaptivity: *incremental* and *bulk* mesh modification. Both approaches provide low-level primitives for *edge bisect*, *flip*, and *edge contract* operations. In the incremental case, these are self-contained parallel primitive operations that leave the mesh in a consistent state, updating all ghost layers and adjacencies as needed. The faster, lightweight bulk operations currently under development in ParFUM perform en masse mesh modifications before updating the ghost layers and adjacencies.

These primitives lock the affected mesh entities so that multiple operations can simultaneously modify adjacent areas of the mesh, using the ParFUM locking functionality described earlier. Once the affected region of the mesh is locked, modification of the mesh can proceed.

An example of the edge bisect primitive is shown in Figure 5. When such an operation takes place across a partition boundary, as shown by the thicker line in the figure, the communication is very localized and specific. In (a), we highlight an element that we wish to bisect along its longest edge, which happens to be on the boundary between partitions A and B. The first step is to lock the region of the mesh that will be modified. This is shown in (b) which highlights the original element, the neighbor across the edge to be split, and one adjacent element for each of these (which will need to have adjacency data updated). Locking requires the first element to send a message to all the affected elements and then suspend to wait for a response about the success or failure of the lock. If locking is successful, two new elements and a node will then be added by the operation. The new node will bisect the longest edge, and the two new elements will be along the new edge created between the new node and the node n in (b). This new topology is shown in (c). Because the node is shared, a record must be made

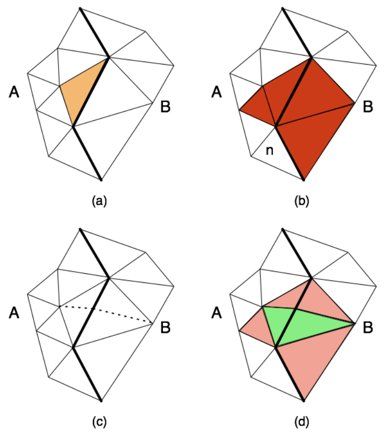


Fig. 5. Parallel edge bisection

of it on both partitions A and B. The bisect is performed on the first side of the edge to be bisected, and all information about the new element and node on that side is then transmitted to the adjacent element on partition B. This side completes the second half of the operation, updating all relevant adjacency data, and in return sends information about the new element created back to partition A where the adjacency of the new element is updated. In (d), the lightly shaded elements have connectivity and adjacency updates performed on them, while the darker shaded elements are new elements added to the mesh.

Due to the unpredictable nature of modification messages, it is impossible for a partition to predict when one of its neighbors will invoke adaptivity functions. To accomplish this with MPI, we would need a polling loop consisting of a wild card receive. Once a message is received, its type is checked and it is processed accordingly. This amounts to a re-implementation of some capabilities of the CONVERSE scheduler. However, the message-driven paradigm is ideally suited for this problem. The receiving partition does not need to expect incoming messages and processes messages as they are received.

CHARM++ is excellently suited to adaptivity algorithms, as operations are confined to regions of the mesh determined by the state of the solution at a particular point in time. These problems are highly irregular and dynamic, and as such are also a perfect match for the virtualization capabilities provided by CHARM++. Having multiple partitions per processor makes load balancing straightforward when refinement over particular partitions increases their load. Mesh adaptivity is achieved in PARFUM by associating a special type of chare array, called a *bound* array, with the AMPI VPs. Thus, each partition has a chare array element associated with it which performs the message-driven aspects of mesh adaptivity. The "bound" aspect of these elements means that when migration takes place, a VP is bound to its associated chare array element such that they always migrate together.

5 Multi-paradigm applications

PARFUM has been used to develop a number of parallel unstructured mesh applications [24]. These applications utilize the many features provided by PARFUM for faster development and better performance. Since each component of PARFUM was written using the paradigm or paradigms most suitable for it, PARFUM applications are examples of multi-paradigm parallel programming.

For example, TentPitcher is a novel algorithm for solving hyperbolic systems via the Spacetime Discontinuous Galerkin (SDG) method developed at the Center for Process Simulation and Design at UIUC [1]. In converting this algorithm to run in parallel, we made extensive use of the multi-paradigm capabilities of ParFUM.

This algorithm operates on a triangular mesh. Rather than picking a timestep interval and advancing each vertex in time by that interval, TentPitcher performs local solutions by choosing a single vertex at a local time minimum and moving it as far forward in time as possible based on causality constraints and error estimates. The result is a highly asynchronous algorithm in which many local solutions may be computed independently, with no global synchronization.

To efficiently solve problems with sharp features, such as shock propagation, high degrees of mesh refinement and coarsening are required. However, since this algorithm has no explicit timestepping, there is no natural opportunity to globally modify the mesh, as is standard practice in parallel adaptive finite element codes. In addition, typically only small areas of the mesh require modification and the vast majority of elements are unaffected, so doing global adaptivity will hurt performance. Therefore, we must perform all mesh modifications locally. Adaptivity operations like this are well suited to a CHARM++ approach, where the programmer can send a lock/unlock or adaptivity messages and invoke the appropriate function on another partition.

This code mingles parallel programming paradigms with ease. MPI calls are used for bulk communication such as checkpointing and output, while CHARM++ is utilized for locking and adaptive operations. Multiple paradigms coexist transparently, even within a single function. This leaves the programmer free to use whatever paradigm is most suitable at a very fine granularity, rather than choosing which paradigm is suitable at an application level.

6 Conclusions

Developing parallel software involves additional complexity over sequential software. The ability to develop modules in the most suitable parallel programming paradigm and to reuse them in applications that incorporate multiple paradigms improves programming productivity. The complex, adaptive multi-physics applications of the future require sophisticated and automated resource management. A multi-paradigm adaptive run-time system (ARTS) provides such support.

We demonstrated such an ARTS, called CONVERSE, that allows multiple work units on each processor and interleaves their execution based on availability of remote data and messages. These abilities are crucial to our goal of efficiently supporting tightly coupled multi-paradigm interoperability in a parallel programming environment. The utility of this approach was illustrated by 1) describing multiple paradigms implemented using our ARTS, and 2) showcasing a parallel unstructured mesh framework and two associated applications that leverage this multi-paradigm interoperability.

ARTS also have positive implications for new parallel programming paradigms. In order for new paradigms to come into use, programmers need to 1) hear about them, 2) learn how to use them, 3) find an implementation of them, and 4) not sacrifice re-usability of their code should they choose to adopt them. Incorporating new paradigms directly on the CONVERSE ARTS lowers the barrier to entry for the adoption of new paradigms by satisfying three of those criteria, providing the knowledge of and access to new paradigms in a way that will make them most immediately usable and subsequently re-usable in future codes.

We believe this approach is essential for productive and efficient parallel programming, particularly for the complex applications and petascale computing environments of the near future. We have been advancing the ARTS approach for over a decade, and hope that many new paradigms will be developed with it.

References

1. R. Abedi, S.-H. Chung, J. Erickson, Y. Fan, M. Garland, D. Guoy, R. Haber, J. M. Sullivan, S. Thite, and Y. Zhou. Spacetime meshing with adaptive refinement and coarsening. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 300–309, New York, NY, USA, 2004. ACM Press.
2. G. Almasi, C. Archer, J. G. Castanos, M. Gupta, X. Martorell, J. E. Moreira, W. Gropp, S. Rus, and B. Toonen. Mpi on blue gene/l: Designing an efficient general purpose messaging solution for a large cellular system.
3. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Dept, University of California, Berkeley, Dec 2006.
4. C.-C. Chiang. Low-level language constructs considered harmful for distributed parallel programming. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 279–284, New York, NY, USA, 2004. ACM Press.
5. S.-E. Choi and E. C. Lewis. A study of common pitfalls in simple multi-threaded programs. *SIGCSE Bull.*, 32(1):325–329, 2000.
6. J. Corbalan, A. Duran, and J. Labarta. Dynamic load balancing of mpi+openmp applications. *icpp*, 00:195–202, 2004.
7. L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1), January-March 1998.
8. Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A multi-platform co-array fortran compiler. In *Proceedings of the 13th International Conference of Parallel Architectures and Compilation Techniques (PACT 2004)*, Antibes Juan-les-Pins, France, October 2004.
9. T. El-Ghazawi and F. Cantonnet. Upc performance and potential: a npb experimental study. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.
10. A. Ferrari and V. S. Sunderam. Multiparadigm distributed computing with TPVM. *Concurrency: Practice and Experience*, 10(3):199–228, 1998.
11. I. Foster, B. Avalani, A. Choudhary, and M. Xu. A compilation system that integrates high performance fortran and fortran M. In *Proceedings 1994 Scalable High Performance Computing Conference*, 1994.
12. I. Foster and M. Xu. Libraries for parallel paradigm integration. In R. Kalia and P. Vashishta, editors, *Toward Teraflop Computing and New Grand Challenge Applications.* Nova Science Publishers, 1994.
13. A. Gursoy and L. Kalé. Performance and modularity benefits of messagedriven execution. *Journal of Parallel and Distributed Computing*, 64:461–480, 2004.
14. J. Hardwick. Porting a Vector Library: a Comparison of MPI, Paris, CMMD and PVM. Technical Report CMU-CS-94-200, Carnegie Mellon University, 1994.
15. C. Huang and L. V. Kale. Charisma: Orchestrating migratable parallel objects. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2007.

16. C. Huang, C. W. Lee, and L. V. Kalé. Support for adaptivity in armci using migratable objects. In *Proceedings of Workshop on Performance Optimization for High-Level Languages and Libraries*, Rhodes Island, Greece, 2006.
17. C. Huang, G. Zheng, S. Kumar, and L. V. Kalé. Performance evaluation of adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.
18. L. V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.
19. L. V. Kale, M. Bhandarkar, R. Brunner, and J. Yelon. Multiparadigm, Multilingual Interoperability: Experience with Converse. In *Proceedings of 2nd Workshop on Runtime Systems for Parallel Programming (RTSPP) Orlando, Florida - USA*, Lecture Notes in Computer Science, March 1998.
20. L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng. Programming Petascale Applications with Charm++ and AMPI. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press, 2008.
21. G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 35, 1996.
22. S. Kumar. *Optimizing Communication for Massively Parallel Processing.* PhD thesis, University of Illinois at Urbana-Champaign, May 2005.
23. A. L. Lastovetsky. mpc: a multi-paradigm programming language for massively parallel computers. *SIGPLAN Not.*, 31(2):13–20, 1996.
24. O. Lawlor, S. Chakravorty, T. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. Kale. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22(3-4):215–235.
25. J. Leichtl, P. E. Crandall, and M. J. Clement. Parallel programming in multiparadigm clusters. In *HPDC '97: Proc. of the 6th IEEE Int. Sym. on High Performance Distributed Computing*, page 326, Washington, DC, USA, 1997. IEEE Computer Society.
26. S. O. Marc Snir and etc. *MPI: The Complete Reference*, volume 1. The MIT Press.
27. J. Nieplocha and B. Carpenter. Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *J. Rolim eat al. (eds.) Parallel and Distributed Processing, Springer Verlag LNCS 1586*, 1999.
28. J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *J. Supercomputing*, (10):197–220, 1996.
29. L. Smith and M. Bull. Development of mixed mode mpi / openmp applications. *Scientific Programming*, 9(2-3/2001):83–98. Presented at Workshop on OpenMP Applications and Tools (WOMPAT 2000), San Diego, Calif., July 6-7, 2000.
30. J. Wang, J. Lai, Y. Zhao, and S. Zhang. Multi-paradigm and multi-grain parallel execution model based on smp-cluster. In *IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing*, pages 266–272, 2006.
31. G. Zheng. *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.
32. G. Zheng, O. S. Lawlor, and L. V. Kalé. Multiple flows of control in migratable parallel programs. In *2006 International Conference on Parallel Processing Workshops*, pages 435–444, Columbus, Ohio, August 2006. IEEE Computer Society.