A Unified Programming Model for Distributed Shared Arrays and Message-Driven Execution

Foo Bar Baz

Affiliation

Email Addresses

Abstract

One way to simplify parallel programming is to use programming models that sacrifice generality in exchange for simplicity, improved safety guarantees, and broader opportunities for optimization. When using such models, the composability of modules written using different models is critical to their utility. This paper presents enhancements to the limited *Multi-Phase Shared Arrays* (MSA) programming model that improve its safety and composability with the general-purpose message-driven execution model. We demonstrate the effectiveness of this approach using examples drawn from real-world programs which benefit from the simplicity of task-specific models without sacrificing performance.

Earlier versions of MSA excluded data races by constraining when and how client code could access any given shared array, but relied on runtime checking to identify constraint violations. These checks imposed a performance penalty on MSA code and left the possibility that violations in rare edge cases would go undetected. This paper describes an encoding of the constraints in question in the C++ type system, enabling static detection of most violations.

MSA's strict synchronization requirements previously limited the flexibility of composing multiple modules that would interact through a common shared array. The paper describes mechanisms used to lift that limitation.

1. Introduction

Message-driven execution is a convenient and effective model for parallel programming. The flow of control in message-driven systems is dictated by the arrival of messages. This asynchronous approach has proven effective in a variety of systems, including Active messages [von Eicken et al. 1992], Split-C [Culler et al. 1993], and Charm++ [Kale and Krishnan 1996], which has yielded a number of successful parallel applications [Bhatele et al. 2008, Bohm et al. 2008, Jetley et al. 2008]. In message-driven applications, the problem is decomposed into collections of communicating parallel objects, providing the opportunity for easy overlap of communication with computation and runtime-level optimizations such as automatic load balancing. In the loosely-coupled style encouraged by the message-driven model, the assembly of separate parallel modules in a single application requires adaptation only of the interfaces, rather than the more dramatic structural changes that might be required in single program multiple data (SPMD) models such as MPI and partitioned global address space (PGAS). In fine-grained and irregular applications, this style can be a necessity for attaining high performance.

However, the message-driven model is not an ideal choice for all classes of parallel applications. In cases where shared data is essential to concise expression of the algorithm, the code needed to explicitly communicate this shared data in a message-driven style can dominate the structure of the program, and overwhelm the programmer. In this situation, a shared address space programming model, as exemplified by the Global Arrays library [Nieplocha et al. 1996] and partitioned global address space (PGAS) languages [Koelbel et al. 1994, El-Ghazawi et al. 2005, Barriuso and Knies 1994, Numrich and Reid 1998, Charles et al. 2005] can be highly advantageous. Applications which require data structures too large to fit in memory local to one processor may also become dramatically simpler when expressed in a shared address space model. The ability to access data in the global address space without explicit messaging can offer substantial productivity benefits, and in many cases remote accesses can be effectively optimized by a compiler, as demonstrated by Co-Array Fortran [Dotsenko et al. 2004] and others. Programs which use explicit messaging can benefit substantially from the elimination of boilerplate messaging code which accompanies a switch to a shared address space model, particularly in cases where communication structure is irregular or data-dependent. However, shared memory applications are prone to data races, where concurrent access to globally visible data yields a non-deterministic result. These races can be difficult to identify and resolve without substantial expertise, degrading the productivity benefits of the global address space.

One approach to addressing the difficulties posed by data races is simply to constrain the programming model to disallow races altogether. In exchange for this safety guarantee, the programmer gives up the ability to express arbitrary parallel interactions, sacrificing the completeness of general-purpose programming models. This is the approach taken by Multiphase Shared Arrays (MSA) [DeSouza and Kalé 2004]. In MSA, each shared array is decomposed under user control and distributed across the parallel system by an underlying adaptive runtime system. These *MSAs* are restricted to a few simple access modes which guarantee freedom from data races. Although MSA is an incomplete programming model that cannot express all types of parallelism, in practice it is well-suited for a large variety of parallel algorithms, and for those problems, the expression is elegant and race-free.

The shared address space model and the message-driven model offer sharply contrasting views of how to write parallel programs. In shared address space models, the programmer always has a view of the entire system. Global coordination is a natural construct, and is used as necessary to maintain a globally consistent view of memory. In message-driven applications, the focus is local. Access to global state is extremely limited, and global coordination is cumbersome. To combine these programming models, the gap between the global, coordinated view offered by shared address space models and the local, asynchronous nature of message-driven programming must be reconciled.

In this paper we describe our approach to combining the shared memory and message-driven programming models in a single, unified system. Our approach offers access to a global address space through Multiphase Shared Arrays, where array accesses are divided into distinct phases which are demarcated by synchronization. MSA benefits greatly from this combined approach because it is an incomplete programming model, so although it is often well-suited to expressing particular algorithms or modules of a complex program, it is rarely an ideal choice for all portions of a complicated parallel application. The combination of MSA with a general-purpose message-driven programming model makes the safety and convenience of MSA available while allowing arbitrary parallel interactions. Based on the implementation of MSA available online, we have implemented new features to facilitate this combined approach and to detect more violations of the MSA programming model at compile-time rather than run-time. We call our enhanced version of MSA message-driven MSA (MDMSA).

We begin by describing the different MSA phases and the associated safety guarantees in section 2. We discuss the relationship between MSA and the underlying runtime system, including the effects of overdecomposition and asynchrony. We then describe our own enhancements to MSA. In section 3 we describe the addition of typed handles, which facilitate the discovery of MDMSA access violations at compile-time. Then in section 4 we describe a method for specifying global structure for applications using MSAs via high level orchestration code. This code dispatches message-driven constituent pieces, allowing simple, high-performance composition of components accessing shared arrays with other message-driven components without giving up the benefits of a simple, high-level view of application structure. Each of these enhancements is accompanied by a real-world example which demonstrates its use.

2. Multiphase Shared Arrays

Multiphase Shared Arrays (MSA) provide an abstraction common to several HPC libraries, languages, and applications: arrays whose elements are simultaneously accessible to multiple client threads of execution, running on distinct processors. These clients are user-level threads, typically many on each processing element (PE), which are tied to their PE unless explicitly migrated. Application code specifies the dimension, type, and extent of an array at the time of its creation, and then distributes a reference to it among client threads. Client threads access array elements by conventional subscripting and bulk-transfer operations. Each element has a particular *home* location, defined by the array's *distribution*, and is accessed through software-managed caches.

The initial development of MSA [DeSouza and Kalé 2004] was based on the observation that applications that use shared arrays typically do so in phases. Within each phase, all accesses to the array use a single mode, in which data is read to accomplish a particular task, or updated to reflect the results of each thread's work. MSA formalizes this observation, preventing race conditions by requiring *synchronization points* between *phases*, and allowing a single specifically-defined *access mode* during each phase.

Previous work with MSA led to applications in which the access mode of each phase was implicit in the structure of the code. Some `sync()` calls would be commented to indicate the new phase of the array, but this was not universal, and the comments were not always accurate. Thus, the implicit nature of the access modes is problematic, in that accesses not allowed in the current phase might go undetected until run-time.

Another problem with MSA is that proper synchronization requires all threads operating on the shared array to take part in all phases. This requirement forces any threads that do not access the array during a phase to persist and sit idle during that phase. It also requires all client code to track an identical sequence of synchronizations, tightly coupling client modules that could otherwise be oblivious to each other. In the message-driven model, where the code to handle a message may operate entirely within one phase, proper synchronization is a hurdle to the use of MSA.

The work presented in this paper addresses these problems. We enforce access modes at compile-time, by the use of statically-typed *handle objects*, enhancing the safety of the language and reducing some overheads. We also eliminate the requirement for all client threads to take part in end-of-phase synchronization, decoupling client code from the global life-cycle of an array.

The remainder of this section describes the programming model and implementation of MSA.

2.1 Data Decomposition and Distribution

The decomposition and distribution of data is an important consideration in the use of shared arrays. MSA decomposes arrays not into fixed chunks per PE, but rather into *pages* of a common shape. Developers can vary the shape of the pages to suit applications' needs. For example, a 10×10 array could be broken into ten pages, each in a 10×1 shape, or four pages of 5×5, or other suitable combinations. Thus, the library does not couple the number of pages that make up an array to the number of processors on which an application is running or the number of threads that will operate on that array. If the various parts of a program are *overdecomposed*, that is, decomposed into sufficiently more pieces than there are processors, the runtime system can hide latency by overlapping the communication of one piece with computation from another.

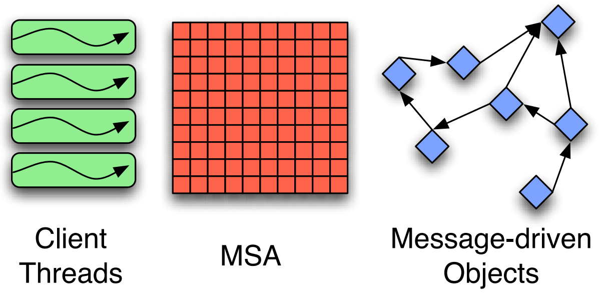
Once the array is split into pages, the pages are distributed among PEs. The pages are computational objects managed by the runtime system. This abstraction means that portions of a shared array can be managed by the same runtime infrastructure for object mapping and load balancing as other parts of a parallel program. Thus, each MSA offers control of the way in which array elements are mapped to pages, and the mapping of pages to PEs. This affords substantial opportunities to tune MSA code for both application and hardware characteristics. The user view of an MSA program and corresponding mapping by the runtime system are illustrated in figure 1.

One possibility enabled by decoupling data distribution from physical processor identity is that load balancers can treat 'hot' portions of the array the same way they would treat any other object that was performing intensive work or communication. These performance-critical segments can result from uneven access to the array by the computational threads.

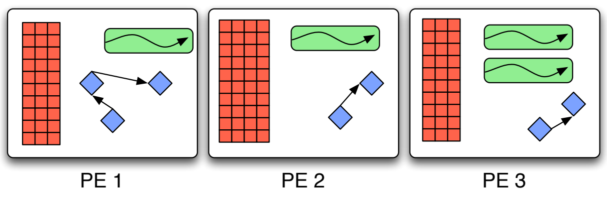
2.2 Caching

Data accessed from an MSA is cached by the runtime in buffers managed by the implementation. This approach differs from Global Arrays [Nieplocha et al. 1996], where the user must either explicitly allocate and manage buffers for pre-determined remote array segments or potentially incur remote communication costs for each array access. Runtime-managed caching offers several benefits, including simpler application logic, the potential for less memory allocation and copying, sharing of cached data among threads, and consolidation messages from multiple threads.

When an MSA is used by an application, each access checks whether the element in question is present in the local cache. If the data is available, it is returned and the executing thread continues uninterrupted. The programmer can also make prefetch calls



(a) The user view of an MSA application.



(b) One possible mapping of program entities onto PEs

Figure 1. The developer works with MSAs, client threads, and parallel objects without reference to their location, allowing the runtime system to manage the mapping of entities onto physical resources.

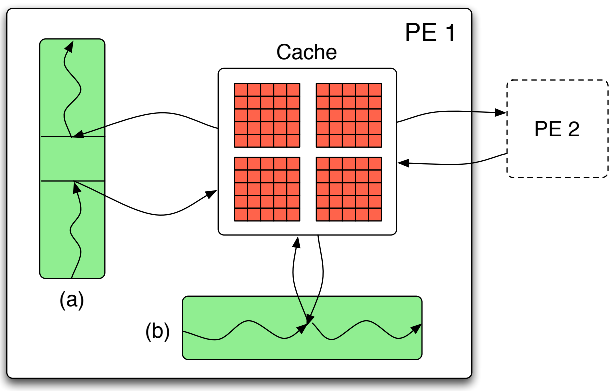


Figure 2. When thread (a) requests data which is not available in the local cache, the cache manager requests the data from its home PE and the thread blocks. While thread (a) waits, other local work can be done. When the data becomes available, thread (a) is unblocked and continues. When thread (b) requests data that is available in the local cache, it receives the data immediately and continues executing.

spanning particular ranges of the array, with subsequent accesses specifying that the programmer has ensured the local availability of the requested element. Bulk operations allow manipulation of an entire section of the array at once, as in Global Arrays.

When a thread requests data that is not present in the local cache, the cache object sends a request for it to its home page, then suspends the thread that made the request. At this point, messages queued for other threads are delivered. When the home page receives the request, it sends back data to the remote cache object. The cache manager receives this message and makes the blocked thread runnable. This process is illustrated in figure 2.

Each PE hosts a cache management object which is responsible for moving remote data to and from that PE. Synchronization work is also coalesced from the computational threads to the cache objects to limit the number of synchronizing entities to the number of PEs in the system. Depending on the mode that a given array is in, the cache managers will treat its data according to different coherence protocols, as in the Munin distributed shared memory system [Bennett et al. 1990]. However, the MSA access modes have been carefully chosen to make cache coherence as simple and inexpensive as possible. In no case are cache invalidations or unbuffered writes required at the time of access.

2.3 Access Modes

By limiting the programmer to the accesses allowed by well-defined modes and requiring synchronization from all threads which access the MSA to pass from one mode to another, race conditions within the array are excluded without requiring the programmer to understand a complicated or opaque memory model. The access modes MSA provides are suitable for a variety of common parallel access patterns, but it is not clear that these modes are the only ones necessary or suitable to this model. As we extend MSA further, we expect to discover more as we explore a broader set of use cases.

Read-Only Mode As its name suggests, read-only mode makes the array immutable, permitting reads of any element but writes to none. Remote cache lines can simply be mirrored locally, and discarded at the next synchronization point.

Write-Once Mode Since reads are disallowed in this mode, the primary safety concern when threads are allowed to make assignments to the array is the prevention of write-after-write conflicts. We prevent these conflicts by requiring that each element of the array only be assigned by a single thread during any phase in which the array is in write-once mode. This is checked at runtime as cached writes are flushed back to their home locations. Sophisticated static analysis could allow us to check this condition at compile time for some access patterns and elide the runtime checks when possible.

Accumulate Mode This mode effects a reduction into each element of the array, with each thread potentially making zero, one, or many contributions to any particular element. While it is most natural to think of accumulation in terms of operations like addition or multiplication, any associative, commutative binary operator can be used in this fashion. One example, used for mesh repartitioning in the ParFUM framework [Lawlor et al.], uses set union as the accumulation function.

The various access modes are illustrated in the following toy code that computes a histogram in array H from data written into array A by different threads:

```
A.syncToWrite();

for (int i = 0; i < N/P; ++i)
    A(tid + i*(P-1)) = f(x, i);

A.syncToRead();
H.syncToAccum();

for (int i = 0; i < N/P; ++i) {
    int a = A(i + tid*N/P);
    H(a) += 1;
}
```

Array subscripts are set off by parentheses, rather than the more conventional square brackets, so that syntax remains consistent when accessing arrays of dimension greater than one. This is a restriction imposed by C++'s different rules for overloading the subscript (`operator`

) and call (`operator()`) operators.

2.4 Safety Guarantees

The constrained structure of MSA accesses enforces a strict guarantee that no MSA operations will suffer from data races. The difficulty of reasoning about relaxed memory consistency models and the difficulty in avoiding, detecting, and resolving data races in shared memory programs makes this guarantee very attractive. This condition follows directly from the definition of the access modes. Clearly no data race can occur due to accesses from different modes, because there is synchronization between each phase. In read-only mode, there are no writes to produce possible races. In write-once mode, write-after-write conflicts are disallowed by definition. In accumulate mode, the associativity and commutativity of the accumulate operator guarantee that ordering of accumulate operations does not affect the final result. Owner-computes mode allows only local accesses, so races are again impossible.

2.5 Synchronization

A shared array moves from one phase to the next when its client threads have all indicated that they have finished accessing it in the current phase, by calling the synchronization method. During synchronization, each cache flushes modified data to its home location and waits for its counterparts on other PEs to do the same. Logically, client threads cannot access the array again until synchronization is complete. In MSA's SPMD-style code, as in MPI, this requires that threads explicitly wait for synchronization to complete sometime before any post-synchronization access. The present work lifts the need to explicitly wait by delivering messages when synchronization is complete. This advance is described as part of section 4.2.

3. Typed Handles

One drawback of the basic MSA model is its weak support for error detection. Because MSA is implemented as a C++ library, it has no compiler infrastructure to detect violations of its access modes until runtime. This lengthens the debugging process (while using potentially scarce parallel execution resources) and leaves the possibility that unexercised code paths contain serious errors. It also adds avoidable per-access runtime checks that each operation is consonant with the current access mode.

To address these problems, we have developed a way to detect a variety of access mode violations at compile-time in MDMSA by providing all array accesses through lightweight handle objects whose types correspond to the current mode of the array. The operations disallowed by an array's current access mode are excluded by omitting them from the corresponding handle type. Once a handle is used in synchronization, it is invalidated, so that future uses will dereference a `NULL` pointer. An example application using this idiom, parallel k-means clustering, is shown in section 3.1. We currently rely on run-time checks to detect threads synchronizing into different modes, and intersecting write sets during write-once mode.

Ideally, the programmer would access the shared array by a persistent name within any block of code. However, the use of the C++ type system to enforce access modes requires that we name a distinct variable in each phase so that it can have a distinct type. These handles then take the place of the array itself as an argument to the various operations. Each handle's type only defines the operations that are allowed in its associated mode, so that attempts to perform disallowed operations induce a compiler error.

There are numerous alternatives to our typed handle scheme, but they all suffer from either greatly increased complexity or the need for tools beyond a C++ compiler. With a more capable type system in C++, we could define the array itself with a linear type [Wadler 1990] such that synchronization operations would change the array's type in the same way that handle types are currently changed. This would also eliminate the need to verify that handles are still valid when they are used. If we wished to construct more complex constellations of allowed operations, an approach of policy templates and static assertions (such as provided by Boost [Maddock and Cleary]) would serve. Such policy templates would have a boolean argument for each operation or group of operations that is controlled. We feel that this construct creates less clear error reporting and complicates the implementation.

A more conventional approach to the problem of enforcing high-level semantic conditions is writing *contracts* [Helm et al. 1990] describing allowable operations. We avoided the use of contracts in MDMSA for three reasons. First, contracts require either an enforcement tool external to the compiler, or a language that natively supports contracts, such as Eiffel [Meyer 1992]. Second, these conditions would necessarily depend on state variables that aren't visible in the user code. Finally, we prefer a form in which the violation is local to the erroneous statement, rather than dependent on context.

Another approach to problems like this, common in the software engineering literature, is the definition of MDMSA's access modes and phases in a static analysis tool. Again, this implies enforcement by a tool other than the compiler. The rules so defined would necessarily be flow-sensitive, which makes this analysis fairly expensive and bloats the errors that would result from a rule violation.

3.1 Example: Parallel k-Means Clustering

In this example, each processor in a large-scale parallel application run has collected timing data for various segments of the program. At the end of the run, these metrics need to be reduced to avoid the slow output of an overwhelming volume of data. A two-part process identifies representative processors to report measurements for. The first part groups the processors by similarity of their execution profiles using k-means clustering, and the second part selects an exemplar and outliers from each cluster to report.

An initial implementation of this module was written in Charm++, but it was found that the large number of reductions with processors contributing to different parts of the output was too cumbersome. This same process would be fairly straight-forward to implement using common MPI functions such as `MPI_Allreduce`. However, the experimental nature of this analysis feature makes it desirable to try it several times on the same end-of-run data, with varying parameters. Runs could be executed one after another, in a loop over the input parameters, but this is wasteful of expensive machine time given that each run is largely communication-bound. As an alternative, runs for all of the input parameters could be executed together, with more complex bookkeeping code to track where each run's data lives and whether a given run has converged yet.

MDMSA admits straightforward solutions to all of these concerns. The communication pattern is expressed as adding to and reading from a shared matrix. Multiple concurrent runs are expressed as separately instantiated collections of objects, one for each set of parameters. Because each of the concurrent runs is expressed as an independent collection of objects, each run's sequential segments can be mapped to different processors, avoiding a bottleneck at a shared 'root' processor present in the Charm++ implementation.

```
 1  // One instance is created and called on each PE
 2  void KMeansGroup::cluster()
 3  {
 4    CLUSTERS::Write w = clusters.getInitialWrite();
 5    // PEs selected as initial seeds write their
 6    // positions into the array
 7    initialize(w);
 8
 9    // Put the array in Read mode
10    CLUSTERS::Read r = w.syncToRead();
11
12    do {
13      // Each PE finds the seed closest to itself
14      double minDistance = distance(r, curSeed);
15
16      for (int i = 0; i < numK; ++i) {
17        double d = distance(r, i);
18        if(d < minDistance) {
19          minDistance = d;
20          newSeed = i;
21        }
22      }
23
24      // Put the array in Accumulate mode,
25      // excluding the current value
26      CLUSTERS::Accum a = r.syncToExcAccum();
27      // Each PE adds itself to its new seed
28      for (int i = 0; i < numMetrics; ++i)
29        a(newSeed, i) += metrics[i];
30
31      // Update membership and change count
32      a(newSeed, numMetrics) += 1;
33      if (curSeed != newSeed)
34        a(0, numMetrics+1) += 1;
35      curSeed = newSeed;
36
37      // Put the array in Read mode
38      r = a.syncToRead();
39    } while(a(0, numMetrics+1) > 0);
40  }
```

Listing 1. Parallel k-Means Clustering implemented using an MSA named **clusters**. This function is run in a thread on every processor. First, processors selected as initial 'seeds' write their locations into the array (call on line 7). Then, all the processors iterate finding the closest seed (lines 13–22) and moving themselves into it (24–35). They all test for convergence by checking an entry indicating whether any processor moved (39).

The core code of the clustering process is shown in listing 1. It traces out the full life-cycle of a shared array, `clusters`, of summed per-processor performance metrics. The array has k columns, each of which represents a cluster of processors. The first `numMetrics` entries in each column are sums of actual measurements taken by the processors. There are two additional entries in each column, the first for the number of processors in the associated cluster (so that the metrics can be averaged), and the second for whether any of those processors joined that cluster in the current iteration.

In each iteration, the array alternates between a read phase, during which every processor finds the closest cluster to itself, and an accumulate phase, in which the processors contribute their position to their respective closest clusters. Every processor performs the same convergence test, checking whether any processor changed cluster membership during the current iteration.

The total implementation of the process described is ~610 lines of code, while the Charm++ implementation ran to ~800 lines of code before this new approach was taken. This represents a code-length reduction of 23.8%.

4. MDMSA and Message-Driven Components

Although MSA is well-suited to a variety of parallel algorithms, the limitations it places on parallel interactions make it unlikely that it will be an ideal choice for all parts of a large and complex application with many distinct components. Even if MSA could express all possible parallel algorithms, it would still be highly desirable to allow easy interaction between MSA components and components written using other models. The ability to freely compose MSA code with non-MSA code is essential for MSA's practical utility.

In particular, composability allows programmers to identify parts of their application which would benefit from shared memory and implement those parts in MSA. However, this type of composition is often highly cumbersome or impossible in MSA because of its phased synchronization requirements and use of SPMD-style threads, as shown in the next example.

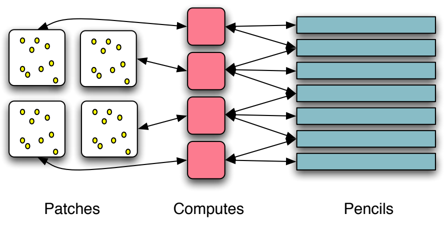
4.1 Long-range Force Calculation for Molecular Dynamics

Consider the case of computing long range interaction forces in a molecular dynamics simulation using particle mesh Ewald summation (PME), written in Charm++. In this application, particles are grouped into patches according to their physical locations. The positions of these particles are collected from each patch into an associated PME `compute` object, which is responsible for interpolating the charges from their locations in continuous space onto the points of a discrete grid composed of `pencils`, each of which represents a single row of the grid. The compute objects serve to offload as much work as possible from the patch objects for performance reasons. Once the charges have been interpolated, effective interaction potentials are calculated on the grid points by a spatial FFT, convolution with a kernel representing the force law, and inversion of the FFT. The `computes` then interpolate the force on each particle from the potentials at nearby grid points.

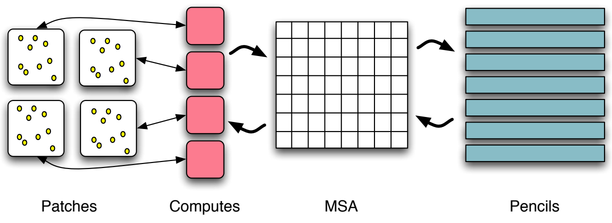
The difficulty in this code lies in the interpolation from PME computes to grid pencils and back. Each compute is responsible for a particular portion of the simulation region and tracks which pencils intersect that region. To perform interpolation, messages to each of these pencils are built using indirect array accesses to map physical coordinates onto the identity of the recipient. The pencils track how many computes have sent them messages so far and proceed only when all their computes have delivered messages. A similar procedure occurs in the opposite direction when forces are interpolated back onto the particles at the end of PME. This communication structure is shown in figure 3(a).

While this approach is acceptable, the large amount of bookkeeping involved in identifying the number and identities of message sources and recipients in each interpolation is undesirable, as it leads to convoluted code that is difficult to optimize. MSA offers the possibility of much simpler code by allowing the PME computes and pencils to read and write from a common shared array, avoiding the need for computes to track which pencils overlap them and vice-versa. The resulting communication structure is shown in figure 3(b).

Unfortunately, in the base MSA programming model we have presented, it is infeasible to replace the complex interpolation code with a simpler MSA alternative. The communicating PME compute and pencil objects are message-driven entities which are not expressed in terms of blockable MSA threads. Even if they could be modified easily, the interpolation process involves transferring control from one set of threads (the computes) to another (the pencils) when the array changes phase. This delegation of control cannot be expressed in the MSA model we have presented thus far: all threads involved in the MSA must participate in each phase. While



(a) In a purely message-driven application, PME involves complex messaging to accomplish the interpolation to and from the discrete grid.



(b) When MDMSA is used to manage the interpolation, no complex tracking of the relationship between compute objects and pencils is required.

Figure 3. Communication pattern of PME with and without MDMSA.

it might be possible to have all threads (both compute and pencil) take part in all MSA phases, this tightly couples two components unnecessarily and requires semantically meaningless synchronization points to be inserted into the code of each component.

This type of orchestration problem, in which an MSA would be used by multiple components in turn, must be solved in order to allow useful composition of MSA modules. To this end, we have developed a higher-level orchestration language for MDMSA that describes synchronization phases and flow of control over the lifetime of a single array, allowing effective composition of MSA modules with message-driven modules.

4.2 Orchestration and Client Code

In the present scheme, application developers explicitly encode the sequence or cycle of phases experienced by each kind of shared array in a high level orchestration code described in this section. For each phase, this code describes the access mode of the array and what (if anything) each participating client does with the array during that phase. Logically, each phase is started by the client threads executing during the prior phase signalling that they have finished working with the array, allowing it to synchronize. When the array has been synchronized, the clients listed in the orchestration code for the coming phase are sent messages indicating that the array is in the necessary mode.

Clients of a shared array are written in the message-driven style characteristic of Charm++ code. The problem to be solved is decomposed among collections of parallel objects, known as *chares*. These objects define *entry methods*, which can be invoked remotely and asynchronously by sending a message containing a method's input parameters. Each entry method typically encapsulates one logically distinct step of the parallel algorithm being expressed. As such, they are well-suited to express a chare's work during one phase of access to a shared array. The signatures of these methods comprise the bulk of the module interface definition files to which orchestration code will be added.

Beyond the benefits of reduced synchronization and better software engineering, this abstract, declarative view of an array's life-

```
1  MSA<3, double, summation>
2  ChargeGrid(ComputePmeMsa computes,
3             PmeMsaZPencil pencils) {
4    while (true) {
5      // Grid ready for forward interpolation
6      Accum { computes.recv_grid(@);  }
7      // First-dimension FFT and transpose to
8      // workers for other dimensions
9      Read  { pencils.grid_ready(@);  }
10     // Invert first-dimension FFT after
11     // receiving inverse transpose
12     Write { pencils.rgrid_ready(@); }
13     // Interpolate from grid
14     Read  { computes.ungrid(@);     }
15   }
16 };
```

Listing 2. The syntax of our proposed high-level orchestration language for message-driven MSA programs, illustrated using the PME example

cycle provides the opportunity to generate efficient code to activate clients and manage the array's synchronization. This arrangement also works well with another means of coordination in Charm++, Structured Dagger [Kale and Bhandarkar 1996a], which can be used to describe how the overall control flow of a chare derives from asynchronous invocation of a collection of entry methods.

4.3 Orchestration Code Structure and Semantics

Our goal in describing a language for MDMSA orchestration is to provide a centralized, consolidated view of the life-cycle of each kind of shared array used in a parallel program. This code would appear in the parallel module interface description files of a Charm++ application. Thus, it will be translated along with the rest of each module's interface to generate header and source files to compile into the program. To that end, the orchestration code for each kind of array pulls together a wide variety of information that would otherwise be scattered throughout the application:

- Name: What name will be used when creating this kind of array and referring to it in client code?
- Type: How many dimensions does this kind of array have? What data type are its elements? What accumulation operation is to be applied to those elements?
- Parameters: Who are the array's clients? What other parameters affect how the array is to be used?
- Life cycle: What happens with the array once it's created?

A sample showing all of these things as they would appear in an implementation of PME can be seen in listing 2. As can be seen, this largely mimics that structure of a C++ class or function definition. The first line begins with the token "`MSA`" to set this off from other kinds of coordination code used in Charm++. It is followed by effective template parameters describing the invariant type characteristics of this kind of array. The name and parameters follow through the end of line 3.

Body code, seen on lines 4–16, illustrates the life-cycle. Phases are indicated by blocks headed by their access mode. We use '`@`' as a shorthand for an appropriately-typed handle to the current array. Programmers can declare and operate on variables in the orchestration code, as in common sequential code, and use control structures like loops and conditionals to dictate the overarching flow of control. In this case, the PME process is simply looped until the program exits.

4.4 PME Implemented in Message-driven MSA

To better illustrate the structure of programs written using the newly-implemented message-driven semantics for MDMSA, listing 3 fully elaborates the PME implementation described earlier. Since translation of our proposed syntax is not yet implemented, we include a simple, manually-translated, centralized implementation of its functionality. The correspondence between phase blocks in the high level syntax translated code is quite direct. For example, the *Read* phase described on line 9 of listing 2 is translated to the synchronization call and asynchronous method invocation on lines 14 and 15 of listing 3, respectively.

The code structure directly reflects the shared memory interpolation algorithm. Computes and pencils are represented by `ComputePmeMsa` and `PmeMsaZPencil` objects, respectively. The orchestration code and `run` methods of the computes and pencils manage the overall flow of the algorithm, while the work of interpolation and force computation are done in `grid`, `ungrid`, `forwardFFT`, and `reverseFFT`. Message-driven code from elsewhere in the application calls the `recv_particles` method when particle data is available. The orchestration code calls `recv_grid` once all synchronization from the previous phase is finished. Once these methods have both been called, the compute's `run` method is free to call `grid`, which performs the actual interpolation onto the grid. The interpolation is performed using MDMSA bulk operations. When all computes finish `grid`, the orchestration code invokes `grid_ready` on the pencils, which go on to perform their 1-D FFT and transpose. The work of performing FFTs in the other dimensions and convolution with the force kernel are handled by message-driven objects elsewhere in the application, invoked by `transposeToYZ` on line 115. The convolved data returns and is subjected to a final inverse FFT on lines 57-59. When they are finished, the computes interpolate forces back onto particles in `ungrid`, and the iteration is complete.

4.5 Generating Efficient Parallel Code

The sample translation described in the previous section preserves the semantics and readability of the high-level code seen in listing 2, but there are potential scalability concerns in the resulting program. We seek to generate efficient code to implement this orchestration language, avoiding performance pitfalls and bottlenecks that would plague a naïve translation. The problems we foresee are as follows:

- High inter-phase latency as a result of having a critical path through a single processor ‘conducting’ the phase transition by receiving the message signalling end-of-synchronization and then subsequently broadcasting to clients of the next phase.
- Such a processor becoming a communication bottleneck from receiving and sending phase-transition messages to every other processor in the system.
- Control flow decisions in orchestration code magnifying the above.

The answer to these concerns lies in distributing or replicating execution of the orchestration code in parallel across all of the PEs. The per-PE cache objects need to know when synchronization is complete, and so the cache or another connected set of per-PE objects can directly activate client threads on each of their respective PEs. This essentially solves the first two concerns.

The final concern is more involved, but not too challenging conceptually. The data available to the high-level code comprise ‘local’ variables declared as part of the high level code and the contents of the subject array. Any operations on this information can be viewed as operations being performed collectively by the entire set of participating processors. Thus, a compiler for this

```
 1  // Hand-translated MSA orchestration code
 2  void PmeMsaOrch::run() {
 3    ChargeGrid grid =
 4      MSA3D<double, sum>(x, y, z);
 5
 6    ChargeGrid::Accum a =
 7      grid.getInitialAccum();
 8    while (true) {
 9      // Grid ready for forward interpolation
10      computes.recv_grid(a);
11
12      // 1st-dimension FFT and transpose to
13      // workers for other dimensions
14      ChargeGrid::Read r = a.syncToRead();
15      pencils.grid_ready(r);
16
17      // Invert first-dimension FFT after
18      // receiving inverse transpose
19      ChargeGrid::Write w = r.syncToWrite();
20      pencils.rgrid_ready(w);
21
22      // Interpolate from grid
23      ChargeGrid::Read r2 = w.syncToRead();
24      computes.ungrid(r2);
25
26      a = r2.syncToAccum();
27    }
28  }
29
30  // compute's Structured Dagger orchestration
31  void ComputePmeMsa::run() {
32    while (true) {
33      overlap {
34        // Received from MSA orchestration
35        when recv_grid(ChargeGrid::Accum a)
36          atomic { grid = a; }
37        // Received from associated patch
38        when recv_particles(vector<Particles> p)
39          atomic { particles = p; }
40      }
41      // Interpolate charges to the grid
42      atomic { grid(grid, particles); }
43      // Received from MSA orchestration
44      when recv_ungrid(ChargeGrid::Read r)
45        // Interpolate forces from the grid
46        atomic { ungrid(r, particles); }
47    }
48  }
49
50  // pencil's Structured Dagger orchestration
51  void PmeMsaZPencil::run() {
52    while (true) {
53      // Received from MSA orchestration
54      when grid_ready(ChargeGrid::Read r)
55        // Do first dimension FFT and transpose
56        atomic { forwardFFT(r); }
57      // Received from other dimension's pencils
58      when inv_transpose(vector<double> d)
59        atomic { reverseFFT(d); }
60      // Received from MSA orchestration
61      when rgrid_ready(ChargeGrid::Write w)
62        atomic { write_grid(w); }
63    }
64  }
```

Listing 3. Orchestration code for PME in message-driven MSA. The translation of the `ChargeGrid` array's orchestration can be seen on lines 1–28. Structured Dagger code representing the flow of control for the `compute` and `pencil` objects can be seen on lines 30–48 and 50–64, respectively.

```
// 'Grid' the particles' charges
void ComputePmeMsa::
grid(ChargeGrid::Accum a,
     vector<Particle> particles) {
  int x0, y0, z0;
  double dep[64];
  for (int i = 0; i < particles.size(), ++i)
  {
    // Get the position and interpolated
    // charges for the ith particle
    interpolate(particles[i],
                x0, y0, z0,
                dep);

    // Bulk interface -
    // contribute a 4*4*4 patch
    a.accumulate(x0  , y0  , z0,
                 x0+3, y0+3, z0+3,
                 dep);
  }
  a.syncDone();
}

// 'Ungrid' the forces on the particles
void ComputePmeMsa::
ungrid(ChargeGrid::Read r,
       vector<Particle> particles) {
  int x0, y0, z0;
  double potential[64];
  vector<Force> forces(particles.size());
  for (int i = 0; i < particles.size(), ++i)
  {
    getPosition(particles[i], x0, y0, z0);
    r.read(potential,
           x0  , y0  , z0,
           x0+3, y0+3, z0+3);
    forces[i] = anterpolate(particles[i],
                            potential);
  }
}

// FFT, transpose, iFFT, and write result
void PmeMsaZPencil::
forwardFFT(ChargeGrid::Read r) {
  double d[X_PENCIL_LENGTH];
  r.read(d,
         0,                 yPos, zPos,
         X_PENCIL_LENGTH-1, yPos, zPos);
  r.syncDone();
  fft1d(d);
  transposeToYZ(d);
}
void PmeMsaZPencil::
reverseFFT(vector<double> d) {
  data = d;
  ifft1d(data);
}
void PmeMsaZPencil::
write_grid(ChargeGrid::Write w) {
  w.write(0, yPos, zPos,
          X_PENCIL_LENGTH-1, ypos, zPos,
          data);
  w.syncDone();
}
```

Listing 4. Sequential code implementing the entry methods called by the orchestration code in listing 3. These describe the work of interpolation and reverse interpolation ('`anterpolate`') done by the `compute` objects, and the Fourier transform, transpose, and inverse Fourier transform done by the `pencil` objects.

code can decide to localize or replicate the work in question as appropriate. If, for example, a decision was to be made based on the contents of one entry in the array, a sensible translation would be for the home processor of that element to examine it, make the decision, and broadcast the result. A decision based on a reduction over some of the contents of the array could be implemented as an actual reduction across the PEs, with either the output value or the resulting decision distributed to all. In effect, we can use the techniques developed for compilers of the PGAS languages [Dotsenko et al. 2004] and apply them here.

5. Related Work

Software distributed shared memory (DSM) systems have been widely studied as a programming model for simplifying cluster programming. These systems commonly use hardware designed to support virtual memory page faults to detect non-local accesses and hide the underlying messaging. This approach is combined with relaxed memory consistency models to improve performance and compensate for false sharing, which can otherwise be debilitating when the unit of sharing is a memory page.

Treadmarks [Keleher et al. 1994] and its successor Cluster OpenMP [Terboven et al. 2008] are successful DSM implementations that closely mimic physical shared memory from the programmer's perspective. They impose no synchronization burden beyond what is needed in a general shared memory program. A multiple-writer coherence protocol ameliorates the performance penalty of false sharing by allowing non-conflicting writes by multiple threads within a single page. However, false sharing and the resulting cache invalidations remains a major source of performance degradation in these systems.

Munin [Bennett et al. 1990, Carter et al. 1995] introduces the idea of multiple cache coherence protocols based on common memory access patterns. For example, *read-mostly* objects are read far more often than they are written. Munin replicates read-mostly objects and updates their values via broadcast. The authors identify a variety of common access modes, including *write-once, result, producer-consumer,* and *migratory*. All objects that do not fall into an optimized category are handled with a general-purpose coherence protocol.

In Munin, each variable's mode is statically determined at compile-time. Unfortunately, Munin's virtual memory mechanism requires each shared variable to be located on its own page. Despite this handicap, the efficiencies provided by specialized access modes led to substantial performance gains.

These DSM systems are similar to MDMSA in that accesses to shared arrays do not include any information about where the accessed data is located. They differ in their lack of control over data decomposition. Within each page, MDMSA supports a variety of data layouts specified by the programmer, such as row- and column-major, to allow matching between the array's memory organization and access patterns of the application. Each page is dynamically mapped to a PE by a combination of programmer specification and runtime modelling and measurement. In contrast, Cluster OpenMP and Munin do not offer mechanisms to control data distribution, although Huang et al. have implemented mapping directives in OpenMP as part of an effort to implement OpenMP on top of Global Arrays [Huang et al. 2005].

Global Arrays [Nieplocha et al. 1996] (GA) is a partitioned global address space model that combines a global view of memory with explicit asynchronous *gets* and *puts* over RDMA. GA provides no caching or replication of remote data, preferring to allow the programmer to directly control all memory transfers. One-sided communication is used to access remote memory, which is staged into a buffer provided by the programmer. In the case of discontiguous array accesses, RDMA operations can be used directly to avoid unnecessary overhead. Like MDMSA, the unit of sharing in GAs can be controlled by the programmer and is not tied to cache line or memory page size. MDMSA's composability with other programming models is similar in spirit to GA's composability with MPI and discrete asynchronous tasks [Krishnamoorthy et al. 2006].

X10 [Charles et al. 2005] is a PGAS language with strong support for asynchronous operations and flexible synchronization. Its synchronization constructs, initially *clocks* and presently *phasers* [Shirako et al. 2008], are somewhat similar in spirit to the less restrictive synchronization we have introduced to MDMSA, although the syntax and implementation are significantly different.

Charisma [Huang and Kale 2007] and Structured Dagger [Kale and Bhandarkar 1996b] both aggregate low-level message-driven methods into unified higher-level code with the goal of increasing readability. Charisma targets the static dataflow case, while Structured Dagger provides constructs for expressing a directed acyclic control flow graph. These systems are both limited to simplifying programming in the message-driven model and do not address the issue of composing multiple programming models. Like MDMSA, they are specific to the Charm++ runtime system.

6. Conclusions and Future Work

A collection of interoperating languages for parallel programming allows concise expression of algorithms in their most appropriate form. Each language by itself may be incomplete, as long as it allows clean expression of a significant class of parallel algorithms. These languages can provide enhanced safety guarantees over fully general programming models. In implementing such a collection, composability of modules in different languages is critical to the overall utility of the collection for writing applications. In this paper, we considered the combination of the *Multi-Phase Shared Arrays* programming model, which sacrifices some flexibility of a shared memory system to prevent data races, with the general-purpose message-driven execution model.

To improve on the safety guarantees of MSA, we introduce a system of typed handle objects. An MSA's access mode in each phase of a parallel program defines the operations allowed on the array during that phase. In MSA, the programmer was previously responsible for manually keeping track of each array's phase and avoiding inappropriate accesses. Now, this state information is encoded in the type system and checked automatically at compile-time.

Building on the improved safety provided by typed handles, we tackled the problem of integrating MSAs into programs composed of message-driven objects. This is accomplished by constructing orchestration code that sends messages containing appropriate handles on a shared array to clients involved in each phase. In line with this, we modified the synchronization semantics such that client threads not participating in an entire series of phases need not block while waiting for synchronization to complete. Our enhanced version of the MSA programming model with typed handles is called message-driven MSA, or MDMSA

To demonstrate the advances described above, we presented a pair of examples drawn from real applications. The first, a parallel implementation of k-means clustering, demonstrates the use of typed handles in SPMD-style MDMSA code. The second, particle-mesh Ewald summation for molecular dynamics, is used to motivate our synthesis of MSA with message-driven execution and to illustrate use of the resulting design.

In the future, our plans for MDMSA extend in a few different directions. Implementing the translator for orchestration code is an important step to take. Additionally, while many optimizations of MDMSA are possible in its implementation, the underlying runtime, and specialized compilers, such work needs to be properly motivated and directed. Thus, we will undertake a comprehensive empirical study of the performance characteristics of programs using MDMSA. The results from this study will guide both our own efforts and those of application developers.

Acknowledgments

References

R Barriuso and A Knies. Shmem user's guide for c. *Cray Research Inc*, Jan 1994. URL `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/cmcl/link.iwarp/ccom/afs/OldFiles/archive/fx-papers/cri-shmem-users-guide.ps`.

John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90)*, pages 168–177, 1990. URL `citeseer.nj.nec.com/bennett90munin.html`.

Abhinav Bhatele, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and Laxmikant V. Kale. Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

Eric Bohm, Abhinav Bhatele, Laxmikant V. Kale, Mark E. Tuckerman, Sameer Kumar, John A. Gunnels, and Glenn J. Martyna. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, 52(1/2):159–174, 2008.

John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for reducing consistency-related communications in distributed shared memory systems. *ACM Transactions on Computers*, 13(3):205–243, Aug. 1995. URL `http://www.cs.utah.edu/flux/papers/munin.html`.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094852.

D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. Supercomputing '93*, 1993.

Jayant DeSouza and Laxmikant V. Kalé. MSA: Multiphase specifically shared arrays. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, September 2004.

Yuri Dotsenko, Cristian Coarfa, and John 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.

T El-Ghazawi, W Carlson, T Sterling, and K Yelick. UPC: Distributed shared memory programming. *books.google.com*, Jan 2005. URL `http://books.google.com/books?hl=en&lr=&ie=UTF-8&id=n4pknjxmh7EC&oi=fnd&pg=PP9&dq=%2522upc%2522+parallel&ots=dugC4cvld2&sig=RbsYqGBDyYpZqRQmcDe1K0vZGf4`.

Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. *SIGPLAN Not.*, 25(10):169–180, 1990. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/97946.97967.

Chao Huang and Laxmikant V. Kale. Charisma: Orchestrating migratable parallel objects. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2007.

L Huang, B Chapman, and Z Liu. Towards a more efficient implementation of openmp for clusters via translation to global arrays. *Parallel Computing*, Jan 2005. URL `http://linkinghub.elsevier.com/retrieve/pii/S0167819105001158`.

Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Massively Parallel Cosmological Simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

L. V. Kale and Milind Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996a.

L. V. Kale and Milind Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996b.

L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994. URL `citeseer.nj.nec.com/article/keleher94treadmarks.html`.

C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

Sriram Krishnamoorthy, Umit Catalyurek, Jarek Nieplocha, Atanas Rountev, and P. Sadayappan. An extensible global address space framework with decoupled task and data abstractions. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006. doi: 10.1109/IPDPS.2006.1639577.

Orion Lawlor, Sayantan Chakravorty, Terry Wilmarth, Nilesh Choudhury, Isaac Dooley, Gengbin Zheng, and Laxmikant Kale. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22(3-4):215–235.

John Maddock and Steve Cleary. *Boost.StaticAssert*. Boost Library Project. URL `http://www.boost.org/doc/libs/1_38_0/index.html`.

B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

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.

R Numrich and J Reid. Co-array fortran for parallel programming. *ACM SIGPLAN Fortran Forum*, 17, August 1998. URL `http://portal.acm.org/citation.cfm?id=289918.289920`.

Jun Shirako, David Peixotto, Vivek Sarkar, and William Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, Jun 2008. URL `http://portal.acm.org/citation.cfm?id=1375527.1375568`.

C Terboven, D Mey, D Schmidl, and M Wagner. First experiences with intel cluster openmp. *Lecture notes in computer science*, Jan 2008. URL `http://www.springerlink.com/index/h5448q6571t5116m.pdf`.

T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

Philip Wadler. Linear types can change the world! In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*. 1990.