# Using Shared Arrays in Message-Driven Parallel Programs

Phil Miller Aaron Becker Laxmikant Kalé
Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: {mille121,abecker3,kale}@illinois.edu

***Abstract*—**
**This paper describes a safe and efficient combination of the object-based message-driven execution and shared array parallel programming models. In particular, we demonstrate how this combination engenders the composition of loosely coupled parallel modules safely accessing a common shared array. That loose coupling enables both better flexibility in parallel execution and greater ease of implementing multi-physics simulations. As a case study, we describe how the parallelization of a new method for molecular dynamics simulation benefits from both of these advantages. We also describe a system of typed handle objects that embed some of the determinacy constraints of the Multiphase Shared Array programming model in the C++ type system, to catch some violations at compile time. The combined programming model communicates in terms of these handles as a natural means of detecting and preventing errors.**

## I. Introduction

Asynchronous message-driven execution is a convenient and effective model for general-purpose 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 [1], Split-C [2], and Charm++ [3], which has yielded a number of successful parallel applications ([4], [5], [6]). In message-driven applications, the problem to be solved 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 or non-overlapped (and hence non-performant) time or processor division 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 [7] and PGAS languages [8], [9], [10] 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 [11] and DeSouza [12]. 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 the communication structure is irregular or data-dependent.

This paper describes the combination of Charm++'s object-based message-driven execution with shared arrays provided by Multiphase Shared Arrays (Section IV). In particular, we demonstrate how this engenders the composition of loosely coupled parallel modules safely accessing a common shared array. That loose coupling enables both better flexibility in parallel execution and greater ease of implementing multi-physics simulations. As a case study, we describe how the parallelization of a new method for molecular dynamics simulation benefits from both of these advantages (Section V). We also describe a system of *typed handle objects* (Section III) that embed some of the constraints of the Multiphase Shared Array programming model in the C++ type system, to catch some violations at compile time. The combined programming model works with these handles as a natural means of detecting and preventing errors.

## II. Multiphase Shared Arrays

Multiphase Shared Arrays (MSA) [13] 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.

One problem common to shared memory applications are 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. The initial development of MSA 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, by requiring *synchronization points* between *phases*, and allowing one of several specifically-defined *access modes* (described below) during each phase. By establishing this discipline, programs using MSA are inherently deterministic[1]. However, in exchange for this guarantee, the programmer gives up some of the freedom of a completely general-purpose programming model.

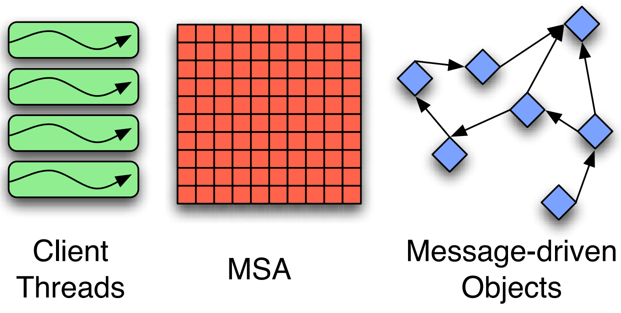
## A. Data Decomposition and Distribution

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 $10 \times 10$ array could be broken into ten pages, each in a $10 \times 1$ shape, or four pages of $5 \times 5$, 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.

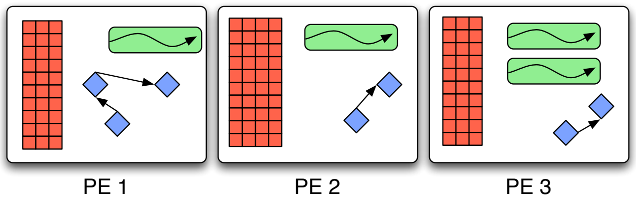
Once the array is split into pages, the pages are distributed among PEs. The pages are computational objects managed by the CHARM++ runtime system. This abstraction means that portions of a shared array can be managed by the same runtime infrastructure for object mapping and load balancing as other parts 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, or even specific threads of execution, 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

[1]This determinism holds up to the limits of operations on the contained data. For instance, floating-point numerical results may vary due to rounding error, and unsorted sets may return their elements in different orders.



(a) The user view of an MSA application.



(b) One possible mapping of program entities onto PEs

Fig. 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.

performance-critical segments can result from uneven access to the array by the computational threads.

## B. Caching

Data accessed from an MSA is cached by the runtime in buffers managed by the implementation. This approach differs from Global Arrays [7], 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 consolidating 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 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.

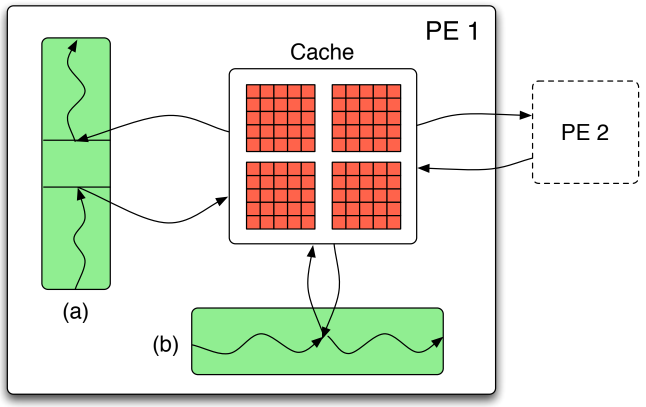


Fig. 2. When thread (a) requests data which is not available in the local cache, the cache manager requests the data from its home PE and thread (a) blocks. While thread (a) waits, other local work can be done. When the data becomes available, thread (a) is unblocked and continues. When thread (b) requests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

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 [14]. 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.

## C. 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.

1) **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.
2) **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.
3) **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 [15], 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);

// Done writing A; data can now be read
A.syncToRead();
// Get ready to increment entries in H
H.syncToAccum();

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

## D. 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.

## E. 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 SPMD-style MSA code, 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 in section IV.

## III. Typed Handles

One drawback of the basic MSA model is its weak support for error detection. 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 MSA's access modes is problematic. 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 by routing all array accesses through lightweight *handle* objects whose types correspond to the current mode of the array. The operations allowed by an array's current access mode are presented as methods in the corresponding handle type's interface. The synchronization methods return a handle in the new mode and mark the old handle as 'invalid'. An example application using this idiom, parallel k-means clustering, is described in section III-A. We currently rely on run-time checks to detect threads synchronizing into different modes, and intersecting write sets during write-once mode. Converting ParFUM's [15] mesh repartitioning code to use typed handles exposed previously undetected bugs that we subsequently fixed.

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 [16] 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 [17]) would serve. Such policy templates would have a boolean argument for each operation or group of operations that is controlled.

A more conventional approach to the problem of enforcing high-level semantic conditions is writing *contracts* [18] describing allowable operations. We avoided the use of contracts in MSA for three reasons. First, contracts require either an enforcement tool external to the compiler, or a language that natively supports contracts, such as Eiffel [19]. 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 MSA'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.

### A. 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.

MSA 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.

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.

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

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 5). Then, all the processors iterate finding the closest seed (lines 14–20) and moving themselves into it (22–33). They all test for convergence by checking an entry indicating whether any processor moved (37).

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 $23.8\%$.

## IV. Composing Shared-Array and Message-Driven Modules

In an SPMD environment, the combination of shared-array and message-passing interaction is fairly natural. Global Arrays and MPI are frequently used together, and Multiphase Shared Arrays [13] has previously been used toward the same end with AMPI [20] on top of Charm++'s message-driven runtime system. In that setting, issues of array synchronization and transfer of control are straightforward, if occasionally cumbersome. However, the message driven model has not previously been combined directly with shared arrays, despite MSA's implementation in terms of message-driven objects.

In the message-driven execution model, the asynchronous flow of control presents the possibility of unbridled non-determinism, which is detrimental to ease of program construction and verification. The Charm++ ecosystem offers various methods of constraining this non-determinism to obtain good performance without imposing an overwhelming burden on programmers [21], [22]. Shared arrays in general create another potential source of undesirable non-determinism, but MSA's programming model (described in Section II) requires that each array be accessed in a fashion that produces deterministic results with respect to that array. The basic challenges in the use of shared arrays by message-driven code, while retaining some useful safety guarantees, are dual: determining when any given object can access the array, and determining when the overall array's state has changed and what should happen to it next.

In the simplest cases, the challenges of combining Charm++ and MSA code have been easy to address since MSA was first implemented. In that case, there is only one collection of objects that accesses each array (or set of arrays), each running one persistent thread. The objects of this distinguished collection can still interact with others via messages, but access to their array(s) is fully encapsulated. The problem of asynchronously processed messages spurring improper array access is addressed by specifying the objects' control flow in SDAG [21]. With respect to the shared array, this is essentially the same SPMD arrangement as seen before. The extension of this style to multiple object collections of the same type[2] sharing an array is no different than one collection–we can view these objects as part of one larger collection with an extra index dimension.

As we introduce multiple modules that want to share an array, the complexity of the existing approach can increase dramatically. If we are to maintain MSA's guarantee of determinism, the code in each module must respect the phase and synchronization behavior of the others. The obvious but painful way to accomplish this is for each module to trace out the synchronization for every phase, even those in which it does not participate. There are two downsides to this approach: modules not participating in a phase will nevertheless be blocked while that phase proceeds, and the phase-change code of every module must be updated in lockstep, lest the programmer witness hangs, assertion failures, or wrong results (depending on the exact changes made).

Our approach to composing the two programming models is based on sending messages containing array handles to client objects that react by spawning a thread to perform one phase

[2]Really, any collections whose phases of array access are identical

worth of computation on the array. These messages serve to signal the client objects that the array has been synchronized properly and is in the mode they expect and depend on. At the end of the phase, these threads call a newly added method, `Handle::syncDone()`, to signal completion and deactivate the handle.

To drive the interactions of various collections of parallel objects with the shared arrays, we construct explicit coordination code that sends the messages mentioned above. This 'driver' code centralizes the knowledge of what phases an array will pass through and which objects participate in which phase. For objects not participating in a given phase, the driver code passes the same array handle as it sent to the active chares to a special entry method that simply calls `syncDone` and returns. This fulfills those objects' participation in the phase without interrupting the flow of their normal work.

Giving multiple independent modules access to common shared arrays presents new opportunities beyond coupling pieces of an application that wish to interact through a shared array. Different steps of computation can easily have widely-varying demands in terms of the work to be done, the amount of data to be accessed, and the pattern in which that access occurs. In as much as these steps don't depend on persistent state outside the array (exhibited in the case study in Section V, and seen in other applications in development), we can separate each phase into its own collection of objects, allowing us to vary the degree and nature of parallel decomposition and mapping, taking into account grain size, load balance, and data locality.

This approach supports a separation of concerns between application scientists writing the bodies of the individual methods that perform the computation and computer scientists focusing on the issues of efficient parallel execution. There are also software engineering benefits to this approach, in that distinct computational steps are distinguished and named as loosely coupled components in the resulting code. Decoupling these modules also enables testing each module and its accesses to shared arrays in isolation from the others.

## V. Case Study: Long-Range Forces in Molecular Dynamics

To direct and test our composition of shared-array and message-driven execution, we have developed a prototype parallel implementation of the novel long-range force-calculation method for classical molecular dynamics (MD) known as the Multi-Level Summation Method (MSM) [23]. MSM is a potential replacement to the popular particle-mesh Ewald (PME) summation that operates on a hierarchy of progressively-coarsened grids to compute the electric potential across the simulation space from the distribution of particles' electric charges. Compared to PME, MSM can produce similarly accurate results in $\mathcal{O}(n)$ time, while PME requires $\mathcal{O}(n \log n)$, where $n$ is the number of particles in the system. MSM has the additional practical advantage of an isoefficient 3D stencil structure, as opposed to PME's 3D FFTs.

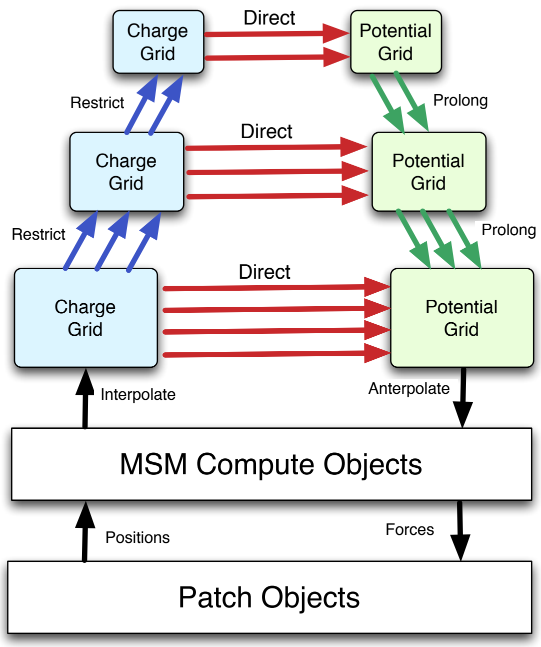


Fig. 3. The multi-level summation process as implemented using message-driven multiphase shared array code

Our prototype is built atop the Molecular3D package, a simplified MD benchmark modeled on the hybrid spatial-force decomposition used by NAMD [4]. In this decomposition, the particles are divided among a collection of `Patch` objects, each responsible for the particles within a patch of the simulation box. The forces among the particles are computed by a collection of `Compute` objects that receive particle positions from one or two patch objects, calculate the pair-wise electrostatic interactions among the particles, and transmit the resulting forces back to the patches. The patches are responsible for integrating the net forces on the particles to obtain their position and velocity for the next time step.

We incorporate MSM into this existing structure by adding a new set of `Compute` objects responsible for interpolating particles' charge contributions onto the finest charge grid and interpolating the forces on the particles back from the finest potential grid. The restriction, prolongation, and direct sum steps are each performed by separate chares at each level. Each grid is represented by an MSA. The entire process is illustrated in figure 3.

It is important to note that this process exposes parallelism at two distinct levels: each of the computational steps at each level can begin as soon as its input data is available, and the work of each step can be divided among a number of objects. In our prototype, we have only implemented the first, by creating a separate object for each step of the process, so that we can map these steps onto separate processors.

We present some extracts from our prototype code in listing 2. Lines 1–16 show the coordination code (written in Structured Dagger [21]) for the work each `MSMCompute` does

```
1  entry void MSMCompute::step() {
2    when interact(ParticleDataMsg *msg),
3         contributeCharges(Accum charges) {
4      particles = msg;
5      computeChargeGrid(charges);
6      charges.syncDone();
7    }
8
9    when readPotentials(Read potentials) {
10     forceMsg = new ParticleForceMsg;
11     computeForces(potentials, forceMsg);
12     patch.receiveForces(forceMsg);
13     potentials.syncDone();
14     delete particles;
15   }
16 }
17
18 entry void Energy::calculate(Read charges,
19                              Read potentials)
20 {
21   double u = 0.0;
22
23   for(int i = 0; i < grid_x[0]; ++i)
24     for(int j = 0; j < grid_y[0]; ++j)
25       for(int k = 0; k < grid_z[0]; ++k)
26         u += charges(i,j,k) * potentials(i,j,k);
27
28   contribute(u); // Contribute to a reduction
29
30   charges.syncDone();
31   potentials.syncDone();
32 }
```

Listing 2. Two illustrative functions: a) One timestep's work for the `Compute` objects that interface between the `Patch`es and the MSM computation. b) A method that computes the system's long-range potential energy by summing the point-wise product of the charge and potential grids.

in a single run of the MSM process. First, it waits for messages containing particle positions via its `interact` entry method and an accumulate-mode handle on the charge grid via its `contributeCharges` entry method (lines 2–3). Once both have arrived, it stores the particles' positions, interpolates those particles' charges onto the grid, and synchronizes the grid to indicate completion. Later, when the potential grid is ready to have forces anterpolated back to the particles, as indicated by receiving a read-mode handle (line 9), it prepares and sends a message reporting the forces it calculates back to its corresponding `Patch`, synchronizes the potential grid, and discards the old particle positions.

One of the computations internal to MSM, of the potential energy of longe-range interactions, can be seen in lines 18–32. This entry method receives read-mode handles to both finest-resolution grids (18–19), computes their pointwise product (or this `Energy` objects's portion thereof, when we decompose this step), contributes it to a reduction, and synchronizes the grids.

Small-scale timing results for our implementation can be seen in figure 4. There is a single chare performing each step of the method at each level, and so there is insufficient decomposition to attain good overlap. Our future work includes developing this further and integrating it with NAMD.

| Processors | Time (s) |
|---|---|
| 1 | 87 |
| 2 | 56 |
| 3 | 48 |
| 4 | 39 |

Fig. 4. Performance results for an untuned preliminary implementation of multi-level summation in Molecular3D using MSA

## VI. 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 [24] and its successor Cluster OpenMP [25] 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 [14], [26] 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 MSA 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, MSA 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 modeling 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 [27].

Global Arrays [7] (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 MSA, the unit of sharing in GAs can be controlled by the programmer and is not tied to cache line or memory page size. MSA's composability with other programming models is similar in spirit to GA's composability with MPI and discrete asynchronous tasks [28].

X10 [10] is a PGAS language with strong support for asynchronous operations and flexible synchronization. Its *clock* synchronization construct, and the subsequent proposal of *phasers* [29], are somewhat similar in spirit to the less restrictive synchronization we have introduced to MSA, although the syntax and implementation are significantly different. Recently, proposals to add more asynchronous mechanisms to UPC have appeared as well [30].

## VII. Conclusion

In this paper, we consider 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. We describe an extension of MSA's implementation to enforce its constraints more inexpensively at compile-time. We then build on this new mechanism to compose MSA safely with existing message-driven code.

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.

To demonstrate the advances described above, we present 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 MSA code. The second, multi-level summation for molecular dynamics, motivates our synthesis of MSA with message-driven execution and illustrates the resulting design.

## Acknowledgements

We would like to thank Chris Harrison for his input on MSA's development, and Abhinav Bhatelé, Eric Bohm, Isaac Dooley, Lukasz Wesolowski, and Ramprasad Venkataraman for their helpful comments on this paper. The initial implementation of MSA was developed by Jayant DeSouza and Orion Lawlor. The authors of this work were supported by NSF grants ITR-HECURA-0833188 and OCI-0725070, and the UIUC/NCSA Institute for Advanced Computing Applications and Technologies.

## References

[1] T. von Eicken, D. Culler, S. Goldstein, and K. 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.

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

[3] L. V. Kale and S. Krishnan, "Charm++: Parallel Programming with Message-Driven Objects," in *Parallel Programming using C++* (G. V. Wilson and P. Lu, eds.), pp. 175–213, MIT Press, 1996.

[4] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale, "Overcoming scaling challenges in biomolecular simulations across multiple platforms," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

[5] E. Bohm, A. Bhatele, L. V. Kale, M. E. Tuckerman, S. Kumar, J. A. Gunnels, and G. 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*, vol. 52, no. 1/2, pp. 159–174, 2008.

[6] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively parallel cosmological simulations with ChaNGa," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[7] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," *J. Supercomputing*, no. 10, pp. 197–220, 1996.

[8] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, "UPC: Distributed shared memory programming," *books.google.com*, Jan 2005.

[9] R. Numrich and J. Reid, "Co-array fortran for parallel programming," *ACM SIGPLAN Fortran Forum*, vol. 17, August 1998.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. 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*, (New York, NY, USA), pp. 519–538, ACM, 2005.

[11] 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.

[12] J. DeSouza, *Jade: Compiler-Supported Multi-Paradigm Processor Virtualization-Based Parallel Programming*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2004.

[13] J. DeSouza and L. 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.

[14] J. K. Bennett, J. B. Carter, and W. 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)*, pp. 168–177, 1990.

[15] 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*, vol. 22, pp. 215–235, September 2006.

[16] P. Wadler, "Linear types can change the world!," in *Programming Concepts and Methods* (M. Broy and C. Jones, eds.), 1990.

[17] J. Maddock and S. Cleary, *Boost.StaticAssert*. Boost Library Project.

[18] R. Helm, I. M. Holland, and D. Gangopadhyay, "Contracts: specifying behavioral compositions in object-oriented systems," *SIGPLAN Not.*, vol. 25, no. 10, pp. 169–180, 1990.

[19] B. Meyer, "Applying "Design by Contract"," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[20] S. Chakravorty, A. Becker, T. Wilmarth, and L. V. Kalé, "A Case Study in Tightly Coupled Multi-Paradigm Parallel Programming," in *Proceedings of Languages and Compilers for Parallel Computing (LCPC '08)*, 2008.

[21] L. V. Kale and M. Bhandarkar, "Structured Dagger: A Coordination Language for Message-Driven Programming," in *Proceedings of Second International Euro-Par Conference*, vol. 1123-1124 of *Lecture Notes in Computer Science*, pp. 646–653, September 1996.

[22] 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.

[23] D. J. Hardy, "Multilevel summation for the fast evaluation of forces for the simulation of biomolecules," tech. rep., Department of Computer Science, University of Illinois at Urbana-Champaign, May 2006.

[24] 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*, pp. 115–131, 1994.

[25] C. Terboven, D. Mey, D. Schmidl, and M. Wagner, "First experiences with intel cluster openmp," *Lecture notes in computer science*, Jan 2008.

[26] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Techniques for reducing consistency-related communications in distributed shared memory systems," *ACM Transactions on Computers*, vol. 13, pp. 205–243, Aug. 1995.

[27] 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.

[28] S. Krishnamoorthy, U. Catalyurek, J. Nieplocha, A. 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.

[29] J. Shirako, D. Peixotto, V. Sarkar, and W. 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.

[30] A. G. Shet, V. Tipparaju, and R. J. Harrison, "Asynchronous programming in upc: A case study and potential for improvement," in *Workshop on asynchrony in the PGAS model*, 2009.