PGAS in the Message-Driven Execution Model

Aaron Becker
abecker3@illinois.edu

Phil Miller
mille121@illinois.edu

Laxmikant V. Kalé
kale@illinois.edu

ABSTRACT

Asynchrony is increasingly important for high performance on modern parallel machines. A common approach to providing asynchrony in PGAS languages is to add additional language constructs to support asynchronous execution. In this paper we describe Multiphase Shared Arrays (MSA), a restricted PGAS programming model that takes the opposite approach, layering PGAS semantics over a fundamentally asynchronous runtime environment. We sidestep many of the difficulties of asynchronous programming through a discipline that offers desirable safety properties while exposing opportunities for optimization at multiple levels. We retain generality by offering composability with general purpose parallel programming models.

1. INTRODUCTION

Partitioned global address space (PGAS) programming languages and libraries have become increasingly common in parallel computing [31, 15, 12, 36, 1, 37, 9]. The ability to refer to any data in the global address space is a substantial convenience, and in many cases a compiler can effectively optimize these remote accesses, as demonstrated in the case of Co-Array Fortran [14], among others. Unfortunately, these systems are often prone to data races, in which the result of multiple modifications to globally visible data produces a non-deterministic result. Subtle data races can be difficult to diagnose and fix without sophisticated knowledge of the memory model of the programmed system, and the effort needed to avoid data races detracts significantly from the programmability and convenience of PGAS models.

One possible approach to address the difficulties that data races present is simply to constrain the programming model to disallow races altogether. In exchange for this safety guarantee, the programmer must give up the ability to express arbitrary parallel interactions, sacrificing the completeness of general-purpose programming models. Our model for PGAS programming, Multiphase Shared Arrays (MSA), embodies this approach. In MSA, each global array is decomposed under user control and distributed across the parallel system by an underlying adaptive runtime system. These arrays, which are themselves referred to as 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 that it can express, the expression is elegant and race-free.

Because MSA is an incomplete programming model, it is often well-suited to expressing particular algorithms or modules of a complex program, but it is rarely an ideal choice for all portions of a complicated parallel application. Our design philosophy is therefore to provide the programmer with a collection of incomplete models backed by complete models. Each module of a program can then be written in the model most appropriately suited to express it.

To be successful, this approach requires support for parallel composition and interoperability. MSA is based on the Charm++ runtime system, whose message driven scheduler provides such composability naturally [26]. Computational entities can be defined using any of a variety of programming models, and the execution of these entities is mediated by an event-driven scheduler. The scheduler will automatically interleave the execution of the computational entities, which is a key to supporting compositionality.

The coupling of MSA to the Charm++ runtime system means that MSA communication is based on asynchronous messaging. The performance benefits of asynchronous parallelism have been shown throughout its long history, for example in the Chare Kernel [23], Active messages [40], Split-C [10], and Charm++ [27]. Charm++ also addresses the issue of resource management by supporting overdecomposition into a large number of objects, which are mapped dynamically to hardware resources by an intelligent runtime system. Together, asynchrony and overdecomposition engender automatic adaptive overlap of communication and computation, while the runtime automates load balancing. This approach has been quite successful in parallelizing production quality applications [5, 6, 21].

A preliminary version of MSA was developed a few years ago and published in a short workshop paper [12], though further development of MSA was stopped until recently due to a lack of funding. This paper provides a rationale for MSA and its relationship with its asynchronous context, and includes some recent improvements in compile-time detection of violation of safety properties. In this paper, we briefly describe the MSA programming model and its associated safety guarantees. We discuss the relationship between MSA programs and the underlying runtime system, including the effects of overdecomposition and asynchrony. We demonstrate safety checks that can be supported at compile time without requiring a compiler that has knowledge of MSA, and illustrate the utility of MSA through example code. We

also describe optimizations possibilities with varying levels of programmer, compiler, and runtime support.

2. PROGRAMMING MODEL

Multiphase shared arrays provide an abstraction common to many HPC libraries, languages, and applications: arrays whose elements are simultaneously accessible to multiple threads of execution. Application code specifies the dimension, type, and extent of an array at the time of its creation, distributes a reference to it among relevant threads, then accesses array elements by conventional subscripting operations. Each element has a particular *home* location, defined by the array's *distribution*, and is accessed through software-managed caches on other processing elements (PEs). Race conditions are prevented by requiring synchronization points between operations that may conflict. This requirement is enforced via *access modes*. In code accessing an MSA, asynchrony is provided by executing more threads than processors ('overdecomposition'), and blocking threads when they request data that is not in the local cache or when they are waiting to synchronize. The communication associated with blocked threads can be overlapped with both other MSA threads and with the execution of other program modules which share the same scheduler.

2.1 Data Decomposition and Distribution

An important consideration in the use of MSA is how data residence is distributed. We decompose arrays not into fixed chunks per PE, but rather into *pages* of a common shape. The number of pages is not coupled to the number of PEs or the number of accessing threads, and overdecomposing the array into a large number of pages offers increased opportunities for latency hiding. At its simplest, the pages can take a blocked row- or column-major arrangement, with the block shape determined by the library to suit the underlying memory and communications hardware. At present, we allow the application programmer to manually specify one of a few simple decompositions, but we plan to extend this to cover more complicated cases as application needs dictate.

Once the data is split among pages, the pages are distributed among PEs. The pages are computational objects like any others in the program. This abstraction means that runtime infrastructure for object mapping and load balancing applies just as well to chunks of a shared array as any other part of a parallel program. Thus, each MSA offers control of the way in which array elements are mapped to pages, the page size, and the mapping of pages to PEs. This affords substantial opportunities to tune MSA code for both application and hardware characteristics.

2.2 Caching

Data accessed from an MSA is cached by the runtime in implementation-managed buffers. This approach differs noticeably from Global Arrays [36], 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, potentially less memory allocation and copying, sharing of cached data among threads, and consolidation of multiple threads' communications.

In the most straightforward use by application programmers, 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. Otherwise, the thread will request it and block. 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.

2.3 Access Modes

MSA derives its name from its most distinctive feature: legal accesses to each array are defined by a particular access mode during each inter-synchronization 'phase'. By limiting the programmer to the accesses allowed by well-defined modes and requiring synchronization to pass from one mode to another, we exclude race conditions within the array without requiring the programmer to understand a complicated or invisible set of semantics. Operations not conforming to an array's current mode are not permitted within the MSA programming model, and we are exploring various ways to flag such accesses as early as possible and with minimal programmer burden.

We will now present the set of access modes available in MSA. These modes are suitable for a variety of common parallel access patterns, but we make no claim that these modes are the only ones necessary or suitable to this model. We expect to discover more as we explore a broader set of use cases.

2.3.1 Read-Only Mode

As its name suggests, read-only mode makes the array immutable, permitting reads of any element but writes to none.

2.3.2 Write-Once 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 this 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 where possible.

2.3.3 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 in the mesh repartitioning code of the ParFUM framework [33], uses set union as the accumulation function. We have found use cases for both including and excluding the existing value of each entry, as opposed to starting from the reduction's identity. Both options are equally easy to implement, but we currently support only exclusive reductions.

2.3.4 Owner-Computes Mode

This mode restricts access not by disallowing certain operations, but rather by restricting the range of accessible indices. Specifically, computation in owner-computes mode is carried out by the page objects themselves, rather than the external threads of execution. This limits the set of accessible indices to a single page and preserves encapsulation. When the threads synchronize to owner-computes mode, they specify a function object that each page calls on its local chunk of the array. Because of the limitation to working on one page, the programmer can safely access the underlying storage of each page of the array without restriction. In combination with a well-specified memory layout, this mode enables the local use of fast library implementations of common algorithms, such as BLAS routines [34] and FFTs [16] or offload to accelerators such as GPUs [42] and Cell SPEs [32].

2.4 Synchronization

Threads of execution change the access mode of an array by synchronizing with all other threads accessing that array. When threads synchronize, they must specify the access mode that the array should enter, and their specifications must match. This synchronization should not be seen in the same light as a barrier, since the only threads involved are those holding references to the array entering a new phase. In this sense, each array can be thought of as having an associated 'clock' (as in X10 [9]) that each accessing thread is registered on. We currently do not impose a requirement to synchronize on all arrays simultaneously, as we feel this would unduly restrict sharing of arrays among threads performing different tasks. We are considering adding batched and split-phase synchronization to reduce overhead and expose more opportunities to overlap communication with computation.

2.5 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.

2.6 Composability

For an incomplete programming model such as MSA, integration with other models is essential to accommodate situations where a general-purpose programming model is needed. MSA is integrated with the Charm++ runtime system, which provides both interoperability between MSA and other models and a way of turning synchronous MSA semantics into asynchronous messages.

MSA's use of the Charm scheduler makes it very natural to integrate MSA code into applications based on Charm++ or on AMPI, an adaptive MPI implementation built on the Charm runtime [19]. MSA, Charm++, and AMPI can all coexist in the same application and take advantage of the same runtime facilities, such as topology aware object mapping [3] and measurement-based load balancing [25]. This approach has also been used in the development of Charisma [18], another incomplete programming model, targeted at applications with static data flow. Other incomplete models could make use of the same infrastructure, facilitating the use of multiple incomplete programming models, each targeted at a specific class of problems.

3. IMPLEMENTATION

There are a few facets of the implementation of MSA that offer important insights. The API uses the C++ type system to partially enforce the access modes described above. The intelligent caching of remote data is a feature we feel is important to effective use distributed arrays. Finally, the object-based decomposition of the array itself exposes it to the same runtime load and communication optimization that other components benefit from.

3.1 Programming Interface

One of our goals in the current effort is to detect erroneous programs as early as possible, to avoid time wasted on debugging with potentially scarce parallel execution resources. To this end, we aim to detect errors at compile-time where possible, and eagerly at run-time where necessary. Specifically, operations disallowed by an array's current access mode are excluded by checks we implement through the C++ type system. We currently rely on run-time checks to detect threads synchronizing into different modes, intersecting write sets, and attempts to access an array in anything other than its current 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. We do this through handles whose types represent the access mode that the array is currently in. 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 this construct, but they all fail to satisfy for one reason or another. With a more capable type system in C++, we could define the array itself with a linear type [41] 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 [35]) 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* [17] describing allowable operations. There are a few down-sides to this approach for our purposes. For one, contracts require either an enforcement tool external to the compiler, or a language that natively supports contracts, such as Eiffel. For another, 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 the MSA modes 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.

3.2 Cache

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 Munin [2]. However, the MSA access modes have been carefully chosen to make cache coherence as simple and inexpensive as possible.

In read-only mode, remote cache lines can simply be mirrored locally, and discarded at synchronization time. In write-once mode, all writes to remote data can be buffered until the end of the phase, minimizing communication costs. Runtime verification that the write-once guarantee has not been violated takes place within the home objects (see below) when remote writes are committed at the end of the phase. Similarly, accumulations are performed in a local buffer, and the result is consolidated with the remote data during the phase change. In no case are cache invalidations or unbuffered writes required.

3.3 Asynchronous Message-Driven Execution

The elements of each MSA are distributed into pages, each of which is managed by a home object. The page size and array layout can be controlled by the programmer. These pages are the place to which locally buffered data is synchronized during phase changes, and are responsible for array modifications in owner-computes mode. The caches communicate with each other and with the home objects via messages delivered to asynchronously executed *entry methods*. All of these objects can be placed automatically by the Charm++ runtime or placed explicitly by the application. The Charm scheduler is responsible for scheduling all threads and the entry methods of the pages, as shown in figure 1.

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

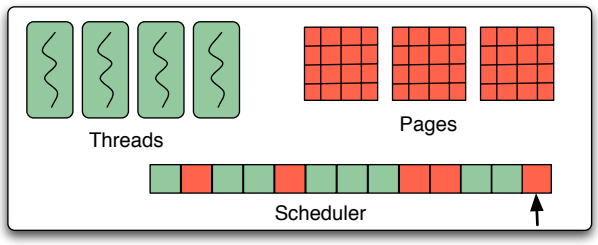

Figure 1: On each PE, the Charm scheduler manages the execution of each MSA page and each accessing thread. Whenever a thread blocks, the next entry method is scheduled. Remote requests for data are served by the page that owns the data.

When a computation thread synchronizes, it notifies its local cache manager that it is at a synchronization point and suspends. The cache manager keeps count of how many local threads are waiting to synchronize. When all threads reach that point, the cache manager starts synchronization with its siblings on other PEs. This entails flushing dirty array entries to their home locations and waiting for the other cache managers to finish doing the same. When all of this is complete, the blocked computational threads are awakened.

One advantage of decoupling data distribution from physical processor identity is that the load balancer can treat 'hot' portions of the array the same way it would treat any other object that was doing a lot of work or communication. Such 'hot' segments can result either from uneven access to the array by the computational threads, or from having data that takes a long time to process in owner-computes phases. It also allows communication optimizations, as described in Section 5.

The drawback of this scheme is high latency for non-local reads and phase changes. The runtime can compensate by overlapping the execution of other local threads with blocking MSA operations. This process is facilitated by overdecomposition, so that on each PE there are many threads using the MSA. When the active thread blocks, either due to a MSA cache miss, phase change, or a non-MSA operation, another thread is scheduled.

3.4 Tools

One common barrier to entry for new programming models is the lack of good tools for debugging and performance analysis. Particularly for specialized models such as MSA, the cost of creating these tools is unjustifiably high, at least while the model is not widely used. MSA avoids this problem by relying upon tools targeting Charm++. The Projections [29] visualization tool allows the user to view all MSA-related sends and receives and identify performance bottlenecks and outlier processes. CharmDebug [22] provides breakpoints, message inspection, and memory tagging for both shared memory and cluster environments. MSA derives significant utility from the fact that any efforts un-

dertaken to provide tools for its underlying runtime system translate directly into benefits for MSA programs.

4. EXAMPLE

To demonstrate the use of MSA, we present the kernel of a classical (Newtonian) molecular dynamics simulation using Plimpton's algorithm [38]. The code for this example can be seen in Listing 1. This is a force-based decomposition, meaning that each thread is responsible for a subset of the pairwise interactions between particles. Each thread's portion of the interactions is specified by variables `i_start`, `i_end`, `j_start`, and `j_end`. Once the interaction forces have been computed, the particles' velocities and positions are updated by integrating the force on each particle over a short timestep.

The two separate phases of computation described above, force summation and integration, correspond to distinct usages for the shared arrays holding the data. In the force summation phase (lines 6–15), particle coordinates are read and forces are accumulated. The application code accesses these arrays as if they were local, and the runtime works to limit communication overhead. For the coordinates, caching ensures that access is mostly local. For the forces, the buffered reduction offered by accumulation mode minimizes communication needs.

The integration phase (lines 17–19 and 26–40) operates on the particle coordinates in owner-computes mode. The code instructs each page of the coordinates MSA to run `Integrator::operator()` over its contents. We pass read handles for the `atominfo` and `forces` arrays to the constructor for this function object so that it can access all necessary data. The main threads of execution are then free to continue doing other work until they next synchronize on the coordinates array. In essence, this forks off a task for each page of the array, and joins all of them at the next synchronization.

5. OPTIMIZATION

The design of MSA offers opportunities for optimization both at the application level and the runtime level. This allows the programmer to tune MSA to handle the memory access characteristics of a particular algorithm through explicit means such as manually prefetching remote data and adjusting the granularity of remote data access while still benefiting from active optimization at runtime. In addition, there are opportunities for the compiler to improve performance by identifying array accesses that can be proven not to access remote data and transparently eliminating the checking that would normally be needed for such accesses. In this section we present possible compile-time and run-time optimizations for MSA, some of which are still speculative or under development.

5.1 Programmer- and Compiler-driven Optimizations

In a distributed shared memory system, the utmost care must be taken to ensure that the code executed at runtime doesn't naïvely incur the expense of accessing remote memory when it's not necessary. Alongside this overriding concern, we wish to avoid the overheads imposed by unneeded runtime checks and poor communication patterns. In other PGAS languages [15, 43, 8, 37, 9], these responsibilities are shared among the programmer, the compiler, and the runtime implementation, and MSA is no different.

5.1.1 Prefetching

At the application level, the programmer has the ability to explicitly prefetch remote data to ensure that subsequent reads will access only locally available data. Prefetching prevents long latency remote reads in read-only mode, potentially leading to a dramatic increase in performance.

Consider, as an example, a simple matrix multiplication where data from two MSAs in read-only mode is used to calculate contributions to a third MSA in accumulate mode. All remote accumulate operations are locally buffered until the array's phase changes, so the threads accessing these MSAs will block only when they attempt to read non-local data. Even if the runtime is able to compensate for all blocking reads by scheduling other work, performance can suffer. When the blocked thread's data arrives and it is rescheduled, any data it had in the hardware cache has likely been evicted during the execution of other threads. Given the reliance of many optimized codes on a high cache hit rate, the performance penalty of remote reads may be high even if the resulting communication is fully overlapped with computation.

5.1.2 Efficient Local Operations

Effective prefetching leads to a further optimization: the elimination of the check for remote data accesses in reads known to be safe. Conservatively, every array read can be assumed unsafe and explicitly checked to see if a blocking non-local access is necessary. However, in code that has been written to avoid these reads, these checks produce an unwanted performance penalty, particularly when the reads in question occur in the body of tight loops. Although an unsafe read operation could be exposed to the programmer, this approach is inelegant and produces an additional failure mode in MSA programs. Compiler analysis to identify safe reads and eliminate their associated runtime checks can provide the same benefits without the drawback of a more complex and failure-prone API in many cases. Steps toward this optimization have been taken in other work [11], but current support is incomplete.

5.1.3 Tunable Cache Parameters

The fact that MSA's local cache is software controlled presents another avenue to optimization. Both the size of the local cache and the granularity of data exchange for the MSA (effectively the cache line size) can be set programmatically on a per-MSA basis. This allows the programmer to select sizes appropriate to the memory access pattern of the MSA in question. In contrast, the granularity of many distributed shared memory systems is determined by the operating system memory page size. The decoupling of these variables from the operating system and from one MSA to another provides another opportunity to tune MSA performance.

5.2 Internal Optimizations

Independent of any changes to program organization made by the programmer or compiler, there are some techniques

```
1  void PlimptonMD(CoordMSA::Handle &hCoords,
2                  XyzMSA::Handle &hForces,
3                  AtomInfoMSA::Read &rAtominfo)
4  {
5      for (int timestep = 0; timestep < NUM_TIMESTEPS; timestep++) {
6          // Force calculation for a section of the interaction matrix
7          XyzMSA::Accum aForces = forces.syncToAccum(hForces);
8          CoordMSA::Read rCoords = coords.syncToRead(hCoords);
9          for (int i = i_start; i < i_end; ++i)
10             for (int j = j_start; j < j_end; ++j) {
11                 XYZ force = calculateForce(rCoords(i), rAtominfo(i),
12                                            rCoords(j), rAtominfo(j));
13                 aForces(i) += force;
14                 aForces(j) += force.negate();
15             }
16
17         // Movement Integration for our subset of atoms
18         XyzMSA::Read rForces = forces.syncToRead(aForces);
19         CoordMSA::Owner oCoords = coords.syncToOwner(rCoords, Integrator(rAtominfo, rForces));
20
21         hCoords = oCoords;
22         hForces = rForces;
23     }
24 }
25
26 class Integrator
27 {
28     const AtominfoMSA::Read &rAtominfo;
29     const XyzMSA::Read &rForces;
30
31 public:
32     Integrator(const AtominfoMSA::Read &rAtominfo_, const XyzMSA::Read &rForces_)
33         : rAtominfo(rAtominfo_), rForces(rForces_) { }
34
35     operator()(CoordsMSA::page_ref page)
36     {
37         for (int k = page.begin(); k != page.end(); ++k)
38             integrate(rAtominfo(k), rForces(k), page(k));
39     }
40 };
```

Listing 1: MSA-style kernel of Plimpton's algorithm for classical molecular dynamics simulation. For simplicity, we show only the function executed by each thread, omitting array declarations and thread creation. Full examples, as well as our MSA implementation, are available as part of the Charm distribution at `charm.cs.uiuc.edu`

that the MSA implementation can apply to improve performance. Exactly when and how these are applied can be controlled either through configuration made by users at runtime, or by decision procedures within the adaptive runtime system [13].

5.2.1 Pushing

In applications that exhibit locality of reference to particular arrays, we can predict which array elements each thread is likely to access in each phase. Given this foreknowledge, the implementation can *push* data to caches on PEs with threads that are expected to access it. This pushing can happen as soon after each synchronization as each page can be sure that the data to be pushed is correct. At worst, this occurs when the synchronization is complete (if some elements are not assigned during a write phase), but can occur sooner if writes to all elements in the range to be pushed have already reached the page. This optimization is currently under development in MSA.

5.2.2 Variable Data Movement Granularity

Logically, it makes sense to think about the chunks of the array that are sent from home pages to remote PEs in terms of cache lines, as in coherent multi-processor systems. However, unlike in hardware implementations, there is no reason to force these units to a fixed size. Depending on information about what data is likely to be accessed, the unit of transfer can vary. In an application which exhibits no locality, such as an unstructured mesh with indirection from one array to another, data can be moved one element at a time. On the other hand, if persistent locality of reference allows the runtime to predict that in a given phase, some thread will access a particular chunk of a page, then the page can send exactly that chunk to the thread's host PE before or as its first element is requested. This knowledge can be acquired either through runtime tracking or from prefetch calls inserted by the programmer or compiler.

Taking this idea further, a requested or pushed chunk of data might be large enough that its transfer would last longer than other available work on the destination PE. This would lead to idle time until the transfer completes and the requesting thread unblocks. Thus, it might be preferable to transmit smaller chunks so that computation can proceed while more data is transferred. This is similar in spirit to cache prefetch instructions in modern micro-architectures, though at a much different scale.

5.2.3 Reductions

As noted in the description of the MSA programming model, the accumulate mode effects a reduction into each element of the shared array. The write mode can also be seen as a reduction, with assignments to different elements of each page merged as they move to the page's home PE. Hence, we can take advantage of other efforts toward efficient cross-node reduction operations.

5.2.4 Object Mapping

One benefit of programming above an adaptive runtime system is access to the optimization techniques that it provides. In the Charm++ model, objects are decoupled from the processors on which they execute [24]. This means that the runtime system is free to map both threads of program execution and homes for array pages as it sees fit across the machine. Two prime drivers of mapping decision are load balance [25] and communication minimization [28, 4].

6. 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 [30] and its successor Cluster OpenMP [39] 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 [2, 7] 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 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 [20].

Global Arrays [36] (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.

7. CONCLUSIONS AND FUTURE WORK

Asynchrony in execution is important to attaining high performance for a variety of reasons: the need for overlap of communication and computation, the increasingly dynamic structure of scientific codes, and the need to efficiently assemble a multitude of interacting parallel components. A long line of research has shown the programmability benefits of easy access to non-local memory. In this paper, we have described the addition of Multiphase Shared Arrays, a PGAS array library, to the established Charm++ asynchronous programming environment.

MSA's programming model offers some advantages over other PGAS implementations. By restricting application code to simple, well-specified access modes in each phase between synchronization points, we eliminate the possibility of race conditions. We demonstrate enforcement of these access modes at compile time using pure C++ code. Additional checks are implemented in the runtime, but could be elided by a more specialized compiler or translator.

Like other PGAS languages, we do not ask the programmer to explicitly transfer data from remote PEs to local buffers. Where they use compile-time optimization to attain performance, we primarily rely on adaptive software-managed caching.

Since MSA is not general enough to express the bulk of parallel programs, we rely on composition with other programming models. Our approach allows easy integration with code written in different styles, all implemented on top of a common runtime system.

We have described a variety of optimization opportunities presented by MSA. Some of these are common to implementations of PGAS and asynchronous message-driven execution. In the context of distributed shared arrays, we consider the benefits of decoupling decomposition from distribution and pushing to be distinct contributions.

Throughout this paper, we have identified a variety of possible directions for future work on MSA. One area that particularly needs to be addressed is a detailed performance study of the effects of overdecomposition, data distribution, and object mapping on real MSA applications. The performance effects of prefetching and other optimizations likewise need to be quantified. We also plan to implement the more speculative optimizations discussed in this paper, such as pushing, and observe their effectiveness. MSA currently lacks optimizations specifically targeted at multicore scenarios; this is another promising direction for optimizations. Finally, we would like to explore the possibility of access modes other than the ones presented here, while increasing the number of access mode errors we can detect at compile-time.

8. ACKNOWLEDGEMENTS

We would like to thank Abhinav Bhatelé, Eric Bohm, Isaac Dooley, and Lukasz Wesolowski for their helpful comments on this paper. Earlier implementations of some facets of MSA were 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.

9. REFERENCES

[1] R. Barriuso and A. Knies. Shmem user's guide for c. *Cray Research Inc*, Jan 1994.

[2] 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)*, pages 168–177, 1990.

[3] A. Bhatelé and L. V. Kalé. Benefits of Topology Aware Mapping for Mesh Interconnects. *Parallel Processing Letters (Special issue on Large-Scale Parallel Processing)*, 18(4):549–566, 2008.

[4] A. Bhatele and L. V. Kale. An Evaluation of the Effect of Interconnect Topologies on Message Latencies in Large Supercomputers. In *Proceedings of Workshop on Large-Scale Parallel Processing (IPDPS '09)*, May 2009.

[5] 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.

[6] 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*, 52(1/2):159–174, 2008.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communications in distributed shared memory systems. *ACM Transactions on Computers*, 13(3):205–243, Aug. 1995.

[8] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing*, Jan 2007.

[9] 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*, pages 519–538, New York, NY, USA, 2005. ACM.

[10] 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.

[11] 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.

[12] 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.

[13] I. Dooley and L. V. Kale. Control points for adaptive parallel performance tuning. November 2008.

[14] 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.

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

[16] M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. *Acoustics*, Jan 1998.

[17] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. *SIGPLAN Not.*, 25(10):169–180, 1990.

[18] 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.

[19] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, pages 306–322, College Station, Texas, October 2003.

[20] 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.

[21] 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.

[22] R. Jyothi, O. S. Lawlor, and L. V. Kale. Debugging support for Charm++. In *PADTAD Workshop for IPDPS 2004*, page 294. IEEE Press, 2004.

[23] L. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 17–25, Aug. 1990.

[24] L. V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[25] L. V. Kale, M. Bhandarkar, and R. Brunner. Run-time Support for Adaptive Load Balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, volume 1800, pages 1152–1159, March 2000.

[26] L. V. Kale and A. Gursoy. Modularity, reuse and efficiency with message-driven libraries. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 738–743, San Francisco, California, USA, February 1995.

[27] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[28] L. V. Kale, S. Kumar, and K. Vardarajan. A Framework for Collective Personalized Communication. In *Proceedings of IPDPS'03*, Nice, France, April 2003.

[29] L. V. Kale, G. Zheng, C. W. Lee, and S. Kumar. Scaling applications to massively parallel machines using projections performance analysis tool. In *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, volume 22, pages 347–358, February 2006.

[30] 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.

[31] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[32] D. Kunzman, G. Zheng, E. Bohm, and L. V. Kalé. Charm++, Offload API, and the Cell Processor. In *Proceedings of the Workshop on Programming Models for Ubiquitous Parallelism*, Seattle, WA, USA, September 2006.

[33] O. Lawlor, S. Chakravorty, T. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. Kale. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22(3-4):215–235.

[34] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for FORTRAN Usage. *ACM Transactions on Mathematical Software*, 5:308–323, 1979.

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

[36] 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.

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

[38] S. J. Plimpton and B. A. Hendrickson. A new parallel method for molecular-dynamics simulation of macromolecular systems. *J Comp Chem*, 17:326–337, 1996.

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

[40] 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.

[41] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*. 1990.

[42] L. Wesolowski. An application programming interface for general purpose graphics processing units in an asynchronous runtime system. Master's thesis, Dept. of Computer Science, University of Illinois, 2008. http://charm.cs.uiuc.edu/papers/LukaszMSThesis08.shtml.

[43] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13), September – November 1998.