Using Shared Arrays in Message-Driven Parallel Programs

Phil Miller[1], Aaron Becker[2], Laxmikant Kalé

Department of Computer Science, University of Illinois, Urbana, Illinois 61801, USA

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 Multi-phase 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.

Keywords: programming models, composition, distributed shared arrays

1. 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. In Charm++-based [1] 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 only requires adaptating interfaces, rather than the deeper structural changes or non-overlapped (and hence non-performant) time or processor division that

Email addresses: `mille121@illinois.edu` (Phil Miller), `abecker3@illinois.edu` (Aaron Becker), `kale@illinois.edu` (Laxmikant Kalé)

[1]Corresponding author. Supported by NSF grant OCI-0725070

[2]Supported by NFS grant ITR-HECURA-0833188 and the UIUC/NCSA Institute for Advanced Computing Applications and Technologies

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.

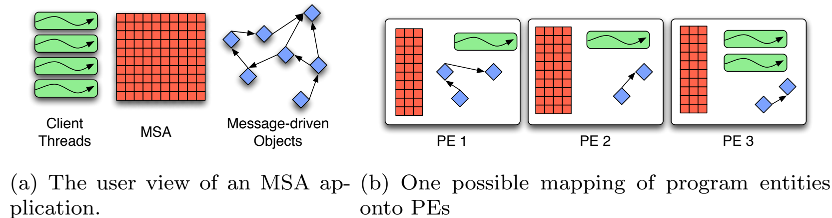
However, the message-driven model is not an ideal choice for all parallel applications. In cases where shared data is essential to concise expression of the algorithm, the code needed to explicitly communicate this shared data in a message-driven style can dominate the structure of the program, and overwhelm the programmer. In this situation, a shared address space programming model, as exemplified by the Global Arrays library [2] and PGAS languages [3, 4, 5] can be advantageous. Applications which require data structures too large to fit in memory local to one processor may also become much 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 optimized by a compiler, as demonstrated by Co-Array Fortran [6] and Jade [7]. Programs which use explicit messaging can benefit 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 4). 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 5). We also describe a system of *typed handle objects* (Section 3) 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.

2. Multiphase Shared Arrays

Multiphase Shared Arrays (MSA) [8] 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 by the runtime system or by the programmer. 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. 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. The initial development of MSA was based on the observation that applications that



(a) The user view of an MSA application. (b) One possible mapping of program entities onto PEs

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

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 limiting the array to only be accessed using one one of several specifically-defined *access modes* (described below) during each phase. By establishing this discipline, MSA usage is inherently deterministic. However, in exchange for this guarantee, the programmer gives up some of the freedom of a completely general-purpose programming model.

2.1. Data Decomposition and Distribution

MSA decomposes arrays not into fixed chunks per PE, but rather into *pages* of a common shape. Developers can vary the shape of the pages to suit applications' needs. For example, a 10×10 array could be broken into ten 10×1-shaped pages, or four 5×5 pages, etc. 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* 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 page objects are managed by the Charm++ runtime system. 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 opportunities to tune MSA code for both application and system characteristics. The page objects are initially distributed according to a mapping function, either specified by application code or following the defaults in Charm++. As the program executes, the runtime may redistribute the pages by migrating them to different PEs in order to account for load imbalance, communication locality, system faults, or other concerns. The user view of an MSA program and corresponding mapping by the runtime system are illustrated in figure 1.

2.2. Caching

The runtime library caches data accessed from MSAs. This approach differs from Global Arrays [2], where the user must either allocate and manage buffers for bulk remote array segments or incur remote communication costs for each 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 accesses data that is not cached locally, the cache requests it from its home page, then suspends the requesting thread. At this point, messages queued for other threads are delivered. The cache manager receives the home page's response and unblocks the requesting thread. Previous work with MSA [7] has shown that the overhead of caching and associated checks is reasonable, and well-tuned application code can match the performance of equivalent sequential code.

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 the Munin system does [9]. However, the MSA access modes are designed to make cache coherence simple and inexpensive. Accesses never require remote cache invalidations or immediate writeback.

2.3. Access Modes and Safety

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

Read-Only Mode: As its name suggests, read-only mode makes the array immutable, permitting reads of any element but writes to none. Remote cache lines can simply be mirrored locally, and discarded at the next synchronization point. In this mode, there are no writes to produce race conditions.

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. Static analysis could allow us to check this condition at compile time for some access patterns and elide the runtime checks when possible.

Accumulate Mode: This mode effects a reduction into each element of the array, with each thread potentially making zero, one, or many contributions to any particular element. While it is most natural to think of accumulation in terms of operations like addition or multiplication, any associative, commutative binary operator can be used in this fashion. One example, used for mesh repartitioning in the ParFUM framework [10], uses set union as the accumulation function. The operator's properties guarantee that the order in which it's applied does not introduce non-deterministic results.

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

```
A.syncToWrite();

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

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

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

2.4. 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. In section 4, we describe how to relax the need to explicitly wait by delivering messages when synchronization is complete.

3. 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 3.1. We currently rely on run-time checks to detect error conditions, such as threads synchronizing into different modes, and intersecting write sets during write-once mode. Converting ParFUM's [10] mesh repartitioning code to use typed handles exposed previously undetected bugs that we subsequently fixed.

There are alternatives to our typed handle scheme, but they all suffer from either increased complexity or the need for non-standard tools. With a more capable type system in C++, we could define the array with a linear type [11] such that synchronization would change the array's type in the same way that handle types are currently changed. If we wished to construct more complex constellations of allowed operations, an approach of policy templates and static assertions [12] would serve. Such policy templates would have a boolean argument for each operation or group of operations that is controlled.

We could also enforce MSA's high-level semantic conditions in *contracts* [13] 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 like Eiffel [14] that natively supports contracts. Second, these conditions would 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 context-dependent.

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.

3.1. Example: Parallel k-Means Clustering

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

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

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

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

4. Composing Shared-Array and Message-Driven Modules

Prevalent parallel software needing distributed arrays combines MPI with GA, and implementations of those libraries are designed to interoperate. Multiphase Shared Arrays have also previously been used in concert with Adaptive MPI [15] on top of Charm++'s message-driven runtime system. The environment provided by those packages is well suited to interactions within a single internally coordinated, easily synchronized parallel module. However, application development is trending toward the combination of multiple, interacting parallel modules. In that context, issues of data synchronization, transfer of

```
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    CLUSTERS::Read r = w.syncToRead(); // Put the array in Read mode
8
9    do {     // Each PE finds the seed closest to itself
10     double minDistance = distance(r, curSeed);
11
12     for (int i = 0; i < numClusters; ++i) {
13       double d = distance(r, i);
14       if(d < minDistance) {
15         minDistance = d;
16         newSeed = i;
17       }
18     }
19
20     // Put the array in Accumulate mode,
21     // excluding the current value
22     CLUSTERS::Accum a = r.syncToExcAccum();
23     for (int i = 0; i < numMetrics; ++i)
24       a(newSeed, i) += metrics[i]; // Each PE adds itself to its new seed
25
26     a(newSeed, numMetrics) += 1; // Update membership and change count
27     if (curSeed != newSeed)
28       a(0, numMetrics+1) += 1;
29     curSeed = newSeed;
30
31     r = a.syncToRead(); // Put the array in Read mode
32   } while(r(0, numMetrics+1) > 0);
33 }
```

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).

control, and performance tuning become substantially more challenging with a purely SPMD-oriented software stack.

The message-driven execution model addresses the concerns of transfer of control and coupled performance tuning directly, by fully interleaving all parallel computation and communication. In concert with a shared-array model, however, the same data synchronization concerns arise, and are potentially amplified by the nondeterminism in control flow. As described in section 2, MSA addresses the general data synchronization issues, but does not speak to the

coordination or synchronization of control flow around it. We address precisely those issues in this section.

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 an 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 [16]. With respect to the shared array, this is essentially the same SPMD arrangement as seen before. Extending this style to multiple object collections accessing one or more arrays through an identical sequence or cycle of phases is similarly straightforward.

As we introduce multiple modules that want to share an array, the complexity of the existing approach increases. Each module module must respect the phase and synchronization behavior of all 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).

We avoid this complexity by sending messages containing array handles to client objects that react by spawning a thread to perform one phase 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 separate 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. Where these steps don't depend on persistent state outside the array, we can separate each phase into its own col-

lection of objects, allowing us to vary the 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 computational methods and computer scientists focusing on efficient parallel execution. This approach also produces better-engineered software in which distinct computational steps are distinguished and named as loosely coupled components, each of which could then be tested in isolation.

5. 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) [17]. 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.

Our prototype is built on the popular MD application NAMD [18]. For its short-range interactions, NAMD uses a hybrid spatial-force decomposition. In this decomposition, the particles are divided among a collection of `Patch` objects, each responsible for the particles within a portion of the simulation box. The forces among the particles are computed by several collections of `Compute` objects that receive particle positions from one or two `Patch` objects, calculate the interactions among the particles, and transmit the resulting forces back to the `Patch`es (Figure 2). The `Patch`es are responsible for integrating the net forces on the particles to obtain their velocity and position for the next time step. These portions of the code are purely message-driven Charm++ code, and remain unchanged in our modified version.

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 entire process is illustrated in figure 3. The restriction, prolongation, and cutoff steps are each performed by separate collections of chares at each level. Each level's grids are stored and transfered by MSA.

This process exposes parallelism at two distinct levels: each of the computational steps at each level (direct interaction, restriction, and prolongation) can begin and run independently as soon as its input data is available, and the work of each step can be divided among a number of objects. We have benchmarked this early implementation on Blue Gene P using the 92224 atom ApoA1 molec-

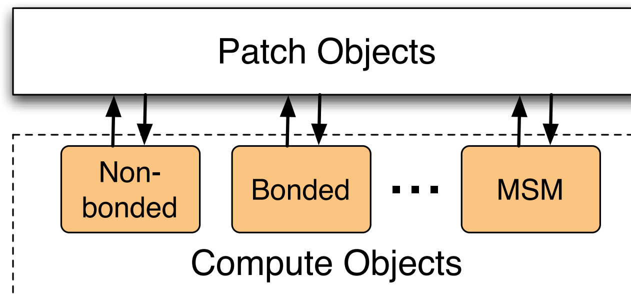


Figure 2: The multi-level summation methed is one of many types of `Compute` objects used in NAMD.

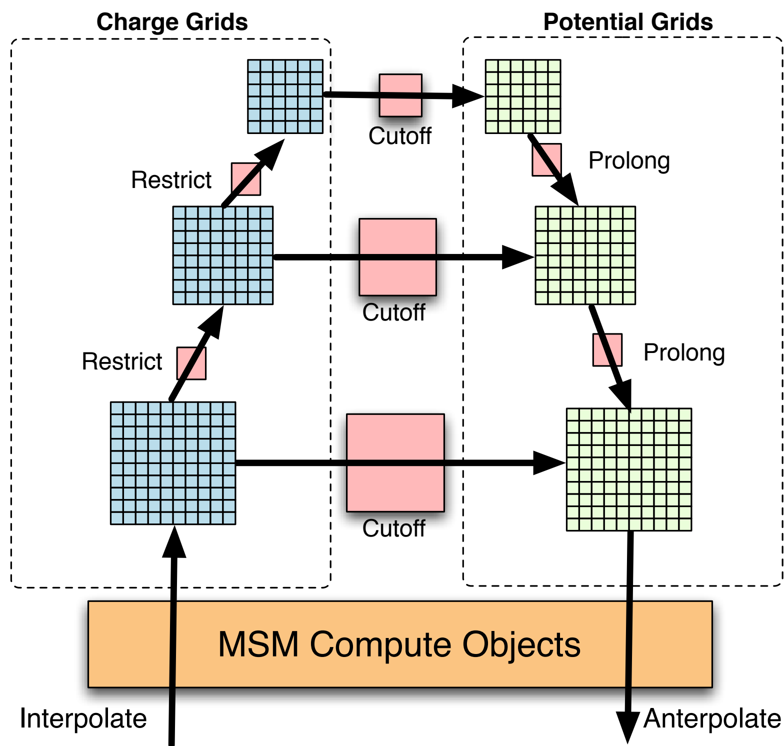


Figure 3: Multi-level summation as implemented using collections of message-driven objects (`MSMCompute, Cutoff, Restrict, Prolong`) accessing MSAs.

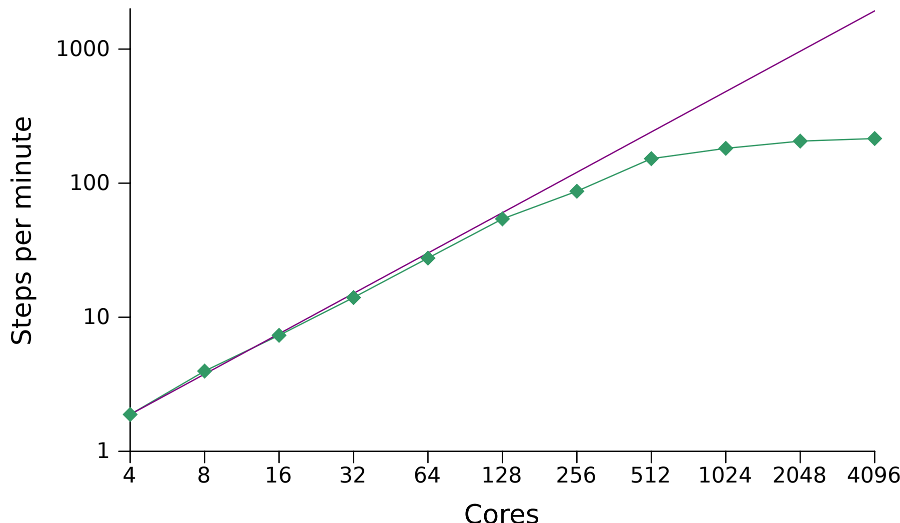


Figure 4: Scaling results for the multi-level summation method implemented in NAMD using MSA on Blue Gene P. Step time on 4 processors (1 node) is 31.9 seconds.

ular system, and scaling results are shown in Figure 4. At present, the arrays are distributed in the Charm++ runtime's default linear blocked manner.

Some extracts from our prototype code appear in listing 2. Lines 1–15 show the Charm++ coordination code (in Structured Dagger notation [16]) for the work each `MSMCompute` does 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. When the potential grid is ready to have forces anterpolated back to the particles, as indicated by receiving a read-mode handle (line 8), 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 16–25. This entry method receives read-mode handles to both finest-resolution grids (16), 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.

6. Related Work

MSA draws some ideas from earlier implementations of distributed shared memory. Munin [9, 19] lets the programmer statically specify that each shared

```
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    when readPotentials(Read potentials) {
9      forceMsg = new ParticleForceMsg;
10     computeForces(potentials, forceMsg);
11     patch.receiveForces(forceMsg);
12     potentials.syncDone();
13     delete particles;
14   }
15 }
16 entry void Energy::calculate(Read charges, Read potentials) {
17   double u = 0.0;
18   for(int i = 0; i < grid_x[0]; ++i)
19     for(int j = 0; j < grid_y[0]; ++j)
20       for(int k = 0; k < grid_z[0]; ++k)
21         u += charges(i,j,k) * potentials(i,j,k);
22   contribute(u); // Contribute to a reduction
23   charges.syncDone();
24   potentials.syncDone();
25 }
```

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.

variable should be cached in one of a few different ways, depending on their expected access pattern. These cache modes include *read-mostly, write-once, result, producer-consumer*, and *migratory.* It also provides an unconstrained generic coherence protocol for variables with no specified treatment. Treadmarks [20] and the subsequent Cluster OpenMP [21] try to offer developers a model very similar to physical shared memory in programmability and performance on distributed systems. Unlike MSA, 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 [22].

X10 [5] is a partitioned global address space (PGAS) language with strong support for asynchronous operations and flexible synchronization. Its *clock* synchronization construct, and the subsequent proposal of *phasers* [23], allow dynamically varying sets of tasks to coordinate their activities. Unlike MSA's access mode discipline, X10 does not directly associate its synchronization mechanisms with the state of shared data structures, and thus requires the programmer to ensure correct usage. UPC has also seen proposals to add more asynchronous mechanisms [24].

7. Future Work

In many cases, particular computational elements will access the same parts of a shared array repeatedly, such as from one iteration of an algorithm to the next. This persistence of reference means that migrating data-containing shared array objects and computation objects closer together should reduce communication latency and network contention. In the ideal case, most 'shared' array accesses can actually take place within one processor or node's own memory, turning system-level interconnect traffic to memory bus traffic. When access patterns are static and predictable based on knowledge of the algorithms or problem domain in question, such a mapping can be constructed by a sufficiently knowledgable and dedicated programmer. However, the patterns may be both unpredictable and dynamically varying. Even when the patterns are simply dependent on the input problem or data, optimal performance would still require a carefully crafted mapping for each instance. Based on runtime instrumentation, we can apply a graph partitioning library like METIS [25] or Scotch [26] to approximately equalize load and minimize communication.

Another avenue for improvement would be to make the MSA implementation adapt its behavior at runtime. Within a phase, accesses to nearby locations can be prefetched on the same basis as a hardware memory prefetch system. Across phases, when the runtime identifies that an object accesses the same part or parts of a shared array in successive phases, the cache on the object's host processor can exploit this observation to proactively request that section's contents as soon as the array enters a read phase. These optimizations aim to reduce latency, at the potential cost of additional network traffic.

8. 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 by sending messages containing appropriate shared array handles to clients involved in each phase. Using that structure, we modified MSA's synchronization semantics such that client threads not participating in an entire series of phases need not block while waiting for intermediate synchronization steps 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.

References

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

[2] J. Nieplocha, R. J. Harrison, R. J. Littlefield, Global arrays: A nonuniform memory access programming model for high-performance computers, J. Supercomputing (1996) 197–220.

[3] T. El-Ghazawi, W. Carlson, T. Sterling, K. Yelick, UPC: Distributed shared memory programming, books.google.com (2005).

[4] R. Numrich, J. Reid, Co-array fortran for parallel programming, ACM SIGPLAN Fortran Forum 17 (1998).

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, X10: an object-oriented approach to non-uniform cluster computing, in: OOPSLA 2005: ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications.

[6] Y. Dotsenko, C. Coarfa, 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.

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

[8] J. DeSouza, 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.

[9] J. Bennett, J. Carter, W. Zwaenepoel, Munin: distributed shared memory based on type-specific memory coherence, PPOPP '90: ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (1990).

[10] O. Lawlor, S. Chakravorty, T. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, L. Kale, Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications, Engineering with Computers 22 (2006) 215–235.

[11] P. Wadler, Linear types can change the world!, in: M. Broy, C. Jones (Eds.), Programming Concepts and Methods, 1990.

[12] J. Maddock, S. Cleary, Boost.StaticAssert, Boost Library Project, 2005.

[13] R. Helm, I. M. Holland, D. Gangopadhyay, Contracts: specifying behavioral compositions in object-oriented systems, SIGPLAN Not. 25 (1990).

[14] B. Meyer, Applying "Design by Contract", IEEE Computer 25 (1992).

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

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

[17] D. J. Hardy, Multilevel Summation for the Fast Evaluation of Forces for the Simulation of Biomolecules, Technical Report UIUCDCS-R-2006-2546, Dept. of Computer Science, Univ. Illinois at Urbana-Champaign, 2006.

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

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

[20] P. Keleher, S. Dwarkadas, A. L. Cox, W. Zwaenepoel, Treadmarks: Distributed shared memory on standard workstations and operating systems, in: Proc. of the Winter 1994 USENIX Conference, pp. 115–131.

[21] C. Terboven, D. Mey, D. Schmidl, M. Wagner, First experiences with intel cluster openmp, Lecture notes in computer science (2008).

[22] L. Huang, B. Chapman, Z. Liu, Towards a more efficient implementation of openmp for clusters via translation to global arrays, J. Par. Co. (2005).

[23] J. Shirako, D. Peixotto, V. Sarkar, W. Scherer, Phasers: a unified deadlock-free construct for collective and point-to-point synchronization, International Conference on Supercomputing (2008).

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

[25] George Karypis and Vipin Kumar, A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm, in: Proc. of the 8th SIAM conference on Parallel Processing for Scientific Computing.

[26] C. Chevalier, F. Pellegrini, PT-Scotch: A tool for efficient parallel graph ordering, J. Parallel Computing (2008) 318–331.