MSA: Multiphase Specifically Shared Arrays

Jayant DeSouza and Laxmikant V. Kalé

University of Illinois, Urbana IL 61801, USA
`jdesouza@uiuc.edu`,`kale@cs.uiuc.edu`

Abstract. Shared address space (SAS) parallel programming models have faced difficulty scaling to large number of processors. Further, although in some cases SAS programs are easier to develop, in other cases they face difficulties due to a large number of race conditions. We contend that a multi-paradigm programming model comprising a distributed-memory model with a disciplined form of shared-memory programming may constitute a "complete" and powerful parallel programming system. Optimized coherence mechanisms based on the specific access pattern of a shared variable show significant performance benefits over general DSM coherence protocols. We present MSA, a system that supports such specifically shared arrays that can be shared in `read-only`, `write-many`, and `accumulate` modes. These simple modes scale well and are general enough to capture the majority of shared memory access patterns. MSA does not support a general `read-write` access mode, but a single array can be shared in `read-only` mode in one phase and `write-many` in another. MSA coexists with the message-passing paradigm (MPI) and the processor virtualization-based message-driven paradigm(Charm++). We present the model, its implementation, programming examples and preliminary performance results. [1]

1 Introduction

Parallel programming remains a complex task, even though parallel machines and their use in applications has spread widely, especially with deployment of thousands of clusters. The predominant programming paradigm used is message-passing among independent processes (each with its own address space), as embodied in MPI. It is often argued that shared address space (SAS) is an easier method of programming. Although quantitative empirical support for such a statement is lacking, their probably is an intuitive basis for this belief among a section of researchers.

Our experience with a number of parallel applications over the years indicates that there are distinct programming situations where SAS is an easier programming model whereas there are equally distinct situations where it is not. E.g. when there are data races, shared memory paradigm, which allows for a much

[1] This work was supported in part by the National Institute of Health under Grant NIH PHS 5 P41 RR05969-04 and the local Department of Energy ASCI center under Grant DOE B341494

larger number of distinguishable interleavings of executions of multiple threads, tends to be a difficult paradigm. In contrast, in a computation such as matrix multiply, where the data in input matrices is only read, and data in the output matrix is only written, is relatively easier to express in SAS. Further, since SAS views all data as uniformly accessible, it does not lend itself to locality-conscious programming, which is needed for efficiency.

We suggest that the problems with SAS are due to trying to do everything (i.e. all kinds of information exchange between processes) with SAS. It may be more productive to incorporate SAS as a *part* of a programming model that also allows private data for each thread, and mechanisms for synchronization and data-exchange such as message-passing or method-invocations. This frees us to support only those SAS access modes that can be efficiently and scalably supported on distributed memory machines including clusters, without being encumbered by having to support a "complete" programming model.

Which access modes can be efficiently supported? `read-only` accesses, `write-many` accesses (where each location is updated by at most one thread), and `accumulate` accesses (where multiple threads may update a single location, but only via a well-defined commutative associative operation) seem to be the obvious candidates. The idea of distinguishing between access patterns was studied in Munin[2] and also in Chare Kernel[8, 20] (the precursor to Charm[13, 14]) and is used in TreadMarks/NOW[15]. The generalized notion of accumulate accesses (see Section 1) is relatively new, although compiler research (e.g. Polaris[4]) has often focused on identifying commutative-associative operations.

Another observation stemming from our application studies is that the access pattern for the same data changes distinctly between computation phases. For example, a matrix multiply operation (C = AxB) may calculate a C matrix in Phase I of the computation (where A and B matrices are accessed in read-only manner, and C is written-only or accumulated), whereas in the phase II, C matrix may be used in a read-only manner while A and B may be updated. These phases may then iterate.

This suggests the idea of *multi-phase shared arrays*. For each array, the programmer specifies its access mode, and may change it between phases. The phases may be separated by array-specific synchronizations such as barrier (as in release consistency).

The restricted set of operations simplifies the consistency protocol and traffic associated with that: no invalidations are needed, all writes can be buffered, etc. For all other operations not covered, one is free to use other mechanisms such as message passing.

One of the original motivations for this work was computations performed at initialization, where efficiency is less important, and coding productivity is the dominant consideration. However, it quickly became clear that the method is useful more broadly beyond initialization. Of course, such broad use requires more serious consideration of efficiency issues. With "prefetch" calls (See Section 2) , we provide efficiency comparable to that of local array accesses. Further, one of the costs of DSM systems is the long latency on "misses". Processor virtualization techniques that we have been exploring in Charm++ and Adaptive MPI (AMPI)[9] allow many user-level (lightweight) threads per processor, which help tolerate such latencies.

The MSA abstraction has been implemented as a library in Charm++ and AMPI, and as a language-level feature in a compiler-supported parallel language called Jade [7]. Compiler support simplifies syntax and automates optimizations which have to be done manually by MSA users (such as inserting prefetches).

2 MSA Description

Conceptually, an MSA is an array of data elements that can be globally accessed in an MIMD program in one of several dynamically chosen global access modes and in units of a user-defined page size. The modes are not expected to be fixed for the lifetime of an array, but for a phase of execution of the program. MSA's are implemented as a templated, object-oriented C++ class. Currently 1D and 2D MSA arrays are supported.

The elements of an MSA can be one of the standard C++ built-in types, or a user-defined class with certain operations. The number of elements in the MSA is specified in the constructor when the MSA is created. Currently, an MSA cannot be resized once created.

For complicated element types (such as linked lists), a `pup()` method must be defined by the user for the element type: this is used to pack and unpack the element when pages are shipped around. This allows each element to be a linked list or other variable sized data structure. (More details of the *PUP* framework can be found in [12].)

Internally, the MSA array is split up into "pages" of a user-defined size. The page size is user specified for each MSA at the time the MSA is created. The page size is specified as a number of elements and can range from one element to the total number of elements in the array. For 2D MSA arrays, the data layout can also be specified at creation time. Currently, row-major and column-major data layouts are supported, and we plan to support block partitioned layout.

The array of "pages" is implemented as a Charm++ `ChareArray` object. `ChareArray` objects are managed by the Charm++ runtime system (RTS); they can be migrated across processors under the control of the RTS and can participate in system-wide load-balancing based on communication patterns, computational load, etc.[14]

The MSA runtime system on each processor fetches and replicates remote pages into local memory on demand, or instantiates blank local copies of remote pages when needed, based on the mode (described below) of the MSA. The amount of local memory so used (the *local cache*) on each processor can be constrained by the user for each MSA. When the local cache fills up, pages are flushed out using a user-defined page replacement policy. A default policy is provided that keeps track of a few most recently used pages and flushes out any page not listed there.

The MSA is globally accessed in one of several *access modes*, which can be changed dynamically over the life of the program. The mode of an MSA is set implicitly by the first operation (read, write, or accumulate) on it after a `sync` or `enroll`. Each phase of execution in a particular mode is terminated by running a `sync` operation on the array.

The modes supported are `read-only`, `write-many` and `accumulate`, and have been chosen for simplicity and efficient implementation.

In the `read-only` mode, the elements of the array are only read by the worker threads. `read-only` is efficiently implemented by replicating pages of the array on each processor on demand. Reading a page that is not available locally causes the accessing thread to block until the page is available. User-level or compiler-inserted explicit `prefetch` calls can be used to improve performance. Since the page is read-only, no invalidates or updates need to be propagated.

In the `write-many` mode, all threads are permitted to write to the elements of an MSA, but to different elements, i.e. at most one thread is permitted to write to any particular index. `write-many` is efficiently implemented by instantiating a blank local copy of an accessed page on all processors that need the page. No page data and no page invalidates or updates need to be communicated during the phase. The MSA runtime on each processor keeps track of which elements of the page are written, in a local bit-vector. At the end of the phase, the changed data in the local cache are forwarded to the above-mentioned `ChareArray` page object, where they are merged.

In the `accumulate` mode, multiple threads can perform an accumulate operation on any given element. `accumulate` is implemented by accumulating the data into a local copy of the page on each processor, (instantiated on first access), and combining these local values at the end of the phase. Once again, no page data or coherence traffic is transmitted during the phase.

The commutative-associative operation to be used in an `accumulate` is specified at creation of the MSA. In addition, it can be changed at any time by invoking a method of the MSA. Accumulation using the common addition, product and max operations is provided via built-in types. For the general case, setting the `accumulate` operation involves passing in a class that contains `accumulate()` and `getIdentity()` methods. This allows the user to define the accumulate operation. In combination with the `pup` framework, `accumulate` can handle complex operations such as set union, appending to a hash table in which each element is a linked list, and so on.

The user is responsible for correctness and coherence; e.g., if an array is in `write-many` mode, the user must ensure that two processors do not write to the same location. The system assists by detecting errors at run-time.

At startup, the threads accessing an MSA must perform an `enroll` operation for the system to detect the number of worker threads on each processor.

When accessing data, the user does not need to check if data is available in the local cache or not. Unlike DSM, MSA does not use VM hardware page faults to detect whether a local copy of the data exists. Thus, MSA page sizes are not tied to the VM page size but can be controlled by the user as described above.

In principle, every MSA access is checked for whether the data is local or remote with an `if`. Since this is expensive, using compiler support in Jade we strip-mine *for* loops and use local (non-checked) accesses within a page.

3 Example Programs

3.1 Matrix Multiplication

The pseudocode for a matrix-matrix multiplication using a straighforward row-wise decomposition is as follows (assuming $N * N$ matrices and P processors):

```
for i= subsection of size N/P       // Rows of A matrix
  for j=1..N     // Columns of B matrix
    for k=1..N
      result += A[i][k] * B[k][j]
    C[i][j] = result
```

The i dimension is shared among the worker threads. Thus in this case, each thread will request a subset of the rows of A and C, and the entire B matrix. If all matrices are $N * N$ matrices, the required number of elements required by each of P processors works out to $N^2 + 2N^2/P$.

Relevant sections of the corresponding MSA program are shown in Fig. 1. 2D MSA arrays are used in this example. We use row-major data layout for the A and C matrices, and column-major for the B matrix. The page size for each MSA (i.e. the minimum number of elements fetched) is specified when defining the MSA (lines 1–2). In this case we set it to 5000, in order to fetch an entire row of A or C, or column of B. (We could set B's page size to the number of elements in B, and that would fetch the entire B matrix into local memory upon the first access to it.) The page size is often a crucial parameter for performance. The per processor cache size is specified when instantiating the MSA (lines 5–7). We set the cache size to hold at least the number of elements calculated above, unless it is too large a number to fit in the available memory. (The MSA API has not been finalized yet, and it is likely that the template parameters will be reduced.)

Next, consider the following pseudocode for a block decomposition matrix product:

```
for i= subsection of size N/sqrt(P) // Rows of A matrix
  for j= subsection of size N/sqrt(P)  // Columns of B matrix
    for k=1..N
      result += A[i][k] * B[k][j]
    C[i][j] = result
```

Here too, one thread is solely responsible for an element of C and `write-many` mode suffices. The number of elements required by each processor is $2N^2/\sqrt{P}$.

Finally, consider the case where we decompose in the k-dimension as well.

```
for i=subsection of size N/cuberoot{P}     // Rows of A matrix
  for j=subsection of size N/cuberoot{P}   // Columns of B matrix
```

```
typedef MSA2D<double, MSA_NullA<double>, 5000,MSA_ROW_MAJOR> MSA2DRowMjr;
typedef MSA2D<double, MSA_SumA<double>, 5000,MSA_COL_MAJOR> MSA2DColMjr;

// One thread/process creates and broadcasts the MSA's
MSA2DRowMjr arr1(ROWS1, COLS1, NUMWORKERS, cacheSize1); // row major
MSA2DColMjr arr2(ROWS2, COLS2, NUMWORKERS, cacheSize2); // column major
MSA2DRowMjr prod(ROWS1, COLS2, NUMWORKERS, cacheSize3); //product matrix

// broadcast the above array handles to the worker threads.
...

// Each thread executes the following code
arr1.enroll(numWorkers); // barrier
arr2.enroll(numWorkers); // barrier
prod.enroll(numWorkers); // barrier

while(iterate)
{
    for(unsigned int c = 0; c < COLS2; c++) {
        // Each thread computes a subset of rows of product matrix
        for(unsigned int r = rowStart; r <= rowEnd; r++) {

            double result = 0.0;
            for(unsigned int k = 0; k < cols1; k++)
                result += arr1[r][k] * arr2[k][c];

            prod[r][c] = result;
        }
    }

    prod.sync();
    // use product matrix here
}
```

Fig. 1. MSA Matrix Multiplication Code in Jade.

```
for k=subsection of size N/cuberoot{P}
  result += A[i][k] * B[k][j]
C[i][j].accumulate(result);
```

Here, the MSA `accumulate` mode comes in useful. An *Add* accumulator class can be specified as the default when creating the C MSA. MSA also provides templated $Null$, $Product$, Max and Min Accumulators.) The number of elements required by each processor reduces from the previous case to $2N^2/P^{2/3}$.

3.2 Molecular Dynamics

In classical molecular dynamics based on cut-off distance (without any bonds, for this example), forces between atoms are computed in each timestep. If two atoms are beyond a cutoff distance the force calculation is not done (to save computational cost, since force drops as a square of the distance). After adding forces due to all atoms within the cutoff radius, one calculates new positions for each atom using Newtonian mechanics.

The pseudocode for a particular molecular dynamics algorithm using MSA is shown below. The key data structures used are:

– `coords[i]`: a vector of coordinates (x,y,z values) for each atom *i*.
– `forces[i]`: a vector containing forces (x,y,z values) on atom *i*.
– `atomInfo[i]`: a struct/class with basic read-only information about each atom such as its mass and charge.
– `nbrList[i][j]`: is true if the two atoms are within a cutoff distance.

```
// Declarations of the 3 arrays
class XYZ; // { double  x; double y; double z; }
typedef MSA1D<XYZ, MSA_SumA<XYZ>, DEFAULT_PAGE_SIZE> XyzMSA;
class AtomInfo;
typedef MSA1D<AtomInfo, MSA_SumA<AtomInfo>,
              DEFAULT_PAGE_SIZE> AtomInfoMSA;
typedef MSA2D<int, MSA_NullA<int>,
              DEFAULT_PAGE_SIZE, MSA_ROW_MAJOR> NeighborMSA;

XyzMSA coords;
XyzMSA forces;
AtomInfoMSA atominfo;
NeighborMSA nbrList;

//broadcast the above array handles to the worker threads.
...

// Each thread executes the following code
coords.enroll(numberOfWorkerThreads);
forces.enroll(numberOfWorkerThreads);
atominfo.enroll(numberOfWorkerThreads);
nbrList.enroll(numberOfWorkerThreads);

for  timestep = 0 to Tmax {
  /**************** Phase I : Force Computation ****************/
  // for a section of the interaction matrix
  for i = i_start to i_end
    for j = j_start to j_end
      if (nbrlist[i][j]) { // nbrlist enters ReadOnly mode
        force = calculateForce(coords[i], atominfo[i],
                               coords[j], atominfo[j]);
        forces[i] += force; // Accumulate mode
        forces[j] += -force;
      }
  nbrlist.sync(); forces.sync();  coords.sync(); atominfo.sync();

  /**************** Phase II : Integration ****************/
  for k = myAtomsbegin to myAtomsEnd
     coords[k] = integrate(atominfo[k], forces[k]); // WriteOnly mode
  coords.sync(); atominfo.sync(); forces.sync();

  /**************** Phase III ****************/
  if  (timestep %8 == 0) { // update neighbor list every 8 steps
    // update nbrList with a loop similar to the force loop above
    ... nbrList[i][j] = distance(coords[i], coords[j]) < CUTOFF;

    nbrList.sync(); coords.sync();
  }
}
```

There are three phases in each timestep. The `atomInfo` array is `read-only` in all phases. During the force computation phase, the `forces` array is `write-many` whereas the `coords` array is `read-only`; while during the integration phase, this is reversed. Every 8 steps (here) we recalculate the `nbrList` in phase III, where `nbrList` is `write-many` and `coords` is `read-only`. The code assumes a block partitioning of the force matrix as suggested by Saltz[10] or Plimpton[18].

3.3 FEM Graph Partitioning

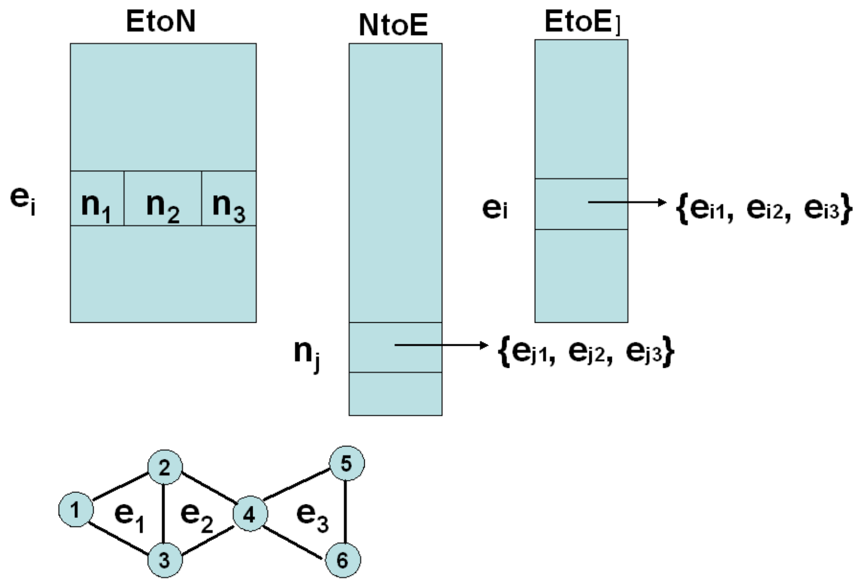

Fig. 2. FEM Graph Partitioning Example.

As an example of the power of the generalized `accumulate` operation, we present a part of a program that deals with an unstructured mesh for a finite-elemment method (FEM) computation. Here, the mesh connectivity data is available at input in the `EtoN` array: for each element i, the `EtoN` [i] contains 3 node numbers (we assume triangular elements). The objective is produce `EtoE` array, where `EtoE` [i] contains all element (numbers) that are neighbors of E. E_1 is said to be a neighbor of E_2 if they share a common node. So, e2 and e3 are neighbors because they share Node 4.

The algorithm for doing this using MSA proceeds in two phases. In the first phase, an intermediate array `NtoE` is created by accumulation: `NtoE` [j] contains all elements that have n_j as their node. To construct this, each thread processes a section of the `EtoN` array. In the second phase, $e_1, e_2 \in NtoE[j]$ are set to be neighbors of each other.

Note that the `accumulate` operations in lines 5, 15 and 16 are actually set-union operations, implemented as described in Section 2.

```
1  // Phase I: EtoN: RO, NtoE: Accu
2  for i=1 to EtoN.length()
3    for j=1 to EtoN[i].length()
4      n = EtoN[i][j];
5      NtoE[n] += i; // Accumulate
```

```
EtoN.sync(); NtoE.sync();

// Phase II: NtoE: RO, EtoE: Accu
for j = my section of j
  //foreach pair e1, e2 elementof NtoE[j]
  for i1 = 1 to NtoE[j].length()
    for i2 = i1 + 1 to NtoE[j].length()
      e1 = NtoE[j][i1];
      e2 = NtoE[j][i2];
      EtoE[e1] += e2; // Accumulate
      EtoE[e2] += e1;
EtoN.sync(); NtoE.sync();
```

4 Related Work

4.1 DSM, TreadMarks, and Munin

Distributed Shared Memory (DSM) is a much-studied software-level shared memory solution. Typically, DSM software uses the virtual memory page fault hardware to detect access to non-local data, which it then handles. It works at the page level, fetching and delivering virtual memory pages. DSM uses relaxed consistency memory models to reduce false sharing overheads and improve performance.[11, 1]

Munin[2, 3] and TreadMarks[15] are DSM implementations. TreadMarks implements the *release consistency* memory model, which typically does not require any additional synchronization over a general shared memory (sequential consistency) program. To reduce false sharing overheads, their *multiple-writer* coherence protocol allows multiple threads to write to independent locations within a page.

Munin takes such coherence optimizations further, and identifies several access modes with correspondingly efficient coherence protocols, as follows:

– *Synchronization:* Global locks were optimized by using a local proxy to minimize global communication.
– *Private:* No coherence.
– *Write-once* These are read-only after initialization. Optimized by replication.
– *Result:* Read by only one thread. Optimized by maintaining a single copy and propagating updates to it.
– *Producer-Consumer:* Optimized by eager update of copies.
– *Migratory:* Optimized by migrating the object.
– *Write-many:* Optimized by a multiple-writer protocol.
– *Read-mostly:* Optimized by replication.
– *General Read-Write:* Uses standard coherence protocol.

Their study of several shared memory programs and their performance results relative to message-passing are impressive and appear to validate their idea of "adaptive cache coherence"[5].

Munin's modes were applied on both a per-object and a per-variable basis. While TreadMarks attempts to maintain the illusion of a general shared address space, Munin requires the programmer to specify the mode for each variable. This was done at compile time and so a variable's mode could not change during the program, and only statically allocated memory was supported. Munin put each shared variable on a separate page of virtual memory.

Comparison: Munin is designed to be a complete shared memory programming model rather than the blended model of MSA. MSA supports Munin's Private and Write-many modes, and introduces a new `accumulate` mode and prefetching commands. Munin's Write-once, Result, and Read-Mostly modes seem to be of limited use, since synchronization will be required at the application level before accessing the updated data; which leads us to believe that these modes are an artifact of Munin's static style of specifying modes. MSA accomplishes these modes by dynamically specifying a `write-many` mode followed by a `read-only` mode. Munin's Producer-Consumer mode with its eager update offers unique features, but, again, given the need for synchronization, a message send might be more efficient. MSA does not support the General Read-Write or the Read-mostly modes.

Specifying portions of an array to be in different modes is not supported in MSA, but this cannot be done in Munin either because of the static specification. Munin's granularity for data movement is the size of a VM page; whereas MSA works physically on a user-defined page size. MSA's user-defined physical page size allows the "page" to be as small as one element, or as large as several rows of a matrix allowing the user (or Jade compiler) to optimize for the expected access pattern. Munin's modes are static, whereas MSA arrays can change their mode dynamically over the life of the program, which leads to needing fewer modes. Furthermore, MSA supports row-major, column-major, and (in the future) other array layouts, which can further improve performance.

DSM systems suffer considerable latency on "misses" and provide no latency tolerance mechanisms since control transfers to the DSM software in kernel space. MSA is implemented in user space, and Charm++'s virtual processors (user level threads) can tolerate latency by scheduling another virtual processor when one thread suffers a "page" miss.

DSM uses page fault hardware to detect non-local access; MSA checks each data access (similar to Global Arrays) and we need to study the cost of this detection mechanism. MSA also has operations that work on data that is known to be available locally (e.g. using `prefetch`), and the Jade compiler can generate code for some such cases. When combined with MSA's prefetch feature, we expect that the efficiency of array element access will approach that of sequential programs.

4.2 Specifically Shared Variables in Charm

Charm (and its earlier version, the Chare Kernel) supported a disciplined form of shared variables by providing abstractions for commonly used modes in which information is shared in parallel programs. [13, 14]. The modes were readonly (replicated on all processors), writeOnce, accumulator, monotonic variable (useful in branch-and-bound, for example), and distributed tables (basically, read-only or writeonce, with distributed storage and caching). However, unlike MSA, it does not support the notion of phases, nor that of pages. Further, the original version did not have threads, and so supported only a split phase interface to distributed tables.

4.3 Global Arrays

The Global Arrays project[17], like ours, attempts to combine the portability and efficiency of distributed memory programming with the programmability of shared memory programming. It allows individual processes of an MIMD program to "asynchronously access logical blocks of physically distributed matrices" without requiring the application code on the other side to participate in the transfer. GA typically uses RDMA[6] and one-sided communication primitives to transfer data efficiently. GA coexists with MPI.

Each block is local to exactly one process, and each process can determine which block is local. GA provides `get`, `put`, `accumulate` (float sum-reduction), and `int read-increment` operations on individual elements of the array. The GA *synchronization* operation completes all pending transfers. *fence* completes all transfers this process initiated. Global Arrays does not implement coherence. It is the user's responsibility to guard shared access by using synchronization operations.

In GA one-sided communication is used to access a block owned by a remote processor: there is no automatic replication or caching of remote data. The user explicitly `fetch`es remote data for extended local access and then directly accesses the data. This mode of access reduces programming simplicity when using GA's, especially if accessing data irregularly across the entire GA. GA RDMA operations are provided to access data remotely in such cases without `fetch`ing a block of data, but at the cost of reduced performance, since every data access is then checked (with an `if`) and redirected to a local or non-local version of the operation.
Comparison: In shared-memory terminology, it appears that GA maintains a single copy of each "page" and either requires the user to `fetch` the page for efficient local access, or propagates updates to the remote page quickly using RDMA. For the former case, MSA's `prefetch` operation provides the same benefits, and for amenable access patterns the strip-mining compiler optimization allows the user to skip using `prefetch`; and for the latter case, MSA's modes allow optimizations that GA cannot support. Like MSA, GA does not tie the "page" size to the VM page size. It allows the "page" to migrate to be closer (i.e. local) to an accessing process. GA seems well-suited to certain access patterns, but, for example, implementing write-many on GA would involve a lot of unnecessary RDMA operations, and the lack of replication makes reading of elements on a "page" by many threads inefficient. The GA accumulate does not support variable-sized elements/pages.

4.4 HPF and others

Other approaches that deal with similar issues include implementation strategies for HPF. For example, the inspector-executor idea [19, 16] allows one to prefetch data sections that are needed by subsequent loop iterations.

Titanium[21] translates Java into C. It adds a global address space and multi-dimensional titanium arrays to Java and is especially suited to grid-structured computations. Each processor on a distributed memory machine maintains its data in a local demesne, and variables can be declared to limit access to only the local demesne of data, or to have unrestricted global access. Several compiler analyses are performed, including identifying references to objects on the local processor. Barriers and data exchange operations are used for global synchronization.

5 Performance Study

As a preliminary performance study, we present results for the row-wise decomposition matrix multiplication program shown in Section 3.1.

Figure 3 shows the speedup for a 2000x5000x300 matrix multiplication on NCSA's Tungsten cluster. When there are 8 threads per processor, the latency of page misses by one thread is better hidden by overlapping with computations for another thread. This effect (the benefit of processor virtualization) can be clearly seen by comparing results for 1 and 8 threads per processor. With a much larger number of threads per processor (32 or 64) the scheduling overhead and fine-grained communication lead to worse performance, although a more detailed study is needed to ascertain that. It should be noted that raw sequential performance is currently unoptimized. With further optimizations, we expect the times to decrease but possibly speedups may decrease.

Figure 4 shows the effect of limiting cache size: with a smaller cache, the time is almost twice as large as that with an adequately large cache. Smaller caches reduce the reuse of fetched data.

6 Summary and Future Work

We described a restricted shared address space programming model called multi-phase shared arrays (MSA), its implementation and its use via examples. MSA is not a complete model, and is intended to be used in conjunction with other information-exchange paradigms such as message passing. It currently supports only 3 modes: read-only, write-many and accumulate. One important idea in MSA is that the modes for each array can change dynamically in different phases of the program separated by synchronization points. The generalized `accumulate` operation supported by MSA is powerful, and is especially useful for accumulating sets, in addition to the more common use in summations. MSA is implemented in Charm++ and AMPI, which support many light-weight threads (virtual processors) per processor. As a result, the latency inherent in

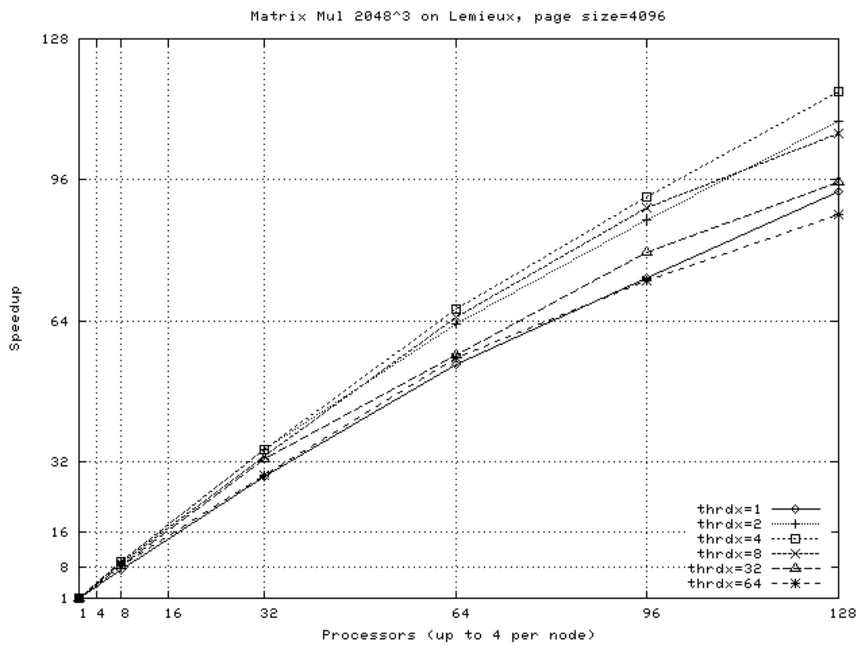

Fig. 3. Scaling with varying number of threads per processor (thrdx).

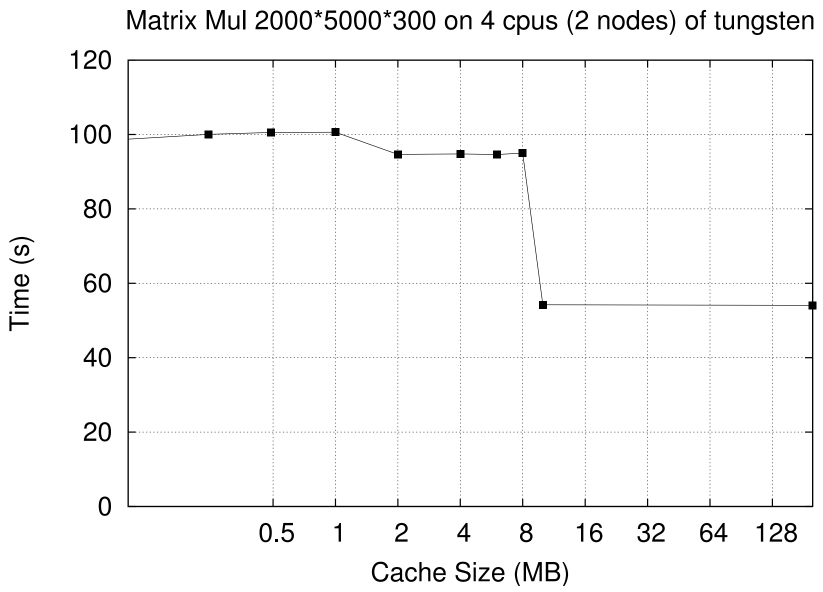

Fig. 4. Effect of MSA software cache size.

page misses is better tolerated. Further, we provide a prefetch operation and correspondingly specialized versions of array accesses, which attains efficiency of sequential code in case of prefetched data.

We plan to search for additional applications where this model is useful. Further, we will explore and support additional access modes beyond the three supported currently. Performance optimization, and detailed performance studies are also planned. We hope that a mixed mode model such as MSA will lead to substantial improvement in programmer productivity, and bridge the current divide between SAS and distributed memory programming styles. Further, compiler support is crucial to simplifying use of MSA, which we plan to explore in the context of the ongoing Jade programming language.

7 Acknowledgements

The authors wish to acknowledge the work of Rahul Joshi on implementing an initial version of MSA, and Orion Lawlor for improving the MSA API and performance.

References

1. S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
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. J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In I. Tartalja and V. Milutinovic, editors, *The cache coherence problem in shared memory multiprocessors: software solutions*. IEEE Computer Society Press, 1995.
4. W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In *Proceedings of 7th International Workshop on Languages and Compilers for Parallel Computing*, number 892 in Lecture Notes in Computer Science, pages 141–154, Ithaca, NY, USA, August 1994. Springer-Verlag.
5. 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.
6. A. Cohen. RDMA offers low overhead, high speed. *Network World*, March 2003. URL http://www.nwfusion.com/news/tech/2003/0324tech.html.
7. J. DeSouza and L. V. Kalé. Jade: A parallel message-driven Java. In *Proc. Workshop on Java in Computational Science, held in conjunction with the International Conference on Computational Science (ICCS 2003)*, Melbourne, Australia and Saint Petersburg, Russian Federation, June 2003.

8. W. Fenton, B. Ramkumar, V. Saletore, A. Sinha, and L. Kale. Supporting machine independent programming on diverse parallel architectures. In *Proceedings of the International Conference on Parallel Processing*, pages 193–201, St. Charles, IL, Aug. 1991.
9. 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 03)*, College Station, Texas, October 2003.
10. Y.-S. Hwang, R. Das, J. Saltz, M. Hodoscek, and B. Brooks. Parallelizing Molecular Dynamics Programs for Distributed Memory Machines. *IEEE Computational Science & Engineering*, 2(2):18–29, Summer 1995.
11. L. Iftode and J. P. Singh. Shared virtual memory: Progress and challenges. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):498–507, 1999.
12. R. Jyothi, O. S. Lawlor, and L. V. Kale. Debugging support for Charm++. In *PADTAD Workshop for IPDPS 2004*, page 294. IEEE Press, 2004.
13. L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.
14. 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.
15. 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.
16. C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Trans. on Parallel and Distributed systems*, 2(4):440–451, 1991.
17. J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A non-uniform-memory-access programming model for high-performance computers. In *Journal of Supercomputing*, volume 10, pages 169–189, 1996.
18. 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.
19. J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.
20. A. Sinha and L. Kalé. Information Sharing Mechanisms in Parallel Programs. In H. Siegel, editor, *Proceedings of the 8th International Parallel Processing Symposium*, pages 461–468, Cancun, Mexico, April 1994.
21. 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.