Supporting Dynamic Parallel Object Arrays

Orion S. Lawlor and Laxmikant V. Kalé

Computer Science Department

Univ. of Illinois at Urbana−Champaign

1304 W. Springfield, 3315 DCL

Urbana, IL 61801−2987

olawlor@acm.org, kale@cs.uiuc.edu

ABSTRACT

We present efficient support for generalized arrays of parallel data driven objects. Array elements are regular C++ objects, and are scattered across the parallel machine. An individual element is addressed by its ‘‘index’’, which can be an arbitrary object rather than a simple integer. For example, an array index can be a series of numbers, supporting multidimensional sparse arrays; a bit vector, supporting collections of quadtree nodes; or a string. Methods can be invoked on any individual array element from any processor, and the elements can participate in reductions and broadcasts. Individual elements can be created or deleted dynamically at any time. Most importantly, the elements can migrate from processor to processor at any time. The paper discusses support for message delivery and collective operations in face of such dynamic behavior. The migration capabilities of array elements have proven extremely useful, for example, in implementing flexible load balancing strategies and for exploiting workstation clusters adaptively. We present the design, an implementation, and performance results.

Additional Keywords: parallel runtime, object migration, parallel hashtable.

1 INTRODUCTION

A perennial problem in computer programming is naming−−if we want something, how and who do we ask for it? This problem is especially acute on parallel machines, because communicating with remote processors is slow.

The situation is akin to the (physical) postal system. We wish to deliver a message to a person−− how can we get the message to them? The imperfect solution adopted by the postal system is to use physical addressing−−we somehow obtain, and then specify, the exact physical location where the person can be found. The problem, of course, is that when a person moves, their mail will either be forwarded (which is slow and duplicates delivery effort), misdelivered (delivered to the current occupant of that address), or lost (returned to the sender or dropped completely).

MPI also uses physical addressing−−messages in MPI are sent to a particular process number and tag. Like the postal service, if the computational entity the message is destined for moves, the message will either have to be manually forwarded or (more likely) will be misdelivered or lost. This means computational entities in MPI programs either never move, or can only be moved with a great deal of intricate, error−prone programming effort.

While it is easy to deliver messages to a physical address, it is evident that moving objects are difficult to support. We present a simple, well−known solution to this problem−−a layer of indirection between objects and physical addresses.

Our framework allows an object to be referenced by a globally unique, problem−domain, user−assigned logical address called an ‘‘array index’’. All communication is via the array index, which allows the object to be migrated in a completely general and user−transparent way.

In this paper, we show how to support message delivery, creation, deletion, and migration scalably and efficiently using this logical addressing scheme. We present a solution to the problem of broadcasts and reductions in the presence of ongoing migrations. Finally, we present performance data from several actual applications built using this system.

1.1 Partition Decomposition

Many of today’s emerging high−end parallel applications are characterized by irregular and dynamic computational structure. These problems demand techniques such as latency hiding and dynamic load balancing to achieve good parallel performance. However, it requires significant programming effort to incorporate these techniques into a parallel program written using the prevalent processor−centric programming model exemplified by MPI.

We abandon this processor−centric model for an object−centric model. We divide the computational work into a large number of parallel objects. Parallel objects resemble processors in that they are self−contained, and can send and receive messages. Unlike processors, however, they can logically addressed, created or deleted, scheduled dynamically, and migrated at run−time to improve load balance. Thus the programmer specifies which actions are to be computed in parallel, and the system decides when and where these actions execute.

Our parallel construct to support this approach is called a dynamic parallel object array, and is built on top of Charm++ [13]. Individual objects are called array elements, and can send and receive messages, participate in broadcasts and reductions, and migrate as needed. Each element of the array is identified by a unique array index, which may be variable−length. Because elements can be individually scheduled and migrated, an ‘‘object array’’ is very different from the ‘‘array objects’’ found in HPF, POOMA [2], P++ [18], Global Arrays [19] and elsewhere. In our construct, each element of the array is a relatively coarse−grained[1] C++ object, with full support for remote method invocation. Our work is quite similar to pC++ [7], but adds migration and reductions. Unlike Concurrent Aggregates [10], Linda [1], or Orca [3], there is no duplication or replication−−message sends address exactly one array element across the entire machine. This work is complemented by fast collective communication libraries such as those of Bartlett, et al [4]; but is not dependent on them.

For example, a large dynamic structural simulation modeled using the finite element method may include 10 million elements in an unstructured mesh. Using our method, the application programmer may decide to partition this mesh into 5,000 chunks using a mesh partitioner such as METIS [15] or Chaco [11]. Each chunk is then implemented as a parallel array element,

[1] The ideal method execution time varies from hundreds of microseconds to a few dozen milliseconds of work.

making a 5,000 element object array. Elements can then communicate with one another without knowing (or worrying about) which processor they reside on.

This approach, which we have been exploring for the past several years, has several advantages:

- As the number of elements is typically much larger than the number of processors, each processor houses several objects. This leads to an adaptive overlap of computation and communication−−while one object is waiting for its data, another object can complete its execution. Scheduling is done dynamically depending on which message arrives first, so this latency hiding requires no additional effort by the programmer.
- Each element's data is small, so our approach can even improve performance on a single processor because of better cache utilization.

Most important, however, is the logical addressing scheme used to deliver messages. With this approach, the run−time system is free to migrate objects across the parallel machine as it pleases, without affecting the semantics of the user program.

The run−time system can use this freedom to effect measurement−based load balancing, for example. During the computation, it can measure the load presented by each element, along with the element communication patterns. It can then remap the objects so as to minimize load imbalance and communication overheads. Even for dynamic applications, such measurement−based load balancing works effectively when load patterns change either slowly or infrequently. We have demonstrated this dynamic load balancer in a production−quality modecular dynamics system, NAMD [9].

Brunner and Kale [8] describe the application−independent dynamic load balancing framework, along with several applications. For example, if the parallel program is using idle desktop workstations, the run−time system can vacate the processors when their owners start using them. Time−shared clusters can also be supported efficiently, by shrinking or expanding jobs [14] to the available set of processors using object migration. In the current paper, however, we confine our attention to the underlying array construct and its implementation.

Our approach is implemented in Charm++ [13], a parallel library for C++. However, to allow the many existing MPI codes to use the load balancing and other facilities of Charm++, we have implemented AMPI [6], an adaptive implementation of MPI, atop Charm++. In AMPI, an MPI process is virtualized as a migratable thread running in an array element. The system thus simulates multiple MPI processors on each real processor, allowing latency hiding and migration−based load balancing. Thus this research is applicable to the wide of class of parallel applications written using MPI as well. For more details on this process, with results from several significant independently developed scientific simulation codes written in Fortran 90 and C++, see [6].

Charm++ already included some support for object arrays [16]. We added to that work dynamic insertion and deletion, arbitrary bitvector indices, scalable migration, and reductions and broadcasts that function with ongoing migrations.

2 MOTIVATING EXAMPLES

We consider several examples to motivate the design and features of parallel arrays.

2.1 Quadtree

Consider a simple heat flow simulation application that discretizes its domain with an adaptive 2D mesh. The mesh is implemented as a quadtree, as shown in Figure 1. Using a separate array element for each leaf of the quadtree would likely result in a tiny grain size and poor performance; so each array element is the root of a small subtree of the mesh.

A quadtree admits a natural indexing scheme−−the directions taken at each level of the tree from the root. Concatenating a binary representation of these directions results in a variable−length bit string that uniquely identifies a leaf of the tree. For example, the element labeled 10;11 in Figure 1 can be reached from the outer box by moving to the lower−left (10) subbox, then to that box's lower−right (11) subbox. For addressing messages to these array elements, our arrays support variable−length indices.

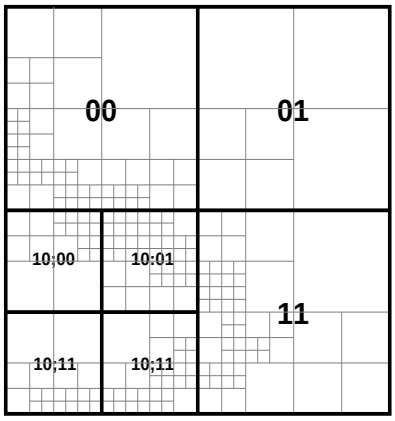

Figure 1. Adaptive 2D quadtree mesh with seven elements, showing element array index and subtrees (in gray).

While running the program, the run−time load balancer[8] collects[2] an object communication graph, as shown in Figure 2, from instrumentation measurements built into the runtime system.

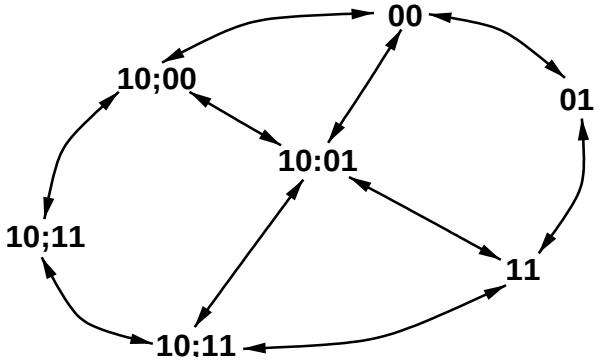

Figure 2. Communication graph for example array.

[2] The local portion of the object communication graph is collected independently on each processor. Load balancers then operate on the distributed graph.

As the computation proceeds, elements will send each other messages to exchange temperatures with their neighbors. They will perform local calculations to propagate heat around their part of the mesh. In a steady−state problem, elements will occasionally contribute their local error values to a reduction to determine whether the convergence criteria have been satisfied. Once the convergence criteria have been met, the program will broadcast a ‘‘report results’’ message to all elements.

The program may decide to create new array elements to refine an existing region. The program may delete array elements when coarsening a region. During the computation, the Charm++ run−time load balancer will migrate elements to improve the load balance. Clearly, the ongoing messaging, broadcasts, and reductions must continue to work even in the face of these migrations, creations, and deletions.

2.2 Document Indexing System

Consider a parallel document indexing system. Each processor accesses a document and parses out a list of words found in the document. The document should then be linked into the document list for each word. In this case, the word itself (a character string) can be used as the array index, with the referenced parallel object storing the word’s document list. The same word−indexed structure could be used to respond to queries.

New words (and hence array elements) will occasionally be encountered and created during the course of the computation. Over time, rarely−used words or misspellings could be deleted from the array. For load balance or better storage utilization, words could be dynamically migrated between processors. Occasionally, summary statistical information such as the average document list length or number of answered queries could be reduced over the entire array.

2.3 Collision Detection

Consider a ‘‘bucket−based’’ contact detection algorithm. Such a computation consists of a group of physical objects scattered through space, some of which may overlap. The problem is to find all overlapping pairs of objects. A slow algorithm is to simply consider all pairs of objects. A natural optimization is to first map objects into disjoint regions of space−−buckets−−and only consider pairs of objects that fall in the same bucket. Of course, a large object may cover several buckets.

If the buckets have a natural indexing (for example, if they form a quadtree or regular grid), then this can be used as the array index for the bucket. Then the collision detection algorithm is to send each object to the appropriate bucket, collide the objects in each bucket independently, and then collect the colliding objects across the array.

Once again, we find dynamic creation (when objects enter new regions of space), deletion (when no objects lie in a region), migration (for load balance), broadcasts (to begin the collision computation), and reductions (to collect the collisions) are all useful operations, and must all work together.

3 API

Parallel array elements are implemented as ordinary C++ classes defined by the user. An array whose elements are of type ***A*** is referenced from other processors via a small, automatically generated, C++ object of type **CProxy_*A***.

Like CORBA, Charm++ reads an IDL−like interface file that describes the object's remotely accessible methods. This interface file is used to generate a proxy, which contains caller−side stubs and callee skeleton C++ code, for each user class. The proxy is compiled and linked along with the regular user code. The details of this process are described in more detail in the Charm++ manual [12]. Charm++ can thus be viewed as a parallel library for C++.

The proxy ckNew method is used to create a new array:

```
CProxy_A ap=CProxy_A::ckNew();
```

The proxy insert call is then used to add array elements:

```
ap[7].insert(parameters);
```

This creates an array element at index 7 on some processor; the variant **insert(*parameters,processor*)** can be used to specify the initial processor explicitly.

The proxy destroy method can be used to delete elements:

```
ap[7].destroy();
```

Elements may be created or destroyed at any time.

User−defined[3] element methods may be invoked as:

```
ap[7].foo(parameters);
```

Like an ordinary C++ method invocation, this call passes the given parameters to the given element method. Unlike C++, the target element need not reside on the same processor, or even in the same address space. Of course, the method may be inherited or dynamically dispatched in the usual C++ fashion.

The array broadcast syntax resembles the method syntax, but omits the index:

```
ap.foo(parameters);
```

This call executes the given method on every array element. An element may also call **contribute** to pass a value to a reduction; or **migrate** to move to another processor. See the example program in Section 10.

3.1 Indexing

For convenience, the system predefines 1D, 2D, and 3D index types. 2D and 3D types are indexed as:

```
int x, y, z;
ap2(x,y).foo(parameters);
ap3(x,y,z).foo(parameters);
```

The more appealing square−bracket '**[x,y]**' syntax cannot be used, because of C++'s unfortunate comma operator.

[3] Unlike system names, user−defined names are displayed here in italic type.

By inheriting from a system index type, a program may define custom array index types.

```
class myIndex : public CkArrayIndex
{
            ...index data...
    public:
            myIndex(...) {nInts=2;...}
};
```

The interpretation of the index data is left to the application, which allows the system to support contiguous 1D, sparse 5D, or tree−structured computations uniformly.

Once defined, a user−defined array index type may be used as:

```
apT[myIndex(...)].foo(parameters);
```

3.2 Terminology and Philosophy

As with Smalltalk, we refer to remote C++ method invocation as ‘‘sending an object a message.’’ The interface file determines whether the remote call is synchronous, for ordinary blocking function call semantics; or asynchronous, for message semantics. As usual, remote message delivery order is not guaranteed.

Unlike many systems, methods are asynchronous by default in Charm++. Asynchronous methods violate the basic function call semantics; but they make concurrency extremely easy to express. For example, to begin some computation on three array elements *i*, *j*, and *k*, we can simply execute:

```
ap[i].go();  ap[j].go();  ap[k].go();
```

The three method invocations will begin executing concurrently. Data can be returned, and control effectively joined, if the called methods execute some ‘‘return method’’ in the caller. Synchronous methods, where the caller simply blocks until each method has finished executing, are also supported.

Also unusual is that Charm++ is nonpreemptive−−messages that arrive for a busy processor are queued until the currently executing method finishes or explicitly blocks. Further, an object is considered to reside on a individual processor of an SMP machine; not a node. These features improve cache utilization, but more importantly enable the run time system and user code to use very few locks, which dramatically simplifies development and slightly speeds execution.

An example of the syntax and philosophy is shown in the ‘‘Simple Example’’ appendix, Section 10.

4 MESSAGE DELIVERY

It is not obvious how to implement this API scalably. In particular, the user may create an element at index 42 on some processor C, then send a message to it from processor A. A must be able to deliver the message despite the fact that A may never have communicated with C. Worse, 42 may migrate to some new processor D while the message is in transit.

Non−scalable methods for location determination are easy to imagine.[4] Processors could be required to broadcast the location of all new or migrated elements. This solution, however,

[4] And frequently implemented in real code!

would waste bandwidth and require every processor to keep track of every array element, which requires a non−scalable amount of storage. Alternatively, a central registry could maintain the locations of all array elements. This conserves bandwidth, but still may have enormous non−distributed storage requirements and also presents a serial bottleneck. Our solution conserves bandwidth, has modest storage requirements, and is well distributed.

4.1 Scalable Location Determination

To solve the location problem scalably, the system can map any array index to a home,[5] a processor that always knows where the corresponding element can be reached. The default index to home function simply returns the hashed array index modulo the number of processors; but we also support user−defined home functions. An element need not reside at its home processor, but must keep its home informed of its current location. In the example above, A will map the index 42 to its home processor B, which will know that 42 is currently living on processor C.

Thus, A sends its message to 42's home B, who then forwards the message to C. Since this forwarding is inefficient, C sends a (short circuit) routing update back to A, advising it to send future messages for 42 directly to C.

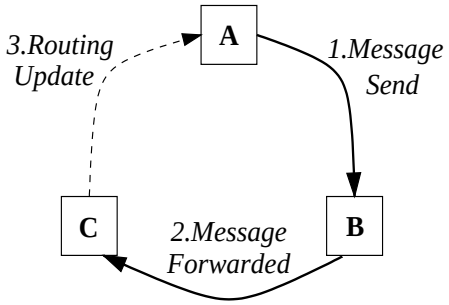

Figure 3. Message forwarding among processors: A, the source; B, the home; and C, the destination

Since elements and homes are scattered across the machine, most forwarded messages must cross the machine twice, wasting cross−section bandwidth. The forward−free alternative−−A asks B where to send, B replies, A sends directly to C−−may use less total bandwidth for large messages, but requires an additional hop in the critical path. Forwarding also generalizes more smoothly to the migration case. Finally, the forwarding approach works quite well when the element actually lives at its home. With either approach, the common case of repeated communication quickly settles to one hop−−that is, with no additional communication overhead.

A simple generalization of this scheme is to use *k* separate mappings to assign *k* homes to each element.[6] Several homes allow messages to be forwarded via any[7] home, saving cross−machine bandwidth, but also requires elements to inform *k* processors when they are created or moved. With *k*=*p*, every processor knows the location of every element, eliminating forwarding; but

---

[5] This same concept is used in many DSM implementations.

[6] With multiple homes, one could imagine creating a system that could survive a node failure.

[7] Typically the home the fewest hops away, or the least loaded.

creations, migrations, and deletions all require a broadcast. The best value of *k* depends on the relative frequency of message forwarding and creations, migrations, and deletions.

4.2 Creation

To create an element, the system need only inform the element's home and call the element constructor. If no processor is specified, the element is created by default at its home processor, which eliminates later message forwarding. It is an error[8] to attempt to create two elements at the same index.

A message may arrive for an element before the element has been created−−this could occur because the create message was delayed on the network. In this case, the system buffers these early messages at home until the element is finally created.

For other applications, a message that arrives for a nonexistent element should result in the creation of the element. For example, in a document indexing system, a document containing a new word that has no corresponding array element will send a ''link to me'' message using that word as an index. Since no processor has any record of an array element at that index, the message will be forwarded to the home processor for that index, which will recognize that the corresponding element does not exist. Thus a new array element is created at that index to handle the message. The new element could be created on any processor. However, it is often most efficient to create the element either at the home processor, where future messages from other processors will be directed; or on the sending processor, which may soon send other messages for the element.

The application specifies the desired early−arrival semantics−−buffered or create−on−demand−−on a per−method basis in the interface file.

4.3 Deletion

To delete an element, the system invokes the object's destructor and informs the element's home processor. No other processors are informed. Any routing cache entries on other processors will remain unused until they eventually expire and are deleted.

More complex methods to reclaim deleted element routing cache entries could be used. When deleting an element, a processor could broadcast a deletion notice. Rather than broadcast, elements could keep track of, and only inform, processors that may have cached their location. A ''deleted list'' could propagate through the system at a low priority. These alternatives are all reasonable, but all use network bandwidth; simple expiration can be completely local and quite efficient.

It is an error for a message to be sent to a deleted element.[9] However, a new element is allowed to reuse an array index vacated by a deleted element.

---

[8] We can easily detect this error when the element's home processor is notified of the second creation.

[9] It is the user's responsibility to ensure this never occurs. The parallel global garbage collector of Piquer [20] would be a substantial improvement.

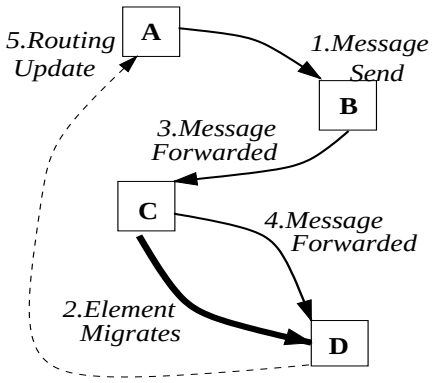

Figure 4. Delivery may require several hops during an element migration.

4.4 Migration

Migration is always under user control−−either explicitly, via a ‘‘migrate’’ call; or implicitly, by enabling run−time load balancing. To migrate an element, the system packs it into a message, sends it to its new location, and informs the element’s home processor of its new location.

A message that arrives for a departed element is forwarded to its last known location, with the usual short circuit routing update (just as in Section 4.1) once it arrives. If an element migrates rapidly and repeatedly, messages may be forwarded an arbitrary number of times (see Figure 4). Of course, migration is normally infrequent, so this pathological case is rare.

Processors that may have cached a migrator’s old location are not informed of the migration. Any stale routing cache entries will be updated upon the next message sent. This lazy update prevents unnecessary traffic and keeps migrations fast. The alternative, to inform all others of your current location after each move, saves time on the first message at the cost of significantly more expensive migration.

Of course, for the common case of repeated communication with stationary elements, the system quickly settles to a single hop.

4.5 Protocol Diagram

Each processor must keep track of each array's local and home elements, as well as maintain a routing cache of ''last−seen'' locations. All this information can efficiently be kept in a per−processor, per−array hash table, keyed by the array index.

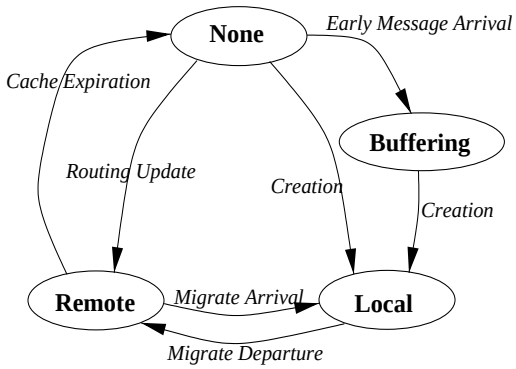

Figure 5. Finite state machine for element information

To deliver a message addressed to an array index, the system looks the index up in its hash table. The represented element will be in one of these states:

- **Local**: the element is on this processor. Messages are delivered directly to the element.
- **Remote**: the element was last seen on another processor; i.e., we have a routing cache entry. Messages for the element are forwarded to that processor. Non−home remote pointers expire if they remain unused for too long.
- **None**: this processor has no idea where the element is located−−the element is not listed in the hash table. Messages for such elements will be sent to their home;[10] or if this is the home, buffered.
- **Buffering**: this processor has messages queued for the element, but the element has not yet been created.[11] This state is only used on an element's home processor.

The element state can change according to the transitions of the finite state machine of Figure 5.

5 COLLECTIVE OPERATIONS

In addition to communicating point−to−point, array elements often need to participate in global operations such as broadcasts and reductions.

5.1 Broadcasts

The semantics of a broadcast are that every existing array element will receive each broadcast message exactly once. Since processors have no shared clock, ''existing'' means created but not destroyed at the instant the broadcast is received on that processor. Array broadcasts are thus

[10] The home is calculated by mapping the array index in the usual way.

[11] This is a rare and short−lived state, but needed because messages may arrive out of order.

first sent to each processor, then delivered to each processor's current local elements. However, this is not enough if there are ongoing migrations.

For example, consider the case where a migrating element leaves processor A before the broadcast is delivered, and arrives on processor B where the broadcast has already been delivered. The migrator may miss the broadcast. Or, reversing the situation, an element may receive a broadcast on processor C, then migrate to D where the broadcast has not yet arrived. When the broadcast reaches D, the migrator may erroneously receive the broadcast again, as shown in Figure 6.

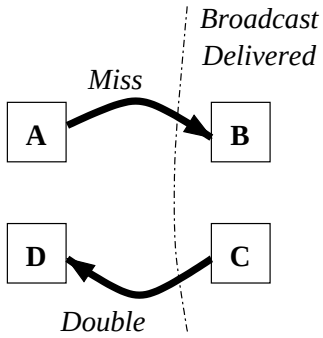

Figure 6. Broadcast delivery problems. Processors A and D have not received the broadcast; processors B and C have.

To solve these problems, the broadcasts are serialized,[12] and processors and elements each maintain a broadcast count. When an element is created, it takes the local processor's broadcast count.

To prevent duplicate delivery, when a broadcast arrives the system compares its count with each element's broadcast count. The system delivers the broadcast only if the count indicates the element has not yet received that broadcast.

To prevent missed broadcasts, the system maintains a buffer of past broadcasts. When an element arrives from migration, the system again compares its broadcast count with the element's. If the element missed any broadcasts while migrating, the element's count will be too low, and the element is brought up to date from the broadcast buffer. Old entries in the broadcast buffer will eventually become useless and must be reclaimed.[13]

Broadcast semantics are easy to enforce when an element is deleted−−simply stop delivering broadcasts to the deleted element. When an element is created, it should receive all broadcasts that arrive at its birthplace after its creation; so a new element's broadcast count is initialized with the local processor's broadcast count.

[12] A broadcast, by definition, must reach every node. Serializing the broadcasts via a single node thus involves little additional cost.

[13] This currently involves a timeout, but much better solutions are clearly possible.

5.2 Reductions

A reduction combines many values scattered across a parallel machine into a single value. A reduction function defines what ‘‘value’’ means and performs the combination. The semantics of a reduction are that each existing element will contribute exactly one value, and the reduction function will be applied to these values in an unspecified order.[14] As before, ‘‘existing’’ means created but not deleted at the time the local reduction completes. Of course, other work may proceed during the reduction.

Reductions can be implemented efficiently by first reducing the values within each processor (the local reduction), then reducing these values across processors. As with broadcasts, in the presence of migration this simple algorithm is not enough.

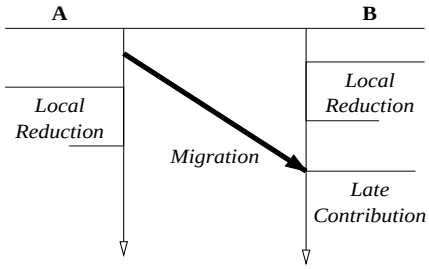

Figure 7. Timeline: reduction skips a migrator. We must ensure the migrator’s contribution is included.

The problem is that during the time a migrating element is in transit, it belongs to no processor.[15] That is, the source processor cannot wait for the migrator’s contribution because it already left; while the destination processor cannot know it is on the way, as shown in Figure 7. Thus the source and destination processors might both complete their local reductions, missing the migrator. However, the reduction must wait until all elements, even migrators, have contributed.

One sensible solution is to count the number of contributed values as the reduction data is collected, and not allow the reduction to complete until the number of values matches the number of elements. Unfortunately, the total number of elements is not available on any processor; and a simple sum of the local element counts will still miss migrating elements.

The approach we use is to sum the net births−−the total number of elements created on a processor minus the total destroyed on that processor.[16] Because of migration, this number may be negative if elements often migrate in and are destroyed (e.g., on ‘‘graveyard’’ processors).

Since for each processor i, the net births n_i is defined as:

$$n_i = c_i - d_i$$

Thus summed across all processors:

$$\sum n_i = \sum (c_i - d_i) = \sum c_i - \sum d_i = c_{total} - d_{total}$$

Summed across all processors, then, we have the total number of elements created but not yet deleted, which is the global element count.

[14] If the order matters, one can use the list−making reduction function to *collect* the data first.

[15] Note that sending another, separate message for synchronization doesn’t actually help.

[16] The net births count is measured at the instant the local reduction completes.

Thus the reduction algorithm actually used is:

- At each processor, collect contributed values from local elements until all current local elements have contributed.[17] At that point, apply the reduction function to the collected values and add the result, contribution count, and the current net births to the across−processor reduction.
- Reduce the values and add the contribution count and net births across all processors.
- As migrators make their late contributions, send their values directly to the root. Once the contribution count equals the total net births, return the reduced value to the user.

The reduction semantics are also slightly more difficult to enforce in the presence of creations and deletions. Element creations are relatively easy−−the net births counter is incremented and the element receives the reduction count of the local processor.

If an element is deleted that has not yet contributed to the current reduction, we simply decrement the net births counter. If the element has already contributed, we must ensure it is included in the net births count for this reduction; but for future reductions it is not included. This is easy to implement with a simple delayed net births adjustment.

6 PERFORMANCE

We have extensively analyzed the performance of the array support, as summarized below.

6.1 Theoretical

Notation:

p the number of processors on the parallel machine
n the total number of array elements
l_i the number of local array elements on processor i
r_i the number of remote elements recently referenced by processor i
h_i the number of elements with processor i as their home

Element creation and deletion, since they only involve the current processor and the element's home, require O(1) time and 1 message. Migration requires O(1) time and 2 messages[18]. Message delivery may require an unbounded number of messages, but only if the element migrates as fast as the message travels. Repeated messages to stationary elements take O(1) time and 1 message.

The local, element−wise operations during reductions and broadcasts require time in O(l_i) on processor i. Without migration, the cross−processor phase of a broadcast or reduction tree requires $p-1$ messages and completes in $\lceil \log_b p \rceil$ hops, with b the tree branching factor (typically 2 to 16).

[17] As with broadcasts, we use a per−element and per−processor reduction count to determine who still needs to contribute.

[18] One message transports the element, one updates the home processor's routing table.

The storage consumed by the element hash table on processor i is $O(l_i+r_i+h_i)$. If each element communicates with a bounded number of other elements, $r_i \in O(l_i)$. If elements and home processors are distributed relatively uniformly, l_i and h_i will both be near n/p. Subject to these assumptions, each processor's hash table requires storage in $O(n/p)$. In the worst case, l_i, r_i and h_i are all bounded by n, so the storage is still in $O(n)$.

6.2 Single−Processor

The system was implemented on Charm++ [13], which also includes non−indexable, non−migratable parallel objects called chares. Table 1 compares the single−processor software overhead for preparing, scheduling, and receiving a message using these non−migratable objects and the array elements described in this paper.

Table 1. Comparison of software overhead with non−arrays

Type	***Linux PC***[19]	***Cray T3E***[20]	***IBM SP3***[21]
Chares	0.92 μs	2.03 μs	1.62 μs
Array Elements	1.85 μs	9.64 μs[22]	4.33 μs

The migration layer adds a few microseconds of overhead to each message. For grain sizes over a few hundred microseconds, array elements add negligible overhead.

6.3 Multiple−Processor

Below, we plot the total time taken for various array operations for varying numbers of processors. In these plots, ‘‘broadcast/reduction’’ means a small broadcast to every array element followed by a reduction across all array elements. ‘‘Migration’’ means the time for a small array element to be packed, shipped across the network, unpacked, and the home processor informed. ‘‘Message’’ means the time to send a short message from one array element to another across processors.

The array elements are distributed in 1D with 16 elements per processor, scaling up with processors. The operations run on every element across the machine simultaneously, and are repeated several thousand times to factor out startup overhead and include any induced non−critical−path load. For migration and messaging, the time reported is the wall clock time for one element to send one message or migrate once. For broadcast/reduction the time reported is wall−clock for one broadcast/reduction cycle. The first data point is with two processors so migration is meaningful.

[19] 400 MHz AMD K6−3, Linux 2.4.0t10, egcs−2.91.66 −O3

[20] 450 MHz DEC Alpha, UNICOS 2.0.5.44, Cray C++ 3.3.0 −O

[21] 375 MHz IBM Power3, AIX 4.3.3, VisualAge C++ 5 −O

[22] The Cray C++ compiler does not support templated member functions, so this version is implemented using function pointers, which cannot be inlined and are significantly slower.

Table 2. Cross−processor performance for 2−32 processors

Type	*Cray T3E*	*IBM SP3*
Short Message	43−56 µs	31−43 µs
Object Migration	151−161 µs	55−77 µs

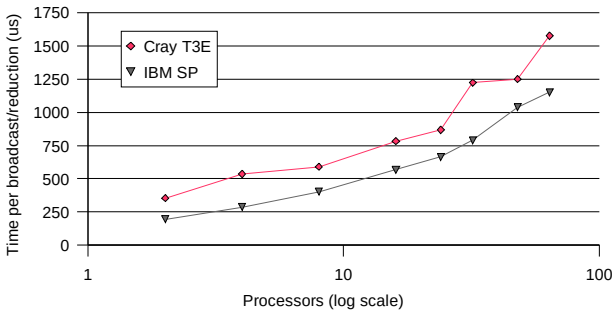

Figure 8. Time per broadcast/reduction operation

The system is indeed highly scalable. The time for collective operations, which should be logarithmic in the number of processors, is slightly worse than expected; but still reasonable.

6.4 Application Performance

We present results from only two of the many applications built using arrays.

The first result comes from an explicit finite−element code. Here, each array element represents a partition of the finite element mesh. At each timestep, each array element loops over the triangles in its mesh partition, adding forces to its local nodes. Next, we exchange forces for the nodes that are shared across processors−−the partition boundary nodes−−by sending messages to our neighboring partitions. Finally, the nodes are moved based on the net forces. Thus each timestep consists of some serial work (looping over triangles), some communication (exchanging forces), and some additional serial work (moving nodes).

We examine two 2D finite−element problems on two parallel machines. The smaller mesh consists of some 600,000 triangles and 300,000 nodes; it was run on an SGI Origin2000.[23] A serial timestep for this problem takes 0.63 seconds. On 128 processors each timestep takes 3.5 milliseconds, a speedup of 178. Speedup for this problem is superlinear because for a given problem size, cache performance improves significantly as processors are added, and the problem shifts from memory−bandwidth bound to compute−bound.

The larger mesh consists of 3.1 million triangles and 1.5 million nodes; it was run on ASCI Red. A serial timestep takes approximately 8 seconds[24]; on 1024 processors each timestep takes 7.1 milliseconds, a speedup of approximately 1150. Again, the performance is superlinear.

[23] 250 MHz MIPS R10000, IRIX64 6.5, MIPSpro C++ 7.3.1.2m

[24] This is estimated from the 2−processor time; the problem is too large to fit on one processor.

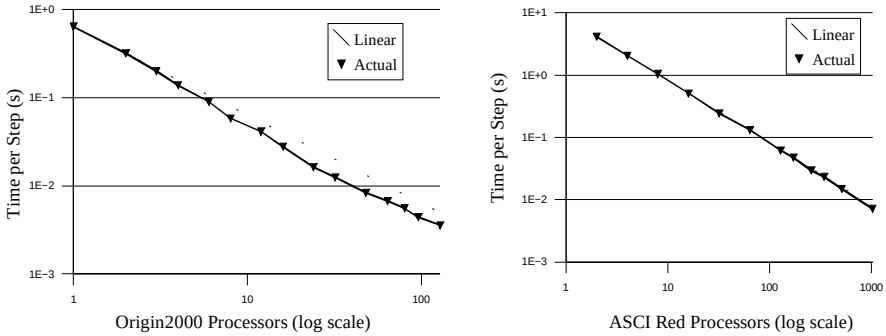

Figure 9. Time per step for an array−based finite−element code.

The second result comes from a collision detection algorithm. The basic approach is voxel−based, as outlined in Section 2.3 and described in more detail by one of us [17]. This particular implementation begins with a ‘‘start step’’ broadcast, sends object lists to array elements, synchronizes, independently collides the object lists, the reduces over the resulting collision list.

We present the results from a scaling benchmark, with a fixed 65,000 triangles per processor, on the ASCI Red machine. The time shown is the wall−clock time per complete collision detection.

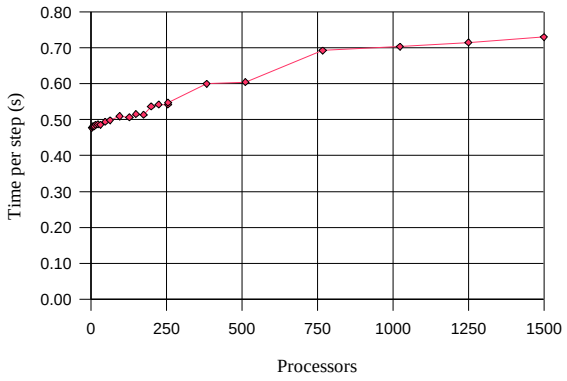

Figure 10. Time per execution for a scaling collision−detection problem

The parallel efficiency on 1,500 processors is 65% (mostly synchronization overhead), for a speedup of 980. The dataset used for this run was almost 100 million polygons.

7 APPLICATIONS

In an application, an array element may:

- Represent a single data item. This approach may appear attractive and general, but is usually much too fine−grained for reasonable performance. For example, in a graphics manipulation program, each pixel of an image could be implemented as a separate array element. But because almost any conceivable processing on a single pixel will happen in just a few microseconds, the time spent doing useful work would be small compared to messaging (dozens of microseconds) and run−time overhead (several more microseconds).

- Represent a group of data items. This is the canonical usage of array elements, as it leads to good performance. Choose the number of items to aggregate so the array element grain size is reasonable. For example, in a graphics manipulation program, an element could represent a 32x32 patch of pixels.[25] If the per−step processing needed by this patch takes a millisecond or more, this system will have good parallel efficiency.
- Represent a thread, processor, or other object. This approach is often taken in simulators, emulators, and run−time support systems.

7.1 Programs

Our array system has been used by a number of highly scalable Charm++ libraries and programs.

- AMPI [6] virtualizes MPI processors as array elements, implementing MPI calls as array method invocations, broadcasts, and reductions. Thus, legacy MPI programs written in C or Fortran can take advantage of automatic load balancing with only minor modifications. A large integrated physical simulation code for solid rocket boosters has been run on AMPI with minimal effort and excellent performance [6].
- The Charm++ finite−element method framework [5] represents partitions of a finite element mesh as array elements. The framework includes Fortran 90 bindings, which are used by several significant engineering applications. Crackprop, a 3D pressure−driven crack propagation code, is a classic finite element structures code. A 3D adaptive mesh dendritic growth metal solidification simulation also uses the framework.
- POSE, a discrete event simulation framework, uses array elements as objects in a discrete event simulation. The objects participate in a global virtual time algorithm. Objects checkpoint their state, optimistically attempt to advance in time, and potentially roll back if they advanced too far.
- A simulator for Blue Gene, an advanced parallel machine from IBM, simulates Blue Gene microprocessor chips as array elements. Using 96 physical processors, we were able to simulate a machine with 8 million simultaneous threads distributed over 40,000 array elements with good efficiency [21].

8 CONCLUSIONS

We have presented efficient support for a general logical addressing scheme for parallel objects. The array index is a user−defined structure, supporting multidimensional, sparse arrays as well as structures such as trees. Objects may be efficiently created, deleted, or migrated at any time; and even in the face of these operations, the system supports array−wide broadcasts and reductions efficiently.

This system has proved to be a robust and useful foundation for several significant applications. Future work on this system will include: further optimization of the implementation; implementing the *k*−homes approach described in Section 3.1; further optimizing communication via message aggregation; and optimization of the collective operations.

[25] This kind of blocking is often already done in serial code, for better cache performance.

9 REFERENCES

[1] S. Ahuja, N. Carriero, D. Gerlenter. ‘‘Linda and Friends,’’ *IEEE Computer*, pages 26−34, August 1986.

[2] S. Atlas, S. Banerjee, J. C. Cummings, et al. (presented by J. Reynders). ‘‘POOMA: A high performance distributed simulation environment for scientific applications,’’ *Supercomputing 1995*, November 1995.

[3] H. Bal, M. Frans Kaashoek, and A. Tanenbaum. ‘‘Orca: A language for parallel programming of distributed systems.’’ *IEEE Transactions on Software Engineering*, 18(3):190−205, March 1992.

[4] M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn and J. Watts. ‘‘Interprocessor Collective Communication Library,’’ *Supercomputing 1994*, November 1994.

[5] M. Bhandarkar, L.V. Kalé. ‘‘A Parallel Framework for Explicit FEM,’’ *Proceedings of the International Conference on High Performance Computing (HiPC 2000)*, Vol. 1970, pp. 385−394. Springer Verlag, December 2000.

[6] M. Bhandarkar, L.V. Kalé, E. Sturler, J. Hoeflinger. *Object−Based Adaptive Load Balancing for MPI Programs*, PPL Technical report 00−03, University of Illinois at Urbana−Champaign, September 2000.

[7] F. Bodin, P. Beckman, D. Gannon, et al. ‘‘Distributed pC++: Basic Ideas for an Object Parallel Language,’’ *Scientific Programming*, Volume 2/Number 3 Fall 1993.

[8] R. Brunner and L.V. Kalé. ‘‘Adapting to Load on Workstation Clusters,’’ *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pp. 106−112. IEEE Computer Society Press, 1999.

[9] R. Brunner, J. Phillips, and L.V. Kalé. ‘‘Scalable Molecular Dynamics for Large Biomolecular Systems.’’ *Proceedings of Supercomputing 2000*, Dallas, TX. November 2000.

[10] A. Chien and W. Dally. ‘‘Concurrent Aggregates,’’ *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pp. 187−196. Seattle, WA, March 1990.

[11] B. Hendrickson, R. Leland. ‘‘Chaco−−A Multilevel Algorithm for Partitioning Graphs,’’ *Supercomputing 1995*, November 1995.

[12] L.V. Kalé et al. *Charm++ Programmer’s Manual*, http://charm.cs.uiuc.edu/, 2000.

[13] L.V. Kalé and S. Krishnan. ‘‘Charm++ : A Portable Concurrent Object Oriented System Based On C++.’’ *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, ACM Sigplan Notes, Vol. 28, No. 10, pp. 91−108. September 1993.

[14] L.V. Kalé, S. Kumar, J. DeSouza. *An Adaptive Job Scheduler for Timeshared Parallel Machines*, PPL Technical Report 00−02, University of Illinois at Urbana−Champaign, September 2000.

[15] G. Karypis, V. Kumar. ‘‘METIS−−Multilevel Algorithms for Multi−Constraint Graph Partitioning,’’ University of Minnesota TR #98−019, http://www−users.cs.umn.edu/~karypis/publications/, 1998

[16] S. Krishnan and L.V. Kalé. ‘‘A parallel array abstraction for data−driven objects,’’ *Proc. Parallel Object−Oriented Methods and Applications Conference*, February 1996.

[17] O. Lawlor. ‘‘A Grid−Based Parallel Collision Detection Algorithm,’’ PPL Technical Report 00−05, University of Illinois at Urbana−Champaign, May 2001.

[18] M. Lemke, D. Quinlan. ‘‘P++, a Parallel C++ Array Class Library for Architecture−Independent Development of Structured Grid Applications,’’ *ACM SIGPLAN Workshop* 1992. pp. 21−23.

[19] J. Nieplocha, R.J. Harrison, and R.J. Littlefield. ‘‘Global Arrays: A nonuniform memory access programming model for high−performance computers,’’ *The Journal of Supercomputing*, Vol. 10, pp. 197−220, 1996.

[20] J. Piquer. ‘‘Indirect distributed garbage collection: handling object migration,’’ *ACM Trans. Program. Lang. Syst.* 18, 5, pp. 615 − 647. September 1996.

[21] N. Saboo, A. Singla, J. Unger, and L.V. Kalé. ‘‘Emulating Petaflops Machines and Blue Gene,’’ *Workshop on Massively Parallel Processing at IPDPS 2001*. April 2001.

10 SIMPLE EXAMPLE

This simple example program implements a stencil−on−regular−grid computation, such as a Jacobi relaxation or SOR. We use one array to represent the entire grid, with each array element representing a small rectangular block. To focus on the parallelism, we hide the numerical details in a class named ‘‘Serial,’’ which stores and manipulates the actual grid values.

The parallel structure of the application is extremely simple. We start by creating the array and elements (in `Main::Main`). At each timestep, each element sends its boundary information to its neighbors (in `sendBorders`), waits until it has received all its neighbor’s boundaries (via `recvBorders`), and then performs the serial computation (in `bordersDone`). Occasionally, we contribute to a global error reduction (which calls `iterationDone`) to determine if the solution has converged. If it has, we exit; otherwise we go on to the next timestep.

This is a complete parallel application−− nothing has been omitted. Since array elements are not related to the number of processors, the program can run with any mesh size on any number of processors. If there are several array elements per processor, communication and computation will automatically overlap. By including a Pack/UnPack routine and using logical addressing, the array elements can be migrated. Thus this program can already use the automatic load balancer, can vacate workstations, and can shrink/expand to the available number of processors.

10.1 Serial Interface File (C++)

```
#define STEPS_PER_ERRCHECK 16
typedef enum { UP=0,DN=1,LT=2,RT=3 } Dir;
class Serial {
    int stepCount;
    int height, width;
    double *temperatures;
public:
    Serial(int xLocation,int yLocation);
    int borderSize(void);
    double *getBorder(Dir dir);
    void setBorder(const double *d,Dir to);
    void compute(void);

    int getStep(void) const;
    double getLocalError(void) const;

    Serial(); //These are needed for migration
    void pup(PUP::er &p);
};
```

10.2 Parallel Interface File (ci)

```
mainmodule jacobi {
    readonly int Xsize;
    readonly int Ysize;
    readonly CProxy_Jacobi Jproxy;
    mainchare Main { entry Main(); };
    array [2D] Jacobi {
        entry Jacobi();
        entry void sendBorders(void);
        entry void recvBorder(Dir to,int size,double border[size]);
    };
};
```

10.3 Parallel Implementation File (C++)

```
#include "jacobi.decl.h"  /*For the generated class CProxy_Jacobi */
#include "jacobi.def.h"
#include "util.h"   /*For the Serial and Dir types*/

/* These global variables are shared across all processors */
int Xsize,Ysize; // Dimensions of the parallel array
CProxy_Jacobi Jproxy; // Main parallel array
```

```
//These helper functions implement boundary condition wraparound
int Xwrap(int x) { return (x+Xsize)%Xsize; }
int Ywrap(int y) { return (y+Ysize)%Ysize; }

/*This routine is called when a global error reduction completes*/
void iterationDone(void *userParam,int dataLen,void *data)
{
    double totalError=*(double *)data;
    if (totalError<1.0e-3)
        {CkPrintf("Converged.\n");CkExit();}
    Jproxy.sendBorders();//Broadcast send, to start the next iteration
}
/* The Main object is called on only one processor (unlike MPI)
 * to begin the computation. It creates our Jacobi array. */
class Main : public Chare {
public:
    Main(CkArgMsg *m)
    {
        Xsize = atoi(m->argv[1]); Ysize = atoi(m->argv[2]);
        delete m;
        //Build the entire array (scatters elements across processors)
        Jproxy.ckNew();
        for (int x=0;x<Xsize;x++)
          for (int y=0;y<Ysize;y++)
            Jproxy(x,y).insert();
        Jproxy.doneInserting();
        Jproxy.setReductionClient(iterationDone);
    }
};

/* This is the array element that does all the work.
 * Instances of this class are scattered across the machine. */
class Jacobi : public ArrayElement2D {
    Serial serial; //Our portion of the problem domain mesh
    int borderCount; //Number of borders we've received so far
public:
    Jacobi(void) :serial(thisIndex.x,thisIndex.y), borderCount(0)
    {
        sendBorders(); //Immediately start working
    }

    /* Send border values to each of our neighbors */
    void sendBorders(void)
    {
        int size=serial.borderSize();
        int x=thisIndex.x, y=thisIndex.y; //Shorthand
        Jproxy(x,Ywrap(y+1)).recvBorder(UP,size,serial.getBorder(UP));
        Jproxy(x,Ywrap(y-1)).recvBorder(DN,size,serial.getBorder(DN));
        Jproxy(Xwrap(x-1),y).recvBorder(LT,size,serial.getBorder(LT));
        Jproxy(Xwrap(x+1),y).recvBorder(RT,size,serial.getBorder(RT));
    }
    /* Receive one set of border values sent by a neighbor.
     * We count the total number of borders we've received so far. */
    void recvBorder(Dir to,int size,double *border)
    {
        serial.setBorder(to,border);
        borderCount++;
        if (borderCount==4) { //Now I have all my borders
            borderCount=0;
            bordersDone();
        }
    }
    /* We now have all the border patches--do a timestep */
    void bordersDone(void)
    {
        serial.compute();
        if ((serial.getStep()%STEPS_PER_ERRCHECK)==0)
        { //Sum up local errors via a reduction (passed to iterationDone)
            double localErr=serial.getLocalError();
            contribute(sizeof(localErr),&localErr,CkReduction::sum_double);
        }
        else //Just start the next step immediately
            sendBorders();
    }

    /* This constructor and ''pup'' routine are all that's needed
     * to support migration, even on distributed-memory machines. */
    Jacobi(CkMigrateMessage *m) { }
    virtual void pup(PUP::er &p)
    {//Pack/UnPack data to/from network, disk, etc
        ArrayElement2D::pup(p);
        serial.pup(p);
        p(borderCount);
    }
};
```