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.

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

1 INTRODUCTION

A perennial problem in computer programming is naming−− if we want something, how do we ask for it? Where do we look for it? This problem is especially important on a parallel machine, 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 after a great deal of intricate, error−prone work.

While it is obviously easy to deliver messages to a physical address, it is evident that moving objects are quite difficult to support with this approach. We present a 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. Techniques such a latency hiding and dynamic load balancing are needed to efficiently parallelize these applications. However, incorporating these techniques into a parallel program written using the prevalent processor−centric programming model, exemplified by MPI, requires significant programming effort.

An alternative approach abandons the processor−centric model for an object−centric model. The computational work is divided 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.

This approach, which we call multi−partition decomposition, separates the task of specifying parallelism from the issues of load balancing, and efficient execution in general. The programmer then specifies which actions are to be computed in parallel, leaving the system to decide when and where these actions execute.

Our parallel construct that supports this approach called a dynamic parallel object array, built on Charm++[1]. 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 quite distinct from the "array objects" found in HPF, POOMA[13], P++[14], Global Arrays[12] 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++[8], but adds migration and reductions. Unlike Concurrent Aggregates[9], Linda[7], or Orca[10], 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 [11]; but 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

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

programmer may decide to partition this mesh into 5,000 chunks using a mesh partitioner such as METIS[16] or Chaco[17]. Each chunk is then implemented as a data−driven[1] array element, making a 5,000 element object array. Elements communicate among one another without 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 because of cache effects our approach often improves performance even on a single processor.

Most important, however, is the logical addressing scheme presented 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.

If the parallel program is using idle desktop workstations, the run−time system can "vacate" the processors when their owners start using them, as described in [2]. Time−shared clusters can also be supported efficiently, by shrinking or expanding jobs to the available set of processors[5] using object migration.

Our research group has been engaged in developing this approach. The general, automatic measurement−based load balancing framework has been described in [2]. In the current paper, we confine our attention to the underlying array construct and its implementation.

Our approach is implemented in Charm++[1], a parallel library for C++. However, due to the popularity of MPI, and to allow 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 several significant independently developed scientific simulation codes written in Fortran 90 and C++, see [6].

2 MOTIVATING EXAMPLES

2.1 Quadtree

Consider a simple heat flow simulation application which discretizes its domain with an adaptive 2D mesh. The mesh is implemented as a quadtree, as 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 which 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, the variable−length index supported by this work is ideal.

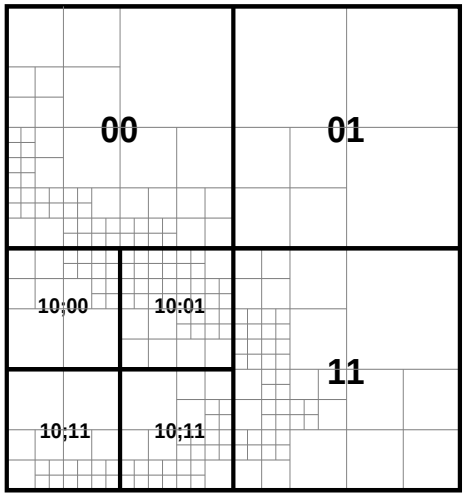

Figure 1. Adaptive 2D quadtree mesh with seven elements, showing element array index and subtrees (in gray).

While running the program, the runtime load balancer[2] will collect the object communication graph shown in Figure 2.

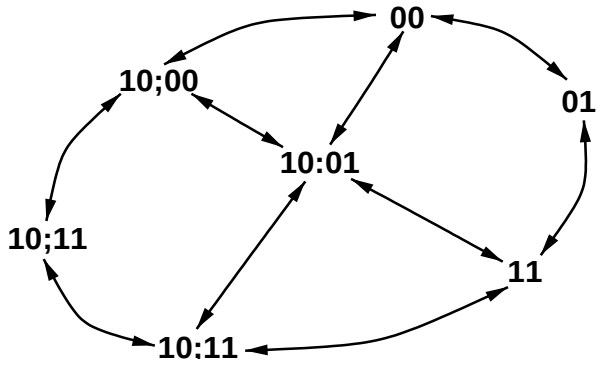

Figure 2. Communication graph for example array.

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 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 objects scattered across space, some of which may overlap. The problem is to determine which pairs of objects are in contact. 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, and must all work together.

3 API

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 which describes the object's remotely accessible methods. This interface file is used to build the proxy, which contains caller−side stubs and callee skeleton C++ code. The proxy is compiled and linked along with the regular user code. The details of this process are quite similar to CORBA, and described in more detail in [4]. 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 version **insert(*parameters,processor*)** can be used to explicitly specify the initial processor.

The proxy destroy method deletes elements:

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

Elements may be created or destroyed at any time.

User−defined[2] 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 Charm++ inherits C++'s unfortunate comma operator.

[2] 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 "joined", if the called methods execute some "return method" of the caller. Of course 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 runtime 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

A scalable implementation of this API is rather subtle. 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 location determination schema are easy to imagine[3]. Processors could be required to broadcast the location of all new or migrated elements. This solution, however, would waste

[3] And frequently implemented in real code!

bandwidth and require every processor to keep track of every array element, which requires a non−scalable amount of storage. Alternately, 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[4], 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 user−defined functions are also supported. 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.

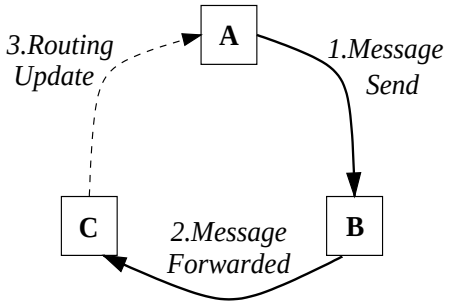

Figure 3. Message forwarding among processors: A, the source; B, the home; and C, the destination

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 home. With either approach, the common case of repeated communication quickly settles to 1 hop−− that is, zero added communication overhead.

A simple generalization of this scheme is to use *k* separate mappings to assign *k* homes to each element. Several homes allow messages to be forwarded via any[5] home, saving 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 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] This exact concept is used in many DSM implementations.

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

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 explicitly specified, the element is created by default at its home processor, which eliminates later message forwarding. It is an error to attempt to create two elements at the same index.

A message may arrive for an element before the element has been created. Some applications, such as the quadtree example, explicitly create all their elements. In this case, the system buffers these early messages 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 which 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 have other messages for the element.

The application specifies which early−arrival semantics is desired 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.

Alternative, more complex methods to reclaim deleted element routing cache entries could be used. When deleting an element, a processor could broadcast a funeral notice. Elements could keep track of which processors may have cached their location and send a funeral notice to each. A systolic body wagon 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 arrive for a deleted element[6]. However, a new element is allowed to reuse an array index vacated by a deleted element.

[6] It is the user's responsibility to ensure this never occurs. The parallel global garbage collector of [15] would be a substantial improvement.

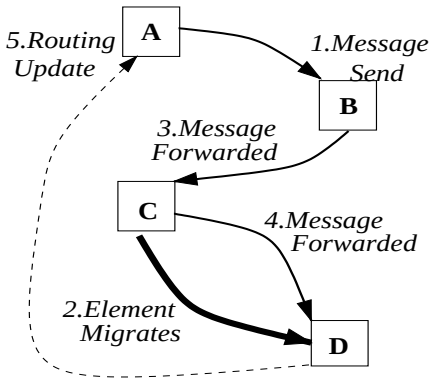

Figure 4. Delivery may require several hops during an element migration.

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 stops the object, packs it into a message, and sends it to its new location. Once the element arrives it is unpacked and the element's home processor is informed of the element's new location.

A message that arrives for a departed element is forwarded to its last known location, with the usual short circuit routing update 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 which 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 actively inform all others of your current location, 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 1 hop.

4.5 Protocol Diagram

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

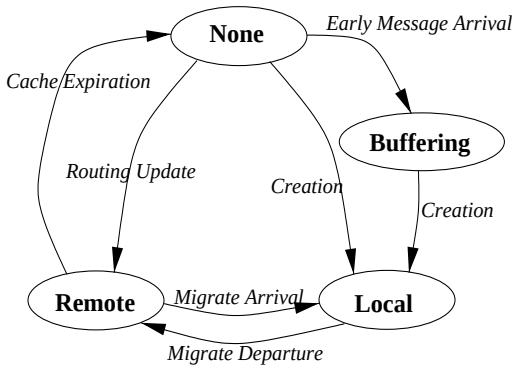

Figure 5. Finite state machine for element information

To deliver a message addressed to an array index, the system looks the index up in its hashtable. 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 hashtable. Messages for such elements will be sent to their home[7]; or if this is the home, buffered.
- **Buffering**: this processor has messages queued for the element, but the element has not yet been created[8]. 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

[7] As calculated by mapping the array index in the usual way.

[8] 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 (Figure 6).

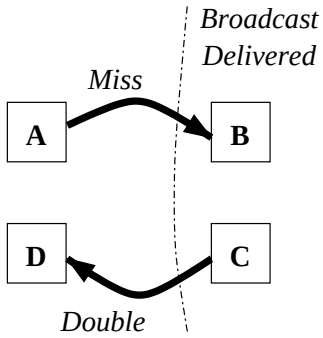

Figure 6. Broadcast delivery problems. Processors A and D have not received the broadcast; processors B and C have.

To solve these problems, the broadcasts are serialized[9], 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 count will be too low, and the element is brought up to date from the broadcast buffer. The broadcast buffer is periodically garbage collected on each processor, removing broadcasts older than any plausible migration delivery time.

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.

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

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

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[10]. 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.

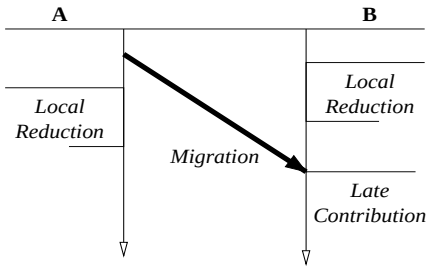

Figure 7. Timeline: reduction skips a migrator. We must ensure the migrator's contribution is included.

The problem is that during the time a migrating element is in transit, it belongs to no processor[11]. 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 (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[12]. 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.

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

[11] A non−blocking control handoff without an in−between period is impossible−− it is an *n*−way handshake problem.

[12] 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[13]. 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, contribution count, and current net births across all processors to the root processor.
- 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 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 update" 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[14]. 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).

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

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

The storage consumed by the element hashtable 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 hashtable 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++ [1], 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*[15]	*Cray T3E*[16]	*IBM SP3*[17]
"Chares"	0.92 μs	2.03 μs	1.62 μs
Array Elements	1.85 μs	9.64 μs[18]	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.

[15] 400 MHz AMD K6−3, Linux 2.4.0t10, egcs−2.91.66 −O3

[16] 450 MHz DEC Alpha, UNICOS 2.0.5.44, Cray C++ 3.3.0 −O

[17] 375 MHz IBM Power3, AIX 4.3.3, VisualAge C++ 5 −O

[18] 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

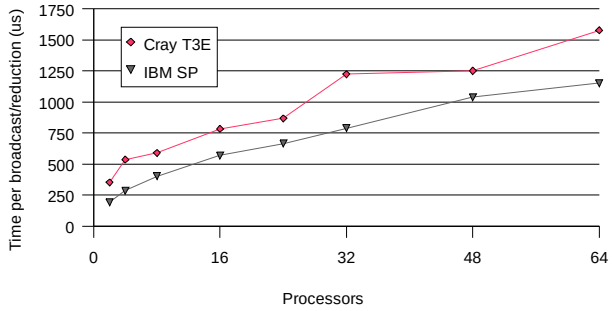

Figure 8. Time per broadcast/reduction operation

The system is indeed highly scalable. Theoretically, we expect the broadcast/reduction time curve to be logarithmic in the number of processors; the system meets our expectation.

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 which 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).

The mesh consists of some 600,000 triangles and 300,000 nodes. Since the mesh is large, on a single processor the computation is memory−bandwidth bound, and each timestep takes approximately one second. Figure 9 shows the speedup obtained on an SGI Origin2000[19].

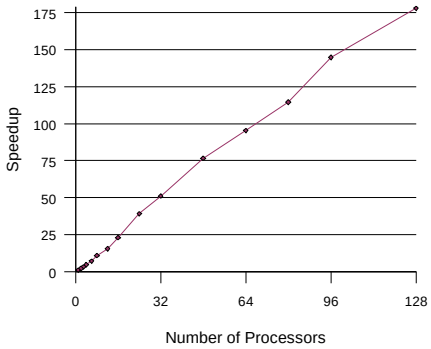

Figure 9. Speedup for an array−based finite−element code.

[19] 250 MHz MIPS R10000, IRIX64 6.5, MIPSpro C++ 7.3.1.2m

As expected, 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 parallel efficiency peaks at 163% on 24 processors; the efficiency at 128 processors is 139% (for a speedup of 178). On 128 processors, each timestep takes 3.5 milliseconds.

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 in [19]. This particular implementation begins with a "start step" broadcast, sends object lists to voxels, 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 collision.

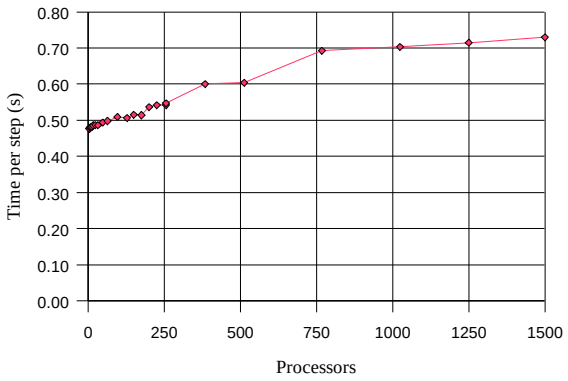

Figure 10. Time per collision for a scaling collision problem

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 program's run time would be dominated by messaging (dozens of microseconds) and runtime 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. 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

The array support 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, with minor modifications can take advantage of automatic load balancing. As described in [6], a large multiphysics solid rocket simulation code has been run on AMPI with minimal effort and excellent performance.
- The Charm++ finite−element method framework[20] 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, attempt to optimistically 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 [18]. 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.

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 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; more work in optimizing communication via message aggregation; and more optimization of the collective operations.

9 REFERENCES

[1] L.V. Kalé and S. Krishnan. "Charm++: Parallel Programming with Message−Driven Objects", Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175−213. MIT Press, 1996

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

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

[4] L.V. Kalé and others. *Charm++ Programmer's Manual*, http://charm.cs.uiuc.edu/, 2000

[5] 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, Sept. 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, Sept. 2000

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

[8] F. Bodin, P. Beckman, D. Gannon, and others. "Distributed pC++: Basic Ideas for an Object Parallel Language", *Scientific Programming*, Volume 2/Number 3 Fall 1993

[9] A. Chien and W. Dally. "Concurrent Aggregates" , *Proceedings of the Second ACM SIGPLAN*, March 1990, Seattle, WA

[10] H. Bal, R. Bhoedjang, R. Hofman, and others. *Orca: a Portable User−Level Shared Object System*, Technical Report IR−408, Vrije Universiteit, Amsterdam, June 1996

[11] M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn and J. Watts. "Interprocessor Collective Communication Library," *Supercomputing 1994*, Nov. 1994

[12] J. Nieplocha, RJ Harrison, and RJ Littlefield. "Global Arrays: A nonuniform memory access programming model for high−performance computers", *The Journal of Supercomputing*, 10:197−220, 1996

[13] S. Atlas, S. Banerjee, J. C. Cummings, and others (presented by J. Reynders). "POOMA: A high performance distributed simulation environment for scientific applications," *Supercomputing 1995*, Nov. 1995

[14] 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

[15] J. Piquer. "Indirect distributed garbage collection: handling object migration", *ACM Trans. Program. Lang. Syst.* 18, 5, pp 615 − 647, Sep. 1996

[16] 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/

[17] B. Hendrickson, R. Leland. "Chaco−− A Multilevel Algorithm for Partitioning Graphs," *Supercomputing 1995*, Nov. 1995

[18] 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

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

[20] M. Bhandarkar, L.V. Kalé. "A Parallel Framework for Explicit FEM", *Proceedings of the International Conference on High Performance Computing (HiPC 2000)*, Springer−Verlag, Dec. 2000.

10 SIMPLE EXAMPLE

This simple example program implements a stencil−on−regular−grid computation, such as a Jacobi relaxation or SOR. The numerical details are hidden in the Serial class, which represents a compact area of the problem domain. Each Serial is contained in a corresponding Jacobi array element, which does all the parallel communication required for border sharing and an occasional global error reduction.

10.1 Interface (ci) File

```
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 dir,int bs,double border[bs]);
    };
};
```

10.2 Serial Interface File (C++)

```
#define STEPS_PER_ITER 16
typedef enum { UP=0,DOWN=1,LEFT=2,RIGHT=3 } Dir;
class Serial {
    int stepCount;
    int height, width;
    double *temperatures;
public:
    void init(int x,int y);
    void pup(PUP::er &p);
    int getStep(void) const;
    int borderSize(Dir to) const;
    void extractBorder(double *a,Dir dir) const;
    void insertBorder(const double *d,Dir dir);
    void compute(void);
    double getLocalError(void) const;
};
```

10.3 Parallel Implementation File (C++)

```
#include "jacobi.decl.h"  /*Get generated code for proxy*/
#include "jacobi.def.h"
#include "util.h"   /*Get Serial and Dir types*/

int Xsize,Ysize; // Dimentions of object grid
CProxy_Jacobi Jproxy; // Main parallel array

/*This routine is called on when a global error reduction completes*/
void iterationDone(void *userParam,int dataLen,void *data)
{
    static int iterationCount=0;
    double totalError=*(double *)data;
    CkPrintf("Total error=%g at %d\n",
        totalError,++iterationCount);
    if (totalError<1.0e−3)
        {CkPrintf("Converged.\n");CkExit();}
    if (iterationCount>=50)
        {CkPrintf("Inf. loop!\n");CkExit();}
    Jproxy.sendBorders();//Broadcast send, to begin the next iteration
}
//Main object begins the computation
class Main : public Chare {
public:
    Main(CkArgMsg *m)
    {
        if (m−>argc!=3)
          {CkPrintf("Use jacobi x y\n");CkExit();}
        Xsize = atoi(m−>argv[1]); Ysize = atoi(m−>argv[2]);
        delete m;
        //Build the array and populate the element grid
        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);
    }
};
```

```
class Jacobi : public ArrayElement2D {
    int Xnbor(Dir in) { //Return x index of neighbor in given direction
        switch(in) {
        case up: case down: return thisIndex.x;
        case right: return (thisIndex.x+1)%Xsize;
        case left:  return (thisIndex.x-1+Xsize)%Xsize;
        }
    }
    int Ynbor(Dir in) { //Return y index of neighbor in given direction
        switch(in) {
        case right: case left: return thisIndex.y;
        case down: return (thisIndex.y+1)%Ysize;
        case up:  return (thisIndex.y-1+Ysize)%Ysize;
        }
    }
    enum {Nnbor=4};//Number of neighbors

    Serial serial; //The serial algorithm
    int borderCount; //Number of borders we've received so far
public:
    Jacobi(void)
    {
        serial.init(thisIndex.x,thisIndex.y);
        borderCount=0;
    }

    void sendBorders(void)
    {//Send border values to neighbors
        Dir d[Nnbor]={UP,DOWN,LEFT,RIGHT};
        for (int i=0;i<Nnbor;i++)
        {//Pack up and send our border to our i'th neighbor
            int bs=serial.borderSize(d[i]);
            double *border=new double[bs];
            serial.extractBorder(d[i],border);
            Jproxy(Xnbor(d[i]),Ynbor(d[i])).recvBorder(d[i],bs,border);
            delete[] border;
        }
    }
    void recvBorder(Dir forDir,int bs,double *border)
    {//Add this border patch
        serial.insertBorder(forDir,border);
        borderCount++;
        if (borderCount==Nnbor)
            bordersDone();
    }
    void bordersDone(void)
    { //We now have all the border patches-- do a serial timestep
        serial.compute();
        borderCount=0;
        if ((serial.getStep()%STEPS_PER_ITER)==0)
        {//Sum up local errors via a reduction
            double localErr=serial.getLocalError();
            contribute(sizeof(localErr),&localErr,
               CkReduction::sum_double);
        }
        else //Just start next step
            sendBorders();
    }

    //To support migration (& thus load balancing)
    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);
    }
};
```