A Parallel Array Abstraction for Data-Driven Objects*

Sanjeev Krishnan and Laxmikant V. Kalé
Dept of Computer Science, University of Illinois, Urbana, IL 61801
E-mail: {sanjeev,kale}@cs.uiuc.edu

Abstract

We describe design and implementation of an abstraction for parallel arrays of data-driven objects. The arrays may be multi-dimensional, and the number of elements in an array is independent of the number of processors. The elements are mapped to processors by a user-controllable mapping function. The mapping may be changed during the parallel computation, which facilitates load balancing, and communication optimization, for example. Asynchronous method invocation is supported, with multicast, broadcast, and dimension-wide broadcast. The abstraction is illustrated using examples in fluid dynamics and molecular simulations.

1 Introduction

Parallel computing, by its very nature, often involves identical or similar computations being repeated on different pieces of data. In object oriented methodology, the data is encapsulated in objects. Therefore, an array of objects that are distributed across processors is likely to be a useful abstraction. An extreme case of object-based parallelism occurs when almost identical operations are applied to all array elements in a synchronous — lock-step — manner; this case is well supported by data parallel languages such as HPF [6]. However, many applications require going beyond such lock-step mode. The progression through the life-cycle for each element object may be different, depending on the needs of the application, although all the elements object of an array may share the same structure of data and sets of methods.

This paper describes a parallel array-object abstraction that supports such applications. It supports asynchronous method invocation via message driven execution, which leads to efficient programs that can adaptively tolerate communication latencies. It has been implemented as an extension to the Charm++ [8] language. Charm++ supports message driven objects, and provides several useful features for parallel programming. In addition to enhancing the utility of Charm++ , the new abstraction can be used directly in C++ based parallel programs, using the Converse [7] interoperability framework.

2 Description

A parallel array is a group of objects (the array elements) with a common global name (id), which are organized in a multidimensional, distributed array, with each array element identified by its coordinates. The mapping of array elements to processors is specified by a user-provided mapping function. A default mapping is also provided for cases when the mapping is not significant.

2.1 Parallel Array Definition

A parallel array is defined as a normal parallel object (*chare*) class in Charm++, except that it must inherit from the system-defined base class *array*. This base class provides the following data fields :

- thishandle : this gives the unique handle (global pointer) of the array element.

*This research was supported in part by the NASA grant NAG 2-897 and NSF grant ASC-93-18159.

- thisgroup : this gives the global id by which the whole array is known.
- thisi, thisj, thisk : these give the coordinates of the array element[1].

Messages that are sent between array elements must inherit from the system-defined message class *arraymsg*. The following code gives an example of an array definition.

```
message class MessageType : public arraymsg {
        // list of data fields to be sent
} ;

chare class MyArray : public array {
        // list of private and public data and function members
    entry:
        // list of "entry functions" where messages are received
        MyArray(MessageType *m) ;  // constructor
        void EntryFunction(MessageType *m) ;
} ;
```

2.2 Parallel Array Creation

A parallel array is created using the operator *newgroup*, which has the following syntax :

```
MapFunctionType mymapfn ;
MessageType *msgptr ;
MyArray  group arrayid1 =  newgroup MyArray[XSize][Ysize](msgptr) ;
MyArray  group arrayid2 =  newgroup (mymapfn) MyArray[XSize][Ysize](msgptr) ;
```

The code above creates two-dimensional parallel arrays. The `newgroup` operator causes all the array element objects to be created (and their constructors invoked) on their respective processors. The parameter `msgptr` is sent to all processors as the parameter to the constructor for each array element. The first array above uses the default mapping function. The second array has a user-specified mapping function `mymapfn`, which takes the coordinates of an element as input and returns the processor where the element is located. `newgroup` is a non-blocking operator that immediately returns the id of the newly created array, which has the type `MyArray group`, and is analogous to a global pointer to an array. Because of its non-blocking nature, the elements of an array might not have been created when *newgroup* returns the array id. If necessary, the programmer may explicitly synchronize after initialization of all array elements on all processors by using a suitable reduction or synchronization operation. Currently, parallel arrays may be created only from processor 0.

2.3 Asynchronous messaging: remote method invocation

The parallel array library provides both point-to-point as well as multicast messaging. All messaging is asynchronous (no reply value is allowed), in keeping with the non-blocking communication paradigm of Charm++. If a reply is desired, the receiving object must send a reply message back to the sender object.

The syntax for point-to-point asynchronous messaging is :

```
arrayid[i][j]=>EntryFunction(msgptr) ;
```

where arrayid is the "global pointer" to the parallel array, i, j are the coordinates of the recipient array element, `EntryFunction` is the function to be invoked in the receiving object, and `msgptr` is the message to be sent across, which is passed as the sole parameter to the function.

The syntax for multicast asynchronous messaging is :

[1] Currently, only 1, 2, or 3-dimensional arrays are supported, although this can be easily extended to higher dimensions. For brevity, all the examples in this section assume a 2-dimensional array.

```
arrayid[i1..i2][j1..j2]=>EntryFunction(msgptr) ; // multicast to sub-array
arrayid[ALL][j]=>EntryFunction(msgptr) ;         // multicast to column
arrayid[i][ALL]=>EntryFunction(msgptr) ;         // multicast to row
arrayid[ALL][ALL]=>EntryFunction(msgptr) ;       // multicast to whole array
```

If an array element is known to be on the local processor, its data members may be accessed as usual :

```
arrayid[i][j]->datamember ;
```

2.4 Remapping and migration

The parallel array library supports both synchronous remapping and asynchronous object migration. Synchronous remapping must be initiated from processor 0 as follows :

```
arrayid->remap((MapFunctionType)newmapfn, return_chare_handle, &(ReturnChareType::ReturnFunction)) ;
```

`newmapfn` is the new mapping function. All array elements will be moved from their original locations to their new locations as specified by the new mapping function. After all elements have been installed on their new locations, a message is sent to the function `ReturnFunction` in the chare object specified by `return_chare_handle`. This provides a synchronization point after remapping. The user program must ensure that no messages are sent to any elements of the array being re-mapped.

Sometimes such synchronization is impossible or inefficient. Asynchronous remapping or "migration" is activated by each array element independently, by calling the function `migrate((MapFunctionType)newmapfn)` on the array element to be moved. The `newmapfn` parameter specifies the new mapping function, which tells the run-time library the destination processor for the array element. The call results in only the specified object being moved to its destination processor. The run-time library will correctly forward messages directed to the migrating array element to its new location.

The actual steps performed by the runtime system while migrating an object are :

1. Before migrating an object, the runtime library calls a user-provided *pack* function on the object, which copies the object's data area into a contiguous message buffer. The programmer must provide a pack function for every object type that needs migration.
2. Send the message to the object's destination processor
3. Create the new object
4. Initialize the object's data area using the message buffer. This is done by another user-provided *unpack* function. Note: the pack and unpack functions are virtual functions defined in the base class `array`.
5. Forward messages directed to the object from the old processor to the new processor.

2.5 Implementation

The parallel array library is implemented on top of the Converse interoperable run-time framework [7]. The library can thus be used in conjunction with modules written in other programming systems such as PVM and MPI. Although the parallel array concepts we developed were implemented in the context of the Charm++ parallel object-oriented language, the essential features are language-independent. Currently we are in the process of modifying the Charm++ translator to translate the parallel array syntax into calls to C++ functions in the runtime library.

3 Example Applications

In this section we describe two examples of the utility of the parallel array library.
NAS SP benchmark:

The NAS Scalar Pentadiagonal (SP) benchmark [1] is a Computational Fluid Dynamics program which uses an Alternating Direction Implicit (ADI) method to solve a system of partial differential equations. The computational space is a large three-dimensional cube consisting of an array of grid points, and the computation involves several iterations. In each iteration there is a "sweep" successively along each of the three coordinate axes. Parallelizing this computation involves decomposing the three-dimensional array among processors. Two of the most common methods to achieve this are :

- The "transpose" method : the array is divided into slabs oriented along the X direction first. After the X-direction sweep completes a transpose operation is done to orient the slabs along the Y-direction, in preparation for the Y-sweep. Thus there are a total of three transpose operations needed per iteration. The advantage of this method is that computations within the sweep are hence completely local to a processor. However, the transpose operation between sweeps can result in significant overhead.
- The multi-partition method [9, 4] : the array is divided into cubes, and each cube is assigned to a processor such that no two cubes with the same X, Y or Z coordinates are assigned to the same processor. This ensures that processor loads are balanced during all sweeps, and also that no transpose operations are needed.

Our objective in using the parallel array library for the SP benchmark was to develop an implementation that could be used to easily experiment with different decomposition strategies. This was done by representing the computational space as a parallel array of cubical sub-spaces. Each cube is a parallel object, which communicates with other cubes by sending/receiving messages. The different decomposition/mapping strategies are expressed by simply specifying a different mapping function for the parallel array. E.g. for the transpose method, all adjacent cubes along the direction of the sweep are mapped to the same processor; the transpose is effected by simply doing a remap operation on the parallel array. Thus we have a very flexible, elegant code which allows us to concentrate on experiments with the application, instead of getting involved in the details of implementing the decomposition.

The asynchronous migration facility provided with parallel arrays allows us to further optimize the transpose method by overlapping communication and computation. Each cube object migrates itself as soon as it has completed its work along one sweep. Thus the communication overhead of transferring its data to another processor is overlapped with the computation performed by other cubes. This overlap gives significant performance advantages over the traditional loosely-synchronous (separate phases of computation and communication) implementations.

Parallel molecular dynamics:

As another example, consider the parallel molecular dynamics simulation program NAMD [2]. The program consists of a three-dimensional computational space which is divided into cubical "cells" or "patches". The patches are distributed over processors by a sophisticated algorithm which maintains load balance as well as reduces communication. Patches need to exchange data (such as atomic parameters and forces) with each other. Each patch is an element in a three-dimensional parallel array. The parallel array construct can be used to provide a common global name using which patches can address each other. Further, the array mapping function can handle irregular mappings generated by the distribution strategy, by maintaining a global array of mappings. The dynamic remapping facility provided with parallel arrays is particularly useful for moving objects in order to balance load periodically.

4 Previous work

Parallel arrays have been used in various forms in several parallel programming systems. In data-parallel languages such as HPF [6] programmers have a shared-memory model in which arrays in the program are distributed over processors using compiler directives, and computations are assigned to processors using the "owner-computes" rule. The advantages of our parallel array abstraction are :

- in contrast to the lock-step operations on array elements in HPF, our array elements may proceed independently of each other executing different methods as necessitated by the application.
- the ability to increase performance through asynchronous operations (e.g. method invocation and object migration) which overlap communication and computation.

- more flexibility in specifying mappings of parallel arrays (e.g. the multi-partition mapping scheme we used in the NAS SP benchmark cannot be specified using HPF's compiler directives).

Some parallel object-oriented languages provide constructs similar to parallel arrays. Concurrent Aggregates [5] allows the programmer to create an aggregate of objects which has a common global name. Other objects interact with this aggregate as a whole, instead of individual objects. *Collections* in pC++ [3] provides parallel arrays with an HPF-like loosely-synchronous object-parallel model. Branched chares [8] are a related construct provided in Charm++, which are essentially parallel arrays with exactly one element per processor.

References

[1] D. Bailey et al. The NAS Parallel Benchmarks. *Intl. Journal of Supercomputer Applications*, 5(3), 1996.

[2] J. Board, L. V. Kale, K. Schulten, R. Skeel, and T. Schlick. Modeling biomolecules: Larger scales, longer durations. *IEEE Computational Science and Engineering*, 1(4), 1994.

[3] F. Bodin, P. Beckman, D. Gannon, and S. Narayana, S. an d Yang. Distributed pC++: Basic Ideas for an Object Parallel Langua ge. *Scientific Programming*, 2(3), 1993.

[4] J. Bruno and P. Capello. Implementing the Beam and Warming method on the hypercube. In *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, Jan. 1988.

[5] A. Chien. *Concurrent Aggregates*. MIT Press, 1993.

[6] High Performance Fortran Forum. *High Performance Fortran Language Specification (Draft)*, 1.0 edition, January 1993.

[7] L. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[8] L. Kale and S. Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993.

[9] N. H. Naik, V. K. Naik, and M. Nicoules. Parallelization of a class of implicit finite difference schemes in computational fluid dynamics. *International Journal of High Speed Computing*, 5(1), 1993.