Interfacing Global Arrays and ARMCI with the PCRC library, Adlib

Bryan Carpenter and Jarek Nieplocha

CRPC-TR99805
April 1999

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted November 1999

Interfacing Global Arrays and ARMCI with the PCRC library, Adlib*

Bryan Carpenter
Northeast Parallel Architectures Center
Syracuse University
Syracuse, NY 13422-410
dbc@npac.syr.edu

Jarek Nieplocha
Pacific Northwest National Laboratory,
Richland, WA 99352
j_nieplocha@pnl.gov

April 1, 1999

Abstract

This document reports work undertaken at NPAC, Syracuse under the DOE Global Array Extension Project. This work was intended to investigate the feasibility of interfacing, and perhaps eventually integrating, GA, ARMCI and the Parallel Compiler Runtime Consortium library, Adlib. In particular, we have reimplented parts of the Adlib library in terms of ARMCI, and also produced a version of GA which internally uses an Adlib-compatible array descriptor.

*Prepared for Battelle/Pacific Northwest National Laboratory under contract 313130-A9E.

Contents

1 Introduction

1.1 Context

The Department of Energy Advanced Computational Testing and Simulation (ACTS) project combines multiple programming tools and environments to support application codes addressing complex multidisciplinary problems. The Global Array (GA) library developed at the Pacific Northwest National Laboratory (PNNL) is one of the ACTS components. In ACTS the GA tool kit will be extended to provide support for higher dimensional arrays and additional distribution formats. The Adlib [4] parallel run-time library has capabilities that can be used to implement these new features.

Development of the Adlib library was completed at NPAC, Syracuse in the framework of the three-year Parallel Compiler Runtime Consortium (PCRC) project. It incorporates experience gained over the years in parallel distributed array technology. PNNL determined that it would be cost effective to transfer some of the multidimensional array capabilities from Adlib to Global Arrays, rather than to develop them from scratch at PNNL.

PNNL proposed to build an implementation of the GA tool kit Application Programming Interface (API) on top of a version of the PCRC library, augmented with the ARMCI aggregate remote memory copy library. ARMCI is being developed by PNNL. Towards this goal an initial study phase in fiscal year 1998 (reported on here) aimed to establish the basic feasibilty of this approach. We had to provide evidence that a new layered implementation would not lead to an unacceptable degradation of performance. For example, it had to be demonstrated that the global-to-local subscript conversion operations supported by the Adlib array descriptor (suitably optimized if necessary) would not compromise the low latency of the current `ga_get/put` operations. It was also considered that it would be very useful to reimplement part of the Adlib collective communication library on top of ARMCI, and compare the remote-memory-access approach with the current message passing implementation.

1.2 Summary of achievements

In the pilot study parts of the Adlib library were reimplemented on top of the ARMCI library. Early benchmarks show that the new implementation is significantly faster than the original MPI implementation on shared memory platforms. In a second part of the study, a version of Global Arrays was created that internally uses the Adlib distributed array descriptor to parametrize the decomposition of multidimensional global arrays, and uses associated functions from Adlib to perform address translations. The modified GA was benchmarked. The results indicate that the overhead of introducing the object-oriented DAD of Adlib into the GA implementation are small—probably less than a microsecond for a `ga_get` operation.

1.3 This report

Section 2 reviews relevant aspects of the Adlib runtime library. For completeness it also briefly reviews the Global Arrays toolkit and ARMCI. Section 3 describes work on using ARMCI in the implementation of Adlib. In particular it describes how various existing Adlib communication schedules were reimplemented in terms of ARMCI, and gives benchmark results. It also discusses how ARMCI will allow the Adlib API to be extended with functions for one-sided-communication. Section 4 describes the pilot work on using Adlib-based techniques in the implementation of GA.

2 Background

2.1 The Adlib library

The Adlib library was completed in the Parallel Compiler Runtime Consortium project [14] project. It is a high-level runtime library designed to support translation of data-parallel languages [4]. Initial emphasis was on High Performance Fortran (HPF), and two experimental HPF translators used the library to manage their communications [16, 10]. Currently the library is being used in the *HPspmd* project at NPAC [3]. It incorporates a built-in representation of a distributed array, and a library of communication and arithmetic operations acting on these arrays. The array model is more general than GA, supporting general HPF-like distribution formats, and arbitrary regular sections. The existing Adlib communication library emphasizes collective communication rather than one-sided communication.

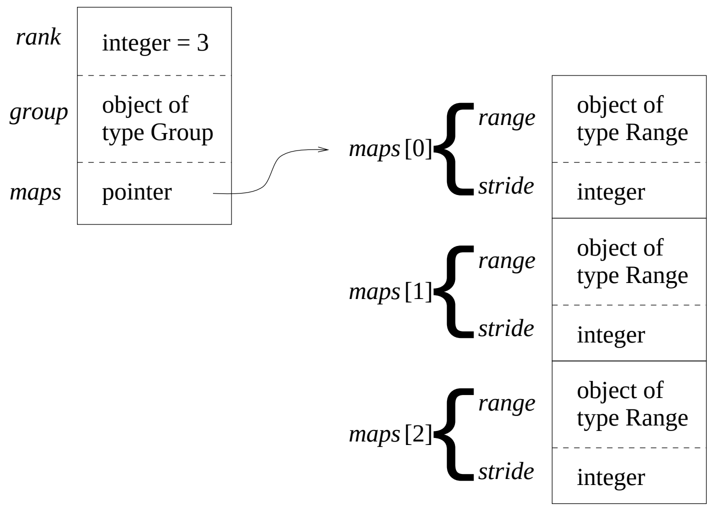

Figure 1: Structure of the Adlib DAD for a 3-dimensional array

Array descriptor

An important part of the Adlib library is its characteristic *Distributed Array Descriptor* (DAD). Adlib is implemented in C++, and the underlying array descriptor is implemented as an object of type `DAD`. A DAD object describes how elements of a particular array are distributed across available processors. It has a simple structure (Figure 1) containing three fields: an integer value (r, say) defining the *rank* (dimensionality) of the array, a process *group* object, and a vector of r *map* objects, one for each dimension of the array. The group defines a multidimensional process grid embedded in the set of processes executing the program, or some subset. Each map object consists of an integer local *memory stride* and a *range* object.

A range object describes the *extent* (size) and distribution format of one dimension of a distributed array. Adlib defines a class hierarchy of different kinds of range object (Figure 2). Each subclass represents a different kind of distribution format for an array dimension. The simplest distribution format is *collapsed* (sequential) format in which the whole of the array dimension is mapped to the local process. Other distribution formats (motivated by High Performance Fortran) include *regular block* decomposition, *simple cyclic* decomposition, and *block cyclic* decomposition. In these cases the index range (thus array dimension) is distributed over one of the dimensions of the process

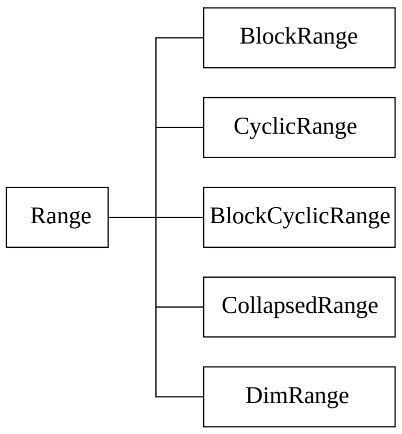

Figure 2: The Adlib `Range` hierarchy

grid defined by the group object in the DAD. All ranges in a specific DAD must be distributed over different dimensions of this grid, and if a particular dimension of the grid is targetted by none of the ranges, the array is said to be *replicated* in that dimension[1]. Some of the range classes allow *ghost extensions* to support stencil-based computations.

The factorization of the DAD into separate map and range objects for each dimension is convenient for several reasons. For example it simplifies construction of Fortran-style *regular sections* of arrays. It means that arrays with common distribution patterns for selected dimensions can share the same range objects for those dimensions. Also, the range object is a natural locus for the methods used to enumerate locally held index blocks for a distributed array dimension. These methods are used in translation of *forall* loops and similar data parallel constructs, and inside collective communication libraries.

The DAD itself does not contain the address of the memory segment where locally held elements are stored. Nor does it contain any information about the type of those elements. This information must be maintained by the user of the kernel library or (more often) in higher-level interface code wrapped around the kernel library calls. This arrangement allows the kernel DAD to be independent of details of the user-level programming language

[1]So there is no direct relation between the array rank and the dimension of the process grid: collapsed ranges means the array rank can be higher; replication allows it to be lower.

and the memory managment scheme used to allocate local array elements. As an example of the kind of interface that is possible, there is a C++ user-level interface to Adlib called *ad++*. This defines a series of template classes

```
template<class T>
  class Array1 ;

template<class T>
  class Array2 ;

template<class T>
  class Array3 ;

...
```

These implement distributed container classes for elements of type `T` (arrays of rank 1, 2, 3, ...). They are derived from the base class `DAD`, but add a field pointing to vectors of local elements; their constructors allocate these vectors.

Communication schedules

The Adlib communication library supports *collective operations* on distributed arrays. A call to a collective operation must be invoked simultaneously by all members of some *active process group*, which may or may not be the entire set of processes executing the program.

Communication patterns supported include HPF/F90 intrinsics such as CSHIFT and TRANSPOSE. More importantly they include the regular-section copy operation, `remap`, which copies elements between shape-conforming array sections regardless of source and destination mapping. Another function, `writeHalo`, updates ghost areas of a distributed array. Various collective gather and scatter operations allow irregular patterns of access. The library also provides essentially all F90 arithmetic transformational functions on distributed arrays and various additional HPF library functions.

All collective operations in the library are based on communication *schedule* objects. Each kind of operation has an associated class of schedules. Particular instances of these schedules, involving particular data arrays and other parameters, are created by the class constructors. Executing a schedule initiates the communications required to effect the operation. A single schedule may be executed many times, repeating the same communication pattern.

In this way, especially for iterative programs, the cost of computations and negotiations involved in constructing a schedule can often be amortized over many executions. This paradigm was pioneered in the CHAOS/PARTI libraries [5]. If a communication pattern is to be executed only once, simple wrapper functions can be made available to construct a schedule, execute it, then destroy it. The overhead of creating the schedule is essentially unavoidable, because even in the single-use case individual data movements generally have to be sorted and aggregated, for efficiency. The associated data structures are just those associated with schedule construction.

A characteristic example from the standard Adlib communication library is the remap schedule. The associated class has public interface:

```
class Remap {
public :
  Remap(const DAD* dst, const DAD* src, const int len) ;

  void execute(void* dstDat, void* srcDat) ;
} ;
```

The constructor is passed the DADs for source and destination arrays (or sections) and the size of the array element in bytes. The `execute` member is passed the base addresses for the locally held elements (the destructor for the schedule is not displayed here). The ad++ wrapper function for single use of this schedule with two dimensional arrays is

```
template<class T>
inline void remap(const Section2<T>& dst, const Section2<T>& src) {
  Remap schedule(&dst, &src, sizeof(T)) ;

  schedule.execute(dst.dat(), src.dat()) ;
}
```

The inquiry `dat` is a member of the ad++ array classes which returns a pointer to the local data vector. The *remap* function copies all elements of the source array (or section) to the destination array (or section)[2].

Top-level schedules such as `Remap`, which deal explicitly with distributed arrays, are implemented in terms of some lower-level schedules that simply operate on blocks and words of data. These lower-level schedules do not engage in operations on the DAD or its range and group components. The

[2]The `Section`R`<T>` classes are base classes of the `Array`R`<T>` classes.

Table 1: Low-level Adlib schedules

	operations on "words"	operations on "blocks"
Point-to-point	`MessSchedule`	`BlockMessSchedule`
Remote access	`DataSchedule`	`BlockDataSchedule`
Tree operations	`TreeSchedule`	`BlockTreeSchedule`
	`RedxSchedule`	`BlockRedxSchedule`
	`Redx2Schedule`	`BlockRedx2Schedule`

lower level schedules are tabulated in Table 1. Here "words" are contiguous memory blocks of constant (for a given schedule instance) size. "Blocks" are multidimensional (r-dimensional) local array sections, parametrized by a vector or r extents and a vector of r memory strides. The point-to-point schedules are used to implement collective operations that are deterministic in the sense that both sender and receiver have a priori knowledge of all required communications. Hence `Remap` and other regular communications such as `Shift` are implemented on top of `BlockMessSchedule`. The "remote access" schedules are used to implement operations where one side must inform the other end that a communication is needed. These negotiations occur at schedule-construction time. Irregular communication operations such as collective `Gather` and `Scatter` are implemented on these schedules. The tree schedules are used for various sorts of broadcast, multicast, synchronization, and reduction.

The original Adlib is implemented on top of MPI. All MPI communications are isolated in the low-level schedules. Ideally, one might hope that retargetting suitable parts of Adlib to operate on top of ARMCI would be simply a matter of reimplementing the low-level schedules. In practise, as explained in section 3, the layered implementation eased the task of retargeting to ARMCI, but the API of the low-level schedules had to be changed.

2.2 ARMCI and Global Arrays

ARMCI (Aggregate Remote Memory Copy Interface) is a new portable remote memory copy library developed at PNNL for optimized communication in the context of distributed arrays. ARMCI aims to be fully portable and compatible with message-passing libraries such as MPI or PVM. Unlike most existing similar facilities, such as Cray SHMEM [1] or IBM LAPI

[9], it focuses on the noncontiguous data transfers. ARMCI offers a simpler and lower-level model of one-sided communication than MPI-2 [12] (no epochs, windows, datatypes, Fortran-77 API, complicated progress rules, etc.) and targets a different user audience. In particular, ARMCI is meant to be used primarily by library implementors rather than application developers. Examples of libraries that ARMCI is aimed at include Global Arrays, P++/Overture [15, 2], and Adlib.

ARMCI supports three classes of operation

- data transfer operations including put, get and accumulate.
- synchronization operations—local and global fence and atomic read-modify-write, and
- utility operations for allocation and deallocation of memory (as a convenience to the user) and error handling.

The data transfer operations are available with two noncontiguous data format:

1. *Generalized I/O vector.* This is the most general format intended for multiple sets of equally-sized data segments, moved between arbitrary local and remote memory locations. It extends the format used in the UNIX *readv/writev* operations by minimizing storage requirements in cases when multiple data segments have the same size. The associated functions are `ARMCI_PutV` and `ARMCI_GetV`.

2. *Strided.* This format is an optimization of the generalized I/O vector format for sections of dense multidimensional arrays. Instead of including addresses for all the segments, it specifies for source and destination only the address of the first segment in the set. The addresses of the other segments are computing using the stride information. The associated functions, `ARMCI_PutS` and `ARMCI_GetS`, take short *vectors* of strides as arguments allowing *multidimensional* array segments to be handled in a single operation.

The Global Arrays (GA) [13] toolkit provides a shared-memory programming model in the context of 2-dimensional distributed arrays. GA has been the primary programming model for numerous applications, some as big as 500,000 lines of code, in quantum chemistry, molecular dynamics, financial calculations and other areas. The toolkit was originally implemented directly on top of system-specific communication mechanisms (NX *hrecv*, MPL *rcvn-call*, SHMEM, SGI *arena*, etc.). The original one-sided communication engine of GA had been closely tailored to the two-dimensional array semantics supported by the toolkit. This specificity hampered extensibility of the GA implementation. It became clear that a separation of the communication layer from the distributed array infrastructure is a much better approach. This was accomplished by restructuring the GA toolkit to use ARMCI hence making the GA implementation itself fully platform independent. ARMCI matches the GA model well and allows GA to support arbitrary dimensional arrays efficiently.

3 Implementing Adlib Communication in terms of ARMCI

As explained in section 2.1, the kernel Adlib library is implemented directly on top of MPI. We anticipated that reimplementing parts of the collective communication library on top of ARMCI would lead to improved performance on shared memory platforms (at least). Also, this task exercises the ARMCI API in a new context and should help to prove it as a good interface.

There is a difference between the way the GA array descriptor is implemented and the way distributed arrays are represented in Adlib, arising from the different assumptions about communication: remote memory access versus collective communication. In Adlib the basic DAD need only be accompanied by a single pointer to the base address for the locally held elements. In Global Arrays there is locally-held *table of pointers* containing base addresses where the array segments are stored in every peer process.

To exploit ARMCI, an Adlib program must at some point set up this table of remote pointers. In principle (at least in non-shared memory implementations) this could be done either when the distributed array was created, or at the time the remote pointers are needed (at schedule construction or execution time). The advantage of the latter approach is that the old Adlib interface might be preserved—the user code could maintain just the local base address, and remote address tables could be regenerated on the fly. But regenerating the table clearly involves some rendezvous between the peers, which is likely to involve a significant and repeated overhead. Also it would imply some complication of the ARMCI interface, because currently the `ARMCI_Malloc` function is responsible for both allocating memory for array elements and distributing handles to peers. These two functions would have to be separated.

These considerations indicate that the table of remote pointers should be set up at array creation time (as in GA). As explained in section 2.1 the kernel DAD structure does not hold pointers to array data, so this requirement does not imply any change to the organization of that structure. It *does* imply some changes at the level of higher-level "container classes" (such as the ad++ `Array`R`<T>` classes) and—more importantly—in the interface to the schedule classes. The constructors (or `execute` members) of the schedules must be passed tables of pointers rather than individual pointers.

3.1 New classes vs reimplementation of existing ones

As noted above, the ARMCI implementation implies changes to the actual interfaces of kernel Adlib classes. We might consider changing the Adlib API in future releases, but this wasn't practical on the time scale of the current project. In any case it seems that there will continue to be a place for arrays that *do not* carry full remote table pointers. Creating a standard Adlib distributed array is a purely local operation and therefore very fast. In the data parallel style of programming it is quite common to create short-lived distributed arrays as temporaries; to properly support this paradigm we want creation of these arrays to be as fast as possible. Creating a GA array is itself a collective operation, and intrinsically slower.

In summary, we can distinguish two useful kinds of distributed array: the "local" distributed array of Adlib and the "global" distributed array of GA.

For the initial experiments we added a new series of container classes to ad++ that parallel the pre-existing "local" distributed arrays, but implement GA style "global" distributed arrays:

```
template<class T>
  class ArrayRM1 ;

template<class T>
  class ArrayRM2 ;

template<class T>
  class ArrayRM3 ;
```

```
      ...
```

(In fact these classes are derived from the corresponding classes in standard ad++, so ordinary Adlib operations that don't need remote pointer tables are still applicable to them.)

Similarly for communication schedules, we have implemented a subset that parallels standard Adlib ones, with different APIs. For example

```
class RemapRM {
public :
  RemapRM(const DAD* dst, void* dstDat,
          const DAD* src, const int srcOff, void** srcPtrs,
          const int len) ;

  void execute() ;
} ;
```

For further discussion see section 3.3.

In the longer term we may want to consider a more uniform treatment of the two kinds of distributed array. One possibility is to elevate the `group` field of the DAD to include more general contextual information, in a manner reminiscent of the way MPI decorates its groups to form communicators. This contextual information might include information about whether the array supports remote memory operations, the associated extra tables, and perhaps other flags specifying information such as whether an array is held in-core or out-of-core.

3.2 ARMCI-compatible Adlib arrays

One other minor problem was encountered in the construction of Adlib-like arrays supporting ARMCI-based communication.

In the Adlib model it is supposed to be possible for *any* group of processes to collectively create a distributed array. The elements are divided across processes in this group, which may or may not include all processes executing the program as a whole. Processes *outside* the group do not have to engage in the array creation at all—they may be executing some completely different code at the time. This feature of the library was originally introduced to support translation of a kind of nested parallelism that can occur in HPF programs when (for example) some intrinsically parallel Fortran 90 array expressions appear nested inside a *forall* construct. It has been used in practise to support "multiple data parallelism" [11], and also in work on a parallel database classification algorithm using a divide-and-conquer style of parallelism, combined with data parallelism [7].

The ARMCI functions `ARMCI_Malloc`, `ARMCI_free` allocate raw shared or remotely accessible memory. In the original definition they were collective across all processes on which the main program is initiated. To support the Adlib requirements we prototyped versions with interface

```
ARMCI_MPI_Malloc(MPI_Comm comm, void *ptr_arr[], int bytes) ;
ARMCI_MPI_Free(MPI_Comm comm, void *ptr) ;
```

These versions are collective only within the process group associated with the communicator argument. Similar functions will now be incorporated in the "official" ARMCI API.

To illustrate the how construction of an Adlib array is affected, we sketch the ad++ constructors for the original `Array2<T>` and the new `Array2RM<T>` class[3]. The original constructor is

```
Array2(const Range& x_0, const Range& x_1, const Group& p) :
                                 Section2<T>(p) {
  int size = 1 ;

  maps [0] = Map(x_0, size) ;
  size *= x_0.volume() ;
  maps [1] = Map(x_1, size) ;
  size *= x_1.volume() ;

  data = new T [size] ;
}
```

The total size of the local segment is accumulated in the `size` variable. The `volume` member of `Range` returns the number of locally held index values for each dimension. Intermediate values of `size` are memory strides associated with array dimensions and these are stored with the ranges in the `Map` objects of the DAD. Finally the space for the local elements is allocated. The destructor just deletes `data`.

The ARMCI-compatible class contains two extra fields, `ptrs`—the table of remote pointers—and `offset`—a constant offset to be added to these

[3]We re-emphasize that ad++ is just one possible, particularly simple, interface to kernel Adlib. It illustrates the issues.

pointers (it may become non-zero for certain array sections constructed from the original array). The constructor is

```
ArrayRM2(const Range& x_0, const Range& x_1, const Group& p) :
                                   SectionRM2<T>(p) {
  int size = 1 ;

  maps [0] = Map(x_0, size) ;
  size *= x_0.volume() ;
  maps [1] = Map(x_1, size) ;
  size *= x_1.volume() ;

  ptrs = new void* [AdlibNProcs()] ;
  malloc_rm(ptrs, &comm, sizeof(T) * size) ;
  data = (T*) ptrs [AdlibLocId()] ;

  offset = 0 ;
}
```

The difference is that `ptrs` must be allocated, and then there is a call to `malloc_rm`. This is a wrapper for `ARMCI_MPI_Malloc`. The `malloc_rm` call sets up an MPI communicator for the Adlib active process group and passes it to `ARMCI_MPI_Malloc`. Note that the `comm` argument is actually an *out* argument; the communicator is cached in the `ArrayRM2` record so that it can eventually be passed to a matching `free_rm` call in the `~ArrayRM2` destructor, and thus to `ARMCI_MPI_Free`. The destructor also deletes `ptrs`.

3.3 ARMCI-based collective communication schedules

A representative selection of the Adlib communication schedule classes were reimplemented in terms of ARMCI. These included the `Remap` class, which implements copying between regular sections of distributed arrays, the `Gather` class, which implements copying of a whole array from a source array indirectly subscripted by some other distributed arrays, and a few related classes.

As explained in section 3.1 the ARMCI-based schedules needed a different API to the standard Adlib schedules. Different arguments are needed for constructor and `execute` member. In the event we implemented a completely new set of low-level schedules based on ARMCI. They are listed in

Table 2: Low-level ARMCI-based schedules

	operations on "words"	operations on "blocks"
non-collective	`RMSchedule`	`BlockRMSchedule`
collective	`CollRMSchedule`	`CollBlockRMSchedule`

Table 2, which should be compared with Table 1. Compared with the MPI-based schedules these classes have a simple role (and implementation). They just store lists of individual data requests. The execution members simply dispatch the ARMCI operations indicated by these lists. The only difference between the collective and non-collective versions is that the execution members of the collective versions incorporate barrier synchronizations and, where appropriate, `ARMCI_AllFence` operations.

The `RMSchedule` and `CollRMSchedule` schedules use the ARMCI operations `ARMCI_GetV` or `ARMCI_PutV` in their execution members. The `BlockRM-Schedule` and `CollBlockRMSchedule` schedules generally use `ARMCI_GetS` or `ARMCI_PutS` (recall that the "blocks" recognized by low-level Adlib schedules are multidimensional local array sections, parametrized by vectors of extents and memory strides).

New top-level schedules are implemented in terms of these lower level ones. The interface of `RemapRM`, for example, was already displayed in section 3.1:

```
class RemapRM {
public :
  RemapRM(const DAD* dst, void* dstDat,
          const DAD* src, const int srcOff, void** srcPtrs,
          const int len) ;

  void execute() ;
} ;
```

Where the old remap operation was implemented on top of the point to point schedule `BlockMessSchedule`, the natural basis of `RemapRM` is `CollBlockRM-Schedule`. The detailed API and usage of `CollBlockRMSchedule` is rather different from `BlockMessSchedule`, so the old implementation of `Remap` in terms of `BlockMessSchedule` cannot easily be recycled. Luckily Adlib includes a generalization of `remap` called `vecGather`. The schedule of the latter is implemented on top of the old "remote access" schedule `BlockDataSched-`

ule, whose usage is quite similar to CollBlockRMSchedule. So in practise VecGather was retargetted to CollBlockRMSchedule to create an ARMCI-based schedule called VecGatherRM. The resulting code was simplified to create RemapRM.

A minor difference between the the new RemapRM schedule and the old Remap schedule (and conventional Adlib schedules in general) is that source and destination data pointers are passed to the *constructor* rather than the execute member. This is because the lists stored in the ARMCI-based schedules are essentially arguments for ARMCI functions, which include pointers. Conventionally Adlib stored only offsets (not absolute data addresses) in schedules. This is less convenient when targetting ARMCI[4].

As discussed earlier, the table of remote pointers for the source array must be passed in to the schedule. Only the local pointer for the destination array is needed. The argument srcOff is the offset mentioned in the previous section. This may be non-zero if the source is an array section.

3.4 Benchmarking the ARMCI-based schedules

The benchmark presented here is based on the *remap* operation. The particular example chosen abstracts the communication in the array assignment of the following HPF fragment

```
      real a(n, n), b(n, n)
!hpf$ distribute a(block, *) onto p
!hpf$ distribute b(*, block) onto p
      a = b
```

The destination array is distributed in its first dimension and the source array is distributed in its second dimension, so the assignment involves a data redistribution requiring an all-to-all communication. In practice the benchmark was coded directly in ad++ (rather than Fortran) and run on four processors of an SGI Power Challenge. The timings for old and new versions are given in Figure 3 and table 3. Generally the implementation of the operation on top of ARMCI is more than twice as fast as the MPI imple-

[4]A goal in developing Adlib is always that it should be as "lean" as possible. Of course we could have preserved more features of the old API. They were changed to avoid superfluous array allocation and copying inside the implementation.

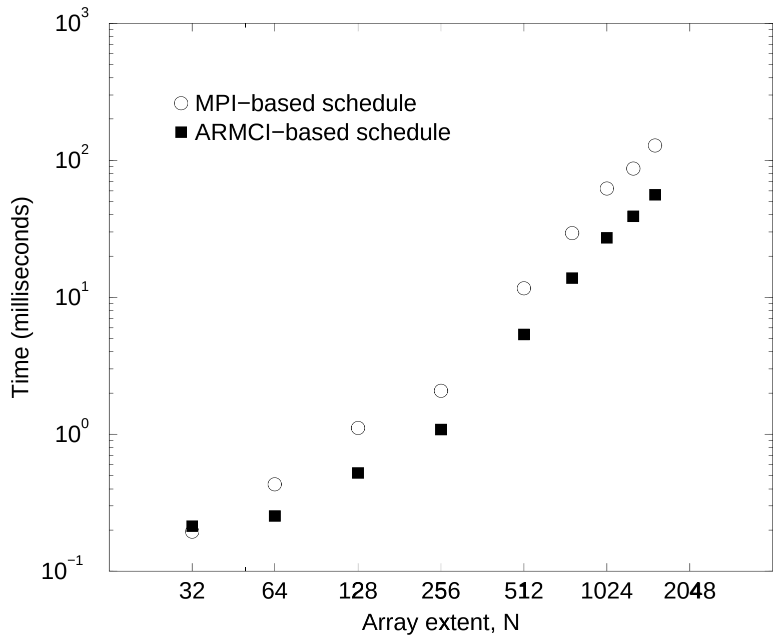

Figure 3: Timings for original MPI vs new ARMCI implementation of *remap*. The operation is a particular redistribution of an N by N array.

mentation[5]. The MPI implementation is MPICH using the shared memory device, so the underlying transport is the same in both cases.

3.5 One-sided communication schedules for Adlib

By replacing the `CollBlockRMSchedule` base class of `RemapRM` with the non-collective version `BlockRMSchedule` (which doesn't include barrier synchronizations), and stripping down the code so that the destination array is just a local sequential array rather than a distributed array, we produced a schedule called `Get`. This schedule supports one-sided "get" (also "put") operations functionally equivalent to the GA operations `ga_get` and `ga_put`.

The ad++ `get` functions based on the `Get` schedule have interfaces like:

```
template<class T>
```

[5]For small arrays the MPI version was actually slightly faster due to extra barrier synchronizations used in the initial ARMCI implementation of `RemapRM`. For now these synchronizations were implemented naively in terms of message-passing, and contribute around 190μs to the total times.

Table 3: Times in milliseconds plotted in Figure 3

Array extent	MPI-based	ARMCI-based
32	0.20	0.21
64	0.43	0.25
128	1.11	0.52
256	2.07	1.08
512	11.7	5.35
768	29.4	13.8
1024	62.4	27.2
1280	86.8	39.0
1536	127.8	56.1

```
  void get(const int str_0, const int str_1, T* base,
           const SectionRM2<T>& src) ;
```

The destination local array is parametrized by a base address and a set of local memory strides. The last argument is an arbitrary regular section of a global distributed array. In particular this can be an arbitrary rectangular patch, as for `ga_get`.

As a test case we transcribed the `jacobi.F` example in the GA release to C++ using ad++ `ArrayRM2` arrays and the `get` operation, and verified that it reproduced the same results as the original Fortran.

In the past the lack of this kind of one-sided communication has been a notable gap in the functionality of Adlib. ARMCI will allow us to close this gap in future releases.

4 Adapting Global Arrays to use the Adlib array descriptor

In an orthogonal but complementary development, we have produced a PCRC version of GA that internally uses a PCRC/Adlib array descriptor to maintain the global array distribution parameters. The modularity of the GA implementation, enhanced by the introduction of the ARMCI layer, made changing the internal array descriptor a relatively straightforward task. An immediate benefit is that direct calls to the optimized Adlib collective communication library will be possible from the modified GA.

4.1 Extensions to the DAD

The original Adlib array descriptor supported all distribution formats specified in HPF 1.0 [8]. These formats include simple block, simple cyclic, and block-cyclic distributions. It did not support the kind of irregular block distributions allowed in Global Arrays. Version 2.0 of HPF [6] specified equivalent block irregular distributions as an "approved extension", so provisions had been made in Adlib for this extension. But the extension had not been implemented at the time the current project started.

A new subclass of `Range` called `IrregRange` was added to the hierarchy of figure 2. The documented interface is

```
class IrregRange : public Range {
public :
  IrregRange(Dimension dim, int blocks []) ;
} ;
```

The constructor creates an irregular block-distributed range distributed over the process grid dimension `dim`. The array `blocks` has P elements where P is the extent of the grid dimension. These elements define the block size associated with each process. The extent of the range is

$$\sum_{i=0}^{P} \texttt{blocks [}i\texttt{]}$$

This interface for the constructor is a natural one for directly supporting the new `GEN_BLOCK` distribution format in HPF 2.0. We also included a second (undocumented) constructor that is slightly more convenient for implementing Global Arrays API:

```
IrregRange(const int extent, Dimension dim, Integer map []) ;
```

The definition of `map` agrees with the definition of the corresponding arguments of `GA_CREATE_IRREG`: the elements are the smallest global index held in each process.

Internally `IrregRange` uses an algorithm borrowed from the GA implementation for address translation. In particular the inline function `find-Block`, used in the global-to-local subscript conversion method called `locat-ion`, is an adapted version of the corresponding GA macro.

There remains one significant difference between the layout of elements in a GA array and the corresponding Adlib array, even using `IrregRange`.

In Adlib the same volume of memory is allocated in every member of the process group specified for the array: this volume is the product of the *maximum* local block sizes from each range. GA allocates a volume equal to the product of the local block sizes (*not* the maximum), and this volume is usually different in each process. Adlib exploits the constancy of the volume, assuming that memory strides for remote array segments are the same as for the local segment. These strides are stored in the DAD. Rather than try to modify the DAD it was decided to live with this difference. Although it is quite likely that some Global Arrays programs rely on the assumption that the locally allocated array volume is identical to the number of locally-held global array elements, we will see later that we did not encounter problems with the example programs in the GA release package.

4.2 Modifications to Global Arrays

The original plan called for implementation of a small subset of Global Arrays, including `ga_create`, `ga_destroy`, `ga_get`, `ga_put` and `ga_acumulate`. Originally it was intended that these would be Fortran wrappers around calls to the `Get` schedule described in section 3.5.

To validate the subset, one or more test programs would be needed. Ideally these would be semi-realistic Global Arrays programs. All the examples in the GA release use around a dozen different GA operations. Unless some very minimal test code was invented, it seemed that the proposed "small subset" would have to be bigger than the original plan assumed. It became clear that it would be very useful to recycle code from the existing Global Arrays implementation to create the new interface. Most of the relevant routines are implemented in the original GA source file `global.core.c`. That file was about 3000 lines long. It was hoped that by cutting down the number of entry points to just those needed for the current study, and retargetting to Adlib with ARMCI/MPI, a much smaller file would be produced. Before that attempt got very far, PNNL provided a new implementation of core global arrays based on ARMCI. The corresponding file is called `global.armci.c`.

The plan now altered slightly. Rather than providing wrappers around the new Adlib `Get` schedule, we would modify the file `global.armci.c` to use the Adlib DAD, but leave direct calls to ARMCI in the file. The principle functions that would need modification were the ones listed in table 4. The structure `global_array_t` from GA would be retained, because it incorporates various fields used for management of arrays that have no counterpart

Table 4: Functions from `global.armci.c` requiring modification

`nga_create`	`nga_locate_`	`ga_scatter_acc_local`
`nga_create_irreg`	`nga_locate_region_`	`ga_scatter_`
`ga_create_irreg`		`ga_scatter_acc_`
`ga_duplicate`	`ga_fill_`	`ga_gather_local`
		`ga_gather_`
`nga_distribution_`	`nga_proc_topology_`	
`ga_access_`		`nga_read_inc_`
	`nga_put_`	
	`nga_get_`	
	`nga_acc_`	

in the Adlib DAD. But several fields—`dims`, `chunk`, `nblock`, `scale`, `mapc`, `lo`—would be deleted and replaced by a DAD.

Various kinds of changes were needed to `global.armci.c` and the associated header `global.armci.h`. A number of global changes relate to need to compile these files as C++ rather than C. Many of the functions had old-style C argument lists, and these had to be changed to ANSI-style. Function prototypes had to be added in various place, or the function definitions had to be reordered to avoid forward references without prototypes. In some places `extern "C"` clauses were needed. In some places explicit casts were needed to pass the more stringent type-checking of C++.

Various old functions and macros became irrelevant and were deleted, including `findblock`, `gam_GetRangeFromMap`, `gam_CountElems`, `gam_Compute-PatchIndex`, `gam_ComputeCount` and `gam_setstride`.

A few new classes and functions were added. A class similar to the `Get` schedule, but specialized to reproduce exactly the functionality of `nga_get`, etc, was defined (lists of block moves had to be constructed anyway, because of the way GA randomizes the order of of individual remote accesses). The class is called `PatchTransfers`.

Finally, functions in table 4 had to be modified to create and use the DAD fields instead of the deleted `global_array_t` fields. In `ga_create_irreg` a suitable Adlib process grid has to be constructed or found before setting up the DAD. The initialization and finalization functions had to initialize and finalize Adlib.

Various other header files included by `global.armci.c` also had to be modified to meet the strict prototyping requirements of C++. These included `globalp.h`, `global.h` and `ma/macdecls.h`.

While this work was in progress PNNL provided a beta version of the new n-dimensional GA. It was updated with the NPAC modifications, and our work continued from that basis.

Once the new version of the core GA file could be compiled, we had to address the problem of linking it. One option was to try to extract a suitable subset of necessary modules in the GA release (including, for example, the MA memory allocator). Selecting a suitable subset seemed more difficult than recompiling the whole GA release to use the new core module. So the make files were modified to compile just the core module with C++ and link it with the rest of the library.

After some debugging, all relevant examples in the `global/testing` directory— `test.x`, `jacobi.x`, `patch.x`, `perf.x`, `perfmod.x`, `perform.x`, `p.x`, `t.x`, `testeig.x`, `testsolve.x` and `ndim.x`—could be run successfully with the modified GA. A more detailed list of changes to the standard version of GA is available from NPAC.

4.3 Benchmarking the DAD-based GA primitives

Figures 4 and 5 and tables 5 and 6 give timing results for the modified version of `ga_get`, compared with the (new ARMCI-based) version of GA provided by PNNL. Timings were obtained using the program `global/testing/perf.F` from the GA release (once again on an SGI Power Challenge).

We conclude that using the object-oriented DAD of Adlib introduces a small overhead in the `ga_get` operation. This overhead appears to be generally less than a microsecond[6].

[6]The first naive implementation of GA using the DAD introduced a larger overhead (a few microseconds). Some optimizations were applied to reach the performance quoted here. However the current version is still not really aggressively optimized—the work stopped when overall performance was comparable with the original GA.

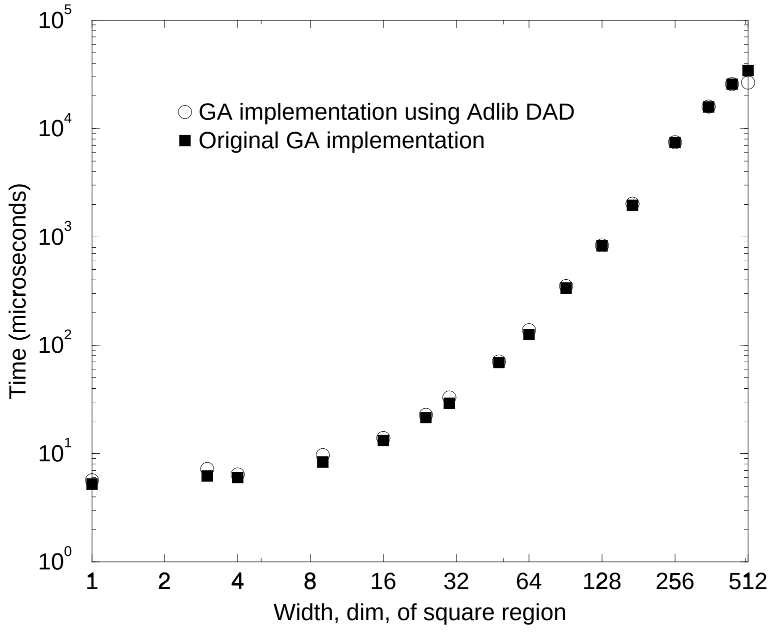

Figure 4: Timings for modified Global Arrays vs original. The operation is a *get* operation on a *dim* by *dim* patch of an array. Timings when patch is locally resident.

Table 5: Times in microseconds plotted in Figure 4

Patch width	Using DAD	Original GA
1	5.69	5.23
3	7.20	6.22
4	6.45	6.03
9	9.66	8.33
16	13.9	13.3
24	22.8	21.5
30	33.0	29.1
48	70.7	69.1
64	138	126
91	351	337
128	835	825
171	2021	1967
256	7513	7433
353	15930	15766
440	25780	25638
512	26623	34371

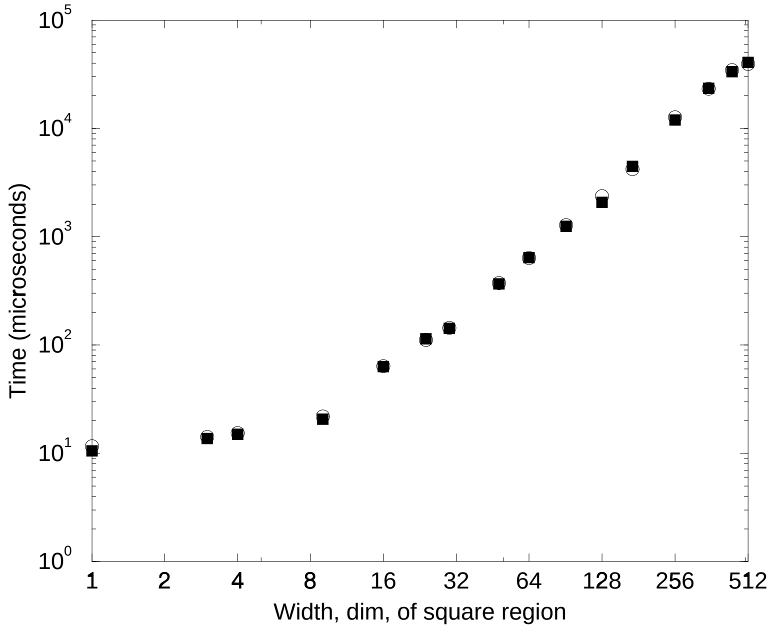

Figure 5: Timings as for Figure 4, but in case where patch is held remotely.

Table 6: Times in microseconds plotted in Figure 5

Patch width	Using DAD	Original GA
1	11.6	10.5
3	14.2	13.7
4	15.3	15.0
9	21.8	20.7
16	63.8	63.0
24	111	114
30	144	143
48	373	368
64	636	640
91	1276	1247
128	2370	2075
171	4215	4449
256	12936	11970
353	23314	23443
440	34540	33341
512	39244	40716

References

[1] R. Barriuso and Allan Knies. *SHMEM User's Guide. SN-2516.* Cray Research Inc, 1994.

[2] F. Bassetti, D. Brown, K. Davis, W. Henshaw, and D. Quinlan. OVERTURE: An object-oriented framework for high performance scientific computing. In *Proceedings of SC98: High Performance Networking and Computing*. IEEE Computer Society, 1998.

[3] B. Carpenter, G. Fox, D. Leskiw, X. Li, Y. Wen, and G. Zhang. Language bindings for a data-parallel runtime. In Michael Gerndt and Hermann Hellwagner, editors, *Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*. IEEE Computer Society Press, 1998.

[4] B. Carpenter, G. Zhang, and Y. Wen. NPAC PCRC runtime kernel definition. Technical Report CRPC-TR97726, Center for Research on Parallel Computation, 1997. http://www.npac.syr.edu/projects/pcrc/kernel.html.

[5] R. Das, M. Uysal, J.H. Salz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[6] High Performance Fortran Forum. High Performance Fortran language specification, version 2.0, January 1997. http://www.crpc.rice.edu/HPFF/hpf2.

[7] L. Fu, B. Carpenter, S. Ranka, and G. Fox. Parallelization of a hierarchical classifier using Adlib. In preparation.

[8] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, special issue, 2, 1993.

[9] IBM Corp. Understanding and using the communication Low-level Application Programming Interface (LAPI). In *IBM Parallel System Support Programs for AIX Administration Guide, GC23-3897-04*. 1997. (Available at http://ppdbooks.pok.ibm.com:80/cgi-bin/bookmgr/bookmgr.cmd/BOOKS/sspad230/9.1).

[10] J. Merlin, B. Carpenter, and A. Hey. shpf: a subset High Performance Fortran compilation system. *Fortran Journal*, pages 2–6, March 1996.

[11] J. H. Merlin, S. B. Baden, S. J. Fink, and B. M. Chapman. Multiple data parallelism with HPF and KeLP. In *High-Performance Computing and Networking: Proceedings of HPCN Europe 1998*, volume 1401 of *Lecture Notes in Computer Science*, pages 828–839. Springer, 1998.

[12] MPI Forum. MPI-2: Extension to message passing interface. Technical report, University of Tennessee, July 1997.

[13] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *Journal of Supercomputing*, 10:197–220, 1996.

[14] Parallel Compiler Runtime Consortium. Common runtime support for high-performance parallel languages. In *Supercomputing '93*. IEEE Computer Society Press, 1993.

[15] D. Quinlan and R. Parsons. A++/P++ array classes for architecture independent finite difference calculations. In *Proceedings of the Second Annual Object Oriented Numerics Conference*, April 1994.

[16] G. Zhang, B. Carpenter, G. Fox, X. Li, X. Li, and Y. Wen. PCRC-based HPF compilation. In *10th International Workshop on Languages and Compilers for Parallel Computing*, volume 1366 of *Lecture Notes in Computer Science*. Springer, 1997.