appeared in The Journal of Supercomputing, 10:169-189, 1996

Global Arrays: A Non-Uniform-Memory-Access Programming Model For High-Performance Computers

Jaroslaw Nieplocha, Robert J. Harrison and Richard J. Littlefield

Pacific Northwest National Laboratory‡, P.O. Box 999, Richland WA 99352

Abstract

Portability, efficiency, and ease of coding are all important considerations in choosing the programming model for a scalable parallel application. The message-passing programming model is widely used because of its portability, yet some applications are too complex to code in it while also trying to maintain a balanced computation load and avoid redundant computations. The shared-memory programming model simplifies coding, but it is not portable and often provides little control over interprocessor data transfer costs. This paper describes an approach, called Global Arrays (GA), that combines the better features of both other models, leading to both simple coding and efficient execution. The key concept of GA is that it provides a portable interface through which each process in a MIMD parallel program can asynchronously access logical blocks of physically distributed matrices, with no need for explicit cooperation by other processes. We have implemented GA libraries on a variety of computer systems, including the Intel DELTA and Paragon, the IBM SP-1 and SP-2 (all message-passers), the Kendall Square KSR-1/2, Convex SPP-1200 (nonuniform access shared-memory machines), the Cray T3D (a globally-addressable distributed-memory computer) and networks of Unix workstations. We discuss the design and implementation of these libraries, report their performance, illustrate the use of GA in the context of computational chemistry applications, and describe the use of a GA performance visualization tool.

Keywords

NUMA architecture, parallel-programming models, shared memory, parallel-programming environments, distributed arrays, global arrays, one-sided communication, scientific computing, Grand Challenges, computational chemistry

1 Introduction

This paper addresses the issue of how to program scalable scientific applications. Our interest in this issue has both long-term and short-term components. As participants in a U.S. Federal High Performance Computing and Communications Initiative (HPCCI) Grand Challenge Applications project, our long-term goal is to develop the algorithmic and software engineering techniques necessary to permit exploiting future teraflops machines for computational chemistry. At the same time, we and our colleagues at the Pacific Northwest National Laboratory (PNNL) are developing a suite of parallel chemistry application codes to be used in production mode for chemistry research at PNNL's Environmental Molecular Science Laboratory (EMSL) and elsewhere. The programming model and implementations described here have turned out to be useful for both purposes.

Two assumptions permeate our work. The first is that most high performance parallel computers currently and will continue to have physically distributed memories with Non-Uniform Memory Access (NUMA) timing characteristics, and will thus work best with application programs that have a high degree of locality in their memory reference patterns. The second assumption is that extra programming effort is and will continue to be required to construct such applications. Thus, a recurring theme in our work is to develop techniques and tools that allow applications with explicit control of locality to be developed with only a tolerable amount of extra effort.

‡. Pacific Northwest National Laboratory is a multiprogram national laboratory operated for the U.S. Department of Energy by Battelle Memorial Institute under contract DE-AC06-76RL0 1830.

There are significant tradeoffs between the important considerations of portability, efficiency, and ease of coding. The message-passing programming model is widely used because of its portability, yet some applications are too complex to code in it while also trying to maintain a balanced computation load and avoid redundant computations. The shared-memory programming model simplifies coding, but it is not portable and often provides little control over interprocessor data transfer costs. Other more recent parallel programming models, represented by languages and facilities such as HPF [1], SISAL [2], PCN [3], Fortran-M [4], Linda [5], and shared virtual-memory, address these problems in different ways and to varying degrees, but none of them represents an ideal solution.

In this paper, we describe an approach, called Global Arrays (GA), that combines the better features of message-passing and shared-memory, leading to both simple coding and efficient execution for a class of applications that appears to be fairly common. The key concept of GA is that it provides a portable interface through which each process in a MIMD parallel program can independently, asynchronously, and efficiently access logical blocks of physically distributed matrices, with no need for explicit cooperation by other processes. In this respect, it is similar to the shared-memory programming model. However, the GA model also acknowledges that remote data is slower to access than local, and it allows data locality to be explicitly specified and used. In these respects, it is similar to message passing.

Development of Global Arrays has been and continues to be motivated by applications, specifically computing the electronic structure of molecules and other small or crystalline chemical systems. These calculations are used to predict many chemical properties that are not directly accessible by experiments, and play a dominant role in the number of supercomputer cycles currently used for computational chemistry.

We have implemented libraries and tools to support the GA model on a variety of computer systems, including the Intel DELTA and Paragon and the IBM SP-1/2 (all message-passers), the Kendall Square KSR-2 and Convex SPP-1200 (non-uniform access shared-memory machines), the Cray T3D, and on networks of Unix workstations.

The organization of this paper is as follows. Section 2 discusses the NUMA characteristics of current computers. Sections 3, 4 and 5 describe the GA NUMA programming model, the GA toolkit, and its implementations. The performance of several implementations is discussed in Section 6, and Section 7 describes a GA performance visualization tool. Section 8 describes applications that motivated our work. Section 9 outlines future work. Finally, Section 10 summarizes our results and conclusions.

2 NUMA Architecture

NUMA is an important concept in the performance of all modern computers. Consider for instance a standard RISC workstation. Its fast performance is from reliance on algorithms and compilers that optimize usage of the memory hierarchy formed by registers, on-chip cache, off-chip cache, main memory, and virtual memory, as shown in Figure 1.

If a program ignores this structure and constantly flushes the cache or -- even worse -- flushes the virtual memory, performance is seriously degraded. A classic solution to this problem is to access data in blocks small enough to fit in the cache, and then ensure that the algorithm makes sufficient use of the data to justify the data movement costs.

Parallel computers add at least one extra layer to the NUMA hierarchy of sequential computers -- remote memory. Access to remote memory on distributed memory machines is predominantly accomplished through message-passing. Message-passing requires cooperation between sender and receiver which make this programming paradigm difficult to use, and introduces degradation of latency and bandwidth in access to remote memory as comparing to local memory. Many scalable shared-memory machines, such as the Kendall Square Research KSR-2 or the Convex SPP-1200, are actually distributed-memory machines with hardware support for shared-memory primitives. They allow access to remote memory in the same fashion as local memory. However, this uniform mechanism for accessing both local and remote memory is only a programming convenience -- on both shared and distributed memory computers, the latency and bandwidth are significantly larger than for local memory, and therefore must be incorporated into performance models.

If we think about the programming of MIMD parallel computers (either shared or distributed memory) in terms of NUMA, then parallel computation differs from sequential computation only in the essential difference of concurrency[‡], rather than in nearly all aspects. By focussing on NUMA we not only have a framework in which to reason about the perfor-

‡. Concurrent execution would include important issues such as load balancing.

mance of our parallel algorithms (i.e., memory latency, bandwidth, data and reference locality), we also conceptually unite sequential and parallel computation.

3 Global Arrays Programming Model

Our approach, called Global Arrays (GA) [6, 7] is motivated by the NUMA characteristics of current parallel architectures. The key concept of GA is that it provides a portable interface through which each process can independently, asynchronously, and efficiently access logical blocks of physically distributed matrices, with no need for explicit cooperation by other processes. By removing unnecessary processor interactions that are required in the message-passing paradigm to access remote data, the GA model greatly simplifies parallel programming. In this respect, it is similar to the shared-memory programming model. However, the GA model also acknowledges that remote data is slower to access than local (i.e., remote memory as yet another layer of NUMA), and it allows data locality to be explicitly specified and used. While GA manages transfer of data between local and non-local NUMA memory layers, it is up to each MIMD process to manage its local memory. Explicit distinction between local and remote memory and the availability of two distinct mechanisms for accessing local and remote data in GA model are advantages over the shared-memory programming model because they allow the programmer to exploit the NUMA characteristics of parallel computers.

The current GA programming model can be characterized as follows:

- MIMD parallelism is provided using a multi-process approach, in which all non-GA data, file descriptors, and so on are replicated or unique to each process.
- Processes can communicate with each other by creating and accessing GA distributed matrices, and also (if desired) by conventional message-passing.
- Matrices are physically distributed blockwise, either regularly or as the Cartesian product of irregular distributions on each axis.
- Each process can independently and asynchronously access any patch of a GA distributed matrix, without requiring cooperation by the application code in any other process.
- Several types of access are provided, including *get*, *put*, *accumulate* (floating point sum-reduction), and *read-and-increment* (integer). This list is expected to be extended as needed.
- Operations that transfer data from local to remote memory, like put or scatter, might return before the data transfer is complete. A *synchronization* operation implies completion of all pending data transfers. In addition, the *fence* operation allows the calling process to wait for completion of data transfers it initiated.
- Each process is assumed to have fast access to some portion of each distributed matrix, and slower access to the remainder. These speed differences define the data as being 'local' or 'remote', respectively. However, the numeric difference between 'local' and 'remote' access times is unspecified.
- Each process can determine which portion of each distributed matrix is stored 'locally'. Every element of a distributed matrix is guaranteed to be 'local' to exactly one process.

This model differs from other common models as follows. Unlike HPF, it allows task-parallel access to distributed matrices, including reduction into overlapping patches. Unlike Linda, it efficiently provides for sum-reduction and access to overlapping patches. Unlike shared virtual-memory facilities, GA requires explicit library calls to access data, but avoids the operating system overhead associated with maintaining memory coherence and handling virtual page faults, and allows the implementation to guarantee that all of the required data for a patch can be transferred at the same time. Unlike Active Messages, GA does not include the concept of getting another processor's cooperation, which permits GA to be implemented efficiently even on shared-memory systems. Finally, unlike some other strategies based on polling[‡], task duration is relatively unimportant in programs using GA, which simplifies coding and makes it possible for GA programs to exploit standard library codes without modifying them.

---

‡. John Salmon, personal communication, describes a split-request programming strategy in which processes post many requests, then poll for requests to them, poll for replies to their own requests, handle them, and repeat the process.

4 Global Arrays Toolkit

The GA programming model has been implemented in the Global Array toolkit that currently supports two-dimensional arrays. The interface has been designed in the light of emerging standards. In particular, HPF [1,8] may provide the basis for future standards definition of distributed arrays in Fortran. The basic functionality described below (create, fetch, store, accumulate, gather, scatter, data-parallel operations) all may be expressed as single statements using Fortran-90 array notation and the data-distribution directives of HPF. What HPF does not currently provide is random access to regions of distributed arrays from within a MIMD parallel subroutine call-tree, and reduction into overlapping regions of shared arrays.

4.1 Supported operations

Each operation may be categorized as being either an implementation dependent primitive operation or constructed in an implementation independent fashion from primitive operations. Operations also differ in their implied synchronization. A final category is provided by interfaces to third party libraries.The following are primitive operations that are invoked collectively by all processes:

- create an array, controlling alignment and distribution;
- create an array following a provided template (existing array);
- destroy an array; and
- synchronize all processes.

The following are primitive operations that may be invoked in true MIMD style by any process with no implied synchronization with other processes and, unless otherwise stated, with no guaranteed atomicity:

- fetch, store and atomic accumulate into a rectangular patch of a two-dimensional array;
- gather and scatter array elements;
- atomic read and increment of an array element;
- inquiry about the location and distribution of the data; and
- direct access to local elements of array to support and/or improve performance of application specific data-parallel operations.

The following are a set of BLAS-like data-parallel operations that have been developed on top of the primitive operations (synchronization is included as a user convenience):

- vector operations (e.g., dot-product or scale) optimized to avoid communication by direct access to local data;
- matrix operations (e.g., symmetrize) optimized to reduce communication and data copying by direct access to local data; and
- matrix multiplication.

The vector, matrix multiplication, copy, and print operations exist in two versions that operate on either entire arrays or specified sections of arrays. The array sections in operations that involve multiple arrays do not have to be conforming -- the only requirements are that they must be of the same type and contain the same number of elements.

The following is functionality that is provided by third party libraries made available by using the GA primitives to perform necessary data rearrangement. The $O(N^2)$ cost of such rearrangement is observed to be negligible in comparison to that of $O(N^3)$ linear-algebra operations. These libraries may internally use any form of parallelism appropriate to the computer system, such as cooperative message passing or shared memory:

- standard and generalized real symmetric eigensolver; and
- linear equation solver (interface to SCALAPACK [9]).

An example structure of GA-based programs is shown in Figure 2.

4.2 Sample code fragment

The following code fragment uses the Fortran interface to create an $n \times m$ double precision array, blocked in at least 10×5 chunks, which is zeroed and then has a patch filled from a local array. Undefined values are assumed to be computed elsewhere. The routine `ga_create()` returns in the variable `g_a` a handle to the global array with which subsequent references to the array may be made.

```
      integer g_a, n, m, ilo, ihi, jlo, jhi, ldim
      double precision local(1:ldim,*)
c
      call ga_create(MT_DBL, n, m, 'A', 10, 5, g_a)
      call ga_zero(g_a)
      call ga_put(g_a, ilo, ihi, jlo, jhi, local, ldim)
```

The above code is very similar in functionality to the following HPF-like statements

```
      integer n, m, ilo, ihi, jlo, jhi, ldim
      double precision a(n,m), local(1:ldim,*)
!hpf$ distribute a(block(10), block(5))
c
      a = 0.0
      a(ilo:ihi,jlo:jhi)=local(1:ihi-ilo+1,1:jhi-jlo+1)
```

The difference is that this single HPF assignment would be executed in a data-parallel fashion, whereas the global array put operation would be executed in MIMD parallel mode such that each process might reference different array patches.

5 Implementation

We currently support four distinct environments:

1. Distributed-memory, message-passing parallel computers with interrupt-driven communications or Active Messages (Intel iPSC/860, Delta and Paragon, IBM SP-1/2).
2. Networked clusters of multiple- or single-processors with simple message passing (using the MPI [10] or the TCGMSG portable message-passing library [11] on top of TCP/IP).
3. Shared-memory parallel computers (KSR-1/2, Convex SPP, SGI PowerChallenge, also most brands of UNIX workstations‡).
4. Globally-addressable distributed-memory computers (Cray T3D).

The GA can be configured (at compile time) to run with the MPI or TCGMSG message-passing libraries. Regardless of this configuration choice, internally the implementation uses native message-passing libraries on message-passing distributed memory systems (for example, MPL on the IBM SP or NX on the Intel Paragon) which is required to access interrupt-driven communication functionality.

Implementations on the different platforms share nearly all of their code. The distinction arises in the manner in which data are distributed and accessed.

‡. On a single-processor workstation, parallel execution can be emulated with time-sharing multiple processes.

5.1 Internal structure of GA primitive operations

The implementation of primitive operations that provide access to GA data (*get*, *put*, *accumulate, scatter, gather*, *read-and-increment*) attempts to exploit data locality and encapsulate different mechanisms used to access local and remote data at the lowest level.

A reference to a patch of a global array is internally decomposed into references to patches on specific processors. Depending on the mechanism required to access data in the patch (direct copy or message-passing with data server or interrupt receive) either a local or remote version of the operation is executed.

```
ga_operation(handle, GA_range, local_data_range)
{
  Verify if <handle> is valid.
  Verify if <GA_range> is valid.
  Determine <list> of processors that own data in <GA_range>.
  FOREACH processor <p> on <list>
    Determine <GA_subrange> held by <p>.
    Determine <local_data_subrange> corresponding to <GA_subrange>.
    IF direct access to <p> data possible
       ga_operation_local(<handle>, <GA_subrange>, <local_data_subrange>, <p>)
    ELSE
       Decompose <GA_subrange> held by <p> into <chunks> to fit in communication buffer.
       FOREACH <chunk>
         Determine <GA_subrange_chunk>.
         Determine <local_data_subrange_chunk>.
         ga_operation_remote(<handle>,<GA_subrange_chunk>, <local_data_subrange_chunk>, <p>)
       END
    ENDIF
  END
}

ga_operation_local(handle,GA_range,local_data_range)
{
  Verify if <handle> is valid.
  Verify if <GA_range> is valid.
  operation(DATA(<GA_range>),DATA(<local_data_range>))
}

ga_operation_remote(handle,GA_range,local_data_range)
  Verify if <handle> is valid.
  Verify if <GA_range> is valid.
  send_request(<GA_range>,DATA(<local_data_range>), <p>)
  Receive response/data if required.
}
```

5.2 Distributed-memory and networked cluster environments

The availability of interrupt-driven communications on distributed memory machines permits us to establish handlers that support remote access to data which is stored within the contexts of the application processes. The application processes can access 'local' data very fast. Some care is needed to mask interrupts to ensure coherency and guarantee deadlock free execution.

The networked cluster environment assumes the model of shared-memory multiprocessors connected through the network. In this environment, we do not attempt to implement interrupt-driven communications. Instead, we use a data-server model in which server processes manage the data and respond to requests from the client application processes. The current implementation uses only one data server process per cluster as the network bandwidth is considerably lower than intra-cluster memory bandwidth. However, for very fast networks multiple data servers per cluster might be employed if needed.

The data resident in shared-memory within a cluster is directly accessible to all processes executing on this cluster. The remaining data is accessed through message passing between client and data-server on the cluster where the data resides. Mutual exclusion and atomicity of some GA operations are enforced through the shared memory synchronization mechanisms.

With this approach, access to local data is as fast as if the data resided directly in the application processes. However, an additional layer is required on top of the message-passing tools to hide the additional server processes from the application. This requires some effort when the GA toolkit is configured to be used with TCGMSG but can be easily handled with MPI or other message-passing systems that support groups and contexts. There are several other ways that the GA model could be implemented in the network environment, for example using a single process with separate application and server threads with preemptive scheduling.

To complete execution of a GA primitive operation for remote data, as described in Section 5.1, `ga_operation_local()` is invoked either in the interrupt-receive or Active Message handler on the message-passing machines or by the data sever process in the networked cluster environment.

The protocol used to communicate between client and server is almost the same for both environments. Operations such as store or accumulate that require no synchronization cause the requesting process to send a single message. The message contains information that describes the requested operation and data size, followed by the data itself. A read operation requires that the client wait for the response. The current protocol on distributed memory machines has been influenced by features of the most restrictive message-passing system with interrupt-receive capability -- the EUI-H message passing library on the IBM SP-1 -- relatively small (8KB) system message buffers and the in-order message receive rule. The requesting processor posts an asynchronous receive before sending a request for the data. In the network environment, the requesting processor sends a request and then posts a blocking receive for the message that contains the data.

The gather and scatter operations are designed to minimize the number of messages sent. The input list of index pairs are sorted by the process in which the data element resides so that requests for data on that process are bundled into a single message.

5.3 Shared-memory and globally-addressable distributed-memory environments

In order to maintain complete consistency with the other implementations, we provide a distributed-memory environment in which the only shared data is that provided by the Global Arrays library. Our implementation in the shared-memory environment is a special case of the networked cluster implementation with a single cluster and without data server.

The current implementation uses System V shared memory and heavy-weight UNIX processes, rather than threads. On machines such as the Kendall Square Research KSR-2 or SGI PowerChallenge native locks are used to support mutual exclusion, while on other shared-memory platforms, semaphores are used.

On the KSR-2, a substantial performance improvement may be obtained by prefetching subpages (128 bytes) of shared data with the correct access mode (read-only for a get operation, exclusive for put and accumulate operations). The KSR memory architecture permits memory subpages to be put into atomic mode with similar cost to an ordinary non-atomic access to that page. This facility might be used to provide fine grain locking in the accumulate operation which increases scalability. Also on the KSR, we use the dynamic F-way barrier, the fastest barrier algorithm for this machine [12]. On other shared-memory machines, the central barrier algorithm is used.

The code for globally-addressable distributed-memory and shared-memory implementations is almost identical since it uses generalized locking, copy and memory allocation abstractions. Our generalized copy mechanisms instantiate to optimized local-memory copy routines on shared-memory machines and/or to the Cray T3D SHMEM library calls like *shmem_put* or *shmem_get*. Similarly, for synchronization and mutual exclusion, the SHMEM library operations are used. The globally addressable memory on the Cray T3D imposes additional restrictions that have to be taken into account when memory for global arrays is allocated: all addresses on the remote processor have to be valid on the local processor and all

processes have to know the starting address of a block of GA memory on every processor for each global array. The lack of cache coherency with *shmem_put* or *shmem_get* operations requires special care in the implementation on this platform.

6 Performance of Communication Primitives

The efficiency of the primitive communication operations, *get, put* and *accumulate*, might be crucial to the overall performance of the applications that use the toolkit. We demonstrate the performance of these primitives on

- message-passing distributed-memory architectures: the Intel Touchstone Delta, Paragon, the IBM SP-1 and the SP-1.5 (the same machine with faster interconnection network and software of the SP-2, available at the Argonne National Laboratory),
- a NUMA shared-memory architecture, the Kendall Square KSR-2 which is essentially a two-fold faster version of the KSR-1 [13],
- a globally-addressable distributed-memory machine the Cray T3D[14], and
- a multiprocessor shared-memory bus architecture, the SGI PowerChallenge (with 75MHz clock).

In general, each primitive operation can reference data that is physically local, physically remote or both. Also, either contiguous or noncontiguous blocks of memory are accessed depending on whether a one- or a multi-dimensional patch of an array is being referenced. The tests described in this section involved either exclusively local or exclusively remote accesses to square patches of a two-dimensional array resident on a single processor. Latency and bandwidth in access to local and remote data using *put*, *get*, and *accumulate* operations are given in Tables 1, 2, and 3. Latency is defined as time required to access a single element of a global array, and it was measured by timing the execution of a series of hundreds of operations, which in each case referenced different elements of a matrix, and then dividing the time by the number of operations. The consecutively accessed elements were not in the same cache line. Bandwidth was measured for access to a 353×353 square section (approximately 1MB of data) of a 710×710 double precision matrix which was evenly distributed between four processors, each assigned to a 355×355 array block. Since the referenced global array data is noncontiguous in memory, the library optimizations applicable to contiguous data accesses (like avoiding one memory copy on message-passing systems‡) were not in effect. The references to noncontiguous blocks of memory, in this case, correspond to the data access patterns in our targeted applications and in many parallel algorithms in dense numerical linear algebra using block decomposition.

The performance of a remote *get* on the message-passing systems is lower than for the other communication primitives. It has the following components:

$$t_{get} = t_{get\text{-}startup} + n\,[2T_{copy} + T_{comm}]\,, \tag{1}$$

where $t_{get\text{-}startup}$ is the overhead for subroutine calls, array index translation, message sending and receiving, and generating an interrupt on the remote processor, n is the number of bytes, T_{copy} is the per-byte time for a local memory copy, and T_{comm} is the per-byte communication transfer time. Performances of the remote *put* and *accumulate* operations are basically identical to each other:

$$T_{put} = t_{put\text{-}startup} + n\,[T_{copy} + T_{comm}]\,, \tag{2}$$

but differ from that of remote *get* since one memory copy, the message receipt and the remote interrupt are not on the critical path. The processor issuing a remote *put/accumulate* request sends the data and does not wait for the completion of the operation (there is also an option available to wait for acknowledgment sent after request has been processed).

Latency for the *get* operation in access to remote data on message-passing machines greatly depends on the efficiency of the implementation of interrupt-receive. Unfortunately on both the Intel and the IBM machines performance of this useful operation has deteriorated with newer generations of software and/or hardware. This effect is the most profound on the SP-1 (under EUI-H message-passing library) and SP-1.5 which in addition to faster [sic] communication hardware runs under the

‡. Similarly, on the Cray T3D 353 calls to *shmem_put* or *shmem_get* library were required instead of one call that would suffice to transfer one contiguous block of data.

new MPL message-passing library, see Table 2. New implementation of interrupt-receive on the Paragon makes operation that can be used to enable and disable interrupts, *masktrap*, much more costly on the Paragon than on the Delta and explains the higher latency of *accumulate* (an atomic operation) in access to local data, see Table 3.

Unlike the distributed-memory implementations, on the shared-memory platforms, including the SGI PowerChallenge and the KSR-2, computations in the *accumulate* operation are performed by the requesting processor. However, this operation on the KSR-2 is almost no more expensive than the *get* and *put* since prefetching of subpages allows overlapping computations with communication. Prefetching is also crucial in reducing the performance gap between local and remote operations.

While the performance of *get*, *put*, *accumulate* operations on message-passing machines like the Delta, Paragon, or SP-1/1.5 are almost independent of the physical distance between the processor requesting the data and the data owner, on the KSR there can be a significantly variable cost for access to remote data. The bandwidth in the get and put operations is roughly identical and varies with the source/destination of the data from 66MB/s for the processor cache, 32-33 MB/s for the local memory, and 28-29 MB/s for the remote memory of another processor on the same ring to 13.5 MB/s in the case where the data has to be transferred between memories of two processors not on the same ring.

The very high bandwidth in access to the local array elements on the SGI PowerChallenge is due to the fact that before the measurements were taken, each section of the global array had been initialized by the processor that was logically assigned to that section, which placed the data in the second-level cache.

On the Cray T3D, computations in the *accumulate* operation are also performed by the requesting processor. In this case, however, the data is copied to a local buffer, updated, and then copied back. The bandwidths of GA *put* and *get* operations on remote data in this test are in excess of 95% of the Cray *shmem_put* and *shmem_get* bandwidths. The latencies of GA *put* and *get* operations are around 15-20 μs higher than those of the Cray *shmem_put* and *shmem_get* operations. The additional overhead arises from the two-dimensional array index translation, subroutine call overhead, error checking and cache flushing. Unlike the primitive Cray *shmem_put* and *shmem_get* operations, the GA library enforces cache coherency for all operations. The latencies of GA *put* and *get* operations are better than the latency of the native message-passing library, PVM.

The *get*, *put*, and *accumulate* operations are defined in terms of copying to and from process-local buffers. For algorithms designed to operate on local patches, copying the data imposes an unnecessary cost. To address this problem, we have a separate *access* operation whose semantics are defined in terms of obtaining and releasing access to patches. For *access* operation, GA simply returns a reference to the patch so that the application can access the data in-place. The performance improvement of using *access* rather than *get* and *put* primitives can be significant for BLAS-1 type operations that touch each matrix element only once or twice. For example, Figure 3 shows the execution time for a scaled matrix addition,

$$C = \alpha A + \beta B\,, \tag{3}$$

that was implemented on the Delta using the two techniques. The *access* version runs about 25% faster.

7 Visualization Tool for Global Arrays

In order to aid in tuning the performance of applications using global arrays, a visualization and animation tool has been developed. This tool helps the programmer design efficient task scheduling strategies for MIMD algorithms that operate on the distributed two-dimensional data. Minimization of data access contention profits such applications by improving processor utilization.

This particular tool uses trace data that is gathered in a file during the program execution. The global array library is instrumented to generate necessary tracing information whenever the distributed data is accessed. Patterns of accesses are visualized by the tool that processes sequentialized trace events. The user may adjust a time scale for the animation. A color coding is used to differentiate levels of access contention for the particular data blocks. After the animation of events recorded in a tracefile is completed, a composite access contention index is displayed using a different color coding for the entire distributed array. The tool has been implemented using the X Windows and Xt libraries.

The tool was applied to our new distributed SCF program [15]. Our first scheduling of the tasks realized poor parallel efficiency, but the reason was not apparent from simple timing data. The performance tool showed that significant contention for data was the problem, as shown in Figure 4. This problem was readily addressed by reordering the tasks so as to

spread out more uniformly references to the GA matrices, and by the introduction of caching (in the application code) to eliminate redundant data references. This eliminated most of the time lost due to contention, but the parallel speedup was still not as good as expected. The dynamic visualization (animation) then showed that some large tasks were getting scheduled too near the end of the computation, causing a load-balance problem. This was resolved by incorporating a stratified randomizing scheme that approximately preserved the large-to-small order of tasks, necessary for load-balancing, while still reordering tasks of similar size enough to spread out the GA references and avoid contention. The final scheduling, depicted in Figure 5, is both load-balanced and almost contention-free.

The performance of the main computational kernel of the SCF program was improved approximately four-fold by the above tuning, and currently realizes an estimated speedup (relative to a single processor) of 496 on 512 processors of the Intel Delta. The single processor performance is estimated from the performance of smaller calculations and the performance of the same problem size on 64 processors, the smallest number of processors on which it was possible to hold the data.

8 Target Applications

The development of GA was motivated by our development of a large suite of scalable parallel computational chemistry codes. In the three years since GAs were first prototyped many applications have adopted GAs, totalling nearly one-million lines of code, including some 200,000 lines of new code. Although the GA programming model extends the message-passing programming model, no point-to-point message passing is used in any of this application‡ code which illustrates the good match between the GA model and computational chemistry. Examples of applications that attain high parallel scalability and demonstrate the ease of use of GAs in computational chemistry are included in the following references [15, 16, 17, 18, 19].

Generically, the applications that motivated our work [20] can be characterized as

- requiring task parallelism (MIMD), possibly in addition to data parallelism,
- accessing relatively small blocks of matrices that are too large to hold in the memory of any single processor (thus requiring blockwise physical distribution),
- having wide variation in task execution time (thus requiring dynamic load balancing, with attendant unpredictable data reference patterns), and
- having a fairly large ratio of computation to data movement (thus making it possible to retain high efficiency while accessing remote data on demand).

More specifically, we are concerned with computing the electronic structure of molecules and other small or crystalline chemical systems. These calculations are used to predict many chemical properties that are not directly accessible by experiments, and play a dominant role in the number of supercomputer cycles currently used for computational chemistry. A review of electronic structure algorithms for parallel computers can be found in reference [21].

All of the methods considered compute approximate solutions to the non-relativistic electronic Schrödinger equation. In addition to the general characteristics noted above, these applications also

- have large volumes of I/O that can be eliminated by caching or recomputation,
- benefit from specific irregular distributions of data, with alignment of related quantities,
- require linear algebra operations on distributed dense matrices (multiplication, eigensolving, and linear equation solving).

The iterative Self Consistent Field (SCF) method [22] is the simplest method. The major computational kernel contracts integrals with a density matrix to form the Fock matrix. Both matrices are of dimension of an underlying basis set ($N_{basis} \approx 10^3$). The number of integrals scales between $O(N^2_{basis})$ and $O(N^4_{basis})$ depending on the nature of the system and level of accuracy required. To avoid an I/O bottleneck, integrals are recomputed as required [23]. Blocks of one global array, the density matrix, are read and results accumulated into blocks of another global array, the Fock matrix [15]. Parallel efficiency in excess of 97% is reported [15].

‡. Only parallel linear algebra libraries use point-to-point message passing in these applications.

The second-order Møller-Plesset Perturbation method [21] is the simplest theory to improve upon SCF. The dominant computation is the transformation of the integrals used in the SCF algorithm into an orthonormal basis, which is an $O(N^5_{basis})$ process. The resulting very large matrix must be distributed in a specific fashion for subsequent data parallel operations [19]. The related Coupled-Cluster method [24] has a similar structure. Gather and scatter operations are required to access elements of arrays of variable length records packed into linear global arrays.

The Multi-Reference Configuration Interaction (MRCI) method is a highly accurate post-SCF electronic structure method. The parallel COLUMBUS MRCI program [25] was, until the development of these tools, limited in its parallel scalability by either the large amounts of I/O performed upon intermediate quantities or the requirement that these entities be replicated within the memory of each processor to eliminate I/O. The dominant use of the global array tools in this application [17] are to provide a shared, secondary I/O cache. The COLUMBUS I/O library searches local memory, and then global memory for data items before accessing disk. Speedups of 200 on 256 processors of the CRAY-T3D are reported [17] which are unprecedented for this application.

9 Future Work

Although our GA model and toolkit have already demonstrated their value in producing high performance scalable applications, GA development is still evolving. Substantial performance improvement can be gained simply through tuning and incorporating better internal algorithms, such as split-phase remote access for logical blocks that span physical processors. More importantly, even larger improvements can be made by extending the API, such as to allow applications to expose split-phase *get* to the application for even more effective latency hiding. Such improvements will involve tradeoffs in ease-of-use versus performance and will thus require serious evaluation. Finally, we are considering providing support for more general data structures (e.g. sparse matrices) and extending the GA NUMA model and toolkit with operations that provide access to yet another layer of NUMA memory -- with large capacity but slow access -- disk.

10 Summary and Conclusions

We have designed, implemented, used, and characterized a programming facility, called Global Arrays (GA). The key concept of GA is that it provides a portable interface through which each process in a MIMD parallel program can efficiently access logical blocks of physically distributed matrices, with no need for explicit cooperation by other processes (or processors) where the data resides. In this respect, it is similar to the shared-memory programming model. However, the GA model also acknowledges that remote data is slower to access than local, and it allows data locality to be explicitly specified and used. In these respects, it is similar to message passing.

For certain kinds of applications, the GA model provides a better combination of simple coding, high efficiency, and portability than are provided by other models. The applications that motivated our work are characterized by 1) accessing relatively small blocks of very large matrices (thus requiring blockwise physical distribution), 2) having wide variation in task execution time (thus requiring dynamic load balancing, with attendant unpredictable data reference patterns), and 3) having a fairly large ratio of computation to data movement (thus making it possible to retain high efficiency while accessing remote data on demand). Although these characteristics may seem restrictive, our experience to date suggests that many applications would qualify. For example, the GA model provides good support for a large part of computational chemistry, especially electronic structure codes, and it is also promising for application domains like global climate modeling, where application codes often exhibit both spatial locality and load imbalance [26].

Our application interface for GA is designed to permit efficient implementation on a wide variety of platforms. We currently have GA implementations based on 1) interrupt-driven communication using a single process per processor, 2) data server implementation, 3) hardware shared-memory support using multiple processes and mutual exclusion primitives, and 4) get/put hardware primitives in globally-addressable distributed-memory environments.

In addition to the basic programming facilities of GA, we have also developed a performance visualizer tailored to the GA model. The visualizer can provide animations showing instantaneous temporal and spatial access patterns to distributed arrays, and can also provide time-averaged static displays showing aggregate processor time lost due to contention for GA data. In its first use, the visualizer was instrumental in virtually eliminating the time lost due to contention in a large chemistry application.

The Global Arrays toolkit is public domain software available on an anonymous ftp server *ftp.pnl.gov* in directory *pub/global*.

The GA model and tools were developed under our HPCCI Grand Challenge project in Computational Chemistry, the of which focus is on developing algorithms, techniques, and tools to allow computational chemistry applications to exploit future teraflops machines. However, they have turned out to have immediate benefit, and now there is nearly one-million lines of code in high-performance computational chemistry applications that take advantage of GA. Such rapid adoption of a new programming strategy illustrates the power of the HPCCI program in bringing together an effective collaboration of researchers from computer science and the application domains. We hope to see many similar results in the future.

11 Acknowledgments

This work was performed under the auspices of the High Performance Computing and Communications Program of the Office of Scientific Computing, U.S. Department of Energy under contract DE-AC-6-76RLO 1830 with Battelle Memorial Institute which operates the Pacific Northwest National Laboratory. The Environmental Molecular Science Laboratory project is managed by the Office of Energy Research. We thank Dr. David Bernholdt, Dr. Alistair Rendell, Prof. Hans Lischka, Dr. Matthew Rosing and George Fann for valuable discussions.

References

[1] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation , Rice University, 1993.

[2] J. Stephen and R. Oldehoeft. HEP SISAL: Parallel functional programming. In J. Kowalik, editor, *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, pages 123–150. The MIT Press, Cambridge, MA, 1985.

[3] I.T. Foster, R. Olson and S. Tuecke. Productive parallel programming: The PCN approach. *Scientific Programming*, 1(1):51–66, 1992.

[4] I. Foster and K. Chandy. Fortran M: A language for modular parallel programming. *J. Parallel and Distributed Computing*, 25(1), 1995.

[5] N. Carriero and D. Gelernter. *How To Write Parallel Programs. A First Course*. The MIT Press, Cambridge, MA, 1990.

[6] J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: A portable 'shared-memory" programming model for distributed memory computers. In *Proc. Supercomputing 1994*, pages 340–349. IEEE Computer Society Press, 1994.

[7] J. Nieplocha, R. J. Harrison, R. J. Littlefield. The Global Array programming model for high performance scientific computing. *SIAM News*, 28(7):12–14, 1995.

[8] High Performance Fortran Forum II. information available from chk@cs.rice.edu.

[9] Scalapack, scalable linear algebra package. code and documents available through netlib.

[10] Message Passing Interface Forum. MPI: A message-passing interface. May 5 1994.

[11] R. Harrison. Portable tools and applications for parallel computers. *Int. J. Quant. Chem.*, 40:847–863, 1991.

[12] D. Grunwald and S. Vajracharya. Efficient barriers for distributed shared memory computers. In *Proc. of 8th IPPS*, pages 202–213. IEEE Computer Society, 1994.

[13] R.H. Saavedra, R.S. Gaines and M.J. Carlton. Micro benchmark analysis of the KSR1. In *Proc. of Supercomputing 93*, pages 202–213. IEEE Computer Society, 1993.

[14] Cray Research Inc. *Cray T3D System Architecture*, 1994.

[15] R.J. Harrison, M.F. Guest, R.A. Kendall, D.E. Bernholdt, A.T. Wong, M.S. Stave, J.L. Anchell, A.C. Hess, R.J. Littlefield, G.I. Fann, J. Nieplocha, G.S. Thomas, D. Elwood, J. Tilson, R.L. Shepard, A.F.Wagner, I.T. Foster, E. Lusk and R. Stevens. Toward high-performance computational chemistry: II. A scalable self-consistent field program. *J. Comp. Chem.*, 17(1):124–132, 1996.

[16] D. Bernholdt and R. Harrison. Large-scale correlated electronic structure calculations: The RI-MP2 method on paral-

lel computers. *Chem. Phys. Lett.*, in press.

[17] H. Dachsel, H. Lischka, R.L. Shepard and R.J. Harrison. A massively parallel multireference configuration interaction program - the parallel COLUMBUS program. *Journal of Chem. Phys.*, submitted.

[18] A.T. Wong, R.J. Harrison and A.P. Rendell. Parallel direct four-index transformations. *Theo. Chim. Acta*, in press.

[19] D. Bernholdt and R. Harrison. Orbital-invariant second-order many-body perturbation theory on parallel computers: An approach for large molecules. *Journal of Chem. Phys.*, 102(24):9582–9589, 1995.

[20] M.F. Guest, E. Apra, D. E. Bernholdt, H. A. Fruechtl, R. J. Harrison, R. A. Kendall, R. A. Kutteh, X. Long, J. B. Nicholas, J. A. Nichols, H. L. Taylor, A. T. Wong, G. I. Fann, R. J. Littlefield and J. Nieplocha. High performance computational chemistry: Nwchem and fully distributed parallel algorithms. In J. Dongarra, L. Gradinetti, G. Joubert, and J. Kowalik, editor, *High Performance Computing: Technology, Methods, and Applications*, pages 395–427. Elsevier Science B. V., 1995.

[21] R. J. Harrison and R. Shepard. Ab initio molecular electronic structure on parallel computers. *Annu. Rev. Phys. Chem.*, 45:623–58, 1994.

[22] A. Szabo and N. Ostlund. *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*. McGraw-Hill, Inc., New York, 1989.

[23] J. Almlöf, K. Faegri and K. Korsell. The direct SCF method. *J. Comp. Chem.*, 3:385, 1982.

[24] A.P. Rendell, M.F. Guest and R.A. Kendall. Distributed data parallel coupled-cluster algorithm: Application to the 2-hydroxypyridine/2-pyridone tauto-merism. *J. Comp. Chem.*, 14:1429–1439, 1993.

[25] M. Schueler, T. Kovar, H. Lischka, R. Shepard and R.J. Harrison. A parallel implementation of the COLUMBUS multireference configuration interaction program. *Theor. Chim. Acta*, 84:489–509, 1993.

[26] J. Michalakes. Analysis of workload and load balancing issues in NCAR community climate model. Technical Report MCS-TM-144, Argonne National Laboratory, Argonne, IL., 1991.

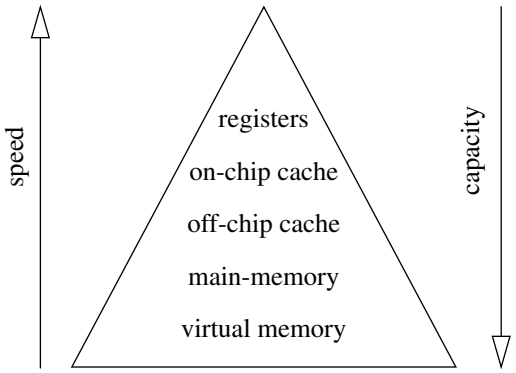

Figure 1: NUMA memory hierarchy.

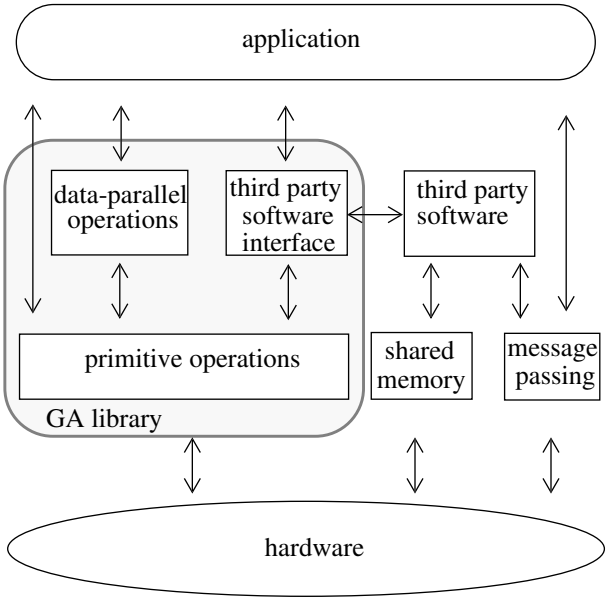

Figure 2: Structure of a GA-based program

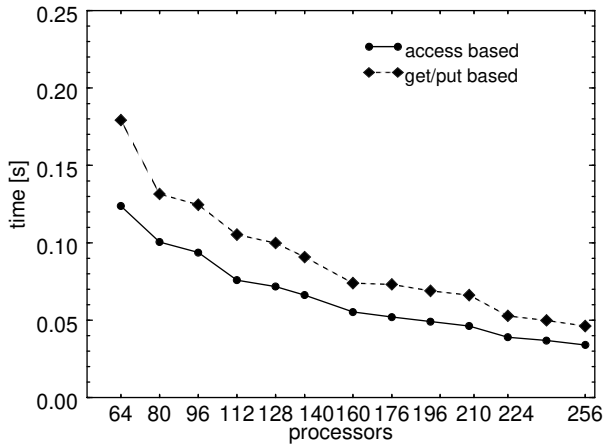

Figure 3: Performance of the scaled add operation implemented using the access or get and put operations for a 3000×3000 problem on the Intel Touchstone Delta.

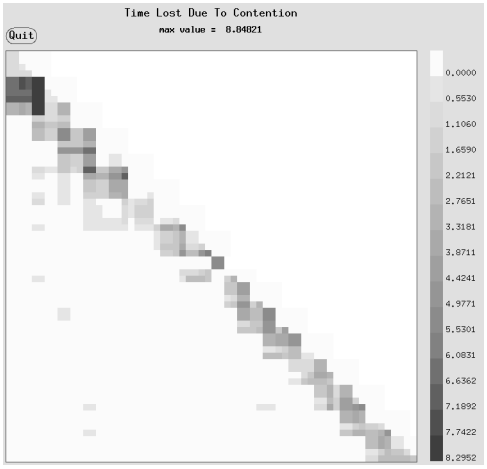

Figure 4: A snapshot of the GA performance analysis tool prior to the optimizations discussed in the text. The pixels correspond to the elements of a 2-dimensional array. Depicted is the performance of a 300 basis function calculation on a chain of water molecules. Elements are shaded according to the time processes wasted waiting for those elements.

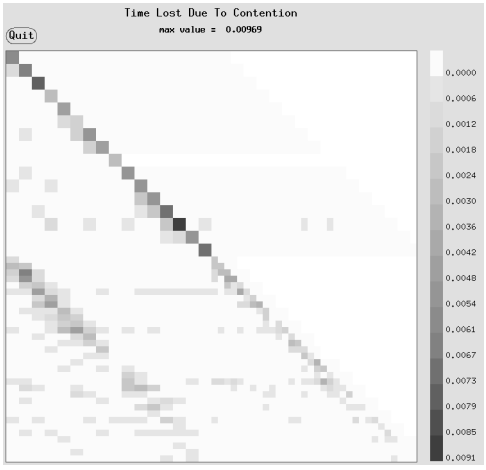

Figure 5: A snapshot of the Global Array performance analysis tool after optimizations have been included. The scale for contention is three orders of magnitude smaller than in the preceding display.

Table 1: Performance of *put*

machine	local data		remote data	
	latency µs	bandwidth MB/s	latency µs	bandwidth MB/s
Delta	34	19.5	190	6.3
Paragon	29	93	62	32
SP-1	17	84	72	7.7
SP-1.5	13	86	75	18
Cray T3D	17	153	18.7	120
KSR-2	35	32	36	29
SGI	9.2	430	12	76

Table 2: Performance of *get*

machine	local data		remote data	
	latency µs	bandwidth MB/s	latency µs	bandwidth MB/s
Delta	35	19.5	360	5.3
Paragon	31	92	497	17
SP-1	19	84	185	7.3
SP-1.5	13	86	517	14
Cray T3D	22	151	24.4	34
KSR-2	36	32	37	28
SGI	12	300	14	70

Table 3: Performance of *accumulate*

machine	local data		remote data	
	latency µs	bandwidth MB/s	latency µs	bandwidth MB/s
Delta	35	7.7	197	5.9
Paragon	166	46	64	36
SP-1	19	73	76	7.4
SP-1.5	18	74	77	17
Cray T3D	27	94	34.7	20
KSR-2	38	29	45	26
SGI	11.5	360	12.6	55