Global Arrays: A Portable "Shared-Memory" Programming Model for Distributed Memory Computers

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

Pacific Northwest 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 a new 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 (all message-passers), the Kendall Square KSR-2 (a nonuniform access shared-memory machine), 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.

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 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 Laboratory (PNL) have a short-term goal of developing, within the next three years, a suite of parallel chemistry application codes to be used in production mode for chemistry research at PNL's Environmental and 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.

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 a new 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

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

ISSN: 1063-9535.

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.

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 (all message-passers), the Kendall Square KSR-2 (a nonuniform access shared-memory machine), and on networks of Unix workstations. We have also used GA to implement several large chemistry applications for the HPCCI project and EMSL, and plan to continue its use.

The organization of this paper is as follows. Section 2 outlines and characterizes the applications that we are most interested in. Sections 3 and 4 describe the GA programming model and its implementations. The performance of two implementations is discussed in Section 5, and Section 6 describes a GA performance visualization tool. Section 7 outlines future work. Finally, Section 8 summarizes our results and conclusions.

2 Target Applications

Generically, the applications that motivated our work 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.

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 [6] 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 [7]. Blocks of the density matrix must be read and results accumulated into blocks of the Fock matrix [8].

The second-order Møller-Plesset Perturbation method [6] 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. The related Coupled-Cluster method [9] 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 the most accurate post-SCF electronic structure method. The parallel COLUMBUS MRCI program [10] 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 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.

3 Functionality and Interface

3.1 Programming model

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 two-dimensional patch of a GA distributed matrix, without requiring cooperation by the application code in any other process.
- Several types of access are supported, including 'get', 'put', 'accumulate' (floating point sum-reduction), and 'get and increment' (integer). This list is expected to be extended as needed.
- 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.

3.2 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 synchronously 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 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 array(s) or specified sections of array(s). 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 [11]).

3.3 Sample code fragment

This interface has been designed in the light of emerging standards. In particular, HPF [1,12] will certainly provide the basis for future standards definition of distributed arrays in FORTRAN. The basic functionality described above (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.

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.

4 Implementation

We currently support three distinct environments:

1. Distributed-memory, message-passing parallel computers with interrupt-driven communications or Active Messages (Intel Gamma, Delta and Paragon, IBM SP-1).
2. Networked workstation clusters with simple message passing (using the TCGMSG portable message-passing library [13] on top of TCP/IP).
3. Shared-memory parallel computers (KSR-2, SGI, most UNIX workstations).

4.1 Distributed-memory and network environments

Implementations on the distributed-memory and network environments share nearly all of their code. The distinction arises in the manner in which data are distributed and accessed. The availability of interrupt-driven communications on distributed memory machines permits us to establish handlers that support remote access to local data which is then stored within the application processes. This permits very fast access to 'local' data. Some care is needed to mask interrupts to ensure coherency and guarantee deadlock free execution. In the network 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. While this approach is very portable, access to local data is not as fast as if the data resided directly in the application processes. Also, an additional layer is required on top of the message-passing tools to hide the additional server processes from the application. There are several other ways that the GA model could be implemented in the network environment, notably either sharing the memory associated with GA matrices between server and client processes, or using a single process with separate application and server threads. Both of these would likely be faster but less portable than our current approach.

A reference to a patch of a global array is internally decomposed into references to patches on specific processors. 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 for read operation on the distributed memory machines has been influenced by features of the EUI-H message passing library on the IBM SP-1: relatively small (8KB) system message buffers and the in-order message delivery 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 protocols will be unified when asynchronous communication becomes supported in TCGMSG.

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.

4.2 Shared-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 array library. 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 native memory locks are used to support mutual exclusion, while on other platforms semaphores are used. Implementations that use semaphores currently sequentialize access to the entire array in 'read_and_increment' and 'accumulate' operations, but a more scalable mutual exclusion algorithm is planned.

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 is used to provide fine grain locking in the accumulate operation which increases scalability. Also on the KSR, we use a dynamic F-way barrier which is claimed to be the fastest barrier for this machine [14]. On other machines, the central barrier algorithm is used.

5 Performance of Communication Primitives

The efficiency of the elementary communication operations, get, put and accumulate, might be crucial to the overall performance of the applications that use the toolkit. We demonstrate performance of these primitives on a message-passing distributed-memory architecture, the Intel Touchstone Delta, and on a NUMA shared-memory architecture, the Kendall Square KSR-2 which is essentially a two-fold faster version of the KSR-1 [15].

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. 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 latency of local and remote get, put, and accumulate operations, as a function of the number of bytes, referenced in small to medium size patches of an array, is illustrated in Figure 1 for the Intel Touchstone Delta, in Figure 2 for the KSR-2 and Figure 3 for the IBM SP-1. While the latencies on the Delta and the SP-1 are almost independent of the physical distance between the processor requesting the data and the data owner, on the KSR platform there can be a significantly variable cost for access to remote data. This effect is clearly seen in Figure 2 for remote data located in the memory of another processor on the same and a different ring.

Both visual inspection of Figure 2 and statistical analysis (linear and nonlinear regression) of the gathered data

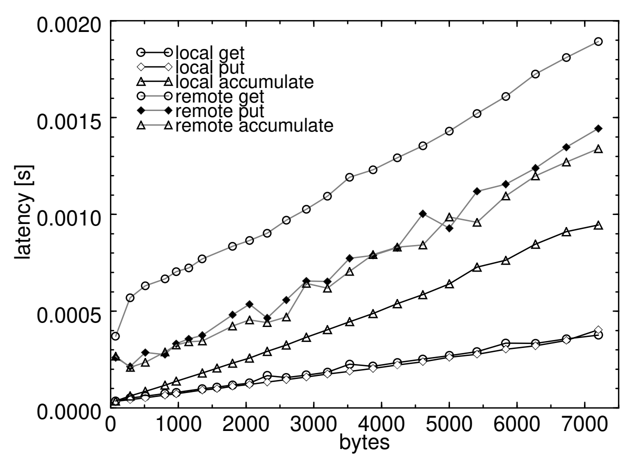

Figure 1: The latency of local and remote get, put, and accumulate operations as a function of the number of bytes, referenced in small to medium size patches of an array for the Intel Touchstone Delta.

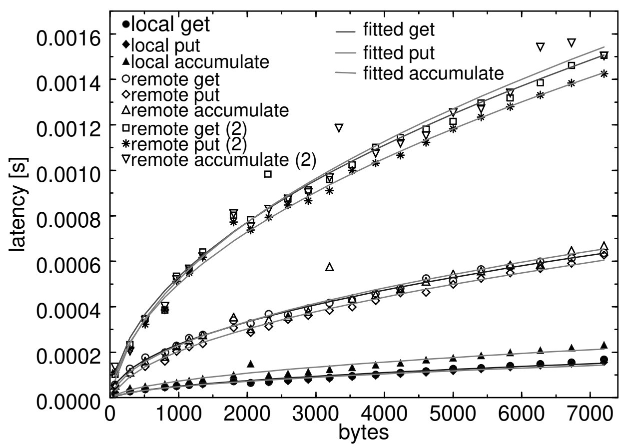

Figure 2: The latency of local and remote get, put, and accumulate operations as a function of the number of bytes, referenced in small to medium size patches of an array for the Kendall Square KSR-2.

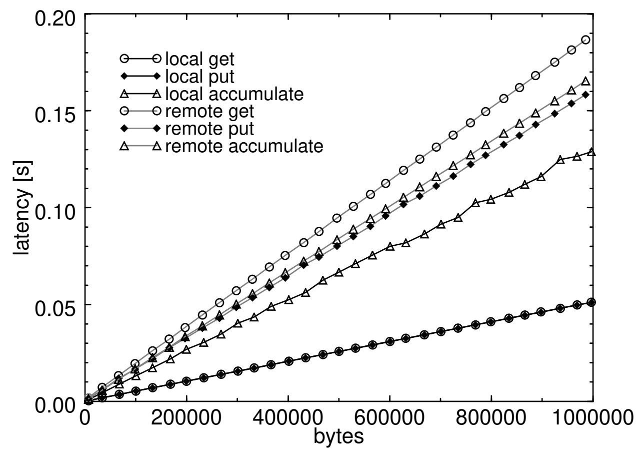

Figure 4: The latencies of local and remote get, put, and accumulate operations for larger array patches for the Intel Touchstone Delta.

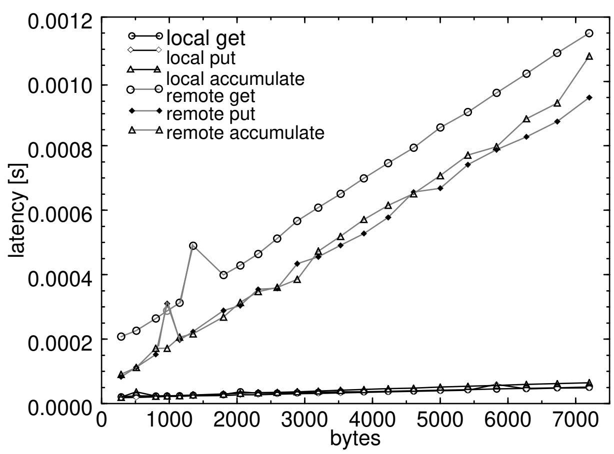

Figure 3: The latency of local and remote get, put, and accumulate operations as a function of the number of bytes, referenced in small to medium size patches of an array for the IBM SP-1.

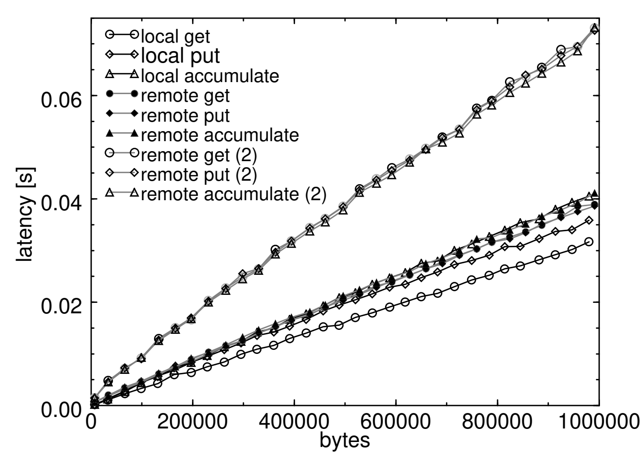

Figure 5: The latency of local and remote get, put, and accumulate operations for larger array patches for the Kendall Square KSR-2.

reveal the presence of a nonlinear relationship between the latency and the number of bytes in remote operations on the KSR-2. The statistically significant nonlinear -- square root -- term is present due to the fact that each column of a small square patch of an array will reside in a separate 128-byte subpage. Thus for small patches, the number of subpages that must be transferred is proportional to the number of columns, or the square root of the total patch size. The curves in this figure were prepared based on the actual regression model fitted for the empirical data.

The latencies of local and remote get, put, and accumulate operations for larger array patches are shown in Figure 4 for the Delta, Figure 5 for the KSR-2 and Figure 6 for the SP-1. The functional relationship between latency and the number of bytes appears to be linear.

The latency of remote get on the Delta and the SP-1 is higher than for the other communication primitives. It has the following components:

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

where $t_{get\text{-}startup}$ is the overhead for subroutine calls, sending and receiving two messages, and generating an interrupt on the remote processor, n is the number of bytes, T_{copy} is

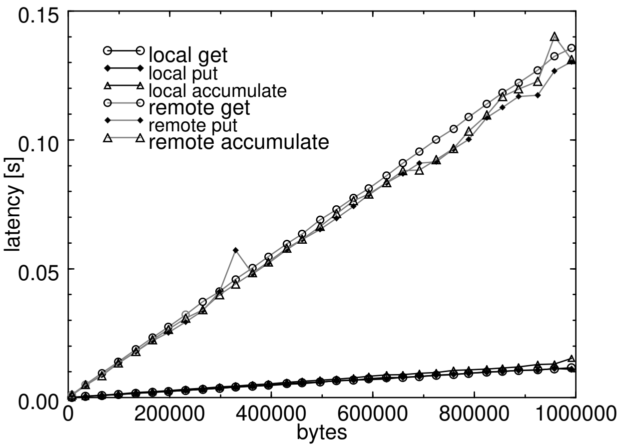

Figure 6: The latency of local and remote get, put, and accumulate operations for larger array patches for the IBM SP-1.

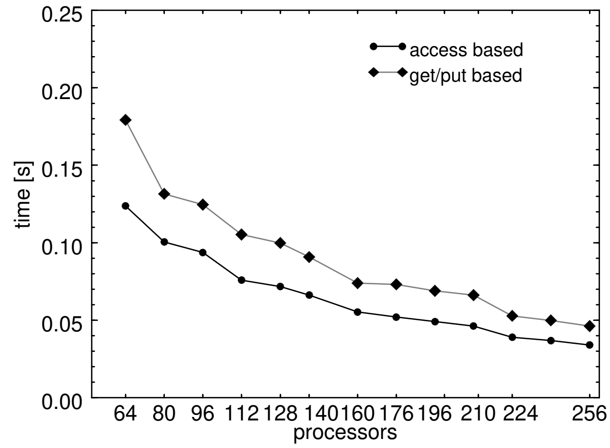

Figure 7: Performance of the scaled add operation implemented using the access or get and put operations for a 3000×3000 problem on the Intel Touchstone Delta.

the per-byte time for a local memory copy, and T_{comm} is the per-byte communication transfer time. Latencies of the remote put and accumulate operations are basically identical to each other:

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

but differ from that of remote get since one memory copy, the two message receipts 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). On the Delta, the bandwidth in access to the local data is 19.5 MB/s, and 5.3 to 6.25 MB/s for remote get and remote put, respectively. These results are consistent with the performance of the message-passing communication on this machine[16]. On the IBM SP-1, the bandwidth in access to the local data is 85 MB/s, and 7.3 to 7.7 MB/s for remote get and remote put, respectively.

Unlike the distributed-memory implementations, on the KSR-2 and other shared-memory platforms, 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. 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, 33 MB/s for the local memory, and 25.4 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 ‘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 added a separate set of ‘access’ operations whose semantics are defined in terms of obtaining and releasing access to patches. For ‘access’ operations, 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 7 shows the execution time for a scaled matrix addition,

$$C = \alpha A + \beta B, \qquad (3)$$

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

6 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

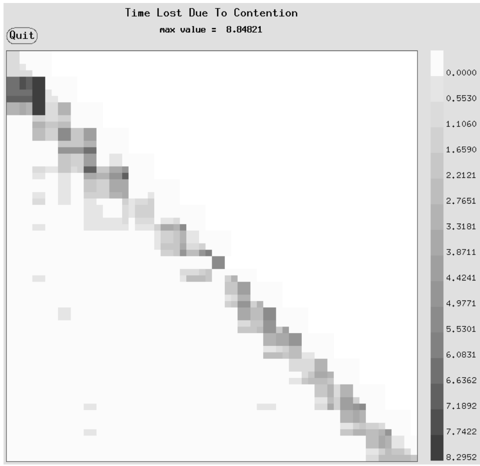

Figure 8: A snapshot of the GA performance analysis tool prior to the optimizations discussed in the text. The pixels correspond to the elements of a 2-dimensional array. Depicted is the performance of a 300 basis function calculation on a chain of water molecules. Elements are shaded according to the time processes wasted waiting for those elements.

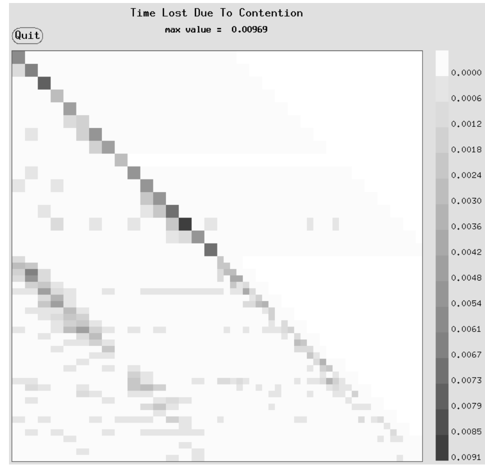

Figure 9: A snapshot of the Global Array performance analysis tool after optimizations have been included. The scale for contention is three orders of magnitude smaller than in the preceding display.

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 [8]. 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 8‡. This problem was readily addressed by reordering the tasks so as to more uniformly spread out 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 9‡, 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.

7 Future Work

Although our GA model and tools have already demonstrated their value in producing high performance scalable applications, GA development is far from complete. Our current implementations are first-generation research efforts, not robust production quality versions. 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 specify access patterns in addition to physical distributions, or to expose split-phase ‘get’ to the application for even more effective latency hiding. Such

‡. Color versions of Figures 8 and 9 appear at the end of Proceedings.

improvements will involve tradeoffs in ease-of-use versus performance and will thus require serious evaluation.

8 Summary and Conclusions

We have designed, implemented, and used a new 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 seems to provide 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 [17].

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) blocking communications, using two processes per processor (an application process and a data server), and 3) hardware shared-memory support using multiple processes and mutual exclusion primitives.

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 GA model and tools were developed under our HPCCI Grand Challenge project in Computational Chemistry, whose focus is on developing algorithms, techniques, and tools for allowing computational chemistry applications to exploit future teraflops machines. However, they have turned out to have immediate benefit as well, and are now being widely adopted in parallel chemistry codes developed for production use in PNL's Environmental and Molecular Science Laboratory. 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.

9 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 Laboratory. The Environmental and 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, Rice University, 1993.

2. J.A. Stephen and R.R. Oldehoeft, 'HEP SISAL: Parallel Functional Programming' in *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, pp. 123-150, ed. J.S. Kowalik, The MIT Press, Cambridge, MA, 1985.

3. I.T. Foster, R. Olson and S. Tuecke, 'Productive Parallel Programming: The PCN Approach,' *Scientific Programming*, pp. 51-66, 1, 1992.

4. I.T. Foster and K.M. Chandy, *Fortran M: A Language for Modular Parallel Programming*, Argonne National Laboratory, preprint MCS-P327-0992, 1992.

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

6. A. Szabo and N.S. Ostlund, *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory.* 1st Ed. Revised, McGraw-Hill, Inc., New York, 1989.

7. J. Almlöf, K. Faegri and K. Korsell, 'The Direct SCF Method,' *J. Comp. Chem.*, 385, 3, 1982.

8. 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, 'Fully Distributed Parallel Algorithms -- Molecular Self Consistent Field Calculations,' *J. Comp. Chem.*, submitted for publication, 1994.

9. A.P. Rendell, M.F. Guest and R.A. Kendall, 'Distributed Data Parallel Coupled-Cluster Algorithm: Application to the 2-Hydroxypyridine/2-Pyridone Tautomerism,' *J. Comp. Chem.,* pp. 1429-1439, 14, 1993.

10. M. Schuler, T. Kovar, H. Lischka, R. Shepard and R.J. Harrison, 'A parallel implementation of the COLUMBUS multireference configuration interaction program,' *Theor. Chim. Acta*, pp. 489-509, 84, 1993.

11. SCALAPACK, scalable linear algebra package, code and documents available through `netlib`.

12. High Performance Fortran Forum II, information available from `chk@cs.rice.edu`.

13. R.J. Harrison, 'Portable Tools and Applications for Parallel Computers,' *Int. J. Quant. Chem.*, pp. 847-863, 40, 1991.

14. D. Grunwald and S. Vajracharya, 'Efficient barriers for distributed shared memory computers,' *Proceedings of 8th IPPS*, pp. 604-608, 1994.

15. R.H. Saavedra, R.S. Gaines and M.J. Carlton, Micro Benchmark Analysis of the KSR1, *Proc. of Supercomputing '93*, IEEE Computer Society, pp. 202-213, 1993.

16. R.J. Littlefield, 'Characterizing and Tuning Communications Performance on the Touchstone DELTA and iPSC/860,' *Proc. of the Intel Supercomputer Users' Group 1992 Annual Users conference,* pp. 309-313, 1992.

17. J. Michalakes, *Analysis of Workload and Load Balancing Issues in NCAR Community Climate Model*, Argonne National Laboratory, technical report MCS-TM-144, 1991.