SIAM News, August/September 1995

The Global Array Programming Model for High Performance Scientific Computing

J. Nieplocha, R.J. Harrison and R.J. Littlefield
Pacific Northwest Laboratory

Motivated by the characteristics of current parallel architectures, we have developed an approach to the programming of scalable scientific applications that combines some of the best features of message-passing and shared-memory programming models. Two assumptions permeate our work. The first is that most high performance parallel computers have, and will continue to have, physically distributed memories with non-uniform memory access (NUMA) timing characteristics. NUMA machines 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 the development of techniques and tools that minimize the extra effort required to construct application programs with explicit control of locality.

There are significant tradeoffs among the important considerations of portability, efficiency, and ease of coding. The message-passing programming model is widely used because of its portability. Some applications, however, are too complex to be coded in a message-passing mode, if care is to be taken 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 such languages and facilities 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. None of these models represents an ideal solution.

Global Arrays (GAs), the approach described here, lead to both simple coding and efficient execution for a class of applications that appears to be fairly common. The key concept of a GA model 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

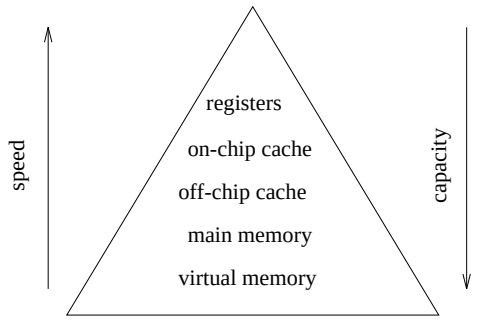

Figure 1: The memory hierarchy of a typical NUMA architecture.

matrices, with no need for explicit cooperation by other processes. In this respect, it is similar to the shared-memory programming model. In addition, however, the GA model acknowledges that more time is required to access remote data than local data, and it allows data locality to be explicitly specified and used. In these respects, it is similar to message passing.

NUMA Architecture

The concept of NUMA is important even to the performance of modern (sequential) personal computers or workstations. On a standard RISC workstation, for instance, good performance of the processors results from algorithms and compilers that optimize usage of the memory hierarchy. The memory hierarchy is formed by registers, on-chip cache, off-chip cache, main memory, and virtual memory (see Figure 1).

If the programmer ignores this structure and constantly flushes the cache or, even worse, thrashes the virtual memory, performance will be seriously degraded. The 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 encached data to justify the costs of moving the data.

To the NUMA hierarchy of sequential computers,parallel computers add at least one extra layer: remote memory. Access to remote memory on distributed-memory machines is accomplished through message passing. Message passing, in addition to the required cooperation between sender and receiver that makes this programming paradigm difficult to use, introduces degradation of latency and bandwidth in the accessing of remote, as opposed to local, memory.

Scalable shared-memory machines,i.e., architecturally distributed-memory machines with hardware support for shared-memory operations (for example, the KSR-2 or the Convex Exemplar), allow access to remote memory in the same fashion as to local memory. However, this uniform mechanism for accessing local and remote memory should be seen only as a programming convenience—

on both shared- and distributed-memory computers, the latency and bandwidth for accessing remote memory are significantly larger than for local memory and therefore must be incorporated into performance models.

If we think about programming of MIMD parallel computers (either shared- or distributed-memory) in terms of NUMA, then parallel computation differs from sequential computation only in terms of concurrency. By focusing on NUMA, we not only have a framework in which to reason about the performance of our parallel algorithms (i.e., memory latency, bandwidth, data and reference locality), we also conceptually unite sequential and parallel computation.

Global Array Model

The GA programming model is motivated by the NUMA characteristics of current parallel architectures. By removing the unnecessary processor interactions required to access remote data in message-passing paradigm, the GA model greatly simplifies parallel programming and is similar in this respect to the shared-memory programming model. However, the GA model also acknowledges that it is more time consuming to access remote data than local data (i.e., remote memory is yet another layer of NUMA), and it allows data locality to be explicitly specified and used. Advantages of the GA model over a shared-memory programming model include its explicit distinction between local and remote memory and the availability of two distinct mechanisms for accessing local and remote data. Global arralys, instead of hiding the NUMA characteristics, expose them to the programmer and make it possible to write more efficient and scalable parallel programs.

The current GA programming model can be characterized as follows:

- MIMD parallelism is provided via a multiprocess 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, as well as (if desired) by conventional message passing.
- Matrices are physically distributed block-wise, 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 from 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 can be extented 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 memory 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[5], it efficiently provides for sum-reduction and access to overlapping patches. Unlike shared-virtual-memory software facilities,the GA paradigm requires explicit library calls to access data but avoids the overhead associated with the maintenance of memory coherence and handling of virtual page faults. The GA implementation guarantees that all of the required data for a patch can be transferred at the same time. Unlike active messages[6], the GA model does not incorporate the concept of interprocessor cooperation and can thus be implemented efficiently[7] even on shared-memory systems. Finally, unlike some other strategies based on polling, task duration is relatively unimportant in programs that use GAs, which simplifies coding and makes it possible for GA programs to exploit standard library codes without modification.

Global Array Toolkit

This GA interface has been designed in the light of emerging standards. In particular, High Performance Fortran (HPF) will certainly provide the basis for future standards definition for distributed arrays in Fortran. The operations that provide the basic functionality (create, fetch, store, accumulate, gather, scatter, data-parallel operations) all can be expressed as single statements in Fortran-90 array notation and with the data-distribution directives of HPF. The GA model is, however, more general than that of HPF, which currently precludes the use of such operations in MIMD (task-parallel) code.

Supported Operations

Each GA operation may be categorized as either an implementation- dependent primitive operation or an operation that has been constructed in an implementation-independent fashion from primitive operations. Operations also differ in their implied synchronization. Interfaces to third-party libraries provide a final distinction. The following primitive operations 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;
- synchronize all processes.

The following primitive operations can 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 array elements;
- inquire about the location and distribution of the data;
- directly access local elements of array to support and/or improve performance of application-specific data-parallel operations.

The following set of BLAS-like data-parallel operations currently available in GAs can easily be extended (efficient implementation can be done in an architecture-independent fashion on top of the GA primitive operations):

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

Functionality provided by third-party libraries–standard and generalized real symmetric eigensolvers and linear equation solvers(interface to ScaLAPACK)–is made available by using the GA primitives to perform necessary data rearrangement. The $\mathcal{O}(N^2)$ cost of such rearrangements is observed to be negligible in comparison to that of $\mathcal{O}(N^3)$ linear algebra operations. These libraries can internally use any form of parallelism appropriate to the computer system, such as cooperative message-passing or shared-memory.

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; after zeroing, a patch is filled from a local array. Undefined values are assumed to be computed elsewhere. The routine `ga_create()` returns the variable `g_a` as a handle to the global array for subsequent references to the array.

```
      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)
```

This 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 `ga_put` operation would be executed in MIMD parallel mode, with each process able to reference different array patches.

Supported Platforms and Availability

The public-domain GA toolkit is supported on a wide range of distributed- and shared-memory computer systems, including:

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. Networks of uniprocessor and multiprocessor Unix workstations.
3. Shared-memory parallel computers (KSR-1/2, Cray T3D, SGI).

The GA toolkit is available via anonymous ftp on `ftp.pnl.gov` in the directory `pub/global`. Further information is provided on the WWW at the URL, `http://www.emsl.pnl.gov:2080/docs/global/ga.html`.

Applications

Most applications of the GA toolkit have been in the area of computational chemistry, in determinations of the electronic structures of molecules or crystalline chemical systems. These calculations, which can predict many chemical properties that are not directly observed experimentally, account for a large fraction of the supercomputer cycles currently used for computational chemistry. All of these methods, of which the iterative self consistent field (SCF) method [8] is the simplest, compute approximate solutions to the nonrelativistic electronic Schrödinger equation.

As an example of the programming simplifications and performance improvements that can be realized with GAs, we consider the parallel SCF application here is slightly more detail. Full details and a recent literature survey can be found in [9, 10].

The kernel of the SCF calculation is the contraction of a large, sparse four-index matrix (electron-repulsion integrals) with a two-index matrix (the electronic density) to yield another two-index matrix (the Fock matrix). The irregular sparsity and the available symmetries of the integrals drive the calculation. The dimensions of both matrices are determined by the size of an underlying basis set ($N \approx 10^3$). The number of integrals scales between $\mathcal{O}(N^2)$ and $\mathcal{O}(N^4)$ depending on the nature of the system and level of accuracy required. Integrals are most efficiently computed in batches, and each batch connects up to six blocks of the density matrix with the corresponding blocks of the Fock matrix. The cost of evaluating these batches can vary by a factor of more than 1000, which causes a load-balancing problem.

The two previous significant distributed-data algorithms both used explicit message passing. In the systolic loop algorithm of Colvin et al. [10], the best possible execution time is no better than $\mathcal{O}(N_{basis}^2)$. Overall efficiencies of approximately 50% were obtained on 256 processors of an nCUBE-2. The most efficient explicit message-passing algorithm is that of Furlani and King [10], who implemented rather complicated distributed-matrix schemes with polling. The overhead assciated with communication and waiting for responses to requests for access to the density matrix was reduced by explicit double-buffering and asynchronous prefetching. This approach scales well, but the polling causes high latencies in access to the distributed matrices, thus requiring the introduction of the additional complexities of prefetching.

Given the high degree of complexity of these message-passing algorithms, and the simplicity of SCF as compared with other ab initio algorithms, we have been seeking more appropriate programming models; the GA model is the current result. Our latest SCF program, which has been developed on top of the GA toolkit, is very simple, and all computational steps with complexity greater than $\mathcal{O}(N_{atom})$ have been parallelized. The four nested loops over the unique integrals are stripmined into blocks, similar in spirit to Furlani and King. Geometric decomposition permits the use of sparsity for reducing both computation and references to global data. Assignment of multiple atom quartets to a task improves the caching of reads of the density matrix and accumulation into the Fock matrix, although too large a task size degrades load balancing. All tasks are dynamically assigned, which is made possible by the one-sided data access provided by GAs. The GA visualization program, which demonstrates access patterns to distributed arrays, was instrumental in designing an efficient task-scheduling strategy for the SCF program [7].

A simple performance model predicts a constant efficiency of about 99% for the Fock matrix construction for up to $\mathcal{O}(N_{atom}^2)$ processors (for extended molecular systems), at which point load imbalance will degrade performance. In

a modestly sized calculation (731 basis functions) using 512 processors of the Intel Delta, we obtained a speedup of 496 (97% efficiency) for the Fock-matrix construction.

Conclusions

Global Arrays are a new parallel programming environment for the development of scientific applications on massively parallel computers. The GA model 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.

For applications of certain types, the GA model provides a better combination of simple coding, high-level efficiency, and portability than do do other models. The applications that motivated the development of GA are characterized by (1) the need to access relatively small blocks of very large matrices (thus requiring block-wise physical distribution); (2) wide variation in task execution time (thus requiring dynamic load balancing, with attendant unpredictable data reference patterns); and (3) a fairly large ratio of computation to data movement (thus making it possible to retain a high efficiency while accessing remote data on demand). The GA model provides good support for many areas of computational chemistry, especially electronic structure codes. It also appears promising for such application domains as global climate modeling, in which the codes are often characterized by both spatial locality and load imbalance.

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] T. von Eicken, D.E. Culler, S.C. Goldstein and K.E. Schauser, *Active messages: A mechanism for integrated communications and computation*, Proc. 19th Ann. Int. Symp. Comp. Arch., pp. 256-266, 1992.

[7] J. Nieplocha, R.J. Harrison and R.J. Littlefield, *Global Arrays: A Portable "Shared-Memory" Programming Model for Distributed Memory Computers*, Proc. Supercomputing 1994, IEEE Computer Society Press, pp. 340–349, 1994.

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

[9] 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., in press.

[10] R.J. Harrison, R.L. Shepard, *Ab Initio Molecular Electronic Structure on Parallel Computers*, Annu. Rev. Phys. Chem. 45: 623-58, 1994.