EXPLICIT MANAGEMENT OF MEMORY HIERARCHY

Jarek Nieplocha   Robert Harrison
Pacific Northwest National Laboratory
Richland, WA 99352, USA
<j_nieplocha,rj_harrison@pnl.gov>

Ian Foster
Argonne National Laboratory
Argonne, IL 60439, USA
*<itf@mcs.anl.gov>

Abstract

All scalable parallel computers feature a memory hierarchy, in which some locations are "closer" to a particular processor than others. The hardware in a particular system may support a shared memory or message passing programming model, but these factors effect only the relative costs of local and remote accesses, not the system's fundamental Non-Uniform Memory Access (NUMA) characteristics. Yet while the efficient management of memory hierarchies is fundamental to high performance in scientific computing, existing parallel languages and tools provide only limited support for this management task. Recognizing this deficiency, we propose abstractions and programming tools that can facilitate the explicit management of memory hierarchies by the programmer, and hence the efficient programming of scalable parallel computers. The abstractions comprise local arrays, global (distributed) arrays, and disk resident arrays located on secondary storage. The tools comprise the Global Arrays library, which supports the transfer of data between local and global arrays, and the Disk Resident Arrays (DRA) library, for transferring data between global and disk resident arrays. We describe the shared memory NUMA model implemented in the tools, discuss extensions for wide area computing environments, and review major applications of the tools, which currently total over one million lines of code.

1 Introduction

Chuck Seitz, a pioneer in parallel computing, named parallelism and memory as the two fundamental issues in computer architecture [1]. However, the programming languages and tools developed for parallel scientific programming during the last two decades have focused primarily on parallelism, and more specifically on the issues of control flow, communication structures, and load balancing. Memory-related issues such as data locality have been largely ignored.

Nevertheless, while issues relating to parallelism are important, the exploitation of data locality and the effective use of memory hierarchy are critical to efficient parallel execution. As Van der Velde [2] indicates: “While control flow issues are certainly present, they are relatively straightforward in concurrent scientific computing. The more difficult issue by far is data locality, the key to obtaining high-performance programs. Current computer languages offer little or no support for introducing data distribution that promote locality. This is the primary reason why scientific programs are messy.”

Virtually all scalable architectures possess nonuniform memory access characteristics that reflect their multi-level memory hierarchies. These hierarchies typically comprise processor registers, multiple levels of cache, local memory, and remote memory. In future systems, both the number of levels and the cost (in processor cycles) of accessing deeper levels can be expected to increase [3,4].

Many parallel languages and tools assume a flat memory model, in which the physical location of data is not represented explicitly, for instance Fortran-90, PCN [5], or Data Parallel C [6] are examples of such languages. However, this simple memory model hinders the understanding and tuning of parallel program performance. The variable cost of accessing data that resides at different memory levels cannot be ignored if we are to construct efficient parallel programs. The systems that provide some support for management of locality include High Performance Fortran (HPF) [7], the Message Passing Interface (MPI) [8], Split-C [9], and Compositional C++ (CC++) [10]. HPF augments Fortran 90’s flat memory model with data distribution directives, which can provide an implicit specification of locality. However, HPF can specify only data-parallel algorithms, and requires advanced compiler technology. MPI distinguishes local and remote memory explicitly, but requires that remote memory access be performed by means of explicit send calls with matching receives. Split-C and CC++ use global pointers to represent references to remote data.

In this paper we discuss the issues of memory hierarchy and data locality in common programming models and describe a set of tools for development of scalable scientific MIMD algorithms that address these issues. The tools have been designed based on the assumption that current as well as future-generation scalable architectures will possess NUMA characteristics, regardless of the programming paradigm(s) supported directly by the hardware such as message-passing or shared-memory. Therefore, the primary focus has been on the development of abstractions for the management and transfer of data between different layers of the NUMA memory hierarchy, namely local memory, remote memory, and high-performance secondary storage. The abstractions comprise local arrays, global (distributed) arrays, and disk resident arrays located on secondary storage. The tools comprise the Global Arrays library, which supports the transfer of data between local and global arrays, and the Disk Resident Arrays (DRA) library, for transferring data between global and disk resident arrays. They expose to the programmer the NUMA characteristics of modern high-performance computer systems, and by recognizing the communication overhead for remote data transfer, promote data reuse and locality of reference.

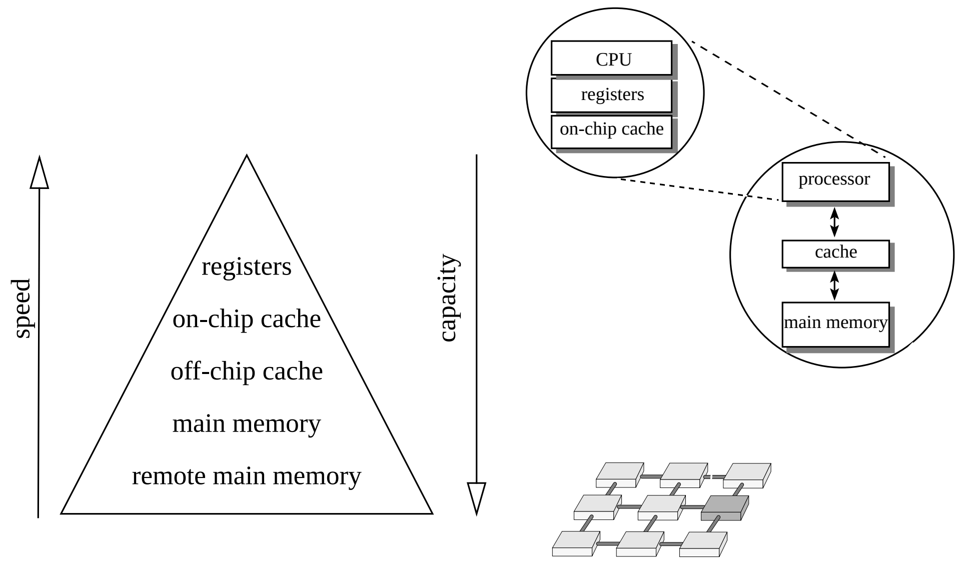


Figure 1: NUMA memory hierarchy in a scalable MPP system.

2 Memory Hierarchy

Hierarchical memory structures appear in modern computers as a result of economic and physical constraints. Faster memory components tend to be less dense and hence more expensive than slower components. For example, for comparable fabrication technology, DRAM chips have 16 times more capacity (density) than SRAM chips, but a cycle time that is 8 to 16 times slower [3]. Capacity and access time both increase yet again when we consider secondary storage (disk). Hence, system designers make cost-performance trade-offs by using the faster but more expensive SRAM memory to cache frequently used values, DRAM for slower and much larger main memory, and disk for virtual memory. Because access time is also correlated with physical distance, due to fundamental laws of physics (the speed of light), faster components are placed nearer the processor; this technique accentuates the phenomenon that access time depends on the distance of the memory component from the processor. The result is the typical workstation memory hierarchy, comprising registers, on-chip cache, off-chip cache, main memory, and virtual memory. In massively parallel processing (MPP) systems, memory located on other processors introduces one or more additional layers in the hierarchy, producing what is called the Non-Uniform Memory Access architecture (Figure 1).

The effectiveness of the NUMA architecture depends critically on the principle of locality of reference which says that programs tend to reuse data and instructions that they have most recently used. In an ideal situation, this principle means that the data and instructions required by the processor are almost always located in the small amount of fast memory (cache) that the designer places nearby the processor. Hence, a computer in which most memory has slow memory costs can operate at fast memory speeds. However, if locality is not achieved, we can encounter situations in which many memory references require that data be moved into cache from main memory--or even from secondary storage in the case of virtual memory. Performance is then seriously degraded.

In an attempt to maximize locality of reference, and hence performance, various algorithms and compilers have been developed that seek to optimize memory hierarchy usage. A classic technique is to access data in blocks small enough to fit in the cache or main memory. If an algorithm makes sufficient use of the data contained in these blocks, data movement costs are then justified.

As we indicated above, MPP systems extend the NUMA memory hierarchy by introducing the concept of “remote” memory: memory associated with another processor. Physical distance means that this memory takes longer to access that “local” memory, regardless of the hardware mechanisms which are used to perform remote memory access. 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. Even so-called 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. This hardware support allow programs to specify remote memory accesses with the same load and store operations used for local memory accesses. However, this uniform mechanism for accessing both local and remote memory is only a programming convenience--on both shared and distributed memory scalable computers, the cost of remote memory access is significantly higher than for local memory, and therefore must be incorporated into performance models and taken into account when developing scalable applications.

If we think about the programming of MIMD parallel computers (either shared or distributed memory) in terms of management of NUMA memory hierarchy, 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 performance of our parallel algorithms (i.e., memory latency, bandwidth, data and reference locality), we also conceptually unite sequential and parallel computation.

3 Programming Models

The two predominant programming models for MIMD concurrent computing are message passing and shared memory. Message passing, which has roots in the CSP model [11], has been implemented in many flavors over the last two decades and recently standardized in the Message Passing Interface (MPI) [8]. One-sided communication is an extension of this model that has proved useful in some irregular problems. Shared memory has been implemented in many forms both in hardware and software [12]. High Performance Fortran (HPF) [13] is an emerging standard for developing data-parallel codes, and can be thought of as an implicit, compiler-based approach to shared memory programming. We now review differences between these models with respect to data locality, ease of use, and performance issues.

3.1 MESSAGE PASSING

Message passing assumes a distributed-memory model in which distinct processes each have their own "local" data, and share data only through cooperative communication. A process can access its own local data directly, but access of remote data requires the cooperation of the process that owns the data. The remote process must send the required data in an explicit message, and hence must know which piece of data is needed by which process and when. This requirement makes the message passing model hard to use for irregular problems and applications that use dynamic load balancing. This is because the coordination of large number of processes that operate on uneven chunks of data or that require access to remote data at irregular time intervals increases algorithmic complexity and magnifies associated programming effort.

A message passing operation not only transfers data but also synchronizes sender and receiver. Asynchronous (nonblocking) send/receive operations can be used to diffuse the synchronization point, but cooperation between sender and receiver is still required. The synchronization effect is beneficial in certain classes of algorithms such as parallel linear algebra where data transfer usually indicates completion of some computational phase; in these algorithms, the synchronizing messages can often carry the results produced in the preceding computational phase. For other algorithms, synchronization constitutes an unnecessary and undesirable effect and a source of performance degradation [14]. The time that each process wastes waiting for a rendezvous can be readily minimized in certain regular problems, by performing other computation; however, in other problems, this optimization effort can require extensive programming effort and complexity that usually compromises code clarity and ultimately increases software maintenance costs.

Despite programming difficulties, the message-passing paradigm's memory model maps well to the distributed-memory architectures used in scalable MPP systems. Because the programmer must explicitly control data distribution and is required to address data locality issues, message-passing applications tend to execute efficiently on such systems. However, on systems with multiple levels of remote memory, for example networks of SMP workstations or metacomputers, the message-passing model's classification of main memory as local or remote can be inadequate. For example, on SMP workstations connected with Ethernet, message-passing latency can vary by two to three orders of magnitude according to whether processes are on the same or different machines. While some algorithms are capable of exploiting data locality at different levels of remote memory by decomposing the data in a fashion that minimizes associated cost of communication across the network, unfortunately, most message-passing libraries, including MPI, do not provide locality information about process mapping to the hardware and associated variable data transfer cost.

3.2 SHARED MEMORY

In the shared-memory programming model, data is located either in “private” memory (accessible only by a specific process) or in “global” memory (accessible to all processes). In some shared-memory systems, global memory is accessed in the same manner as local memory. Systems based on this approach may rely on hardware or operating support to recognize load and store operations that reference non-local memory (e.g., KSR-2, Convex-SPP) or use purely software-based approaches, as in the various distributed shared memory libraries, for example Treadmarks [15] or Midway [16]. In other shared-memory systems, global memory is accessed by using distinguished mechanisms, such as language constructs [17,9,10], special user-defined operations [18], or library functions [19,20]. Regardless of the implementation, the shared-memory paradigm eliminates the synchronization that is required when message passing is used to access shared data.

A disadvantage of many shared-memory models is that they do not expose the NUMA memory hierarchy of the underlying distributed-memory hardware. Instead, they present a flat view of memory making it hard for programmers to understand how data access patterns effect the application performance or to exploit data locality. Hence, while programming effort involved in application development tends to be much lower than in the message-passing approach, achieved performance is usually less competitive.These shortcomings are not uncommon among vendors of shared-memory hardware. For example, KSR provided no software support to distinguish between memory subpages located on the same and different rings despite significant latency and bandwidth differences [21]. As a notable exception, the Convex-SPP supports a view of *near shared memory* (within the hypernode) and *far shared memory* (remote hypernode) which better reflects the memory hierarchy of the system [22].

3.3 HIGH PERFORMANCE FORTRAN

High Performance Fortran (HPF) [13] represents an alternative approach to data distribution and data locality management, in which these attributes of a parallel program are specified implicitly, via compiler directives. HPF programs use Fortran 90 array notation and other statements (e.g., FORALL statements and INDEPENDENT directives) to specify data-parallel operations. Data distribution is specified separately, via directives that describe array decomposition (e.g., DISTRIBUTED, BLOCK, CYCLIC) and alignment (e.g., ALIGN, REALIGN). HPF compilers use this data distribution information to determine placement of data and computation, and hence the data locality properties of the program.

HPF provides a particularly high-level programming model which, when effective, can simplify parallel programming. However, sophisticated compiler technology is required to generate efficient programs. In addition, the range of problems that can be expressed in HPF is restricted by the lack of support for certain features required in dynamic, task-parallel programs, such as random access to regions of distributed arrays from within a MIMD parallel subroutine call-tree, and reduction into overlapping regions of distributed arrays. We also note that HPF does not address explicitly the problem of data placement and problem decomposition in more complex memory hierarchies, such as clusters of SMPs.

3.4 ONE-SIDED COMMUNICATION

Traditional message passing is insufficient for many applications whose communication patterns and work distribution are determined dynamically at run-time. These applications are better supported by a one-sided communication model in which processes can access remote data without the explicit cooperation of processes that own that data. One-sided communication assumes that a process can access data on a remote node

- asynchronously,
- without explicit cooperation of the process on the remote node, and
- with latency and overhead costs that are comparable to standard send and receive operations.

One-sided communication can be seen as an extension of the message-passing model in which only one side specifies the communication parameters usually present in point-to-point message passing operations, including origin, target process, number of bytes, and memory addresses for origin and target locations. The distributed memory view of this model maps very well to the memory hierarchy of MPPs. While the high-performance implementations of this model are becoming available, the MPI Forum is currently working on standardizing the one-sided communication interface to be included in MPI-2 [23].

4 Tools For Explicit Management of Memory Hierarchy

In an attempt to merge better features of message passing and shared memory models we developed a set of tools that provide explicit support for management of memory hierarchy and provide one-sided access to distributed data structures. The development of these tools have been motivated by algorithmic requirements of theoretical chemistry electronic structure computations (dynamic load balancing with up to three orders of magnitude variation in task size, blocked access patterns to distributed dense arrays) and NUMA characteristics of high-performance computers [19, 24]. The Global Arrays toolkit implements shared-memory NUMA model for distributed dense arrays. Disk Resident Arrays [25] and Mirrored Arrays [26] are extensions of this model to the secondary storage and metacomputing environments both of which are characterized by the increased latency and reduced bandwidth.

In general, the shared-memory NUMA model merges features of other existing models:

- a distributed memory view of the message passing model,
- a one-sided access to remote data in the spirit of the shared memory paradigm,
- an explicit control over data distribution,
- a data locality, distribution and mapping information,

- recognition of memory hierarchy and performance differences in access to distinct layers in memory hierarchy, and
- includes as a subset message passing (for example to support algorithms that require synchronization on data transfer).

4.1 GLOBAL ARRAYS

The Global Arrays library [19,24] implements a shared-memory programming model in which data locality is managed explicitly by the programmer. This management is achieved by explicit calls to functions that transfer data between a global address space (a distributed array) and local storage. In this respect, the GA model has similarities to distributed shared-memory models that provide an explicit acquire/release protocol [16,17]. However, the GA model acknowledges that remote data is slower to access than local data and allows data locality to be explicitly specified and hence managed. The GA model exposes to the programmer the NUMA characteristics of modern high-performance computer systems, and by recognizing the communication overhead for remote data transfer, it promotes data reuse and locality of reference. The GA library allows each process in a MIMD parallel program to access, asynchronously, logical blocks of physically distributed matrices, without the need for explicit cooperation by other processes. This functionality has proved useful in numerous computational chemistry applications, and today many programs, totaling over one million lines of code, make use of GA with NWChem [27] alone exceeding 400,000 lines. One of the primary design goals for GA was to make the development of scalable applications easier. The applications experience indicates that the toolkit meets these expectations thanks to its high-level array-oriented interface combined with the one-sided access to the shared data and locality management features.

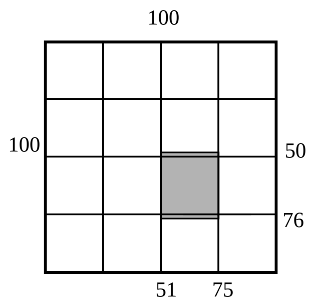
The GA provides extensive support for controlling array distribution and accessing locality information. Global arrays can be created by:

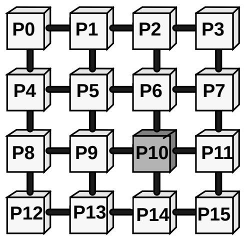
- allowing the library to determine array distribution,
- specifying decomposition only for one array dimension and allowing the library to determine the others,
- specifying the distribution block size for all dimensions, and
- specifying irregular distribution as a cartesian product of irregular distributions for each axis.

The distribution and locality information is available through library operations that:

- specify the array section held by a given process,
- specify which process owns a particular array element, and
- return list of processes and blocks of data for the given section of an array (see the following example and Figure2).

In the following Fortran code fragment, a two-dimensional array of integers (type handle *MT_INT*) is created by specifying array dimensions 100×100, array name "*array A*", and requesting regular distribution in both dimensions (block size "-1"). The array handle needed for future references to the array is returned in variable *g_a*. After an array





$$plist = \begin{array}{cl} \begin{bmatrix} 50 & 51 & 76 \\ 50 & 75 & 76 \\ 51 & 51 & 51 \\ 75 & 75 & 75 \\ 6 & 10 & 14 \end{bmatrix} & \begin{array}{l} \textit{ilo} \\ \textit{ihi} \\ \textit{jlo} \\ \textit{jhi} \\ \textit{process} \end{array} \end{array}$$

Figure 2: Example of a regular distribution of an 100x100 array for 16 processes. The array *plist* contains results of the *ga_locate_region* corresponding to the shaded section and returned for process *P10*.

is created, each process determines the coordinates of the array section it holds, described in Fortran 90 notation as (*ilo*:*ihi*, *jlo*:*jhi*), by calling *ga_distribution*. Next, with the *ga_locate_region* operation evenly numbered processes inquire distribution information for a section of array that augments the section they own from North and South while odd numbered processes do the same for the section augmented from East and West. The variable *np* returns the number of processes that hold data in the specified section and array *plist* returns process Id and the corresponding subsection coordinates, see Figure 2.

```
  integer g_a, me, ilo, ihi, jlo, jhi, n, np, plist(1:5,*)
  parameter (n=100)
c
  call ga_create(MT_INT, n, n,‘array A’, -1, -1, g_a)
  me = ga_nodeid()
  call ga_decomposition(g_a, me, ilo, ihi, jlo, jhi)
  if(Mod(me,2) .eq.0) then
    call ga_locate_region(g_a,MAX(ilo-1,1),MIN(ihi+1,n), jlo,jhi,plist, np)
  else
    call ga_locate_region(g_a,ilo,ihi,MAX(jlo-1,1),MIN(jhi+1,n), plist,np)
  endif
```

The primary mechanism provided by GA for accessing data are copy operations that transfer data between layers of memory hierarchy, namely global memory (distributed array) and local memory. In addition, each process is able to access directly data held in a section of a global array that is assigned to that process. Atomic operations are provided that can be used to implement synchronization and assure correctness of an accumulate operation (floating-point sum reduction that combines local and remote data) executed concurrently by multiple processes and targeting overlapping array sections.

4.2 MIRRORED ARRAYS: METACOMPUTING EXTENSIONS TO GA

In recent years there has been increasing interest in metacomputing. Metacomputing environments comprise multiple supercomputers and other devices (mass storage systems, display devices) connected via wide area networks (WANs). Important to the usability of such systems is an integrated software environment that allows these resources to be treated, to some extent at least, as a single virtual system [28]. One example of such an environment was the I-WAY system constructed for the Supercomputing 95 conference [29].

Metacomputing is interesting since it can potentially provide increases in the computational power accessible to an individual user. From the memory hierarchy perspective, the metacomputer environment provides one or more additional layers of memory, with typically much higher latencies and lower bandwidth than in a supercomputer. Conceptually, this environment is similar to the clustered network of multiprocessor workstations. The main differences arise from the ratios of floating-point performance of nodes and network performance. There are three major performance factors to consider when analyzing the effectiveness of network high-performance computing:

- Floating point performance of a network compute node: tens to hundreds of GFLOPS (aggregate for all processors sharing the same network connection) for a supercomputer connected to a WAN vs. tens to hundreds of MFLOPS for a typical workstation.
- Network bandwidth: the LAN supports rates from a fraction of a MB/s to tens of MB/s (e.g., Myrinet network [30]); in contrast common WAN technologies provide tens of KB/s to a few MB/s. Even more seriously, the bisection bandwidth of a WAN-connected supercomputer will typically be much lower than that of a LAN-connected supercomputer: while technologies such as Myrinet provide a crossbar, in a WAN multiple (hundreds and even thousands) processors typically have to share the bandwidth of a single link, making the effective network bandwidth per processor orders of magnitude lower than for tightly coupled clusters of workstations.
- Network latency: in the LAN environment, latency can range from a few tens of microseconds for highly integrated systems (e.g., Fast Messages on the Myrinet network [30] or Active Messages on the ATM network [31]) to a few milliseconds for TCP/IP over Ethernet. In contrast latencies of tens to hundreds of milliseconds were not unusual for the I-WAY WAN. While WAN bandwidth can be expected to improve significantly with advanced network technologies, the latency has physical limits that are currently being approached. For example, 12 milliseconds is required

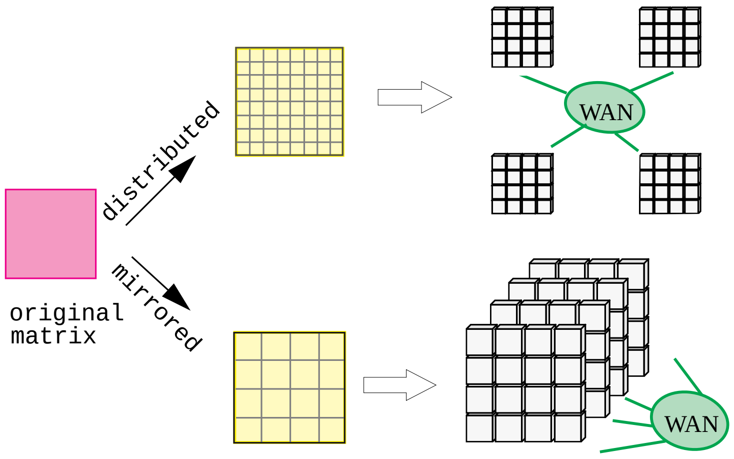


Figure 3: Distribution and mapping of a matrix using standard fully-distributed and mirrored aproach

for light to cover the distance between the East and West coast of North America. For a 200 MFLOPS processor, this time is equivalent to 2,400,000 floating point instructions.

In summary, the relative cost of access to remote memory (weighted with floating-point performance) is much higher for an I-WAY metacomputer than for clusters of workstations. Despite progress in network technology, the high latencies are and will continue to discourage frequent transfers of small amounts of data over the wide area networks.

In spite of the communication limitations of WAN computing, there are some classes of applications that can take advantage of the potentially enormous computational power of multiple supercomputers. A number of techniques can be used to program these applications. For example, applications can asynchronously prefetch data while computations take place. However, this approach requires significant restructuring of the code and is applicable only to some algorithms. It is important to consider higher-level approaches that do not require such explicit code restructuring.

In the GA NUMA programming model, an alternative approach to programming metacomputing applications is possible based on what we call mirrored arrays. Mirrored arrays are replicated in each WAN-connected supercomputer (see Figure 3). Arrays are fully distributed within each machine, so the amount of data held by each processor increases by a factor roughly proportional to the number of networked supercomputers. Each supercomputer operates on its own mirrored array independently, and a *ga_net_merge* primitive is provided for enforcing consistency of the different mirrored arrays. This primitive is a collective operation across all supercomputers and merges entire or user-specified sections of the mirrored arrays. Upon completion of this operation, all machines have identical copies of the specified array or array section. Mirrored arrays can be seen as a cache for WAN-remote memory with the *write-back* policy. This means that an update is not propagated to the lower-level memory immediately (unlike in the *write-through* protocol) but only when needed.

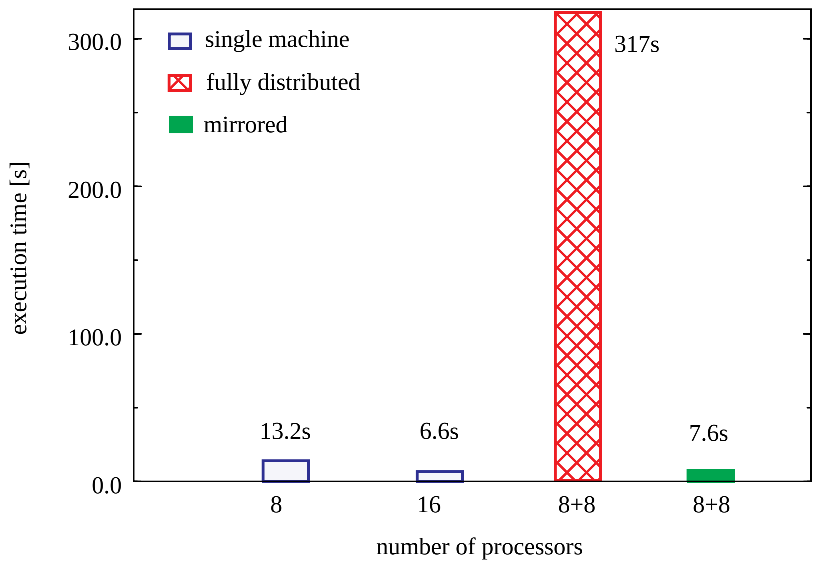


Figure 4: Execution times for SCF on a single Paragon and two Paragons connected with WAN

Mirrored arrays work in the GA framework for applications that are structured to access data in blocks (to reduce sensitivity to latency) and do not have to read and write to the same array in the same computational phase (any other part of the algorithm can be executed in the replicated fashion). For example, processes can read components of one distributed array and update another array; at the end of this computational phase processes enforce consistency of the updated arrays with *ga_net_merge*. The result is that most GA communication occurs locally within supercomputers. Total network traffic is lower and average message size sent across the network is larger than in the fully distributed approach. This approach can be effective if a portion of the algorithm that cannot use mirrored arrays (or other technique that reduces sensitivity to WAN latency) does not dominate the computations.

This use of mirrored arrays is similar in spirit to replicated shared memory, which minimizes latency in access to shared data by maintaining fully replicated copies of the data at each node [32]. Data consistency is assured by broadcasting modified pages to all the nodes. In contrast to replicated shared memory, mirrored arrays make the programmer responsible for enforcing consistency.

To evaluate the utility of mirrored arrays, we used a large computational chemistry application self-consistent field (SCF), implemented on top of GA, and measured its performance on the I-WAY metacomputer. Our experiments used two Intel Paragons located at San Diego Supercomputer Center and at the California Institute of Technology in Pasadena, connected with a wide-area network (70 Kbytes/s bandwidth and 35 ms latency). Two versions of Global Arrays were used: the standard fully-distributed version and the version with metacomputing extensions. In the standard version, the fact that some memory was WAN remote was not exposed to the application. In the second version, the application was presented with two layers of remote memory -- one available within each machine and the other available through the WAN and supported through mirrored arrays. Figure 4 shows the results obtained in four different system configurations: 8 and 16 nodes on a single Paragon, and 8+8 nodes split between two Paragons, using the two versions of the program. We see that the performance penalty for ignoring the memory hierarchy is tremendous in the standard fully-distributed version; this penalty prevented us from executing the application for any but a very small problem size. In contrast, the mirrored array version of SCF performed well, almost as well as the single machine version.

4.3 DISK RESIDENT ARRAYS

Disk Resident Arrays (DRA) extend the GA model to another level in the storage hierarchy, namely, secondary storage. DRA introduces the concept of a disk resident array--a disk-based representation of an array--and provides functions for transferring blocks of data between global arrays and disk resident arrays. Hence, it allows programmers to access data located on disk via a simple interface expressed in terms of arrays rather than files. This extends the benefits of global arrays (in particular, the absence of complex index calculations and the use of optimized, blocked communication) to programs that operate on arrays that are too large to fit into memory.

By providing distinct interfaces for accessing objects located in main memory and on the disk, GA and DRA render visible the different levels of the memory hierarchy in which objects are stored. Hence, programs can take advantage of the performance characteristics associated with access to these levels. Recall that memory hierarchies consist of multiple levels, but are managed between two adjacent levels at a time [3]. For example, a page fault causes the transfer of a data block (page) to main memory while a cache miss transfers a cache line. Similarly, GA and DRA allow data transfer only between adjacent levels of memory. In particular, data transfer between disk resident arrays and local memory is not supported. Since we would be jumping between nonadjacent levels in the NUMA hierarchy, performance is expected to be disappointing, and portable implementations problematic.

DRA read and write operations can be applied both to entire arrays and to sections of arrays (disk and/or global arrays); in either case, they are collective and asynchronous. The focus on collective operations within the DRA library is justified as follows. Disk resident arrays and global arrays are both large, collectively created objects. Transfers between two such objects seem to call for collective, cooperative decisions. (In effect, we are paging global memory.) We note that the same model has proved successful in the GA library: operations that move data between two global memory locations are collective while transfer between global and local memory is noncollective. This collective I/O strategy has been adopted in many other projects for similar reasons [33, 34, 35]. A DRA asynchronous interface to its I/O operation permits applications to overlap time-consuming I/O with computation.

5 Conclusions and Future Work

We have described techniques and tools that support explicit but high-level management of data movement in memory hierarchies. The GA library is a tool for transferring data between local memory and global (distributed memory); the DRA library extends the GA model to allow data transfer from global memory to secondary storage. The application experience with the GA and DRA libraries demonstrates that these tools can simplify significantly application development and at the same time, by exposing to the programmer NUMA memory hierarchy and promoting data locality, can be instrumental in development of highly scalable algorithms [27, 24].

Our experiences with GA and DRA suggest several directions for future work. The capabilities of the GA and DRA libraries themselves can be extended, for example to support data structures other than dense arrays. A long-term direction might be to extend the current three-level model to support the additional memory hierarchy layers that can be expected in future highly parallel architectures. For example, in a system comprising clusters of distributed memory or shared memory nodes, we can imagine extending GA operations that return distribution information (such as *ga_locate_region* or *ga_distribution*) with some numeric value that represents relative "distance" from the calling process. Alternatively, we could provide additional data movement operations that transfer data from "global" memory to "cluster" memory, and then to "local" memory.

6 Acknowledgments

Work at PNNL was performed under the auspices of the High Performance Computing and Communications Program of the Office of Computational and Technology Research, U.S. Department of Energy under contract DE-AC066-76RLO1830 with Battelle Memorial Institute which operates the Pacific Northwest National Laboratory. Work at ANL was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the Scalable I/O Initiative, a multi-agency project funded by ARPA, DOE, NASA, and NSF.

7 References

[1] C. Seitz. High-performance workstations + high-speed interconnect ≥ multicomputers. In *Scalable Parallel Libraries Conf.*, Missisippi State, 1993.

[2] E.F. Van der Velde. Book review on 'Studies in Computational Science: Parallel Programming Paradigms'. *IEEE Computational Science and Engineering*, 2(4):85–87, 1995.

[3] D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[4] T. Sterling, P. Messina, and P. Smith. *Enabling Technologies for Petaflops Software*. MIT Press, 1995.

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

[6] P. Hatcher and M. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991.

[7] C. Koelbel, D. Loveman, R. Schreiber, G. S. Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[8] Message Passing Interface Forum. *MPI: A Message-Passing Interface*. University of Tennessee, Knoxville, Ten., May 5, 1994.

[9] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, T. v. E. S. Lumetta, and K. Yelick. Parallel programming in Split-C. In *Proc. Supercomputing'93*, pages 262–273. ACM Press, 1993.

[10] K. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming notation. In *Research Directions in Object Oriented Programming*. The MIT Press, Cambridge, MA, 1993.

[11] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[12] M. Stumm and S. Zhou. Algorithms for implementing distributed shared memory. *IEEE Computer*, 24(5):54–64, 1990.

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

[14] H. Kung. Synchronized and asynchronous parallel algorithms for multiprocessors. In J. Traub, editor, *Algorithms and Complexity*, pages 153–200. Academic Press, 1976.

[15] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[16] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared memory system. In *Proc. '93 CompCon Conference*, pages 528–537, 1993.

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

[18] H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Software Eng.*, 18(3):190–205, 1992.

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

[20] E. D'Azevedo and C. Romine. DOLIB: Distributed object library. Technical Report ORNL/TM-12744, Oak Ridge National Lab., Oak Ridge, TN, 1994.

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

[22] Convex Computer Corp. *Exemplar SPP1000/1200 Architecture*. Convex Computer Corp., Richardson, Tex., 1995.

[23] MPI Forum. MPI-2. information available from http://www.mcs.anl.gov/mpi.

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

[25] J. Nieplocha and I. Foster. Disk Resident Arrays: An array-oriented I/O library for out-of-core computations. In *Proceedings of Frontiers of Massively Parallel Computation*. IEEE Computer Society Press, 1996.

[26] J. Nieplocha and R. Harrison. Shared-memory NUMA programming on I-WAY. In *Proc. of IEEE HPDC-5*, pages 432–441. IEEE Computer Society Press, 1996.

[27] D. Bernholdt et al. Parallel computational chemistry made easier: The development of NWChem. *Intl J. Quantum Chem. Symp.*, 29:475–483, 1995.

[28] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. In *Proc. 3rd Workshop on Environments and Tools for Parallel Scientific Computing*. SIAM, 1996. to appear; see also http://www.globus.org/.

[29] T. DeFanti, I. Foster, M. Papka, R. Stevens, and T. Kuhfuss. Overview of the I-WAY: Wide area visual supercomputing. *Int. J. Supercomputing Applications*, 10(2):123–130, 1996.

[30] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proc. Supercomputing'95*, 1995.

[31] T. von Eicken, V. Avula, A. Basu, and V. Buch. Low-latency communication over ATM networks using active messages. *IEEE Micro*, 15(1):46–53, 1995.

[32] M. Oguchi, H. Aida, and T. Saito. A proposal for a DSM architecture suitable for a widely distributed environment and its evaluation. In *Proc. 4-th IEEE Int. Symp. HPDC*. IEEE CS Press, 1995.

[33] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, W. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–15, April 1995.

[34] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: Parallel and scalable software for input-output. Technical Report SCCS-636, NPAC, Syracuse, NY, 1994.

[35] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proc. Supercomputing '95*, December 1995.