

# Shared Memory Mirroring for Reducing Communication Overhead on Commodity Networks

Bruce Palmer

Jarek Nieplocha

Edoardo Aprà

Computational Sciences & Mathematics      Environmental Molecular Sciences Laboratory

Pacific Northwest National Laboratory, Richland, WA 99352, USA

*This paper proposes using shared memory for caching latency sensitive distributed data structures on Symmetric Multi-Processor nodes of clusters connected with commodity networks. Shared memory mirroring is a hybrid approach that replicates data across cluster nodes and distributes data within each node. The user is responsible for managing consistency of the data cached within the mirrored data structures. The method is shown to be very effective in improving the performance of a real scientific application on clusters equipped with Ethernet, Myrinet, or Quadrics networks.*

## 1. Introduction

There are multiple tradeoffs in building cost-effective clusters based on commodity components. Among the most fundamental decisions is selection of the network. Faster networks such as Myrinet, Quadrics, SCI, or Infiniband are substantially more costly than Ethernet (100- or even 1000 Mbit/s). Given a fixed budget, if Ethernet is selected more cluster nodes can be purchased, but the efficiency of applications will be compromised due to reduced bandwidth and increased latency in interprocessor communication. From the cost/performance perspective, analytical models have been proposed to provide guidance for cluster procurements [1]. For some applications, the performance degradations experienced on slower networks can be reduced using proven techniques for hiding latency, aggregating small messages, or reducing the volume of data communicated across the network (e.g., through data replication).

Overlapping communication and computation with the support of nonblocking communication is one of the most commonly used techniques for latency hiding at the application level. However, its effectiveness depends on the ability of the underlying communication layer (e.g., MPI) to make progress in the absence of explicit communication calls in the purely computational phase of the program execution [2,3]. Moreover, many algorithms offer only limited or no opportunities for issuing nonblocking communication calls if the computed data is immediately required on other processors (e.g., computing search direction vectors in the conjugate gradient method). The effectiveness of yet another technique for reducing sensitivity to high network latency, message

aggregation, is also limited to certain classes of algorithms where data is communicated by small messages that are not immediately required by the receiver.

In our previous work [4,5], mirroring of data structures was introduced as a mechanism for reducing sensitivity to the communication delays in grid computing environments. In that approach, all distributed data structures used by applications were effectively replicated on each of the systems connected to the wide area network, and distributed across the processors inside the system. All message passing was confined to each individual machine. The user was responsible for managing inconsistent data, primarily using a merge operation that could combine the data across the wide area network using the TCP/IP sockets. Although shown effective for some applications in the grid environments, to our best knowledge this technique has not been used beyond the proof of concept demonstration. Although the original mirroring approach is relevant to clusters, it would lead to inefficient use of resources (memory and network) when used unmodified in that environment.

In the current paper, we propose mirroring of only *selected* latency sensitive data structures as a latency hiding mechanism for clusters based on inexpensive commodity networks. The current paper describes an implementation and use of mirroring as a shared memory cache, shared between processors on SMP nodes. Shared memory is the fastest communication protocol available on clusters and reduces communication latency by one or even several orders of magnitude. Instead of applying mirroring to all distributed arrays, as in [4], the user can decide, depending on the nature of the algorithm and the communication requirements (number and size of messages), which arrays can or should use mirroring and which should be left fully distributed and accessed without the shared memory cache. The set of operations on mirrored arrays has been extended to provide increased opportunities for application optimization. Instead of using socket communication in merging arrays between supercomputers connected to a wide-area network, the native communication protocols can be used for all the communication and there is no

negative impact on performance of parts of the application that does not involve mirroring. Using the high performance network and communication protocols is important as the latency hiding techniques traditionally require increased network bandwidth [7]. Shared memory mirroring could be considered as one of the techniques for explicit management of memory hierarchy at the application level [6] and is also related to latency hiding techniques used in shared memory microprocessors such as software-controlled prefetching [8,9,10], bulk transfers [10], or weaker consistency models e.g., [11]. The primary difference is that shared memory mirroring is performed by the user in context of specific distributed data structures in the SMP cluster environments.

The idea of mirroring latency sensitive data structures is general and it can be used in applications based on MPI that rely on distributed arrays. However, here the benefit of this technique is shown in context of the Global Array (GA) toolkit [12] that supports an extensive set of operations on dense distributed arrays. These also include one-sided put/get communication. In addition to applications in other areas, GA has become de facto standard programming interface for quantum computational chemistry [20], most major scalable parallel packages either use it as the communication interface (NWChem, Molpro, Molcas, QChem, COLUMBUS, GAMESS-UK) or emulate a subset of its functionality (GAMESS-US).

The paper makes several contributions to the field. It discusses algorithmic and performance benefits and limitations of mirroring. It describes design issues involved in efficient implementation of this technique on SMP clusters. It shows that mirroring can greatly improve performance of a real scientific application on a cluster employing any of the three common networks: Ethernet, Myrinet, and Quadrics. For the density functional application investigated here, mirroring improved application performance by 72.6% on the 48 Itanium-2 processors connected with Ethernet.

This paper is organized as follows. Section 2 describes mirroring. Section 3 discusses design tradeoffs and implementation of this technique on the SMP clusters. Section 4 describes experimental results for communication operations on mirrored arrays, matrix multiplication benchmark, and a real scientific application. Finally, summary and conclusions are presented in Section 5.

## 2. Approach, Advantages and Limitations

Mirroring of latency sensitive data structures is meant to extend the range of scalability of midsized problems to larger numbers of processors when running on a cluster of SMP nodes with commodity networks. The goal is to achieve optimum per processor performance, even for large number of processors, by replicating data across

SMP nodes but distributing it within the SMP node. This hybrid approach is particularly useful for problems where it is important to solve a moderate sized problem many times, such as an *ab initio* molecular dynamics simulation of a moderate size molecule. A single calculation of the energy and forces that can be run in a few minutes may be suitable for a geometry optimization, where a few tens of calculations are required, but is still too long for a molecular dynamics trajectory, which can require tens of thousands of separate evaluations. For these problems, it is still important to push scalability to the point where single energy and force calculations can be performed on the order of seconds. Similar concerns exist for problems involving Monte Carlo sampling or sensitivity analysis where it is important to run calculations quickly so that many samples can be taken.

Many modern parallel computer configurations consist of clusters of SMP nodes, with individual nodes containing multiple processors. Communication within the SMP node can be achieved using shared memory, which is the fastest interprocessor communication mechanism available, while communication between nodes involves a network, which is typically (much) slower. In the case of a Beowulf cluster built with an Ethernet network, the difference between intranode and internode communication can vary by orders of magnitude. The goal of mirroring is to minimize the communication overhead that can occur in such systems by distributing copies of an array on each SMP node. Communication overhead is limited by confining operations within an SMP node to as great an extent as possible and then recovering the full result using a “merge” operation to combine data from individual arrays on each SMP node. The strategy is to use memory to offset communication losses and is particularly aimed at hardware configurations where there is fast communication within nodes but slow communication between nodes. Note, the merge operation is not required in all computations involving mirrored data. In principle, it is needed if the cached data is modified and the modification needs to become globally available.

Mirrored arrays differ from traditional replicated data schemes in two ways. First, mirrored arrays can be used in conjunction with distributed data and there are simple operations that support conversion back and forth from mirrored to distributed arrays. This allows developers maximum flexibility in incorporating mirrored arrays into their algorithms. Second, mirrored arrays are distributed within an SMP node (Figure 1). For systems with a large number of processors per node, e.g., 32 in the current generation IBM SP, this can result in significant distribution of the data. Even for systems with only 2 nodes per processor, this will result in an immediate savings of 50% over a conventional

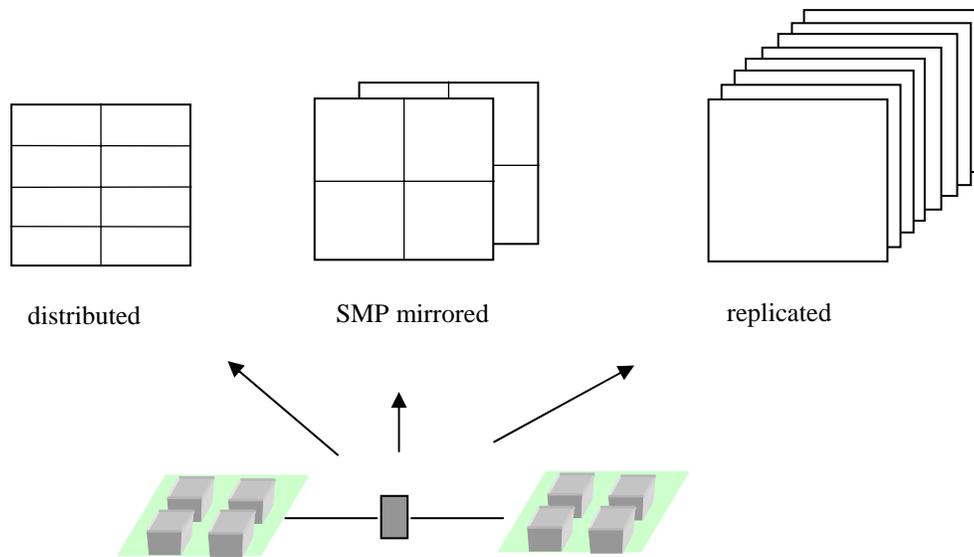


Figure 1: A two-dimensional array fully distributed, SMP mirrored, and replicated on two 4-way SMP cluster nodes

replicated data scheme. This can be partially offset by the fact that almost all array operations can be supported on both mirrored and distributed arrays, so that it is easy to develop code that can switch between using mirrored arrays and conventional distributed arrays, depending on problem size and the number of available processors.

### 3. Interfaces and Implementation

There are three significant components to implementing mirrored arrays. The first is the processor list that contains the mirrored array configuration, the second is incorporating recognition for this special type of array into a subset of functions that operate on arrays, and third is the addition of functions for merging mirrored arrays and an array copy that allows transfer of data between distributed and mirrored arrays. In our case, once this small group of functions has been adapted to properly deal with mirrored arrays, most of the remaining Global Array functions work with little or, in most cases, no modification. The processor list data structure is associated with every array, whether mirrored or not, and contains information about the mirrored array configuration. The most important elements of this data structure are maps between the local processor index on a node and the global processor index for the entire set of nodes. Currently only two processor lists are supported, the default processor list corresponding to a standard non-mirrored configuration of an array distributed across all processors in the system and the mirrored configuration containing a distributed copy of the array on each SMP node.

Most of the remaining work involves extending the routines that create distributed arrays to account for the processor list, modifying some core functions to take account of the array mirroring, when it is present, and

the creation of a merge operation that combines the contents of all mirrored arrays so that each copy is identical. The create functions must be modified so that the array is decomposed in blocks based on the number of processors on the SMP node, instead of the total number of processors for the job. The create functions also assign a processor list to each array. For the Global Arrays toolkit, the core task in implementing mirrored arrays was modifying all functions that locate data, or pointers to data, along with the basic communication operations. Examples of the higher level data location routines that need to be modified are functions that return the set of array indices representing the block of array data that is held by a particular processor and the call that returns the processor(s) that own either a particular piece of data or a block of data. For mirrored arrays, these functions only return processors that are on the same SMP node, even though the data is duplicated over many nodes. There are also functions that return pointers to the location in memory where a particular chunk of data begins. Once the modifications to these data location functions are complete, the communication operations work correctly on mirrored data.

To convert between mirrored and distributed arrays, the copy function was extended to handle the situation where one array is mirrored and the other is distributed. The copy operation is unambiguous if copying from a distributed array to a mirrored array, but copying from a mirrored array to a distributed array is only well-defined if all copies of the mirrored array are the same. It is up to the programmer to ensure that the mirrored arrays are all the same in this case. The copy from a distributed array to a mirrored array is accomplished by first zeroing out the mirrored array and then copying the

portion of the distributed array held on the SMP node to the corresponding portion of the mirrored array on the same node using shared memory copies. The full mirrored array can be recovered by then performing a merge operation. The copy from a mirrored array to a distributed array can be done very quickly, since only shared memory copies are required with no communication. Most of the other basic functions in the Global Array toolkit functioned for mirrored arrays without any alteration.

The only completely new functionality is the merge function, which does a sum of all instances of the mirrored array across all SMP nodes. This can be used to accumulate portions of a calculation that has been distributed across nodes into a single array. The merge operation uses the fact that if each node has the same number of processors, the data for each array will be laid out in the same way relative to the origin of the data. Since the data is distributed within each SMP node, there may be gaps in the data between the portions corresponding to the memory allocation of the array associated with each processor, but these gaps are reproduced on each SMP node. Furthermore, unless ghost cells are present the gaps typically are the size of

only a few array elements and are guaranteed not to contain any useful data. Because of this, the merge operation can be performed using the following sequence of operations:

Locate the origin of the data in memory on each SMP node.

Find the total length of the region containing the array data on each SMP node (including gaps).

Zero out the data in the gaps (This is to prevent accidental overflows due to uninitialized data in the gaps. The gaps do not contain useful data, so zeroing them is benign.)

Perform a standard global sum across SMP nodes with only one process per node participating in the operation explicitly (in  $O(\log(N))$  rather than  $O(\log(P))$  time, where  $N$  is the number of cluster nodes and  $P$  is the number of processors).

This method can make use of the heavy optimization associated with global sums and results in an operation that scales like the logarithm of the number of SMP nodes. This is illustrated schematically in Figure 2. For the case in which the number of processors on each SMP node are not equal (expected to be a rarity in

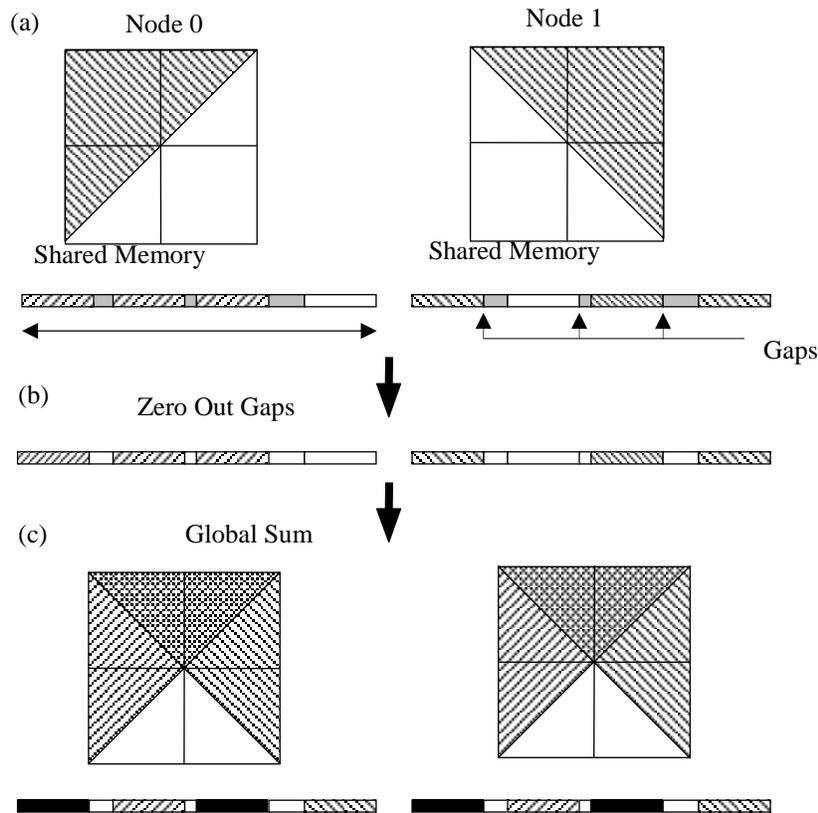


Figure 2. Schematic diagram of merge operation. (a) Origin of data in shared memory is located, size and location of gaps is determined, and total length of data calculated. (b) Data in gaps is zeroed out. (c) global sum of data across nodes is performed.

actual practice and only included to protect against unnecessary failures) the merge operation is implemented by creating a temporary distributed array. Each processor then takes the portion of the mirrored array that it holds locally and uses a put-accumulate (atomic reduction) operation to accumulate it into the appropriate part of the distributed array. After the accumulate operations are completed, each processor then grabs the portion of the temporary array corresponding to the locally held chunk and copies back to local memory. The temporary array is then destroyed and each copy of the mirrored array contains the merged data.

The interface for this operation is fairly simple and involves the extension/addition of only a few functions. The array creation functions were all extended to incorporate an additional parameter representing the processor list handle. Some additional functions were provided that return the handles for the mirrored processor list and the processor list for a standard distributed array. The type of array created then depends on the handle that is provided in the array creation call. The only other addition is the merge call that sums all copies of the mirrored array together.

#### 4. Experimental Evaluation

The SMP mirroring was evaluated using microbenchmarks to measure communication performance, a simple matrix multiplication kernel, and a computational chemistry application. The primary platform used in these experiments was a Linux cluster, based on HP RX2600 SMP nodes, each employing two 1GHz Itanium-2 processors. Three networks were available: 100Mbit/s Ethernet, Myrinet-2000, and Quadrics Elan-3. In addition, the scientific application was also tested on a 4-way Alphaserver cluster with the Quadrics Elan-3 network.

##### Communication Performance

We measured performance of basic one-sided communication operations on distributed and mirrored arrays. The table below presents the performance for the get operation. With mirroring the performance is independent of the network used. The get operation reduces to the shared memory copy (note that when comparing to the copy bandwidth reported by the STREAM benchmark for the HP RX2600 a factor 2 multiplier should be used). The latency corresponds to the overhead in the GA library that involves translation of array indices to the processor and memory location. Regarding the distributed case, the results correspond to performance of the `elan_get` operation (note that compatibility issues between the HP ZX1 chipset used in the RX2600 and the Elan-3 PCI card reduce bandwidth over what that card delivers on many IA32 systems). For Ethernet and Myrinet, the results in the

distributed case correspond to the performance of the ARMCI library that implements the missing get operation functionality using the client server approach [13,14].

Table 1: Performance of get operation

		Ethernet	Myrinet	Quadrics
latency [μs]	distributed	144	30.8	12
	mirrored	3.58		
bandwidth [MB/s]	distributed	11.7	219	225
	mirrored	1560		

In figures 3-5, performance of the merge operation on Ethernet, Myrinet, and Quadrics is presented as a function of the number of processors and data size. As expected, for fixed message size the cost grows logarithmically as function of the number of processors. Moreover, the performance is directly related to the speed of the network (Ethernet being the slowest among three).

##### Matrix Multiplication Example

A basic test of the mirrored arrays is to see if they can improve matrix multiplies on a cluster with a relatively slow network. A typical matrix multiply across many nodes will require on the order of  $P^{0.5}$  separate messages per node, while a mirrored matrix multiply requires on the order of  $\log P$  messages per node. If communication is slow relative to computation and memory is not a problem, it is expected that a considerable improvement in scalability can be achieved by replacing distributed matrix multiplies with mirrored matrix multiplies. If A, B, and C are mirrored arrays representing three matrices, then the matrix multiplication  $C = A \cdot B$  can be accomplished by dividing C between the SMP nodes such that each SMP node has an approximately equal

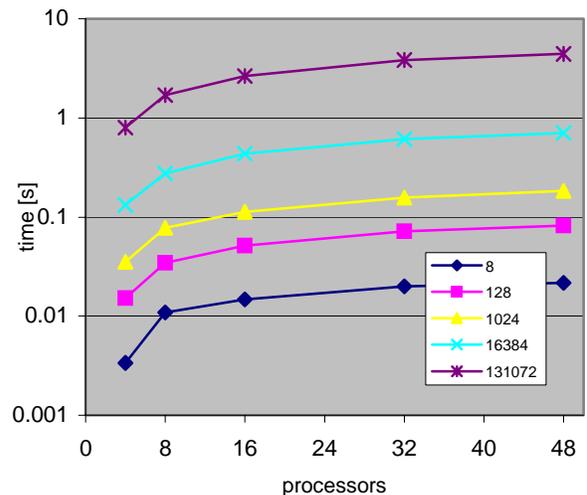


Figure 3: Cost of the merge operation for different message size on Ethernet

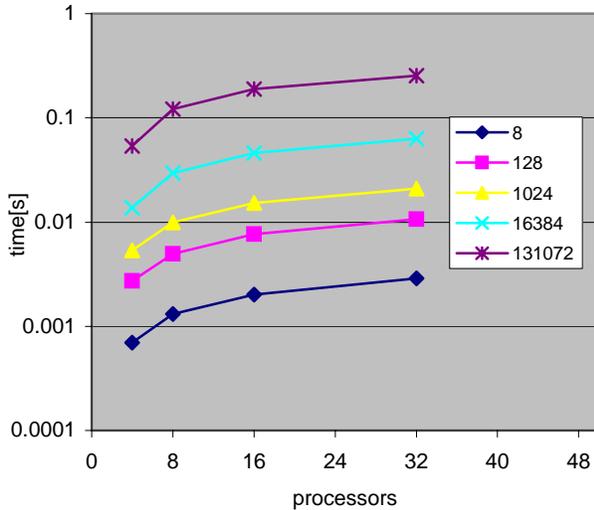


Figure 4: Cost of the merge operation for different message size on Myrinet

share and then doing the multiplications of the submatrices of A and B that yield the corresponding patch of C. Since A and B are duplicated on all the SMP nodes, this can be done with no communication. When the calculation of the local portion of C is completed on each node, the full matrix can be recovered by performing the merge operation. Note that within each node, the multiplication of the submatrices of A and B is distributed.

Results from tests of matrix multiplication of two square 1025x1025 matrices are shown in Figures 6-8 for the Ethernet, Myrinet, and Quadrics network, respectively. For the tests shown here, both mirrored and distributed matrix multiplies involve the same amount of computation, so differences in timings are mainly due to different communication volumes for the two algorithms. Results for the Ethernet network (Figure 6) show that for low numbers of processors, the mirrored arrays substantially outperform the distributed matrix multiply, although for this network neither system exhibits scaling. Given the characteristics of the matrix multiplication and the size of matrices used, it was not expected that 100Mbit/s Ethernet network would be sufficient to satisfy communication needs for a system with two 4GFLOP/s processors on each cluster node and thus achieve scaling.

On Ethernet, for larger numbers of processors, the performance of the distributed matrix multiply turns over and begins to decrease. By the time the system size reaches 32 processors both methods are about equal. The behavior of the distributed matrix multiply can be explained by noting that initially, there is a rapid increase in the number of large messages that must be sent so time required for the multiply increases rapidly. The increase in the number of message is offset at larger numbers of processors by the fact that the length of the

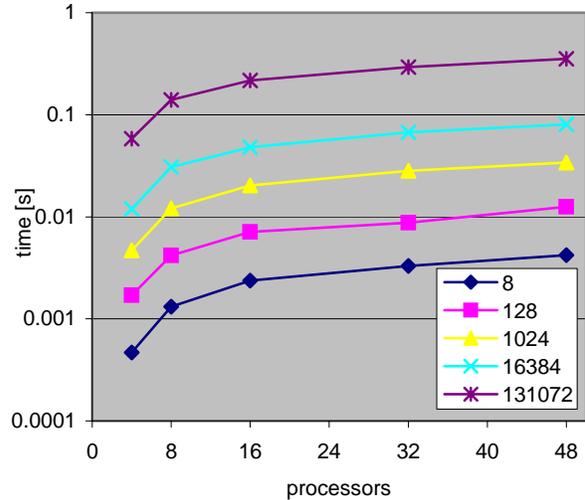


Figure 5: Cost of the merge operation for different message size on Quadrics

messages is decreasing so the time required per message is going down, which is enough to offset the increase in the number of messages. The mirrored matrix multiply shows a monotonic increase in the amount of time required to perform the calculation and this is all due to the logarithmic increase in the amount of time required to perform the merge operation. Unlike the distributed matrix multiply, the size of the messages in the merge operation remains fixed with increasing number of processors. Furthermore, it is clear that for a slow network such as Ethernet, almost all the time required for the matrix multiply is consumed by communication. These results indicate that the mirrored array algorithm may substantially improve performance on low to intermediate numbers of processors. Clearly, there is no point in using larger numbers of processors for the matrix multiply alone, but if it is imbedded in a code where it is a significant bottleneck, then using the mirrored arrays would improve performance over an intermediate range of processors.

The results for the Myrinet and Quadrics networks also show that the mirrored matrix multiply outperforms the distributed algorithm for low to intermediate numbers of processors. Both of these networks are substantially faster than Ethernet, so these results exhibit scaling. For Myrinet, the improvement of the mirrored algorithm is only marginally better than the distributed arrays for low numbers of processors and at 16 processors the two algorithms are substantially the same. At 32 processors, there appears to be another crossover in performance. Surprisingly, for Quadrics, which is the fastest network investigated, there is again a significant improvement in performance of the mirrored algorithm compared to the distributed algorithm. For this network, the mirrored array is faster even for 32 processors, which was the crossover point for the other two networks. These

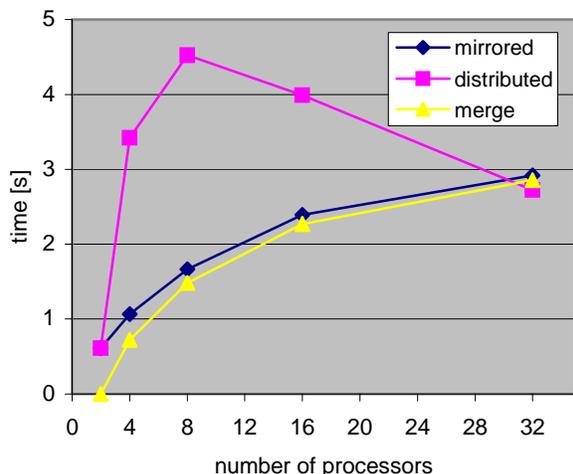


Figure 6: Matrix Multiplication on Ethernet using mirrored and distributed approach. In addition, time spent in the merge operation is shown.

results reinforce the conclusions reached for the Ethernet network, that the use of mirrored arrays can improve performance for intermediate numbers of processors.

### Scientific Application

In recent years, Density-Functional Theory (DFT) has become the most widely used electronic-structure method for calculating the properties of molecules. The mirrored arrays functionality has been implemented in the Gaussian function-based DFT module of NWChem [15,16]. More precisely, it has been implemented in the evaluation of the matrix representation of Exchange-Correlation (XC) potential on a numerical grid [17,18]. Prior to the current work, this quantity was evaluated using a distributed data approach, where the main arrays were distributed among the processing elements by using the GA library.

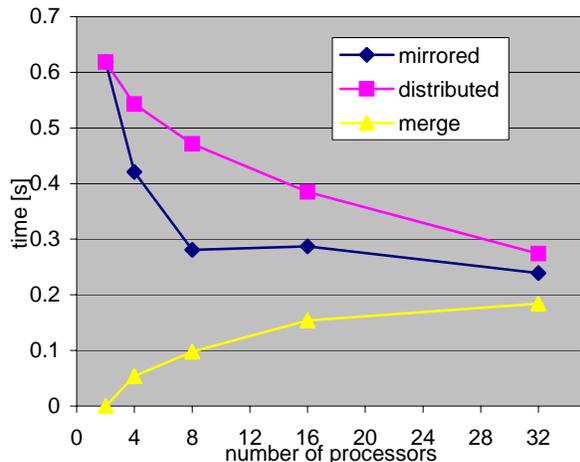


Figure 8: Matrix multiplication on Quadrics using mirrored and distributed approach. In addition, time spent in the merge operation is shown.

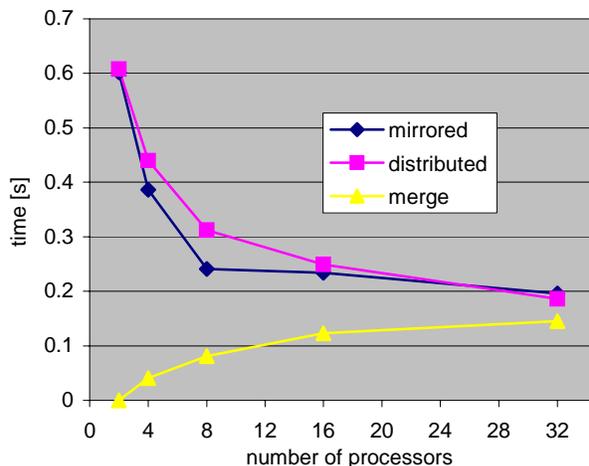


Figure 7: Matrix Multiplication on Myrinet using mirrored and distributed approach. In addition, time spent in the merge operation is shown.

This algorithm is very similar to the Hartree-Fock (a.k.a. SCF) algorithm, since both methods are characterized by the utilization of two main 2-dimensional arrays: *input* density matrix ( $\mathbf{D}$ ) and *output* Kohn-Sham ( $\mathbf{K}$ ) matrix. The major steps of this algorithm be summarized as follows:

- 1) Generate Density Matrix from a parallel matrix multiply into a distributed global array  $\mathbf{g\_DM}$
- 2) Read Density Matrix block from the array  $\mathbf{g\_DM}$  into local quantity  $D_{kl}$  using the *get* operation
- 3) Evaluate the density function  $\rho$  on the grid points  $\mathbf{x}_q$  by multiplying the density matrix with pairs of basis functions  $\chi$

$$\rho(\mathbf{x}_q) = \sum_{kl} D_{kl} \chi_k(\mathbf{x}_q) \chi_l(\mathbf{x}_q)$$

- 4) Evaluate the Exchange Correlation Potential  $V^{xc}[\rho(\mathbf{x}_q)]$  on the grid points  $\mathbf{x}_q$
- 5) Numerical integration combining  $V^{xc}[\rho(\mathbf{x}_q)]$  with the basis functions and the grid weights  $w_q$  to get the  $K_{ij}$  matrix element

$$K_{ij} = \sum_q w_q \chi_i(\mathbf{x}_q) V^{xc}[\rho(\mathbf{x}_q)] \chi_j(\mathbf{x}_q)$$

- 6) Write the local quantity into the array  $\mathbf{g\_K}$  using the *accumulate* (atomic reduction) operation.

Steps 1), 2) and 6) involve communication, whereas steps 3) to 5) can all be executed locally.

The arrays  $\mathbf{g\_DM}$  and  $\mathbf{g\_K}$  are distributed by atomic blocks (as described in [19]), thereby reducing the amount of communication. However, the resulting algorithm is still sensitive to the communication bandwidth. The transformation of the quantities into mirrored arrays has allowed us to hide the communication latency occurring in steps 2) and 6). This process requires the following modifications. In step 1) the distributed array  $\mathbf{g\_DM}$  is now copied into a

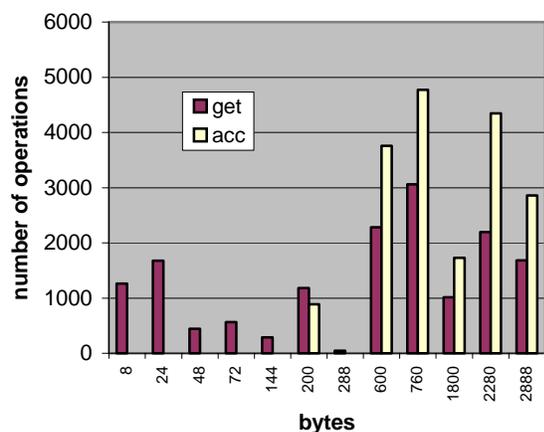


Figure 9: Communication profile in the benchmark illustrating the number operations for specific message size in a four processors' run.

mirrored array, eliminating the inter-node communications previously needed in step 2).  $\mathbf{g\_K}$  is now a mirrored array, therefore no inter-node communications are used in the *accumulate* operation (equivalent to daxy within the SMP node), but after merging changes to  $\mathbf{g\_K}$ , a copy operation is needed to move the data into a distributed array to use it as input for the rest of code, which uses distributed arrays.

These modifications allowed us to achieve substantial performance improvements, as shown by the benchmark results reported in figures 10 and 11. Wall clock timings for the matrix evaluation of the XC potential (step 1 to 6) of a zeolite fragment (SiOSi3) are reported as a function of the number of processors. We collected and analyzed trace data for interprocessor communication for that calculation. Figure 9 illustrates the profile of communication operations in steps 2) and 6) for that particular molecule. The profile indicates the presence of a broad distribution of messages of all sizes in these two steps and that increases in efficiency may be obtained by replacing these communications with shared memory copies using mirrored arrays. By looking at trace data we found that the communication operations in step 6) did not exhibit any particular locality pattern. Figure 10 shows results from runs on the Ethernet, Myrinet and Elan networks. The distributed data approach is responsible for the large gap in performance between slower (Ethernet) and faster (Elan and Myrinet) networks (top three curves in the plot). The mirroring of latency sensitive arrays helped reduce performance gap between slow and fast networks. It should be noted when comparing results shown in Figures 10 and 11, there is a change in the computational platforms since instead of 2-way SMP system, we are now dealing with a 4-way SMP; this hardware configuration shows similar benefits from the mirrored arrays algorithm as in figure 10.

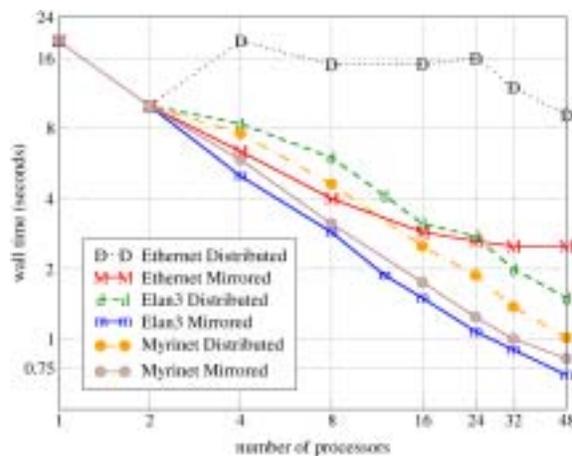


Figure 10: DFT SiOSi3 benchmark using mirrored and fully distributed approach on a 1GHZ Itanium2 dual processor system with three different interconnects: Ethernet, Myrinet, or Elan-3 (Quadrics)

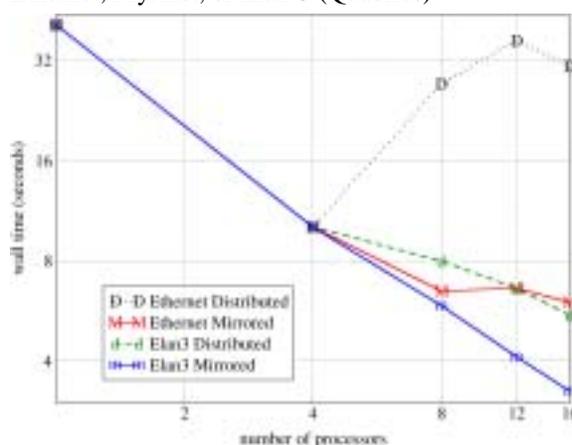


Figure 11: DFT SiOSi3 benchmark on a 1GHZ EV68 Alpha 4-way system using the Elan-3 (Quadrics) and Ethernet interconnects.

## 6. Conclusions

The results presented in this paper indicate that mirroring can be used to substantially improve performance for some algorithms running on clusters of SMP nodes. Although originally targeted for SMP nodes connected by relatively slow networks the timing data obtained for the matrix multiply and DFT calculations suggest that significant performance gains can be achieved even when the network is fast, provided enough memory is available to use mirroring.

The timing data also indicate that the merge operation remains a significant bottleneck to achieving very high levels of performance. The current merge operation is based on a global sum over all the mirrored data and for general data distributions within the mirrored arrays this may be the optimal method for merging the data. However, this has the disadvantage that it is moving all the data contained in the mirrored arrays at each step in

the global sum, which can be wasteful in cases such as the matrix multiply. For this case, most of the data in the product matrix before the merge is initialized to zero and additional savings in time could be achieved only moving nonzero data. The same algorithm could also be used to improve the performance of the copy from distributed to mirrored arrays, which contains a similar embedded merge operation. Future work will focus on developing 1) more sophisticated variations of the merge operation that can exploit these potential savings and 2) developing a more storage efficient scheme for caching the data.

## 7. Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy (DOE) at Pacific Northwest National Laboratory (PNNL). PNNL is operated for DOE by Battelle. This work was supported by the Environmental Molecular Sciences Laboratory and the Center for Programming Models for Scalable Parallel Computing sponsored by the MICS/ASCR program in the DOE Office of Science. This work was partially supported by National Computational Science Alliance under grant CH6MR1P and utilized the PSC AlphaServer cluster. The authors are indebted to Vinod Tipparaju and Manoj Krishnan for useful conversations.

## References

1. Xing Du, Xiaodong Zhang, The impact of memory hierarchies on cluster computing, Proc. Joint 13<sup>th</sup> IPPS and 10<sup>th</sup> SPDP, 1999.
2. James B. White and Steve W. Bova. Where's the overlap? Overlapping communication and computation in several popular MPI implementations. In Proc. 3<sup>rd</sup> MPI Developers' and Users' Conference, March 1999.
3. Bill Lawry, Riley Wilson, Arthur B. Maccabe, and Ron Brightwell, COMB: A Portable Benchmark Suite for Assessing MPI Overlap, IEEE Cluster 2002.
4. J. Nieplocha and R.J. Harrison, Shared-memory programming in metacomputing environments: The Global Array approach, The Journal of Supercomputing, 11:119-136, 1997.
5. J. Nieplocha, R.J. Harrison, "Shared-memory NUMA programming on I-WAY", Proc. of IEEE High Performance Distributed Computing Symposium HPDC-5, IEEE Computer Society Press, 1996.
6. J. Nieplocha, R.J. Harrison, and I. Foster, "Explicit Management of Memory Hierarchy", in "Advances in High Performance Computing", Eds. J. Kowalik, L. Grandinetti, and M. Vajtersic, Kluwer Academic, 1997.
7. S. Kim, AV Veidenbaum, On interaction between interconnection network design and latency hiding techniques in multiprocessors, The Journal of Supercomputing, 16 (3): 197-216, 2000.
8. T. Mowry, A. Gupta, Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors, Journal of Parallel and Distributed Computing, vol 12, 2, 1991.
9. Z. Zhang, J. Torrellas, Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching, 22nd Intern. Symposium on Computer Architecture, 1995.
10. Roh, B. H. Seong, D. Park, Hiding latency through bulk transfer and prefetching in distributed shared memory multiprocessors, Proc. 4th High Performance Computing Asia-Pacific Region, 2000.
11. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. Proc. 17th Annual Intern. Symposium on Computer Architecture, 1990.
12. J. Nieplocha, R.J. Harrison, R.J. Littlefield, Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance computers, The Journal of Supercomputing, vol 10, 1996.
13. J. Nieplocha, V. Tipparaju, A. Saify, D. Panda, Protocols and Strategies for Optimizing Remote Memory Operations on Clusters, Proc. Communication Architecture for Clusters Workshop of IPDPS'02. 2002.
14. J. Nieplocha, V. Tipparaju, J. Ju, E. Aprà, One-sided communication on Myrinet, Cluster Computing, 6, 115-124, 2003.
15. R.A. Kendall, E. Aprà, D.E. Bernholdt, E.J. Bylaska, M. Dupuis, G.I. Fann, R.J. Harrison, J.L. Ju, J.A. Nichols, J. Nieplocha, T.P. Straatsma, T.L. Windus, A.T. Wong, High performance computational chemistry: An overview of NWChem a distributed parallel application, Comput. Phys. Commun. 128 (2000) 260.
16. D.E. Bernholdt, E. Aprà, H.A. Früchtl, M.F. Guest, R.J. Harrison, R.A. Kendall, R.A. Kutteh, X. Long, J.B. Nicholas, J.A. Nichols, H.L. Taylor, A.T. Wong, G.I. Fann, R.J. Littlefield, J. Nieplocha, Parallel Computational Chemistry Made Easier: The Development of NWChem, Int. J. Quantum Chem. Symposium 29 (1995) 475.
17. A.D. Becke, A Multicenter Numerical Integration Scheme for Polyatomic Molecules, J. Chem. Phys. 88 (1988) 1053.
18. B. G. Johnson, P. M. W. Gill and J. A. Pople, The Performance of a Family of Density-Functional Methods, J. Chem. Phys. 98 (1993) 5612.
19. R.J. Harrison, M.F. Guest, R.A. Kendall, D.E. Bernholdt, A.T. Wong, M. Stave, J.L. Anchell, A.C. Hess, R.J. Littlefield, G.I. Fann, J. Nieplocha, G.S. Thomas, D. Elwood, J.L. Tilson, R.L. Shepard, A.F. Wagner, I.T. Foster, E. Lusk, R. Stevens, High Performance Computational Chemistry(II): A Scalable SCF program, J. Comp. Chem. 17, 124 (1993).
20. T. Steinke, Tools for parallel quantum chemistry software, in Modern Methods and Algorithms in Quantum Chemistry, J. Grotendorst (ed.), John von Neumann Insitutute for Computing, Jülich, Vol 1. 2000.