Parallelization of the NAS Conjugate Gradient Benchmark Using the Global Arrays Shared Memory Programming Model

[†]Yeliang Zhang, [‡]Vinod Tipparaju, [‡]Jarek Nieplocha, [†]Salim Hariri
†University of Arizona
‡Pacific Northwest National Laboratory

Abstract

The NAS Conjugate Gradient (CG) benchmark is an important scientific kernel used to evaluate machine performance and compare characteristics of different programming models. Global Arrays (GA) toolkit supports a shared memory programming paradigm and offers the programmer control over the distribution and locality that are important for optimizing performance on scalable architectures. In this paper, we describe and compare two different parallelization strategies of the CG benchmark using GA and report performance results on a shared-memory system as well as on a cluster. Performance benefits of using shared memory for irregular/sparse computations have been demonstrated before in the context of the CG benchmark using OpenMP. Similarly, the GA implementation outperforms the standard MPI implementation on shared memory system, in our case the SGI Altix. However, with GA these benefits are extended to distributed memory systems and demonstrated on a Linux cluster with Myrinet.

1. Introduction

The NAS Conjugate Gradient (CG) benchmark is often used to evaluate machine performance and compare characteristics of different programming models. It solves an unstructured sparse linear system by the conjugate gradient method. The CG benchmark is quite memory intensive; it tests irregular long distance communication and employs unstructured sparse matrix vector multiplication. The benchmark has been parallelized and studied in context of multiple programming models [3, 4, 5, 8, 9, 10, 14, 16], including shared memory.

The shared/global view of data often leads to elegant and efficient implementations, especially for sparse and irregular problems [1, 19]. The OpenMP shared-memory programming model has been shown effective for parallelization of the CG benchmark and has delivered performance superior to that of the reference MPI implementation [8]. Shared memory is well established for small and medium scale systems as both a simple to use and efficient programming approach. However, large scalable shared-memory architectures have been more difficult and expensive to build than distributed memory systems. The underlying Non-Uniform Memory Access (NUMA) characteristics of such systems requires consideration of the data distribution and locality of reference to achieve performance, yet the traditional shared-memory programming style provides very little support to the programmer to address these issues.

The purpose of the current work was to evaluate the effectiveness of the Global Arrays (GA) shared-memory approach as the parallelization strategy for the NAS CG benchmark on representative modern parallel systems. They are the SGI Altix and the Linux cluster with Myrinet that represent shared- as well as distributed- memory designs, respectively. GA is implemented as a library with multiple language bindings (Fortran, C, C++, Python). It provides a portable interface through which each process in a parallel program can independently, asynchronously, and efficiently access logical block of physically distributed matrices, with no need for explicit cooperation by other processes. This characteristic is similar to the traditional shared-memory programming model. However, the GA model also acknowledges that remote data is slower to access than local ones, and it allows data locality to be explicitly specified and used. In these respects, it is similar to message passing. Thanks to these, as well as other advanced features of GA— such as non-blocking communication operations and direct access to shared-memory data— the GA implementation of the NAS CG benchmark exceeded the MPI performance on the systems used in the study. For example, for Class C performance, improvement of 25.65% is achieved on 32 processors of the Linux cluster and 44.32% on 64 processors of the SGI Altix over the standard MPI implementation. The corresponding improvement rates for Class B are 26.49% and 33.82%. For the smallest problem size represented by Class A, performance of the GA and MPI versions were comparable. This paper describes and discusses merits of two parallelization strategies for this benchmark: one fully distributed and the other partially distributed. As the abstract machine model targeted by both implementations, a cluster with shared-memory nodes is assumed.

The rest of this paper is organized as follows. Section 2 introduces the kernel CG algorithm. Section 3 outlines characteristics of GA while Section 4 describes our implementations of the CG benchmark and contrasts them to the MPI-based counterparts. Section 5

Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)
1530-2075/05 $ 20.00 **IEEE**

compares performance of the GA and MPI strategies, related work is described in Section 6, and Section 7 summarizes our research and presents conclusions.

2. Kernel CG Description

NAS kernel CG is to solve an unstructured sparse linear system by the conjugate gradient method. It uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of nonzero values [6]. The inverse power method involves solving a linear system of equations $Az = x$ using the conjugate gradient method. Figure 1 illustrates the algorithm. The values for the size of the system n, number of outer iterations, and the shift λ for different problem sizes in the benchmark are given in [6]. In every iteration, the calculated eigenvalue estimate ζ must agree with the reference value ζ_{REF} within a tolerance of 1.0×10^{-10}, i.e., $|\zeta - \zeta_{REF}| \leq 1.0 \times 10^{-10}$.

```
z=0
r = x
ρ = r^T r
p = r
do i = 1, 25
        q = Ap
        α = ρ / (p^T q)
        z = z + αp
        ρ0 = ρ
        r = r - αq
        ρ = r^T r
        β = ρ / ρ0
        p = r + βp
enddo
compute residual norm explicitly: ||r|| = ||x – Az||
```

Size	n	# iter	non-zeros per row	λ
Class A	14000	15	11	20
Class B	75000	75	13	60
Class C	150000	75	15	110

Figure 1: Conjugate Gradient Method: algorithm (left) and parameters for three problem sizes (right)

3. Global Array Toolkit

In the traditional 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 shared-memory systems, global memory is accessed in the same manner as local memory, i.e., by load/store operations. The shared-memory paradigm eliminates the synchronization that is required when message passing is used to access non-private data. A disadvantage of many shared-memory models is that they hide the Non-Uniform-Memory-Access (NUMA) memory hierarchy of the underlying distributed-memory hardware. The GA programming model exposes to the programmer the hierarchical memory of modern high performance computer systems, and by recognizing the

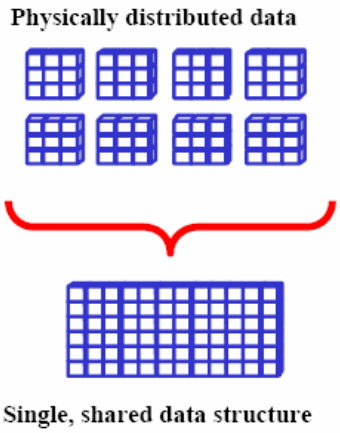

Figure 2: Dual view of global array data structure

communication overhead for remote data transfer, it promotes data reuse and locality of reference [17,22]. The distribution and locality information is available through library operations that 1) specify the array section held by a given process, 2) specify which process owns a particular array element, and 3) return list of processes and the blocks of data owned by each process corresponding to a given section of an array. Figure 2 shows a dual view of the global array data structure.

The GA programming model can be characterized as follows. Processes can communicate with each other by creating and access GA distributed matrices as well as conventional message-passing (MPI). Global arrays are physically distributed blockwise, either regularly or as the Cartesian product of irregular distributions on each axis. Each process can independently and asynchronously access any two-dimensional patch of a GA distributed matrix, without requiring cooperation by the application code in any other process. 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. If the data is 'local', process can directly access the memory block to retrieve data instead of using 'get' access. Each process can determine which portion of each distributed matrix is stored 'locally' and can access it directly (by a local pointer). Every element of a distributed matrix is guaranteed to be 'local' to exactly one process. In addition to the one-sided asynchronous operations, GA provides a set of collective operations that work on sections or whole global arrays. They include BLAS-like data-parallel operations such as dot product, scale, or vector addition. Some of these operations were useful for implementing the CG algorithm.

GA has been used intensively in scientific applications and shown high performance and scalability [17,22].

4. Analysis and Design

We profiled the serial version of the CG benchmark to gain some insight into where most of the computation time was spent. The main iteration loop of CG contains one sparse matrix vector multiplication, two reduction sums, and a few scalar operations. The most computationally expensive part of the code is the sparse matrix vector multiplication: $\boldsymbol{q = A \times p}$. This operation takes 99.6%, 96.5%, and 95.7% of the total execution time on the Class A, B, and C problem sizes, respectively. Thus, this seemed to be the area where we needed to effectively utilize the GA model and analyze the different implementation options we have. More than the mere implementation of the CG benchmark using the GA toolkit, our objective is to analyze the effectiveness in GA's ability and efficiency of its feature set, and to provide efficient shared-memory programming style for both distributed and shared-memory architectures.

We hence start by describing how we implemented a fully distributed version of the CG benchmark where the array A and vectors *q* and *p* were fully distributed among the participating processors. The term "fully distributed" is used here to emphasize the difference from the standard MPI CG version that uses a hybrid distributed/replicated scheme for storing vectors used in the CG algorithm in order to minimize communication time. The first distributed implementation (naïve) was then compared to the standard (hybrid) MPI version. To better understand the effects of replication of vectors in the MPI CG code of the NAS NPB 2.3 suite, we modified it to fully distribute vectors ***q*** and ***p***. We also improved the original naïve GA version of the benchmark by exploiting the locality information the GA model provides and contrasted it against the fully distributed MPI implementation.

After learning that the replicated/distributed based scheme of MPI was essential for reducing the communication volume that limited scaling, we subsequently implemented a similar solution in the GA code. That modification would not be feasible to implement without support for processor groups most recently introduced to GA. An alternative solution would require collective high-level linear algebra operations used in the fully distributed version to be replaced by low-level code.

4.1 Naive GA Implementation

Our first implementation of CG involved distributing the array A in rows among processes and distributing the vectors ***p*** and ***q***. This is the simplest and most natural parallelization strategy for the CG benchmark using GA, and it was derived by closely following the serial version of the benchmark (rather than the MPI version). Figure 3 shows the distribution of matrix ***A*** and vector ***p***, ***q*** with respect to four processors. The subscript in Figure 3 stands for the processor rank. This parallelization scheme leads to quite a compact (in comparison to MPI code) and straightforward code and, in particular, it relies on the vector addition and dot product operations GA provides. We call this implementation "naïve" since this implementation might represent code written by a user unfamiliar with the features GA offers for optimizing locality or exploiting the underlying machine model.

For "i" representing the process rank, the operation of $q_i = A_i \times p$ is shown in Figure 4. On each processor, the partial matrix-vector multiplication result is first stored in a temporary local buffer ***w***, then the ***put*** operation is used to copy the data into a global array. In fact this copy is unnecessary since we could use direct memory access by storing result in the local part of vector ***q***.

Because all processes do not store a full copy of ***p*** and ***q***, the naïve GA implementation uses less local buffer space for these vectors than the standard MPI implementation, described in the following section. In addition, the constraint imposed by the MPI implementation on the number of processors being a power of two is not applicable: the GA version can use an arbitrary number of processors.

4.2 MPI Implementation of CG

The MPI CG code accepts a power of two number of processors that are mapped onto a grid of row by column processors. The total number of processors is equal to processors per row times processors per column. If total number of processors is not a square, then processors per column are double the processors per row. The subroutine ***makea*** generate an ***n/nprows*** by ***n/npcols*** submatrix for each processor. Computation load is equally distributed among processors. The vectors ***p*** and ***q*** are replicated and distributed (same applies to ***r*** and *z*). Specifically, vectors are replicated along the rows -- within every row, vectors are distributed, but among rows, the vectors are duplicated. For example, 16 processors would be divided into 4x4 processor grid. Vectors would be distributed into *four* blocks and then the identical copy of each block is stored on *four* corresponding processors.

In order to obtain the correct copy of ***q*** vector, there are two transpose processor communications to update ***q*** on different processor rows in each iteration. Profiling results on the Linux cluster showed that more than 90% of the computation cost is contributed to the matrix-vector multiplication $\boldsymbol{q = Ap}$. After the calculation, the partial multiplication results are summed across rows. Then processors on different logical grid rows exchange piece of ***q*** to assure every logical processor row has the identical copy of ***q*** for the next iteration. Most of the communication time is spent in these two

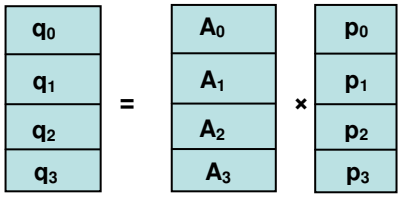

Figure 3: Fully distributed p and q in CG matrix-vector multiplication

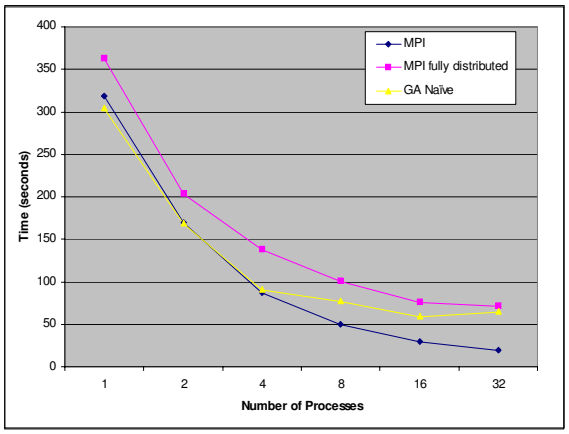

Figure 5: CG Class B performance of the standard and fully distributed MPI implementations compared to the naïve GA implementation on the Linux cluster with Myrinet

operations. Table 1 shows communication time to all-reduce ***q*** from all the processes, the total communication time during the benchmark execution, the time spent on matrix-vector multiplication and the total execution time. (Class A and Class C have similar characteristics.)

The standard MPI version of CG uses multiple replicas of vectors ***p***, ***q***, ***r***, *z* apparently to reduce the volume of communication. To better understand performance implication of this technique, we implemented a version of the MPI benchmark that avoids the replication and compared its performance against the standard version. In the MPI fully distributed CG implementation, every process is assigned a strip of rows of matrix A. Accordingly, every vector is distributed within the processes. Every process first retrieves the whole vector ***p*** then does the matrix-vector multiplication. Then the MPI_Allgather operation is used to store the partial matrix-vector multiplication result into vector q. In this implementation, there are two major data-communication phases. The first occurs in the matrix multiplication; every process needs to obtain the portion of vector ***p*** it does not own. Second is the MPI_Allgather operation to assure every process has the current value of ***q***. Figure 5 illustrates that the

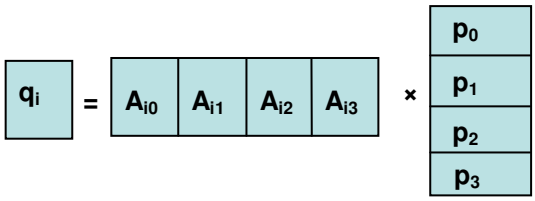

Figure 4: Local product $q_i = A_i \times p$

P	Communication of q vector	Total communication	Matrix-vector multiplication execution time	Total time
2	3.38	3.58	156.42	167.25
4	4.31	5.4	78.40	87.67
8	4.58	5.83	41.79	49.84
16	4.78	5.87	21.12	28.25

Table 1: Major components of the execution time in seconds in the MPI version for Class B on the Linux cluster

replicated/distributed scheme used by the standard MPI implementation is very effective strategy for extending scalability over the fully distributed implementations of CG.

4.3 Optimized Fully Distributed GA Version

GA allows a process to access the memory allocated for a global array by any other process in the same SMP node. This is possible because for performance reasons shared memory is used for storing global arrays. Thus, although every process only updates the portion of ***q*** it owns, all the other processes in the same SMP node are able to access this memory directly, thereby avoiding unnecessary copies. In case of the SGI Altix, a process can access data in the entire global array directly. To make such capability useful to the programmer, GA offers query interfaces about task mapping to individual SMP nodes of a cluster in the parallel job.

In the case of the CG benchmark, we can easily eliminate the ***put*** operation for vector ***q*** (storing the product in the local part of ***q*** instead of the temporary vector ***w***, see section 4.2) and ***get*** operation for parts of ***p*** located on the same SMP node. The communication time across the network can be optimized as well. Specifically we would like to hide the cost of the data transfers by overlapping communication with computations involved in the sparse matrix vector multiplication and pipelining these operations for multiple logical blocks, see Figure 4. The overlap is accomplished through the use of the GA nonblocking ***get*** operation that was introduced in version 3.3 of the package. The number of logical blocks does not have to match the number of processors. We found that a

good performance is achieved if the ratio of the number of blocks and processors ranges from half to one. The best approach to initiate the pipeline process is to start computations with blocks of ***p*** that are stored on the local SMP node for which the data is directly available through shared memory. Thus we can overlap the computation with communication for next remote block.

4.4 Process Group Based Implementation

Global Arrays can be created under two process group contexts: within the whole world group or within a specified processor group. The GA operations can be performed on arrays created in either context. The MPI NAS Parallel CG algorithm replicates its vectors ***p*** and ***q***. One of the ways to achieve a similar replication effect for these vectors in GA is to create the ***p*** and ***q*** vectors over a group of processes. This can be done such that vectors are replicated between the processor groups and distributed within the group. Because of similar distributions, MPI and our implementation will have similar calculation and communication patterns. Collective linear algebra operations in the GA implementation can be made to operate within the scope of any group. A processor group, based on how many processes it is created with, could encompass more than one SMP node.

We start off by determining the number of rows of the matrix *A* that should be assigned to each processor group. Within every processor group, the vectors ***p*** and ***q*** are distributed; across groups, they are replicated. When possible, matrix-vector multiplication exploits shared memory but the computation load remains the same as in the MPI version. Figure 6a shows the access to matrix ***A*** on 4 processes in MPI and GA. The thick lines define array blocks assigned to processors. Because the corresponding vector ***p*** required to perform multiplication is located in shared memory, processors on the same node can access each other's ***p***. We change the access to matrix ***A*** such that the two processes that are on the same SMP node could share access and compute a partial ***q***. Access of matrix A can similarly be modified for a general case such that all the processes in the SMP node compute a partial ***q***.

As in the MPI implementation, after the matrix-vector multiplication, there are two stages of communication: 1) Summation of ***q*** within the processor group, and 2) Exchange vector ***q*** among the groups. The summation operation involves adding parts of the computed *q* within a group. This is done in a pair wise manner using a pair wise all-reduce algorithm as shown in Figure 6b. In addition to this, an exchange with the transpose process needs to be done. This is accomplished by doing a non-blocking put to the transpose processor. From Figure 7, it can be seen that GA has one additional transpose but when running on SMP nodes with more than one process per node, this extra transpose is done within the node.

5. Performance Evaluation

Two classes of systems were chosen to perform the experimental evaluation. They have been chosen to show the impact of shared and distributed memory architectures on effectiveness of the two different programming paradigms evaluated in this paper using the fully distributed and hybrid process group based methods.

1) Cluster of 24 dual 1GHz Itanium-2 nodes. The compute nodes run Red Hat Linux with kernel 2.4-20. The compute nodes have 6 GB of memory per node and are interconnected with the dual port Myrinet E cards.

2) SGI Altix 3000 128-way SMP with the Intel Itanium-2 1.5 GHz processors. It runs ProPack 3.0, with SMP NUMA enhanced Linux 2.4.21. The processors are connected with Dual-plane, fat-tree topology to provide 3.2 GB/s bidirectional bandwidth per link.

We used GNU C compiler version 3.3 and Intel Fortran compiler version 7.0 on both platforms.

5.1 Fully Distributed Implementations

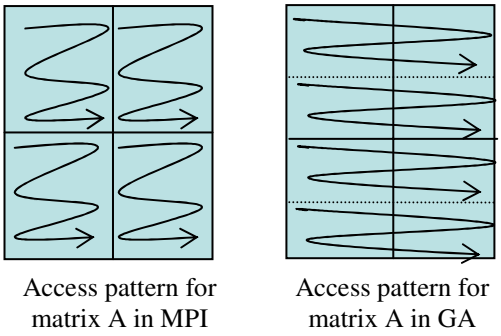

Figure 6a: Access pattern for four processes

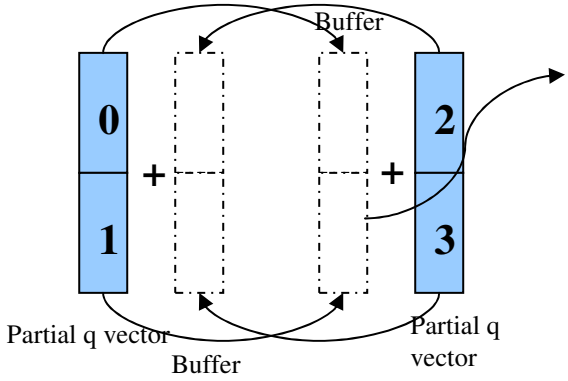

Figure 6b: Summation of *q* within a group of 4 processors – Exchange based All-reduce

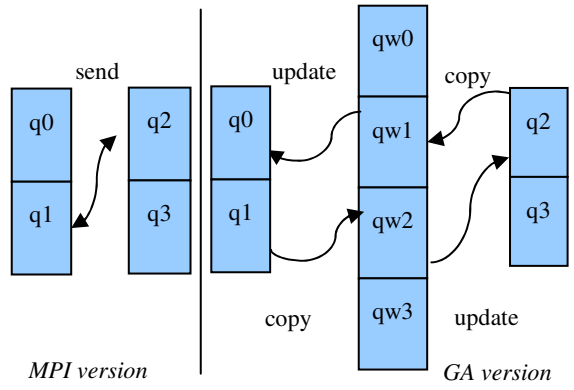

Figure 7: Transpose exchange on 2-way SMP node

The implementation of CG used here is the standard MPI distributed version described in Section 4.2.

We compare it to the optimized fully distributed Global Array implementation of the CG benchmark described in section 4.3. On shared memory architectures like the Altix, GA utilizes the shared memory and does not require any explicit copies to access data. For the fully distributed case on the Altix, both the MPI and the optimized GA implementations scale well for Class B and Class C given that the size of the data transfer is large. For class A, the scaling curve flattens after 16 processors and the constant overhead is clearly reflected in the numbers from there on. Figure 8 shows the execution results of the fully distributed case on Altix1. The time axis in Figure 8 is in log scale.

On the cluster for the optimized fully distributed case, GA implementation performs better than the NAS MPI implementation for Class B and Class C and the results indicate good scalability (Figure 9). The performance of the fully distributed method, on clusters is dependant on how efficient the underlying network communication in GA operations is. With the use of ARMCI's non-blocking RMA underneath, GA

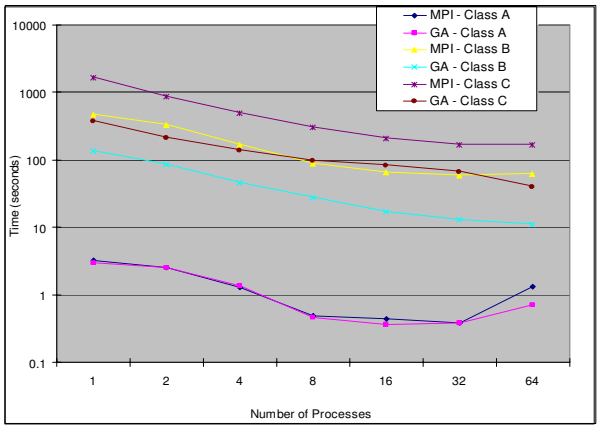

Figure 8: Fully distributed GA and MPI implementations of NAS CG on the SGI Altix

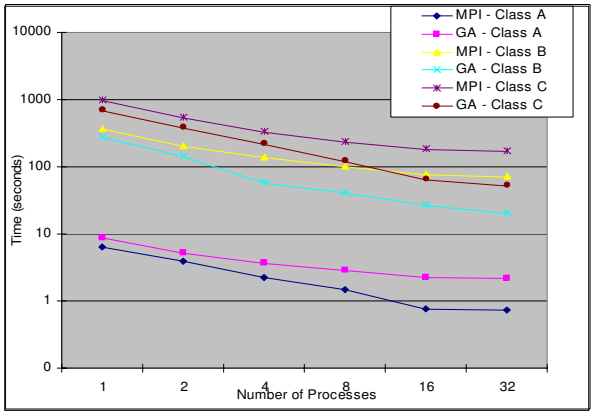

Figure 9: Fully distributed GA and MPI implementations of NAS CG on the Linux cluster

operations are efficient and have been designed to overlap as much of the communication as possible [15]. For the smallest problem size (Class A) when the data each processor operates on is very small the GA optimized fully distributed implementation does not perform as well as MPI primarily due to the amount of computations being insufficient to overlap with communication and the higher overhead associated with global index translation for GA data structures. The Class A results (Figure 9) show the slightly higher overhead in using GA over MPI which is perceived when the amount of data to be handled per processor is very less, and the amount of parallelism is depleted for the small size of the benchmark. This overhead is a small tradeoff for the ease of programming and the higher level of abstraction GA provides [18].

5.2 Process Group Based Implementations

We compared the Global Array implementation described in section 4.4 with the standard unmodified MPI version of the CG benchmark. These are equivalent versions that use similar distribution strategies for vectors in the CG algorithm, see Figure 1.

Performance results presented in Figures 10 and 11,

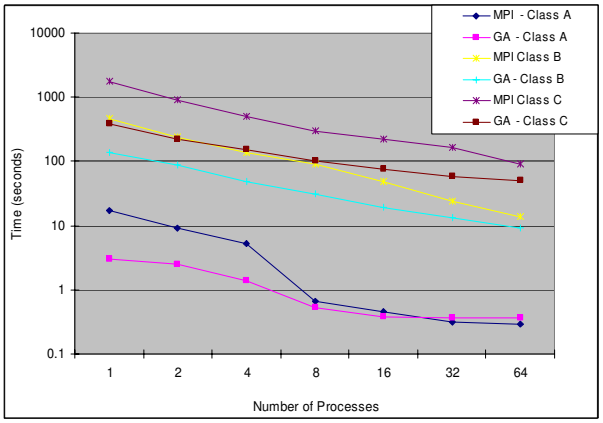

Figure 10: Performance of group-based GA and standard MPI implementations on the SGI Altix

demonstrate that GA version is faster than the fully distributed MPI implementation with exception for a few data points for the smallest problem size. This is in part due to computation overlapped with data exchange that replaces the all-gather operation in MPI. Direct use shared memory access contributes to efficient execution of matrix-vector multiplication and data exchange is done within local node, while the MPI counterpart code requires more time to complete these two operations.

6. Related Work

The NPB is a suite of well-recognized benchmarks for testing compilers, parallel hardware and parallelization tools [20,21]. Its popularity comes the fact that the benchmarks are derived from real world computational fluid dynamics and aerophysics applications.

In addition to the MPI implementation of NPB CG benchmark distributed by NASA, there exist a number of other implementations. In [10], CG was implemented with HPF replicating vector ***p***, ***r*** and ***z*** on each process. However, the HPF implementation shows poor scalability because of the ***daxpy*** operations on a replicated vector ***p***. CG was also implemented using OpenMP in [8] and [9]. Since OpenMP uses compiler directives to achieve parallelization and does not provide much control over data locality, it is rather difficult to optimize performance. Paper [8] shows that an OpenMP implementation can outperform MPI on a shared memory system but requires substantial programming efforts on data set adaptation and the optimization of the inter-thread communications. In [14], the authors compared the Unified Parallel C (UPC) and MPI CG implementations. MPI showed better performance and scalability than UPC. The paper authors claimed that UPC implementation does not give a better result because the high cost of the UPC collective operations. In [16], CG is implemented on PVM system tested on Ethernet and two types of FDDI networks. The authors stated that using PVM fast send and receive calls for direct transfer of data without requiring buffer initialization and packing would bring the PVM performance comparable to that of MPI implementation on certain platforms. CG has also been implemented using a high-level parallel programming language ZPL [2, 3, 12]. The ZPL version used parallel array remapping to reduce the communication time. For Class C ZPL version showed performance comparable to MPI on the IBM SP2 and Linux. It was however less competitive on the Cray T3E [2]. The remap operator of ZPL makes the code simpler and runable on any number of processor while in MPI implementation, the total number of processors is limited to be a power of two.

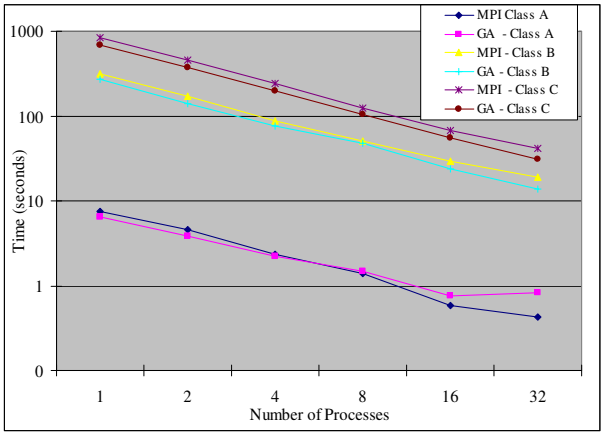

Figure 11: Performance of group-based GA and standard MPI implementations on the Linux cluster

The Co-Array Fortran (CAF) CG implementation in [5] relies on ARMCI which is the same run-time system in GA used in our CG implementation; it can perform data transfers on the memory directly using a NIC DMA engine to achieve good computation-communication overlapping. The CAF implementation allocates buffer for vector ***p*** used in CG as static data. After the DMA ***get*** is initiated, data was copied from temporary buffer into the ***q*** array. The buffer size is large with a starting memory address independent of the addresses of the common blocks. This layout of memory and buffer cause L3 cache misses 3 times larger than the corresponding MPI code. Though converting vector ***q*** into a co-array reduces the potential cache misses, the CAF implementation shows very close performance to that of MPI. However on certain numbers of processes, the CAF implementation performs worse than MPI. Hybrid models were also used to exploit shared memory architectures. Papers [7, 11, 13] try to use a hybrid programming model of MPI and OpenMP to take advantage of the SMP architecture while keeping the flavor of message passing. All of these studies show that hybrid implementations of CG can achieve better results than a pure MPI implementation when a favorable data distribution on each SMP node is adopted. However, the hybrid implementations are difficult to develop.

In all of these studies, MPI was hard to outperform: none of the other models showed as consistent performance advantage over the MPI implementation as the GA group version achieved.

7. Conclusions

The NAS CG benchmark exhibits a large amount of communication, memory reference and computation patterns which are very common in real-world scientific applications. Two implementations, fully distributed version and process group based version, of the CG benchmark using GA were discussed. They involve different distributions and communication patterns that needed to be appropriately optimized. On

both cluster and shared memory architectures, GA implementation of CG demonstrates better performance than MPI in all the cases, except the smallest Class A on larger processor counts. This is significant since GA is a substantially higher-level programming model than MPI, and MPI performance has long been studied and optimized by a broad research community as well as computer vendors. We found that the higher-level abstractions GA model offers enabled a fairly quickly parallelization of the CG benchmark; and the availability of interfaces for exploiting locality and optimizing communication allowed us to achieve competitive performance as compared to the reference MPI implementation [18]. The key factor to achieving performance was exploitation of locality and underlying machine configuration to reduce the access contention, and overlapping the computation with communication. The zero copy protocol used by GA reduces the memory access contention and speeds up the data communication; therefore, more computation can be overlapped with the communication. Accessing directly data located in shared memory leads to further performance improvements.

References

[1] A. George, E. Ng, Some Shared Memory is Desirable in Parallel Sparse Matrix Computation, ACM SIGNUM, Vol. 23, 2, 1988.

[2] B.L. Chamberlain, S. Choi, Steven J. Deitz, L. Snyder. The high-level parallel language ZPL improves productivity and performance. Proc. PPHEC' 04.

[3] C. Lin, L. Snyder, R. Anderson, B. Chamberlain, S. Choi, G. Forman, E. Lewis, and W. D. Weathersby. ZPL vs. HPF: A comparison of performance and programming style. Tech. Report 95--11--05, U. Washington, 1995.

[4] Clmenon, C., K.M. Decker et al., HPF and MPI implementation of the NAS Parallel Benchmarks supported by integrated program engineering tools, Proc. PDCS'96, 1996.

[5] C. Coarfa, Y. Dotsenko, J. Eckhardt, J. M. Mellor-Crummey: Co-array Fortran Performance and Potential: An NPB Experimental Study. LCPC 2003.

[6] D. Bailey, T. Harris, W. Saphir, R. Van der Vijingaart, A. Woo, and M. Yarrow, The NAS Parallel Benchmarks 2.0, Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[7] F. Cappello, D. Etiemble, "MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks," Proc. SC'2000.

[8] G. Krawezik, F. Cappello. Performance Comparison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors. Proc. 15th ACM SPAA, 2003

[9] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. NAS Tech. Report NAS99 -011, 1999.

[10] M. Frumkin, H. Jin, J. Yan, Implementation of NAS Parallel Benchmarks in High Performance Fortran. Proc. IPDPS' 2000.

[11] N. Drosinos,N. Koziris, Performance Comparison of Pure MPI vs. Hybrid MPI-OpenMP Parallelization Models on SMP Clusters, Proc. IPDPS'04.

[12] S. J. Deitz, B. L. Chamberlain, and L. Snyder. The design and implementation of a parallel array operator for the arbitrary remapping of data. ACM Conference on Principles and Practice of Parallel Programming. 2003.

[13] T. Viet et al., Optimization for Hybrid MPI-OpenMP Programs on a Cluster of SMP PCs, Japan-Tunisia Workshop on Computer Systems and Information Technology, 2004.

[14] T. El-Ghazawi, F. Cantonnet, UPC Performance and Potential: A NPB Experimental Study Supercomputing 2002.

[15] V. Tipparaju, M. Krishnan, J. Nieplocha, G. Santhanaraman, D. Panda, Exploiting Non-blocking Remote Memory Access Communication in Scientific Benchmarks, in Proc. HiPC'2003.

[16] S. White, S., Alund, A., and Sunderam, V. S. Performance of the NAS parallel benchmarks on PVM-Based networks. Journal of Parallel and Distributed Computing 26,1 (1995), 61--71.

[17] J. Nieplocha, B. Palmer, V.Tipparaju, M. Krishnan, H. Trease, Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit, to appear in Intern. J. High Perf. Comp. Applications, 2005.

[18] D. Bernholdt, J. Nieplocha, P. Saddayappan, Raising the Level of Programming Abstraction in Scalable Programming Models, Proc. P-PHEC 2004

[19] F. T. Chong, A. Agarwal, Shared Memory versus Message Passing for Iterative Solution of Sparse, Irregular Problems, Parallel Processing Letters, Vol. 9, No. 1, 1999.

[20] T. Ngo, L. Snyder, B. Chamberlain, Portable Performance of Data Parallel Language. SC 97, 1997.

[21] V. Adve, G. Jin, J. Mellor-Crummery, Q. Yi, High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes. SC'98

[22] J. Nieplocha, R. Harrison, R. Littlefield, Global Arrays: A portable shared memory model for distributed memory computers, Supercomputing'94.