Exploiting Architectural Support for Shared Memory Communication on the Cray X1 to Optimize Bandwidth-intensive Computations

Manojkumar Krishnan Jarek Nieplocha Vinod Tipparaju
Computational Sciences & Mathematics
Pacific Northwest National Laboratory

The Cray X1 supports a variety of communication protocols including message passing, shared memory, and remote memory access (RMA) put/get model. This paper uses the parallel dense matrix multiplication as an example of a bandwidth-intensive computational kernel and discusses tradeoffs associated with optimizing performance of this kernel using different shared memory communication schemes. We show that the differences in the architectural support for shared memory communication on the SGI Altix and Cray X1 call for different techniques for performance optimization of the parallel matrix multiplication algorithm. Our findings indicate that explicit awareness of the task mapping to exploit cacheability attributes of shared data plays an important role in optimizing performance of this important kernel algorithm.

1. Introduction

The current architectures differ in several key aspects from those of the earlier massively parallel processor (MPP) systems. Regardless of the processor architecture (e.g., commodity vector, or commodity RISC, EPIC, CISC microprocessors), to improve the cost-effectiveness of the overall system, both the high-end commercial designs and the commodity systems employ as a building block Symmetric Multi-Processor (SMP) nodes connected with an interconnect network. All of these architectures provide hardware support for load/store communication within the underlying SMP nodes, and some extend the scope of that protocol to the entire machine (Cray X1, SGI Altix). The focus of this paper is on determining how we can take advantage of hardware shared memory communication to optimize performance of applications that have high bandwidth requirements and exhibit spatial and temporal locality in data access. As an example, we use a parallel dense matrix multiplication and, in particular, a shared memory version of SRUMMA [1], which on shared memory machines represents a parallel version of the serial block-based algorithm in Figure 1. The parallel version of SRUMMA has been implemented on shared memory systems such as SGI Altix and Cray X1 by exploiting hardware support for globally accessible shared memory. However, the nonuniform memory access (NUMA) costs of shared memory operations require special attention.

In the current effort, we found that cacheability of data stored in shared memory plays a fundamental role in optimizing application performance on the two architectures, both relying on caches to maximize serial processor performance. On the SGI Altix that provides systemwide cache-coherent shared memory, direct shared memory access leads to the best performance. However, the Cray X1 with its ability to cache only shared memory data on the same SMP node required us to switch between direct shared memory and shared memory copy protocols, depending on location of the data, to avoid very significant performance degradation in direct access to remote memory that cannot be cached. Thanks to the hardware support for shared memory communication, an efficient implementation of the matrix multiplication is feasible. For example, in multiplication of square matrices 2000 x 2000, on 128 processors of the Cray X1 the baseline SRUMMA achieves 7.2 higher aggregate GFLOPs performance level than ScaLAPACK *pdgemm* optimized by Cray. However, by exploiting differences

in cacheability attributes of the shared memory within and between SMP nodes, we have been able to improve the performance even further. For example, for matrix size 600x600, an improvement of 18.2% was recorded on 16 CPUs and 15.2% on 64 CPUs relative to the baseline SRUMMA implementation.

The paper is organized as follows. The Section 2 describes the SRUMMA algorithm, its efficiency model, and implementation. In Section 3, we describe the shared memory implementation on the Cray X1 and the SGI Altix systems. Section 4 analyzes performance results for the matrix multiplication algorithm as a function of different communications schemes, and compares it to the *pdgemm* matrix multiplication in PBBLAS/ScaLAPACK. Conclusions are given in Section 5.

2. Background

For many scientific applications, matrix multiplication is one of the most important linear algebra operations. Computer vendors have optimized the standard serial matrix multiplication interface in the open source Basic Linear Algebra Subroutines (BLAS) to deliver performance as close to the peak processor performance as possible. Because the optimized matrix multiplication can be so efficient, computational scientists, when feasible, attempt to reformulate the mathematical description of their application in terms of matrix multiplications. Parallel matrix multiplication has been investigated extensively in the last two decades [2-22]. Agarwal et al. [18] developed a matrix multiplication algorithm that overlaps communication with computation. SUMMA [19] is closely related to Agarwal's approach, and is used in practice in *pdgemm* routine in PBLAS [20], which is one of the fundamental building blocks of ScaLAPACK [21]. DIMMA [22] is related to SUMMA but uses a different pipelined communication scheme for overlapping communication and computation. In the earlier studies, researchers targeted their parallel implementations for MPP architectures with uniprocessor computational nodes (e.g., Intel Touchstone Delta, Intel IPSC/860, nCUBE/2) on which message passing performed very well and was typically the only communication protocol available. In particular, these algorithms relied on optimized broadcasts or send-receive operations.

At the high level, SRUMMA (Shared and Remote-memory based Universal Matrix Multiplication Algorithm) follows the serial block-based matrix multiplication (see Figure 1) by assuming the regular block distribution of the matrices A, B, and C and adopting the "owner computes" rule with respect to blocks of the matrix C. Each process *accesses* the appropriate blocks of the matrices A and B to multiply them together with the result stored in the locally owned part of matrix C. The specific protocol used to *access* non-local blocks varies depending on whether they are located in the same or other shared memory domain [1] as the current processor.

```
1:      for i=0 to s-1 {
2:        for j=0 to s-1 {
3:          Initialize all elements of C_ij=0 (optional)
4:          for k=0 to s-1 {
5:            C_ij = C_ij + A_ik×B_kj
6:          }
7:        }
8:      }
```

Figure 1: Block matrix multiplication for matrices N × N and block size N/s × N/s

In principle, the overall sequence of block matrix multiplications can be similar to that in Cannon's algorithm. However, unlike Cannon's algorithm, where skewed blocks of matrix A and B are shifted using message-passing to the logically neighboring processors, our approach fetches these blocks independently, as needed, without requiring any coordination with the processors that own the matrix blocks. This is possible by the use of direct remote memory or shared memory access protocols. In addition, the specific sequence in which the block matrix multiplications are executed is determined dynamically at run time to more efficiently schedule and overlap communication with computations. The absence of sender-receiver synchronization/coordination (such in Cannon's algorithm) based on message passing makes the overall algorithm more asynchronous and thus more suited for the execution environments where the computational threads share a CPU with other processes and system daemons (e.g., on SGI Altix and commodity clusters). This is because synchronization amplifies performance degradations due to the nonexclusive use of the processor by the application.

2.1 Baseline Efficiency Model

Consider a matrix multiplication operation C = AB, where the order of matrices A, B, and C is m x k, k x n, and m x n, respectively. Let us denote t_w as the data transfer time per element, t_s the latency (or startup cost), p x q the process grid in two-dimensional fashion, and P as the number of processors.We assume (as in [7, 30]) that the cost of the addition and multiplication floating point operation takes unit time (line 5 in Figure 1). For our analysis, we assume a two-dimensional matrix distributed as shown in Figure 2. The sequential time T_{seq} of the matrix multiplication algorithm is N^3 (for simplicity, $m=n=k=N$ *and* $p=q=\sqrt{P}$). The parallel time T_{par_rma} is the sum of computation time and the time to *get* the blocks of matrices A and B. Each process *gets* q blocks of matrix A and p blocks of matrix. From [1],

$$T_{par\_rma} = \frac{N^3}{P} + 2\frac{N^2}{\sqrt{P}} t_w + 2t_s\sqrt{P} \qquad (1)$$

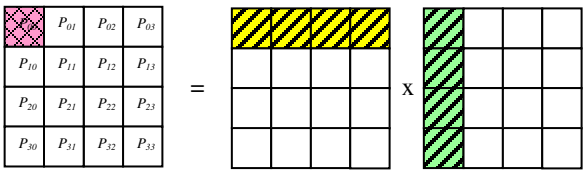

Figure 2: Matrix distribution example. In a 4 × 4 process grid, process P_{00} needs blocks of matrix A from P_{00}, P_{01}, P_{02}, and P_{03}, and blocks of matrix B from P_{00}, P_{10}, P_{20}, and P_{30}.

For a network with sufficient bandwidth, t_s can be neglected when compared to the total communication time. Therefore, the parallel efficiency (η) is

$$\eta = \text{Speedup/P} \approx \frac{1}{1+\frac{2\sqrt{P}}{N} t_w} = \frac{1}{1+O\left(\frac{\sqrt{P}}{N}\right)} \qquad (2)$$

The isoefficiency function of this algorithm is O ($P^{3/2}$), which is the same as Cannon's algorithm [7, 19].

2.2 Details of the Parallel Algorithm

The following algorithm has been proposed in [1]. It assumes the underlying hardware is a cluster of SMP nodes. We will discuss in Section 3 how the algorithm is modified for shared memory systems.

For each processor p and corresponding matrix block C_{ij} held on that processor,

1. Build a *list of tasks* (where a task computes each of the $A_{ik}B_{kj}$ products) corresponding to the block matrix multiplications in

$$C_{ij} = \sum_{k=1}^{n_p} A_{ik} B_{kj} \qquad (3)$$

2. Reorder the *task list* according to the communication domains for processors at which the A_{ik}, B_{kj} are stored. The tasks that involve matrix blocks stored in the shared memory domain of the current processor are moved to the beginning of the list. This is done to ensure overlap of computations and nonblocking communication required to bring matrix blocks from other cluster nodes to compute the other tasks on the list. Because the tasks at the beginning of the list use data accessible directly, we do not have to wait to start the pipeline. Another consideration in sorting the task list is to optimize the locality reference so that the currently held A_{ik} matrix block is used in consecutive matrix products before its copy is discarded and the corresponding buffer reused.

3. For each task on the list,
   - Issue a nonblocking get operation for the matrix block involved in the next task on the list if it is not on the same node.
   - Wait for the nonblocking get operation bringing A_{ik} and/or B_{kj} needed to execute the current task.
   - Call serial matrix multiplication *dgemm* that computes $A_{ik}B_{kj}$ and adds the result to the C_{ij} block.

4. Two temporary buffers (*B1* and *B2*) are used internally, one for communication and one for computation. At a given step, a processor receives data in *B2* while computing the data in *B1*. In the next step, data received in *B2* is used for computation and *B1* is used for receiving data. Overlapping communication with computation is achieved in all steps except the first.

As a further refinement of the algorithm, the "diagonal shift" algorithm [1] is used in Step 2 to sort the task list so that the communication pattern reduces the communication contention on clusters. We verified experimentally on the IBM SP that this indeed improves performance.

3. Shared Memory Implementation

Although the high-performance implementations of message passing can exploit shared memory internally, the performance is less competitive than direct loads and stores. Multiple studies have attempted to exploit the OpenMP shared memory programming model in the parallel matrix multiplication, either as a standalone approach on scalable shared memory systems [23, 24] or as a hybrid OpenMP-MPI approach [25, 26] on SMP clusters. Overall, the reported experiences in comparison to the pure MPI implementations were not encouraging. Unlike OpenMP, our approach is based on using shared memory within the process model. The SRUMMA algorithm was designed to be aware of the memory hierarchy in the system and take advantage of multiple communication methods. On clusters, it uses remote memory access (RMA)—often the fastest communication protocol available, especially when implemented in hardware as zero-copy remote direct memory access

(RDMA) write/read operations and shared memory with the SMP node. Each cluster node is assumed to provide efficient load/store operations that allow direct access to the data. In other words, a node of the cluster represents a *shared memory communication domain*. SRUMMA is explicitly aware of the task mapping to shared memory domains—that is, it is written to use shared memory to access parts of the matrix held on processors within the domain of which the given processor is a part, and other mechanisms to access parts of the matrix outside of the local shared memory or *RMA* domain. Implementing matrix multiplication directly on top of shared and remote memory access communication and realizing the memory hierarchy helps us optimize the algorithm with a finer level of control over data movement and hence achieve better performance.

The design and implementation of SRUMMA on X1 is based on the algorithm for scalable shared memory systems. We will provide an overview of the Cray X1 and SGI Altix as the two shared memory systems used in the current study and then describe details of the implementation.

3.1 Overview of the Cray X1 and SGI Altix

The Cray X1 combines the features of previous Cray vector systems and MPP systems into one design. Like the Cray T90, the X1 has high memory bandwidth. Along the lines of the Cray T3E, the X1 has a high-bandwidth, low-latency, scalable interconnect. The X1 design utilizes commodity CMOS technology and incorporates nontraditional vector concepts like vector caches and multistreaming processors. X1 addresses two of the major issues with large scale NUMA machines by providing a scalable cache coherency protocol and a scalable address translation mechanism. The X1 uses hierarchical design for its processor, memory, and network. There are two level processor hierarchies—MSP (multistreaming processor) and SSP (single-streaming processor). Each MSP is composed of 4 SSPs (Figure 1). Each MSP is capable of 12.8 GFlop/sec for 64-bit operations. Each SSP contains 32 vector registers holding 64 double-precision words and one two-way super-scalar unit and can deliver 3.2 GFlop/sec. The SSP uses two clock frequencies, 800 MHz for the vector units and 400 MHz for the scalar unit. The four SSPs share a 2 MB “Ecache.” The “Ecache” is designed to fill the gap between the bandwidth to main memory and the operation rate of vector units. Each X1 processor is designed to have more single-stride bandwidth than an NEC SX-6 processor. Each node determines whether a memory reference is local to the node or whether the reference is to a memory location on a remote node. Remote memory references are implemented through the network interconnect; however, remote memory cannot be cached due to the memory coherency protocol. Each node contains 32 network ports, and each port supports 1.6 GBps peak per direction. The Cray X1 runs the UNICOS/mp operating system. The UNICOS/mp operating system is based on the IRIX 6.5 operating system with Cray enhancements. It is designed to sustain the high-capacity, high-bandwidth throughput provided by Cray X1 systems. A single UNICOS/mp kernel image runs on the entire system or on each partition of the system. The Cray compilers automatically perform register allocation, scheduling, vectorization, and multistreaming to give optimal performance. Compiler vectorization provides loop-level parallelization of operations and uses the high-performance vector processing hardware. Multistreaming code generation by the compiler permits tightly coupled execution across the four SSPs of an MSP on a block of code.

The SGI Altix 3000 is a scalable cache-coherent shared memory system. It is based on the Intel's Itanium 2 processors and nonuniform memory access (NUMA) design of the SGI's NUMAflex global shared-memory architecture. The NUMAflex design permits modular packaging of CPU, memory, I/O, graphics, and storage into components known as *bricks*. The bricks can then be combined and

configured into larger systems. Two Itanium processors share a common front-side bus and memory. This constitutes a node in the SGI NUMA architecture: access to other memory (on another node) by these processors has a higher latency, and lower bandwidth. Two such nodes are packaged together in each computer brick. The network router uses dual-plane, fat-tree topology and deliver 6.4 GB/sec bidirectional bandwidth per link. Each of the Itanium-2 processors has an on-die L3 cache with 6MB memory.

3.2 Implementation

The SRUMMA algorithm when running on a system with one shared memory communication domain reduces to shared memory version. However, this algorithm has two versions; the one used depends on whether remote shared memory is locally cacheable or not. The Cray X1 with its partitioned shared memory supports load/store operations for its entire memory. It can be visualized as a cluster with four multistream processors (MSPs) on each node. A virtual memory address includes node number and

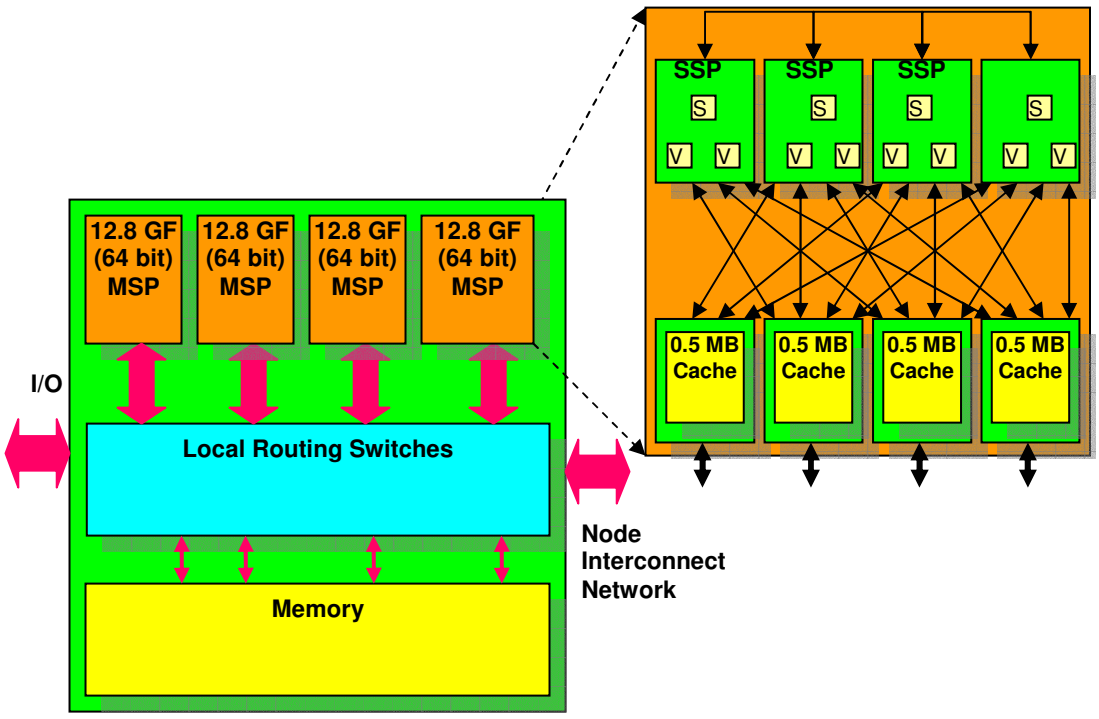

Figure 2: The X1 system: 4 multistreaming processors (MSPs) per node and structure of each MSP

address within that node. The memory on other nodes can be accessed with the load/store operations; however, it cannot be cached due to the memory coherency protocol [31]. Because the performance of the serial matrix multiplication depends critically on the effective cache utilization, on the Cray X1 we copy nonlocal blocks of matrices A and B to a local buffer before calling the serial matrix multiplication.

On the other hand, the SGI Altix is a shared memory system where shared memory data can be cached. Therefore, the matrix multiplication does not require explicit memory copies. Instead, the appropriate blocks of matrix A and B are passed directly to the serial matrix multiplication subroutine. The node interconnect in X1 is implicit, and loads and stores are extended to beyond a node. Even SHMEM and MPI are implemented on top of loads and stores in the X1. The cache coherency protocol in X1 is scalable but does not allow remote memory to be cached. This implies that every time remote data is

needed, it has to be loaded over the network. Because the serial dgemm relies on blocking and reuse of the data, direct access to noncacheable data is expected to be less efficient.

Hybrid Approach

To maximize performance on the Cray X1, we must distinguish between two types of shared memory (SMP and off-SMP) at run time and use direct and copy-based protocols. The resulting hybrid algorithm is shown in Figure 3. The implementation of this algorithm is fairly simple. However, because the Cray does not provide an interface to query task mapping to the physical nodes, the most challenging part was to determine dynamically at run-time task mapping to the SMP nodes of the machine. In the current work, as a part of the initialization and setup we determine SMP node mapping by analyzing the time each task requires to add a series of numbers located in shared memory associated with other neighboring (in a logical sense) tasks. This scheme is sufficient, given the current task mapping strategy used by Cray.

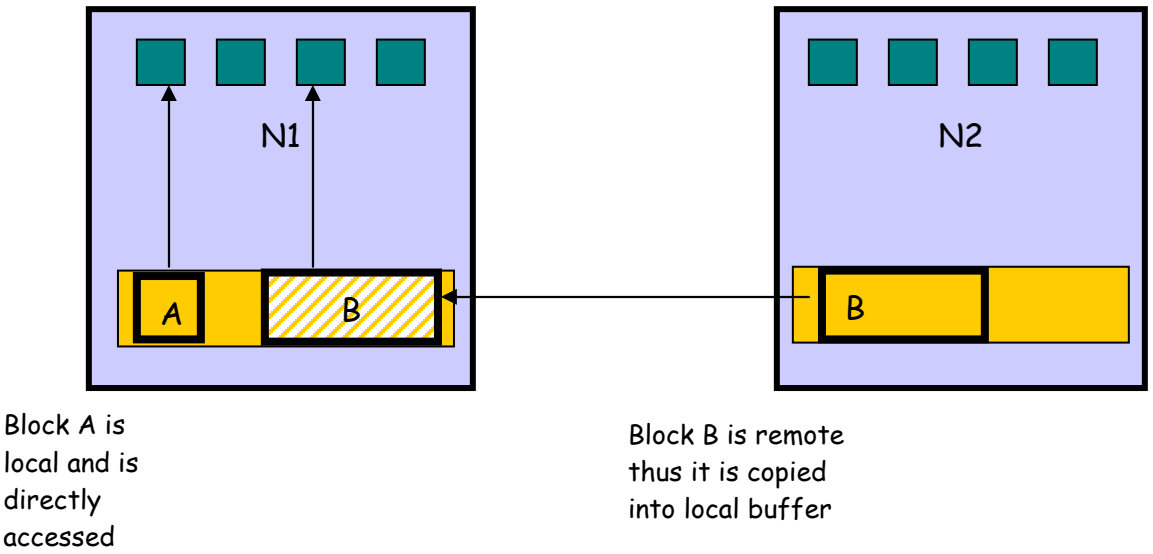

Figure 3: Hybrid algorithm on the Cray X1

4. Experimental Results

The performance of SRUMMA on the SGI Altix is shown here to contrast the architectures. In the current paper we implement and optimize SRUMMA on the Cray X1. The numerical experiments were conducted on the Cray X1 at ORNL and SGI Altix 3000, shared-memory NUMA system with 128 1.5-GHz Intel Itanium-2 CPUs at PNNL. For the comparison, we used the *pdgemm* routine from PBLAS/ScaLAPACK. The same *dgemm* (double precision serial matrix multiplication) routines from a vendor-optimized math library (-lscs for Altix, -lsci for the X1) were used in all parallel algorithms.

4.1 Evaluation of direct access approach

We demonstrate performance of the two basic strategies—copy-based and direct access represented in Figure 3. The SGI Altix is a shared memory system where shared memory data can be cached. Therefore, the matrix multiplication does not require explicit memory copies. Instead, the appropriate blocks of matrix A and B are passed, A comparison between these two schemes for $C = A^T B$ and $C =$

AB on these two platforms is illustrated in Figure 4. The SGI Altix uses 1.5-GHz Intel Itanium-2 processors rated at 6 GFLOP/s whereas the Cray X1 processor is rated at 12.8 GFLOP/s. As expected, the copy-based version is faster than the direct access version on the Cray X1 and somewhat slower on the SGI Altix (the gap between these two versions actually increases for larger processor counts on the Altix).

4.2 Performance of baseline implementation

As shown in Figures 5 and 6, the baseline SRUMMA algorithm outperforms *pdgemm* and scales better, on both shared memory systems, Cray X1 and SGI Altix. This is in part due to the benefit shared memory communication offers on these architectures over message passing.

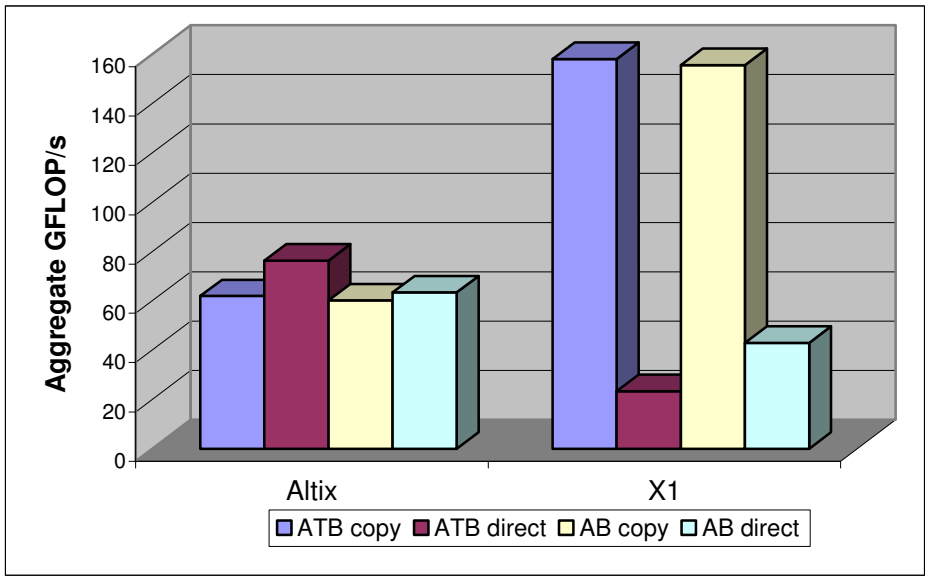

Figure 4: Matrix multiplication (N = 2000) performance on 16 processors using direct access and copy on Altix and X1. AB denotes product of two matrices whereas ATB denote that transposed matrix A is multiplied by matrix B.

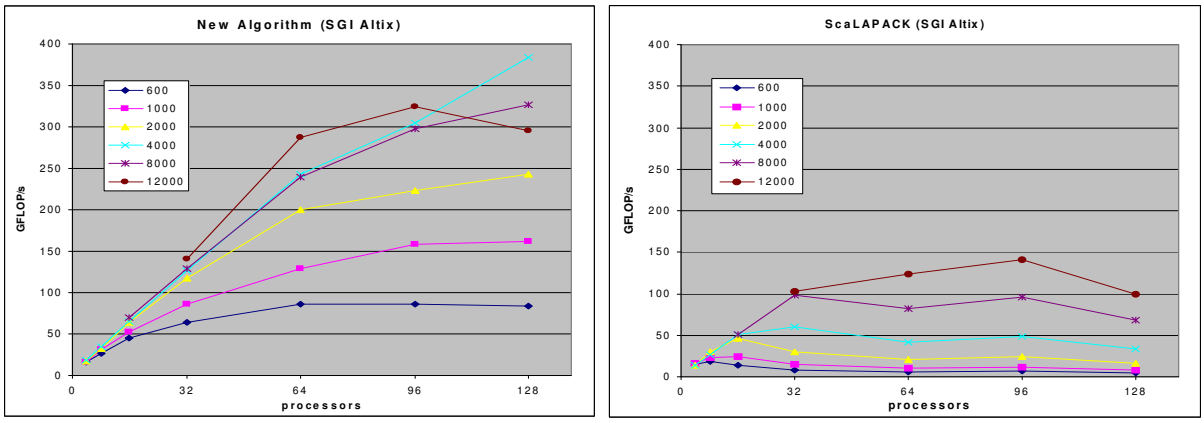

Figure 5: Performance of SRUMMA and *pdgemm* on the SGI Altix

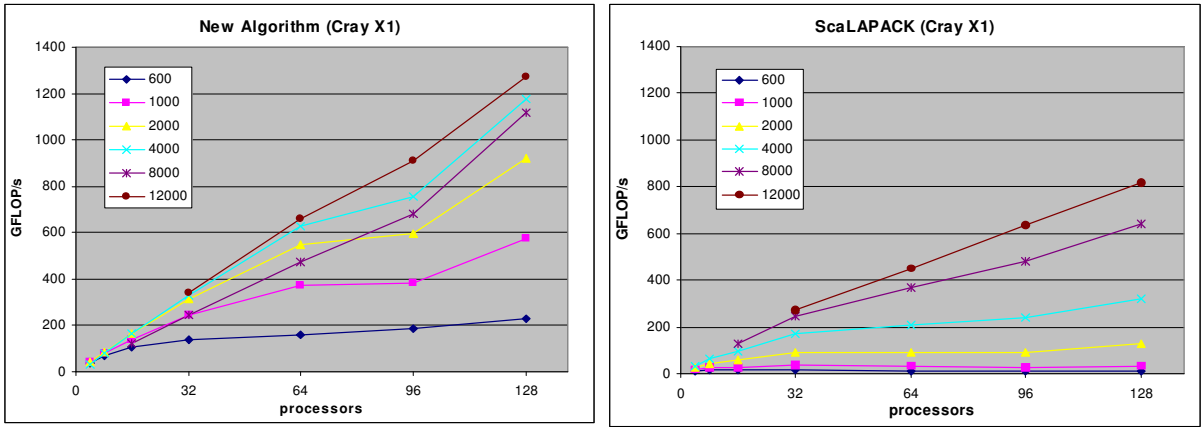

Figure 6: Performance of SRUMMA and *pdgemm* on the Cray X1

4.3 Performance of hybrid protocol

To understand performance advantages of the hybrid protocol, we ran matrix multiplication tests and varied the matrix size and the number of processors (see Figure 7). As expected, the impact of using hybrid protocol is larger for smaller matrices where the communication cost is higher relative to the cost of computations. For example, the hybrid protocol on N = 600 improved performance by as much as 18.2% relative to the baseline SRUMMA implementation.

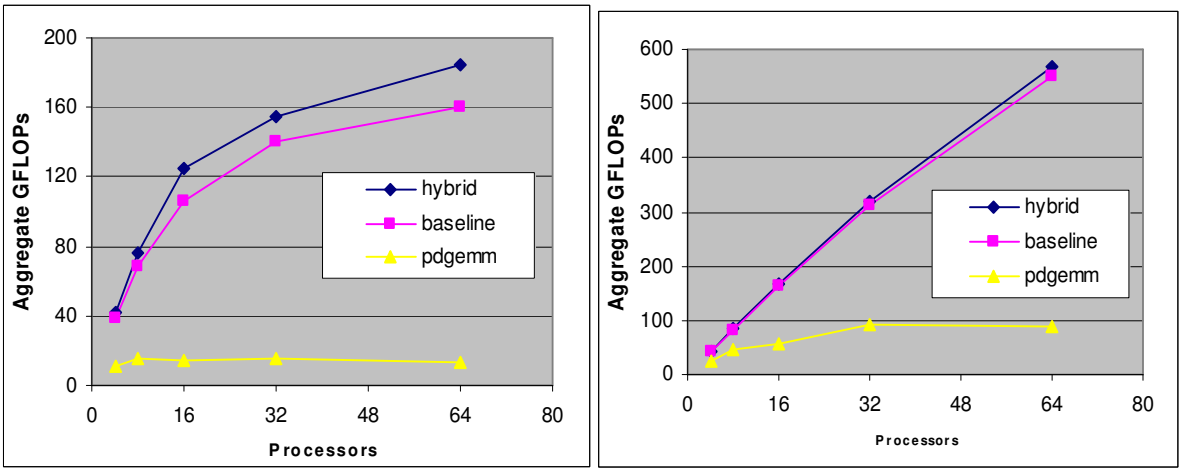

Figure 7: Performance of baseline SRUMMA and hybrid version and *pdgemm* on the Cray X1for matrix sizes N = 600 (left) and 2000 (right)

4. Summary and Conclusions

This paper describes the advantages of using memory/cache hierarchy in bandwidth-intensive computational kernels with parallel matrix multiplication algorithm used as an example. Unlike the other leading parallel algorithms based on message passing, the current approach exploits shared memory. Overall, the algorithm achieved consistent and substantial performance gains over the parallel matrix multiplication in ScaLAPACK. We found that the differences in the architectural

support for shared memory communication on the SGI Altix and Cray X1 require different communication strategies to maximize performance of the parallel matrix multiplication algorithm. Our findings indicate that explicit awareness of the task mapping to exploit cacheability attributes of shared data plays an important role in optimizing performance of this important kernel algorithm. They should also apply to other algorithms that have similar characteristics in terms of bandwidth requirements and locality of reference.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy (DOE) at Pacific Northwest National Laboratory (PNNL).It was supported by the DoE-2000 ACTS project, Center for Programming Models for Scalable Parallel Computing, both sponsored by the Mathematical, Information, and Computational Science Division of the DOE Office of Computational and Technology Research, and the Environmental Molecular Sciences Laboratory. PNNL is operated by Battelle for DOE under Contract DE-AC06-76RL01830.

References

[1] M. Krishnan, J. Nieplocha, ”SRUMMA: A Matrix Multiplication Algorithm Suitable for Clusters and Scalable Shared Memory Systems”, in *IPDPS’2004.*

[2] L. E. Cannon, “A cellular computer to implement the Kalman Filter Algorithm”, Ph.D. dissertation, Montana State University, 1969.

[3] G. C. Fox, S. W. Otto, and A. J. G. Hey, “Matrix algorithms on a hypercube I: Matrix multiplication”, *Parallel Computing*, vol. 4, pp. 17-31. 1987.

[4] G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors*. vol. 1, Prentice Hall, 1988.

[5] G.H. Golub, C.H Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1989.

[6] J. Berntsen, “Communication efficient matrix multiplication on hypercubes:, *Parallel Computing*, vol. 12, 1989.

[7] A. Gupta and V. Kumar, “Scalability of Parallel Algorithms for Matrix Multiplication”, *Proc .Parallel Processing*,’93.

[8] C. Lin and L.Snyder, “A matrix product algorithm and its comparative performance on hypercubes”, *in SHPCC, 199*2.

[9] Q. Luo and J. Drake, "A Scalable Parallel Strassen's Matrix Multiply Algorithm for Distributed Memory Computers", http://citeseer.nj.nec.com/517382.html

[10] S. Huss-Lederman, E. M. Jacobson, and A. Tsao, "Comparison of Scalable Parallel Matrix Multiplication Libraries",*Proc. Scalable Parallel Libraries Conference*’94.

[11] C. T. Ho, S. L. Johnsson, A. Edelman, Matrix multiplication on hypercubes using full bandwidth and constant storage, *Proc.of Distributed Memory Computing Conference*. 1991.

[12] H. Gupta and P. Sadayappan, “Communication Efficient Matrix Multiplication on Hypercubes”, in *Proc. of ACM Symposium on Parallel Algorithms and Architectures*, 1994.

[13] J. Li, A. Skjellum, and R. D. Falgout, “A Poly-Algorithm for Parallel Dense Matrix Multiplication on Two-Dimensional Process Grid Topologies,” *Concurrency, Practice and Experience*, vol. 9(5),’97.

[14] E. Dekel, D. Nassimi, and S. Sahni, “Parallel matrix and graph algorithms”, *SIAM Journal on Computing’81.* vol.10.

[15] S. Ranka and S. Sahni. *Hypercube Algorithms for Image Processing and Pattern Recognition*, 1990.

[16] J. Choi, J. Dongarra, and D. W. Walker, "PUMMA: Parallel Universal Matrix Multiplication Algorithms on distributed memory concurrent computers," *Concurrency:,Practice and Experience*, vol. 6(7),'94.
[17] S. Huss-Lederman, E. Jacobson, A. Tsao, and G. Zhang, "Matrix Multiplication on the Intel Touchstone DELTA", *Concurrency: Practice and Experience*, vol. 6 (7). 1994.
[18] R. C. Agarwal, F. Gustavson, and M. Zubair, "A high performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication," *IBM J. of Research and Development* '94.
[19] R. van de Geijn, R. J. Watts, "SUMMA: Scalable Universal Matrix Multiplication Algorithm" *Concurrency: Practice and Experience*, vol.9(4),'97.
[20] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and, R. C. Whaley, "A Proposal for a Set of Parallel Basic Linear Algebra Subprograms", University of Tennessee, Knoxville, Tech. Rep. CS-95-292, May 1995.
[21] L. S. Blackford et. al., *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, 1997.
[22] J. Choi, "A Fast Scalable Universal Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers", in *Proc. of IPPS*, 1997.
[23] C. Addison and Y. Ren, "OpenMP Issues Arising in the Development of Parallel BLAS and LAPACK libraries", in *Proceedings EWOMP'01*. 2001.
[24] G.R. Luecke, W. Lin, "Scalability and Performance of OpenMP and MPI on a 128-Processor SGI Origin 2000", *Concurrency and Computation: Practice and Experience*, vol. 13, pp 905-928. 2001.
[25] M. Wu, S. Aluru, and R. A. Kendall, "Mixed Mode Matrix Multiplication", in *Proc. IEEE CLUSTER*'02.
[26] T. Betcke, "Performance analysis of various parallelization methods for BLAS3 routines on cluster architectures", John von Neumann-Instituts Computing, Tech. Rep. FZJ-ZAM-IB-2000-15, 2000.
[27] J. L. Träff, H. Ritzdorf, R. Hempel "The Implementation of MPI-2 One-Sided Communication for the NEC SX-5", *Proc. SC*, 2000.
[28] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand" in *ACM ICS,* 2003.
[29] J. Nieplocha, V. Tipparaju, M. Krishnan, G. Santhanaraman, and D.K. Panda," Optimizing Mechanisms for Latency Tolerance in Remote Memory Access Communication on Clusters", *IEEE Cluster Computing'03.*
[30] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2003.
[31] Cray Online documentation. Optimizing Applications on the Cray X1TM System. http://www.cray.com/craydoc/20/manuals/S-2315-50/html-S-2315-50/S-2315-50-toc.html
[32] J. Nieplocha, B. Carpenter, ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems, *Proc. RTSPP IPPS/SDP* 1999.
[33] ARMCI.http://www.emsl.pnl.gov/docs/parsoft/armci
[34] ORNL Evaluation of Early Systems Webpage. http://www.csm.ornl.gov/evaluation
[35] ORNL Tom Dunigan's Evaluation of Early Systems Webpage. http://www.csm.ornl.gov/~dunigan/
[36] SGI Altix family of servers and super clusters http://www.sgi.com/servers/altix/