IS-M921

Mixed Mode Matrix Multiplication

Meng-Shiou Wu
Department of Electrical
and Computer Engineering
Scalable Computing Laboratory
Ames Laboratory, U.S. DOE
Ames, IA 50011
mswu@iastate.edu

Srinivas Aluru
Department of Electrical
and Computer Engineering
Department of Computer Science
Ames, IA 50011
aluru@iastate.edu

Ricky A. Kendall
Department of Computer Science
Scalable Computing Laboratory
Ames Laboratory, U.S. DOE
Ames, IA 50011
rickyk@scl.ameslab.gov

Abstract

In modern clustering environments where the memory hierarchy has many layers (distributed memory, shared memory layer, cache, ...), an important question is how to fully utilize all available resources and identify the most dominant layer in certain computation. When combining algorithms on all layers together, what would be the best method to get the best performance out of all the resources we have? Mixed mode programming model that uses thread programming on the shared memory layer and message passing programming on the distributed memory layer is a method that many researchers are using to utilize the memory resources. In this paper, we take an algorithmic approach that uses matrix multiplication as a tool to show how cache algorithms affect the performance of both shared memory and distributed memory algorithms. We show that with good underlying cache algorithm, overall performance is stable. When underlying cache algorithm is bad, superlinear speedup may occur, and increasing number of threads may also improve performance.

1. Memory Hierarchies in the modern clustering environments

Figure 1 shows the memory hierarchy that exists in most nodes of modern clustering environments. Globally, many nodes are linked together by a high-speed network; inside each node there may be many processors; along with each processor memory access is either to a high speed memory unit "cache" or the low speed "main memory".

In our mixed-mode programming model we use message passing interface, MPI, for the data communication between the global nodes. Inside each MPI process we have two choices, one is to use POSIX threads for creating threads, one or many threads may belong to the MPI process mapped to the node. The other choice is again to use MPI for local processes mapped to all processors of the node. Inside each process we use different algorithms that utilize the cache. Another option for threads that was not explored in this project was to use the OpenMP standard. Based on the specific programming model, we selected several matrix multiplication algorithms on each layer and implemented them.

Bova et. al., [3] determined that, "On a 100-CPU machine, using 100 MPI workers to perform a 100-component harbor simulation is inefficient due to inappropriate load balance. It would be more efficient to have 25 MPI workers create four OpenMP threads for each assigned wave component." In our experiments, we show that even in a perfectly load balanced computation such as matrix multiplication, the overall mixed mode performance is highly affected by cache algorithms.

Our testing platform is the IBM SP system at the National Energy Research Scientific Computing facility [1].

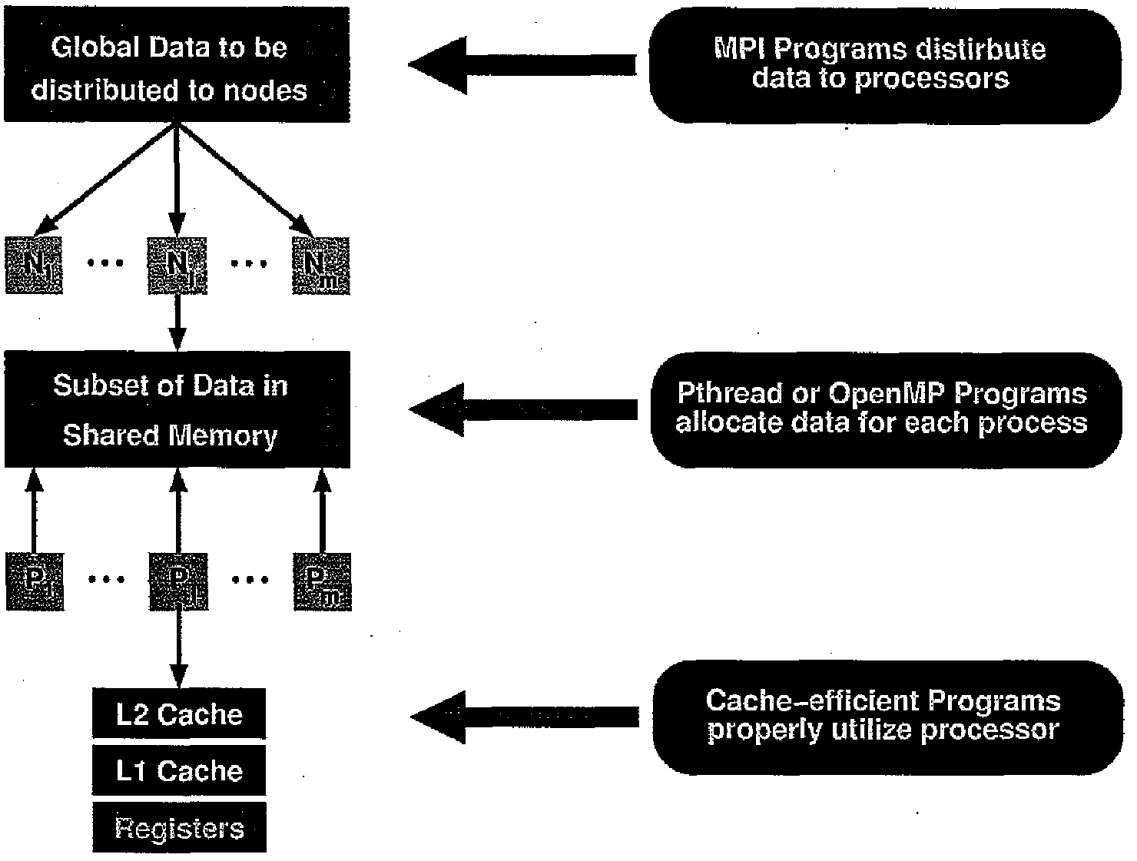

Figure 1. Mixed mode programming model

It is a distributed-memory parallel supercomputer with 184 compute nodes. Each node has 16 POWER3+ processors and at least 16 GBytes of memory, thus at least 1 GBytes of memory per processor. Each node has 64 KBytes of L1 data cache and 8192 Kbytes of L2 cache. The nodes are connected to each other with the IBM proprietary switching network.

2. Cache layer matrix multiplication algorithms

The cache based algorithms used in our research vary from those that have high cache misses to those that effectively use cache. This is by no means a complete coverage of all possible cache algorithms but is representative of those used and taught in the community. Also there is no performance optimization beyond the definition of each algorithm such as what is done in the ATLAS suite [16] where an optimal implementation is produce by balancing tradeoffs between operation count, memory access patterns, etc,. and computed performance metrics. Different optimization techniques can also be found in Crawford and Wadleigh [6] or Dowd and Severance [7].

2.1. Simple three loops algorithm

Figure 2(a), simple three loops algorithm: the figure shows the memory access pattern of this algorithm. This algorithm will incur the most cache misses among all other cache algorithms introduced here. However, it is the algorithm with fewest instruction count. LaMarca and Ladner [11] as well as Chatterjee et. al. [5], mention that for a cache algorithm to get the best performance, the recursion should terminate whenever a block of data fits into cache and then call the algorithm with fewest instruction count. In our implementation, whenever data blocks fit completely in cache, this algorithm is used.

2.2. Blocking C algorithm

Figure 2(b), compute matrix C block by block. Since the algorithm computes according to the square patch of C, the corresponding portions ofmatrices A and B are not required to be square. As the matrix dimensions increase, the

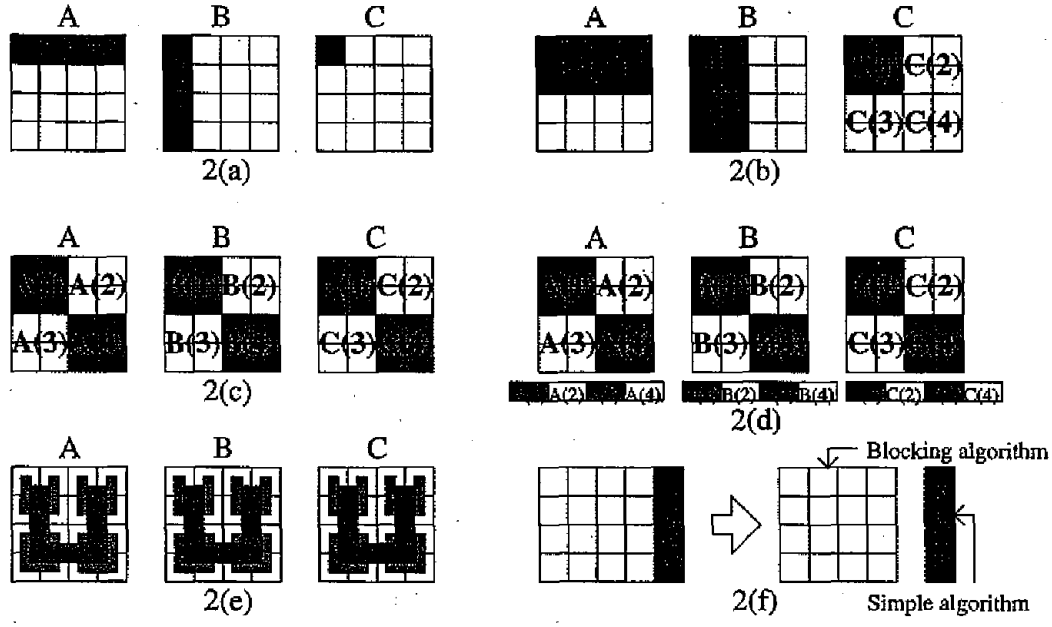

Figure 2. Pictorial Representation of Cache Level Algorithms.

size of A and B patches also increase and thus incur more cache misses. This algorithm performs well when the sizes of matrices are small. If matrix B is transposed first in order to access elements in B consecutively, the performance is much better. The results presented in this work do not transpose matrix B.

2.3. Blocking A algorithm

Figure 2(c), fix a block of A or B: the figure outlines the memory access order of this algorithm. The algorithm keeps a square patch of matrix A in the cache as long as possible. For example, block A(1) is computed with block B(1) and result put into C(1), then A(1) is multiplied by B(2) and stores the data in C(2). A(1) is then swapped out and replaced by A(2) to multiply B(3) and B(4), and so on.

2.4. Transform and blocking A algorithm

Figure 2(d), transform then blocking A: The memory partitioning scheme and computational order for this cache algorithm is the same as the previous algorithm. The only difference is that the layout of all three matrices are transformed before computation starts. The layout of elements in each block is made consecutive by creating a block that is small enough to fit into cache and copying the appropriate portion to the newly allocated block.

2.5. Recursive algorithm

Figure 2(e), recursive layout: Chatterjee and coworkers [5, 4] and Frens and Wise [8] describe the recursive layout of matrix multiplication that the data is transformed into layout according to different space-filling curve order [13]; then computations are done recursively according to that order. Because of the recursion, data has to be a power of 2. Different methods of Chatterjee et. al., [5, 4] and Frens and Wise [8] are used to handle the case when data size is not a power of 2. We implemented a simple version of the "U layout" which works on only square matrices. The blocking shell takes care of the case when data size are not a power of 2.

2.6. Strassen's algorithm

In theory Strassen's algorithm [14, 15] has better run time for matrix multiplication; it use additions and subtractions to reduce the times for multiplication. The algorithm processes data in small blocks recursively, which make it implicitly cache efficient.

Strassen's algorithm, cache oblivious algorithm [9] or Dag-consist algorithm [2] have the advantage that the programs do not need a threshold parameter to adjust the block size. In our experiments we found that recursion down to a single element reduced the performance and terminating the recursion at even a small block size increased the overall performance. For all of our implementations, we assign

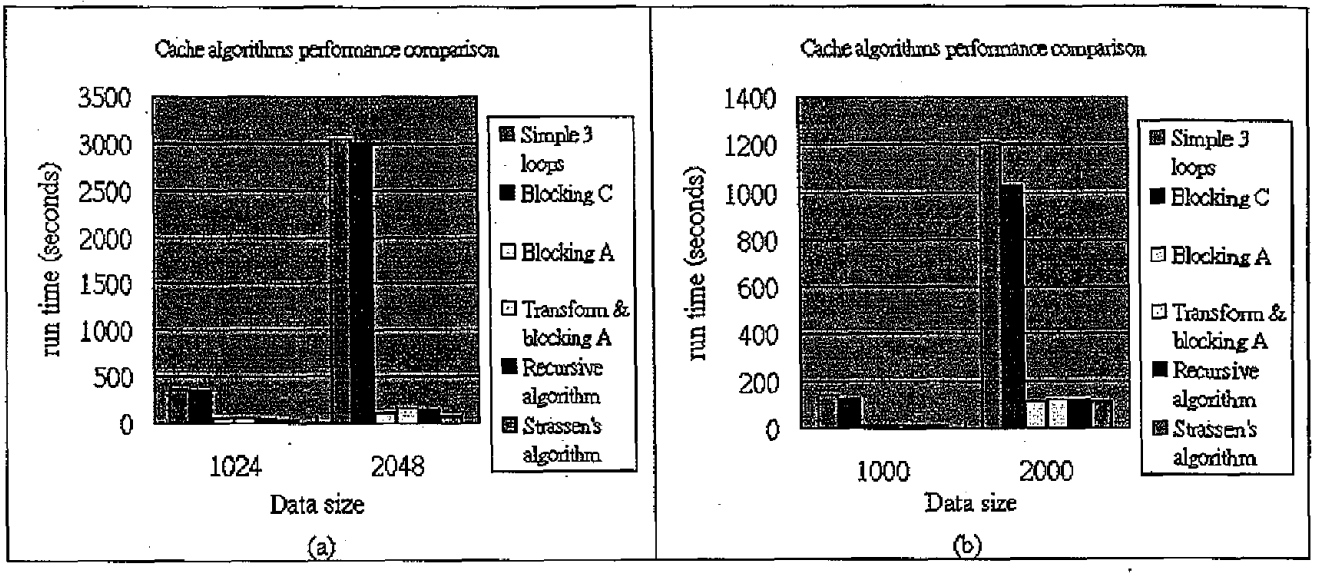

Figure 3. Results of Cache Based Algorithms.

a cache size for each block of 10 Kbytes, which is available for most computers at this point in time.

2.7. Result and performance analysis of cache algorithms

Algorithms such as Strassen's algorithm work on square matrices only, and the dimension has to be a power of 2 to do recursion. When the dimension is not a power of two or the shape of matrices are not square, we use another blocking shell to compute the matrices with square blocks, and leave the rest parts that are not square to simple 3 loops algorithm. Figure 2(f) shows this blocking shell method. In our implementation, three algorithms use this block shell - Strassen's algorithm, recursive algorithm and transform and blocking A algorithm.

Our results are shown in Figures 3. In Figure 3(a), when the matrix size is a power of 2, Strassen's algorithm shows the best performance. However, in a more realistic situation when we have to deal with the dimensions that are not a power of 2, or the shapes of matrices are not square, the cache misses caused by the block shell method and copying overhead reduces the performance of three algorithms - transform and blocking A, recursive algorithm and Strassen's algorithm. The best performing algorithm is to blocking A algorithm.

2.8. Programming issues

One issue to keep in mind is that, the result here is not to show that one algorithm is absolutely better than the others. It just means in our implementation, one algorithm shows better performance than the other. During our research we determined that programming cache algorithms is a nebulous task. The same algorithm be implemented in different ways giving a totally different performance. We use these algorithms, coded directly as describe above, simply as a basis to combine with algorithms in multiple layers of the memory hierarchy. We do *not* give any conclusion about which algorithm is optimal for the cache layer.

3. Shared memory layer matrix multiplication algorithms

We outline four possible shared memory matrix multiplication algorithms that can be easily implemented in the Pthreads programming model, represented in Figure 4. This is by no means an exhaustive set of shared memory algorithms but representative of those that are used by the community.

3.1. Overlapping matrix B

Figure 4(a), overlapping matrix B: the algorithm divides matrix A into several rectangle blocks horizontally accord-

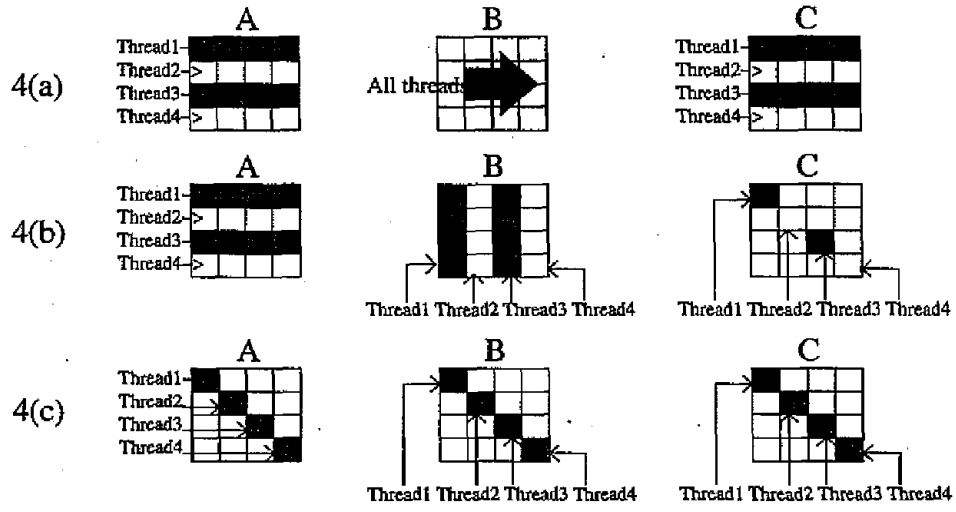

Figure 4. Pictorial Representation of Shared Memory Algorithms

ing to the number of threads. Each thread computes certain rectangle block of matrix A with whole matrix B and produces a complete contribution to a portion of matrix C. The algorithm has the possibility of causing read contention when different threads try to read matrix B.

3.2. Non-overlapping algorithm

Figure 4(b), non-overlapping: the algorithm divides matrix A horizontally and matrix B vertically into blocks. Each thread first computes a block of matrix A multiplied by a block matrix B thus produces a full contribution of a square block of matrix C. In the next stage every thread still use the same block of matrix A, but shifts to another block of matrix B, thus producing another full contribution to a different square block of matrix C. In this way, if all threads are executed concurrently then different threads have less chance of accessing the elements of matrix B at the same time when the number of threads is less than the number of blocks of A and B pairs.

3.3. Blocking algorithm

Figure 4(c), blocking: the algorithm divides all three matrices into smaller square blocks, and each thread computes a square block of matrix A with a square block of B thus producing a partial square contribution to matrix C. The block computation order is the same as element computation order in simple 3-loop algorithm. Care must be taken to update matrix C atomically with mutex locks or assign the contributions required for the full block of C being computed to a single thread.

3.4. Transform and blocking algorithm

Transform and blocking: the algorithm has the same computation order and work on the same shapes of matrices as the previous algorithm, except before the computation begins, all three matrices are transformed into blocks of consecutive data as in the cache algorithm delineated in section 2.4, and each thread works on three square patches with consecutive elements.

3.5. Results and performance analysis of shared memory algorithms

Our results are presented in Figure 5(a) to 5(f). Figure 5(c) to Figure 5(f) show that, with a good underlying cache algorithm, the performance of all four shared memory algorithms are similar. An increase in the number of threads does not substantially affect the overall performance.

On the other hand, Figure 5(a) and figure 5(b) show that, a "bad" cache algorithm combined with a shared memory algorithm that is insensitive to cache, has poor performance as the number of threads is increased. However, when a cache sensitive shared memory algorithm is combined with a "bad" cache algorithm, there is a performance gain with an increased number of threads. The performance can almost meet that of a good cache algorithm. This is due to the fact that as we increase the number of threads the smaller block size per thread actually fits into cache thus reducing the cache misses.

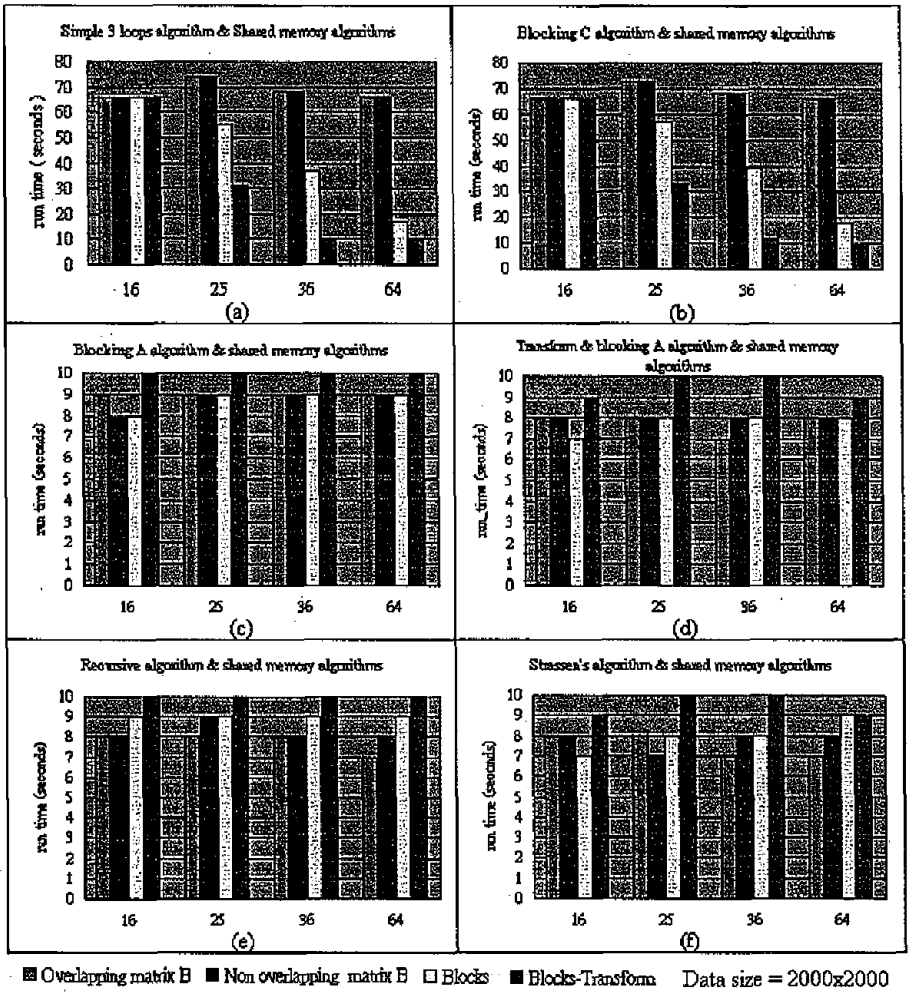

Figure 5. Results of Shared Memory Based Algorithms

4. Distributed memory layer matrix multiplication algorithms

Here we focus on two common distributed matrix multiplication algorithms. Many more are available but these represent a common denominator of many algorithms.

4.1. Broadcasting algorithm

Two dimensional broadcasting algorithm: According to the number of physical nodes and physical grid, if the physical grid is $p \times q$, then matrices are partitioned into least common multiples of p and q parts. Each node takes turns broadcast the part of matrix A vertically or B horizontally or both, and computes according to the data it receives.

4.2. Cannon's algorithm

The algorithm is described in almost every parallel algorithm book such as Kumar et. al., [10]. We use a generalized Cannon's algorithm similar to Lee and Fortes [12] instead of the original algorithm that requires the number of processors to form a certain square. In our implementation, we first form a grid using the available processors, then depending on the shape of the grid, we distribute data accordingly. If the grid is square, we distribute data as in Cannon's original algorithm. If the grid is rectangular, find the least common multiple of two dimensions, use the least common multiple as the dimension of a virtual square grid and then distribute data according to this virtual grid.

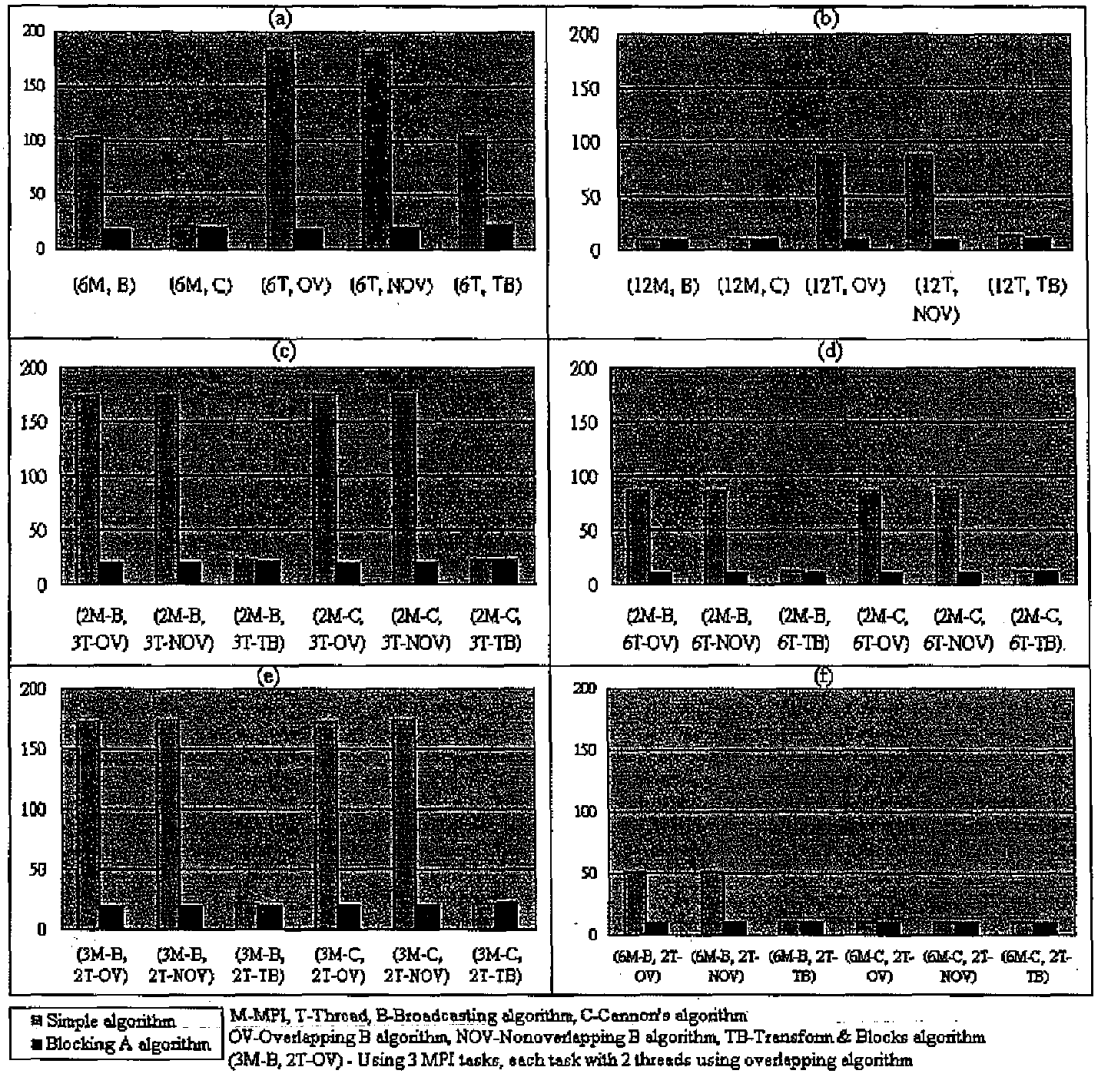

Figure 6. Results of Mixed Mode Algorithms

4.3. Result and performance analysis of Distributed Memory and Mixed Algorithms

In our observations, when using the same number of processors, MPI algorithms exhibit better performance than Pthreads algorithms when the underlying cache algorithm is "bad." The reason is that the distributed memory MPI algorithms always partition data into smaller blocks while shared memory algorithms work on a bulk data, and thus causing more cache misses.

However, with good underlying cache algorithm, the choice of how many MPI tasks combined with how many threads does not seem to be so important. Figure 6(a) to 6(f) show our result of computing matrices of size 2000x2000 using 6 and 12 nodes, with different combinations of algorithms, MPI tasks and thread tasks.

Figure 6(a) and 6(b) show that when bad cache algorithm combines with distributed algorithms or bad cache algorithm combines with shared memory algorithms, run time is not stable and doubling number of processors sometimes has superlinear speedup. Figure 6(c) and 6(d) show mixed algorithms of three layers and doubling number of processors have a speedup of 2. Figure 6(e) and 6(f) show mixed algorithms together and doubling number of processors when superlinear speedup happen again. With good underlying cache algorithm, performance are stable.

From Figure 6(a) to 6(f), we observed that, doubling the number of processors, good cache algorirhms give speedup of two while bad cache algorirhms combines with algorithms from other layers that do not partition data small

	1000	2000	4000	8000	16000
Performance gain from improving MPI algorithm	0.43	0.244	1.348	8	12
Performance gain from improving cache algorithm	0.07	0.251	114	1018	> 6000

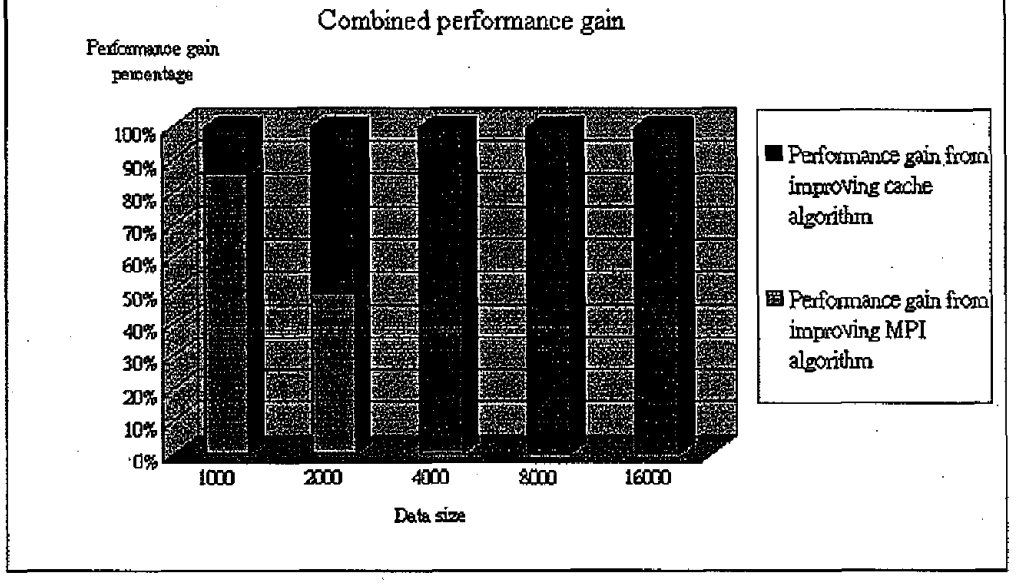

Figure 7. Percentage of Performance Gain from Cache Layer and Distributed Memory Layer

enough, also give speed up of two. On the other hand, performance of bad underling cache algorithms have superlinear speedup when data is partitioned small enough by algorithms from the other two layers.

These results show that, without a good cache algorithm, timings fluctuate. Different combinations of MPI tasks and numbers of threads exhibit different performance. The main dependence is on how the algorithms divide data and thus make good use of cache as the chunk size decreases. On the other hand, if the underlying cache algorithms is "good," the way data is partitioned matters much less.

We use a simple model to show the effects of cache algorithms:

$$T_{Total} = T_{Comp} + T_{Comm} + T_{Penalty}$$

where T_{Total} is the total time, T_{Comp} is the computational time, T_{Comm} is the communication time, and $T_{Penalty}$ is the time associated with cache misses. $T_{Comp} + T_{Comm}$ is the normal "total time" for many parallel computational models as described in Kumar et. al., [10], and for $T_{Comp} + T_{Penalty}$ model we refer to the two layers model of Matteo et. al., [9]. For matrix multiplication, communication cost is at most $O(n^2)$ while cache misses range from $O(n^2)$ to $O(n^3)$ depending on the algorithms. When data size are small, we can almost ignore cache misses penalty; when data size increases, cache misses penality becomes a factor that affects total run time. When the data size is huge, $O(n^3)$ cache misses is now the bottleneck for the performance.

Figure 7 shows the fraction of improvement coming from modifying the underlying cache algorithm for the distributed memory algorithms. We measured the total performance gain and the percentage that distributed layer algorithms and cache layer algorithms contribute. The data size are from 1000 to 16000, using 64 nodes. Shared memory algorithms are not used here since we don't have a node of 64 processors. As shown in figure 7, when data size are small, all data block can fit into cache, the major improvement is from good distributed algorithms that reduce communication time. When data size increase, data block size also increase, incurring more cache misses, and cache algorithms became dominate contributer of performance gain. Eventually, cache layer algorithms contribute almost all performance gain when data size are very large.

5. Conclusion

In this paper we use different matrix multiplication algorithms on different layers to show how performance will be affected in mixed mode programming without a good cache algorithm, even when the work load is perfectly balanced. Since the core of parallel computations are still sequential computations, to improve the overall performance, not only do we need a model to utilize memory on every layer, but also good sequential core algorithms to achieve high performance. From our experiments, we believe that cache algorithms play a dominate role in many high performance computations, especially when processing large segments of data. If the computations is divided into many stages, and each stages only works on small data size, improving distributed algorithms improve the performance since cache misses do not matter much on computing small data size. On the other hand, if the computation has to work on large chucks of data, it is important to combine a good cache algorithms with an increase in the number of processors. Furthermore, parallel algorithms with "bad" underlying cache algorithms utilized in a mixed programming mode, the saturation of the thread space beyond the total number of computing threads equal to the number of available processors should provide a modest performance enhancement.

6. Acknowledgements

This work was performed under the auspices of the Department of Energy under contract under contract W-7405-ENG-82 at Ames Laboratory operated by the Iowa State University of Science and Technology. Funding was provided by the Mathematical, Information, and Computational Sciences division of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy. We would also like to acknowledge the support of Iowa State University and the University Research Grant program which funded a segment of this work. This research was performed in part using computational resources in the Scalable Computing Laboratory which were partially donated from IBM Corporation under the Shared University Research grant program. This research also used computational resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098 with the University of California.

References

[1] http://hpcf.nersc.gov/computers/SP.

[2] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the Eighth ACM Symposium on Parallel Algorithms and Architectures(SPAA)*, pages 297–308, Pauda, Italy, June 1996.

[3] S. W. Bova, C. P. Breshears, H. Gabb, B. Kuhn, B. Magro, R. Eigenmann, G. Gaertner, S. Salvini, and H. Scott. Parallel Programming with Message Passing and Directives. *Computing in Science and Engineering*, 3(5):22–37, 2001.

[4] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of 1999 ACM International Conference on Supercomputing*, pages 444–453, Rhodes, Greece, June 1999.

[5] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of the 11th ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, Saint-Malo, France, June 19 1999.

[6] I. L. Crawford and K. R. Wadleigh. *Software Optimization for High Performance: Creating Faster Applications*. Prentice Hall PTR, 2000. ISBN:0130170089.

[7] K. Dowd and C. Severance. *High Performance Computing, 2nd Edition*. O'Reilly & Associates, 1998. ISBN:156592312X.

[8] J. D. Frens and D. S. Wise. Auto-blocking matrix-multiplication or tracking blas3 performance from source code. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 206–216, Las Vegas, NV, June 1997.

[9] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundation of Computer Science(FOCS'99)*, pages 285–298, 1999.

[10] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms*. Addison-Wesley Pub Co., 1994. ISBN:0805331700.

[11] A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 370–379. SIAM press, 1997.

[12] H.-J. Lee, J. P. Robertson, and J. Fortes. Generalized cannon's algorithm for parallel matrix multiplication. In *Proceedings of the 11th ACM International Conference on Supercomputing*, pages 44–51, Vienna, Austria, July 1997.

[13] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994. ISBN:0387942653.

[14] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[15] M. Thottethodi, S. Chatterjee, and A. R. Lebeck. Tuning strassen's matrix multiplication for memory efficiency. In *Proceedings of SC98, High Performance Networking and Computing*, Orlando, Florida, November 1998.

[16] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. Technical Report UT-CS-97-366, University of Tennessee, 1997.