UCRL-ID-136661

Improving Cache Utilization of Linear Relaxation Methods: Theory and Practice

F. Bassetti, K. Davis, M. Marathe and D. Quinlan

December 1, 1999

Improving Cache Utilization of Linear Relaxation Methods: Theory and Practice

Federico Bassetti, Kei Davis, Madhav Marathe, and Dan Quinlan

Los Alamos National Laboratory, Los Alamos, NM 87545, USA,
and
Lawrence Livermore National Laboratory, Livermore, CA 94550, USA,
kei,fede,marathe@lanl.gov,dquinlan@llnl.gov

Abstract. Application codes reliably achieve performance far less than the advertised capabilities of existing architectures, and this problem is worsening with increasingly-parallel machines. For large-scale numerical applications, stencil operations often impose the greater part of the computational cost, and the primary sources of inefficiency are the costs of message passing and poor cache utilization. This paper proposes and demonstrates optimizations for stencil and stencil-like computations for both serial and parallel environments that ameliorate these sources of inefficiency. Additionally, we argue that when stencil-like computations are encoded at a high level using object-oriented parallel array class libraries, these optimizations, which are beyond the capability of compilers, may be automated.

1 Introduction

Modern supercomputers generally have deep memory hierarchies comprising a small fast memory (registers) and increasingly large, increasingly slow memories. Five such levels exist for the Los Alamos National Laboratory ASCI Blue machine: each processor sees a register file, level one (L1) and level two (L2) caches, main memory, and remote main memory. Future architectures may rely on yet deeper memory hierarchies.

It is clear that the realization of modern supercomputer potential relies critically on program-level cache management via the staging of data flow through the memory hierarchy to achieve maximal data re-use when resident in the upper memory levels. In general it is important to utilize the spatial- and temporal locality of reference in a problem. Data exhibits temporal locality when multiple references are close in time; spatial locality when nearby memory locations are referenced. This is important because memory access time (from the same level) may not be uniform, e.g. in loading a cache line or memory page.

Much research has been devoted to studying pragmatic issues in memory hierarchies [3, 9–13] but relatively little to the basic understanding of algorithms and data structures in the context of memory access times. Recently hierarchical memory models have been proposed [1, 2] and efficient algorithms for a number of basic problems have been proposed under these models.

A promising approach for modeling memory hierarchies as a part of the computational model was undertaken by Aggarwal et al. [1, 2]. In their Hierarchical Memory Model (HMM) are an unbounded number of registers R_1, R_2, $R_3, \ldots$, each of which can store a datum of fixed type. The operations are similar to those of a RAM [4], except that accessing register R_i takes time roughly $\log R_i$. The HMM can be viewed as having a hierarchy of N memory levels where for $0 \leq i \leq N-1$ there are 2^i memory locations requiring time i to access. The authors prove that the HMM is fairly robust. They and others have devised algorithms for a variety of problems using this model of computation. In the context of the LANL ASCI Blue Mountain machine it is easy to why the model is valid: each level of the memory hierarchy is close to an order of magnitude larger and slower than its predecessor-- a coarsened realization of the HMM.

2 Background and Relationship to Previous Work

Relaxation is a well-known technique in the solution of many numerical problems. We follow the notation of Leiserson et al. [6] for the rest of this section to describe the ideas. A prototypical problem in this context can be described by a graph $G(V, E)$ in which each vertex $v \in V$ contains a numerical value x_v which is iteratively updated. At iteration t, of the linear relaxation process, the state of v is updated using the equation $x_v^{(t)} = \sum_{(u,v) \in E} A_{uv}^{(t)} x_u^{(t-1)}$ where $A_{uv}^{(t)}$ denotes the relaxation weight of the edge (u, v). This equation may written using matrix notation as $x^{(t)} = A^{(t)} x^{(t-1)}$, where $x^{(t)} =< x_1^{(t)}, x_2^{(t)}, \ldots, x_{|V|}^{(t)} >$. The goal of the relaxation is to compute a final state vector $x^{(T)}$ given an initial state vector $x^{(0)}$. In this paper, we will assume that A does not change over time.

A simple and effective way to compute relaxation is to update the state of each vertex in the graph according to the equation given above and then repeat this step until we obtain $x^{(T)}$. This is inefficient in today's computing architectures where the memory system typically has two or more levels of cache between the processor and the main memory. The time required to access a data value from memory depends crucially on the location of the data; the typical ratios of memory speeds are $1 : 10 : 100$ between the first level cache, second level cache and the main memory. As CPUs get faster these processor-memory gaps are expected to widen further. As a result it is important to design transformations for applications (in this case relaxation methods) that take into account the memory latency. Such algorithms seek to stage the computation so as to exploit spatial and temporal locality of data. Common compiler transformations such as blocking mainly address spatial locality. As will be shown, significant performance can be obtained by addressing temporal locality.

These optimizations form a specific instance of a more general optimization that originates in older out-of-core algorithms, main memory in this case is treated as slower storage (out-of-core). In this case we treat the instance of a stencil operation on a structured grid as a graph and define covers for that graph. The covers define localized regions (blocks) and form the basis of what we will define to be τ-*neighborhood-covers*. The general idea behind applying this transformation to structured grid computations is to cover (or *block* or *tile*) the entire grid by smaller sub-grids, solving the problem for each subgrid sequentially.

Let n be the number of grid points in a two dimensional square domain, and A be a $\sqrt{n} \times \sqrt{n}$ grid. Let the size of L_1 cache be M. Now consider solving the linear relaxation problem for a subgrid S of size $k \times k$. In the first iteration, all the points in S can be relaxed. After the second iteration, points in the grid of size $(k-2) \times (k-2)$ have the correct value. After continuing the procedure for τ iterations, it follows that a subgrid of size $(k-2\tau) \times (k-2\tau)$ has been computed up to τ time steps. Thus we can now cover the $\sqrt{n} \times \sqrt{n}$ by $\frac{\sqrt{n} \times \sqrt{n}}{(k-2\tau)\times(k-2\tau)}$ which can be rewritten as $\frac{n}{(k-2\tau)^2}$. The total number of loads into the L_1 cache is no more than $k^2 \frac{n}{(k-2\tau)^2}$. In order to carry out the relaxation for T time units, the total number of L_1 loads is no more than $k^2 \frac{n}{(k-2\tau)^2} \frac{T}{\tau}$. By setting $k = \sqrt{M}$, and $\tau = \sqrt{M}/4$, we can obtain an improvement of $O(\sqrt{M})$ in the number of L_1 loads over a naive method (which would take $O(Tn)$ time).

In practice, for the solution of elliptic equations using multigrid methods on structured grids, τ is typically a small constant between 2 and 10 and thus the asymptotic improvement calculated above does not directly apply; nevertheless the analysis shows that we can get constant factor improvements over the naive method in terms of the memory access. The naive method we refer to here is the blocking introduced by the compiler. Of course less efficient methods could be proposed (e.g. local relaxation methods) which would permit significantly larger values of τ, but these less efficient solution methods are not of practical interest. The idea can be extended in a straightforward way to the cases when the value at a grid point is calculated by using not only the neighboring values but all neighbors that are a certain bounded distance away.

In the above calculations we have chosen to recalculate the values on the boundary of each subgrid since this does not give us any additional asymptotic disadvantage. In practice, however, these values are maintained in what we call *transitory* arrays. This introduces additional complications, namely the need to ensure that the transitory arrays are resident in the L_1 cache. This implies that for a 2-dimensional case we need to maintain arrays with total size $4k + 4(k-2) + \cdots + 4(k-2(\tau-1)) \sim O(k^2)$. Although asymptotically this is still the same as the size of our subgrid, the constants need to be carefully calculated in order to determine the best parameter values. The memory requirements for the transitory array could be reduced slightly by lexicographically ordering the subgrids in terms of the grid coordinates. This allows us to throw away the left and the top boundary of a subgrid.

These ideas can be extended to general graphs that capture the dependency structure. For this purposes, given a graph $G(V,E)$ let $N^\tau(v) = \{w \mid d(w,v) \leq \tau\}$. Here $d(w,v)$ denotes the distance in terms of number of edges from v to w.

Definition. τ-**Neighborhood Cover**: Given a graph $G(V,E)$, a τ-neighborhood cover $\mathcal{G} = \{G_1, \ldots, G_k\}$ is set of subgraphs G_i, $1 \leq i \leq k$, such that $\forall v \in V$, there exists a G_i with the property that $N^\tau(v) \subseteq V_i$.

Given a graph $G(V, E)$, a *sparse* τ-neighborhood cover for G is a τ-neighborhood cover with the additional properties $\forall i,\ 1 \leq i \leq k,\ |G_i| \leq M$, and $k = O(|E|/M)$. Note that a vertex v can belong to more than one subgraph G_i but for the purposes of computing its value there is at least one subgraph G_v that may be regarded its "home", meaning that the state $x_v^{(\tau)}$ at a vertex v at time τ can be calculated by just looking at the graph G_v. Existence of sparse neighborhood covers for a graph G imply the existence of memory efficient algorithms for linear relaxation on G. Specifically, given a graph G with a sparse τ-neighborhood cover, it is possible to complete T steps of relaxations using $O(\frac{T}{\tau}|E|)$ loads in the L_1 cache as opposed to $O(T|E|)$ loads in the naive implementation. This can be seen by observing that τ steps of the linear relaxation can be completed using no more than $O(M)O(|E|/M)$ L_1 loads. Certain well known classes of graphs (e.g. graphs for finite difference computational grids) have sparse neighborhood covers.

The concept of graph covers closely resembles the concept of *tiling* that has been extensively used in the past to reduce the synchronization and memory access costs. Tiling tranformation is one of the primary transformations used to improve data locality. Tiling generalizes two well-known transformations, namely, *strip-mine and interchange* and *unroll and jam*. Graph covers can be thought of as a generalization of tiling. In order to see this, we recall some basic definitions from compiler theory [14, 7] to which we refer the the reader, and to the referenced cited therein, for more details on this topic.

Iteration Space. A set of n nested for loops are represented as a polytope $\mathbf{Z}^b$ with the bounds of the polytope corresponding to the bounds placed on the loops. Each iteration now corresponds to a point in $\mathbf{Z}^n$. We think of these points as vertices of a graph called the *dependence graph*. The vertex is identified by a vector $\boldsymbol{p} = (p_1, \ldots, p_n)$, where p_i is the value of the i^{th} loop index in the nest while counting from outermost to the innermost loop. It is easy to see that each axis represents a loop and each vertex represents an iteration that is performed. With this notation, it is clear that an iteration $\boldsymbol{p} = (p_1, \ldots, p_n)$ has to be executed before another iteration $\boldsymbol{q} = (q_1, \ldots, q_n)$ iff $\boldsymbol{p}$ is *lexicographically smaller* than $\boldsymbol{q}$[1] (denoted $\boldsymbol{p} \prec \boldsymbol{q}$). This intuition can be captured via the notion of *distance and dependence vectors or edges.* Viewing the points corresponding to the iterations in $\mathbf{Z}^n$ as vertices of a graph, we have a directed edge from a vertex $\boldsymbol{p}$ to a vertex $\boldsymbol{q}$ with label $\boldsymbol{d}$ if $\boldsymbol{p} \prec \boldsymbol{q}$ and $\boldsymbol{d} = \boldsymbol{q} - \boldsymbol{p}$. The directed graph (called the dependence graph) assigns a partial order to the vertices corresponding to the iterations; the important point is that any complete ordering of the original vertices that respect the partial ordering is a feasible schedule for executing of the iterations.

Tiling. In general tiling maps a n-deep nested loop structure into a $2n$ deep nested loop structure with only a small fixed number of iterations. Graphically, tiling can be viewed as way to break the iteration space into smaller blocks with the following conditions:

1. The blocks are disjoint.

[1] p $\prec$ q iff $1 \leq i \leq n$, $p_i \leq q_i$.

2. The individual points in each block and the block themselves are ordered in such a way so as to preserve the dependency structure in the original iteration space; i.e. if $\boldsymbol{p} \prec \boldsymbol{q}$, then $\boldsymbol{p}$ is executed before $\boldsymbol{q}$. Stated another way, the complete ordering for scheduling the execution of the iterations should respect the partial ordering given by the dependence graph.
3. The blocks are by and alreg rectangular, excepting when the iteration spaces are themselves of a different shape (e.g. trapezoidal, triangular).
4. Typically, if the original problem had a set of n loop indices, the transformation yields $2n$ indicies (one new index for each old index) such that the bounds on original n indices do not depend on each other, but rather depend only on the corresponding new index created in the process of the transformation.

In contrast, graph covers create a set of smaller subgraphs and yield a transformed instance in which *either* certain iterations are revisited (i.e. steps at those times are recalculated) thus creating overlapping tiles, *or* it stores the intermediate values and avoids the extra computation. Intuitively, tiling based on graph covers creates trapezoidal tiles, and uses temporary storage to make sure that the dependency graph partial order is maintained.

In this paper Jacobi relaxation is used as an example relaxation code, using a two-dimensional array and a five-point stencil (diagonal elements are excluded). Such computations appear as parts of more sophisticated algorithms as well (e.g. multigrid methods). A single iteration or *sweep* of the stencil computation is of the form

```
for (int i=1; i != I-1; i++)
 for (int j=1; j != J-1; j++)
   A[i][j] = w1*A[i-1][j] + w2*A[i+1][j] + w3*A[i][j-1] + w4*A[i][j+1];
```

Typically several sweeps are made, with A and B swapping roles to avoid copying. It is easy to see that the dependence graph of this loop is not strict. Thus the tiling theory developed elsewhere [14, 15, 7] cannot be applied directly to yield the type of transformations we obtain.

In typical serial or parallel C++ array class library syntax the statement is

```
  A(I,J) = w1*A(I-1,J) + w2*A(I+1,J)+ w3*A(I,J-1) + w4*A(I,J+1);
```

While the transformations we will detail are general, it is the prototypical array class syntax that is targeted for such optimizations.

3 Performance Results

Our technique is compared to a naive implementation relying on an optimizing compiler. The performance data were gathered on SGI Origin 2000 system comprising 128 MIPS R10000 processors running at 250 Mhz. Each processor has a split L1 cache with 32 Kbytes for instructions and 32 Kbytes for data. L2 is a unified 4 Mbytes. L1 and L2 are both 2-way associative. The ratio of L1 and L2 line size 1:4. The system has 250 Mbytes per processor. The test codes were compiled using the MIPSpro C++ with O3 optimization. Performance data were gathered via the hardware performance monitors available on the processors. The test code is a Jacobi iterative solver based on the code fragment above.

3.1 Single Processor Performance

Ideal performance is obtained when the entire working set fits in primary cache; this is shown in Figures 1, 2, and 3. Three metrics are used: *cycles* to describe the overall execution; *primary cache data misses* to describe the impact of memory penalty, and *flops* to describe how efficiently, compared to the ideal, the test code performs. All results are based on the average cost of one iteration. Figure 1 shows that the cost of each iteration generally decreases as the number of iterations increases: all misses are first-time-load misses— once a datum is loaded into cache it remains resident for the duration of the computation—as shown in Figure 2, so increasing the number of iterations amortizes the cost of first-time-load misses.

The vast majority of scientific applications carrying out meaningful computations have a working set that is many times larger than any cache. Figure 4 shows the performance of the naive implementation of the test. The cost of each iteration is on average the same regardless of the number of iterations. Again, this is a consequence of the cache miss behavior; Figure 5 confirms that every iteration entails the same number of misses regardless of the number of iterations. The efficiency is less than half of the idealized cache-resident version as shown in Figure 6.

Most scientific applications have inherent locality; this is a simple consequence of array-based computation. Modern compilers are able to detect and exploit some locality. Compilers exploit locality only within one iteration; reuse is minimal or nil between iterations. Exploiting temporal locality with the technique presented in the previous sections demonstrably reduces the difference in performance from the ideal cache-resident version. Figures 4, 5, and 6 give performance results for optimized version of the test code. In all the figures *2D-blocked* and *1D-blocked (square)* represent cache blocking performed using, respectively, a two dimensional tile and a one dimensional tile (a stripe) for a square two dimensional problem. The figures show that the *2D-blocked* version performs better than the others. Quantitatively, it is possible to observe a factor of two improvement compared to the naive implentation. The explanation for such differences in performance is in the number of misses, as show in Figure 5. While in the naive version the number of misses is the same per iteration, the optimized 2D-blocked one amortizes the number of misses. This says that in the optimized code the number of misses performed during the first iteration are the significant ones, besides those the number of misses performed is significantly less, such that the total count of misses could appear, relatively, independent of the number of iterations executed. There are noticeable similarities between the performance of the 2D-blocked version and the ideal (cache resident) version: the cost in cycles per iteration decreases when increasing the number of total iterations; the number of misses per iteration decreases when increasing the number

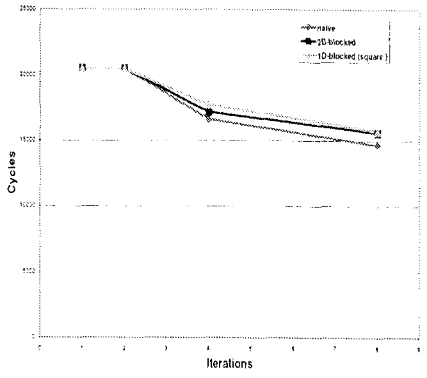

Fig. 1. Average Cycles

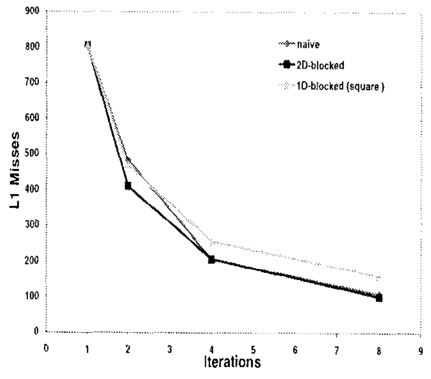

Fig. 2. Average L1 Misses

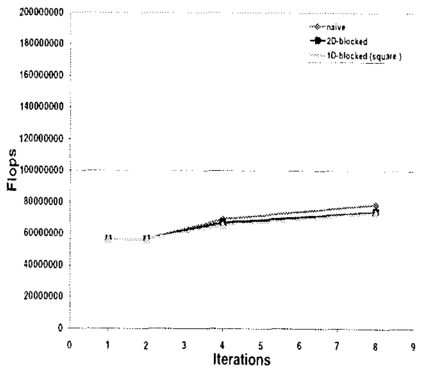

Fig. 3. Average Flops

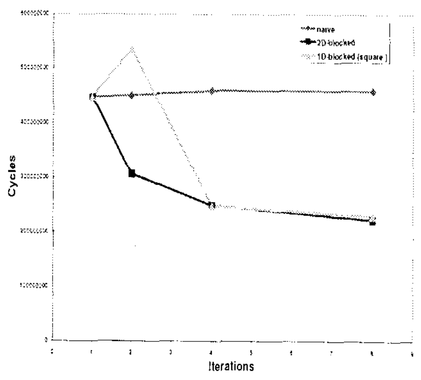

Fig. 4. Average Cycles

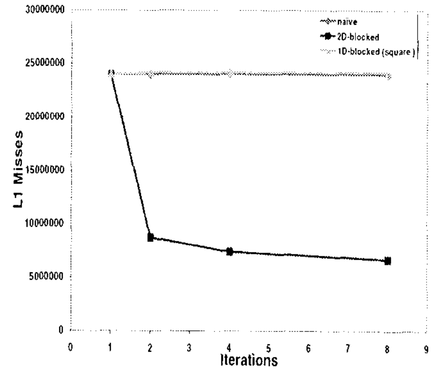

Fig. 5. Average L1 misses

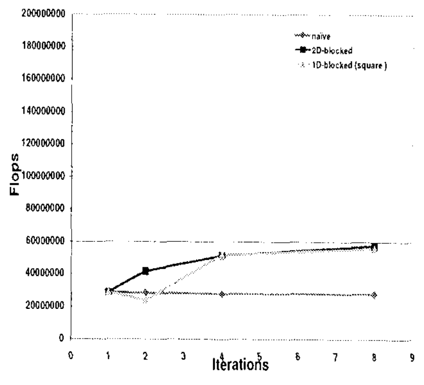

Fig. 6. Average Flops

of total iterations; the efficiency, in flops per second, increases when increasing the number of iterations. Nevertheless, it appears that there is still room for further investigation and possible improvement since the relative flop rates show that there is still a difference between ideal and 2D-blocked performance. The performance obtained with the 1D-blocked version, overall, can be considered similar to the 2D-blocked one with the exception of primary cache misses. For a large problem size, a tile is a stripe that cannot fit in L1 or L2 cache. However, a stripe offers more reuse potential for L2 cache.

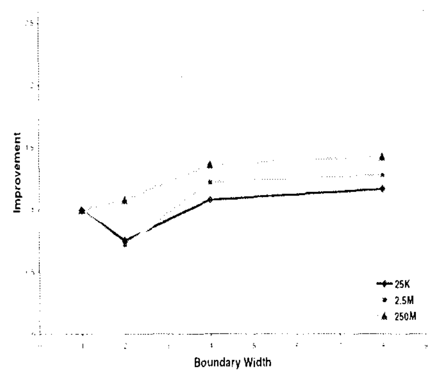

Fig. 7. Improvement by widening boundaries on 8 processors. Problem size is given in number of points.

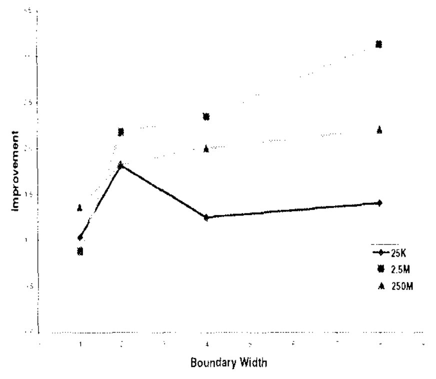

Fig. 8. Improvement by the optimized implementation on 8 processors. Problem size is given in number of points.

The performance figures show that the transformation is in fact an optimization; validation on different platforms is needed. Of particular interest is the comparison between 2D-blocked and 1D-blocked implementations. In theory, a 2D-blocked implementation should outperform an equivalent 1D-blocked one. In practice, it appears that this theoretical difference in performance is not noticeable. With the implementation of the temporal locality optimization the loop structure, if compared to the naive one, has been significantly modified: loops have been added and a temporary data structure is needed. We believe that while the technique presented has a great impact on the memory access pattern, improving by factor of 5-10 the number of misses, overall performance improves by a smaller factor because of the a more complicated loop structure that makes pipelining less efficient. This effect might be the limiting factor for the temporal blocking technique. The maximum achievable improvement factor is still under investigation.

3.2 Multiple Processors

In a parallel environment the arrays are typically distributed across multiple processors. To avoid communication overhead for calculations near the boundaries of the segments of the the arrays, space is traded for time by creating *ghost boundaries*—read-only sections of the other processors' array boundaries. In the parallel case the C++ array statement transparently abstracts the distribution of data, the parallelism in the execution of the operations on the data, and the message passing required along the partitioned edges of the distributed data.

The basic idea is to trade computation, which is relatively cheap, for communication, which is expensive, to reach an optimal compromise, by increasing the amount of data at block boundaries, making possible multiple iterations over the data before communication is required. Figure 7 demonstrates that this strategy is beneficial to overall performance. This technique is described elsewhere [5].

In this work we use the message passing paradigm, using the *native* implementation of MPI. Figure 8 shows improvement achieved by the optimized code. In almost all of the cases studied the optimized version is between two and three times faster than the naive implementation. The improvement that has been demonstrated for single processor is maintained, undegraded, in the multiprocessor case. We have verified this effect up to 64 processors.[2]

4 Automating the Transformations

To be of practical use an optimizing transformation must be automated. In the context of C++ array classes it does not appear possible to provide this sort of optimization within the library itself because the applicability of the optimization is context dependent—the library can't know how its objects are being used. Two mechanisms for automating such optimizations are being actively developed: the use of *expression templates* by others, and a source-to-source transformation system (a pre-processor), which we are currently developing. The ROSE II preprocessor is a mechanism for C++ source-to-source transformation, specifically targetted at optimizing (compound) statements that manipulate array class objects. ROSE II is described in detail elsewhere [8].

5 Conclusions and Future Work

We posit that current optimizations for stencil-based applications are inadequate for which desirable optimizations exist that cannot reasonably be expected to be implemented by a compiler. One such optimization for cache architectures has been detailed and demonstrated to give a factor of two improvement in performance in a realistic setting. The transformation is language-independent, though it is demonstrated only for C code.

[2] Since a multidimensional space graph is needed to have a complete picture only a portion of those are presented.

The use of object-oriented frameworks is a powerful tool, but performance is generally less than that of FORTRAN 77. We expect that in the future one will use such object-oriented frameworks because they represent *both* a higher-level, simpler, and more productive way to develop large-scale applications *and* a higher performance development strategy. We expect higher performance because the representation of the application using the higher level abstractions permits the use of new tools such as the ROSE II optimizing preprocessor.

There is scope for generalizing the algorithm to N-dimensional tiling with M ($M \leq N$) data partitioning. While the analytical performance predictions agree with the empirical data, it will be revealing to make use of sophisticated architectural simulators to both study potential performance gains on proposed hypothetical architectures as well as validate the model. Use of different compilers may impact performance; register allocation has a significant impact on both overall performance, and optimized relative to unoptimized performance.

Parallel optimization is a subject of ongoing investigation. Currently only explicit message passing using the MPI library has been evaluated; currently other paradigms such as threads and OpenMP pragmas are being investigated.

References

1. A.K. Chandra A. Aggarwal and M. Snir. Hierarchical memory with block transfer. In *Proc. 28th Annual IEEE Symposium on Foundations of Computer Science*, 1987. pp. 204-216.
2. A.K. Chandra A. Aggarwal, B. Alpern and M. Snir. A model for hierarchical memory. In *Proc. 19th Annual ACM Symposium on Theory of Computing*, pages pp. 305 314, 1987.
3. A. Aggarwal and J. Vitter. The i/o complexity of sorting and related problems. In *Comm. of the ACM (CACM)*, 1988. pp. 1116-1127.
4. Ullman Aho, Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
5. Federico Bassetti, Kei Davis, Madhav Marathe, and Dan Quinlan. Loop transformations for performance and message latency hiding in parallel object-oriented frameworks. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, 1998.
6. S. Rao C. Leiserson and S. Toledo. Efficient out-of-core algorithms for linear relaxations using blocking covers. In *Proc. 34th Annual IEEE Symposium on Foundations of Computer Science*, 1993.
7. S. Carr. *Memory Hierarchy Management*. PhD thesis, Rice University, 1992.
8. Kei Davis, Bobby Philip, and Dan Quinlan. Rose: A general tool for the independent optimization of object-oriented frameworks. In *ISCOPE'99*. (submitted).
9. P.J. Denning. Virtual memory. In *ACM Computing Surveys*, 1970. pp. 153-189.
10. P.C. Fischer and R.L. Probert. Storage reorganization techniques for matrix computation in a paging environment. In *Comm. of the ACM (CACM)*, 1979. pp. 405-415.
11. R.W. Floyd. Permuting information in idealized two level storage. In R.E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*. Plenum Press, 1972. pp. 105-109.

12. J. W. Hong and H.T. Kung. I/o complexity: The red blue pebble game. In *Proc. 13th Annual ACM Symposium on Theory of Computing*, 1981. pp. 326-333.
13. D.R. Shultz R.L. Mattson, J. Gacsei and I.L. Traiger. Evaluation techniques for storage hierarchies. In *IBM Systems Journal*, pages pp. 78-117, 1970.
14. M. Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Department of Computer Science, Stanford University, 1992.
15. Micheal Wolfe. *High Performance Compilers for Parallel Computing.* Addison Wesley, 1996.