LA-UR 98-2404

Approved for public release; distribution is unlimited

Title: Optimizing Transformations of Stencil Operatons for Parallel Oject-Oriented Scientific Frameworks on Cache-Based Architectures

CONF-981207--

Author(s): Federico Bassetti
Kei Davis
Dan Quinlan

Submitted to: ISCOPE '98
Santa Fe, New Mexico
December 8–11, 1998

RECEIVED
DEC 21 1998
OSTI

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Los Alamos
National Laboratory

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. The Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

Form 836 (10/96)

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Optimizing Transformations of Stencil Operations for Parallel Object-Oriented Scientific Frameworks on Cache-Based Architectures

Federico Bassetti, Kei Davis, and Dan Quinlan

CIC-19, Los Alamos National Laboratory

Abstract. High-performance scientific computing relies increasingly on high-level large-scale object-oriented software frameworks to manage both algorithmic complexity and the complexities of parallelism: distributed data management, process management, inter-process communication, and load balancing. This encapsulation of data management, together with the prescribed semantics of a typical fundamental component of such object-oriented frameworks—a parallel or serial array-class library—provides an opportunity for increasingly sophisticated compile-time optimization techniques. This paper describes two optimizing transformations suitable for certain classes of numerical algorithms, one for reducing the cost of inter-processor communication, and one for improving cache utilization; demonstrates and analyzes the resulting performance gains; and indicates how these transformations are being automated.

1 Introduction

Current ambitions and future plans for scientific applications, in part stimulated by the Accelerated Scientific Computing Initiative (ASCI), practically mandate the use of higher-level approaches to software development, particularly more powerful organizational and programming tools and paradigms for managing algorithmic complexity, making parallelism largely transparent, and more recently, implementing methods for code optimization that could not be reasonably expected of a conventional compiler.

An increasingly popular approach is the use of C++ object-oriented software *frameworks* or hierarchies of extensible libraries. The use of such frameworks has greatly simplified (in fact, made practicable) the development of complex serial and parallel scientific applications at Los Alamos National Laboratory (LANL) and elsewhere. Examples from LANL include Overture [2], and POOMA [3].

Concerns about performance, particularly relative for FORTRAN 77, are the single greatest impediment to widespread acceptance of such frameworks, and our (and others') ultimate goal is to produce FORTRAN 77 performance (or better, in a sense described later) from the computationally intensive components of such C++ frameworks, namely their underlying *array classes.* There

are three broad areas where potential performance, relative to theoretical machine capabilities, is lost: language implementation issues (which we address for C++ elsewhere [1]), communication, and with the trend toward ever-deeper memory hierarchies and the widening differences in processor and main-memory bandwidth, poor cache utilization.

Experience demonstrates that optimization of array classes themselves is not enough to achieve desired performance, rather, their *use* must also be optimized. One approach, championed by the POOMA project (and others), is the use of *expression templates*. Another, being pursued by ourselves, is the use of an optimizing preprocessor.

This paper presents optimizing transformations applicable to stencil or stencil-like operations, which can impose the dominant computational cost of numerical algorithms for solving PDEs. The first is a parallel optimization which hides communication latency. The second is a serial optimization which greatly improves cache utilization. These optimizations dovetail in that the first is required for the second to be of value in the parallel case. Last is an outline of an ongoing effort to automate these (and other) transformations in the context of parallel object-oriented scientific frameworks.

2 Array Classes

In scientific computing arrays are the fundamental data structure, and as such compilers attempt a large number of optimizations for their manipulation. For the same reason, array class libraries are ubiquitous fundamental components of object-oriented frameworks. Examples include A++/P++ [6] in Overture, *valarray* in the C++ standard library [8], Template Numerical ToolKit (TNT) [7], the GNU Scientific Software Library (GNUSSL) [4], and an unnamed component of POOMA.

The target of transformation is the A++/P++ array class library which provides both serial and parallel array implementations. Transformation (and distribution) of A++/P++ array statements is practicable because, by definition, they have no hidden or implicit loop dependence. Indeed, it is common to design array classes so that optimization is reasonably straightforward—this is clearly stated, for example, for *valarray*. Such statements vectorize well, but our focus is on cache-based architectures because they are increasingly common in both large- and small-scale parallel machines.

An example of an A++/P++ array statement implementing a stencil operation is

```
for (int n=0; n != N; n++)  // Outer iteration
  A(I) = (A(I-1,J) + A(I+1,J) + A(I,J-1) + A(I,J+1)) * 0.25;
```

The statement may represent either a serial or parallel implementation of Jacobi relaxation. In the parallel case the array data represented by A is distributed in some way across multiple processors and communication (to update the ghost boundary points along the edges of the partitioned data) is performed by the "=" operator. The equivalent (serial) C code is

```
for (int n=0; n!=N; n++) {  // Outer iteration
  for (int j=1; j!=SIZE_Y-1; j++)  //calculate new solution
    for (int i=1; i!=SIZE_X-1; i++)
      a_new[j][i] = (a[j][i-1] + a[j][i+1] +
                     a[j-1][i] + a[j+1][i]) * 0.25;
  for (int j=1; j!=SIZE_Y-1; j++)  //copy new to old
    for (int i=1; i!=SIZE_X-1; i++)
      a[j][i] = a_new[j][i];
}
```

3 Reducing Communication Overhead

Tests on a variety of multiprocessor configurations, including networks of workstations, shared memory, DSM, and distributed memory, show that the cost (in time) of passing a message of size N, cache effects aside, is accurately modeled by the function $L + CN$, where L is a constant per-message latency, and C is a cost per word. This suggests that *message aggregation*—lumping several messages into one—can improve performance.[1]

In the context of stencil-like operations, message aggregation may be achieved by widening the ghost cell widths. In detail, if the ghost cell width is increased to three, using `A` and `B` as defined before, `A[0..99,0..52]` resides on the first processor and `A[0..99,48..99]` on the second. To preserve the semantics of the stencil operation the second index on the first processor is 1 to 51 on the first pass, 1 to 50 on the second pass, and 1 to 49 on the third pass, and similarly on the second processor. Following three passes, three columns of `A` on the first processor must be updated from the second, and vice versa. This pattern of access is diagrammed in Figure 1.

Clearly there is a tradeoff of computation for communication overhead. In real-world applications the arrays are often numerous but small, with communication time exceeding computation time, and the constant time L of a message exceeding the linear time CN. Experimental results for a range of problem sizes and number of processors is given in Figures 2 and 3.

Additional gains may be obtained by using asynchronous (non-blocking) message passing, which allows computation to overlap communication. Here the computation involving the ghost boundaries and adjacent columns is performed first, communication initiated, then interior calculations performed. Widening the ghost boundaries and so allowing multiple passes over the arrays without communication decreases the ratio of communication time to computation time; when reduced to one or less communication time is almost entirely hidden.

4 Temporal Locality, Cache Reuse, and Cache Blocking

Temporal locality refers to the closeness in time, measured in the number of intervening memory references, between between a given pair of memory references.

[1] Cache effects are important but are ignored in such simple models.

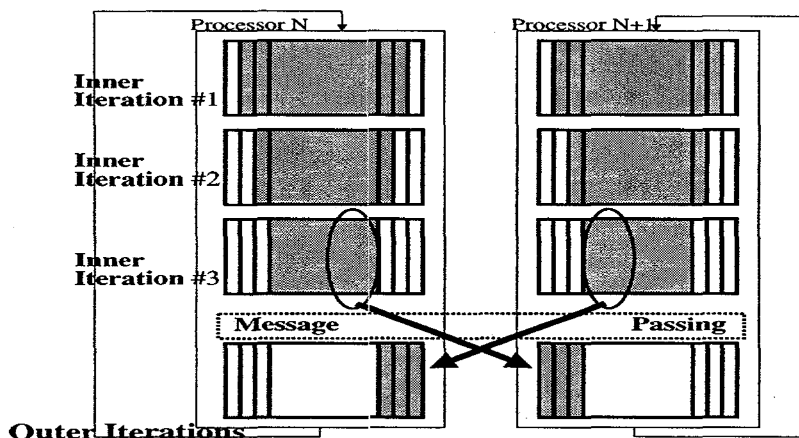


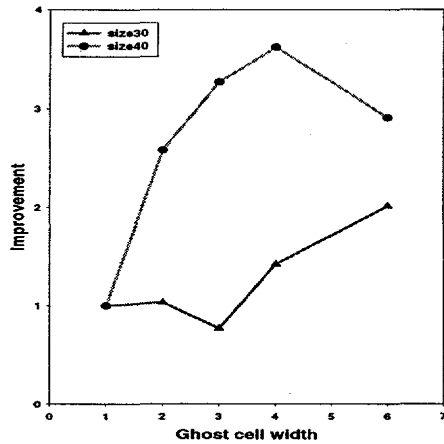
Fig. 1. Pattern of access and message passing for ghost boundary width three.

Of concern is the temporal locality of references to the same memory location—if sufficiently local the second reference will be resolved by accessing cache rather than main or non-local shared memory.

A cache miss is *compulsory* when it results from the first reference to a particular memory location—no ordering of memory references can eliminate a compulsory miss. A *capacity* miss occurs when a subsequent reference is not resolved by the cache, presumably because it has been flushed from cache by intervening memory references. Thus the nominal goal in maximizing cache utilization is to reduce or eliminate capacity misses. We do not address the issue of *conflict* misses: given that a cache is associative and allowing a small percentage of the cache to remain apparently free when performing cache blocking, their impact is has proven unimportant. For architectures where their impact is significant various solutions exist [5].

To give an example of the relative speeds of the various levels of the memory hierarchy, on the Origin 2000—the machine for which we present performance data—the cost of accessing L1 cache is one clock cycle; 10 clock cycles for L2, 80 clock cycles for main memory, and for non-local memory 120 clock cycles plus network and cache-coherency overhead.

A problem with loops that multiply traverse an array (as in the given code fragment) is that when the array is larger than the cache, the data cycles repeately through the cache. This is common in numerical applications, and stencil operations in particular. *Cache blocking* seeks to increase temporal locality by re-ordering references to array elements so that small blocks that fit into cache undergo multiple traversals without intervening references to other parts of the array.



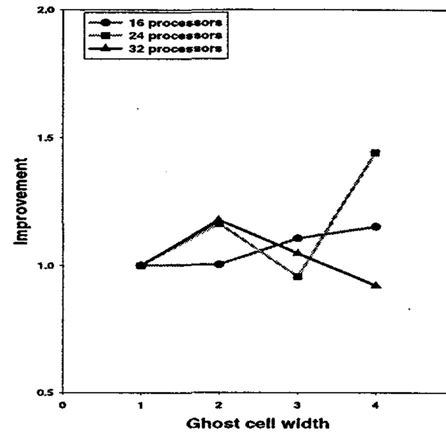


Fig. 2. Message aggregation: improvement as a function of problem size and ghost cell width.

Fig. 3. Message aggregation: improvement as a function of number of processors and ghost cell width.

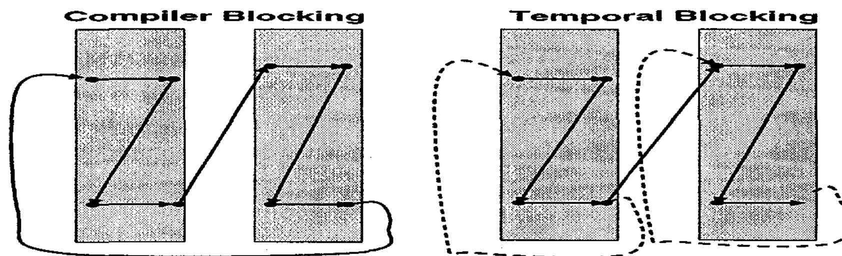


Fig. 4. Pattern of access for compiler blocking versus temporal blocking.

We distinguish two kinds of cache blocking: blocking done by a compiler (also called *tiling*) which we will refer to as *compiler blocking* (Figure 3), and our more effective technique, which we call *temporal blocking.* (Figure 4). In the case of e.g. stencil operations, a compiler won't do the kinds of optimizations we propose because of the dependence between outer iterations. A compiler may still perform blocking, but to a lesser effect. For both, the context in which the transformation may be applied is in sweeping over an array, typically in a simple regular pattern of access visiting each element using a stencil operator. Such operations are a common part of numerical applications, including more sophisticated numerical algorithms (e.g. multigrid methods). What we describe is independent of any particular stencil operator, though the technique becomes more complex for higher-order operators because of increased stencil radius. Temporal blocking is also applicable to a loop over a sequence of statements.

5 The Temporal Blocking Algorithm

The basic idea behind the algorithm is that of applying a stencil operator to an array, *in place*, generalized to multiple applications of the stencil (iterations over the array) in such a way that only one traversal is required in each of one or more dimensions.

Consider first a stencil operator `f(x,y,z)` applied to a 1D array `A[0..N]`. Ignoring treatment at the ends, the body of the loop, for loop index variable `i`, is

```
t = A[i];
A[i] = f( u, A[i], A[i+1] );
u = t;
```

Here `t` and `u` are the temporaries that serve the role of the array of initial data (or the previous iteration's values) for algorithm that does not work in place.

Next we generalize to `n` iterations. For three iterations the code is

```
for (j=2; j!=-1; j--) {
  t[j] = A[i+j];
  A[i+j] = f( u[j], A[i+j], A[i+j+1] );
  u[j] = t[j];
}
```

First we observe that the 'window' into the array—here `A[i..i+3]`, may be as small as the stencil radius plus the number of iterations, as is the case here. Second, at the cost of slightly greater algorithmic complexity, so saving space but with small cost in time, only one temporary array is required, of length one greater than the minimum size of the window, rather than two the minimum size of the window.

It is this window into the array that we wish to be cache-resident. It may be any size greater than the minimum (the temporary storage requirements do not change); for our performance experiments the various arrays are sized so that they nearly fill the L1 cache.

Another observation is that given an n-dimensional problem with an m-dimensional decomposition, this technique may be applied with respect to any subset of the m dimensions of the decomposition—the more common and more simply coded multiple-traversal and/or old-new approximations approach applied to the remaining dimensions. The goal is to make all applications of the stencil operator to any given data element for a single cache miss (the compulsory one) for that element, which indicates that for larger problem size (relative to cache size) the technique must be applied with respect to a larger number of dimensions. Figure 6 depicts the stencil operation and temporary storage for a 1D decomposition of a 2D problem.

6 Performance Analysis

It is possible to predict the number of cache misses generated by the Jacobi relaxation code. In the case of compiler blocking the first sweep through the

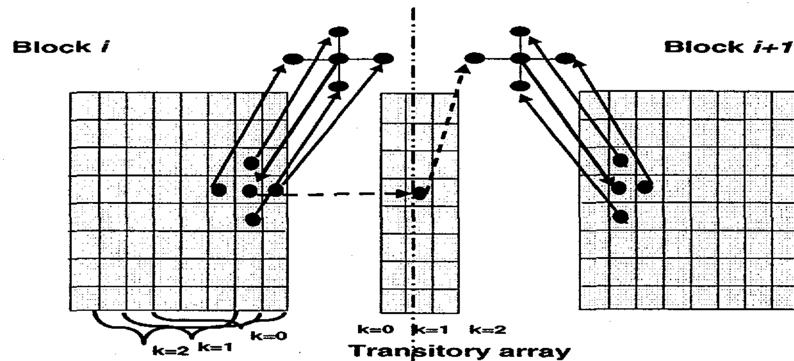


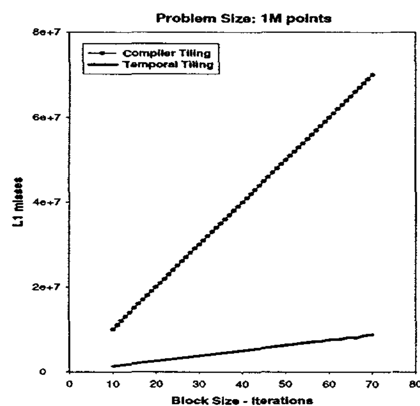
Fig. 5. Stencil operation and temporary storage for 1D decomposition of a 2D problem.

array should generate a number of cache misses equal to the number of elements accessed; these are compulsory. Each subsequent sweep will generate the same number of capacity misses. For temporal blocking only the compulsory misses should be generated. Experimental results shown in Figure 6 bear this out. The data were collected on a MIPS R10000-based Silicon Graphics Inc. Origin 2000 with 32K of primary data cache and 4M of secondary unified cache, using on-processor hardware performance monitor counters; the programs were compiled at optimization level 3 using the MIPSpro C++ compiler. Figures 7 and 8 contrast the performance of compiler blocking and temporal blocking in terms of CPU cycles. The block size as well as the number of Jacobi iterations varies on the x-axis.

Figure 7 shows that the temporal blocking version is twice as fast as the compiler blocking version until the block size exceeds the size of primary cache, beyond which temporal blocking and compiler blocking generate a similar number of cache misses. As expected, temporal blocking yields an improvement in performance linear in the number of iterations so long as the various data associated with a particular block fit in cache. Figure 8 shows that there is an ideal block size (relative to cache size)—it terms of CPU cycles, blocks smaller than ideal suffer from the constant overhead associated with the sweep of a single block; blocks larger than ideal generate capacity misses. (The spikes are attributable to anomalies of the hardware counters that do not capture results absolutely free of errors.)

The presence of multiple cache levels requires no special consideration: other experiments show optimization with respect to L1 cache is all that is required, and optimization with respect to L2 only is not nearly as beneficial.

For problem sizes exceeding the size of L2 (usually the case for meaningful problems), a straightforward implementation gives rise to a number of cache misses proportional to the number of iterations; with our transformation the number of misses is effectively constant for a moderate numbers of iterations.



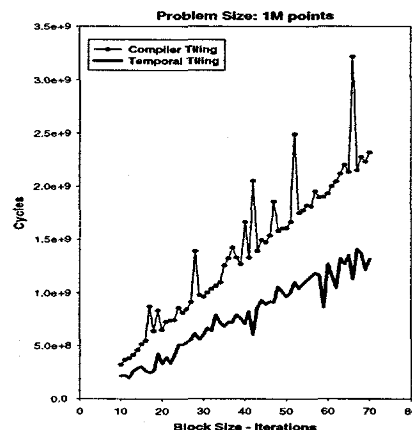


Fig. 6. L1 misses as a function of block size/number of iterations.

Fig. 7. CPU cycles as a function of block size/number of iterations.

In all the experiments for which results are given the block size is the same as the number of iterations. In this implementation of temporal blocking the number of possible iterations is limited by the block size. However, in most cases the number of iterations is dictated by the numerical algorithm. The choice then becomes that of the best block size given a fixed number of iterations. For a Jacobi problem a good rule of thumb is to have the number of elements in the transitory array be a small fraction of the number of elements that could fit in a block that fits in primary cache.

The figures show, and Figure 9 makes clear, that achieved performance improvement is not as good as predicted—performance improves with the number of iterations, but never exceeds a factor of two. The figure shows that the achieved miss behavior is not relatively constant, but depends on the number of interations. The test code currently ignores the fact that permanent residency in cache for the transitory array cannot be guaranteed just by ensuring that there is always enough cache space for all the subsets of the arrays. Different subsets of the other arrays can map to the same locations in cache as does the transitory array, resulting in a significant increase in the number of conflict misses; this is a recognized problem with cache behavior; a solution is suggested in [5]. Various approaches are being evaluated under the assumption that there is still room for improvement before reaching some physical architecture-dependent limitations.

7 Automating the Transformation

An optimizing transformation is generally only of academic interest if it is not deployed and used. In the context of array classes, it does not appear possible to provide this sort of optimization within the library because the applicability of the optimization is context dependent—the library can't know how its objects

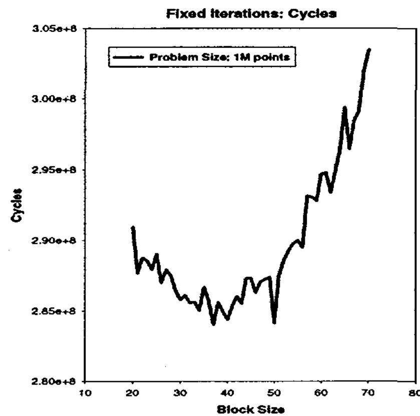


Fig. 8. CPU cycles as a function of block size for fixed number of iterations.

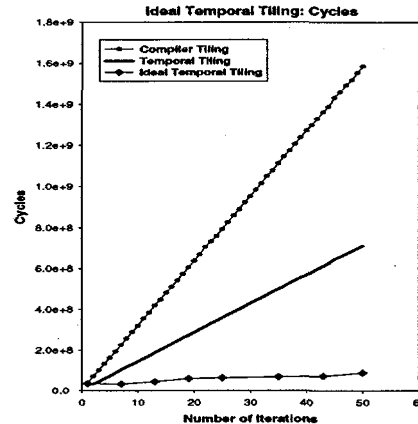


Fig. 9. Relative predicted performance assuming ideal cache behaviour.

are being used. Two mechanisms for automating such optimizations are being actively developed the use of *expression templates* (e.g. in POOMA), which seems too limited; and a source-to-source tranformational system (a pre-processor), which we are currently developing.

The ROSE preprocessor is a mechanism for (C++) source-to-source transformation, specifically targetted at optimizing the use of statements manipulated array class objects. It is based on the Sage II C++ source code restructuring tools and provides a distinct (and optional) step in the compilation process. It recognizes the use of the A++/P++ array class objects, and is 'hard-wired' with (later parameterized by) the A++/P++ array class semantics, so obviating the need for difficult or impossible program analysis. It is also parameterized by platform properties such as cache size. There are in principle no limits (within the bounds of computability) on the types of transformations that can be performed using this mechanism.

8 Conclusions

Previous work has focused on the optimization of the array class libraries themselves, and the use of techniques such as expression templates to provide better performance than the usual overloaded binary operators. We posit that such approaches are inadequate, that desirable optimizations exist that cannot be implemented by such methods, and cannot reasonably be expected to be implemented by a compiler. One such optimization for cache architectures has been detailed and demonstrated.

A significant part of the utility of this transformation is in its use to optimize array class statements (a particularly simple syntax for the user which hides the

parallelism, distribution, and communication issues) and in the delivery of the transformation through the use of a preprocessing mechanism.

The specific transformation we introduce addresses the use of array statements or collections of array statements within loop structures, thus it is really a family of transformations. For simplicity, only the case of a single array in a single loop has been described. Specifically, we evaluate the case of a stencil operation in a `for` loop. We examine the performance using the C++ compiler, but generate only C code in the transformation. We demonstrate that the temporal blocking transform is two times faster than the standard implementation.

The temporal blocking transformation is language independent, although we provide no mechanism to automate the transformation outside of the Overture object-oriented framework. The general approach could equally well be used with FORTRAN 90 array syntax.

Finally, the use of object-oriented frameworks is a powerful tool, but limited in use by the performance being less than that of FORTRAN 77; we expect that work such as this to change this situation, such 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 optimizing preprocessor) that can introduce more sophisticated transformation (because of their more restricted semantics) than compilers could introduce (because of the broader semantics that the complete language represents).

References

1. Federico Bassetti, Kei Davis, and Dan Quinlan. Toward fortran 77 performance from object-oriented scientific frameworks. In *Proceedings of the High Performance Computing Conference (HPC'98)*, 1998.
2. David Brown, Geoff Chesshire, William Henshaw, and Dan Quinlan. Overture: An object-oriented software system for solving partial differential equations in serial and parallel environments. In *Proceedings of the SIAM Parallel Conference*, Minneapolis, MN, March 1997.
3. J.V.W. Reynders et. al. *POOMA: A Framework for Scientific Simulations on Parallel Architectures*, volume Parallel Programming using C++ by Gregory V. Wilson and Paul Lu, chapter 16, pages 553–594. MIT Press, 1996.
4. Gnu scientific software library. http:// KachinaTech.com.
5. Naraig Manjikian and Tarek Abdelrahman. Array data layout for the reduction of cache conflicts. In *????*, 1997.
6. Rebecca Parsons and Dan Quinlan. A++/p++ array classes for architecture independent finite difference computations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference (OONSKI'94)*, April 1994.
7. Roldan Pozo. Template numerical toolkit. http:// math.nist.gov/ tnt/.
8. Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition edition, 1997.