Optimizing Transformations of Stencil Operations for Parallel Cache-based Architectures

Federico Bassetti
Scientific Computing Group, CIC-19
Los Alamos National Laboratory
Los Alamos NM, 87545, USA

Kei Davis
Scientific Computing Group, CIC-19
Los Alamos National Laboratory
Los Alamos NM, 87545, USA

Abstract ***This paper describes a new technique for optimizing serial and parallel stencil- and stencil-like operations for cache-based architectures. This technique takes advantage of the semantic knowledge implicitly in stencil-like computations. The technique is implemented as a source-to-source program transformation; because of its specificity it could not be expected of a conventional compiler.***

Empirical results demonstrate a uniform factor of two speedup. The experiments clearly show the benefits of this technique to be a consequence, as intended, of the reduction in cache misses. The test codes are based on a 5-point stencil obtained by the discretization of the Poisson equation and applied to a two-dimensional uniform grid using the Jacobi method as an iterative solver. Results are presented for a 1-D and 2-D tiling for a single processor. For the parallel case both blocking and non-blocking communication have been tested. However, the parallel case is not discussed here.

Keywords: parallel stencil operations, optimization, cache-based architectures.

1 Introduction

All general purpose computer systems, from the ubiquitous PC to the largest supercomputers, rely on a multilevel memory hierarchy to achieve a reasonable fraction of the theoretical performance of the CPU core. Excluding the bottom of the hierarchy (disk storage and backing store) this typically comprises a register file, one or more levels of cache, and a nominal main memory; on the largest of such machines, e.g. the DOE ASCI Blue distributed shared memory multiprocessor, main memory is further divided into local—on-box, faster, smaller–and remote–off-box, slower, larger. It is clear to those that create the large-scale scientific (numerically intensive, array-oriented) applications for such large parallel machines that careful data management with respect to inter-processor communication and the memory hierarchy is critical to achieving acceptable or even useful—literally more than a few percent of peak theoretical—performance.

In the context of large-scale parallel scientific applications, research and experience has shown that the greatest performance gains are realized by directing the most effort toward efficient utilization of the uppermost levels of the memory hierarchy, namely the register file and the L1 (primary) cache, and data distribution/interprocessor communication. Benchmarking has shown that even more than is the case for more mainstream application areas, for numerical codes the bulk of the computation is represented by a very small fraction of the code.

2 Stencil-like Operations

For several families of numerical algorithms the dominant computational kernels are stencil and stencil-like operations, made conspicuous and amenable to parallelization and optimizing transformation by their repetitive and regular access of (typically) very large arrays.

Numerical algorithms used for solving partial differential equations—multigrid methods and those entailed by linear solvers—are rich with stencil-like operations to effect smoothing and relaxation. The distinguishing characteristic of stencil-like operations is the evaluation of a computationally inexpensive expression, for each element of a (possibly multi-dimensional) array, in which the arguments are all of the elements within a given radius of that element.

In this paper Jacobi relaxation is used as a canonical example, using a two-dimensional array and a five-point stencil (diagonal elements are excluded). 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];
```

where `A` is dimensioned `[0..`n_i`,0..`n_j`]`. Generally several sweeps are made, with `A` and `B` swapping roles to avoid copying. In a parallel environment the arrays are typically distributed across multiple processors.

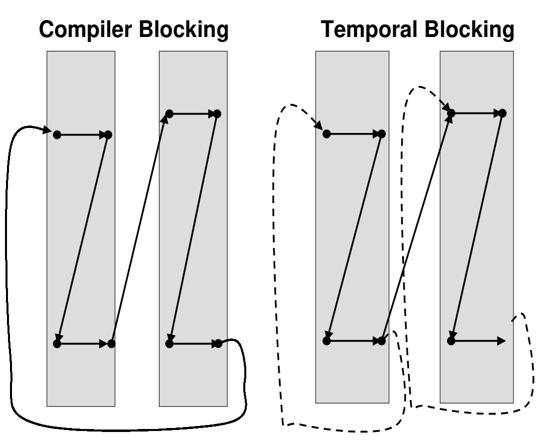

Figure 1: Exploiting Temporal Locality

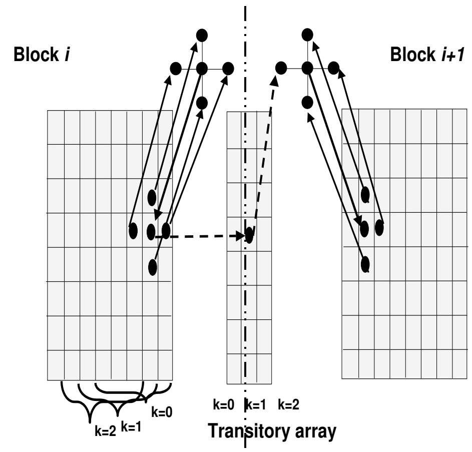

Figure 2: Stencil operation and temporary storage for 1D decomposition of a 2D problem.

3 Efficiency

Naively implemented, stencil operations are severely memory-bound, hence very inefficient, even in the presence of a cache. In essence the problem is that the arrays, being many times the size of the cache, cycle through the cache with minimal (only the overlapping of the stencil arguments) spatial or temporal reuse. Hardware 'prefetching' of cache lines or blocks, and compiler-inserted prefetch instructions on superscalar architectures, is of little help [1].

Optimizing compilers attempt to improve performance by *tiling*—reordering array access such that one or more blocks remain cache resident for several sweeps. Again, in this context improvement is measurable but minimal [1].

Our target machine is the DOE ASCI Blue SGI Origin 2000 distributed shared memory multiprocessor, for which the 32 KB L1 cache has access time of one clock cycle; the 4 MB L2 cache 10 clock cycles, and the local main memory 80 clock cycles. Analysis and experiments show that for perfectly uniform sequential single-array access the amortized cost is approximately 6 cycles per word; for simple Jacobi relaxation approximately 10 cycles. This latter is entirely the cost of memory access: on this superscalar architecture the stencil computation is effectively free, the arithmetic is performed in (instruction-level) parallel with memory accesses. In principle average access time could be close to one cycle.

Elsewhere we have shown how processor time can be traded for communication time [2] for net performance gain, and shown simple sequential optimizations (also applicable in the parallel case), to give significant improvement in performance. Others have also proposed various optimizations in this context [5, 3, 4]. This paper describes a new technique which we call *sliding block temporal tiling*, and give performance figures for a number of architectures: a network of workstations, single workstation, and the ASCI Blue Origin 2000.

4 Sliding Block

Following is a sketch of the algorithm for a 2D tiling for a 2D Jacobi 5-point stencil. The terms are defined as follows. *Mixed_Relaxation* performs the relaxation on points that are on the frontier of adjacent blocks. In order to compute correct results the needed values were previously stored. Thus, one of the four points comes from a different array (Figure 1). Each dimension of the block has an associated storage array. *Jacobi_Relaxation* performs the Jacobi 5-point stencil relaxation using only the interior points of a block. *Store* stores the points on the frontier of a block and its adjacent blocks. *Slide* modifies the position of the block so that the points on the frontier of the current block can overlap with points on the frontier of adjacent blocks.

```
INPUT
     Problem Size: X_size, Y_size
     Total Iteration: T_iter
     Block Iteration: B_iter
     Block Size: X_block, Y_block

REPEAT T_iter / B_iter times
  Move along X dimension in X_block increments
    Move along Y dimension in Y_block increments
      REPEAT B_iter times
        Mixed_Relaxation along X_edge
        Mixed_Relaxation along Y_edge
        Jacobi_Relaxation on
          current block (X_block, Y_block)
        Store Y_edge
        Store X_edge
        Update solution on
          current block (X_block, Y_block)
        Slide current block
```

5 Performance

In this section we present the performance improvement achieved when our technique is compared agaist a naive implementation relying on an optimizing compiler's capabilities. The performance data were gathered on SGI Origin 2000 system. The configuration of the system used includes 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, in line size, between L1 and L2 cache is 1:4 (one L2 line contains four L1 lines). The system has 32 Gbytes total memory, or 250 Mbytes per processor. The test codes were compiled using the MIPSpro C++ compiler at the O3 optimization level. The performance data were gathered via the hardware performance monitors available on MIPS R10000 processors.

5.1 Single Processor

Figure 1 pictorially describes the idea introduced with the temporal blocking optimization. The Jacobi iterative solver with a 5-point stencil is the code adopted to quantify the benefits of temporal blocking.

5.1.1 Ideal behavior

Every code will achieve its best performance if the whole working set fits in primary cache, thus we can take in-cache performance as targeted performance. Figures 3, 4, and 5 show the performance of the ideal case for a Jacobi solver whose working set fits in cache. Three metrics are used: *cycles* in Figure 3 to describe the overall execution; *primary cache data misses* in Figure 4 to describe the impact of memory penalty, and *flops* in Figure 5 to describe how efficiently (if compared to peak)

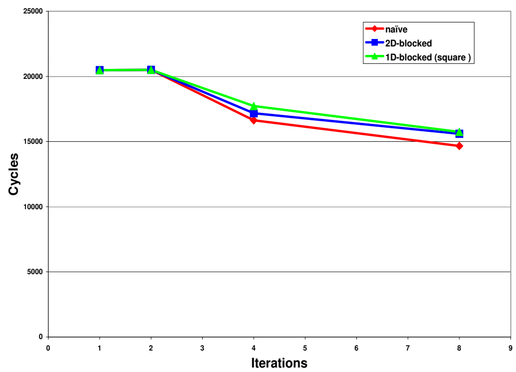

Figure 3: Average Cycles per iteration (in-cache problem size)

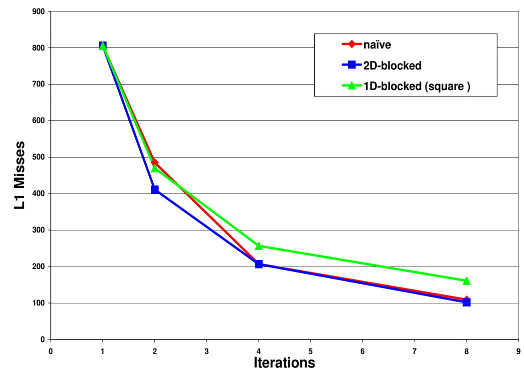

Figure 4: Average L1 Misses per iteration (in-cache problem size)

the test code performs. All the figures present results based on the average cost of one iteration. Thus, the figures show how the cost average cost varies in different scenarios. Every figure presented in this paper is for a fixed problem size, varying the number of iterations. The cost of each iteration is calculated using the total value of a particular metric divided by the number of iterations performed. Figure 3 shows that the cost of each iteration generally decreases as the number of iterations increases. This is easily explained: Figure 4 shows the number of primary cache misses. Clearly, all the misses involved are first-time-load misses; once a datum is loaded into cache it remains resident for the duration of the computation. Increasing the number of iterations simply amortizes the cost first-time-load misses. The efficiency of the test code is illustrated in Figure 5 using the flops metric, confirming the results of the first two figures.

5.1.2 Practical reality

The vast majority of scientific applications carrying out meaningful computations have a working set that is by far larger than cache (both primary and secondary level). Figure 6 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; Figures 7, and 8 confirm 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 9.

5.1.3 Exploiting Temporal Locality

Most scientific applications inherently have locality; this is a simple consequence of array-based computation. Modern compilers are able to detect and exploit some locality. However, as depicted in Figure 1, compilers exploit locality only within one iteration; reuse is minimal or nil across each iteration. Exploiting temporal locality with the technique presented in the previous sections demonstrably reduces the difference in performancefrom the ideal cache-resident version. Figure 6, 7, 8, and 9 give performance results for optimized version of the testcode 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

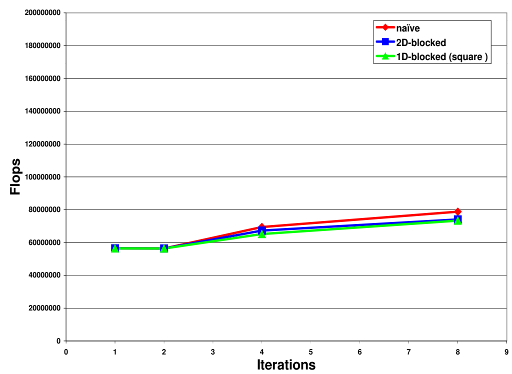

Figure 5: Average Flops per iteration (in-cache problem size)

show that the *2D-blocked* version performs better than the others. Quantitatively, it is possible to observe a *2x* factor improvement compared to the naive implentation. The explanation for such differences in performance is in the number of misses, as show in Figures 7 and 8. 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 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 indeed a stripe that cannot fit in L1 or L2 cache. However, a stripe offers more reuse potential for L2 cache, as can be seen from Figure 8.

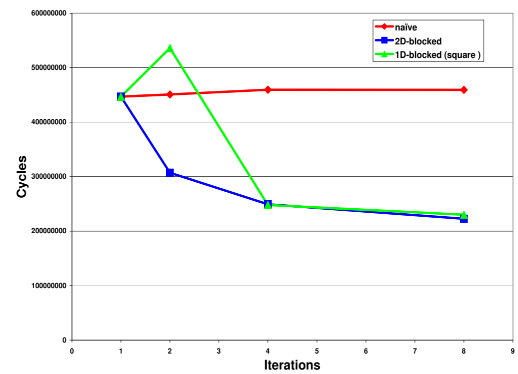

Figure 6: Average Cycles per iteration (out-of-cache problem size)

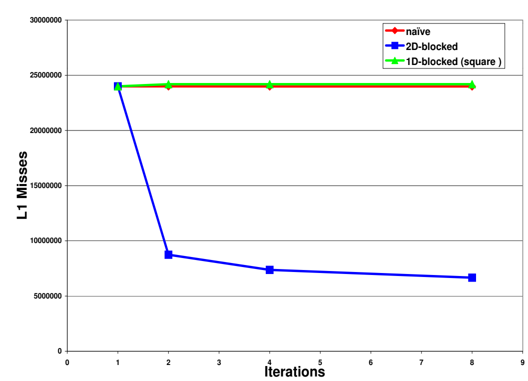

Figure 7: Average L1 Misses per iteration (out-of-cache problem size)

From the peformance figures presented it is clear that the transformation is in fact an op-

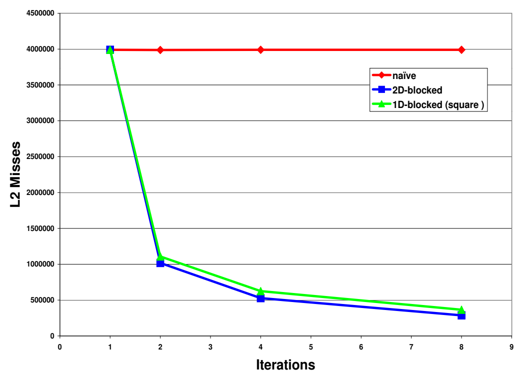

Figure 8: Average L2 Misses per iteration (out-of-cache problem size)

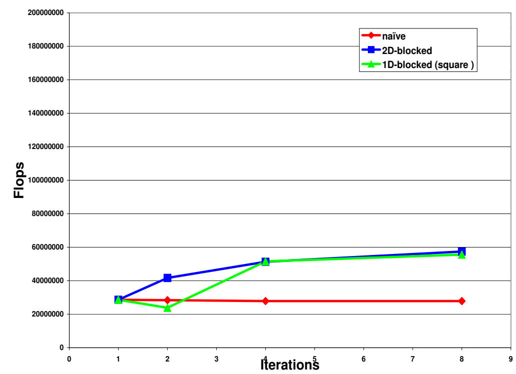

Figure 9: Average Flops per iteration (out-of-cache problem size)

timization. However, a validation on different platforms is needed to better complete the picture. 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 the pipelining process a little harder. This effect might be the limiting factor for the temporal blocking technique. The maximum achievable improvement factor is still under investigation.

5.2 Multiple Processors

The exploitaton of temporal locality is equally achievable in the context of multiple processors, with an additional 'layer' of transformation required. Figure 10 shows how computation is traded for communication, so that multiple iterations are possible on each individual processor before requiring communication. This transformation *enables* the temporal blocking optimization on each processor. The results, not included in this preliminary draft, demonstrate the same factors of improvement achieved by the single processor version.

6 Automating the Transformations

An optimizing transformation is generally only of academic interest if it is not deployed and used. 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, which seems too limited; and a source-to-source tranformational 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. 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 array class objects, and is 'hard-wired' with (later parameterized by) the particular array class semantics, so obviating the need for difficult or impossible program analysis. It is also parameterized by platform properties such as cache sizes and depths. There is in principle no limit (within the bounds of computability) on the types of transformations that can be performed using this mechanism.

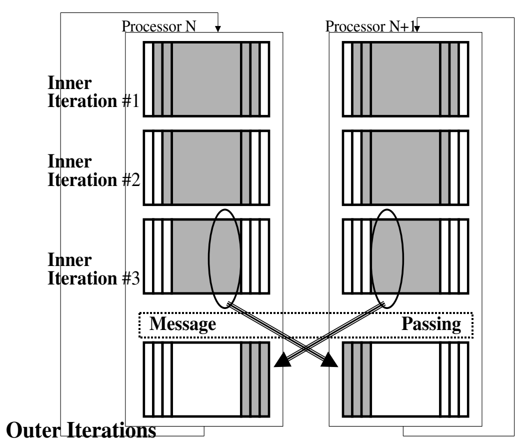

Figure 10: Reducing Communication

7 Future Work

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. Also of interest is the performance impact when using different compilers: though not discussed here, register allocation has a significant impact on both overall performance, and optimized relative to unoptimized performance.

The parallel aspect of this technique is a subject of ongoing investigation. Currently only explicit message passing using the MPI library has been evaluated; in future other paradigms such as threads and OpenMP pragmas will be investigated.

References

[1] 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.

[2] Federico Bassetti, Kei Davis, and Dan Quinlan. Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures. In D. Caromel et al., editor, *International Scientific Computing in Object-Oriented Parallel Environments, ISCOPE 98*, volume 1505 of *LNCS*. Springer, 1998.

[3] Craig Douglas, Johathan Hu, Ulrich Reude, and Marco Bittencourt. Cache based multigrid on unstructured two dimensional grids, 1998.

[4] Ulrich Reude. Iterative algorithms on high performance architectures. In *EUROPAR'97*, LNCS. Springer Verlag, 1997.

[5] Christian Weiss, Markus Kowarschik, Ulrich Ruede, and Wolfgang Karl. Cache-aware multigrid methods for solving poisson's equation in two dimensions, 1999.