Improving Scalability with Loop Transformations and Message Aggregation in Parallel Object-Oriented Frameworks for Scientific Computing

Federico Bassetti, Kei Davis, and Dan Quinlan
Scientific Computing Group CIC-19
Computing, Information, and Communications Division
Los Alamos NM, 87545, USA
Tel: (505) 667-7492, Fax: (505) 667-1126
{fede,kei,dquinlan}@lanl.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 are 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.

Achieving scalability, we believe, requires both algorithm design *and* compile-time support. The optimizations we present are automatable because our stencil-like computations are implemented at a high level of abstraction using object-oriented parallel array class libraries. These optimizations, which are beyond the capabilities of today compilers, may be performed automatically by a preprocessor such as the one we are currently developing.

1 Introduction

Current initiatives in design and deployment of parallel supercomputers seek performance of one teraflop (ASCI Blue Mountain, ASCI Red, ASCI Blue Pacific). While the current machines, as designed, are in principle capable of significant performance, realized performance may be only a few percent of maximum theoretically sustainable performance. Achieving this maximum performance presumes the absence of a number of performance impediments, primarily: access of memory other than L1 cache; and secondly, inter-processor communication. Today we are have *scalable machines*, such as the SGI Origin 2000. Unfortunately, machine scalability does not induce scalability of the applications it may run. Any realistic parallel application will require the use of main memory and inter-processor communication. The route to scalability is the minimization of the primary performance impediments. Related, as will be shown, is the development of software to simplify both the development of large practical applications and simultaneously simplify the optimization of such applications.

Our investigations are driven by the need to realize good performance from large-scale parallel object-oriented numerical frameworks such as OVERTURE[1] [1], the performance of which is heav-

[1]OVERTURE is available from http://www.c3.lanl.gov/cic19/teams/napc/napc.shtml

ily impacted by that of the underlying parallel object-oriented array class library A++/P++[2] [2]. It has become clear that optimization of A++/P++ itself is insufficient; its *use* must also be optimized. This paper focusses on stencil-like computations because of their importance in numerical computation and their corresponding impact on performance on our parallel applications.

The optimization techniques described are language- and library-independent, but we will argue that in the context of array-class libraries such as A++/P++ such optimization may be *automated*, much as a compiler performs lower-level optimizations. Much current work is devoted to this automation process, though this is not the subject of this paper.

2 Parallel Stencil Operations

The numerical algorithms used for solving partial differential equations (PDEs) are rich with stencil-based computational kernels. In most cases these impose the dominant computational cost of a numerical application. In the solution of PDEs both second- and fourth-order methods are commonly used with stencil widths of three and five, respectively; higher order methods are also used with correspondingly greater stencil widths. In numerical applications, and especially those with complex geometry, the distinguishing characteristic of stencil-like operations is the evaluation of expressions in which the operands are all of the elements within a given 'radius' of a given element of a *single* array. In our applications scaling of the array operands (coefficents) is required because of the geometry of the grids on which the computation is performed.

In this paper Jacobi relaxation is used as a canonical example. Such computations appear as parts of more sophisticated algorithms such as 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*B[i-1][j] + w2*B[i+1][j] + w3*B[i][j-1] + w4*B[i][j+1]
```

where `A` and `B` are dimensioned `[0..I,0..J]`. Typically several passes are made, with `A` and `B` swapping roles to avoid copying.

In a parallel environment the arrays are typically distributed across multiple processors; we take as a concrete case `I=J=100`, with both arrays distributed along one dimension over two processors, conceptually `A[0..99,0..49]` on one, `A[0..99,50..99]` on the second, and similarly for `B`. In practice 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 processor's segment of the array. In this case, with stencil radius one, one processor stores `A[0..99,0..50]`, the other `A[0..99,49..99]`, and similarly for `B`, `j` ranges from 1 to 49 on the first processor and 50 to 98 on the second. Thus a single pass of the stencil operation over each half may be performed without any communication. After each pass inter-processor communication is required: `A[0..99,49]` on the first processor is copied to `A[0..99,49]` on the second processor, and `A[0..99,50]` on the second processor is copies to `A[0..99,50]` on the first. This generalizes easily to more dimensions, more processors, and division of the arrays along more than one axis. It should be clear that in the parallel environment the communication phase at the end of each iteration over array data is required for correctness and that the adjacent processor's data must be received before the subsequent iteration.

The equivalent A++/P++ array statement is simply:

```
A(I,J) = W1*B(I-1,J) + W2*B(I+1,J) + W3*B(I,J-1) + W4*B(I,J+1);
```

[2] A++/P++ is available from http://www.c3.lanl.gov/cic19/teams/napc/napc.shtml

In this form the array statement uses the P++ parallel array class library to encapsulate the parallelism and hide the message passing. While the transformations we detail are general, it is the array class library that we target for such optimizations.

3 Reducing Communication Overhead

Tests on a variety of multiprocessor configurations show that the cost (in time) of passing a message of size N 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* can improve performance.

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 sweep, 1 to 50 on the second sweep, and 1 to 49 on the third sweep, and similarly on the second processor. Following three sweeps, 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.

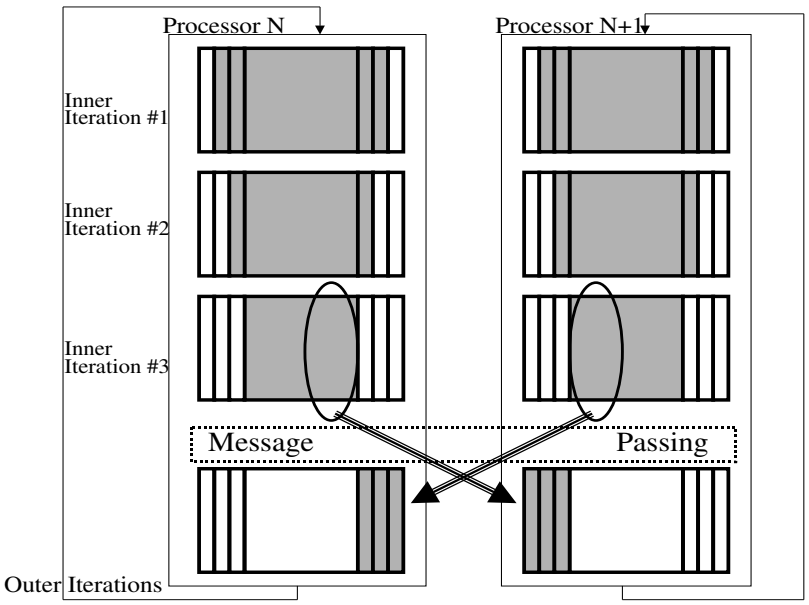

Figure 1: Pattern of access and message passing for ghost boundary width three. The communication it is performed only every three iterations. Larger boundary are exchanged. The third iteration uses the same elements as had the ghost boundary been width one..

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 Figure 2.
Additional gains may be obtained by using asynchronous (non-blocking) message passing, which allows computation to overlap communication.

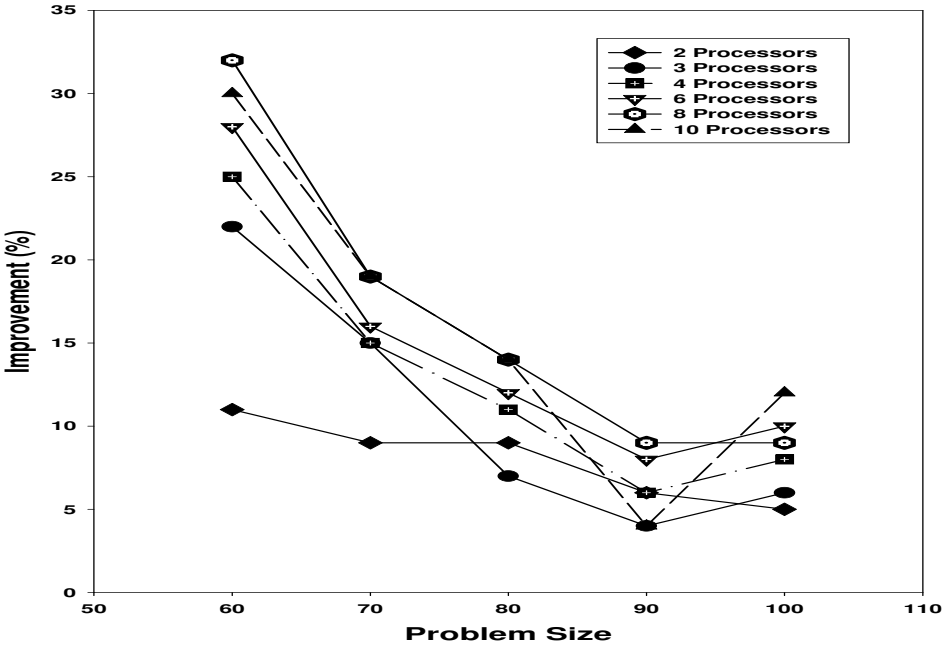

Figure 2: Data collected on a 16 node SGI Origin 2000, showing the percentage of improvement obtained using message aggregation. Problem sizes are given in number of elements on a side of a square array. The improvement is the ratio between the time taken with message aggregation over the time taken without without message aggregation. Only data for the best aggregation factor are shown; improvement is achieved by almost all the factors but the best in all the cases is achieved using an aggregation factor between 3 and 6. The communication used in the testcode is implemented using synchronous message passing.

Widening the ghost boundaries and so allowing multiple sweeps over the arrays without communication decreases the ratio of communication time to computation time. In particular, this technique increases the scalability of the code. Since the amount of communication is reduced, the amount of computation needed to balance the communication can be smaller and so the work can divided among more processors.

4 Loop Transformations for Exploiting Cache

Single array computations often require space greater than cache size, even when distributed over multiple processors (at some point increasingly distributing data increases communication cost beyond the gains of increased processing power and available caches). For stencil-like computations in which entire arrays are swept over repeatedly, if the array or working set of arrays exceeds the size of cache a thrashing effect occurs.

We first present this transformation as a simple loop exchange that many compilers currently perform, wherein the code fragment

```
for (int n=0; n != N ; n++)
   for (int i=0; i != I; i++)
      for (int j=0; j != J; j++)
         A[i][j] = B[i][j];
```

becomes

```
for (int i=0; i != I; i += BLOCK_I)
   for (int j=0; j != J; j += BLOCK_J)
      for (int n=0; n != N; n++)
         for (int x=0; x != BLOCK_I; x++)
            for (int y=0; y != BLOCK_J; y++)
               A[i+x][j+y] = B[i+x][j+y];
```

In other words, compilers can fragment repeated operations on large segments of arrays into smaller segments that are cache-resident over several iterations. Such a transformation relies critically on the absence of data-dependence, which does not hold for stencil computations.

Increasing ghost cell width to allow multiple passes without communication or copying enables *blocking*—breaking up the array on a single processor such that each piece fits in cache. Multiple passes are then made on entirely cache-resident blocks, so that the cache misses associated with the initial load of the array(s) are amortized over all iterations. In the parallel case this benefit is in addition to the benefits of message aggregation. Experimental results are shown in Table 1.

Code	size	Iterations	Cycles	Loads	Stores	H1	H2	Hm
B	1000	10	110000000	50237700	19932500	0.989	0.834	0.165897
NB	1000	10	580000000	49830300	19920300	0.856	0.751	0.248975
B	1000	100	892817000	502319000	199269000	0.996	0.947	0.052415
NB	1000	100	5858870000	498293000	199201000	0.856	0.750	0.249505
B	200	10	3573410	1979660	784816	0.988	0.997	0.002194
NB	200	10	5268710	1961460	784305	0.855	0.999	0.000143
B	200	100	34312800	19784000	7843790	0.995	0.999	0.000482
NB	200	100	52409200	19605000	7841300	0.855	0.999	0.000008
B	50	10	235242	118309	46381	0.989	0.999	0.000566
NB	50	10	264745	116247	46305	0.927	0.999	0.000085
B	50	100	2234740	1173820	461461	0.995	0.999	0.000433
NB	50	100	2563220	1153960	461295	0.930	0.999	0.000026

Table 1: Data was collected on a MIPS R10000 with a 2MB L2 cache. The particular problem sizes are representative of problems that, respectively, fit in L1, fit in L2, and are larger than L2. The number of iterations is varied so that the amount of reuse is varied. Cycles, loads, and stores were measured using hardware counters. H1, H2, and Hm represent the hit ratios for each level of memory present on this architecture. L1 and L2 misses were also measured using hardware counters. Here B denotes blocking, NB non-blocking.

5 Automating Optimization

In the context of parallel object-oriented array class libraries we are developing an automated solution, the *optimizing preprocessor* ROSE[3], which takes as input and produces as output C++ code. This approach is practically possible because the preprocessor is hardwired with (and in time will be parameterized by) information about the array class library semantics; this information could not be reasonably determined or acted upon by a compiler. Importantly, because the preprocessor is semantics-preserving, its use is optional.

[3]ROSE++ Web Site: http://www.c3.lanl.gov/ROSE/

6 Conclusions

For stencil-like codes we have demonstrated optimizations that significantly reduce the two greatest sources of inefficiency in parallel environments: message-passing latency and poor cache utilization.

The justification for the use of object-oriented frameworks has traditionally been to allow faster development of more portable applications, accepting the loss of performance relative to carefully crafted and tuned lower-level, but machine-specific, code. We have argued that in addition to the recognized benefits, use of object-oriented numerical frameworks makes practicable, in the form of a preprocessor, automatic optimization that could not be performed by a compiler. The high level of abstraction at which complex large scale applications, such as Overture, are built has provided us with enough knowledge to design a class of optimizations that are more specific and performance effective than for a lower-level paradigm (C- or Fortran-like). As a consequence, these larger codes are *more scalable* than their predecessors.

References

[1] 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.

[2] Dan Quinlan and Rebecca Parsons. A++/p++ array classes for architecture independent finite difference computations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, April 1994.