Optimizations for Parallel Object-Oriented Frameworks

Fede Bassetti* Kei Davis* Daniel Quinlan†

Abstract

Application codes increasingly fail to realize the potential performance of modern parallel architectures; compilers have only limited mechanisms with which to effect sophisticated transformations to arrest this trend. Compilers must must perform optimizations in the context of the semantics of a complete language specification and thus cannot perform more sophisticated transformations. Object-oriented frameworks provide a level of tailoring of the C++ language to specific restricted contexts, but compilers are still relied upon for most performance optimization. No mechanism exists to express the restricted semantics of a class library such that a compiler could perform more sophisticated optimizations.

In this paper we explore a family of optimizing transformations appropriate to object-oriented frameworks for scientific computing, and present a preprocessing mechanism, ROSE, which delivers the more sophisticated transformations automatically from the use of abstractions represented within high level object-oriented frameworks. We have found that these optimizations permit improved performance over FORTRAN 77 by factors of three to four, sufficiently interesting to suggest that higher level abstractions can contain greater semantics and that the greater semantics can be used to drive more sophisticated optimizations than are possible with lower level languages.

1 Introduction

Application codes increasingly fail to realize the potential performance of modern parallel architectures; compilers have only limited mechanisms with which to effect sophisticated transformations to arrest this trend. Compilers must must perform optimizations in the context of the semantics of a complete language specification and thus cannot perform more sophisticated transformations. Object-oriented frameworks provide a level of tailoring of the C++ language to specific restricted contexts, but compilers are still relied upon for most performance optimization. No mechanism exists to express the restricted semantics of a class library such that a compiler could perform more sophisticated optimizations.

In this paper we explore a family of optimizing transformations appropriate to object-oriented frameworks for scientific computing, and present a preprocessing mechanism, ROSE, which delivers the more sophisticated transformations automatically from the use of abstractions represented within high level object-oriented frameworks. We have found that these optimizations permit improved performance over FORTRAN 77 by factors of three to four, sufficiently interesting to suggest that higher level abstractions can contain greater semantics and that the greater semantics can be used to drive more sophisticated optimizations than are possible with lower level languages.

*Scientific Computing Group (CIC-19), Los Alamos National Laboratory, Los Alamos, NM.

†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA.

In our current work we apply this technology to the A++/P++ array class, a serial/parallel array class for scientific computing. A++/P++ forms the basis for the OVERTURE framework, an object-oriented framework for complex geometries and adaptive mesh refinement. The mechanisms we will discuss are central to portable performance of object-oriented frameworks on varying architectures (cache based architectures in particular), and are not specific to a particular framework.

Initial results have demonstrated that we can use the ROSE preprocessing technique to optimize the performance of array class code. ROSE is completely programmable and in principle any transformation may be introduced. The research issue then becomes one of defining transformations that are appropriate. Separate work on types of transformations has demonstrated performance 3.5 times that of semantically equivalent FORTRAN 77 code (on the SGI Origin 2000). The ROSE preprocessing mechanisms clearly permits the introduction of much more general transformations than expression template mechanisms, provides greater portability, and does not entail the extremely long compile times associated with expression templates. This paper presents our current results and evaluations of performance compared to alternative mechanisms such as C or FORTRAN 77.

2 A++/P++ Array Class Library

A++ and P++ [6] are array class libraries for performing array operations in C++ in serial and parallel environments, respectively. The use of P++ is the principle mechanism by which the OVERTURE framework operates in parallel.

A++ is a *serial* array class library similar to FORTRAN 90 in syntax, but not requiring any modification to the C++ compiler or language. A++ provides an object-oriented array abstraction well suited to large scale numerical computation. It provides efficient use of multidimensional array objects which serve to both simplify the development of numerical software and provide a basis for the development of parallel array abstractions. P++ is the *parallel* array class library and shares an identical interface to A++, effectively allowing A++ serial applications to be recompiled using P++ and run in parallel. This provides a simple and transparent mechanism for allowing serial codes to be run in parallel.

P++ provides a data parallel implementation of the array syntax represented by the A++ array class library. To this extent it shares a lot of commonality with FORTRAN 90 array syntax and the HPF programming model. However, in contrast to HPF, P++ provides a more general mechanism for the distribution of arrays and the greater control required for the multiple grid applications represented by both the Overlapping Grid model and the Adaptive Mesh Refinement (AMR) model. Current work is addresses the addition of task parallelism for parallel adaptive mesh refinement.

Here is a simple example code segment that solves Poisson's equation with the Jacobi method in either a serial or parallel environment using the A++/P++ classes. Note that the Jacobi iteration for the entire array is written in one statement.

```
  // Solve u_xx + u_yy = f by a Jacobi Iteration
Range R(0,n)                            // ... define a range of indices: 0,1,2,...,n
floatArray u(R,R), f(R,R)               // ... declare two two-dimensional arrays
f = 1.; u = 0.; h = 1./n;               // ... initialize arrays and parameters
Range I(1,n-1), J(1,n-1);               // ... define ranges for the interior

for( int iteration=0; iteration<100; iteration++ )
  u(I,J) = .25*(u(I+1,J)+u(I-1,J)+u(I,J+1)+u(I,J-1)-f(I,J)*(h*h)); // ... data parallel
```

The use of highly parallel computers is so pervasive in the development and use of high performance scientific software that some mechanism is required to encapsulate parallelism at some level in a general enough way to permit more productive application development. It seems clear that higher level abstractions would encapsulate parallelism at some lower level, it is a question as to how low a level it is appropriate to encapsulate parallelism for scientific computing. The use of array operations within an object-oriented framework for serial and parallel operations is natural and desirable and forms one of the lowest levels where parallelism can be encapsulated so that higher level abstractions can be built. At the same time it is one of the highest levels where parallelism can be encapsulated because lower levels of abstraction are not tied as directly to the large volumes of data present within arrays which typically form the basis of scientific computing (at least for finite difference and finite volume based applications). Also, with the development of array languages in the 1970's and the introduction of array abstractions in FORTRAN 90 most scientists are aware of the array abstraction already.

3 Alternative Program Models

The use of array objects as an abstraction within scientific computation, while being relatively well understood (because of FORTRAN 90 array abstractions), is not commonly seen as hiding an alternative more traditional and simpler programming model, the *local patch* model: the more explicit programming model of using simple local patches of contiguous data on a single processor to express an algorithm. The array abstraction is a higher level abstraction representing a collection of such local patch data, where it exists, on the collection of processors making up the entire machine. Of course within this model the communication is handled by the user but the development of the abstraction permits most such communication to be handled simply using a single function call (instead of a rather tedious number of MPI function calls).

The use of a parallel array abstraction does not preclude the use of the local data directly and in fact encourages this since it can be done only where it is perceived to be important instead of globally within a application code as would be required otherwise. The use of the lower level local patch model also permits the leveraging of existing serial and parallel application codes (older FORTRAN and C serial or parallel kernels or applications). The performance of using the *local patch* abstraction is of course that of the FORTRAN or C it is written in, with absolutely none of the overhead introduced by the array abstraction. Thus both abstractions live comfortably together and each may be used where appropriate.

The following example demonstrates these two programming models on a simple application code fragment:

- The Array Abstraction

```
doubleArray X(100);
doubleArray Y(100);
X = 1.0;
Y = 1.0;
Range I (1,98);
X(I) = ( Y(I+1) + Y(I-1) ) * 0.5;
```

  In this code fragment using the array abstraction, notice that in the serial and parallel environments the source is identical. The communication is handled internally, with no more messages generated than in the explicit message passing programming model. In fact, since the message passing is hidden there is greater opportunity for optimizations specific to message latency hiding, the effect is to provide an automated level of optimization beyond that which would be provided in an explicit message passing model and which is architecture dependent, resulting in greater performance (something we will demonstrate later in this paper). Note that we seek to provide the user some refuge from the explicit message passing programming model to provide increased productivity, while at the same time making it accessible as required (because people write software and people are have different tastes, styles, biases, and requirements).

- The *Local Patch* Abstraction

```
doubleArray X(100);
doubleArray Y(100);
double* X_ptr = X.getLocalDataPointer();
double* Y_ptr = Y.getLocalDataPointer();
int i;
for (i=0; i < X.localLength(0); i++)
   {
    *X_ptr[i] = 1.0;
    *Y_ptr[i] = 1.0;
   }
for (i=0; i < X.localLength(0)-2; i++)
   {
     *X_ptr[i] = (Y_ptr[i+1] + Y_ptr[i-1]) * 0.5;
   }
X.updateGhostBoundaries();
```

Example code using the *local patch* programming model. Such code clearly operates at the efficiency of C, since it is C code. The communication at the ghost boundary (where the local data is partitioned across processors) is updated using simple function call, since the communication schedule is known more sophisticate optimizations are possible and which may well be architecture specific.

The two examples are semantically the same, but have different implementations to reflect the two different programming models. It should be clear that A++/P++ and the presence of the higher level array abstraction does not preclude the use of alternative (what we might consider more primitive) programming methods. *It might be felt that this later* **local patch** *programming model is the right way to proceed if performance is to be preserved, but we will show just how throughly in error this assumption really is.* Because the later example nails down the implementation so specifically as to provide no significant room for optimization and it is the optimization (an architecture dependent issue) that is the key to good performance.

4 Relaxation as a Case Study

The case of local relaxation methods form an important part of elliptic solution methods particularly for multigrid methods. This work on optimization of relaxation methods is part of more general work on optimizations specific to multigrid for cache based architectures. Such solution techniques form important parts of more sophisticated application codes and a dominate the performance in many cases. The use of techniques to improve the performance of relaxation techniques and multigrid more specifically can be expected to play an important role in improving performance of such applications more generally.

The perceived truth about performance is that FORTRAN 77 compilers define it for any given algorithm. That for a particular code fragment such as we have shown, the FORTRAN 77 implementation and execution defines the upper bound of the performance. Where it is believed that this may be in some doubt it is perceived to be true within some marginal percentage. That the FORTRAN 77 compiler could define, within narrow limits, the performance potential is however quite in error. The reasons have to do with the potential of optimizations which a FORTRAN 77 compiler could not do because of the the minimal semantics that can be understood from the implementation.

5 Temporal Blocking as an Optimization

These optimizations are described in greater detail in [2]. They 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), while cache is treated as faster storage (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.* In this case we introduce optimizations based upon these graph coverings. The τ-neighborhood covers are treated much like compiler blocking in the trivial case of $\tau = 1$, but provide greater temporal locality for values of $\tau > 1$.

The general idea behind applying this transformation to structured grid computations is to cover (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. Thus let 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 and the experimental results bear bear this out. 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, in the design of modern algorithms. 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.

As demonstrated, temporal blocking can give a factor of two improvement in performance over compiler blocking. What has not yet been shown is the dependency of performance improvement on the number of iterations for a fixed block size. The goal has been to

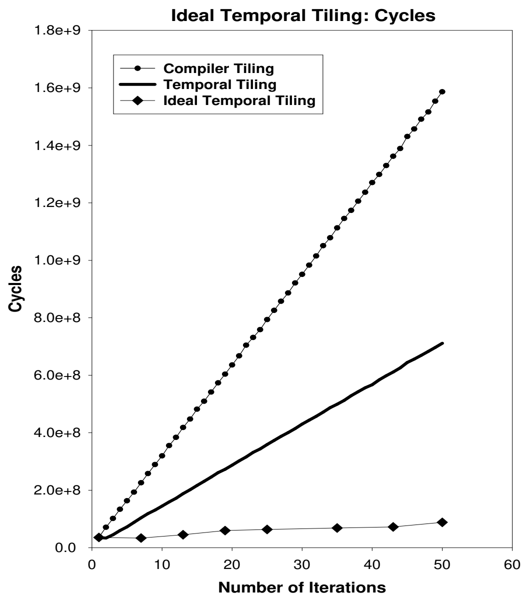

FIG. 1. *Better Performance of temporal blocking (cycles = time).*

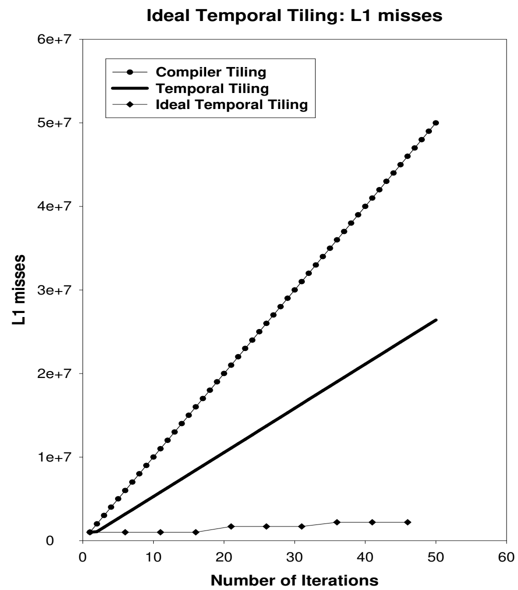

FIG. 2. *Better performance through fewer L1 misses.*

reduce the number of capacity misses to a negligible value. Figures 1 and 2 show the result of varying the number of iterations for a fixed block size for a problem that doesn't fit in secondary cache. The figures show that achieved performance improvement is not as good as predicted—performance improves with the number of iterations, but never exceeds a factor of two. At this stage the ideal performance is a crude estimate; in future this estimate will be improved. Figure 2 makes clear that the improvement factor is determined by the reduction in the number of misses. The figure shows that the achieved miss behavior is not relatively constant, but depends on the number of iterations. The problem appears to be an artifact of the test code, which assumes that the transitory array it is always resident in cache, which is ideally correct. The assumption is based on an implementation that ensures that there is always enough space in cache for the transitory array and a subset of the other arrays. 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 would map to the same locations as does the transitory array, resulting in a significant increase in the number of conflict misses. Various solutions are being evaluated under the assumption that there is still room for improvement before reaching some physical architecture-dependent limitations.

We note that this optimization can't be introduced at the local patch level of manipulation of the data.

6 Message Latency Hiding as an Optimization

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*

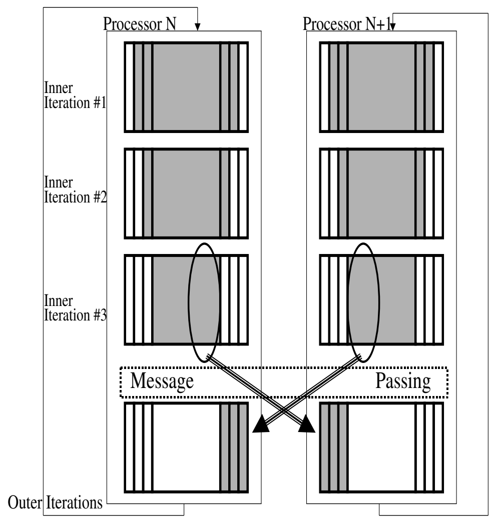

FIG. 3. *Pattern of access and message passing for ghost boundary width three.*

aggregation—lumping several messages into one—can improve performance.[1] Note that this sort of message aggregation does not require the packing and unpacking of multiple messages.

In the context of stencil-like operations, message aggregation may be achieved by widening the ghost cell widths. As a specific example, 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 3.

Clearly there is a tradeoff of computation for communication overhead. In real-world applications the arrays are often numerous and thus not as large as memory permits. Distributed across numerous processors the arrays become relatively small, with communication time exceeding computation time, and the constant time L of a message nearly matching or exceeding the linear time CN. Experimental results for a range of problem sizes and number of processors is given in Figure 4.

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.

This is the principal mechanism for hiding message latency, but it also enables the more sophisticated cache based transformations (presented in the previous section) in the parallel

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

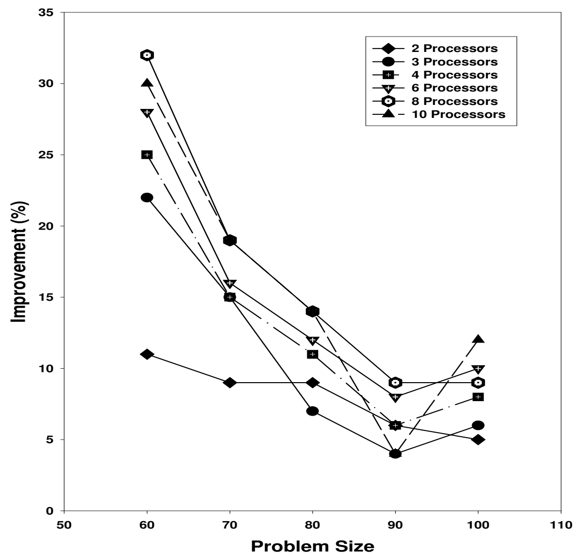

FIG. 4. *Performance improvement of message aggregation as a function of problem size and number of processors (synchronous message passing) (results are mostly independent of ghost boundary widths (values of 2-6 are used)).*

environment.

Again, we note that this optimization can't be introduced at the local patch level of manipulation of the data.

7 The Key to Superior Performance

For better or worse, the existence of many cache based optimizations require semantic knowledge (or program analysis) largely unavailable at existing FORTRAN 77 or C level languages.

The key to obtaining better performance is more sophisticated optimizations. The key to having more sophisticated optimizations is the ability to recognize their applicability and a mechanism for introducing them. The key to recognizing applicability is to have semantic knowledge of higher-level abstractions. Because gleaning the semantics of the intended algorithm from its implementation in a low level language is too difficult, we suggest that if greater semantics is to be captured, the program should be written using higher level abstractions and that these abstractions be designed purposefully given specific optimizations. However, having the abstractions is not sufficient, some preprocessing mechanism with knowledge of the abstractions' semantics *must* be added to automate the optimizations. Many simple optimizations could be placed into the implementation of the abstractions themselves but these are not an important target of our work. The more relevant abstractions are the ones which interact and have possible optimizations given their context within loops and their interaction with one another, such as array abstractions in scientific applications.

It is not clear how far this approach toward high performance can be taken. It is clear that this approach yields some dramatic examples. The development of improved performance for relaxation techniques is just one example.

8 Conclusion

In our alternative program models it was percieved that the *local patch* model was the better performance oriented solution but it is clear that the specific nature of its implementation does not provide sufficient room for either the message passing latency hiding optimization or the temporal blocking optimization and thus restricts its overall performance. Its implementation literally ties it to a specific style of execution and prevents the compiler from providing substantial optimizations (as if the compile could anyway). It is not the case yet that the ROSE preprocessor is finished or distributed so the final results are not certain. For the moment we can only form a guess what approach might be ultimately most productive.

Ultimately a programmer must express the semantics of an algorithm as source code in one or more programming languages. When this is done in FORTRAN 77 or C the semantics of the algorithm in the programmer's mind is lost and only the semantics of the language is saved for the compiler to see. From what the compiler sees, and from the programmer's view point, the original algorithm's semantics cannot in general be reconstructed. The most fundamental step where information was lost that could have saved significant semantic information is the translation of the algorithm into software in the first place (the writing of the computer code). If at this level the translation had been to a language containing higher level semantics less of the algorithm's semantic might have been lost and more of it used to generate optimizations. Unfortunately, we are not funded to design new programming languages, only to provide higher level abstractions to simplify the application development. It is a bonus, yet to be well capitalized upon, at this stage of the process to consider the optimizations that the higher level semantics enable. To date, the possible optimizations for possible high level abstractions are not a part of the object-oriented design, but it seems that they should be. We have demonstrated some of the potential gain from this approach.

The algorithm's translation into higher level abstractions might be sufficient to preserve the intended semantics of the algorithm if the optimizations could be introduced at this level. Such would require a mechanism to recognize the higher level abstractions and generate automated optimizations. The mechanism we propose for this is a preprocessor—the responsibility of the preprocessor is to recognize the higher level abstractions and introduce optimizations based upon the context of their use within the application code. The preprocessor must know the semantics of the abstractions in order to introduce the optimizations automatically. In the case of an object-oriented framework, this makes the preprocessing mechanism a component of the framework. The design of a high performance object-oriented framework is then guided by the design of objects with the intent of being optimized by existing optimizations. A good design of an object-oriented framework thus weighs the value of an abstraction and the semantics it represents against the semantics that is to be recovered by the preprocessor and is intended to help guide the optimization. The existing constraints to the object-oriented design have not had to include the possibility of optimizations beyond that which could be expected (or hoped for) by the compiler (very low level indeed and of marginal value in many cases for scientific applications).

In modern architectures access to cache memory is typically many times faster than to main memory (factors of 100 are common); the tradeoff (given constant dollar cost) is speed for size. The potential of these and other cache based transformations is directly dependent upon the relative cost of cache versus main memory. Clearly, no cache is large enough to make a difference *and* this relative cost is increasing quickly with faster CPUs and memory systems that fail to keep pace. The greater hope of phenomenal processing speeds in the future is largely held hostage by the relatively poor bandwidth to main memory (where the data is located and which scientific applications require) unless significant cache based transformations are automated. Our reward for tolerating this is ever deeper memory hierarchies. The current poor state of the art in compiler technology to introduce these significant cache based transformations only hides this from us. The common perception that FORTRAN 77 performance is the best one can do is an example of this.

9 Software Availability

The OVERTURE framework and documentation is available for public distribution at the Web site `http://www.llnl.gov/casc/Overture`. A++/P++ dates back to its first version in 1990 and has been publicly distributed since 1994; the current version was released in 1996. The OVERTURE libraries have been under development since 1994, and have been available to the public since 1996. The AMR++ classes for adaptive mesh refinement in OVERTURE are still under development and are expected to be released in the fourth quarter of 1998.

References

[1] D.L. Brown, Geoffrey S. Chesshire, William D. Henshaw and Daniel J. Quinlan, *OVERTURE: An Object Oriented Software System for Solving Partial Differential Equations in Serial and Parallel Environments*, Proceedings of the Eight SIAM Conference on Parallel Processing for Scientific Computing, 1997.

[2] F. Bassetti, K. Davis, and D. Quinlan, *Loop Transformations for Performance and Message Latency Hiding in Parallel Object-Oriented Frameworks*, 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98). July 13–16, 1998, Las Vegas, Nevada.

[3] , D.L. Brown, William D. Henshaw and Daniel J. Quinlan, *OVERTURE: An Object Oriented Framework for Solving Partial Differential Equations*, Scientific Computing in Object-Oriented Parallel Environments, Springer Lecture Notes in Computer Science, 1343, 1997.

[4] G. Chesshire and W.D. Henshaw, *Composite overlapping meshes for the solution of partial differential equations*, J. Comp. Phys., 90 (1990), pp. 1–64.

[5] Satish Balay, William Gropp, Lois Curfman McInnes and Barry Smith, *The Portable Extensible Toolkit for Scientific Computation*, `http://www.mcs.anl.gov/petsc/petsc.html`.

[6] ——, *A++/P++ manual*, LANL Unclassified Report 95-3273, Los Alamos National Laboratory, 1995.