Preliminary Experience in Developing a Parallel Thin-Layer Navier Stokes Code and Implications for Parallel Language Design

Daryl Olander
and
Robert B. Schnabel

CU-CS-582-92 February 1992

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

To appear in *Proceedings of 1992 Scalable High Performance Computing Conference*, IEEE Press, April 1992.

Preliminary Experience in Developing a Parallel Thin-Layer Navier Stokes Code and Implications for Parallel Language Design[1]

Daryl Olander
and
Robert B. Schnabel

CU-CS-582-92 February 1992

Department of Computer Science
Campus Box 430
University of Colorado at Boulder
Boulder, Colorado 80309-0430 USA
olander@cs.colorado.edu
bobby@cs.colorado.edu

[1]Research supported by Boeing Computer Services and NSF grants CDA-8922510 and ASC-9015577.

Abstract

This paper describes preliminary experience in developing a parallel version of a reasonably large, multi-grid based computational fluid dynamics code, and implementing this version on a distributed memory multiprocessor. Creating an efficient parallel code has involved interesting decisions and tradeoffs in the mapping of the key data structures to the processors. It also has involved significant reordering of computations in computational kernels, including the use of pipelining, to achieve good efficiency. We discuss these issues and our computational experiences with different alternatives, and briefly discuss the implications of these experiences upon the design of effective languages for distributed parallel computation.

1 Introduction

We are currently engaged in a project to create a parallel version of a significant computational fluid dynamics code, and to program the parallel algorithm on a distributed memory multiprocessor using the DINO [7] language. This project is being undertaken in conjunction with Boeing Computer Services to help assess the feasibility of scalable parallel computation for computational fluid dynamics calculations. The code that was selected is an extensively used, 5000 line Fortran code, TLNS3D [9]. It is based upon a multigrid method, and is likely to be representative of even larger codes in this field.

The two key computational kernels in this code, Smooth and Filter, are typical of computational kernels in many grid based numerical methods. The first seems difficult to parallelize efficiently on almost any large scale parallel machine, and appears to require pipelining to obtain good performance. The second appears to present significant challenges if it is to run efficiently on any machine where interprocess communication is even moderately slow. Parallelizing these kernels, and the remainder of the code, effectively, requires the parallelization of many low-level loops that run over multi-dimensional data structures. This turns out to imply that the parallel language or compiler must transform almost every line of the sequential code. It also implies that, as with most scientific codes, the mapping of the multi-dimensional data structures used in the code to the processors is one of the key issues that the parallel algorithm development must address.

The next section of this paper presents a little more background on the TLNS3D code and on the DINO parallel programming language, and discusses the basic challenges in efficiently parallelizing the code. Sections 3 and 4 describe the research issues and work we have done concerning the two main kernels of the code, Smooth and Filter. We discuss several alternatives for implementing each kernel on a distributed memory multiprocessor, and present our results of the implementations of these alternatives in DINO on an Intel iPSC/2 hypercube. We also discuss the implications of this experience for the design of distributed parallel languages, such as Fortran D [3] and the successor to DINO, that are currently under development. Section 5 briefly summarizes the current status and future directions of this research.

2 Background and Basic Challenges

The code being converted into a parallel code is the TLNS3D Navier-Stokes code developed by V. Vatsa and B. Wedan at NASA Langley Research Center [9]. This code is used to simulate the viscous transonic flow around a three dimensional airfoil. It is based upon a regular, three dimensional discretization of the airfoil and surrounding space, and uses a multigrid method with explicit Runge-Kutta time stepping to solve the Navier-Stokes equations. The key computational kernels within the multigrid method are the Smooth and Filter routines discussed in Sections 3 and 4 below. The code initially was developed for a Cray Y-MP, and consists of approximately 5,000 lines of Fortran. A typical run on the Y-MP requires more than an hour of CPU time.

A multigrid technique is used to increase the rate of convergence of the algorithm. It involves alternating between solving the equations on a fine grid and on several coarser grids. Each coarser grid contains half the number of points in each spatial dimension as the previous grid. Since the problem is three dimensional, this reduces the number of points by eight at each multigrid level. In the problems typically run at Boeing Computer Services, the finest grid is size 192 x 64 x 48, with each grid point containing 5 floating point numbers. At least two coarser grids are used as well, so that the smallest grid is no larger than 32 x 16 x 12. An iteration of the multigrid method consists of a "W-cycle" that alternatives between these grids, and the entire algorithm typically requires about 100 iterations to converge. (For a basic introduction to multigrid methods including this terminology, see e.g. [1].) The code contains several data structures of size 192 x 64 x 48 x 5 and numerous additional data structures; the total amount of data storage required for a typical run is about 200 megabytes.

The main goals in designing a parallel version of this code are to obtain as high parallel efficiency as possible, and to allow for scalability for machines with greater numbers of processors. One of the key design issues is the mapping of the main data structures to the parallel machine. These data structures can be broken up along one, two, or all three spatial dimensions. As is discussed in Sections 3 and 4 below, this choice affects the amount of parallelism, the efficiency of communication, the scalability of the parallel algorithm, and the ease of parallel programming, sometimes in opposing manners. In addition, different partitionings may be optimal for different portions of the code, which requires investigation of whether remapping of the data within the code is cost-effective.

The multigrid structure also has a considerable effect on the basic design decisions in the parallel implementation. On the coarser grids, the number of grid points is reduced, as is the length of boundaries between processors, but the number of stages in the algorithm is unchanged. This implies that the amount of computation and the amount of data that is communicated between processors are reduced proportionally, but that the number of communications between processors remains the same. Thus there is less computation to mask the latency of the communications. In addition, load balancing problems arise if the number of grid points along an axis becomes smaller than the number of processors mapped to that axis. This makes it especially desirable to partition the data structures along two or three spatial dimensions rather than just one. These types of considerations play roles in Sections 3 and 4.

We have chosen the DINO programming language for this project because implementing this code using vendor-supplied low level primitives seemed prohibitive, because DINO is one of the few higher level languages available for programming distributed memory machines and one that we have experience with, and because DINO appears well suited to this problem. The main difficulties with using vendor-supplied support is that all communications would have to be explicitly specified using detailed, low level primitives, and that there is no global address space. The latter point would require the programmer to change indices throughout the the code when writing code for each processor. For both reasons, using vendor supplied primitives would be tedious and error-prone.

DINO is an parallel extension of C that is oriented towards expressing regular, data parallel algorithms considerably more simply than using vendor-supplied primitives. Its main parallel features are the ability to declare a virtual parallel machine consisting of a single or multiple dimensional array of processors, the ability to map global data structures to this virtual parallel machine, and the ability to write code that will be executed by each virtual processor using its portion of the data. DINO code uses a global address space in referring to distributed data structures, meaning that no conversion from the sequential view of the data is required. Communication is indicated using a special operator attached to the data item, meaning that the programmer must still explicitly think about communication but that the mechanics of specifying it are far simpler than with low-level primitives. The version of DINO that we have used is described in [7]. This version is considered a first step towards a broadly expressive and easy to use parallel language; recent research that builds upon this version is described in [6].

DINO uses an explicitly parallel, "single-program multiple-data" model of computation, as opposed to deriving parallelism from sequential code with parallel annotations as is done in languages such as Fortran D [3]. Each of these language models has advantages and disadvantages; some comments about their appropriateness to the main kernels of the TLNS3D code are made at the ends of Sections 3 and 4. A drawback of using DINO for this project was that the Fortran code first needed to be converted to C. This was done using an automatic Fortran to C converter. Since a majority of the loops in the code needed to be restructured in any case to optimize communication, we found that this translation had only a small impact upon the size of the project.

3 "Smooth" Computational Kernel

3.1 Basic Algorithm

The smoothing routine is one of the most computationally intensive portions of the code. It performs groups of tridiagonal solves in three sweeps, one sweep for each of the spatial directions (i, j, k) of the main data structures. Each sweep consists of two loops, the first in the forward and the second in the backward direction. For example, the first loop of the sweep in the i direction has the basic form

```
DO J = 1, NJ
  DO K = 1, NK
    DO I = 1, NI-1
      A(I+1,J,K) = A(I+1,J,K) - C*A(I,J,K)
```

while in the second loop of this sweep the I loop runs backwards and calculates A(I-1,J,K) using A(I,J,K). The sweeps in the j and k directions are analogous, with J and K being the innermost loop variables and the variables for which there is a data dependence in the loop, respectively. Thus for each sweep, the computations in two grid directions (the two outermost loop indices) are entirely independent and can be performed concurrently, while the computations in the third direction are inherently sequential due to the data dependence carried by the loop.

3.2 Alternative Parallel Implementations

The main challenge in creating an efficient parallel variant of the smoothing routine comes from the fact that, for each sweep, the algorithm is trivially parallel along two spatial directions but inherently sequential along the third. Furthermore, each of the three spatial directions is the sequential direction for one of the three sweeps. Thus, there is no fixed mapping of the data structure to the processors that trivially parallelizes the entire computation. Instead, there appear to be several alternatives that seem worthy of consideration for mapping the data structure to the processors. In addition, several schemes can be used to perform the computation once the mapping is set. The main alternatives appear to be :

1. *Transpose* – The three dimensional data structure can be partitioned among one (or two) dimensions, in a way that the dimensions of the first two (or one) sweeps are not partitioned. This means that these sweeps require no communication and can be performed with perfect parallelism. Then the data structures can be transposed so that dimensions that remain to be swept are no longer partitioned and can now be swept with perfect parallelism. This involves communicating the entire data structure between processors. Most likely, the entire data structure must be transposed back at the end of the smoothing algorithm so that it is in the same alignment at the end of the algorithm as at the start. (If this is not done then there will have to be multiple versions of all other sections of the code.) Thus, this option requires transposing the entire data structure at least twice for each execution of the smoothing routine.

2. *Static 1D, 2D, or 3D Block Mapping* – The data structures can be statically partitioned in one, two, or all three spatial dimensions, meaning that the same mapping of data to processors is used throughout the computation. This has the advantage that the whole data structure never is communicated, as it is in the transpose version. It has two disadvantages, both of which arise when the sweeps of the algorithm cross processor boundaries. First, data values must be communicated between processors at these points. Second, the data dependencies at these points impose constraints upon the order of computation that may cause the computation to become (partially) sequential. The second disadvantage may be overcome in part by using pipelining. We now elaborate upon these comments by discussing, for simplicity, the case when the data structure is partitioned in one dimension. Then we briefly generalize this discussion to 2D and 3D mappings. The idea of pipelining to parallelize algorithms with data dependencies has been used in many parallel algorithms, for example the solution of triangular systems of linear equations [5], and we assume that the reader has some familiarity with this technique.

In the case when the data structure is partitioned in the i direction only, the sweeps in the j and k directions parallelize perfectly, but the most simple-minded implementation of the sweep in the i direction is entirely sequential. That is, the first processor would do all its computations, for all values of J and K and the first NI/p values of I, where p is the number of processors, then communicate the border values to the second processor which would do the same (using the second NI/p values of I), etc. Such an algorithm would at best have the same speed as a sequential algorithm in the i direction and perfect speedup in the j and k directions, meaning that the overall speedup would be less than three regardless of the number of processors available.

In a pipelined version, the first processor instead does its computations for one (or several) values of J and K (going through the first NI/p values of I for this J and K), then communicates the resulting border value(s) of $A(NI/p, J, K)$ to the second processor which starts its computations for this value(s) of J and K while the first processor moves on to the next value(s) of J and K. In this manner, after a start-up period equal to $p-1$ stages, all processors are busy and remain busy until an ending period with $p-1$ stages. The advantage of this approach that it introduces substantial parallelism without increasing the amount of computation or the amount of data that is communicated in comparison to a non-pipelined version. A possible disadvantage is that data is communicated more often in smaller chunks, which incurs an extra cost on any machine where message latency is substantial.

Analogous considerations arise when the data structure is partitioned along two or three of its spatial dimensions. In the 2D case, two of the three sweeps cross processor boundaries, although each of these sweeps only crosses $O(p^{1/2})$ boundaries and involves $O(p^{1/2})$ portions that may trivially be made concurrent. In the 3D case all three sweeps cross $O(p^{1/3})$ processor boundaries, and involve $O(p^{2/3})$ portions that may trivially be made concurrent. If pipelining is not used, then the overall speedups for the smoothing algorithm are bounded by $O(p^{1/2})$ and $O(p^{2/3})$, respectively. Pipelining can be used to increase the speedup within each each of the portions of the algorithm that crosses processor boundaries, using the same techniques as discussed above. As the number of dimensions that the data structure is partitioned along increases, the overall parallelism of the pipelined algorithms increases, although their complexity and the difficulty in programming them increases as well.

There are several other important considerations in designing a pipelined algorithm for this problem that we mention only briefly. First, for any mapping of data structures to processors, the pipelined algorithm allows a choice between the amount of computation done in each pipeline stage (width) and the maximum available amount of parallelism (depth). For example, if the sweep is in the i direction and the data is partitioned in this direction only, then each pipeline stage can calculate a given processor's i values for either one, or some larger number, of (j,k) pairs before communicating to the next processor. The total amount of computation and data communicated is unaffected, but the smaller the width, the more, smaller messages are sent between processors. With larger widths, fewer pipeline stages are created. Thus this choice involves a tradeoff between communication latency costs and computation costs; as latency increases relative to the amount of computation in the innermost loop, the optimal width increases. Second, the multigrid structure introduces additional complexity into the decision of how to partition the data, because the algorithm must be efficient both for large data structures (fine grids) and for small data structures (coarse grids). For example, a one dimensional partition may be a good option for large data structures but not for small data structures since the number of grid points in any single dimension may be smaller than the number of processors. Assuming that a consistent partition is desired throughout the code so that data may be transferred between grid stages without requiring extensive communication, this consideration may influence the choice of data mapping.

3.3 Computational Comparisons

We have implemented and tested a number of the alternative parallel versions of the smoothing algorithm described above. All tests described in this paper were performed

Table 1: Smoothing Test Results

1D Mapping			2D Mapping		
Processors	Time	Speedup	Processors	Time	Speedup
4	16.6127	3.378	2x2	21.8505	2.569
8	9.0548	6.198	4x2	12.0785	4.647
16	5.3846	10.423	4x4	6.9038	8.129
32	3.6960	15.185	8x4	4.1932	13.38

Problem sizes: 96x32x24x5
Times are in seconds

on an Intel iPSC2 hypercube with a maximum of 32 processors, and were coded in DINO. While the Boeing Cray implementation uses a 192 x 64 x 48 finest grid, with 5 floating point numbers at each grid point, the largest grid used in our experiments is 96 x 32 x 24 (x 5) because the full code using the larger grid does not fit on our machine. (The code contains several data structures of the maximum grid size.)

The transpose-based version of the smoothing algorithm was tried in some simple computational experiments, and found to be less efficient than implementations based upon static mappings and pipelining. For example, on a two dimensional problem with a 192 x 64 data structure mapped onto 32 processors, the cost of two transposes (the minimum required per iteration) alone was about the same as the cost of an entire pipelined algorithm based upon a one dimensional static partition of the data to processors. It seems clear that a transpose-based algorithm would have a greater disadvantage on larger problems on the same machine, because the ratio of the total amount of data that must be communicated in a transpose-based algorithm to the total amount of data that is communicated in a statically mapped algorithm grows as the problem size grows relative to the number of processors. For example, in the above example this ratio is about 3; for a 192 x 64 x 48 grid on 32 processors the ratio is about 6 for a 1D mapping of the data to the processors and 12 for a 2D or 3D mapping. In addition, the static mapping based algorithm with pipelining can mask communications latency with computation for realistic problem sizes, while the transpose-based algorithms can not. While it is possible that different results would be obtained using transpose on different parallel machines or through different implementations of transpose, transpose does not appear likely to be the preferable option for realistic size problems.

Various versions of the pipelined algorithm with different 1D and 2D static mappings of data to processors also were tried. (In the 1D mapping only the i dimension is partitioned, in the 2D mapping the i and j dimensions are partitioned.) Some of the results, on a problem of size 96 x 32 x 24 x 5, are summarized in Table 1. The results given are for the near-optimal width/depth choices for these problem sizes and mappings. These turned out to be to aggregate the computations for all the indices along k (and the fourth) axis but none along the i or j axes.

The results in Table 1 illustrate that the inherent sequential nature of the smoothing algorithm is only partially overcome by the use of pipelining. The efficiency with 32 processors is about 50%. The advantage of the 1D mapping over the 2D mapping is slightly surprising, since the 1D algorithm has about twice as much data to communicate as the 2D algorithm. The advantage probably is due to data contiguity effects : all messages in the 1D algorithm consist of contiguous data, while this is not the case in the 2D algorithm, and message passing with contiguous data is faster in most message passing systems. We have decided to use the 2D mapping in our overall implementation, however, because a 1D mapping would result in poor load balancing and even idle processors for coarser grids. We did not implement the 3D static mapping with pipelining because it is more difficult to program, and because these results indicated that it little if any benefit would result.

3.4 Implications for Parallel Language Design

The type of computation performed in the smoothing algorithm is common in grid based methods, and illustrates the need for parallel languages to either be able to express pipelined computations explicitly, or to generate them automatically. How this is done will depend upon the type of parallel language that is used. It appears (see [8]) that parallel languages for distributed scientific computations are falling into two broad classes, those like Fortran D [3] or Kali [4] where the programmer provides a complete, sequential program augmented by annotations such as data distributions and parallel loops, and explicitly parallel languages like Dino [7], or Fortran or C augmented by vendor provided primitives, where the programmer writes parallel code using a single-program multiple-data (SPMD) paradigm. The issues in expressing pipelined computations are different in these two models.

It is fairly easy to express pipelined computation in an explicitly parallel language, like Dino or Fortran/C with low-level primitives, where the user also designates communication points. The programmer only needs to write the sequential code that will be performed on a generic processor, a minor modification of the sequential code, and indicate the places where border data is sent or received. The produce-consume semantics of the communications then enforce the pipelining. Understanding that this program will produce the desired pipelined algorithm may seem subtle, however, and thus the process may be error-prone. In addition, if different widths or depths of pipelining are desired, the programmer must code these explicitly. While in our own experience this method of programming has been satisfactory, it did require understanding of the pipelined algorithm and some rewriting of loop indices. It remains to be seen how favorably other programmers will regard this option.

For an annotated sequential language such as Fortran D to express a pipelined computation, it appears that either the programmer or the compiler must transform the sequential loop into an equivalent skewed loop from which the pipelined execution order can readily be extracted, or they must perform some analogous conversion. (For a discussion of loop skewing transformations, see e.g. [10].) Having the programmer write the skewed loop appears to be the least desirable option, as it is difficult and error-prone to make this transformation manually. On the other hand, it is an open research issue as to how well compilers will be able to recognize situations that require pipelining and transform the code into efficient parallel code, although the Fortran D project has recently reported some preliminary success with this problem [2]. The challenges will include recognizing situations that require pipelining but where where the loops, and data dependencies, are not as simple as in the example in this section, as well as making optimizations such as pipeline width and depth automatically.

4 "Filter" Computational Kernel

4.1 Basic Algorithm

The code in the Filter kernel is representative of much of the remaining computationally intensive code in the system. It also performs sweeps alternately across each of the three spatial directions of the main data structure, but the computations at each grid point depend only upon the values of neighboring grid points from the *previous* sweep. For example, a sweep in the i direction may have the form

```
DO J = 1, NJ
  DO K = 1, NK
    DO I = 1, NI-1
      D(I) = A(I+1,J,K) - A(I,J,K)
    DO I = 2, NI-1
      E(I) = D(I+1) - 2*D(I) + D(I-1)
```

The crucial distinction is that the only data dependencies are between the two "DO I" loops, not within them as in the Smooth kernel. The variables D and E are temporary variables that, while conceptually three dimensional, have been made one dimensional in the sequential code to optimize storage. Some consequences of this optimization for parallelization are mentioned below.

This portion of the code has trivial, fine grain data parallelism in each inner loop, as the operations for all I, J, and K can be performed concurrently. The only real impediment to achieving an efficient parallel implementation is the cost of the communications that are required between the two loops. On most scalable MIMD multiprocessors, both the latency and bandwidth costs of communications are fairly high. It may be desirable to aggregate communications to reduce the number of messages sent, and to reorder computations to mask inter-loop communications with computations, in order to reduce the effects of the communication costs on the overall execution time of the parallel algorithm. Both of these optimizations require transformations of the loops.

4.2 Alternative Parallel Implementations

There are two main, orthogonal issues that we considered in determining how to implement this kernel efficiently. They are :

1. *1D, 2D, or 3D Block Mapping of Data Structures to Processors* – As in the Smooth kernel, the main data structures may be partitioned along one, two, or all three spatial dimensions. Higher dimensional partitionings have the advantage that they require less total data to be communicated, since fewer grid points are border points. As mentioned

before, higher dimensional partitions may have the disadvantage of requiring more work by the programmer.

2. *Reordering of Computations and Communications* – There are two main places where communication costs associated with latency and/or bandwidth can be reduced by reordering the computation. First, due to the latency costs associated with each message, it is preferable to aggregate all the border elements produced during a sweep and send them to the neighboring processor in one message, rather than individually as they are calculated. Note that additional data structures are required as a consequence of this optimization; for example in the code fragment above, D and E may have to become three dimensional data structures.

Second, if the sweeps are implemented in the most straightforward way, then there is likely to be significant processor idle time while processors are waiting for communicated values. This is because, in a case like the above code fragment, half of the border values in each sweep will be produced last in the first loop, and then sent to the neighboring processor, but required first in the second loop, which will thus be idle until the communication is completed. However the order of computation within each loop is totally arbitrary. Thus an obvious optimization is to move all computations involving border points to the beginning of each sweep, so that the time between when any data is sent to a neighboring processor and when it is needed by that processor is an entire sweep. The reordering should mask some or all of the communication time with computations. This optimization may also require expanding the dimensions of some data structures.

4.3 Computational Comparisons

Table 2 summarizes the results of some of our tests comparing 1D to 2D data partitionings, and overlapped (border computations and communications moved to the beginning of each sweep) to non-overlapped (standard loop order) communications. The results indicate that reordering the computations to overlap communication and computation is quite advantageous for this algorithm on this parallel machine. The parallelism achievable in the non-overlapped algorithm seems to be quite limited, while the overlapped algorithm seems to successfully mask much of the communication cost with computation. The tests also show a slight gain by using a 2D rather than a 1D mapping. This gain presumably is due to the smaller amount of data that is communicated. As was seen in Section 3.3, however, it this advantage is offset somewhat by the higher costs of sending noncontiguous data as messages in the 2D-mapped version as opposed to contiguous data in the 1D-mapped version. We did not program a version of this kernel with a 3D mapping. There is little or no reduction in the amount of data that is communicated in a 2D versus a 3D mapping for this particular problem size and number range of processors, so there appears to be little advantage to going to a 3D mapping, and the disadvantage that there is an increase in programming complexity.

4.4 Implications for Parallel Language Design

The type of computation performed in the Filter algorithm is very common in scientific computation. For example, the same pattern is found in a Jacobi iterative method for

Table 2: Filter Test Results

1D Mapping		
Non-Overlapped Communications		
Processors	Time	Speedup
16	15.4990	10.839
32	13.0884	12.836
1D Mapping		
Overlapped Communications		
Processors	Time	Speedup
16	12.8029	13.122
32	7.2037	23.321
2D Mapping		
Overlapped Communications		
Processors	Time	Speedup
4x4	12.5331	13.405
8x4	6.7114	25.032
Estimated Single Processor time = 168		
Problem sizes: 96x32x24x5		
Times are in seconds		

solving linear equations, or in any grid-based method where the value of a grid point at a given time step depends upon its value and those of some neighboring grid points at the previous time step. Thus, since communication latency and bandwidth costs are likely to remain significant in comparison to floating point operation costs on scalable MIMD multiprocessors, it is likely that optimizations such those described above, i.e. loop reorderings to aggregate communications and to assure that border values are computed and sent by one processor well before they are needed on another, will remain necessary for good parallel efficiency.

In an explicitly parallel SPMD programming language such as DINO, the programmer is able to express such reorderings of computation and communication explicitly, at the cost of rewriting the sequential loops. Our experience has been that this is satisfactory, but that the programmer needs to be concerned with transformations, such as the explicit calculation of index ranges, that ideally should not be the programmer's responsibility. In addition, the amount of such work increases with the number of dimensions that the data structure is partitioned along.

In an annotated sequential language, one would hope that the programmer would write the standard sequential code along with data distribution annotations, and the compiler would perform the desired optimizations. It is an interesting research issue to determine whether compilers will be able to automatically determine the needs for the types of optimizations discussed in this section, and make them efficiently. Of the two optimizations described above, it seems quite likely that the compiler will be able to make the first, aggregation of communication. It is less clear how often the compiler would be able to effectively make the second type of optimization, reordering the computation to better overlap computation and communication.

5 Summary and Ongoing Research

Parallelization of this computational fluid dynamics code has presented interesting challenges that are common to many scientific codes. While the data structures in the code are regularly shaped arrays and the main computational kernels are fairly simple algorithms, each of the kernels has required transformations and optimizations from the sequential code to achieve good parallel efficiency. In one case this included the use of pipelining, in the second case the reordering of computations to aggregate-communications and to overlap communication with computation. The overall design of the code has also required us to consider various mappings of the data structures to the processors in order to determine which mapping leads to the best performance. From the point of view of parallel programming language design for distributed memory multiprocessors, even these fairly simple algorithms point out challenges for either explicitly parallel or annotated sequential programming languages. These languages will need to make it easy to express or obtain nearly optimal versions of these algorithms, as well as to experiment with various distributions of the data structures.

We are currently completing the implementation of the entire TLNS3D code and are beginning to test the parallel algorithm on model problems. This work includes various considerations, for example effects related to the multigrid algorithm, that are not discussed here. The development and performance of the overall algorithm will be discussed in a future paper.

References

[1] W. Briggs, *A Multigrid Tutorial*, SIAM, Philadelphia, 1987.

[2] S. Hiranandani, K. Kennedy, C. Tseng, "Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines," Technical Report TR91-170, Department of Computer Science, Rice University, Nov. 1991.

[3] G. Fox, S. Hiranandani, K. Kennedy, C, Koelbel, U. Kremer, C. Tseng, M. Wu, "Fortran D Language Specification," Center for Research on Parallel Computation Technical Report CRPC-TR90079, Rice University, Dec. 1990.

[4] P. Mehrotra, J. Van Rosendale, "Programming Distributed Memory Architectures using Kali," ICASE Report 90-69, Institute for Computer Applications in Science and Engineering, Hampton, VA., Oct. 1990.

[5] G. Li, T. Coleman, "A Parallel Triangular Solver for a Hypercube Multiprocessor," *Hypercube Multiprocessors 1987*, SIAM, 1987, pp. 539-551.

[6] M. Rosing, R. Schnabel, "Efficient Language Constructs for Large Parallel Programs – An Overview of Dino2", Technical Report CU-CS-578-92, Department of Computer Science, University of Colorado, Jan. 1992.

[7] M. Rosing, R. Schnabel, R. Weaver, "The DINO Parallel Programming Language," *Journal of Parallel and Distributed Computing*, 13, 1991, pp. 30-42.

[8] M. Rosing, R. Schnabel, R. Weaver, "Scientific Programming Languages for Distributed Memory Multiprocessors : Paradigms and Research Issues," University of Colorado Computer Science Technical Report CU-CS-537-91, July 1991, to appear in *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, J. Saltz and P. Mehrotra, eds., 1992.

[9] V. Vatsa, B. Wedan, "Development of an Efficient Multigrid Code for 3-D Navier-Stokes Equations", *AIAA 20th Fluid Dynamics, Plasma Dynamics and Laser Conference*, June 12-14, 1989.

[10] M. Wolfe, "Loop Skewing: The Wavefront Method Revisited", *International Journal of Parallel Programming* 15, 1986, pp. 279-293.