Software Support for Irregular and Loosely Synchronous Problems

A. Choudhary G. Fox
S. Hiranandani K. Kennedy
C. Koelbel S. Ranka
J. Saltz

CRPC-TR92258
May 1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Appears in Computing Systems in Engineering, Vol. 3, Nos. 1-4, pp. 43-52, 1992.

Computing Systems in Engineering Vol. 3, Nos 1–4, pp. 43–52, 1992
Printed in Great Britain.

0956-0521/92 $5.00 + 0.00

SOFTWARE SUPPORT FOR IRREGULAR AND LOOSELY SYNCHRONOUS PROBLEMS

A. CHOUDHARY,† G. FOX,† S. HIRANANDANI,‡ K. KENNEDY,‡ C. KOELBEL,‡
S. RANKA†‖ and J. SALTZ§

†NPAC, 111 College Place, Syracuse University, Syracuse, NY 13244-4100, U.S.A.
‡CRPC, CITI, P.O. Box 1892, Rice University, Houston, TX 77251-1892, U.S.A.
§ICASE, MS 132C, NASA Langley Research Center, Hampton, VA 23665, U.S.A.

Abstract—A large class of scientific and engineering applications may be classified as irregular and loosely synchronous from the perspective of parallel processing. We present a partial classification of such problems. This classification has motivated us to enhance Fortran D to provide language support for irregular, loosely synchronous problems. We present techniques for parallelization of such problems in the context of Fortran D.

1. INTRODUCTION

Although parallel computer systems have been widely available for several years, they have not yet fulfilled their enormous promise. In spite of the widespread interest in parallel systems, few scientists and engineers are using parallel machines to do their most important calculations, relying instead on conventional supercomputers. There are two reasons for this. First, parallel computer systems have only recently become powerful enough to outperform conventional supercomputers. Second, and more importantly, there exists no machine-independent programming interface for parallel machines that can achieve an efficiency comparable to programs hand coded in languages that reflect the specific underlying architectures. This second problem is particularly troublesome because it puts the parallel programming investment at risk—if a program is converted at great effort to run on a parallel machine, the investment may be lost when the next generation of parallel computers emerges with an entirely different programming interface. Today, each new parallel architecture requires a significantly different software implementation.

1.1. *Software model*

An important lesson learned from using parallel machines has been the need for a close coupling between software and applications. Even though the problems that we and others have looked at tend to be in a limited domain, predominantly scientific and engineering simulations, we expect this lesson to be valid in general. Good performance for a parallel machine requires a good mapping of the problem onto the machine. Getting this mapping "right" seems to imply a close coupling between the application requirements and the software environment. Good mappings for many large problems have been discovered by users tuning their codes "by hand" using relatively crude software approaches. The Caltech Computation Project, for example, developed 50 successful parallel applications using node Fortran or C plus message passing on a distributed memory MIMD multicomputer. Building on that success requires a more automatic method of detecting and implementing good problem mappings. Our thesis is that providing such an environment will be a great help toward establishing a portable programming model for parallel machines.

The success of hand-parallelization should be contrasted with the experience of parallelizing compilers where false dependencies often prevent the compiler from exploiting the available parallelism. We can understand this as follows: the problem has a computational graph (such as a mesh for many signal processing or partial differential equation algorithms) that needs to be mapped onto the underlying parallel machine topology. In hand-coding programs, users are responsible for identifying the problem and machine topology and performing the mapping. The automatic compiler approach to parallelizing the C, Fortran or ADA code version of the problem fails when the compiler is unable to identify the underlying graph and the relation between program components. This can happen for a number of reasons.

1. The compiler's analysis can simply fail, reporting a dependence when none exists. (This is a particular problem in the loosely synchronous problems in Sec. 3, due to the data structures required there.) In these cases, there is little the programmer can do except complain to the compiler vendor.
2. An actual dependence may be an artifact of a sequential optimization, such as reusing an array's storage to save memory. In these cases, it is often possible to rewrite the program to

‖ Author to whom all correspondence should be addressed.

allow parallelization, if the user can detect the problem.

3. The program may use an inherently sequential algorithm, or an algorithm with limited parallelism. For example, the standard method of solving a tridiagonal system uses a first-order recurrence that cannot be directly parallelized. In this case, the best option is to change to a different algorithm.

Our experience has been that fully automatic compilers often fail on realistic applications, although they may perform better on individual loop nests. Language such as *LISP, C* and CM Fortran have succeeded on larger-scale problems because unlike Fortran 77 or C, these "data-parallel" languages properly express the structure of the problem and its computation.

Generalizing from the above discussion, we feel that successful parallel software models must provide a mechanism for expressing the decomposition by the programmer (as in C with message passing extensions) or provide this mechanism indirectly (as in C*). We feel that the interaction of applications and software support (languages, run time systems) is very important for parallel computing. In other words, parallel computing demands "high-level" software support—software that precisely and effectively captures the structure of the application resulting in automatic generation of good parallel programs. Our belief is that there is no need to write software designed for a single specialized domain. On the other hand, it is very hard to design universal software models. Indeed, we define broad classes of computations (we now have a total of about ten) which together can cover a large range and each is itself large enough to warrant individually tailored category-specific software support. We believe that our approach can be effectively extended to a much broader range of application. Although this work was motivated by our Fortran D compiler project for SIMD and MIMD distributed memory machines, we believe the classification can immediately be used for these applications with other languages including C, C++ and ADA.

1.2. *Problem classification*

We have classified problems into five broad categories in terms of the parallelization and software support issues they address:

synchronous
loosely synchronous
asynchronous
embarrassingly parallel
loosely synchronous complex.

Each problem category covers a broad range of applications. Current data parallel languages such as C* and Fortran D provide language support for expressing regular synchronous and loosely synchronous problems. The success of the Fortran D compiler project is partly due to our experience in parallelizing this class of scientific applications. In this paper we examine scientific applications that are irregular and loosely synchronous in nature. We present an overview of techniques for parallelizing such problems. Although we use specific applications as examples, our parallelization techniques are applicable to other disciplines and are in no way restricted to these particular codes. We propose language extensions and compiler techniques that are useful for successfully expressing such problems in a data parallel language such as Fortran D.

Section 2 provides a review of the architectural classification for problems. In Sec. 3 we describe different subclasses of irregular and loosely synchronous problems. In Sec. 4, we discuss several parallelization strategies for the inclusion of these problems in the solution space of Fortran D.

2. PROBLEM ARCHITECTURE

We have looked at many applications in a detailed survey in Ref. 1. Our analysis of problem architecture or structure is based on a break-up of each problem into spatial (data) and temporal (control) aspects. Following Fox[2] we describe three problem architecture classes in terms of their temporal (time or synchronization) structure. The temporal structure of a problem is analogous to the hardware classification into SIMD and MIMD. The spatial structure of a problem provides the computational graph of the problem at a given instant and is analogous to the interconnect or topology of the hardware. The detailed spatial structure is important in determining the performance of an implementation but it does not affect the broad categories.

Synchronous problems are data parallel with the restriction that the time dependence of each data point is computed by the same operations. Both algorithmically and in the natural SIMD implementation, the problem is synchronized microscopically at each computer clock cycle. Such problems are particularly common in academia as they naturally arise in any description of a system in terms of identical fundamental units. We believe that Fortran D (in its current version) should be able to address almost all of these problems.

Loosely synchronous problems are also typically data parallel but now we allow different data points to be evolved with distinct algorithms. Points are also often connected in an irregular, data-dependent manner; for this reason we sometimes refer to this class as "irregular problems." Such problems appear when one describes the world macroscopically in terms of the interaction between irregular inhomogeneous objects evolved in a time synchronized fashion. Loosely synchronous problems are spatially irregular but temporally regular. This class is the main focus of this paper.

The asynchronous problem class is irregular in space and time. Because of this irregularity, it is difficult to give general methods for parallelizing asynchronous problems. Some run well with functional decompositions, some require real-time synchronization techniques, and some have never been run successfully on massively parallel machines. For a detailed description of these classes the reader is referred to Ref. 3.

The class of embarrassingly parallel problems contains those problems that are totally disconnected in space and time. In these problems, no synchronization or communication is needed at all. (Actually, there is typically a final synchronized phase to collect the computed answers, but this only uses a small part of the total time.) Depending on the structure of the problem at each point, these can be run efficiently on either SIMD or MIMD hardware. We believe that Fortran D and other data-parallel languages should be able to express these problems well.

The class of loosely synchronous complex contains problems that are an asynchronous collection of loosely synchronous problems. A typical application in command and control belongs in this class. Each of the tasks in such an application is synchronous or loosely synchronous and can be parallelized individually. An overall asynchronous expert system coordinates the interaction between these tasks.

3. TYPES OF LOOSELY SYNCHRONOUS PROBLEMS

General purpose mapping tools and run time support must be able to handle a reasonably broad range of problems. As mentioned in the previous section, we intend to develop a parallel software environment for what we call loosely synchronous problems, linked to the Fortran D compiler at Rice and Syracuse Universities. This concept has been explained in detail in Refs 2, 4 and 5. The current Fortran D is designed to handle the special cases of synchronous problems and loosely synchronous problems with regular interconnection patterns. In extending the Fortran D environment, we have found it useful to divide this problem into several subclasses, which are described below. All loosely synchronous problems can, by definition, be divided into a sequence of concurrent computational phases. The differences between the subclasses lie in how the phases are separated and when the computation and communication patterns within the phases are set. In the remainder of this section, we will describe several subclasses of loosely synchronous problems, illustrated by actual applications. We present these subclasses to give an idea of the types of problems we plan to address, but we do not claim at this point to be in a position to present any kind of formal taxonomy. As described in Sec. 3.5, our classification is of course not complete and we are continuing our study of problem structures.[2,4]

```
S1  do i=1,N
      S2  do j=1,M
            y(i) = y(i) + a(i,j)*x(col(i,j))
          end do
    end do
```

Fig. 1. Sparse matrix vector multiply.

3.1. *Static single phase computations*

A static single phase computation consists of a single concurrent computational phase, which may be executed repeatedly without change. Examples of static single phase computations are iterative solvers using sparse matrix vector multiplications (e.g. Ref. 6) and explicit unstructured mesh fluids calculations (e.g. Ref. 7). The key problem in efficiently executing these programs is partitioning the data and computation to minimize communication while balancing load. This partitioning then dictates the program's synchronization and communication requirements, which must also be computed. Because the computational pattern is only set at run time, this cannot be done directly by the compiler; instead, calls to a run time environment must be generated to do the partitioning dynamically. Reducing the overhead of these calls, both by reusing information computed in the calls and by performing the calls efficiently, is also vital for high efficiency. The PARTI library[8] and the Kali compiler[9] introduced the inspector/executor paradigm to perform these optimizations.

In the remainder of this section, we describe some of the details that must be considered in implementing these kernels.

In some cases, there is a straightforward relationship between the way we partition distributed arrays and the way we partition work. Figure 1 depicts a sparse matrix vector multiply. The integer array *col* is used to represent the sparsity structure of the matrix. Loop S1 sweeps over the matrix rows, while loop S2 sweeps over the columns of the sparse matrix and calculates the required inner product. If the sparse matrix vector multiply in Fig. 1 is to be carried out repeatedly, it is reasonable to partition x and y between processors in a conforming manner. In such a problem, we can follow the common convention of carrying out computational work associated with computing a value for distributed array element $y(i)$ on the processor onto which $y(i)$ is mapped.[10]

There are other common cases in which the assignment of distributed array elements to processors and assignment of work to processors cannot be coupled in such a straightforward fashion. Figure 2 depicts a loop that sweeps over the edges of a mesh; indirection is used to index array x on the right hand side of S3 while indirection is used to index array y on the left hand side of S4 and S5. In this loop, it appears to be

```
C This is a simplified sweep over edges of a mesh. A flux across a
C mesh edge is calculated. Calculation of the flux involves
C flow variables stored in array x. The flux is accumulated to array y.

do i = 1, N
  S1 n1 = nde(i, 1)
  S2 n2 = nde(i, 2)
  S3 flux = f(x(n1), x(n2))
  S4 y(n1) = y(n1) + flux
  S5 y(n2) = y(n2) - flux
end do
```

Fig. 2. Another example of static single phase.

advantageous to assign each iteration of loop to a single processor. By doing this, we avoid having either to recalculate or to communicate values for *flux*, since $y(n1)$ and $y(n2)$ appear on the left hand sides of statements. We can see that we must now determine separately how to partition distributed array elements and loop iterations.

3.2. *Multiple phase computations*

A multiple phase computation consists of a series of dissimilar loosely synchronous computational phases. Such applications usually have several parallelizable loops that involve a variety of distributed arrays. In this section, we will only consider the case where each individual phase is a static single phase computation as defined above. Examples of these computations include unstructured multigrid (e.g. Ref. 11), parallelized sparse triangular solver (e.g. Refs 12 and 13), particle-in-cell codes (e.g. Refs 14 and 15), and vortex blob calculations.[16] The key problem in implementation is again partitioning computation and data, but now the task is complicated because the interfaces between phases must be considered in the partitioning. The synchronization and communication requirements are similarly complicated by the multiple phases. As for static single phase computations, this partitioning must be performed at run time. Saltz and coworkers have recently extended the PARTI library to include *incremental* routines which will be applicable to these problems.[17] It is not clear whether further extensions will also be needed. It is clear, however, that these computations can again take advantage of saving information computed in the run time environment.

In the remainder of this section, we describe the unstructured multigrid application to show some of the implementation complexities of this class.

Unstructured multigrid codes[11] carry out mesh relaxation over each of several increasingly refined meshes $M_1, \ldots, M_n$. Figures 3 and 4 depict two levels of these meshes from a fluid dynamics code that we have parallelized. Both of these grids represent the

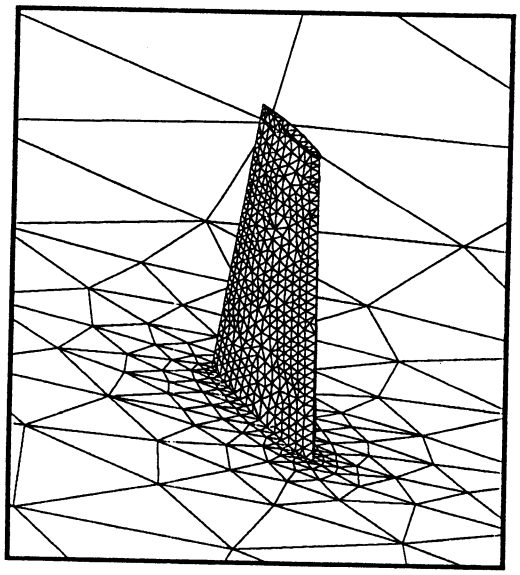

Fig. 3. Unstructured multigrid—coarse grid.

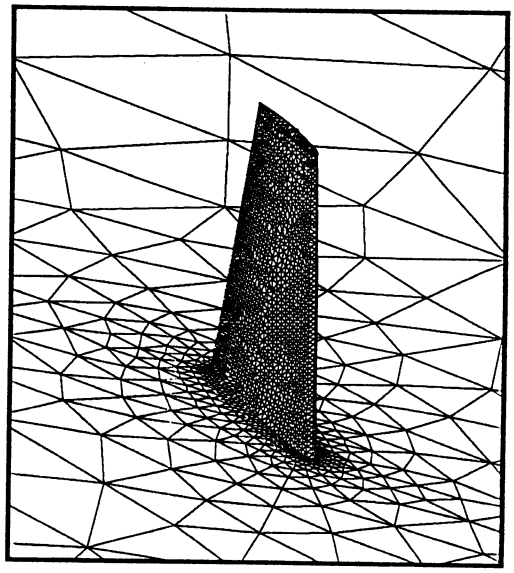

Fig. 4. Unstructured multigrid—refined grid.

same physical geometry but the grid in Fig. 4 is more highly refined than the grid in Fig. 3. The algorithm alternates between sweeping over each mesh and moving data between meshes, as shown in Fig. 5. The meshes $M_1, \ldots, M_n$ should be partitioned so that

(1) sweeps over each mesh M_i do not require excessive amounts of interprocessor communication,
(2) the computation involved in sweeping over each mesh should exhibit good load balance and
(3) interpolations and projections should only require modest amounts of data movement.

We have partitioned the grids in our example using the partitioner described in Ref. 18 with good results, but there are many other possible partitioners.

3.3. *Adaptive irregular computations*

An adaptive irregular computation consists of a loosely synchronous computation executed repeatedly in which the data access pattern changes between iterations. The changes may be gradual, reflecting adiabatic changes in the physical domain, or large-scale, reflecting additions to a data structure. Molecular dynamics applications often exhibit the first behavior because interactions between particles are implemented by neighbor lists which change as the atoms move.[19] Adaptive PDE solvers are often examples of the second behavior, as discussed below. Other examples with which we are familiar include some vision algorithms including region growing and labeling,[20,21] statistical physics simulations near critical points[22] and the particle sorting phase of a direct monte carlo simulation.[23] The key problems in implementing these algorithms are to react quickly

```
C Greatly oversimplified multiple mesh computation - Sweep over coarse
C mesh, transfer information to fine mesh, sweep over fine mesh
C and transfer information back to coarse mesh. xc,yc represent coarse
C mesh variables, xf,yf represent fine mesh variables.
C Typically these computations are carried out in an iterative manner.
C Sweep over coarse mesh

do i = 1, Ncoarse
 do j = 1, Kcourse
 yc(i) = yc(i) + ac(i,j) * xc(ic(i,j))
 end do
end do

C Transfer data from coarse mesh to fine mesh
do i = 1, Nfine
 do j = 1, Ninterpf(i)
 xf(i) = xf(i) + weightf(i,j) * yc(interpf(i,j))
 end do
end do

C Sweep over fine mesh
do i = 1, Nfine
 do j = 1, Kfine
 yf(i) = yf(i) + af(i,j) * xf(if(i,j))
 end do
end do

C Transfer data from fine mesh to coarse mesh
do i = 1, Ncoarse
   do j = 1, Ninterpc(i)
   xc(i) = xc(i) + wc(i,j) * yf(interpc(i,j))
   end do
end do
```

Fig. 5. Static multiple phase.

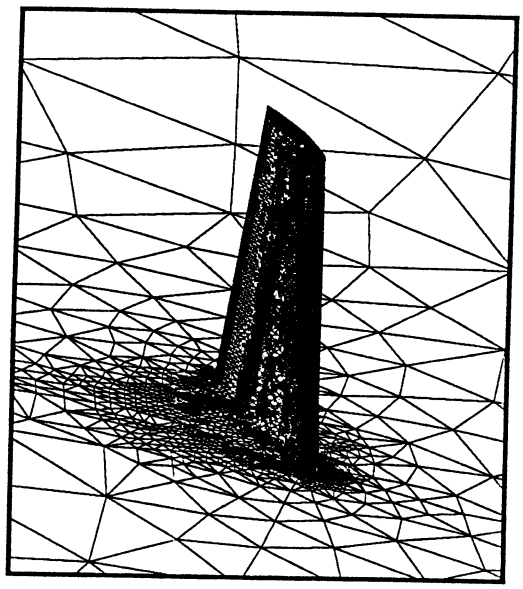

Fig. 6. Adaptive grid—after refinement.

to changes in the data structure. The physical and numerical properties of these algorithms typically guarantee that large-scale restructuring of data is only needed infrequently. New constructs are needed, however, to communicate this to the underlying system software.

Adaptive algorithms are useful for solving Euler and Navier–Stokes problems that arise in aerodynamics. In these algorithms, mesh refinement is carried out in portions of a computational domain where it is estimated that additional resolution may be required (e.g. see Refs 24 and 25). The grid in Fig. 6 is an adaptive refinement of the grid in Fig. 4. The initial mesh-point distribution is determined from the geometry of the airfoil to be simulated. Adaptive mesh refinement is achieved by adding new points in regions of large flow gradients. A simple version of the algorithm is presented in Fig. 7. The remapping needs to be performed before the inner do loop is executed.

3.4. *Implicit multiphase loosely synchronous computations*

An implicit multiphase computation is one containing irregular inter-iteration dependencies. The problems discussed thus far have consisted of a sequence of clearly demarcated computational phases. There are a number of problems in which there are inter-iteration dependencies that might at first appear to inhibit parallelization. These data dependency patterns

(1) are known only at run time but,
(2) can be fully predicted before a program enters the irregular loop or loops. Figure 8 shows the back substitution phase of a sparse matrix factorization, a simple algorithm of this type.

This is similar to solving sparse triangular systems of linear equations arising from ILU preconditioning methods.[26,27] Another example of this class is the tree generation phase of the adaptive fast multipole algorithms for particle dynamics.[28,29] The key problem in implementing these algorithms is to detect and exploit opportunities for partial parallelization. In Fig. 8, it is often possible to carry out many simultaneous row substitutions. The sparsity structure of the system determines which row substitutions can be carried

```
C Adaptive Two Mesh Algorithm
C Coarse mesh U_c covers entire domain

C Refined mesh U_r covers "active" portion of domain
C Location, shape, and size of refined mesh all change


do k_c = 1 to K
  Sweep over the U_c
  Flag region of U_c that should be refined.
  If flagged region is not empty.
    Modify shape of U_r
    Interpolate boundary values for U_r from U_c.
    do k_r = 1 to K_r
      Sweep over U_r
    end do
    Inject values of U_r into U_c
end do
```

Fig. 7. Adaptive two mesh algorithm.

```
C Implicit Multiphase
C Example - sparse triangular solve (unit diagonal)

do i = 1, N
  y(i) = rhs(i)
  doj = ija(i), ija(i+1) - 1
    y(i) = y(i) - a(j) * y(col(j))
  end do
end do
```

Fig. 8. Implicit multiphase.

out concurrently; however, this information is only available at run time. In such problems, we carry out a form of run time preprocessing with the goal of defining a sequence of loosely synchronous computational phases. In bus based shared memory multiprocessors, we have demonstrated that it is possible to integrate run time parallelization with compilers.[30] We anticipate that it will also be possible to link run time parallelization with compilers aimed at scalable multiprocessors and have carried out preliminary work in this area.

A more difficult problem is that of run time aggregation of work and data. When we carry out sparse computations such as sparse triangular solves or sparse direct factorizations,[31] our run time preprocessing can determine the *number and content* of the concurrent computational phases that will comprise a computation. We will call this process *run time aggregation or run time tiling*. There have been a variety of numerical algorithms to carry out what we call run time tiling for multiprocessor and vector computers; a small subset of this extensive collection of methods may be found in Refs 32 and 33.

3.5. *Static and dynamic structured problems*

This class of problems consists of highly structured computations on sets of subdomains that are coupled in an irregular manner. The computations on each individual subdomain are frequently highly structured, but the computational relationship between the subdomain is known only at run time. Furthermore, the relationship between the subdomains frequently changes dynamically during the course of a computation. The examples described in this subsection differ from the examples described in the previous four subsections in that the previous problems consist of irregularly coupled "points", whereas we now deal with collections of nontrivial structures. Examples of such problems include the adaptive mesh method described below and a combined hydrodynamics and particle astrophysical simulations implemented by Edelsohn at Syracuse.[34] The key to efficiency on these problems is to aggressively apply optimizations to the regular subproblems, which can be implemented with lower overheads. Also, the larger granularity of the coupled subproblems can be exploited to reduce preprocessing overheads and also reduce memory requirements.[35]

An example of this class is shock profiling as described in Ref. 35. The basic problem is to solve a partial differential equation in the presence of a shock, computing the profile (detailed shape) of the shock. Resolution of the profile implies that a highly refined grid must be used in a neighborhood of the shock. The method initially computes the solution on a coarse mesh. An error estimator is then applied to determine the regions that will be covered by a refined mesh. An example mesh from this two-level refinement is shown in Fig. 9. The solution is time-dependent.

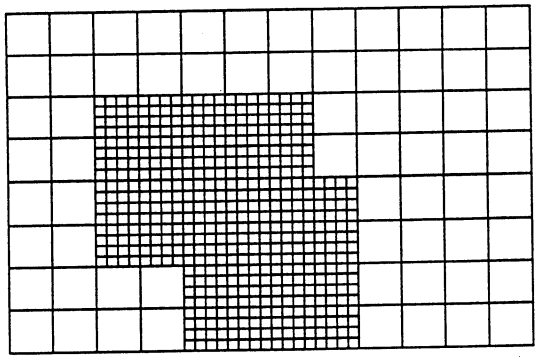

Fig. 9. Two-mesh refinement.

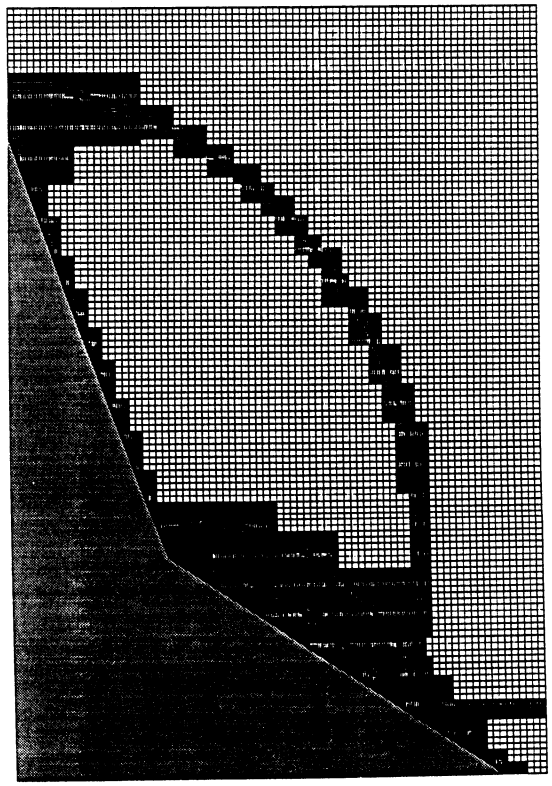

Fig. 10. Mesh used to calculate interaction of planar shock wave with a double wedge.

Time-marching on the refined mesh is performed by taking many (e.g. 100) time steps on the refined mesh for a single coarse-grid time step. The refined mesh is dynamic—its location, shape, and size all change. This means that the relationship of the two meshes will change during the execution of the program. Hence the structures of the computations change with time and a nonuniform communication pattern arises due to the sharing of data between grids. This example also generalizes to a full structured adaptive multigrid. An example of a mesh employed in such a full structured adaptive multigrid may be seen in Fig. 10. This mesh is used in a solution of the Euler equations used to simulate interaction of a planar shock wave with a double wedge.[36]

4. CONCLUSIONS

In this paper, we presented a partial classification of scientific and engineering applications which are irregular and loosely synchronous from the perspective of parallel processing. This classification should be helpful in extending Fortran D to permit its application to a large class of loosely synchronous problems. There are a few important tasks which may be necessary for the above. While we have made significant progress on each of these tasks, there is still much work that remains to be carried out.

Firstly there is a need for development of automatic and semi-automatic data partitioners and a strategy for incorporating these in a compiler. Currently, partitioners are designed using programmers' *a priori* knowledge about a problem's computational structure and its expected computational behavior. There has been significant progress in the development of robust partitioners for static single phase loosely synchronous calculations, see e.g. Refs 18 and 37, but much work remains to be done in order to deal with other problem classes. Similarly, we have proposed a scheme for integrating data partitioners into compilers that appears to be appropriate for static single and perhaps for multiphase loops.[17] Much work is needed to generalize these methods before they are able to handle the more challenging classes of computations. Some preliminary work along these lines has been reported in Refs 38 and 39.

Time-dependent or iterative loosely synchronous computational problems can exhibit a range of dynamic behaviors. These behaviors can be divided into three rough categories:

(A) data dependency pattern is static and does not change between iterations;
(B) data dependency pattern is modified on occasions but between changes, the dependency pattern remains static for many iterations;
(C) data dependency pattern changes every iteration.

Problems in category A would fall either into the class of static, single phase loosely synchronous computations (Sec. 3.1) or into the class of static, multiple phase loosely synchronous computations (Sec. 3.2), while problems in categories B and C would fall into the class of unstructured adaptive problems (Sec. 3.3 and 3.4) or structured adaptive problems (Sec. 3.5). It is also useful to categorize irregular problems by whether a given *iteration or time step* is composed of multiple, dissimilar loosely synchronous computational phases. In such cases, it is often necessary to partition a problem in a way that takes into account all of the computational phases in an iteration. Further, there are issues related to partitioning and run time aggregation,[32,33,38] which can affect the performance of these problems.

Secondly, we need to standardize extensions to Fortran D to facilitate the specification of partitioning strategies and irregular meshes. These extensions will be used to

(1) indicate which loops in a program should be taken into account when considering how to partition distributed arrays;
(2) allow users to force the selection of a particular partitioner;
(3) allow users to assert that a given set of loop dependencies can or cannot change when the loop is iteratively invoked; and
(4) allow users to specify the granularity with which parallelism is to be exploited.

In Ref. 17, we have proposed extensions (and developed run time support) that fulfil the first two of the above mentioned goals. There is also a need for development of new data structures targeted towards problems in which highly structured computations on a set of subdomains are coupled in an irregular manner. We are particularly interested in representing structured adaptive problems in which subdomains are coupled by irregular tree dependency structures.

In this volume, in the paper by Sussman *et al.*, we describe the portable run time support for static single and multiphase problems, and for static structured problems. This run time support is oriented towards distributed memory MIMD architectures. The run time support for static single and multiphase problems has also been ported to SIMD architectures, but the static structured run time support has as yet not been implemented on an SIMD architecture. There is still a clear need for development of appropriate run time support for Adaptive Irregular Computations, Implicit Multiphase Loosely Synchronous Computations, and Dynamic Structured Problems targeted towards SIMD and MIMD distributed memory architectures.

Acknowledgements—We wish to thank Mavriplis at ICASE for the mesh illustrations depicted in Figs 3, 4, 6 and 7 and thank James Quirk for the mesh depicted in Fig. 10. The research of Alok Choudhary, Geoffrey Fox and Sanjay Ranka was supported in part by DARPA under contract no. DABT63-91-C-0028. The content of the information

does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. The research of Joel Saltz was supported by the National Aeronautics and Space Administration under NASA contract NAS1-18605, while the author was in residence at ICASE, and supported from NSF grant ASC-8819374 while the author was in residence at Yale University. The research of Seema Hiranandani, Ken Kennedy and Charles Koelbel was supported in part by the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center. Additional support was provided by DARPA under contract no. DABT63-91-C-0028. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

REFERENCES

1. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Computers*, *Prentice-Hall*, Englewood Cliffs, New Jersey, 1988.
2. G. C. Fox, "Parallel problem architectures and their implications for parallel software systems," DARPA Workshop, Providence, RI, February 1991 (also SCCS-78).
3. G. C. Fox, "The architecture of problems and portable parallel software systems," Supercomputing 91, November 1991, also SCCS-134.
4. G. C. Fox, "What have we learnt from using real parallel machines to solve real problems?" *The Third Conference on Hypercube Concurrent Computers and Applications*, Vol. 2, January 1988.
5. G. C. Fox, "Fortran D as a portable software system for parallel computers," Presentation at *Supercomputing USA/Pacific '91 Conference*, Santa Clara, CA, June 1991 (also SCCS-91).
6. Y. Saad, "Communication complexity of the Gaussian elimination algorithm on multiprocessors," *Linear Algebra Applications* **77**, 315–340 (1986).
7. D. L. Whitaker, D. C. Slack and R. W. Walters, "Solution algorithms for the two-dimensional Euler equations on unstructured meshes," in *Proceedings AIAA 28th Aerospace Sciences Meeting*, Reno, Nevada, January 1990.
8. R. Das, J. Saltz and H. Berryman, "A manual for parti runtime primitives—revision 1 (document and parti software available through netlib)," Interim Report 91-17, ICASE, 1991.
9. C. Koelbel and P. Mehrotra, "Compiling global name-space loops for distributed execution" (to appear in *IEEE Transactions on Parallel and Distributed Systems*, July 1991), Report 90-70, ICASE, 1990.
10. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng and M. Wu," Fortran D language specification," Department of Computer Science, Rice COMP TR90-141, Rice University, December 1990 (also SCCS-42C).
11. D. J. Mavriplis, "Three-dimensional unstructured multigrid for the Euler equations," paper 91-1549cp, in *AIAA 10th Computational Fluid Dynamics Conference*, June 1991.
12. D. Baxter, J. Saltz, M. Schultz, S. Eisentstat and K. Crowley," An experimental study of methods for parallel preconditioned krylov methods," in *Proceedings of the 1988 Hypercube Multiprocessor Conference*, Pasadena, CA, pp. 1698, 1711, January 1988.
13. E. Anderson and Y. Saad, "Solving sparse triangular linear systems on parallel computers," *International Journal of High Speed Computing* **1**(1), 73–95 (1989).
14. D. W. Walker, "Characterizing the parallel performance of a large-scale, particle-in-cell plasma simulation code, *Concurrency: Practice and Experience* **2**(4), 257–288 (1990).
15. P. C. Liewer and V. K. Decyk, "A general concurrent algorithm for plasma particle-in-cell simulation codes, *Journal of Computational Physics* **85**(2), 302–322 (1989).
16. S. Baden, "Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors," *SIAM J. Scientific and Statistical Computation* **12**(1), 145–157 (1991).
17. R. Das, R. Ponnusamy, J. Saltz and D. Mavriplis, "Distributed memory compiler methods for irregular problems—data copy reuse and runtime partitioning," in *Compilers and Runtime Software for Scalable Multiprocessors* (edited by J. Saltz and P. Mehrotra), Elsevier, Amsterdam, to appear.
18. H. Simon, "Partitioning of unstructured mesh problems for parallel processing," in *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*, Pergamon Press, Oxford, 1991.
19. B. R. Brooks, R. R. Bruccoleri, D. B. Olafson, D. J. States, S. Swaminathan and M. Karplus, "Charmm: a program for macromolecular energy, minimization, and dynamics calculations," *Journal of Computational Chemistry* **4**, 187 (1983).
20. A. N. Choudhary and J. H. Patel, *Parallel Architectures and Parallel Algorithms for Integrated Vision Systems*, Kluwer, Boston, MA, 1990.
21. The DARPA Image Understanding Benchmark for Parallel Computers, *Journal of Parallel and Distributed Computing* **11**, 1–24 (1991).
22. P. D. Coddington and C. F. Baillie, "Cluster algorithms for spin models on MIMD parallel computers," *The Fifth Distributed Memory Computing Conference*, Charleston, South Carolina, 9–12 April.
23. L. Dagum, "Data parallel sorting for particle simulation," NASA Ames Research Report, September 1991.
24. G. Warren, W. Anderson, J. Thomas and T. Roberts, "Grid convergence for adaptive methods," paper 91-1592, in *AIAA 10th Computational Fluid Dynamics Conference*, June 1991.
25. J. J. Quirk, "An adaptive grid algorithm for computational shock hydrodynamics," Ph.D. thesis, Cranfield Institute of Technology, U.K., 1991.
26. P. Venkatakrishnan, "Preconditioned conjugate gradient methods for the compressible Navier–Stokes equations," *AIAA Journal* (June 1991).
27. P. Venkatakrishnan, J. Saltz and D. Mavriplis, "Parallel preconditioned iterative methods for the compressible Navier–Stokes equations," in *12th International Conference on Numerical Methods in Fluid Dynamics*, Oxford, U.K., July 1990.
28. G. C. Fox, "Hardware and software architectures for irregular problems architectures," invited talk at ICASE Workshop on Unstructured Scientific Computations on Scalable Multiprocessors, Nagshead, NC, October 1990 (also SCCS-111).
29. J. K. Salmon, "Parallel hierarchical N-body methods," Tech. Report, CRPC-90-14, Center for Research in Parallel Computing, Caltech, Pasadena, CA, 1990.
30. J. Saltz, R. Mirchandaney and K. Crowley, "Run-time parallelization and scheduling of loops," to appear in *IEEE Transactions on Computers*, 1991, Report 90-34, ICASE, May 1990.
31. I. S. Duff and J. K. Reid. *Direct Methods for Sparse Matrices*, Oxford Science Publications, Oxford University Press, New York, 1986.
32. A. George, M. T. Heath, J. Liu and E. Ng, "Sparse Cholesky factorization on a local memory multiprocessor," *SISSC* 327–340 (1988).
33. C. Ashcraft, S. C. Eisenstat and J. W. H. Liu, "A fan-in algorithm for distributed sparse numerical factorization," *SISSC* **11**(3), 593–599 (1990).

34. D. J. Edelsohn, "Hierarchial tree-structures as adaptive meshes," SCCS Report-193, Syracuse University.
35. H. Berryman, J. Saltz and J. Scroggs, "Execution time support for adaptive scientific algorithms on distributed memory machines," to appear in Concurrency: Practice and Experience 1991, Report 90-41, ICASE, May 1990.
36. J. Quirk, "An alternative to unstructured grids for computing gas dynamic flows around arbitrarily complex two-dimensional bodies," ICASE Report 92-7, 1992.
37. S. Hammond and R. Schreiber, "Mapping unstructured grid problems to the Connection Machine," Report 90-22, RIACS, October 1990.
38. R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol and K. Crowley, "Principles of runtime support for parallel processors," in *Proceedings of the 1988 ACM International Conference on Supercomputing*, St. Malo, France, pp. 140–152, July 1988.
39. L. C. Lu and M. C. Chen, "Parallelizing loops with indirect array references or pointers," *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, August 1991.
40. P. C. Liewer, E. W. Leaver, V. K. Decyk and J. M. Dawson, "Dynamic load balancing in a concurrent plasma PIC code on the JPL/Caltech Mark III hypercube," *Fifth Distributed Memory Computing Conference*, pp. 939–942, 1990.