Automatic Data Distribution of Small Meshes Coupled Grid Applications

Lorie Liebrock
Ken Kennedy

CRPC-TR94500
October, 1994

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Automatic Data Distribution of Small Meshes Coupled Grid Applications

Lorie M. Liebrock*
lorie@cs.rice.edu
Department of Computer Science
Rice University
2315 County Rd.
Calumet, MI 49913
Phone: (906)337-4205
Phone and Fax: (906)337-5817
CORRESPONDING AUTHOR

Ken Kennedy
ken@cs.rice.edu
Department of Computer Science
Rice University
6100 S. Main St.
Houston, TX 77005-1892

October 24, 1994

Abstract

Composite grid problems arise in important application areas, e.g., reactor simulation. Related physical phenomena are inherently parallel and their simulations are computationally intensive. Unfortunately, parallel languages, e.g., High Performance Fortran, provide little explicit support for these problems. We illustrate topological connections via a coupling statement and develop an algorithm that automatically determines distributions for composite grid problems with meshes that are small relative to the number of processors. Our algorithm's alignment and distribution specifications are input to the transformed High Performance Fortran program which applies the mapping for execution of the simulation code. Precompiler transformations, such as cloning for alignment specification, are described. Excerpts from a High Performance Fortran program before and after transformation illustrate user programming style and transformation issues. Some advantages of this approach are: transformations are applied before compilation, and allow communication optimization, and the distribution may be determined for any number of problems without recompilation; user determined distribution for parallelization is unnecessary; and portability is improved. We validate the algorithm using a number of reactor configurations. Two random distribution algorithms provide a basis of comparison with measures of load balance and communication cost. Experiments show that our algorithm almost always obtains load balance at least as good as, and often significantly better than, random algorithms while reducing the total communication per iteration by about 50% or in some cases as much as 90%.

1 Background: Composite Grid Problems

Detailed simulation of complex physical phenomena is computationally intensive. An increasing number of these computationally intensive problems involve more than one grid where each grid corresponds to a different physical entity. Here we focus on composite grid problems where all of the grids are either large enough to be distributed across all processors or small enough to fit on a single processor. Two important application areas for composite grid approaches are electric circuit analysis and nuclear reactor simulation. In nuclear reactor simulations, different grids are used for each of the reactor vessels, pipes, pumps, etc. [1]. Applications of this type are called multiblock, composite grid or irregularly coupled regular mesh (ICRM) problems. To provide an indication of the complexity of the topology(dimensionality, size, and connectivity) of these problems, Figures 1-5 show five of the diagrams from the Westinghouse AP600 advanced reactor design[1]. The AP600 design has two loops, with one hot leg, one

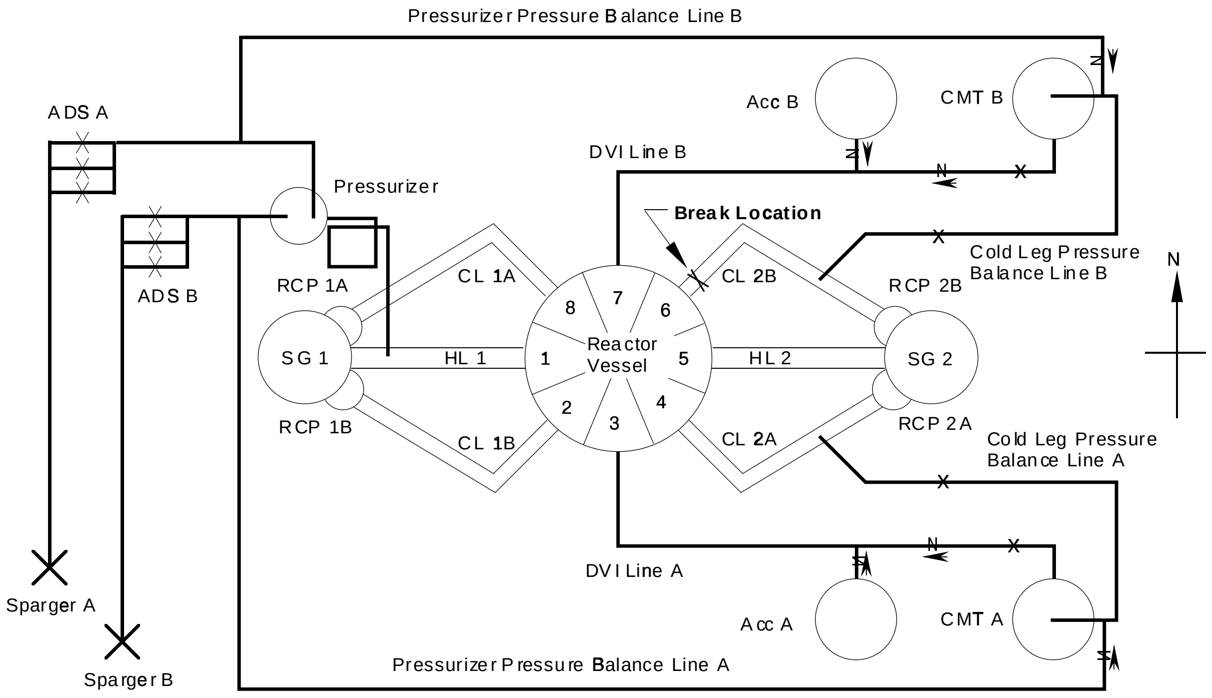

Figure 1: Plan view of AP600 model.

steam generator, two reactor pumps, and two cold legs in each loop. Figure 1 shows the overall plan of the reactor cooling system, automatic depressurization system and the passive safety injection system. Figure 2 illustrates the relationship of the components in the reactor vessel model. Figure 3 shows the relationship of the heat structures to the rest of the components in the reactor vessel model. Figure 4 illustrates the relationship of the components in the first coolant loop of the AP600 model. Figure 5 illustrates the relationship of the components in the safety systems of the model. This configuration has a total of 173 hydrodynamic components (with $10603 - D$ cells and $8651 - d$ cells) and 47 heat structures. The complexity of the AP600 topology illustrates the need for automatic distribution. The AP600 is used as the final test problem for our algorithm.

Many composite grid problems require the use of the fastest computers available, even for simplified simulations. For the solution of the grand challenge simulations in these problems, it is clear that parallelization will be necessary. For example, to verify the design criterion of the AP600 that it be capable of unsupervised operation

[1]These figures and the topology specifications for this problem were provided by Jim F. Lime who developed this TRAC model for the AP600 at Los Alamos National Laboratories with support from the Nuclear Regulatory Commission's Office of Nuclear Regulatory Research. Preliminary large-break loss-of-coolant accident results performed with TRAC-PF1/MOD2 [10].

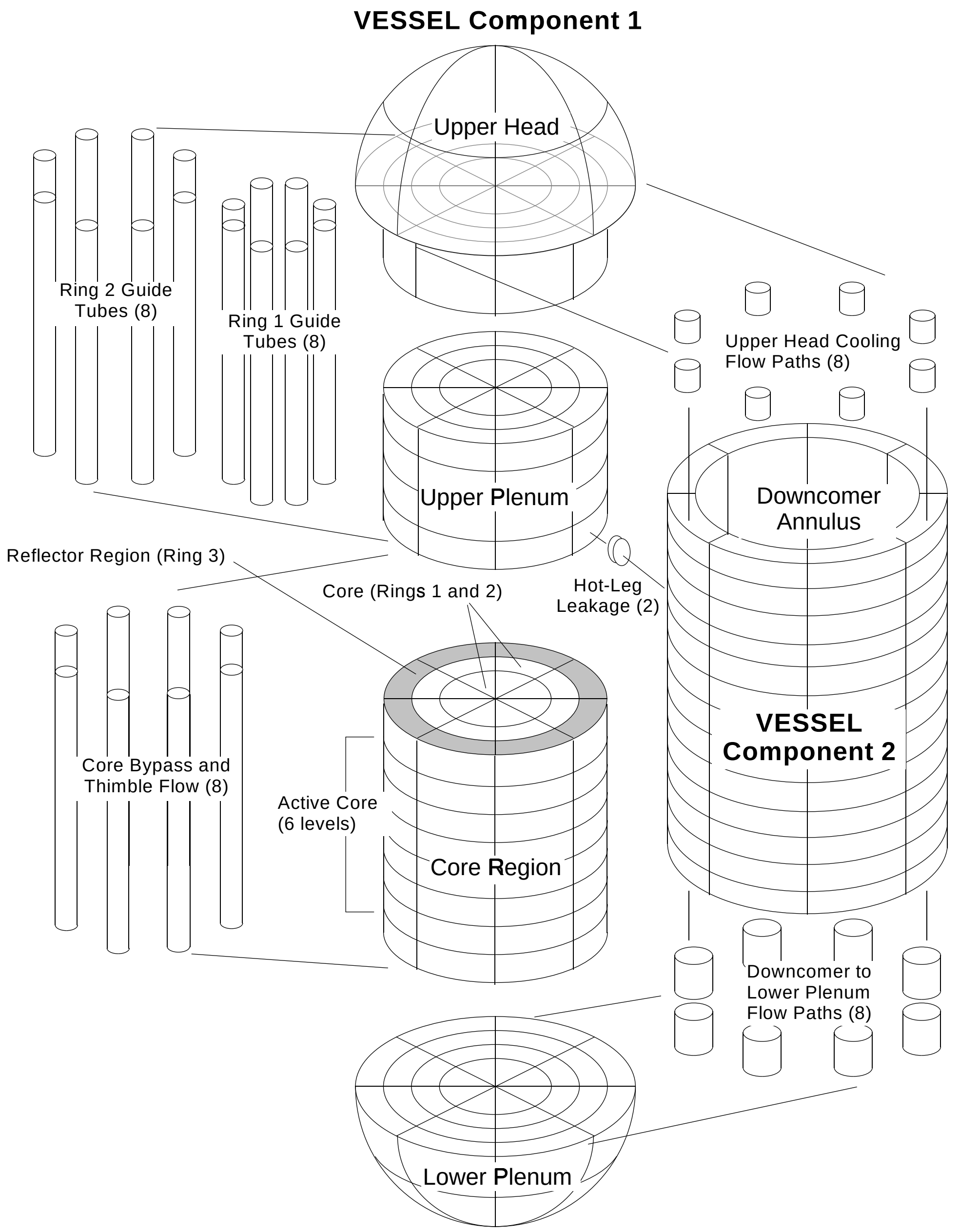

Figure 2: Isometric view of AP600 reactor vessel model.

for three days, simulations of many scenarios need to be run. Since simulation of the AP600 reactor currently takes approximately 40 times as long as real time using two communicating workstations (one running RELAP-5 and the other running Contain), each scenario takes about 120 days to run. Fortunately, the physical phenomena being simulated are inherently parallel. Unfortunately, their simulation is not necessarily easy to parallelize. Even with inherently parallel algorithms for composite grid problems, the generation of a parallel program with all of the clones of compute and coupling update routines to allow input of alignment and distribution specification is at best tedious and error prone. Further support of clones in programs makes code development more tedious and error prone as well. To make rewriting of applications codes such as TRAC and RELAP-5 for parallel processors practical, the resulting program must be portable, easy to develop, and easy to support [6]. For this reason we

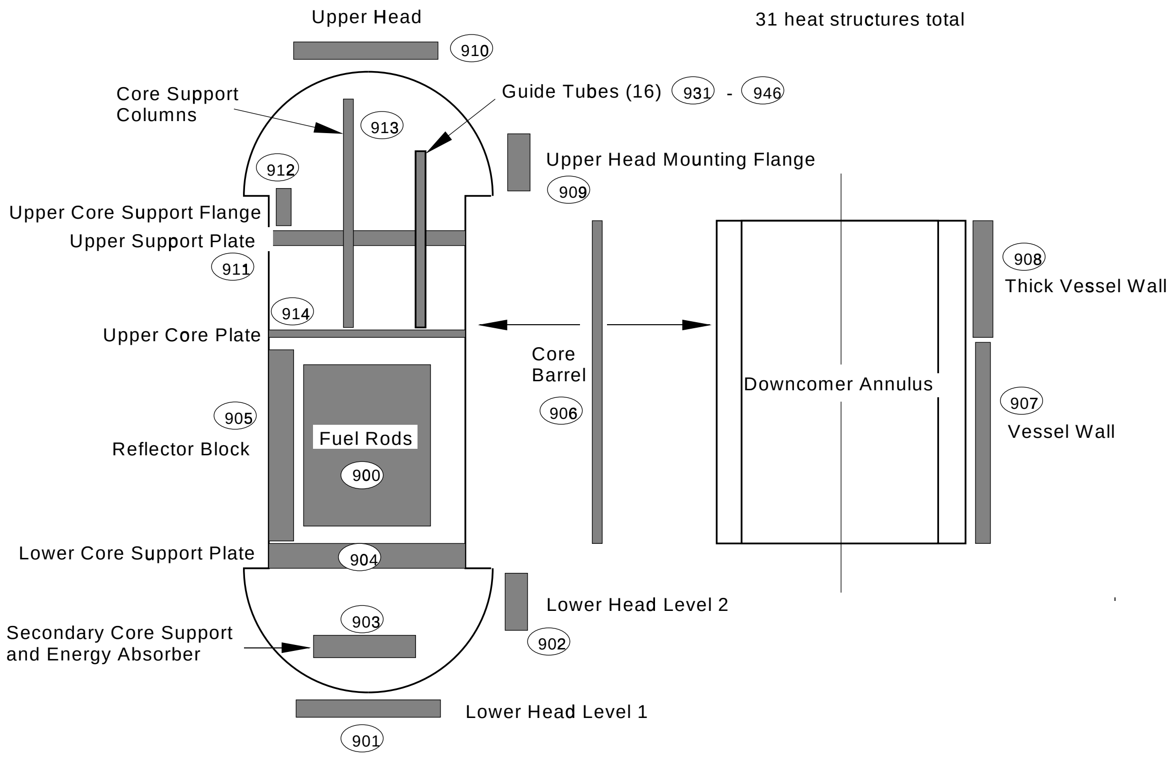

Figure 3: AP600 reactor vessel heat structure model.

discuss not only the automatic generation of alignment and distribution specifications, but also the automatic transformation of High Performance Fortran (HPF) programs to support the generation of clones and the use of alignment and distribution specifications. This means that future code development occurs on a single version of each compute and coupling update routines and is followed by rerunning the transformation process on the modified subroutine(s).

As a result of this discussion we have a set of computationally intensive simulation problems, which are difficult to parallelize efficiently, coming from inherently parallel physical phenomena. None of the current parallelization approaches for these problems, e.g., PARTI [2], save the programmer from having to decipher the grid assembly and determine data distributions. Since the description of the grid assembly is normally part of the input, this implies user intervention for data distribution for every new grid assembly input.

Since each grid is regular but the coupling between the grids is not necessarily regular, we call these problems partially regular. Parallel efficiency is significantly impaired in these partially regular problems when irregular distributions of data are used. Irregular data distributions tend to impair the use of regular communication optimization techniques, e.g., they may inhibit parallel communication generation and increase network contention. We are working to make parallelization of these applications easier for the user as well as efficient and applicable across a large class of architectures. Therefore, we must develop an approach that achieves more of the computational efficiency of a regular approach. To do this we exploit not only the regularity inside of each mesh, but also the topology of the connections between meshes to automatically determine distribution of data structures. We use an extension of Fortran D [3], coupling specification, to illustrate the information that must be extracted from the input file for automatic distribution. For nuclear reactor applications, mesh and coupling descriptions can be be obtained by translating the grid specifications found in the input file for simulators such as TRAC [1]. Therefore the coupling specifications presented here indicate what information must be extracted from the input. A portable style of programming in High Performance Fortran is introduced that allows analysis of these appli-

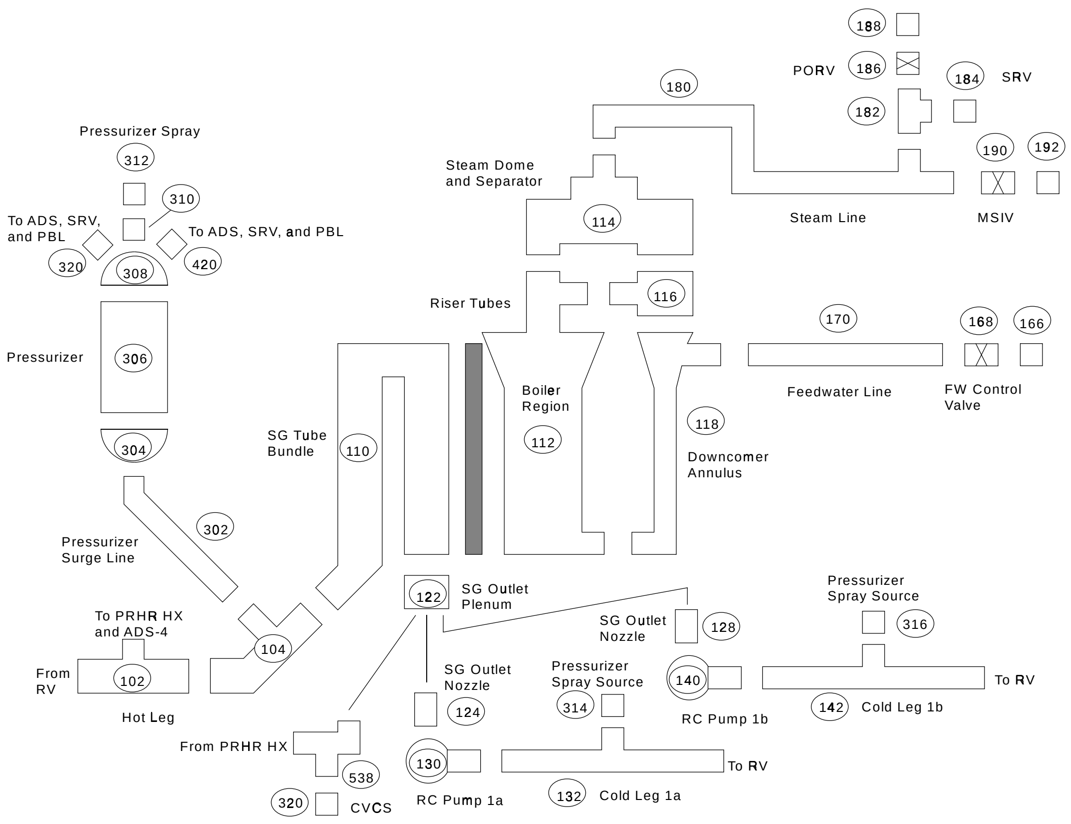

Figure 4: AP600 reactor coolant loop 1 model.

cations codes to determine the amount and type of computation and communication associated with each mesh and coupling. Next an algorithm for automatic distribution of small meshes is presented. Then excerpts from an abstracted application are used to illustrate HPF programming style and transformation. Finally we validate our work with measures of communication and computational load balance on four nuclear reactor configurations.

2 Topology Specification and Programming Style

Here we introduce a version of the coupling statement to illustrate the type of topology information that must be extractable from the runtime constant input. By the topology of a problem we mean the dimensionality, size and connectivity of the grids in the problem. A programming style is then developed to facilitate extraction of runtime constant input and analysis of computation and communication for automatic distribution.

2.1 Specification of Topological Connectivity

We illustrate the specification of topological connectivity via a statement that describes the connections between coupled grids. In the grid illustrations, dashed lines represent couplings and solid lines represent the boundaries of the grids.

As an example, consider two meshes, A and B, each of which are 40x40x40 with one coupled face. These meshes and the coupling between them are illustrated in Figure 6. This problem was used by J. Saltz [2] to show the feasibility of the PARTI multiblock runtime system for parallelization of composite grid problems. Here we need to couple the face of A where the second dimension index is 40 to the face of B where the second dimension

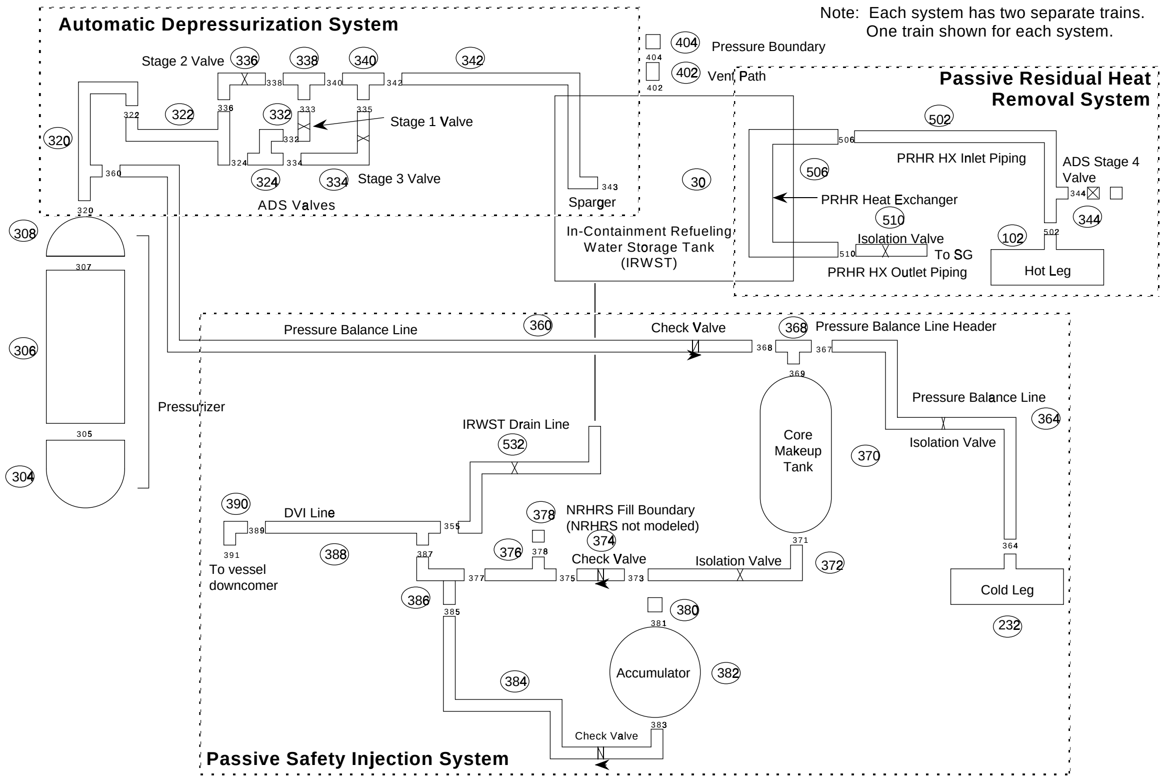

Figure 5: AP600 safety systems modeling noding diagram.

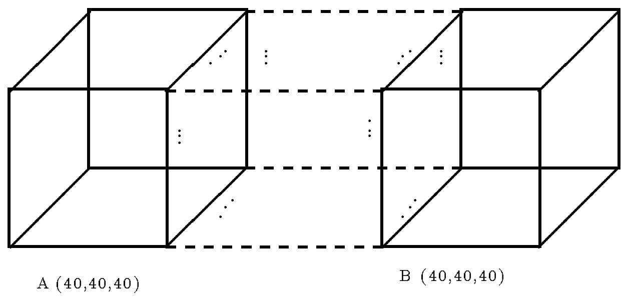

Figure 6: Simple Coupling Example. Dashed lines represent couplings; Solid lines represent grid boundaries.

index is 1. To express this in a COUPLE statement we would declare

$$\text{COUPLE } A[(1, 1:40), (2, 40:40), (3, 1:40)] \text{ WITH } B[(1, 1:40), (2, 1:1), (3, 1:40)]$$

where each entry of the form $(d, s : f)$ is specifying the range of elements (s to f) to be coupled in dimension d. We need to specify the dimensions to be able to express couplings between different dimensions of the decompositions. To illustrate this point, our second example involves a computation associated with the trio of coupled two dimensional grids illustrated in Figure 7. The couplings in this problem would be declared as follows.

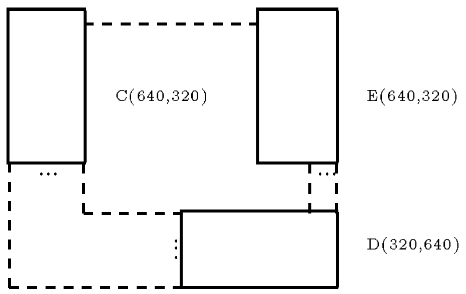

Figure 7: Cross Dimensional Coupling Example.

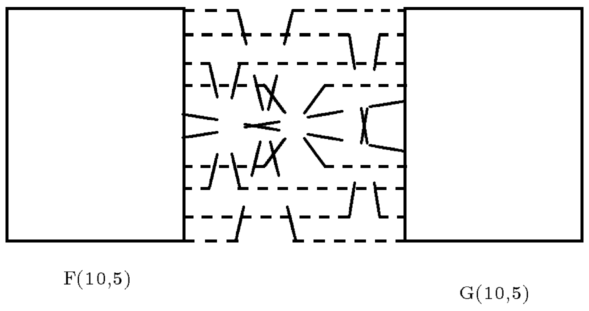

Figure 8: Negative Stride Coupling Example.

$$\text{COUPLE } C[(1,640:640),(2,1:320)] \text{ WITH } D[(2,1:1),(1,1:320)]$$
$$\text{COUPLE } E[(1,640:640),(2,310:320)] \text{ WITH } D[(1,1:1),(2,630:640)]$$
$$\text{COUPLE } C[(1,10:10),(2,320:320)] \text{ WITH } E[(1,10:10),(2,1:1)]$$

This simple form of the COUPLE statement does not support negative or non-unit strides. With this restrictive form it would take ten COUPLE statements to express the coupling in Figure 8. Hence with the addition of an optional stride we can declare:

$$\text{COUPLE } F[(1,1:10),(2,5:5)] \text{ WITH } G[(1,10:1:-1),(2,1:1)]$$

The form for a fairly general coupling specifications would therefore be:

$$\begin{array}{rl} \text{COUPLE} & A[(dim_{A1}, start_{A1}:end_{A1}:stride_{A1}),(dim_{A2}, start_{A2}:end_{A2}:stride_{A2}),...] \\ \text{WITH} & B[(dim_{B1}, start_{B1}:end_{B1}:stride_{B1}),(dim_{B2}, start_{B2}:end_{B2}:stride_{B2}),...] \end{array}$$

This couples decomposition A's elements, in dimension dim_{A1} of A, in the range $start_{A1}$ to end_{A1} starting in position $start_{A1}$ with stride $stride_{A1}$ to B's elements, in dimension dim_{B1} of B, in the range $start_{B1}$ to end_{B1} starting in position $start_{B1}$ with stride $stride_{B1}$. The coupling for the subscript 2 elements is similar. Specifying the dimensions for each range of elements allows fairly arbitrary coupling between different dimensions. The stride is optional and has a default value of 1.

This illustrates that the coupling information that must be extracted from the input file for these applications includes: the meshes being coupled, the dimensions being coupled, the range of elements being coupled and the stride of the coupling.

2.2 Programming Style Recommendations

Here we attempt to outline a programming style that is natural to composite grid applications, such as nuclear reactor simulation, but does not inhibit dependence analysis. The style should also make development and support more efficient in terms of programmer effort [12]. This includes the amount of time that a new researcher must study the code before being able to make contributions to the application development [6]. Further, programs written in this style should achieve good performance when compiled by a good compiler. We will describe this style in the context of HPF [7].

Two of the most important language features, for the current discussion, are dynamic memory allocation and user-defined data types. The user-defined data types allow all of the arrays and scalars for a given mesh to be grouped into one data structure with the actual storage for the arrays being allocatable at runtime according to the input. Further, an allocatable array of such structures can be used to represent all of the meshes of a given type, i.e., all meshes that have the same dimensionality and the same basic computation. For example, in a reactor simulation code, there could be an allocatable array of structures for all of the pumps, another for all of the pipes, another for all of the vessels, et cetera. This implies that the program has one allocatable array of structures for each type of mesh. Alternatively, the user-defined data type may include a type specification and have only one array of structures for each dimensionality (this is the version we illustrate). The basic ideas for data structure creation are:

- arrays for each component are dynamically allocated;
- all arrays and scalars for each component type are grouped into a user-defined data structure;
- arrays of user-defined data structures are allocated dynamically;
- there is an array of user-defined data structures for each type of component.

Associated with each type of mesh there is also a compute routine that is called with each structure on every time step. In addition there must be a routine for internal boundary data exchange for each pair of mesh types. These routines need to be written using a structured programming style, e.g., via the use of select case, not computed gotos. Further, the recommended organization is shown in Figure 9. Given a program written in this

```
      module composite
C     -- user defined data types go here
C     -- global declarations go here
      contains
      subroutine main
C     -- non-runtime-constant input goes here
C     -- initialization goes here
C     -- all computation goes here
C     -- output goes here
      contains
C     all subroutines except runtime constant input and storage allocation go here
      end subroutine main
C     runtime constant input routines go here
C     storage allocation routines go here
      end module composite

      program icrm
      use module composite
C     -- calls to runtime constant input routines go here
C     -- calls to allocation routines go here
      call main()
      end program icrm
```

Figure 9: Recommended organization for composite grid HPF programs.

style and an understanding of how the input file relates to the allocation of these structures, the analysis necessary for execution of the automatic distribution algorithm is similar to the interprocedural analysis performed by Rice University's Fortran D compiler [4].

3 Automatic Distribution Algorithm

Minimally, for the algorithm we are about to describe, we need, in addition to the information in coupling specifications, measures of:

- the amount of computation per element in each mesh,
- the amount of communication in each dimension of each large mesh, and
- the amount of communication between each coupled pair of elements in each coupling between meshes.

The coupling specifications are used to determine which meshes must communicate. The amount of computation for each element of each mesh is used to attempt to balance the computational load. The amount of communication in each dimension of each large mesh is used to attempt to minimize the cost of communication for distribution of the large mesh. The amount of communication for each coupling is used to prioritize the order in which coupled meshes get mapped and to determine the placement of coupled meshes. In nuclear reactor simulations these measures may vary by component type, but the nature of the computation carried out for each mesh of the same type is similar. The computation can also vary by cell according to the material in the cell and the phase of the material.

For this work, we restrict ourselves to consideration of composite grid applications in which

- all of the meshes are either large enough to be efficiently distributed over all processors or small enough to be assigned to a single processor and
- computations inside of each mesh are regular (this implies regular internal communication for each mesh for each distribution generated by our algorithms).

Further, we are currently targeting a torus-based communications topology. This seems reasonable as many available machines either are tori or can have them efficiently embedded in the machine topology.

We are, at last, ready to present our automatic distribution algorithms. We will begin with descriptions of both of the random distribution algorithms and the topology-based small mesh distribution algorithm. Then we will illustrate the suggested user programming style and precompiler transformations with HPF excerpts. Finally, this section contains a discussion of the limitations and advantages of the topology-based small mesh distribution approach.

For efficient parallelization of composite grid applications, we would like to distribute each mesh according to its dimensionality [9]. Hence, we will distribute all of the large meshes over all of the processors in the same dimensionality [8]. During the mapping of the large meshes, a processor configuration is selected for distribution of the entire problem. When there are no large meshes in the problem we select the m dimensional processor topology closest to being "square", where m is the dimensionality of the largest dimensionality mesh of the specified size on the host architecture.

3.1 Random Algorithm

For each mesh a pseudorandom number is used to select the processor which will own the mesh. Note that we place no restriction on assignment of meshes to processors. This approach has been advocated by applications developers. This algorithm takes $O(M)$ time to generate a distribution specification, where M is the number of small meshes.

3.2 Load-Balanced Random Algorithm

With this algorithm the meshes are randomly distributed, as above, but they are distributed in highest computation cost first order. Further there is an upper bound on the amount of computation that may be allocated to a processor which is overridden only after a number of failures to allocate. The number of failures is set to be on the order of the number of processors. Therefore, to map the largest unmapped mesh, a processor is randomly selected. If the processor can accommodate the mesh's computation then map the mesh to the processor, otherwise try again up to some set number of times. If no processor with enough room is found, then assign it to a processor because, even if the load balance is poor, the problem must be solved. This approach produces better worst-case load balance than the random algorithm but is more expensive. This algorithm takes $O(P\ M\ log(M))$ time due to the sorting of the meshes by computational cost and the possible number of failures, where M is the number of small meshes and P is the number of processors.

3.3 Topology-based Small Mesh Algorithm

Here we present a brief description of our topology-based small mesh distribution algorithm and the associated data structures, and then outline the steps in the algorithm in Figures 10 and 11.

It is assumed that before this algorithm is started any large meshes have been mapped. As a result of this mapping a processor configuration is chosen. Further all communications involving large size meshes are scheduled. If there are no large meshes, the largest dimensionality nearly square processor configuration possible is chosen and the highest computation cost small mesh is mapped to an arbitrary processor. All communications to this mapped small mesh are scheduled.

The basic idea used in this algorithm is that we want to map meshes which communicate to the same processor or at least as near as possible. If we simply mapped to the processor with the greatest coupling cost we would end up with very few processors with small meshes allocated to them. The number of processors used in that case would be bounded by the number of couplings between large and small meshes (one in the case of a single premapped small mesh). Therefore we use a load-balance measure to determine when a processor has too much computation already allocated to accommodate the work associated with an unmapped mesh. When this happens, we try to allocate it to a neighbor processor. If all of the neighbors are too full we find the closest processor with no work allocated. If this also fails we save the mesh as a "fill" element to be used after all other meshes are mapped. These "fill" elements are then used in a final load balancing step by assigning them to under-utilized processors.

We begin small mesh distribution by determining the amount of computation, for the small meshes only, that should be assigned to each processor for perfect small mesh load balance. We will statically distribute small meshes by trying to obtain load balance within $\pm\epsilon$ of perfect balance. If the large meshes are not perfectly load balanced, then we could add their computation to this consideration to improve the overall load balance. We are not currently doing this but will in future work. Note that this algorithm could also be used at runtime for dynamic load balancing if computation/communication statistics are collected in the program and the new alignment/distribution specifications are used for data redistribution. This may be useful for problems where the amount of computation per cell changes dramatically over time.

Two types of heaps are used in this mapping algorithm. The *mesh heap* contains all of the unmapped meshes with scheduled communication. Communication is scheduled, between a mapped mesh, M, and each coupled unmapped mesh, N, by adding the coupling communication cost to N's communication for the processor to which M is mapped (see Figure 10). This introduces the second type of heap. A *processor heap* is associated with each mesh in the *mesh heap*. Each entry in a *processor heap* gives the current coupling cost of the associated mesh to that specific processor. From this two level structure we get the mesh with the maximum communication scheduled to a single processor (from the *mesh heap*) and the associated processor (from the mesh's *processor heap*). In the final cleanup stage of the algorithm two heaps are also used, one being a maximum heap of meshes sorted according to the amount of computation in the mesh and the other being a minimum heap of processors sorted according to the amount of computation assigned to the processor.

For each mapped large mesh, all communications to small meshes are scheduled initially. Alternatively the small mesh with the most computation may be mapped to a processor and all communication to other small

```
add M's computation to P's computation
delete M's processor heap
For each unscheduled coupling involving the mapped mesh, M, and an unmapped mesh, N, with weight W
    If N is not in the mesh heap
        create a processor heap with one entry for P having weight W
        add N to the mesh heap with weight W
    else
        if there is not entry for P in N's processor heap
            create an entry for P with weight W and add it to N's processor heap
            add W to N's weight in the mesh heap and bubble N's entry up
        else
            add W to P's weight in N's processor heap and bubble P's entry up
            add W to N's weight in the mesh heap and bubble N's entry up
        endif
    endif
```

Figure 10: Pseudocode for mapping mesh M onto processor P

meshes are scheduled. Now we are ready for the details of the topology-based algorithm. In Figure 10, an outline of the steps needed to map or assign a mesh to a processor is given. In Figure 11, an outline of the steps in the topology-based small mesh distribution algorithm is given. One point we need to mention is that from studying the types of coupling communication that occur in our test problems we learned that there are often many couplings with the same communication cost. This led us to explore further ordering options in the *mesh heap*. In the end we used a combination of maximum coupling cost to a processor and the computation cost associated with the mesh to select the next mesh to map. The coupling cost is still the primary selection criteria with the higher computation cost only used to make the selection when communication cost ties occur.

The runtime of this algorithm is

$$O(M\ log(M)\ C\ log(C)\ P\ log(P))$$

where M is the number of small meshes, C is the maximum number of couplings involving a single mesh, and P is the number of processors. The $P\ log(P)$ term comes from sometimes finding the nearest empty processor. To give an idea of how expensive this algorithm is on a real problem, for the AP600 data set there are 220 meshes to be distributed, 366 couplings, and at most 64 processors.

3.4 Transformation of High Performance Fortran Programs

Here we begin by showing an example of the use of the coding style that we recommend. Then we will discuss the interprocedural transformations that would be performed by a preprocessor. The coding of the routines sketched in this example depends on the application programmer having developed a parallel algorithm for the physical phenomenon simulation. One note about the large mesh update routines: the loops over large meshes are sequential, but the loops inside of the large mesh routines are *forall* loops. Also note that we assume that all of the couplings are independent. If there are overlapping couplings then the loop over couplings can not be a *forall*. Finally, we present some of the transformed HPF routines[2]. For an example of the coding needed for the large multi-dimensional meshes that are distributed across all processors, see [8]. This code shows how the automatically generated distribution to a three-dimensional mesh of processors would be used in an HPF program.

[2]In all of the HPF code segments, every line that is continued should have an & in column 73.

```
max distance = one half of the diameter of selected processor configuration
search distance = 1
While there are unconsidered meshes
    proc = null
    delete the maximum entry, M, from the mesh heap
    best proc = processor in the maximum entry of M's processor heap
    While ((proc is null) && (processor heap is not empty))
        delete the maximum entry, P, from M's processor heap
        If P has room for M
            proc = P
        else if there is any neighbor of P, P′, within search distance with room for M
            proc = P′
        endwhile
    if (proc is null) reset M's processor heap
    While ((proc is null) && (processor heap is not empty))
        delete the maximum entry, P, from M's processor heap
        if there is any neighbor of P, P′, within max distance with room for M
            proc = P′
        endwhile
    if ((proc is null) && (there is an empty processor))
        proc = empty processor nearest best proc
    mark M considered
    if (proc is null)
        save M for filler
    else
        map M onto P (see Figure 10)
    endwhile
create a minimum (by computation allocated) heap of processors
create a maximum (by computation to perform) heap of filler meshes
while there are filler meshes in the max heap
    delete maximum mesh from max heap
    delete minimum proc from min heap
    map M onto P
    insert P with new computation cost into the minimum processor heap
    endwhile
```

Figure 11: Topology-based Distribution Algorithm for Small Meshes

Figure 12 shows the use of a user-defined data type, *mesh_1d*, including the allocation of variable size arrays. Note that in the user program all of the information for a single grid is passed as one parameter into all of the subroutines.

Figure 13 shows the declaration of user defined data types in module *mesh_module*. The only subroutine declarations that are allowed in the *mesh_module* are for *main_routine* and the routines to read the runtime constant input. Only these routines may be called by *mesh_module*. Before entry into *main_routine* all runtime constants must have been read in. All of the actual work including input of non-runtime constant input and initialization occurs in *main_routine* as is shown in Figure 14. All meshes are updated according to their type for a given timestep before coupling communication is performed.

Figure 15 shows how coupling ranges are read in according to the dimensionality of the meshes involved. Further, coupling update routines are based on the dimensionality of both of the meshes involved in the coupling.

```
      subroutine read_meshes_1d(num_before)
      read(*,*) Num_1d_meshes
      allocate(Meshes_1d(Num_1d_meshes))
      do i = 1, Num_1d_meshes
         dims(i+num_before) = 1
         read(*,*) Meshes_1d(i)%size, Meshes_1d(i)%type
      end do
      end subroutine read_meshes_1d

      subroutine allocate_1d(Mesh_info)
      type (mesh_1d) Mesh_info
      real, pointer :: all_array(:)
      allocate(all_array(i_size))
      Mesh_info%p => all_array
      nullify(all_array)
C     ... similar allocation for q, u, v, zm, x, y, z
      end subroutine allocate_1d
```

```
      subroutine allocate_all_meshes()
      do i = 1, Num_1d_meshes
         call allocate_1d(Meshes_1d(i))
      end do
      do i = 1, Num_3d_meshes
         call allocate_3d(Meshes_3d(i))
      end do
      end subroutine allocate_all_meshes

      subroutine update_mesh_pipe(Mesh_info)
      type (mesh_1d) Mesh_info
      use physics
C     ... updates for arrays associated with current mesh
      end subroutine update_mesh_pipe
```

Figure 12: Original HPF input, allocation, and compute routines for 1-d meshes.

```
      module mesh_module
      type mesh_1d
        integer  size, type
        real,pointer,dimension(:)::p,q,u,v,zm,x,y,z
      end type mesh_1d
      type mesh_3d
        integer  size(3)
        real,pointer,dimension(:,:,:)::p,q,u,v,zm,x,y,z
      end type mesh_3d
      type coupling
        integer id_A, id_B
        integer lo_A(3), hi_A(3), lo_B(3), hi_B(3)
      end type coupling
      integer Num_Steps, Num_Meshes
      integer allocatable, dimension(:)::dims
      integer Num_1d_meshes, Num_3d_meshes, Num_couplings
      type (coupling),allocatable,dimension(:)::Couplings
```

```
C     module mesh_module continued

      type (mesh_1d),allocatable,dimension(:)::Meshes_1d
      type (mesh_3d),allocatable,dimension(:)::Meshes_3d
      contains
C     subroutine main_routine goes here
C     input routines go here
      end module mesh_module

      program big_mesh
      use mesh_module
      read(*,*) Num_Steps, Num_Meshes
      allocate(dims(Num_Meshes))
      call read_meshes_3d()
      call read_meshes_1d(Num_3d_meshes)
      call read_couplings()
      call main_routine()
      end program big_mesh
```

Figure 13: HPF module example.

In this example, the order of meshes in a coupling specification is determined by the size of the meshes. This ordering eliminates the need for one case in the *main_routine* of Figure 14. Note that in the transformed *main_routine* we used "..." to shorten the parameter lists for readability.

Now that we have seen examples of the type of code that the user writes, let's consider the transformations that are needed to allow our automatically determined distributions to work with standard HPF compilers. Figure 16 shows how the distribution specifications for each mesh are read from a secondary input file. Further, the routine that allocates all of the meshes, *allocate_all_meshes*, must now have a case statement to select the appropriate large mesh allocation routine according to the mesh's distribution. All of the distributed meshes must be passed explicitly as parameters and not as elements of the user-defined data type. This is because HPF does not allow elements of user-defined data types to appear in align/distribute statements. A sample update routine for small meshes, e.g., *update_mesh_pipe* in Figure 16, shows the specification of the processor that the mesh is assigned to.

```
      subroutine main_routine()
      forall (i=1, Num_1d_meshes)
         call initial_1d(Meshes_1d(i),dthlf,dt)
      end forall
      do i=1, Num_3d_meshes
         call initial_3d_ijk(Meshes_3d(i),dthlf,dt)
      end do

      call system_clock(is_count,is_count_rate,
     &                  is_count_max)
      do i=1, Num_Steps
         forall (i=1, Num_1d_meshes)
            select case (Meshes_1d(i)%type)
            case (pipe)
               call update_mesh_pipe(Meshes_1d(i),dthlf,dt)
C           ... similar case for each mesh type
            end select
         end forall
         do i=1, Num_3d_meshes
            call update_mesh_3d_ijk(Meshes_3d(i),dthlf,dt)
         end do

         forall (i=1, Num_couplings)
C           Note: larger mesh is in B
            if (dims(id_B) .eq. 1) then
               id_A = Couplings(i)%id_A - Num_3d_meshes
               id_B = Couplings(i)%id_B - Num_3d_meshes
               call update_couplings_1d(Meshes_1d(id_A),
     &Couplings(i)%lo_A,Couplings(i)%hi_A,Meshes_1d(id_B),
     &Couplings(i)%lo_B,Couplings(i)%hi_B,dthlf,dt)
```

```
C     subroutine main_routine continued

            else if (dims(id_A) .eq. 3) then
               id_A = Couplings(i)%id_A
               id_B = Couplings(i)%id_B
               call update_couplings_3d(
     &Meshes_3d(id_A),Couplings(i)%lo_A(1),
     &Couplings(i)%lo_A(2),Couplings(i)%lo_A(3),
     &Couplings(i)%hi_A(1),Couplings(i)%hi_A(2),
     &Couplings(i)%hi_A(3),Meshes_3d(id_B),
     &Couplings(i)%lo_B(1),Couplings(i)%lo_B(2),
     &Couplings(i)%lo_B(3),Couplings(i)%hi_B(1),
     &Couplings(i)%hi_B(2),Couplings(i)%hi_B(3),dthlf,dt)
            else
               id_A = Couplings(i)%id_A - Num_3d_meshes
               id_B = Couplings(i)%id_B
               call update_couplings_1d_3d(
     &Meshes_1d(id_A),Couplings_1d_3d(i)%lo_A,
     &Couplings_1d_3d(i)%hi_A,Meshes_3d(id_B),
     &Couplings_1d_3d(i)%lo_B(1),Couplings_1d_3d(i)%lo_B(2),
     &Couplings_1d_3d(i)%lo_B(3),Couplings_1d_3d(i)%hi_B(1),
     &Couplings_1d_3d(i)%hi_B(2),Couplings_1d_3d(i)%hi_B(3),
     &dthlf,dt)
            end if
         end forall
      end do
      call system_clock(ie_count,ie_count_rate,ie_count_max)
C     ... print results, etc.
      contains
C     all subroutines except input routines go here
      end subroutine main_routine
```

Figure 14: Original HPF main_routine example.

A note about the structure of the user-defined data structures, e.g., *mesh_1d*, and the associated storage allocation: Since HPF does not currently allow elements of user-defined data structures to appear in distribution statements, a pointer is used to allocate and distribute the arrays and then the structure's array is set to point to the correct memory. In order to specify the incoming distributions for the user-defined data structure arrays in subroutines, the arrays in the structures are passed individually rather than just passing the structure. Finally, the parameters to the HPF specifications must be constant upon entry to subprogram level. This is the reason we required the routine *main_routine* to do everything except for the input which is done in the main program before calling *main_routine*. This structure allows us to read in the number of processors to use and the processor assignment for each mesh. Note that the input of couplings is not modified in the transformation process.

In *mesh_module*, (see Figure 17) the processor that a small mesh is assigned to is added to the user-defined data type *mesh_1d* and the distribution order, the decomposition size, and offset is added to the user-defined data type *mesh_3d*. Further the number of processors to use in each dimension is declared and input in the *mesh_module* so that they are constant upon entry to *main_routine*. This allows the processors declaration that appears in Figure 18. Further, we add the code to select the appropriate clone for each routine that is called with meshes having different distributions.

Since the end result of our work is to be a standard HPF program and HPF does not require the compiler to perform interprocedural analysis to obtain distribution specifications, we must perform cloning of all routines where meshes with different types of distribution can be passed to the same parameter. The types of distribution for these programs are mappings to a single processor or distributions across all processors with any dimension permutation. For example, the coupling routine, *update_couplings_1d_3d*, shown in Figure 15, is cloned with a

```
      subroutine read_couplings()
      read(*,*) Num_couplings
      allocate(Couplings_(Num_couplings))
      do i = 1, Num_couplings
         read(*,*) Couplings(i)%id_A,Couplings(i)%id_B
         if (dims(Couplings(i)%id_A).eq.1) then
            read(*,*) Couplings(i)%lo_A(1),
     &                Couplings(i)%hi_A(1)
         else
            read(*,*) Couplings(i)%lo_A,
     &                Couplings(i)%hi_A
         end if
         if (dims(Couplings(i)%id_B).eq.1) then
            read(*,*) Couplings(i)%lo_B(1),
     &                Couplings(i)%hi_B(1)
         else
           read(*,*) Couplings(i)%lo_B,
     &                Couplings(i)%hi_B
         end if
      end do
      end subroutine read_couplings

      subroutine update_couplings_1d(Mesh_A,i_A_lo,
     &i_A_hi,Mesh_B,B_x,B_y,B_z,i_B_lo,i_B_hi,dthlf,dt)
      type (mesh_1d) Mesh_A, Mesh_B
      real dt,dthlf
      p_tmp(1:i_A_hi-i_A_lo+1) = Mesh_A%p(i_A_lo:i_A_hi)
C     ... save all the temporaries
      Mesh_A%q(i_A_lo:i_A_hi) = Mesh_A%q(i_A_lo:i_A_hi)
     &     * frac + Mesh_B%q(i_B_lo:i_B_hi) * (1.0-frac)
C     subroutine update_couplings_1d continued

C     ... update all of the ``A'' variables
      Mesh_B%q(i_B_lo:i_B_hi) = Mesh_B%q(i_B_lo:i_B_hi)
     &       * frac + q_tmp * (1.0 - frac)
C     ... update all of the ``B'' variables
      end subroutine update_couplings_1d

      subroutine update_couplings_1d_3d(Mesh_A,
     &  i_A_lo,i_A_hi,Mesh_B,i_B_lo,j_B_lo,k_B_lo,
     &  i_B_hi,j_B_hi,k_B_hi,dthlf,dt)
      type (mesh_1d) Mesh_A
      type (mesh_3d) Mesh_B
      real dt,dthlf
      real p_tmp(i_A_hi-i_A_lo+1),v_tmp(i_A_hi-i_A_lo+1)
      real q_tmp(i_A_hi-i_A_lo+1),u_tmp(i_A_hi-i_A_lo+1)
      real zm_tmp(i_A_hi-i_A_lo+1),x_tmp(i_A_hi-i_A_lo+1)
      real y_tmp(i_A_hi-i_A_lo+1),z_tmp(i_A_hi-i_A_lo+1)
      p_tmp(1:i_A_hi-i_A_lo+1) = Mesh_A%p(i_A_lo:i_A_hi)
C     ... save all the temporaries
      Mesh_A%q(i_A_lo:i_A_hi) = Mesh_A%q(i_A_lo:i_A_hi) * frac
     &    + Mesh_B%q(i_B_lo:i_B_hi,j_B_lo:j_B_hi,k_B_lo:k_B_hi)
     &    * (1.0-frac)
C     ... update all of the ``A'' variables
      Mesh_B%q(i_B_lo:i_B_hi,j_B_lo:j_B_hi,k_B_lo:k_B_hi) =
     &    Mesh_B%q(i_B_lo:i_B_hi,j_B_lo:j_B_hi,k_B_lo:k_B_hi)
     &    * frac
     &     + q_tmp * (1.0 - frac)
C     ... update all of the ``B'' variables
      end subroutine update_couplings_1d_3d
```

Figure 15: Original HPF example input and update routines for 1-d couplings.

```
      subroutine read_meshes_1d(num_before)
      read(*,*) Num_1d_meshes
      allocate(Meshes_1d(Num_1d_meshes))
      do i = 1, Num_1d_meshes
         dims(i+num_before) = 1
         read(*,*) Meshes_1d(i)%size, Meshes_1d(i)%type
         read(8,*) Meshes_1d(i)%proc
      end do
      end subroutine read_meshes_1d

      subroutine allocate_1d(Mesh_info)
      type (mesh_1d) Mesh_info
!HPF$ template decomp(Num_Proc_i,Num_Proc_j,Num_Proc_k)
!HPF$ align all_array(*) with decomp(Mesh_info%proc(1),
!HPF$&    Mesh_info%proc(2),Mesh_info%proc(3))
!HPF$ distribute(block,block,block)onto procs::decomp
      real, pointer :: all_array(:)
      allocate(all_array(i_size))
      Mesh_info%p => all_array
      nullify(all_array)
C     ... similar allocation for q, u, v, zm, x, y, z
      end subroutine allocate_1d
      subroutine allocate_all_meshes()
      do i = 1, Num_1d_meshes
         call allocate_1d(Meshes_1d(i))
      end do
      do i = 1, Num_3d_meshes
         select case (Meshes_3d(i)%dist_order)
         case (123)
            call allocate_3d_ijk(Meshes_3d(i))
C        ... similar cases for other distribution orders
         end select
      end do
      end subroutine allocate_all_meshes

      subroutine update_mesh_pipe(i_size,proc_i,
     &      proc_j,proc_k,p,q,u,v,zm,x,y,z,dthlf,dt)
      use physics
!HPF$ template decomp(Num_Proc_i,Num_Proc_j,Num_Proc_k)
!HPF$ align (*) with decomp(proc_i,proc_j,proc_k)::
!HPF$&          p,q,u,v,zm,x,y,z
!HPF$ distribute (block,block,block) onto procs :: decomp
      real p(:),u(:),v(:),q(:),zm(:),x(:),y(:),z(:),dthlf,dt
C     ... updates for arrays associated with current mesh
      end subroutine update_mesh_pipe
```

Figure 16: Transformed HPF example input, allocation, and compute routines for 1-d meshes.

```
module mesh_module                                 C
type mesh_1d
  integer  size, type, proc(3)
  real, pointer, dimension(:)::p,q,u,v,zm,x,y,z
end type mesh_1d
type mesh_3d
  integer  size(3)                                 C
  integer decomp_size(3), offset(3), dist_order    C
  real, pointer, dimension(:,:,:)::p,q,u,v,zm,x,y,z
end type mesh_3d
type coupling
  integer id_A, id_B
  integer lo_A(3), hi_A(3), lo_B(3), hi_B(3)
end type coupling
integer Num_Steps, Num_Meshes
integer allocatable, dimension(:)::dims
integer Num_1d_meshes, Num_3d_meshes, Num_couplings
integer Num_Proc_i, Num_Proc_j, Num_Proc_k
```

```
module mesh_module continued

type (mesh_1d),allocatable,dimension(:)::Meshes_1d
type (mesh_3d),allocatable,dimension(:)::Meshes_3d
type (coupling),allocatable,dimension(:)::Couplings
contains
subroutine main_routine goes here
input routines go here
end module mesh_module

program big_mesh
use mesh_module
read(*,*) Num_Steps, Num_Meshes
open(unit=8,file='dist.large')
read(8,*) Num_Proc_i, Num_Proc_j, Num_Proc_k
allocate(dims(Num_Meshes))
call read_meshes_3d()
call read_meshes_1d(Num_3d_meshes)
call read_couplings()
call main_routine()
end program big_mesh
```

Figure 17: Transformed HPF module example.

version for each dimension permutation. One cloned and transformed coupling update routine's name ends with "ijk" as shown in Figure 20 and has similar clones with names ending in "jik", "kij", "ikj", "jki", and "kji". The version of this routine shown in Figure 15 is written by the user, the version in Figure 20 and all of the other clones would be generated automatically from the user's version. This cloning implies that separate compilation may provide a significant advantage for these problems. A case structure is used in the main routine to ensure that the correct version of the cloned routine is called (see Figure 18). Rice University's Fortran D compiler uses cloning in a similar manner to optimize communication.

Since the end product of this is a standard HPF program, there are no further compiler enhancements necessary for correct code generation (for either a MIMD or SIMD machine). However, with all of the clones automatically generated, the HPF compiler can apply all of the communication optimizations developed for regular mesh problems with regular distributions on these programs for partially regular problems. One area where HPF compilers could further benefit from the use of coupling information is in coupling communication generation. The coupling specifications could be used as an upper bound on the communication that must be performed for coupling. If this is done then the coupling specification is elevated to an error status similar to that of a forall loop; if the user specification is incorrect then the program results will be incorrect. This is very reasonable as the user's results will probably be incorrect anyway since the simulation program uses the coupling information explicitly.

3.5 Limitations vs. Advantages

This approach to ICRM parallelization is limited to those problems in which it makes sense to distribute every mesh either over all processors or place it on just one. This limits the use of the approach for some problems. For example, this approach does not work when there are too many elements in some meshes to allow them to fit on a single processor, but too few to distribute over all processors. We are currently exploring other approaches for automatic distribution of such problems.

The most important advantage to this approach is that the parallelization burden to the developer of such codes as TRAC is greatly reduced while the applicable communication optimization technology is increased (regular distribution optimizations can be applied). In particular, the developer does not have to explicitly parallelize

```
      subroutine main_routine()
!HPF$ Processors procs(Num_Proc_i,Num_Proc_j,Num_Proc_k)
      forall (i=1, Num_1d_meshes)
         call initial_1d(Meshes_1d(i)%size,
     &Meshes_1d(i)%proc(1),Meshes_1d(i)%proc(2),
     &Meshes_1d(i)%proc(3),Meshes_1d(i)%p,...,
     &Meshes_1d(i)%z,dthlf,dt)
      end forall
      do i=1, Num_3d_meshes
         select case (Meshes_3d(i)%dist_order)
         case (123)
            call initial_3d_ijk(
     &Meshes_3d(i)%decomp_size(1),...,
     &Meshes_3d(i)%decomp_size(3),Meshes_3d(i)%offset(1),
     &...,,Meshes_3d(i)%offset(3),Meshes_3d(i)%p,
     &...,Meshes_3d(i)%z,dthlf,dt)
C        ... similar case for each distribution order
      end do
      call system_clock(is_count,is_count_rate,
     &                  is_count_max)
      do i=1, Num_Steps
         forall (i=1, Num_1d_meshes)
            select case (Meshes_1d(i)%type)
            case (pipe)
               call update_mesh_pipe(Meshes_1d(i)%size,
     &Meshes_1d(i)%proc(1),...,Meshes_1d(i)%proc(3),
     &Meshes_1d(i)%p,...,Meshes_1d(i)%z,dthlf,dt)
C           similar case for other small mesh types
            end select
         end forall
         do i=1, Num_3d_meshes
            select case (Meshes_3d(i)%dist_order)
            case (123)
            call update_mesh_3d_ijk(
     &Meshes_3d(i)%decomp_size(1),...,
     &Meshes_3d(i)%decomp_size(3),Meshes_3d(i)%offset(1),
     &...,Meshes_3d(i)%offset(3),Meshes_3d(i)%p,...,
     &Meshes_3d(i)%z,dthlf,dt)
C          ... similar case for each distribution order
         end do
         forall (i=1, Num_couplings)
C           Note: larger mesh is in B
            if (dims(id_B) .eq. 1) then
               id_A = Couplings(i)%id_A - Num_3d_meshes
               id_B = Couplings(i)%id_B - Num_3d_meshes
               call update_couplings_1d(
     &Meshes_1d(id_A)%size,Meshes_1d(id_A)%proc(1),...,
     &Meshes_1d(id_A)%proc(3),Meshes_1d(id_A)%p,...,
     &Meshes_1d(id_A)%z,Couplings(i)%lo_A,
     &Couplings(i)%hi_A,Meshes_1d(id_B)%size,
     &Meshes_1d(id_B)%proc(1),...,Meshes_1d(id_B)%proc(3),
     &Meshes_1d(id_B)%p,...,Meshes_1d(id_B)%z,
     &Couplings(i)%lo_B,Couplings(i)%hi_B,dthlf,dt)
```

```
C    subroutine main_routine continued

            else if (dims(id_A) .eq. 3) then
               id_A = Couplings(i)%id_A
               id_B = Couplings(i)%id_B
               if ((Meshes_3d(id_A)%dist_order .eq. 123).and.
                   (Meshes_3d(id_B)%dist_order .eq. 123))
                  call update_couplings_3d_ijk_ijk(
     &Meshes_3d(id_A)%decomp_size(1),...,
     &Meshes_3d(id_A)%decomp_size(3),Meshes_3d(id_A)%offset(1),
     &...,Meshes_3d(id_A)%offset(3),Meshes_3d(id_A)%p,...,
     &Meshes_3d(id_A)%z,Couplings(i)%lo_A(1),...,
     &Couplings(i)%lo_A(3),Couplings(i)%hi_A(1),...,
     &Couplings(i)%hi_A(3),Meshes_3d(id_B)%decomp_size(1),...,
     &Meshes_3d(id_B)%decomp_size(3),Meshes_3d(id_B)%offset(1),
     &...,Meshes_3d(id_B)%offset(3),Meshes_3d(id_B)%p,...,
     &Meshes_3d(id_B)%z,Couplings(i)%lo_B(1),...,
     &Couplings(i)%lo_B(3),Couplings(i)%hi_B(1),...,
     &Couplings(i)%hi_B(3),dthlf,dt)
C              ... similar case for each pair of orders
            else
               id_A = Couplings(i)%id_A - Num_3d_meshes
               id_B = Couplings(i)%id_B
               select case (Meshes_3d(id_B)%dist_order)
               case (123)
                  call update_couplings_1d_3d_ijk(
     &Meshes_1d(id_A)%size(1),Meshes_1d(id_A)%proc(1),...,
     &Meshes_1d(id_A)%proc(3),Meshes_1d(id_A)%p,...,
     &Meshes_1d(id_A)%z,Couplings_1d_3d(i)%lo_A,
     &Couplings_1d_3d(i)%hi_A,Meshes_3d(id_B)%size(1),...,
     &Meshes_3d(id_B)%size(3),Meshes_3d(id_B)%align_offset(1),
     &...,Meshes_3d(id_B)%align_offset(3),
     &Meshes_3d(id_B)%p,...,Meshes_3d(id_B)%z,
     &Couplings_1d_3d(i)%lo_B(1),...,Couplings_1d_3d(i)%lo_B(3),
     &Couplings_1d_3d(i)%hi_B(1),...,Couplings_1d_3d(i)%hi_B(3),
     &dthlf,dt)
C              ... similar case for each distribution order
               end select
            end if
         end forall
      end do
      call system_clock(ie_count,ie_count_rate,ie_count_max)
C     ... print results, etc.
      contains
C     all subroutines except input routines go here
      end subroutine main_routine
```

Figure 18: Transformed HPF main_routine example.

```
      subroutine update_couplings_1d(i_A_size,
     &i_A_proc,j_A_proc,k_A_proc,A_p,A_q,A_u,A_v,
     &A_zm,A_x,A_y,A_z,i_A_lo,i_A_hi,i_B_size,
     &i_B_proc,j_B_proc,k_B_proc,B_p,B_q,B_u,B_v,B_zm,
     &B_x,B_y,B_z,i_B_lo,i_B_hi,dthlf,dt)
!HPF$ template decomp(Num_Proc_i,Num_Proc_j,Num_Proc_k)
!HPF$ align (*) with *decomp(i_A_proc,j_A_proc,k_A_proc)
!HPF$&              ::A_p,A_q,A_u,A_v,A_zm,A_x,A_y,A_z
!HPF$ align (*) with *decomp(i_B_proc,j_B_proc,k_B_proc)
!HPF$&              ::B_p,B_q,B_u,B_v,B_zm,B_x,B_y,B_z
!HPF$ distribute (block,block,block) onto procs::decomp
      real A_p(:),A_u(:),A_v(:),A_q(:),A_zm(:),A_x(:)
      real A_y(:),A_z(:),B_p(:),B_u(:),B_v(:),B_q(:)
      real B_zm(:),B_x(:),B_y(:),B_z(:),dt,dthlf
      real p_tmp(i_A_hi-i_A_lo+1),v_tmp(i_A_hi-i_A_lo+1)
```

```
C     subroutine update_couplings_1d continued

      real q_tmp(i_A_hi-i_A_lo+1),u_tmp(i_A_hi-i_A_lo+1)
      real zm_tmp(i_A_hi-i_A_lo+1),x_tmp(i_A_hi-i_A_lo+1)
      real y_tmp(i_A_hi-i_A_lo+1),z_tmp(i_A_hi-i_A_lo+1)
      p_tmp(1:i_A_hi-i_A_lo+1) = A_p(i_A_lo:i_A_hi)
C     ... save all the temporaries
      A_q(i_A_lo:i_A_hi) = A_q(i_A_lo:i_A_hi) * frac
     &                     + B_q(i_B_lo:i_B_hi) * (1.0-frac)
C     ... update all of the ``A'' variables
      B_q(i_B_lo:i_B_hi) = B_q(i_B_lo:i_B_hi) * frac
     &                     + q_tmp * (1.0 - frac)
C     ... update all of the ``B'' variables
      end subroutine update_couplings_1d
```

Figure 19: Transformed HPF input and update routines for 1-d couplings.

```
      subroutine update_couplings_1d_3d_ijk(i_A_size,
     &  i_A_proc,j_A_proc,k_A_proc,A_p,A_q,A_u,A_v,A_zm,
     &  A_x,A_y,A_z,i_A_lo,i_A_hi,i_B_size,j_B_size,
     &  k_B_size,i_B_off,j_B_off,k_B_off,B_p,B_q,B_u,B_v,
     &  B_zm,B_x,B_y,B_z,i_B_lo,j_B_lo,k_B_lo,i_B_hi,
     &  j_B_hi,k_B_hi,dthlf,dt)
!HPF$ template decompA(Num_Proc_i,Num_Proc_j,Num_Proc_k)
!HPF$ template decompB(2*i_B_size,2*j_B_size,2*k_B_size)
!HPF$ align (*) with *decomp(i_A_proc,j_A_proc,k_A_proc)
!HPF$&     ::A_p,A_q,A_u,A_v,A_zm,A_x,A_y,A_z
!HPF$ align (i,j,k)with *decompB(i+i_B_off,j+j_B_off,
!HPF$&     k+k_B_off)::B_p,B_q,B_u,B_v,B_zm,B_x,B_y,B_z
!HPF$ distribute (block,block,block)onto procs::decomp
!HPF$ distribute *(cyclic(i_B_size/Num_Proc_i),
!HPF$& cyclic(j_B_size/Num_Proc_j),
!HPF$& cyclic(k_B_size/Num_Proc_k))onto procs::decompB
      real A_p(:),A_u(:),A_v(:),A_q(:),A_zm(:),A_x(:)
      real A_y(:),A_z(:),B_p(:,:,:),B_u(:,:,:)
      real B_v(:,:,:),B_q(:,:,:),B_zm(:,:,:),B_x(:,:,:)
```

```
C     subroutine update_couplings_1d_3d_ijk continued

      real B_y(:,:,:),B_z(:,:,:),dt,dthlf
      real p_tmp(i_A_hi-i_A_lo+1),v_tmp(i_A_hi-i_A_lo+1)
      real q_tmp(i_A_hi-i_A_lo+1),u_tmp(i_A_hi-i_A_lo+1)
      real zm_tmp(i_A_hi-i_A_lo+1),x_tmp(i_A_hi-i_A_lo+1)
      real y_tmp(i_A_hi-i_A_lo+1),z_tmp(i_A_hi-i_A_lo+1)
      p_tmp(1:i_A_hi-i_A_lo+1) = A_p(i_A_lo:i_A_hi)
C     ... save all the temporaries
      A_q(i_A_lo:i_A_hi) = A_q(i_A_lo:i_A_hi) * frac
     &    + B_q(i_B_lo:i_B_hi,j_B_lo:j_B_hi,k_B_lo:k_B_hi)
     &    * (1.0-frac)
C     ... update all of the ``A'' variables
      B_q(i_B_lo:i_B_hi,j_B_lo:j_B_hi,k_B_lo:k_B_hi) =
     &    B_q(i_B_lo:i_B_hi,j_B_lo:j_B_hi,k_B_lo:k_B_hi)
     &    * frac
     &     + q_tmp * (1.0 - frac)
C     ... update all of the ``B'' variables
      end subroutine update_couplings_1d_3d_ijk
```

Figure 20: Transformed HPF input and update routines for mixed couplings.

for a specific machine. Further, the code is not parallelized for a specific input set (reactor configuration). This is particularly important for codes such as TRAC, where many users do not work on code development and many code developers do not work on reactor model development. Finally, for the scientist trying to analyze the properties of a specific reactor, the parallelization is specific to their reactor design and hence should run significantly faster than a parallelization that is only code and not configuration specific.

4 Algorithm Validation

The two primary goals of this research are: 1) reduce the programmer burden for parallelization of composite grid problems, and 2) take advantage of the partial regularity of composite grid problems to provide acceptable performance in the resulting parallelization. The first goal is achieved via the automatic distribution algorithm, which removes the burden of grid mapping from the user. The second goal is achieved through our cloning and transformation process, which allows the use of regular distribution optimizations such as parallel and blocked communication.

Since the applications we are focusing on in this work are not written in a form that we can work with directly, we present measures of load balance and communication in place of timing results. The measure of load balance that we present is the total number of floating point additions, multiplications, and divisions. We compare the load balance measure for the processor with the most work ($Comp_{max}$) and the processor with the least work ($Comp_{min}$). We also present this number for the highest computation small mesh in each problem. None of these algorithms can produce a mapping with less computation on every processor than that of the highest computation small mesh. Consideration of this fact provides a practical upper bound on the number of processors for a problem when using any of these algorithms. The following measures of communication will be presented:

- maximum distance between any pair of communicating neighbors (*distance*);
- maximum number of communicating neighbors for any processor (*neighbors*);
- maximum amount of communication for a processor ($Comm_{max_1}$);
- maximum communication between any pair of processors ($Comm_{max_2}$); and
- total communication for the simulation ($Comm_{total}$).

With these measures of load balance and communication we will compare the results of the three distribution algorithms on three different problems. In the tables of results, the selected processor configuration is shown below the number of processors.

There are thirteen different types of components associated with reactor simulations: ACCUM, BREAK, FILL, PIPE, PLENUM, PRIZER, PUMP, ROD(or SLAB), STGEN, TEE, TURB, VALVE and VESSEL. Table 1 shows the approximate number of additions/multiplications, and divisions[3] for each element of one-dimensional, two-dimensional(ROD), and three-dimensional (VESSEL) components [5].

Component Type	# Add/Mults	# Divides
1-d	11,798	704
2-d	64,427	888
3-d	117,057	1073

Table 1: **Operation counts for all component types.**

For each test problem, we ran the two random algorithms three times. In the sections that follow, we show complete results for all runs to provide insight into the range of results.

4.1 LOFT Reactor Model

The LOFT model is a small model that has been used to illustrate concepts in the Los Alamos TRAC manuals. The LOFT reactor model nodalization came from the TRAC manual [11]. There are two versions of this model. The difference between the two versions is that the second version replaces the 3-d reactor vessel with a set of 1-d components and changes the heat structures.

The 3-d version of this model has 28 components (192 3-d cells and 128 1-d cells) and 11 heat structures. Table 2 shows the number of each type of component and the total number of elements for each type of component. The validation results for the 3-d LOFT reactor model are shown in Table 3. There are 1,567,560 floating point computations for each cycle associated with the largest mesh in the 3-d LOFT reactor model used as input for this experiment. This implies that at most 5 processors can be used and still achieve near load balance.

[3]These operation counts were obtained from information collected by Susan Woodruff using the CRAY Hardware Performance Monitor using data sets generated by Jim Lime at Los Alamos National Laboratories.

Component Type	# Components	# Elements
BREAK	1	1
FILL	2	2
PIPE	4	9
PRIZER	2	8
PUMP	2	4
SLABS	11	88
STGEN	1	23
TEE	11	68
VALVE	4	13
VESSEL	1	192

Table 2: **Component and element counts for 3-d LOFT reactor model.**

Num. Procs	Validation Measure	Random	Random	Random	Balanced Random	Balanced Random	Balanced Random	Topology-based
2	*distance*	1	1	1	1	1	1	1
	neighbors	1	1	1	1	1	1	1
	$Comm_{max_1}$	642,723	474,765	811,388	733,294	530,110	412,076	258,817
	$Comm_{max_2}$	642,723	474,765	811,388	733,294	530,110	412,076	258,817
	$Comm_{total}$	642,723	474,765	811,388	733,294	530,110	412,076	258,817
	$Comp_{max}$	4,520,278	5,044,080	4,620,294	3,799,008	4,070,206	3,807,664	3,801,572
	$Comp_{min}$	3,076,456	2,552,654	2,976,440	3,797,726	3,526,528	3,789,070	3,795,162
4	*distance*	2	2	2	2	2	2	2
(4)	*neighbors*	3	3	3	3	3	3	2
	$Comm_{max_1}$	424,061	502,350	617,605	577,780	524,199	449,010	227,991
	$Comm_{max_2}$	221,165	283,693	259,068	315,624	333,877	305,599	165,800
	$Comm_{total}$	692,559	991,885	1,017,226	889,352	889,289	760,743	315,573
	$Comp_{max}$	3,288,990	4,268,956	2,801,412	2,090,080	3,135,120	2,612,600	1,917,616
	$Comp_{min}$	833,788	312,550	275,044	1,670,140	362,558	1,137,682	1,880,110
8	*distance*	3	3	3	3	3	3	2
(2x2x2)	*neighbors*	7	5	6	4	5	4	3
	$Comm_{max_1}$	358,652	543,085	311,919	205,838	437,136	383,674	205,838
	$Comm_{max_2}$	155,850	209,023	143,656	143,272	131,078	131,078	143,272
	$Comm_{total}$	1,067,086	976,498	833,685	1,156,150	1,042,058	1,107,449	579,418
	$Comp_{max}$	2,826,416	4,007,696	2,216,382	1,567,560	1,567,560	1,567,560	1,567,560
	$Comp_{min}$	175,028	87,514	250,040	311,268	436,288	511,300	311,268
16	*distance*	4	4	4	4	4	4	3
(2x2x4)	*neighbors*	7	10	7	5	6	5	3
	$Comm_{max_1}$	233,743	308,623	233,743	373,982	308,946	233,739	168,288
	$Comm_{max_2}$	90,407	130,822	90,407	102,762	155,909	90,534	90,407
	$Comm_{total}$	1,014,221	1,079,588	1,014,221	1,042,060	1,054,731	1,054,560	551,581
	$Comp_{max}$	1,828,820	2,713,898	1,828,820	1,567,560	1,567,560	1,567,560	1,567,560
	$Comp_{min}$	62,510	12,502	62,510	0	37,506	0	0

Table 3: **Validation results for the 3-d LOFT reactor model.**

In the 3-d LOFT results we see that the maximum computation for a processor can not be reduced by using more than 8 processors for the topology-based algorithm. The random distribution never achieves optimal load balance, reguardless of the number of processors. The balanced random algorithm sometimes achieves optimal load balance, but generates nearly twice as much total communication as the topology-based algorithm does. These results illustrate our earlier point that the cost of the total small mesh computation divided by the cost of the highest computation small mesh bounds the number of processors that can be used effectively. It is perhaps interesting to note that using more processors than this bound can reduce the communication as in the case of going from 8 to 16 processors for the 3-d LOFT data set, but it does not effectively utilize the extra processors. Therefore, in the rest of the validation results we will only present results for up to the next power of two number of processors past that indicated by the bound imposed by the highest computation small mesh.

The 1-d version of this model has 40 components (169 1-d cells) and 1 heat structure. Table 4 shows the

Component Type	# Components	# Elements
BREAK	1	1
FILL	3	3
PIPE	5	12
PUMP	2	4
PRIZER	2	8
SLAB	1	4
STGEN	1	23
TEE	22	105
VALVE	4	13

Table 4: **Component and element counts for the 1-d LOFT reactor model.**

number of each type of component and the number of elements for each type of component. The validation results for the 1-d LOFT reactor model are shown in Figure 5. There are 287,546 floating point computations for each cycle associated with the largest mesh in the 1-d LOFT reactor model used as input for this experiment. This implies that at most 8 processors can be used and still achieve near load balance.

We again see in the 1-d LOFT results that the topology-based distribution algorithm produces the best load balance. Further, both random algorithms induce nearly twice as much total communication as the topology-based algorithm does.

4.2 H.B. Robinson Reactor Model

Like the LOFT model, the H.B. Robinson model is a small model that has been used to illustrate concepts in the Los Alamos TRAC manuals. The H.B. Robinson reactor model nodalization came from the TRAC manual [1]. This model has 100 components (144 3-d cells and 433 1-d cells) and 21 heat structures. Table 6 shows the number of each type of component and the total number of elements for each type of component. The validation results for the H.B. Robinson reactor model are shown in Figure 7. There are 979,725 floating point computations for each cycle associated with the largest mesh in the H.B. Robinson reactor model used as input for this experiment. This implies that at most 13 processors can be used and still achieve near load balance.

We again see in the H.B. Robinson results that the topology-based distribution algorithm produces good load balance until the maximum number of processors is used. The random algorithms induce from three to ten times as much total communication as the topology-based algorithm does.

Num. Procs	Validation Measure	Random	Random	Random	Balanced Random	Balanced Random	Balanced Random	Topology-based
2	*distance*	1	1	1	1	1	1	1
	neighbors	1	1	1	1	1	1	1
	$Comm_{max_1}$	237,711	499,033	455,996	729,804	468,518	561,100	174,604
	$Comm_{max_2}$	237,711	499,033	455,996	729,804	468,518	561,100	174,604
	$Comm_{total}$	237,711	499,033	455,996	729,804	468,518	561,100	174,604
	$Comp_{max}$	1,623,982	1,423,950	1,373,942	1,236,420	1,237,698	1,225,196	1,187,690
	$Comp_{min}$	75,012	950,152	1,000,160	1,137,682	1,136,404	1,148,906	1,186,412
4	*distance*	2	2	2	2	2	2	2
(2x2)	*neighbors*	2	2	2	2	2	2	2
	$Comm_{max_1}$	368,464	393,118	343,223	511,022	468,503	486,557	100,119
	$Comm_{max_2}$	205,745	267,990	218,095	273,484	230,620	205,751	74,875
	$Comm_{total}$	518,629	605,768	556,095	673,854	749,326	699,075	199,643
	$Comp_{max}$	887,642	937,650	836,356	898,866	875,140	873,862	600,096
	$Comp_{min}$	200,032	200,032	250,040	337,554	462,574	437,570	587,594
8	*distance*	3	3	3	3	3	3	3
(2x4)	*neighbors*	6	7	6	6	7	6	4
	$Comm_{max_1}$	386,487	398,720	268,442	492,068	391,968	391,968	112,236
	$Comm_{max_2}$	155,684	130,656	155,693	143,170	143,170	143,170	37,543
	$Comm_{total}$	761,809	773,979	656,226	929,672	879,960	892,474	299,786
	$Comp_{max}$	623,822	562,590	486,300	573,814	450,072	562,590	312,550
	$Comp_{min}$	112,518	37,506	75,012	112,518	162,526	100,016	275,044

Table 5: **Validation results for the 1-d LOFT reactor model.**

4.3 Westinghouse AP600 Reactor Model

The Westinghouse AP600 reactor model nodalization was developed by Jim Lime at Los Alamos National Laboratories with support from the Nuclear Regulatory Commission [10]. This model has 173 hydro components (1060 3-d cells and 865 1-d cells) and 47 heat structures. Table 8 shows the number of each type of component and the total number of elements for each type of component.

The validation results for the Westinghouse AP600 reactor model are shown in Figure 9. There are 3,396,380 floating point computations for each cycle associated with the largest mesh in the AP600 reactor model used as input for this experiment. This implies that at most 20 processors can be used and still achieve near load balance.

In the AP600 results we find the only case where the balanced random produces a better load balance than the topology-based algorithm (for 8 processors). In this particular case the total communication is more than a factor of two better for the topology-based algorithm and there is one less neighbor for the topology-based algorithm. This is also the first case where we find some of the communication measures to be better for the random algorithms, e.g., see *neighbors* for 32 processors. From these results we conclude that the topology-based algorithm would probably outperform the balanced random algorithm, but this is a case that we will return to at some point in the future to try to learn how to improve the topology-based algorithm. Even for this test problem, both random algorithms induce from one and a half to five times as much total communication as the topology-based algorithm does.

5 Conclusions

We now have an algorithm which automatically finds a distribution of data in ICRM problems given a well-structured HPF program and topological connection specifications. Along with the algorithm to determine distri-

Component Type	# Components	# Elements
ACCUM	3	12
BREAK	1	1
FILL	27	27
PIPE	17	142
PLENUM	2	8
PUMP	2	6
SLAB	21	126
TEE	32	188
VALVE	15	49
VESSEL	1	144

Table 6: **Component and element counts for H.B. Robinson reactor model.**

Num. Procs	Validation Measure	Random	Random	Random	Balanced Random	Balanced Random	Balanced Random	Topology-based
2	*distance*	1	1	1	1	1	1	1
	neighbors	1	1	1	1	1	1	1
	$Comm_{max_1}$	1,925,037	1,594,031	2,070,846	1,695,802	1,752,610	1,769,099	175,212
	$Comm_{max_2}$	1,925,037	1,594,031	2,070,846	1,695,802	1,752,610	1,769,099	175,212
	$Comm_{total}$	1,925,037	1,594,031	2,070,846	1,695,802	1,752,610	1,769,099	175,212
	$Comp_{max}$	7,448,792	8,170,062	8,619,334	6,826,979	6,878,269	6,827,779	6,822,410
	$Comp_{min}$	6,194,264	5,472,994	5,023,722	6,816,077	6,764,787	6,815,277	6,820,646
4	*distance*	2	2	2	2	2	2	2
(2x2)	*neighbors*	3	3	3	3	3	3	3
	$Comm_{max_1}$	1,385,452	1,684,366	1,715,044	1,980,719	1,926,480	1,682,133	340,767
	$Comm_{max_2}$	836,066	856,983	751,173	885,810	888,699	584,231	253,157
	$Comm_{total}$	2,539,938	2,645,116	2,399,371	2,748,397	2,734,669	2,505,357	453,393
	$Comp_{max}$	4,346,050	6,607,312	5,459,451	3,440,296	3,591,843	3,441,578	3,423,225
	$Comp_{min}$	3,026,207	2,012,022	2,234,253	3,381,632	3,174,467	3,374,499	3,393,334
8	*distance*	3	3	3	3	3	3	3
(2x4)	*neighbors*	6	6	7	7	7	7	3
	$Comm_{max_1}$	1,315,890	1,571,388	1,182,595	1,074,040	1,300,735	1,079,338	256,026
	$Comm_{max_2}$	439,537	348,954	339,441	268,420	261,532	243,456	168,424
	$Comm_{total}$	3,069,682	3,082,156	3,057,056	2,876,610	3,094,870	2,926,450	757,014
	$Comp_{max}$	2,765,188	2,973,635	3,615,806	1,863,039	1,835,471	1,804,857	1,723,994
	$Comp_{min}$	671,021	773,842	887,642	1,446,145	1,471,149	1,607,389	1,699,231
16	*distance*	4	4	4	4	4	4	4
(2x2x4)	*neighbors*	12	11	11	9	12	13	5
	$Comm_{max_1}$	1,122,450	1,290,502	107,817	966,800	979,010	1,062,721	268,424
	$Comm_{max_2}$	168,332	208,707	208,603	143,304	168,204	143,352	130,662
	$Comm_{total}$	3,076,498	3,141,973	3,363,202	3,275,604	3,300,632	3,250,504	1,013,236
	$Comp_{max}$	1,887,002	2,306,701	3,419,861	979,725	979,725	979,725	979,725
	$Comp_{min}$	175,028	125,020	162,526	487,578	687,610	737,618	594,727

Table 7: **Validation results for H.B. Robinson reactor model.**

Component Type	# Components	# Elements
ACCUM	2	10
BREAK	10	10
FILL	11	11
PIPE	70	385
PLENUM	3	3
PUMP	4	20
SLAB	47	917
TEE	41	256
VALVE	29	170
VESSEL	3	1060

Table 8: **Component and element counts for Westinghouse AP600 reactor model.**

bution, we have shown the form of HPF program that a user would write for input to the automatic distribution system. We have also shown how, with the use of interprocedural analysis, we can reduce the programming effort involved in development and support by automatically cloning the routines that require distribution specification for the meshes and adding the statements needed for reading and processing distribution specification in a special input file. Although the use of this automatic distribution procedure requires interprocedural analysis to perform the cloning operations, it does not require recompilation of the source when the data set changes as the distribution specifications are read in by the HPF program with the mesh and coupling specifications. Our experiments show that we almost always obtain load balance as good as, and often significantly better than, random algorithms while reducing the total communication per iteration by about 50% or in some cases as much as 90%.

We are currently working on another algorithm for automatic distribution of these problems in which we pack the meshes that are too large for a single processor together and distribute the packed group over all processors. This will increase the number of processors that we can make practical use of for these problems. This approach will also extend the set of applications that we can automatically determine distributions for.

References

[1] B.E. Boyack, H. Stumpf, and J.F. Lime. *TRAC User's Guide*. Los Alamos National Laboratory, Los Alamos, New Mexico 87545, 1985.

[2] C. Chase, K. Crowley, J. Saltz, and A. Reeves. Compiler and runtime support for irregularly coupled regular meshes. *Journal of the ACM*, July 1992.

[3] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[4] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, November 1992.

[5] Susan J. Jolly-Woodruff and James F. Lime. Los Alamos National Laboratories, March 1994. Private communication.

[6] Joe Kelly. Nuclear Regulatory Commission, January 1994. Private communication.

[7] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, Massachusetts, 1994.

[8] L.M. Liebrock and K. Kennedy. Automatic data distribution of large meshs in coupled grid applications. Technical Report 94-395, Rice University, Center for Research in Parallel Computation, Houston, TX, April 1994.

Num. Procs	Validation Measure	Random	Random	Random	Balanced Random	Balanced Random	Balanced Random	Topology-based
2	*distance*	1	1	1	1	1	1	1
	neighbors	1	1	1	1	1	1	1
	$Comm_{max_1}$	6,474,062	5,880,201	5,880,201	4,399,114	4,399,114	4,764,464	898,236
	$Comm_{max_2}$	6,474,062	5,880,201	5,880,201	4,399,114	4,399,114	4,764,464	898,236
	$Comm_{total}$	6,474,062	5,880,201	5,880,201	4,399,114	4,399,114	4,764,464	898,236
	$Comp_{max}$	42,053,568	36,049,960	36,049,960	35,357,856	35,357,856	35,375,492	35,354,184
	$Comp_{min}$	28,654,532	34,658,144	34,658,144	35,350,248	35,350,248	35,332,616	35,353,952
4	*distance*	2	2	2	2	2	2	2
(4)	*neighbors*	3	3	3	3	3	3	3
	$Comm_{max_1}$	5,004,068	5,373,743	5,100,365	4,249,056	5,193,137	6,072,235	2,732,938
	$Comm_{max_2}$	2,124,474	2,187,416	1,853,293	2,156,131	2,230,068	2,467,942	1,994,743
	$Comm_{total}$	9,074,508	8,961,254	8,818,410	7,623,410	9,296,276	8,393,454	3,045,056
	$Comp_{max}$	21,971,402	28,353,922	22,836,846	19,725,130	18,745,406	20,312,964	17,682,254
	$Comp_{min}$	12,689,058	9,814,157	11,843,250	16,081,910	16,588,082	15,029,737	17,673,916
8	*distance*	3	3	3	3	3	3	3
(4x2)	*neighbors*	7	7	7	7	7	7	6
	$Comm_{max_1}$	3,161,931	3,520,754	3,697,177	42,13,511	4,229,617	4,309,920	3,444,543
	$Comm_{max_2}$	831,463	909,152	781,599	1,077,368	838,499	1,099,967	1,188,476
	$Comm_{total}$	9,882,217	9,972,450	9,857,245	10,046,362	10,130,799	10,259,210	4,369,449
	$Comp_{max}$	14,017,239	13,423,794	14,254,777	10,776,975	8,964,021	10,123,825	9,592,649
	$Comp_{min}$	553,318	5,213,498	2,229,684	6,847,655	8,546,404	8,051,934	8,169,262
16	*distance*	4	4	4	4	4	4	4
(2x4x2)	*neighbors*	14	14	14	12	12	11	5
	$Comm_{max_1}$	2,737,633	1,932,648	2,463,119	2,405,384	2,122,664	2,531,268	1,984,966
	$Comm_{max_2}$	417,022	588,027	442,058	417,022	417,022	470,299	653,310
	$Comm_{total}$	10,779,059	10,281,767	10,872,695	10,518,055	10,535,847	10,440,030	5,268,712
	$Comp_{max}$	7,775,049	9,327,302	8,573,895	5,682,405	6,923,390	5,682,405	5,498,962
	$Comp_{min}$	1,650,746	575,092	854,464	323,850	2,467,704	3,134,638	3,765,830
32	*distance*	6	6	6	6	6	6	6
(8x4)	*neighbors*	19	22	13	17	21	19	20
	$Comm_{max_1}$	1,912,084	1,582,016	2,039,157	1,544,462	2,896,651	1,610,225	4,162,976
	$Comm_{max_2}$	221,029	261,324	208,703	205,770	233,547	233,547	482,353
	$Comm_{total}$	11,087,906	11,321,581	11,296,545	11,165,883	11,203,437	11,072,775	7,060,487
	$Comp_{max}$	5,986,540	5,270,880	9,745,719	4,049,530	3,527,010	4,310,790	3,406,077
	$Comp_{min}$	112,518	150,024	112,518	639,125	362,558	939,173	1,250,682

Table 9: **Validation results for Westinghouse AP600 reactor model.**

[9] L.M. Liebrock and K. Kennedy. Parallelization of linearized application in Fortran D. In *International Parallel Processing Symposium '94*, pages 51–60, Washington, DC, April 1994.

[10] J.F. Lime and B.E. Boyack. TRAC large-break loss-of-coolant accident analysis for the AP600 design. In *International Topical Meeting on Advanced Reactors Safety*, Pittsburgh, PA, April 1994.

[11] J.C. Lin, V. Martinez, and J.W. Spore. *TRAC Developmental Assessment Manual.* Los Alamos National Laboratory, Los Alamos, New Mexico 87545, 1993.

[12] George F. Niederauer. Los Alamos National Laboratories, January 1994. Private communication.