Automatic Distribution of Large Mesh Irregularly Coupled Regular Mesh Problems

Ken Kennedy Lorie Liebrock
Joel Saltz

CRPC-TR94523-S
May, 1994

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

This research was supported through Cooperative Agreement No. CCR-9120008, NSF/NASA Agreement No. ASC-9213821, and ONR Agreement No. N00014-93-1-0158.

Abstract

Irregularly coupled regular mesh problems arise in important application areas, e.g., fluid flow simulation and aerodynamic simulation. Such simulations are computationally intensive and inherently parallel. Unfortunately, parallel programming languages such as High Performance Fortran provide little explicit support for these problems. It is essential that these applications be automatically parallelized when the mesh configurations are generated automatically. Here we develop an algorithm that automatically determines, at compile time, distributions for a class of irregularly coupled regular mesh problems. The input for the algorithm is a well-structured modern Fortran program, a description of the regular meshs, and a description of the topological connections between the regular meshs. We illustrate the description of topological connections by extending Fortran D specifications with a statement that allows the user to specify connectivity between meshs. In addition, the user specifies the target architecture and the number of processors, P. A standard processor configuration is an n-dimensional processor mesh with at most the same number of dimensions as the target architecture and exactly P processors. The algorithm begins by finding an optimal (according to a model) mapping of array dimensions onto processor dimensions for each mesh and standard processor configuration. Next, the standard processor configuration and associated mapping that minimizes the total computation and internal (to the meshs) communication is selected. Finally, the meshs are aligned with respect to each other in order to reduce coupling communication cost in the selected mapping. This results in full distribution and alignment specifications. These specifications are used as input to a High Performance Fortran program that applies the selected mapping (distribution and alignment) for the execution of the simulation code. Excerpts from such a High Performance Fortran program are presented to illustrate style and compilation issues. Some of the advantages to this automatic approach are that it is applicable for run-time or compile-time analysis and communication generation; conforms to a uniform memory model; is applicable even when automatic verification of parallelism over meshs is not possible; is applicable for SIMD architectures; and is easily applicable to Fortran D or High Performance Fortran compilation. A sample High Performance Fortran code was used with three real-world irregularly coupled regular mesh configurations to validate this algorithm.

1 Background: Irregularly Coupled Regular Mesh Problems

Detailed simulation of complex physical phenomena is computationally intensive. An increasing number of physical phenomena simulations involve more than one grid. Each grid in these problems corresponds to a different physical entity. A simple example of this type of problem is the simulation of the material flow through an elbow with two parallel vanes inside. Hugh Thornburg at Mississippi State University generated coupled meshs for this simulation, which are shown in Figure 1.

Perhaps one of the best known application areas for multiple grid approachs is aerodynamics. In aerodynamic simulations, different grids are used to resolve flow in the space surrounding the fuselage, wings, foreplane, pylons, etc. [22, 5, 18]. Two examples of grids for such aerodynamic simulations are shown in Figure 2 and Figure 3. The grid descriptions for both of these examples were provided by Paul Craft who works with Dr. Brahat Soni at Mississippi State University. These examples are used for validation of our algorithms in this paper. Applications of this type are called multiblock, composite grid or irregularly coupled regular mesh(ICRM) problems.

Many ICRM 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, HOPE [22] is a winged vehicle for space transportation called the H-II Orbiting Plane planned by the National Space Development Agency of Japan. To make accurate simulation of re-entry feasible, hypersonic aerodynamics and aerothermodynamic characteristics must be precisely evaluated. Simulation of one model of a HOPE vehicle provides an indication of the number of grid points used in such computations. In one unsymmetrical calculation of the HOPE 63 model, a total of approximately 9,000,000 grid points were used in the three-dimensional grids.

Fortunately, these problems are inherently parallel. Unfortunately, they are not necessarily easy to parallelize. From Smith and Eriksson [17], we see that along with the computational complexity there is added complexity due to the need for multiple coupled grids:

> Generating structured grids about complex geometries that map into a single rectangular computational block is, for all practical purposes impossible. Considering this dilemma, there are two directions that

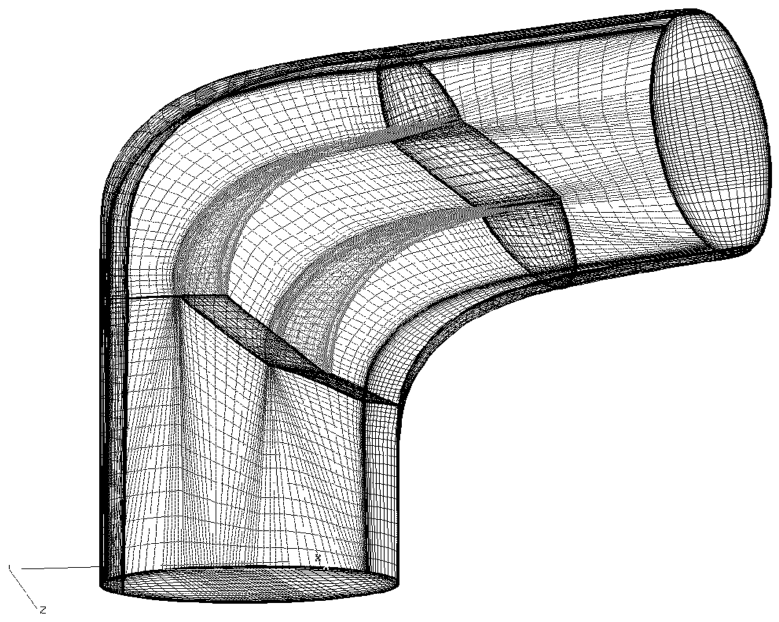

Figure 1: Automatically Generated Grids for Flow through an Elbow

can be taken for creating multiple block structured grids. They are:

1. create multiple grids that adjoin each other, or
2. create multiple grids that overlap each other.

In recent years, it has become possible to automatically generate the grids used in some ICRM problems[15, 19, 14, 1, 17, 20, 16]. Shaw and Weatherill [15] provide an introduction to the problem of automatic grid generation and the difficulty encountered when generating grids for multiblock problems:

> The fundamental problem that is encountered in the construction of structured, body-conforming meshs for general aerodynamic configurations is that each component of the configuration has its own natural type of grid topology and that these topologies are usually incompatible with each other. In other words, in attempting to discretize the flow domain around an arbitrary set of two-dimensional shapes, or some complex three-dimensional shape, a direct conflict arises between the maintenance of a globally structured grid and the preservation of a grid which naturally aligns itself with the local geometric features of a configuration.
>
> This inconsistency has motivated the development of a general category of mesh construction techniques know as multiblock or composite grid generation. Here, the single set of curvilinear coordinates, inherent to a globally structured grid, is replaced by an arbitrary number of coordinate sets that interface node to node with each other at notional boundaries within the physical domain. Returning to the mapping concept, the approach can be viewed as the decomposition of the flow domain into subregions, which are referred to as blocks, each of which is transformed into its own unit cube in computational space. Global structure within a grid is sacrificed, but the concept proves the flexibility of connections required to construct a grid whose topological structure, local to each component of a complex configuration, is compatible with the particular geometric characteristics of the component. All points which are connected by a common coordinate system can be directly referenced with respect to each other, but bear no direct structured relationship to any of the grid points that lie in coordinate-sets in other blocks.
>
> Thus, the multiblock technique necessitates information to be defined describing how the blocks connect together...

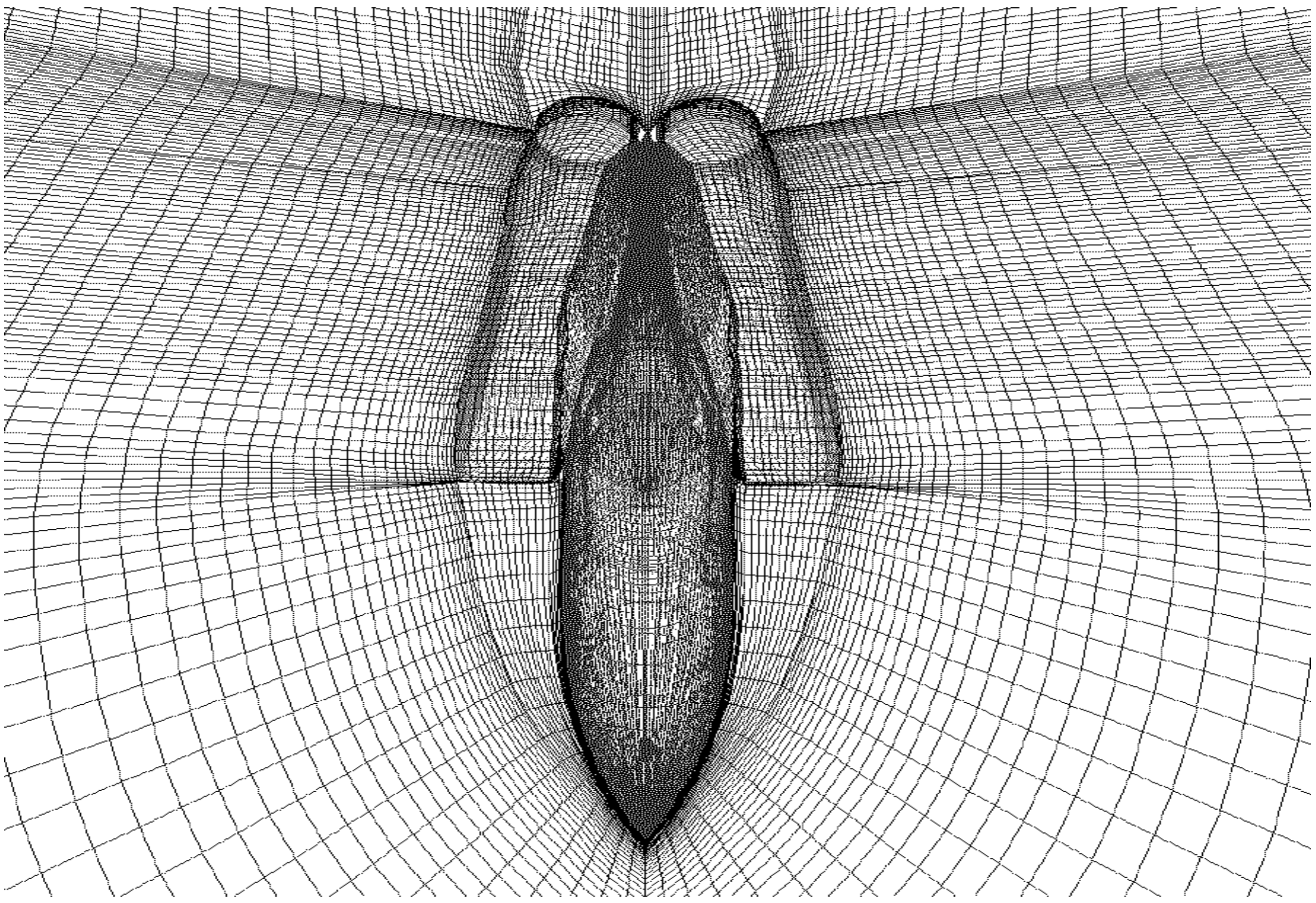

Figure 2: F15e Volume Grid Configuration for Fuselage-Inlet-Nozzle

When grids are generated automatically, it is particularly important that distribution is done automatically. The applications programmer must not be required to decipher the automatically generated grid assembly and parallelize the simulation code for a specific configuration by hand.

As a result we have a set of computationally intensive problems that are inherently parallel but difficult to parallelize efficiently. None of the approaches to parallelization of these problems save the programmer from having to decipher the grid assembly and determine the data distributions. The one type of ICRM problem for which automatic distribution has been explored is multigrid. Thuné has been working on automatic distribution of data structures for multigrid computations[21].

We are working to make parallelization of these applications easier for the user as well as efficient and applicable to a large class of architectures. To this end, we must develop an approach that achieves more of the computational efficiency of a regular approach. We exploit not only the regularity inside of each mesh, but also the topology of the connections between meshs to automatically determine distribution of data structures. We illustrate the topological information that must be extracted via an extension of Fortran D (coupling specifications). We extend compiler technology to illustrate automatic parallelization of large mesh ICRM codes. This research is based on an understanding of problem topology. Here we target applications with automatically generated meshs where the "annotations" are part of the input. This work provides an important first step in automatic parallelization of large mesh ICRM applications. To use this preliminary version of the approach, a programmer would have to:

1. Write a High Performance Fortran program of the type described herein.

2. Specify the mesh and topological connectivity information in a "standard" form.

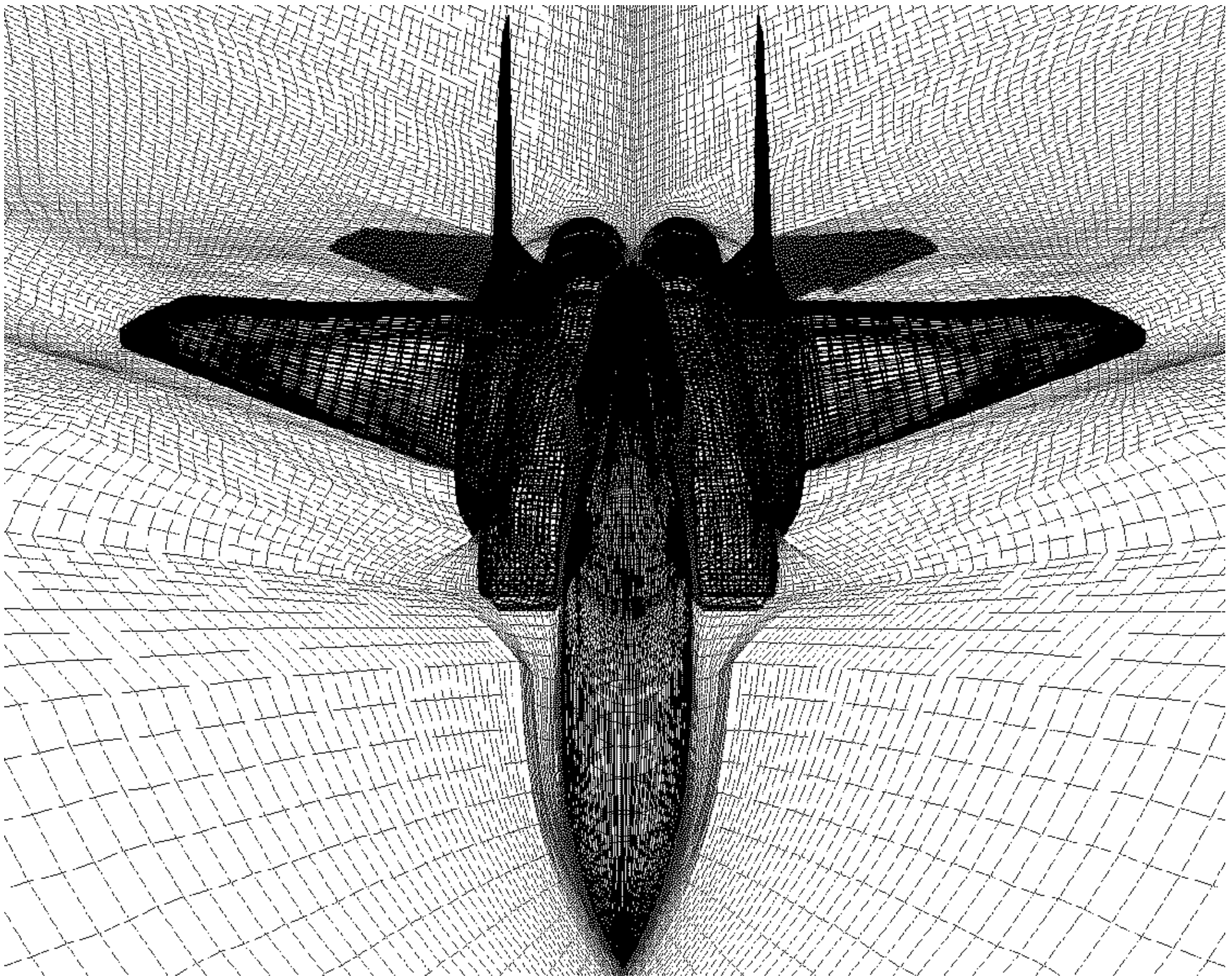

Figure 3: F15e Volume Grid Configuration for Full Aircraft

Note that in the applications under consideration all data structures are aligned to the decompositions in the same way. Our methods determine how to map the decompositions to the specified processor array. When grids are generated automatically, the necessary information could be obtained by translating the grid specifications that are part of the input file. In this case the specifications presented here indicate what information must be extracted from the input.

The example grids for aerodynamic simulations shown in Figures 2 and 3 were automatically generated using GENIE++, as were the grids for the elbow in Figure 1.

2 Language Support for ICRM Problems

For illustrative purposes, we introduce an extension of Fortran D for the specification of mesh coupling. We could have extended other data parallel languages, e.g. Vienna Fortran or HPF, in an analogous manner. Our validation results are obtained by generating an HPF code and compiling it on Digital Equipment Corporation's HPF compiler.

2.1 Specification of Topological Connectivity

We illustrate the specification of topological connectivity via an extension to Fortran D[6, 9] that describes the connections between coupled decompositions. We first introduce two problems that will be used to illustrate the algorithms presented in this paper. In the grid illustrations, dashed lines represent couplings and solid lines represent the boundaries of the grids.

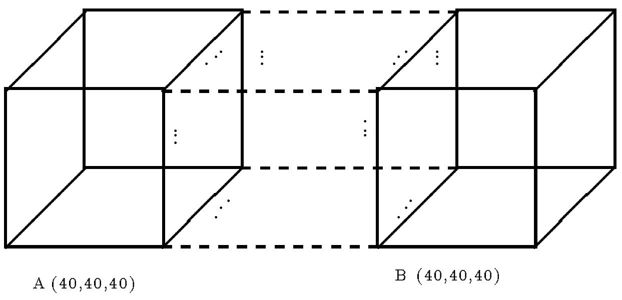

Figure 4: 3D CFD Multiblock Configuration.

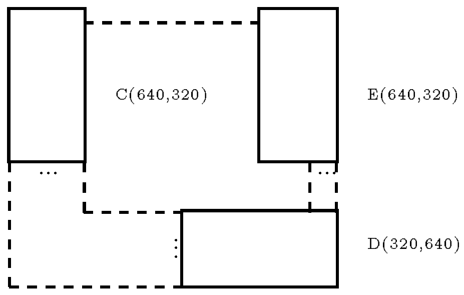

Figure 5: 2D-Multiblock Configuration.

The first example problem is a multiblock CFD problem. The problem is to analyze transonic flow over 3-d configurations by solving the unsteady thin-layer Navier-Stokes equations. The problem was used by J. Saltz [3] to show the feasibility of the PARTI multiblock approach to parallelization of ICRM problems. This problem consists of two meshs, A and B, each of which are 40x40x40 with one coupled face. These meshs are illustrated in Figure 4. We refer to this as the 3D-CFD problem.

A second example for algorithm illustration involves a computation associated with the trio of coupled two dimensional grids illustrated in Figure 5. This problem illustrates some features of the distribution algorithm that are not evident with the first example. We will refer to this as the 2D-Multiblock problem.

Consider our 3D-CFD example. We need to couple two faces: the face of A where the second dimension index is 40 to the face of B where the second dimension index is 1. To express this in a COUPLE statement we could 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. For example, the couplings in our 2D-Multiblock example would be declared as follows.

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

This form does not support negative or non-unit strides. With this restrictive form it would take ten COUPLE statements to express the coupling in Figure 6. Hence we add an optional stride and 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 this coupling specification is:

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

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}

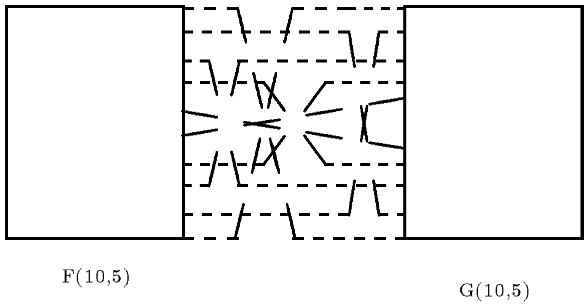

Figure 6: Negative Stride Coupling Example.

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 arbitrary coupling between different dimensions. The stride is optional and has a default value of 1.

This illustrates the type of coupling information that must be extractable from the input file for these applications. For a more general form of coupling specification see[10].

2.2 Programming Style Recommendations

Here we attempt to outline a programming style that is natural to ICRM applications such as aerodynamic simulations, but does not inhibit dependence analysis. The style should also make development and support more efficient in terms of programmer effort. Further, programs written in this style should achieve good performance when compiled by a good compiler. We will describe this style in terms of HPF[7].

Two of the most important features of a modern language, 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 meshs of a given type, i.e., all meshs with the same basic computation. For example, in many aerodynamics simulations there would be one allocatable array of structures for all of the meshs surrounding the aircraft as the same basic computation is performed for each mesh. This implies that the program has one allocatable array of structures for each type of mesh. 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 structures are allocated dynamically;
- there is an array of user defined data structures for each type of component.

There is also a compute routine associated with each type of mesh which 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.

Given a program written in this 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 should be manageable.

3 Automatic Distribution Algorithm

Minimally, for the algorithm we will describe shortly to be used, we need, in addition to the 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 meshs.

Note that for most large mesh applications the nature of the computation is the same on all meshs; the nature of the communication is the same in each dimension; and the coupling cost is the same for each pair of coupled elements. An estimate of these statistics could be provided by the user, but that alone does not eliminate the communication analysis needed in a Fortran D or HPF compiler or a runtime system such as PARTI. We want to save the user as much work as possible, so we would like to move as much of the analysis into the compiler as we can.

For this work, we restrict ourselves to consideration of ICRM applications in which:

- all meshs are large enough to be efficiently distributed over all processors; and
- computations inside of each mesh are regular (this implies regular internal communication for each mesh after distribution).

We are currently targeting a torus based communications topology. This does not seem unreasonable as most available machines either are meshs or can have them efficiently embedded in the machine topology.

In our preliminary experiments all analysis was performed by hand, although a preliminary implementation of the analysis is in progress.

We now begin with an overview of our automatic distribution algorithm and then consider each step of the algorithm in more detail. Details of code generation and sample compiler support will follow next. This section will conclude with a discussion of the limitations and advantages of this approach.

For efficient parallelization of ICRM applications, we would like to distribute each mesh according to its dimensionality [12]. On the other hand, the resulting code should have uniform memory allocation. Let n be the dimensionality of the mesh with the fewest number of dimensions. Then we will distribute all of the meshs over all of the processors in an n-dimensional processor topology. Some of the dimensions of higher dimensional meshs will be serialized. In this manner we attempt to balance the tradeoffs between the computation to communication ratio considerations relating to using the natural topology parallelization for every mesh and the memory allocation issues in SPMD Fortran codes.

The user provides mesh and coupling specifications along with how many processors to use on the target machine. These may be runtime input, but we are assuming we have runtime constants available to the algorithm.

Our automatic distribution algorithm must take into consideration the user provided information, the program and the topology of the target machine. Since we are given the number of processors to use on the target machine, there are only a fixed number of possible m-dimensional processor configurations that can be used, where $1 \leq m \leq \textit{maximum processor dimensionality}$. We limit this set to only those configurations with at most n dimensions (recall n is the number of dimensions in the mesh with fewest dimensions) and call this set the "standard processor configurations". The final mapping of decompositions will be to the standard processor configuration with the fastest total predicted runtime.

The basic idea of the automatic distribution algorithm is to perform the following steps.

1. Analysis: the program is analyzed to determine the approximate computation associated with each decomposition, the approximate communication associated with each dimension of each decomposition and the approximate communication associated with each pair of coupled decompositions. Alternatively this could be user input as for these problems this a single set of numbers.
2. Table Generation: the best mapping and predicted runtime for each decomposition on each standard processor configuration is found independently, ignoring coupling communication cost. Basically, we do an exhaustive search of mappings to find the best mapping to each standard processor configuration. Alternatively, we can use heuristics to limit the search space.
3. Global Minimization: the standard processor configuration that produces the minimum total runtime for all decompositions is found, ignoring coupling communication cost. This requires a sum for each standard processor configuration over all decompositions and selecting the one with the minimum runtime.
4. Coupling Communication Reduction: the decompositions are distributed across the standard processor configuration to reduce coupling communication cost (via transposing and/or shifting or folding around the

Decomposition A		
SPC	Time	Map
(32,1,1)	3483.3	(2,1,3)
(16,2,1)	2570.7	(2,3,1)
(8,4,1)	2128.0	(2,3,1)
(8,2,2)	2118.8	(2,1,3)
(4,4,2)	2112.7	(2,3,1)

Decomposition B		
SPC	Time	Map
(32,1,1)	3483.3	(2,1,3)
(16,2,1)	2570.7	(2,3,1)
(8,4,1)	2128.0	(2,3,1)
(8,2,2)	2118.8	(2,1,3)
(4,4,2)	2112.7	(2,3,1)

Figure 7: 3D-CFD Tables

torus). The heuristic we use is to reduce the maximum coupling communication cost for each decomposition. This is done by aligning the coupled subsections of coupled dimensions of different decompositions.

The distribution decision procedure for selecting a data distribution uses the computational model presented in [8]. The model provides expected runtimes based on application and machine parameters as well as the selected data distribution. Any model could be used which will predict approximate runtimes of different mappings on the target machine.

For the presentation of algorithmic details, we will denote variables associated with a particular decomposition using a "*decomposition name*" dot "*variable name*" notation, e.g., $A.size[i]$ is the number of elements in the i^{th} dimension of decomposition A.

3.1 Table Generation

In this first stage, for each standard processor configuration, we can evaluate the model for all mappings of the decompositions dimensions onto the dimensions of each standard processor configuration. This is an exhaustive search for the best mapping of a decomposition onto each standard processor configuration.

A table of entries is built with one entry per standard processor configuration. Each entry consists of an assignment of decomposition dimensions to processor dimensions and a predicted runtime. Note that only one decomposition dimension may be assigned to any processor dimension and that all unassigned dimensions are sequentialized.

For each standard processor configuration, all assignments of decomposition dimensions to processor dimensions are tried and one with the best predicted runtime is stored. When two mappings produce the same predicted time, one is selected arbitrarily.

In the worst case, for each standard processor configuration, we are looking at all partitions of the decomposition's dimensions, d_d, into the processor's dimensions, p_d. There are $\frac{d_d!}{(d_d-p_d)!}$ such partitions.

Since Fortran allows at most seven dimensions in arrays, d_d and p_d are less than or equal to seven. From this, for ALL problems there are most at 7! partitions for any standard processor configuration. More typically, $d_d \leq 4$ and $p_d \leq 3$ so that the number of partitions is less than or equal to 24.

The total number of standard processor configurations with p_d or fewer dimensions is:

$$\begin{aligned}
&\eta_{p_d}(f) = \eta_{p_d-1}(f) + \eta_{p_d}(f - p_d)\\
&\eta_{p_d}(0) = 1\\
&\eta_1(f) = 1
\end{aligned}$$

where 2^f is the number of processors. This is the same as the number of partitions of an integer, f, into p_d or fewer summands [4, 13].

Consider a machine with 2^{14} processors configurable in up to seven dimensions. For such a machine, there would be 105 standard processor configurations. As a more realistic example, consider a machine with 2^{14} processors that can be configured with one, two or three dimensions then the number of standard processor configurations is 24. For this machine there are

$$1 * \begin{pmatrix} 3 \\ 1 \end{pmatrix} + 7 * \begin{pmatrix} 3 \\ 2 \end{pmatrix} + 16 * \begin{pmatrix} 3 \\ 3 \end{pmatrix} = 40$$

Decomposition C		
SPC	Time	Map
(32,1)	7143.6	(1,2)
(16,2)	7126.4	(1,2)
(8,4)	7124.0	(1,2)

Decomposition D		
SPC	Time	Map
(32,1)	7143.6	(2,1)
(16,2)	7126.4	(2,1)
(8,4)	7124.0	(2,1)

Decomposition E		
SPC	Time	Map
(32,1)	7143.6	(1,2)
(16,2)	7126.4	(1,2)
(8,4)	7124.0	(1,2)

Figure 8: 2D-Multiblock Tables

evaluations of the model for each three dimensional decomposition. Hence, even when all possible mappings are evaluated, there is a constant upper bound on the number of evaluations to be done for each mesh.

The tables generated for the 3D-CFD and 2D-Multiblock examples are shown in Figures 7 and 8 for the case when 32 processors are used. In the tables, a standard processor configuration is specified by the number of elements in each dimension. The mapping specifies which dimension of the decomposition is mapped to the corresponding dimension of the standard processor configuration. For example, (2,1) specifies that the second dimension of the decomposition is mapped to the first processor dimension and vice versa. In the 3D-CFD problem the number of bytes of communication is higher in the first dimensions than in the other two. This makes it more favorable to use fewer processors in the first dimension than in the others. This leads to a heuristic for reducing the number of mappings that are considered.

Heuristic 1 *When one dimension of a decomposition has more communication than the other dimensions, if the standard processor configuration has fewer processors in one dimension than the other dimensions, then map the decomposition dimension having greater communication to the processor dimension with fewer processors.*

In the 2D-Multiblock problem the number of bytes of communication is the same in each dimension of each mesh. This makes the number of elements to be communicated be the only communication factor that influences the choice of mapping. Surface to volume effects[12] imply we want to have each communication "edge" be the same length. This leads to our second heuristic for reducing the number of evaluations of the model.

Heuristic 2 *When every dimension of a decomposition has the same amount of communication, if the standard processor configuration has fewer processors in one dimension than the other dimensions, then map the decomposition dimension having the fewest elements to the processor dimension with the fewest processors.*

Finally consider the case when two dimensions of a mesh are the same size and have the same amount of communication, e.g., the second and third dimension in the 3D-CFD example. This consideration leads to our final heuristic.

Heuristic 3 *When two dimensions of a mesh have the same size and communication, for each pair of mappings with these dimensions interchanged only one of the mappings needs to be evaluated. We record pairs of equivalent dimensions for possible use during coupling communication reduction.*

In practice we only apply the heuristics when the number of elements in each dimension of the decomposition is divisible by the number of processors in each dimension. With this qualification, we have found that the heuristics usually produce the same results as the exhaustive search. The only exception to this is caused by the use of the last heuristic, where the heuristic provides better performance as it allows interchange while the exhaustive does not.

3.2 Global Minimization

In the global minimization step, we want to select the standard processor configuration that will produce the best runtime for all of the meshs combined. To do this, we sum over the decomposition tables to find the standard processor configuration with minimum total predicted runtime. At this point we would have the optimal mapping (according to the model) if there was no communication between the different decompositions.

We begin by creating a new standard processor configuration table with each entry initialized to zero. Iterating over all of the standard processor configurations, for each decomposition's table we add the decomposition's predicted runtime to the total runtime for the standard processor configuration. The minimum value in the total runtime table is indexed by the standard processor configuration which we want to choose as the "best" one.

This stage of the algorithm takes time on the order of the number of standard processor configurations times the number of decompositions. This step of the procedure puts the greatest accuracy requirements on the model as we add predicted runtimes.

For the 3D-CFD problem the best standard processor configuration using 32 processors is (4,4,2) and for the 2D-Multiblock problem it is (8,4).

3.3 Coupling Communication Reduction

The purpose of coupling communication reduction is to use the knowledge of topology to reduce communication costs associated with coupling. The heuristic used attempts to minimize the maximum cost coupling communication for each decomposition subject to the dimension mapping already selected. This is done by aligning the centers of the coupled subranges of dimensions and setting directions (increasing or decreasing) to be the same for each processor dimension. Further, if interchanging two equivalent dimensions will reduce coupling cost, the interchange is performed.

For each dimension of every decomposition we will select a starting processor index and a direction. The processor index, (between 1 and the number of processors in the processor dimension that the decomposition's dimension is mapped to), specifies which processor contains the first block of elements for the given dimension of the decomposition. The direction, (+1 or -1), specifies whether the elements in the decomposition's dimension will be mapped in increasing or decreasing order.

Begin by selecting the decomposition with the single highest coupling communication cost and call it D_1. We set the processor index for D_1 to 1 and the direction to increasing in every parallel dimension. D_1 is now completely mapped to the best standard processor configuration. Make D_1 the only element in the set MAPPED. Create a max heap of all couplings involving D_1 with order determined by the coupling costs.

Now we repeat the following steps until all decompositions are in the set MAPPED.

1. Delete the maximum element from the coupling heap.

2. If both decompositions in the coupling are MAPPED then return to step 1. Otherwise, call the unmapped decomposition in the coupling D_u and call the mapped decomposition D_m.

3. If interchanging equivalent dimensions of D_u decreases coupling cost then do it.

4. Consider each pair of dimensions, where dimension $D_m.i$ is the dimension of D_m coupled to dimension i D_u and the dimensions are both mapped to the same processor dimension. From the coupling specification for the dimension of D_m we get $D_m.start, D_m.end$, and $D_m.stride$ and for the dimension of D_u we get $D_u.start, D_u.end$, and $D_u.stride$. Since D_m is already mapped we have the $D_m.direction$ and $D_m.start\_proc$ for each dimension. To map dimension i of D_u we must find the direction and the starting processor.

   - The direction for D_u is the same as that of D_m if the stride for the coupling for each decomposition is either increasing or decreasing. If one decomposition's stride is increasing and the other decomposition's stride is decreasing then the direction for D_u is minus the direction for D_m.
   - Determine the offset of the processor which owns the middle entry of the coupling in the given dimension of D_m, call it $D_m.Poffset$.
   - Determine the offset of the processor which owns the middle entry of the coupling in the given dimension of D_u, call it $D_u.Poffset$ (as if D_u was mapped beginning in processor 1).
   - Determine the processor which should own the first element of the current dimension of D_u. It is $D_m.start\_proc[D_m.i]$ + $D_m.Poffset$ - $D_u.Poffset$.

5. For each pair of coupled dimensions that are not mapped to the same processor dimension we find the direction and starting processor for D_u as follows.

   - The direction for D_u in the dimension is positive (this is arbitrary).
   - Find the dimension of D_m that is mapped to the same processor dimension as the current dimension of D_u, label it $D_m.i$.

- Let k be the index in the coupling specification that involves dimension $D_m.i$ of D_m.
- Determine the offset of the processor which owns the middle element of the k^{th} coupling entry for D_m, call it $D_m.Poffset$.
- Determine the offset of the processor which owns the middle entry of the coupling in the current dimension of D_u, call it $D_u.Poffset$ (as if D_u was mapped beginning in processor 1).
- Determine the processor which should own the first element of the current dimension of D_u. It is $D_m.start\_proc[D_m.i] + D_m.Poffset$ - $D_u.Poffset$.

6. For each coupling involving D_u which has not already been put into the heap, insert it.

7. Add D_u to the set MAPPED.

Decomposition	Direction	Starting Processor
A	(1,1,1)	(1,1,1)
B	(1,1,1)	(1,1,1)

Figure 9: 3D-CFD Directions and Starting Processors

Decomposition	Direction	Starting Processor
C	(1,1,1)	(8,1,1)
D	(1,1,1)	(1,1,1)
E	(1,1,1)	(1,1,4)

Figure 10: 2D-Multiblock Directions and Starting Processors

The time for this stage of the algorithm is bounded by the number of distributed dimensions multiplied by $c\ log\ c$, where c is the number of couplings in the system ($c \geq\ number\ of\ meshs - 1$).

Figures 9 and 10 show the directions and starting processors for the 3D-CFD and 2D-Multiblock problems respectively that result from the application of coupling communication reduction.

3.4 Support for High Performance Fortran Programs

To eliminate the need for the programmer to perform tedious and highly error prone portions of the parallelization process, we here specify the program form that we will accept for transformation, what transformations will be performed and what the resulting code will look like. The input program form specifies: the levels of "contains" used; the beginning form of the input file; the basic data structure declarations; the high level structure of the main program and main subroutine; and the parameter passing assumptions. We will point out different options for the user in appropriate portions of the code. We of course, assume that the program must avoid difficult to analyze constructs, such as arbitrary use of pointers and computed gotos. The transformations we will discuss include: additions to the user defined data structures; additions to the input routine; additions to all routines called by the main subroutine that have meshs passed in (TEMPLATE, ALIGN, and DISTRIBUTE statements); and subroutine cloning for each possible alignment order. Finally, we will present the High Performance Fortran program that results from these transformations and discuss some important points that are illustrated by the programs form. Note that the user could write their application in this form and then just use the distribution algorithm for generating distributions for all of the meshs.

3.4.1 Original High Performance Fortran Program

Here we show an example of the coding style that we are advocating. In the HPF code segments, shown in Figures 11 - 13, every line that is continued should have an & in column 73. With this exception, the code is the same as the one that the user should generate for the problem in our validation experiments. The guiding principles in the determining the programming style that should be used for these applications are:

- the programming style should be natural for the application to reduce development and support costs;
- the programming style should be relatively easy to analyze to reduce the cost of compilation and increase the the level of performance the compiler can provide;
- the program structure that the programmer uses should be similar to the program structure that precompiler generates to better support debugging;
- the precompilation should be machine independent to allow its use with the machine dependent compiler for any parallel processor.

```
      subroutine read_meshs_3d()
      read(*,*) Num_3d_meshs
      allocate(Meshs_3d(Num_3d_meshs))
      do i = 1, Num_3d_meshs
         read(*,*) Meshs_3d(i)%size
      end do
      end subroutine read_meshs_3d

      subroutine allocate_3d(Mesh_info)
      type (mesh_3d) Mesh_info
      real, pointer :: all_array(:,:,:)
      allocate(all_array(Mesh_info%size(1),
     &                   Mesh_info%size(2),
     &                   Mesh_info%size(3)))
      Mesh_info%p => all_array
      nullify(all_array)
C     ... repeat allocation for q, u, v, zm, x, y, z
      end subroutine allocate_3d
```

```
      subroutine allocate_all_meshs()
      do i = 1, Num_3d_meshs
         call allocate_3d(Meshs_3d(i))
      end do
      end subroutine allocate_all_meshs

      subroutine update_mesh_3d(Mesh_info,dthlf,dt)
      use physics
      type (mesh_3d) Mesh_info
      real dthlf, dt
C     ... updates for arrays associated with current mesh
      end subroutine update_mesh_3d
```

Figure 11: Original HPF example input and compute routines for 3D meshs.

Figures 11 - 13 outline the form of the code that is to be hand-written by the application programmer. The input/allocation structure shown in Figure 11 could be simplified to into a single routine that reads and allocates, which would then have to be split by the compiler to produce a correct HPF code. We suggest this form instead as then the basic structure of the resulting code that is compiled and run will be similar and hence easier to debug.

A note about the structure of the user defined data structure *mesh_module* and the associated storage allocation. Since HPF does not currently allow distribution of elements of user defined data structures, we have the programmer use a pointer to allocate the arrays and then the structure array is set to point to the correct memory. We can allow the program to use allocatable arrays in the user defined data structure as well, but note that these arrays must be allocatable because the sizes are not available until runtime. Hence we can use the input and allocation as flags to notify the precompiler that these are to be distributed data structures. Here we have the user pass the entire structure for each mesh that is passed to any subroutine. We could allow the user to pass the elements of the structure instaed, but that requires more work from the user, and the compiler, without providing significant advantages.

Finally, the parameters to the HPF specifications must be constant upon entry to subprogram level. Hence the programmer writes a routine, *main_routine*, that does everything except for the input which is done in the main program before calling *main_routine*. The calls to the system clock are only for timing and hence are not required to be part of the program.

Next, consider the coupling specifications and updates. The couplings are read into a user defined data structure that is classified according to the dimensionality of the meshs being used, see Figure 13. If this approach leads to having too many user defined types of couplings, then we could extend the approach to allow the user to have one

```
      module mesh_module
      type mesh_3d
        real, pointer, dimension(:,:,:) :: p, q, u, v, zm
        real, pointer, dimension(:,:,:) :: x, y, z
      end type mesh_3d
      type coupling_3d
        integer id_A, id_B
        integer lo_A(3), hi_A(3), lo_B(3), hi_B(3)
      end type coupling_3d
      type (mesh_3d),allocatable,dimension(:)::Meshs_3d
      integer Num_3d_meshs,Num_3d_couplings
      type (coupling_3d),allocatable,
     &     dimension(:)::Couplings_3d
      contains

      subroutine main_routine
      call allocate_all_meshs
      do i=1, Num_3d_meshs
            call initial_3d(Meshs_3d(i),dthlf,dt)
      end do
      call system_clock(is_count,is_count_rate,
     &                  is_count_max)
      do i_step = 1, Num_steps
         do i=1, Num_3d_meshs
            call update_mesh_3d(Meshs_3d(i),dthlf,dt)
         end do
```

```
c     subroutine main_routine continued

         do i=1, Num_3d_couplings
            id_A = Couplings_3d(i)%id_A
            id_B = Couplings_3d(i)%id_B
            call update_couplings_3d(Meshs_3d(id_A),
     &           Couplings_3d(i),Meshs_3d(id_B),dthlf,dt)
         end do
      end do
      call system_clock(ie_count,ie_count_rate,ie_count_max)
C     ... print results, etc.

C     all subroutines except read_meshs_3d & read_couplings_3d
      end subroutine main_routine

C     subroutines read_meshs_3d and read_couplings_3d go here

      end module mesh_module

      program big_mesh
      use mesh_module
C     The computation/communication reflect the CFD codes
C     at Mississippi State for aerodynamic simulations.
      read(*,*) Num_steps
      call read_meshs_3d()
      call read_couplings_3d()
      call main_routine()
      end program big_mesh
```

Figure 12: Original HPF module, main subroutine and main program.

type and dynamically allocate and read the *hi* and *lo* arrays in the structure according to the dimensionality of the mesh being coupled to. The coupling update routine illustrates the use of dynamic temporaries in the update of the coupled sections of the two input meshs. In this example we have used a simple linear weighting of the coupled sections to update the coupled regions, but there is such restriction intended in user programs. Further it is possible that the user program may have couplings that involve more that two meshs. These cases can be handled in essentially the same way as the two mesh case.

3.4.2 Transformation and Final High Performance Fortran Program

Here we discuss how the example from the previous section is transformed and show what the resulting code looks like. In the HPF code segments, shown in Figures 14 - 16, every line that is continued should have an & in column 73. With this exception, the code is the same as we would generate in a precompiler. Further this is the same as the one that we originally generated by hand for our validation experiments. The reason that we want a precompiler here is that the HPF language definition does not explicitly require the interprocedural analysis that is needed to perform the transformations that we will discuss. For example, interprocedural analysis is needed when specific arrays are passed instead of user defined data structures to verify that all of the arrays are from the same structure. If they are not then they may have a different distribution and that requires more cloning. With the use of a precompiler, which is completely machine independent, we can use *any* HPF compiler that is available for the machine that the user is interested in running on.

In Figure 14 the results of the following transformations can be found. The input routine(*read_mesh_3d*) is modified to read the existing data distribution information from a specific file. The array allocation routine(*allocate_3d*) has templates, alignment and distribution information added. It is then cloned for each possible alignment dimension order. Note that if the programmer had declared all of the arrays in the data structure to be allocatable, but not pointers, then the tranformation would require that they be changed to pointers and the pointers be generated as in the figure. This form is necessary, as HPF does not currently allow distribution of elements of user defined data structures. The routine to allocate all of the meshs in all of the user defined data structures(*allocated_all_meshs*) has a case structure added to select the correct cloned allocation routine for each mesh according to the distribution order read from the input file. The routine to perform mesh

```
      subroutine read_couplings_3d()
      read(*,*) Num_3d_couplings
      allocate(Couplings_3d(Num_3d_couplings))
      do i = 1, Num_3d_couplings
         read(*,*) Couplings_3d(i)%id_A,
     &             Couplings_3d(i)%id_B
         read(*,*) Couplings_3d(i)%lo_A,
     &             Couplings_3d(i)%hi_A,
     &             Couplings_3d(i)%lo_B,
     &             Couplings_3d(i)%hi_B
      end do
      end subroutine read_couplings_3d

      subroutine update_couplings_3d(Mesh_A,Couple,
     &                                Mesh_B,dthlf,dt)
      type (mesh_3d) Mesh_A, Mesh_B
      type (coupling_3d) Couple
      real dthlf,dt
      real p_tmp(Couple%hi_A(1)-Couple%lo_A(1)+1,
     &           Couple%hi_A(2)-Couple%lo_A(2)+1,
     &           Couple%hi_A(3)-Couple%lo_A(3)+1)
      real q_tmp(Couple%hi_A(1)-Couple%lo_A(1)+1,
     &           Couple%hi_A(2)-Couple%lo_A(2)+1,
     &           Couple%hi_A(3)-Couple%lo_A(3)+1)
      real u_tmp(Couple%hi_A(1)-Couple%lo_A(1)+1,
     &           Couple%hi_A(2)-Couple%lo_A(2)+1,
     &           Couple%hi_A(3)-Couple%lo_A(3)+1)
      real v_tmp(Couple%hi_A(1)-Couple%lo_A(1)+1,
     &           Couple%hi_A(2)-Couple%lo_A(2)+1,
     &           Couple%hi_A(3)-Couple%lo_A(3)+1)
      real zm_tmp(Couple%hi_A(1)-Couple%lo_A(1)+1,
     &           Couple%hi_A(2)-Couple%lo_A(2)+1,
     &           Couple%hi_A(3)-Couple%lo_A(3)+1)
      real x_tmp(Couple%hi_A(1)-Couple%lo_A(1)+1,
     &           Couple%hi_A(2)-Couple%lo_A(2)+1,
     &           Couple%hi_A(3)-Couple%lo_A(3)+1)
```

```
C     subroutine update_couplings_3d continued

      real y_tmp(Couple%hi_A(1)-Couple%lo_A(1)+1,
     &           Couple%hi_A(2)-Couple%lo_A(2)+1,
     &           Couple%hi_A(3)-Couple%lo_A(3)+1)
      real z_tmp(Couple%hi_A(1)-Couple%lo_A(1)+1,
     &           Couple%hi_A(2)-Couple%lo_A(2)+1,
     &           Couple%hi_A(3)-Couple%lo_A(3)+1)
      p_tmp(1:Couple%hi_A(1)-Couple%lo_A(1)+1,
     &      1:Couple%hi_A(2)-Couple%lo_A(2)+1,
     &      1:Couple%hi_A(3)-Couple%lo_A(3)+1)
     &    = Mesh_A%p(Couple%lo_A(1):Couple%hi_A(1),
     &               Couple%lo_A(2):Couple%hi_A(2),
     &               Couple%lo_A(3):Couple%hi_A(3))
C     ... save all the temporaries
      Mesh_A%p(Couple%lo_A(1):Couple%hi_A(1),
     &         Couple%lo_A(2):Couple%hi_A(2),
     &         Couple%lo_A(3):Couple%hi_A(3))
     &    = Mesh_B%p(Couple%lo_B(1):Couple%hi_B(1),
     &               Couple%lo_B(2):Couple%hi_B(2),
     &               Couple%lo_B(3):Couple%hi_B(3)) * alpha
     &    + Mesh_A%p(Couple%lo_A(1):Couple%hi_A(1),
     &               Couple%lo_A(2):Couple%hi_A(2),
     &               Couple%lo_A(3):Couple%hi_A(3)) * (1-alpha)
C     ... update all of the ``A'' variables
      Mesh_B%p(Couple%lo_B(1):Couple%hi_B(1),
     &         Couple%lo_B(2):Couple%hi_B(2),
     &         Couple%lo_B(3):Couple%hi_B(3))
     &    = p_tmp(Couple%lo_A(1):Couple%hi_A(1),
     &            Couple%lo_A(2):Couple%hi_A(2),
     &            Couple%lo_A(3):Couple%hi_A(3)) * alpha
     &    + Mesh_B%p(Couple%lo_B(1):Couple%hi_B(1),
     &               Couple%lo_B(2):Couple%hi_B(2),
     &               Couple%lo_B(3):Couple%hi_B(3)) * (1-alpha)
C     ... update all of the ``B'' variables
      end subroutine update_couplings_3d
```

Figure 13: Original HPF example input and update routines for couplings.

updates(*update_mesh_3d*) has parameters added for the sizes of the meshs, and the alignment offsets. These values are used in the HPF statements and must therefore be constants at the subroutine entry. Further, if the user defined structure was passed, then the arrays must be passed explicitly and all of their uses must be modified to explicit array use rather than structure use. This is again a result of the fact that HPF does not allow distribution of elements of user defined types.

In Figure 15 we see that alignment and distribution specifications are added to the user defined data structure and that the number of processors in each dimension are declared globally. Other changes are similar to those already discussed: addition of case statements surrounding calls to ensure that the right alignment and distribution are provided to the compiler and addition of paramenters to routines with distributed meshs. The "contain" approach is necessary to ensure that the number of processors and mesh sizes are constant at the level that alignment and distribution takes place. This is a requirement of the HPF language. By using this two level approach we eliminate the need for recompilation based on the input and only require recompilation when changes are made. Further, this allows the precompiler to be machine independent. The only thing that must be done when the input changes is run the distribution algorithm on the new data set to generate the distribution information in the supplementary input file.

Figure 16 illustrates the coupling update routine(*update_couplings_3d*) after it has had: the necessary parameters added to eliminate user defined data structure references, alignment and distribution specifications added; and cloning performed to ensure correct specification to the HPF compiler.

This section has shown how we transform the user program into a machine independent form that is not dependent on the specific problem topology. The resulting code reads in alignment and distribution specifications

```
      subroutine read_meshs_3d()
      open(unit=8,file='dist.file')
      read(8,*) Num_Proc_i, Num_Proc_j, Num_Proc_k
      read(*,*) Num_3d_meshs
      allocate(Meshs_3d(Num_3d_meshs))
      do i = 1, Num_3d_meshs
         read(*,*) Meshs_3d(i)%size
         read(*,*) Meshs_3d(8)%dist_order
         read(*,*) Meshs_3d(8)%align_offset
      end do
      end subroutine read_meshs_3d

      subroutine allocate_3d_ijk(Mesh_info)
      type (mesh_3d) Mesh_info
!HPF$ template decomp(2*Mesh_info%size(1),
!HPF$&   2*Mesh_info%size(2),2*Mesh_info%size(3))
!HPF$ align all_array(i,j,k) with decomp(
!HPF%      i+Mesh_info%align_offset(1),
!HPF$      j+Mesh_info%align_offset(2),
!HPF$      k+Mesh_info%align_offset(3))
!HPF$ distribute(cyclic(2*Mesh_info%size(1)/Num_Proc_i),
!HPF$&     cyclic(2*Mesh_info%size(2)/Num_Proc_j),
!HPF$&     cyclic(2*Mesh_info%size(3)/Num_Proc_k))
!HPF$&     onto procs::decomp
      real, pointer :: all_array(:,:,:)
      allocate(all_array(Mesh_info%size(1),
     &                   Mesh_info%size(2),
     &                   Mesh_info%size(3)))
      Mesh_info%p => all_array
      nullify(all_array)
C     ... repeat allocation for q, u, v, zm, x, y, z
      end subroutine allocate_3d_ijk
```

```
      subroutine allocate_all_meshs()
      do i = 1, Num_3d_meshs
         select case (Meshs_3d(i)%dist_order)
         case (123)
            call allocate_3d_ijk(Meshs_3d(i))
         case (213)
            call allocate_3d_jik(Meshs_3d(i))
C        ...
         end select
      end do
      end subroutine allocate_all_meshs

      subroutine update_mesh_3d_ijk(i_size,j_size,
     &   k_size,i_off,j_off,k_off,p,q,u,v,zm,x,y,z,dthlf,dt)
      use physics
!HPF$ template decomp(2*i_size,2*j_size,2*k_size)
!HPF$ align (i,j,k) with *decomp(i+i_off,j+j_off,k+k_off)::
!HPF$&                           p,q,u,v,zm,x,y,z
!HPF$ distribute (cyclic(2*i_size/Num_Proc_i),
!HPF$&    cyclic(2*j_size/Num_Proc_j),
!HPF$&    cyclic(2*k_size/Num_Proc_k)) onto procs :: decomp
      real p(:,:,:),u(:,:,:),v(:,:,:),q(:,:,:),zm(:,:,:)
      real x(:,:,:),y(:,:,:),z(:,:,:)
      real dthlf, dt

C     ... updates for arrays associated with current mesh
      end subroutine update_mesh_3d_ijk
```

Figure 14: Transformed HPF example input and compute routines for 3D meshs.

along with the user input. Hence, when the input changes the user only needs to run the distribution algorithm to generate the input alignment and distribution specifications. The code does not have to be recompiled.

3.5 Compiler Support

Given the form of the program that is output by the precompiler, as described in the previous section on HPF program tranformation, we now consider how the program is compiled by an HPF compiler. In order to allow all distributions, the allocation, mesh update and coupling update routines were cloned for each possible ordering of dimensions. A case structure is used in the main routine to ensure that the correct version of the cloned routine is called. The Rice Fortran D compiler uses cloning in a similar manner to optimize communication. Since HPF compilers are not required to do interprocedural analysis and cloning for communication, this cloning procedure is necessary to provide complete alignment and distribution to the HPF compiler in every subroutine with distributed data structures.

Since the end product of the transformations in the precompiler is a standard HPF program, there are no further compiler enhancements necessary for correct code generation (for either a MIMD or SIMD machine). There are two areas where special attention in the compiler may improve performance. These concerns are related to the boundary data exchange associated with the coupling of the regular meshs.

Boundary iterations are those iterations of loops that perform updates to the boundary elements of a mesh. If these iterations are peeled from a loop then the communication associated with coupling can be moved outside of the loop. This implies that there should not be as much time spent waiting to get values from other processors. This same approach is seen in hand parallelized material dynamics calculations. Unfortunately, the recognition of this interaction crosses subroutine boundaries. The compiler must recognize that the iterations for boundaries should be peeled in the routine to update meshs and that the send portion of the coupling communication updates should be lifted out of the coupling update routines and set down in the mesh update routine. Another option

```
      module mesh_module
      type mesh_3d
        integer  dist_order,size(3),align_offset(3)
        real, pointer, dimension(:,:,:) :: p, q, u, v, zm
        real, pointer, dimension(:,:,:) :: x, y, z
      end type mesh_3d
      type coupling_3d
        integer id_A, id_B
        integer lo_A(3), hi_A(3), lo_B(3), hi_B(3)
      end type coupling_3d
      integer Num_Proc_i,Num_Proc_j,Num_Proc_k
      type (mesh_3d),allocatable,dimension(:)::Meshs_3d
      integer Num_3d_meshs,Num_3d_couplings
      type (coupling_3d),allocatable,
     &     dimension(:)::Couplings_3d
      contains
      subroutine main_routine
!HPF$ Processors procs(Num_Proc_i,Num_Proc_j,Num_Proc_k)
      call allocate_all_meshs
      do i=1, Num_3d_meshs
         select case (Meshs_3d(i)%distribution_order)
         case (123)
            call initial_3d_ijk(i,Meshs_3d(i)%size(1),
     &Meshs_3d(i)%size(2),Meshs_3d(i)%size(3),
     &Meshs_3d(i)%align_offset(1),
     &Meshs_3d(i)%align_offset(2),
     &Meshs_3d(i)%align_offset(3),Meshs_3d(i)%p,
     &Meshs_3d(i)%q,Meshs_3d(i)%u,Meshs_3d(i)%v,
     &Meshs_3d(i)%zm,
     &Meshs_3d(i)%x,Meshs_3d(i)%y,Meshs_3d(i)%z,
     &dthlf,dt)
C           ... similar case for each distribution order
         end select
      end do
      call system_clock(is_count,is_count_rate,
     &                  is_count_max)
      do i_step = 1, Num_steps
         do i=1, Num_3d_meshs
            select case (Meshs_3d(i)%distribution_order)
            case (123)
               call update_mesh_3d_ijk(
     &Meshs_3d(i)%size(1),Meshs_3d(i)%size(2),
     &Meshs_3d(i)%size(3),Meshs_3d(i)%align_offset(1),
     &Meshs_3d(i)%align_offset(2),
     &Meshs_3d(i)%align_offset(3),Meshs_3d(i)%p,
     &Meshs_3d(i)%u,Meshs_3d(i)%p,Meshs_3d(i)%v,
     &Meshs_3d(i)%zm,Meshs_3d(i)%x,Meshs_3d(i)%y,
     &Meshs_3d(i)%z,dthlf,dt)
C           ... similar case for each distribution order
            end select
        end do
```

```
c    subroutine main_routine continued
        do i=1, Num_3d_couplings
           id_A = Couplings_3d(i)%id_A
           id_B = Couplings_3d(i)%id_B
         if ((Meshs_3d(id_A)%distribution_order.eq.123).and.
     &       (Meshs_3d(id_B)%distribution_order.eq.123))then
         call update_couplings_3d_ijk_ijk(
     &Meshs_3d(id_A)%size(1),Meshs_3d(id_A)%size(2),
     &Meshs_3d(id_A)%size(3),Meshs_3d(id_A)%align_offset(1),
     &Meshs_3d(id_A)%align_offset(2),
     &Meshs_3d(id_A)%align_offset(3),Meshs_3d(id_A)%p,
     &Meshs_3d(id_A)%q,Meshs_3d(id_A)%u,Meshs_3d(id_A)%v,
     &Meshs_3d(id_A)%zm,Meshs_3d(id_A)%x,
     &Meshs_3d(id_A)%y,Meshs_3d(id_A)%z,
     &Couplings_3d(i)%lo_A(1),Couplings_3d(i)%lo_A(2),
     &Couplings_3d(i)%lo_A(3),Couplings_3d(i)%hi_A(1),
     &Couplings_3d(i)%hi_A(2),Couplings_3d(i)%hi_A(3),
     &Meshs_3d(id_B)%size(1),Meshs_3d(id_B)%size(2),
     &Meshs_3d(id_B)%size(3),Meshs_3d(id_B)%align_offset(1),
     &Meshs_3d(id_B)%align_offset(2),
     &Meshs_3d(id_B)%align_offset(3),Meshs_3d(id_B)%p,
     &Meshs_3d(id_B)%u,Meshs_3d(id_B)%p,Meshs_3d(id_B)%v,
     &Meshs_3d(id_B)%zm,Meshs_3d(id_B)%x,Meshs_3d(id_B)%y,
     &Meshs_3d(id_B)%z,Couplings_3d(i)%lo_B(1),
     &Couplings_3d(i)%lo_B(2),Couplings_3d(i)%lo_B(3),
     &Couplings_3d(i)%hi_B(1),Couplings_3d(i)%hi_B(2),
     &Couplings_3d(i)%hi_B(3),dthlf,dt)
C          ... repeated for each pair of distribution orders
           endif
        end do
      end do
      call system_clock(ie_count,ie_count_rate,ie_count_max)
C     ... print results, etc.

C     all subroutines except read_meshs_3d & read_couplings_3d
      end subroutine main_routine

C     read_meshs_3d and read_couplings_3d go here

      end module mesh_module

      program big_mesh
      use mesh_module
C     The computation/communication reflect the CFD codes
C     at Mississippi State for aerodynamic simulations.
      read(*,*) Num_steps
      call read_meshs_3d()
      call read_couplings_3d()
      call main_routine()
      end program big_mesh
```

Figure 15: Transformed HPF module, main subroutine and main program.

would be a form of optimistic communication. In this case you could use the coupling specifications to insert communication of the coupled elements every time they are updated and then just verify reception of messages in the coupling update routine.

The coupling specification may also be used as explicit an upper bound on the necessary communication by compilers that can not determine more precise communication bounds. This use of the coupling directive is analogous to the use of the independent specification for loops in High Performance Fortran. Although it is not an executable statement, it allows optimization that would be incorrect if the specification is incorrect.

```
      subroutine read_couplings_3d()
      read(*,*) Num_3d_couplings
      allocate(Couplings_3d(Num_3d_couplings))
      do i = 1, Num_3d_couplings
         read(*,*) Couplings_3d(i)%id_A,
     &             Couplings_3d(i)%id_B
         read(*,*) Couplings_3d(i)%lo_A,
     &             Couplings_3d(i)%hi_A,
     &             Couplings_3d(i)%lo_B,
     &             Couplings_3d(i)%hi_B
      end do
      end subroutine read_couplings_3d

      subroutine update_couplings_3d_ijk_ijk(i_A_size,
     &  j_A_size,k_A_size,i_A_off,j_A_off,k_A_off,A_p,
     &  A_q,A_u,A_v,A_zm,A_x,A_y,A_z,i_A_lo,j_A_lo,
     &  k_A_lo,i_A_hi,j_A_hi,k_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(2*i_A_size,2*j_A_size,2*k_A_size)
!HPF$ template decompB(2*i_B_size,2*j_B_size,2*k_B_size)
!HPF$ align (i,j,k) with *decompA(i+i_A_off,j+j_A_off,
!HPF$&     k+k_A_off)::A_p,A_q,A_u,A_v,A_zm,A_x,A_y,A_z
!HPF$ 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 (cyclic(2*i_A_size/Num_Proc_i),
!HPF$&  cyclic(2*j_A_size/Num_Proc_j),
!HPF$&  cyclic(2*k_A_size/Num_Proc_k))onto procs::decompA
```

```
C     subroutine update_couplings_3d_ijk_ijk continued

!HPF$ distribute (cyclic(2*i_B_size/Num_Proc_i),
!HPF$& cyclic(2*j_B_size/Num_Proc_j),
!HPF$& cyclic(2*k_B_size/Num_Proc_k)) onto procs::decompB
      real A_p(:,:,:),A_u(:,:,:),A_v(:,:,:),A_q(:,:,:),
      real A_zm(:,:,:)A_x(:,:,:),A_y(:,:,:),A_z(:,:,:)
      real B_p(:,:,:),B_u(:,:,:),B_v(:,:,:),B_q(:,:,:),
      real B_zm(:,:,:)B_x(:,:,:),B_y(:,:,:),B_z(:,:,:),dthlf,dt
      real p_tmp(i_A_hi-i_A_lo+1,j_A_hi-j_A_lo+1,k_A_hi-k_A_lo+1)
      real v_tmp(i_A_hi-i_A_lo+1,j_A_hi-j_A_lo+1,k_A_hi-k_A_lo+1)
      real q_tmp(i_A_hi-i_A_lo+1,j_A_hi-j_A_lo+1,k_A_hi-k_A_lo+1)
      real u_tmp(i_A_hi-i_A_lo+1,j_A_hi-j_A_lo+1,k_A_hi-k_A_lo+1)
      real zm_tmp(i_A_hi-i_A_lo+1,j_A_hi-j_A_lo+1,k_A_hi-k_A_lo+1)
      real x_tmp(i_A_hi-i_A_lo+1,j_A_hi-j_A_lo+1,k_A_hi-k_A_lo+1)
      real y_tmp(i_A_hi-i_A_lo+1,j_A_hi-j_A_lo+1,k_A_hi-k_A_lo+1)
      real z_tmp(i_A_hi-i_A_lo+1,j_A_hi-j_A_lo+1,k_A_hi-k_A_lo+1)
      p_tmp(1:i_A_hi-i_A_lo+1,1:j_A_hi-j_A_lo+1,1:k_A_hi-k_A_lo+1)
     &    = A_p(i_A_lo:i_A_hi,j_A_lo:j_A_hi,k_A_lo:k_A_hi)
C     ... save all the temporaries
      A_q(i_A_lo:i_A_hi,j_A_lo:j_A_hi,k_A_lo:k_A_hi) =
     &A_q(i_A_lo:i_A_hi,j_A_lo:j_A_hi,k_A_lo:k_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_3d_ijk_ijk
```

Figure 16: Transformed HPF example input and update routines for couplings.

3.6 Limitations -vs- Advantages

This approach to ICRM parallelization is limited to those problems in which it makes sense to distribute every mesh over all processors. This limits the use of this approach for some problems. For example, in many Nuclear Reactor Simulations[2] many of the meshs have few elements ($\sim 10 - 100$) and hence cannot reasonably be distributed over many processors. Further, if enough processors are used even large meshs can not be distributed over all processors. We are currently exploring other approaches for automatic distribution of such problems.

This limitation does not say that every mesh must have many more elements than processors. The real requirement is that for every processor the computation outweighs the communication. Hence, this approach may work even when some of the meshs have about the same number of elements as the number of processors.

Offsetting these limitations are a number of advantages to using this approach. Unlike other currently available approaches, all analysis and communication generation may be performed at compile-time. To do this, we require runtime constant data (coupling information). Compile-time analysis and communication generation may provide a significant savings in time when the same mesh/coupling configuration is used with different initial data sets. For example, the same multiblock representation of an aircraft may be used for simulation of the flow over the aircraft with many different initial conditions. Furthermore, once cloning has been done and the HPF program is generated and compiled, the algorithm can be applied to different configurations without recompilation.

Not only does this approach result in a program that fits the SPMD model, but the code also fits a uniform memory model. Each processor uses (approximately) the same amount of memory for each array and the declarations for each processor are identical. When different decompositions are assigned to different processors, the memory allocation and access are much more complicated. This uniform memory model conformance implies that the approach is applicable on SIMD architectures such as the SNAP-32[1]

[1]Robert Means, Bret Wallach, David Busby, and Robert Lengel Jr. won the 1993 Gordon Bell Prize for price/performance using a SIMD Numerical Array Processor (SNAP) with 32 processors. The price/performance index was 7,554 flops per dollar. This indicates that SIMD machines probably still have a future with some applications.

In some application codes, it may be difficult or even impossible to automatically verify that the computations associated with all of the different meshs can be executed in parallel. In this case even if the decompositions are assigned to different processors their computations will be executed sequentially. The approach presented here does not rely on being able to prove this high level parallelism, but instead relies on the parallelism inside each mesh.

Finally, compiling these ICRM applications with a standard HPF compiler is very straight forward. The only extension needed in a good HPF compiler is careful communication analysis for communication associated with couplings. With the use of the cloning technique described above, the communication analysis is further simplified.

The final and most important advantage is that for the first time the distribution can be found automatically for this class of complex topology problems.

3.7 Fortran D Modification/Generation

In this section we assume that a Fortran D program was provided by the user which included DECOMPOSITION and ALIGN statements for each mesh. This section therefore applies only when there is a fixed topology encoded in the program and not when the topology is part of the input. This limitation is due primarily to the fact that Fortran D does not currently support the *contains* statement and without this statement we can not get the processor, alignment, and distribution specification in the input to be constant on entry to the subroutine entry level. Hence we limit ourselves here to programs with statically encoded topology. Here we discuss the modification of DECOMPOSITION and ALIGN statements and the generation of DISTRIBUTE statements. The final output of these modification/generation steps will be a standard Fortran D program. Since the Fortran D compiler does perform the cloning procedure automatically, we do not need to perform cloning in the precompiler when working with Fortran D. We just modify decomposition and alignment specifications at the levels given by the programmer and add distribution specifications at the same levels. However, since we will be explicitly encoding the alignments and distributions for a particular input, the precompilation must be performed for each new data set. At some point the Fortran D language will probably be extended to include "contains" to eliminate this inefficiency.

3.7.1 Decomposition Modification

We replace the original DECOMPOSITION statements with new ones that reflect the intended mapping of data to processors. To reflect this, we reorder the dimensions according to the mapping. Hence, for each decomposition D the specification

$$\text{DECOMPOSITION } D(size_1, size_2, size_3, ..., size_{D.dimensions})$$

becomes

$$\text{DECOMPOSITION } D(size_{map^{-1}(1)}, size_{map^{-1}(2)}, ...size_{map^{-1}(D.dimensions)})$$

where map^{-1} is the inverse of $D.mapping[SPC] \rightarrow map$.

For the 3D-CFD problem the decomposition declarations that result are:

$$\text{DECOMPOSITION } A(40,40,40), B(40,40,40)$$

For the 2D-Multiblock problem the decomposition declarations that result are:

$$\text{DECOMPOSITION } C(640,320), D(640,320), E(640,320),$$

3.7.2 Alignment Modification

Next we transform ALIGN statements according to the mapping to processors, alignment to processors and the original alignment offsets. For each ALIGN statement associated with a decomposition D

$$\begin{array}{ll} \text{ALIGN} & X(I_1, I_2, I_3, I_4, ..., I_{D.dimensions}) \\ \text{WITH} & D(I_1 + \mathit{offset}_1, I_3 + \mathit{offset}_3, I_2 + \mathit{offset}_2, I_4 + \mathit{offset}_4, ..., I_{D.dimensions} + \mathit{offset}_{D.dimensions}) \end{array}$$

becomes

$$\begin{aligned}
\textsc{align}\quad & X(\textsc{wrap}((1 - D.direction[1]) * (-\tfrac{1}{2}) * D.size[1] + D.direction[1] * (I_1)),\\
& \quad \textsc{wrap}((1 - D.direction[2]) * (-\tfrac{1}{2}) * D.size[2] + D.direction[2] * (I_2)), ...,\\
& \quad \textsc{wrap}((1 - D.direction[D.dimensions]) * (-\tfrac{1}{2}) * D.size[D.dimensions]\\
& \qquad\qquad + D.direction[D.dimensions] * (I_{D.dimensions})))\\
\textsc{with}\quad & D(I_{map^{-1}(1)} + D.elt\_per\_proc[map^{-1}(1)] * (start\_proc[map^{-1}(1)] - 1) + \subset toffset_{map^{-1}(1)},\\
& \quad I_{map^{-1}(3)} + D.elt\_per\_proc[map^{-1}(3)] * (start\_proc[map^{-1}(3)] - 1) + offset_{map^{-1}(3)},\\
& \quad I_{map^{-1}(2)} + D.elt\_per\_proc[map^{-1}(2)] * (start\_proc[map^{-1}(2)] - 1) + offset_{map^{-1}(2)},\\
& \quad I_{map^{-1}(4)} + D.elt\_per\_proc[map^{-1}(4)] * (start\_proc[map^{-1}(4)] - 1) + offset_{map^{-1}(4)}, ...,\\
& \quad I_{map^{-1}(D.dimensions)} + D.elt\_per\_proc[map^{-1}(D.dimensions)]\\
& \qquad * (start\_proc[map^{-1}(D.dimensions)] - 1) + offset_{map^{-1}(D.dimensions)})
\end{aligned}$$

For the 3D-CFD problem the alignment that results is:

$$\begin{aligned}
&\textsc{align}\ wa(wrap(i), wrap(j), wrap(k))\ \textsc{with}\ A(j, k, i)\\
&\textsc{align}\ xa(wrap(i), wrap(j), wrap(k))\ \textsc{with}\ A(j, k, i)\\
&\textsc{align}\ ya(wrap(i), wrap(j), wrap(k))\ \textsc{with}\ A(j, k, i)\\
&\textsc{align}\ za(wrap(i), wrap(j), wrap(k))\ \textsc{with}\ A(j, k, i)\\
&\textsc{align}\ wb(wrap(i), wrap(j), wrap(k))\ \textsc{with}\ B(j, k, i)\\
&\textsc{align}\ xb(wrap(i), wrap(j), wrap(k))\ \textsc{with}\ B(j, k, i)\\
&\textsc{align}\ yb(wrap(i), wrap(j), wrap(k))\ \textsc{with}\ B(j, k, i)\\
&\textsc{align}\ zb(wrap(i), wrap(j), wrap(k))\ \textsc{with}\ B(j, k, i)
\end{aligned}$$

For the 2D-Multiblock problem the alignment that results is:

$$\begin{aligned}
&\textsc{align}\ vc(wrap(i), wrap(j))\ \textsc{with}\ C(i + 80 * 7, j)\\
&\textsc{align}\ wc(wrap(i), wrap(j))\ \textsc{with}\ C(j, i + 80 * 7)\\
&\textsc{align}\ xc(wrap(i), wrap(j))\ \textsc{with}\ C(i + 80 * 7 - 1, j - 1)\\
&\textsc{align}\ yc(wrap(i), wrap(j))\ \textsc{with}\ C(i + 80 * 7 - 1, j + 1)\\
&\textsc{align}\ zc(wrap(i), wrap(j))\ \textsc{with}\ C(i + 80 * 7 + 1, j + 1)\\
&\textsc{align}\ vd(wrap(i), wrap(j))\ \textsc{with}\ D(j, i)\\
&\textsc{align}\ wd(wrap(i), wrap(j))\ \textsc{with}\ D(j, i)\\
&\textsc{align}\ xd(wrap(i), wrap(j))\ \textsc{with}\ D(j, i)\\
&\textsc{align}\ yd(wrap(i), wrap(j))\ \textsc{with}\ D(j, i)\\
&\textsc{align}\ zd(wrap(i), wrap(j))\ \textsc{with}\ D(j, i)\\
&\textsc{align}\ ve(wrap(i), wrap(j))\ \textsc{with}\ E(i, j + 80 * 3)\\
&\textsc{align}\ we(wrap(i), wrap(j))\ \textsc{with}\ E(i, j + 80 * 3)\\
&\textsc{align}\ xe(wrap(i), wrap(j))\ \textsc{with}\ E(i, j + 80 * 3)\\
&\textsc{align}\ ye(wrap(i), wrap(j))\ \textsc{with}\ E(i, j + 80 * 3)\\
&\textsc{align}\ ze(wrap(i), wrap(j))\ \textsc{with}\ E(i, j + 80 * 3)
\end{aligned}$$

Note that in HPF the alignments must be shifted as HPF does not support wrap alignment.

3.7.3 Distribution Generation

Finally, we generate distributions for each decomposition D of the form

$$\textsc{distribute}\ D(\textsc{block}(SPC.procs[1]), \textsc{block}(SPC.procs[2]), ...\textsc{block}(SPC.procs[D.dimensions]))$$

where $SPC.procs[i]$ is set to one for $i > SPC.dimensions$.

For the 3D-CFD problem the distribution that results is:

$$\begin{aligned}
&\textsc{distribute}\ A(\textsc{block}(4), \textsc{block}(4), \textsc{block}(2))\\
&\textsc{distribute}\ B(\textsc{block}(4), \textsc{block}(4), \textsc{block}(2))
\end{aligned}$$

For the 2D-Multiblock problem the distribution that results is:

$$\begin{array}{l} \text{DISTRIBUTE } C(\text{BLOCK}(8), \text{BLOCK}(4)) \\ \text{DISTRIBUTE } D(\text{BLOCK}(8), \text{BLOCK}(4)) \\ \text{DISTRIBUTE } E(\text{BLOCK}(8), \text{BLOCK}(4)) \end{array}$$

4 Validation Results

The two primary goals that we are trying to achieve are: 1) reduce the programmer burden for parallelization of ICRM problems; and 2) provide acceptable performance in the resulting parallelization.

Our test problems are the three problems used at the beginning of the paper to illustrate our application area.

Num. Procs.	runtime	speedup	efficiency
1			
2			
4			
8			
16			
32			

Table 1: **Results from flow through an elbow with 2 vanes.**

The Elbow problem, shown in Figure 1, has five three-dimensional meshs with a total of 275,356 three-dimensional cells. The results of timing experiments for this problem are shown in Table 1[2].

Num. Procs.	runtime	speedup	efficiency
1			
2			
4			
8			
16			
32			

Table 2: **Results from F15e Fuselage-Inlet-Nozzle aerodynamic simulation.**

The Fuselage-Inlet-Nozzle aerodynamic simulation, shown in Figure 2, has ten three-dimensional meshs with a total of 713,766 three-dimensional cells. The results of timing experiments for this problem are shown in Table 2[3].

The full F15e aerodynamic simulation, shown in Figure 3, has 32 three-dimensional meshs with a total of 1,269,845 three-dimensional cells. The results of timing experiments for this problem are shown in Table 3[4]. Our timing results will all be done using the HPF compiler under development at Digital Equipment Corporation. The compiler is currently in pre-release (release is scheduled for 30 June 1994) and hence timings can not yet be released. It appears that the first version of the Digital HPF compiler will require modification of the code shown in our figures. Any significant changes will be documented in the final version of this paper.

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

[2]Note that this table can not be filled in until the Digital HPF compiler release date(June 30).
[3]Note that this table can not be filled in until the Digital HPF compiler release date(June 30).
[4]Note that this table can not be filled in until the Digital HPF compiler release date(June 30).

Num. Procs.	runtime	speedup	efficiency
1			
2			
4			
8			
16			
32			

Table 3: **Results from full F15e aerodynamic simulation.**

distribution, we have described and illustrated a programming style that enhances analyzability. We have also shown how the use of interprocedural analysis can further reduce the programming effort involved in development and support by automatically cloning the routines that require distribution specification for the meshs. This results in a standard HPF program which can be compiled by any complete HPF compiler. 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. When meshs are generated automatically, automatic distribution of large-mesh ICRM problems is critical. This research provides an important first step.

Related work includes development of an algorithm to map ICRM problems in which the meshs must be grouped together for mapping onto parallel processors[11]. Two examples of problems of this type are nuclear reactor simulations and electric circuit simulations.

Future work will include an investigation of how to map ICRM problems in which some or all of the meshs must be mapped to a subset of the processors.

A Coupling Communication Reduction Illustration

To illustrate the coupling communication reduction algorithm let's step through how the directions and starting processors are found in our examples.

In the 3D-CFD computation we first select DECOMPOSITION A (arbitrarily) as D_1. Set the starting processor index to one and the direction to increasing for each dimensions of A. Insert the coupling between A and B into the heap. In step 1, delete the maximum entry from the heap, which is the coupling between A and B. In step 2, $D_u = B$ and $D_m = A$. Since the first coupled pair of dimensions is mapped to the same parallel dimension, we proceed via step 3. The strides are both positive so the direction for the first dimension of $D_u(B)$ is the same as the coupled dimension of $D_m(A)$ which is increasing. Dimension one of D_u is coupled to dimension one of D_m and both are mapped to processor dimension three. Therefore

$$D_m.i = 1, \quad D_u.i = 1$$

and

$$D_m.Poffset = 2, \quad D_u.Poffset = 2.$$

Therefore the starting processor for the first dimension of $D_u(B)$ is:

$$D_u.start\_proc[1] = 1.$$

For the second couple pair of dimensions, we again find they are mapped to the same processor dimension and proceed again with step 3. The strides are both positive so the direction for the first dimension of $D_u(B)$ is the same as the coupled dimension of $D_m(A)$ which is increasing.

$$D_m.i = 2, \quad D_u.i = 2$$

and

$$D_m.Poffset = 4, \quad D_u.Poffset = 4.$$

Therefore the starting processor for the second dimension of $D_u(B)$ is:

$$D_u.start\_proc[1] = 1.$$

For the second couple pair of dimensions, we again find they are mapped to the same processor dimension and proceed again with step 3. The strides are both positive so the direction for the first dimension of $D_u(B)$ is the same as the coupled dimension of $D_m(A)$ which is increasing.

$$D_m.i = 2, \quad D_u.i = 2$$

and

$$D_m.Poffset = 4, \quad D_u.Poffset = 4.$$

Therefore the starting processor for the second dimension of $D_u(B)$ is:

$$D_u.start\_proc[1] = 1.$$

The process is exactly the same for dimension three as for dimension two. There are no new couplings to insert in the heap for B and we find that all decompositions are now fully mapped. This illustrates the simplest case for this algorithm. Next let's consider the slightly more complicated case of the 2D-Multiblock computation..

In the 2D-Multiblock computation, we begin by selecting D as D_1. We set the direction to be increasing and the starting processor to be one for each dimension of $D_1(D)$. Next, insert both couplings involving D_1 into the heap. Deletion of the maximum cost entry (step 1) from the heap yields the coupling between D and C. Therefore, in step 2, $D_u = C$ and $D_m = D$. For the first dimension of C we find that it is coupled with the second dimension of D and that they are both mapped to the first processor dimension. Since $D.direction[1]$ is increasing and the stride (by default) is plus one, the $C.direction[1]$ is set to increasing. Therefore

$$D_m.i = 2, \quad D_u.i = 1;$$

and

$$D_m.Poffset = 1, \quad D_u.Poffset = 8.$$

Hence the starting processor for the first dimension of $D_U(C)$ is:

$$D_u.start\_proc[1] = 8.$$

For the second dimension of D_u, since $D_m.direction[2]$ is increasing and the stride (by default) is plus one, the $D_u.direction[2]$ is set to increasing. Therefore

$$D_m.i = 1, \quad D_u.i = 2;$$

and

$$D_m.Poffset = 3, \quad D_u.Poffset = 3.$$

Hence the starting processor for the second dimension of $D_u(C)$ is:

$$D_u.start\_proc[1] = 1.$$

Decomposition C is now fully mapped and aligned. The coupling between C and E is inserted into the heap. Now we return to step 1. Deletion of the maximum cost entry from the heap yields the coupling between D and E. Therefore, in step 2, $D_u = E$ and $D_m = D$. Then for the first coupling entry

$$D_m.i = 1 \quad and \quad D_u.i = 1.$$

We find that the first dimension of D_u is coupled with the first dimension of D_m and that they are <ins>n</ins>ot both mapped to the same processor dimension. Therefore $D_u.direction[1]$ is set to increasing. Then

$$D_m.i = 2 \quad and \quad k = 2$$

so that

$$D_m.Poffset = 8 \quad and \quad D_u.Poffset = 8.$$

Hence the starting processor for the first dimension of $D_u(E)$ is:

$$D_u.start\_proc[1] = 1.$$

For the second coupling entry

$$D_m.i = 2 \quad and \quad D_u.i = .$$

The second dimension of D_u is coupled with the second dimension of D_m, but they are <u>n</u>ot both mapped to the same processor dimension. Therefore $D_u.direction[1]$ is set to increasing. Then

$$D_m.i = 1 \quad and \quad k = 1$$

so that

$$D_m.Poffset = 1 \quad and \quad D_u.Poffset = 4.$$

Hence the starting processor for the second dimension of $D_u(E)$ is:

$$D_u.start\_proc[1] = 4.$$

Decomposition D_u is now fully mapped and aligned and is added to MAPPED. There are no new couplings to insert in the heap for E. Further, when we return to the top of the loop we find that all of the decompositions are in the set MAPPED and we are done.

References

[1] S.E. Allwright. Techniques in multiblock domain decomposition and surface grid generation. In S. Sengupta, J. Hauser, P.R. Eiseman, and J.F. Thompson, editors, *Numerical Grid Generation in Computational Fluid Mechanics '88*, pages 559–568. Pineridge Press, 1988.

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

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

[4] Louis Comtet. *Advanced Combinatorics.* D. Reidel Publishing Company, Boston, Mass., 1974.

[5] Lars-Erik Eriksson. Flow solution on a dual-block grid around an airplane. *Computer Methods in Applied Mechanics and Engineering*, 64:79–83, 1987.

[6] 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.

[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. In *Submitted to ? 1994*, November 1994.

[9] 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.

[10] L.M. Liebrock and K. Kennedy. Automatic data distribution of small meshs in coupled grid applications. In *Submitted to Concurrency Practice and Experience, 1994*, November 1994.

[11] L.M. Liebrock and K. Kennedy. Modeling parallel computation. Technical Report 94-394, Rice University, Center for Research in Parallel Computation, Houston, TX, April 1994.

[12] 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.

[13] Ivan Niven and H.S. Zuckerman, editors. *An Introduction to the Theory of Numbers.* John Wiley and Sons, New York, NY, fourth edition, 1980.

[14] J.A. Shaw, J.M. Georgala, and N.P. Weatherill. The construction of component adaptive grids for aerodynamic geometries. In S. Sengupta, J. Hauser, P.R. Eiseman, and J.F. Thompson, editors, *Numerical Grid Generation in Computational Fluid Mechanics '88*, pages 383–394. Pineridge Press, 1988.

[15] J.A. Shaw and N.P. Weatherill. Automatic topology generation for multiblock grids. *Applied Mathematics and Computation*, 52:355–388, 1992.

[16] Mark S. Shephard and Marcel K. Georges. Reliability of automatic 3d mesh generation. *Computer Methods in Applied Mechanics and Engineering*, 101:443–462, 1992.

[17] Robert E. Smith and Lars-Erik Eriksson. Algebraic grid generation. *Computer Methods in Applied Mechanics and Engineering*, 64:285–300, 1987.

[18] Joseph L. Steger and John A. Benek. On the use of composite grid schemes in computational aerodynamics. *Computer Methods in Applied Mechanics and Engineering*, 64:301–320, 1987.

[19] J.P. Steinbrenner, S.L. Karmen, and J.R. Chawner. Generation of multiple block grids for arbitrary 3-d geometries. In H. Yoshihara, editor, *3-Dimensional Grid Generation for Complex Configurations.* AGARDograph 309, 1988.

[20] Joe F. Thompson. A general three-dimensional elliptic grid generation system on a composite block structure. *Computer Methods in Applied Mechanics and Engineering*, 64:377–411, 1987.

[21] Michael Thuné. A partitioning algorithm for composite grids. *Parallel Algorithms and Applications*, 1(1):69–81, 1993.

[22] Yukimitsu Yamamoto. Numerical simulation of hypersonic viscous flow for the design of H-II orbiting plane (HOPE). *Computer Methods in Applied Mechanics and Engineering*, 89:59–72, 1990.