Final Report

Retargeting of Existing FORTRAN Program and Development of Parallel Compilers

NASA Contract # NAG 2-449
Dated : Sept. 26, 1988

Submitted to Dr. Ken Stevens
Mail Stop 233-14
NASA Ames Research Center
Moffett Field, CA 94035

PI : Dr. Dharma P. Agrawal, Professor
Computer Systems Lab.
Department of Electrical And Computer Engineering
Box 7911
North Carolina State University
Raleigh, NC 27695-7911
Tel: 919-737-3894

Research Assistant : Sukil Kim

Retargeting of Existing FORTRAN Program and Development of Parallel Compilers

SUMMARY

This report describes the software models used in implementing parallelizing compiler for the B-HIVE multiprocessor system. The various models and strategies used in our compiler development are:

1) **Flexible granularity model**, which allows a compromise between two extreme granularity models;
2) **Communication Model**, which is capable of precisely describing the interprocessor communication timings and patterns;
3) **Loop Type detection strategy**, which identifies different types of loops, such as DOALL, DOACR, and DOSEQ;
4) **Critical Path with Coloring scheme**, which is a versatile scheduling strategy for any multicomputer with some associated communication costs;
5) **Loop Allocation Strategy**, which realizes optimum overlapped operations between computation and communication of the system.

Using these models, several sample routines of AIR3D package are examined and tested. It may be noted that automatically generated codes are highly parallelized to provide maximize degree of parallelism, obtaining the speedup up to 28 on a 32-processor system. A comparison of parallel codes for both existing and proposed communication model, is performed and the corresponding expected speedup factors are obtained. The experimentation shows that the B-HIVE compiler produces more efficient codes than existing techniques.

Working is progressing very well in completing the final phase of the compiler. Numerous enhancements are needed to improve the capabilities of the parallelizing compiler.

CONTENTS

CHAPTER 1

INTRODUCTION

Parallel computation is crucial in shortening the turnaround times for massive scientific computations, such as computer aided design applications, computational fluid dynamics, and weather forecastings. Numerous efforts have been made to increase the speedup for such algorithms on various parallel processors, such as vector processors [SwJ85], shared memory multiprocessors [EBS84], and private memory multiprocessors [Cat87]. The major effort in utilizing multiprocessor systems for computational fluid dynamics lies in the way of increasing the degree of parallel operations and cutting down the turnaround time that is extremely long on a uniprocessor machine. Navier-Stokes algorithm that requires massive computations, is a fundamental simulation model for computational fluid dynamics needed in designing a high speed aircraft.

Architectural innovation in multiprocessors has also encouraged the development of new tools in exploring the capabilities of the parallel machines [Fly72]. Traditionally coded programs are not directly suitable for the new architectures, and usually they must be reorganized or modified to utilize the power of the new machines. For this purpose, either we can develop a parallelizing compiler that transforms a sequential source program into parallel code automatically, as described in [All83, AlK85, FER84, KKP81, PKL80, LAM87, LKA88], or we could provide programmers some sort of parallel programming environment such as seen in new programming languages [Hoa78, Ahu86], and in the extended languages [Han77, KuS85, Sha86] so that the parallelism could be specified manually.

The disadvantages of having new parallel language is the need for a new compiler and the mandatory need for rewriting the whole program all over again for a new machine.

We have concentrated our efforts in retargeting existing software. The advantages of having parallelizing compiler are its "user friendliness" and high reusability. The user friendliness means that a user can get the parallel codes without learning the new parallel language. Reusability implies that the existing programs can be run on a new parallel architecture simply by recompiling the existing source code. The drawbacks of having a parallelizing compiler are the higher compiling cost (due to the incorporation of automatic parallelism detection algorithm), and a somewhat lower degree of parallelism (because of inherently sequential algorithms or programmer's coding styles).

Most past efforts on parallelizing compilers have been concentrated on the loop structures. For example, parallelism in loops is extensively analyzed in [PoB87, PKP86], the granularity considered in [Cve87], and dynamic scheduling covered in [PoK87]. These efforts are mainly concentrated on the parallelizing the codes for a shared memory multiprocessor system in which an interprocessor communication overhead is negligible. In a loosely coupled distributed memory multiprocessor system, however, communication overhead is large so that run time task distribution itself would require excessive amount of time. This limitation basically encouraged the use of the static scheduling in which the task allocation is done before run time.

Navier-Stokes program has been restructured and tested under the static scheduling strategy on a two-VAX 11/780 based shared memory multiprocessor, and the speed-up of 1.9 [EBS84] has been reported. This result is not too encouraging and has forced to look into possible utilization of distributed memory multiprocessors for computational fluid dynamics and other scientific computations. This report covers various strategies that have been employed in building a parallelizing compiler for a distributed memory multiprocessor environment and describes the computation models used for the AIR3D package, a version of Navier-Stokes algorithm provided by the NASA. Only the static scheduling strategy is considered as the parallelizing compiler has been developed for the loosely coupled B-HIVE multiprocessor system [AAG86]. The B-HIVE multiprocessor system is a 24-node generalized hypercube based machine designed and built at the North Carolina State University. Each node in the B-HIVE system consists of a pair of processors, an application processor and a communication processor, which communicate with each other through a fast dual-port shared memory.

In Chapter II, we describe the structure of the B-HIVE parallelizing FORTRAN compiler. This chapter also describes the parallel software model and communication model employed in the parallelizing compiler, and is accompanied with few examples. In Chapter III, we consider several AIR3D routines extensively to show how the proposed software models and strategies are used in the B-HIVE compiler. We also show the parallel codes for several sample routines. Performance simulation and test results on some AIR3D subroutines with proposed model is given in Chapter IV. Finally, the current status of the project is summarized, and the future plans are also included.

CHAPTER 2

B-HIVE PARALLELIZING COMPILER AND SOFTWARE MODELS

B-HIVE parallelizing compiler accepts a sequential code and produce a parallel code. It, automatically and interactively, detects parallelism of the source codes, determines the type of loops, allocates parallel tasks, and finally generate parallel codes. This chapter describes the parallelism detection and utilization strategies used by the B-HIVE parallelizing compiler.

Figure 1 outlines seven phases in the compilation. The front end of the compiler includes lexical and syntactic analysis of the source codes written in FORTRAN. The second phase of the compiler reads syntactically verified source codes and builds a program tree in which nodes and arcs represent tasks and program control flows respectively. Source codes are divided into a set of tasks according to the program's natural boundaries, such as loop bodies, comparison bodies, subroutines, and basic blocks. A basic block includes a sequence of consecutive statements without interior branching or stopping. Within each basic block, the execution order of the statements is then carefully analyzed based on the data dependence relations between every pair of statements. Defining 'weak' dependence relation between sets of statements by a precedence relation, such that allocation of every set of statements onto two different processors does not delay the completion time of a basic block, grouping statements into a set of tasks will improve or at least not worsen the performance if every pair of tasks are weakly dependent on each other. In other words, by grouping "strongly" dependent statements into a task, such that a precedence relation between a pair of the statements delays execution if the two statements of every pair are allocated onto different processors. This property of grouping statements inside basic blocks provides another source of potential parallelism of programs besides the parallel loops.

The potential parallelism of basic blocks is explored in the third phase of the compiling process, forming a hierarchy of tasks based on the data dependence relation between every pair of tasks. In a distributed memory multiprocessor environment, a communication overhead is inevitable so that grouping strongly dependent statements into a task is crucial to eliminate excessive communication overhead. A task is represented by a "grain" which is basically a group of operations. Partitioning a basic block into several grains is characterized as a granularity model. In a fine granularity model, each grain consists of only few operations. For instance, in an extreme case, such as a data flow computation, each grain contains one operation and all grains are concurrently

issued for better utilization of the resources. The drawback of this model is that it tends to initiate a lot of communication and synchronization traffic among the processors. Hence, this model is not appropriate for a distributed memory multiprocessor environment. Another extreme example is a coarse granularity model. Every grain has big chunk of computations, such as procedure level seen in a distributed computing environment wherein the communication overhead could be reduced to the minimum level at the expense of the degree of parallelism [Bab84]. Flexible granularity model is a compromise, resulting in the medium size grains. It begins exploring all potential parallelism of the statements within each basic block, producing a set of fine grains, and then regroups cohesive fine grains into medium grains, thereby decreasing the communication overhead while retaining the parallelism among the grains.

In the fourth phase, every basic block is replaced with a block data dependence graph, and a global data dependence graph of the program tree is built to ensure proper code synchronization throughout the entire program. The information obtained in this phase is turned into communication primitives in the code synthesis phase. The next phase allocates all the tasks onto a limited number of processors. Optimal allocation of m tasks to n processors is a well-known NP-complete problem [Sto77]. Thus, it is desirable to devise a heuristic algorithm which gives a near optimal answer within a reasonable amount of time [Efe82]. The allocation phase takes into account the computation times of the tasks and the communication costs among them. Communication is considered according to the overlapped computation/communication model described below and in [LAM87]. Finally, the synthesis phase reorganizes the sequential codes and allocate them onto a number of processors inserting the communication primitives and ultimately producing a separate (cooperating) program for each processor.

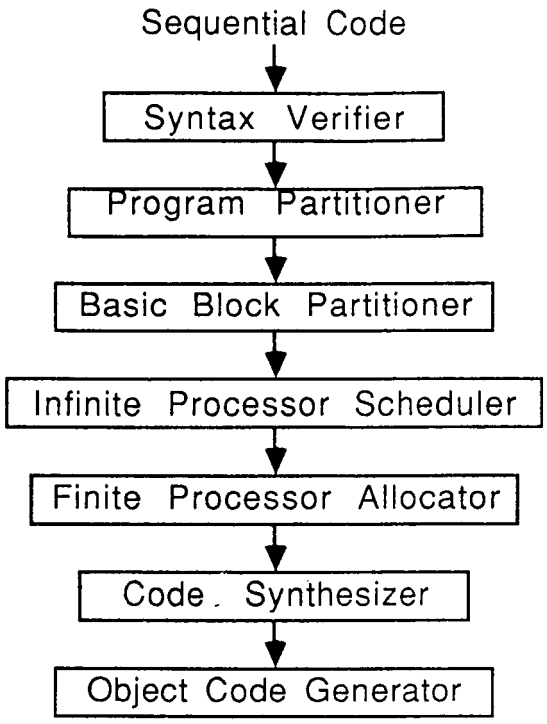


Figure 1. The Structure of the B-HIVE Parallelizing Compiler.

1. Program Division

The program divider builds a program hierarchy tree for a given program into "non-leaf nodes" (that usually has child node) and "leaf node" (that has no child nodes). Control statements, such as loop header, comparison statements are sources of non-leaf nodes, and basic blocks become leaf nodes. For example, every DO-loop is considered as a non-leaf node, and then, every nested loop is represented as a parent-child relation. The algorithm is described in Figure 2. All program hierarchies shown in this report are built by using this algorithm.

input : Sequential Program
output : Program Hierarchy Tree

1. Create root node of the hierarchy tree with the name of the program.
2. **while not** done
   - 2.1. Create a new child node for the current node
   - 2.2. **repeat**
     Put statement into the child node of 2.1
     until boundary statement is encountered.
   - 2.3 Case A: { a starting point of a structure }
     Create a new child node for the current node.
     Current node := new child node of the current node.
   - 2.4. Case B: { an ending point of a structure }
     Current node := parent node of the current node.

Figure 2. Program Division Algorithm

For a given program with n statements, the time complexity of the program division algorithm is $O(n)$, since every statement in the given program is visited exactly once in the algorithm while the time needed to process a statement is constant.

2. Flexible Granularity Model

Basic block partitioning begins with building a "block data dependence graph", in which all potential parallelisms of a basic block is exposed. The problem with a fine grain graph is that it contains too many nodes, and too many communication arcs between nodes. To reduce the communication overhead, cohesive fine grains are grouped into medium grains, decreasing the number of arcs while retaining the parallelism among the grains. Cohesive fine grains are a set of connected nodes in a block dependence graph. In other words, there must be either a direct or an indirect precedence relation between every two nodes.

Three precedence relations are considered in partitioning. The "output dependence" relation occurs when two statements update the same variable. This dependence relation can be easily solved by allocating the statements onto

different processors, where the interference is removed by the use of private memory, or by maintaining the original order of execution if they are allocated onto the same processor. Thus, output dependence is not an obstacle to parallelism. The "anti-dependence" represents the relation that a statement uses a value that the following statement modifies. This relation can be resolved by allocating each task onto different processors in which every processor reads the value into its own private memory before execution. This enables several processors to use and to modify concurrently without any conflicts. The "flow dependence" occurs when a statement produces a value that is used by a successive statement. Whenever a pair of statements that have flow dependence relation are allocated onto different processors, the interprocessor communication is unavoidable, which implies excessive communication overhead. To prevent this, we need a more systematic mechanism to reduce the interprocessor communication by grouping the statements along the flow dependence arc.

A memory aliasing in a distributed memory multiprocessor requires more careful consideration for grain grouping. Assuming array elements $A[i]$ and $A[j]$, such that $i=j$ at run time, it is unknown whether $i=j$ at compile time. For example, given

S_1 i =
S_2 A[i] – ·
S_3 read (j)
S_4 A[j] = ··
S_5 ··· = A[i]

S_5 depends on either S_2 or S_4 since j is unknown at compile time.

Let $T_p : S_i - S_j$ be a task set that has statements S_i and S_j, such that S_i must be preceded by S_j. Then partitioning into two tasks, such as $T_1 : S_1 - S_2 - S_5$ and $T_2 : S_3 - S_4$ causes confliction if T_1 and T_2 reside in the different processors, since S_5 must refer to the processor that has $A[j]$ if $j=i$. Similar situation is observed among the tasks that retain anti dependence relation. To solve this problem, a pair of array elements, that retain any precedence relations should be grouped together. We denote dependence relation among array elements as "array dependence relation" through the paper.

Flexible granularity model performs grain grouping, so as to eliminate excessive communication traffic while retaining all, or at least most, parallelism detected during the data dependence analysis. "Vertical partitioning" provides a way of grouping grains, in which every pair of fine grains has either direct or indirect precedence relations are fused together forming a medium grain. By labeling the dependence arc between every pair of grains with the corresponding communication cost, we can determine every path cost along the arcs.

Assuming as if we allocate the longest path onto a processor, the next longest path to anther processor, and so on, we obtain a number of parallel medium grains that have fine grains on every corresponding path. However, the task sets obtained solely by vertical partitioning do not promise optimum performance, as the intertask synchronization between pairs of tasks some times defer completion time compared to the case that the task pairs are allocated onto the same processor. To avoid this circumstance, we propose a versatile grain grouping algorithm based on "list scheduling" technique.

List schedulings [CoG72, KaN84] are a class of implementable static scheduling methods in which tasks are assigned priorities and placed in a list ordered in descending magnitude of priority. Whenever executable tasks contend for processors, the selection of tasks to be immediately processed is done on the basis of priority with the higher priority tasks executable being assigned processors first. This characteristic of list schedulings tends to evenly distribute tasks over all processors based on the load balancing criterion.

Our flexible granularity model solves both grain grouping and load balancing problems. It begins with finding the longest path task in the vertical partitioning technique. We delete communication overhead within the longest path task, of which the cost implies actual computation time of the task. Then confirm whether other path tasks increase the longest path task cost due to intertask communication between the longest path task and other one. If there is a task that defers the completion time of the longest path task, this task is merged to the longest path task until no more task affects the completion time of the longest path task. During the task merging process, grains are reordered according to the dependence relation between a pair of grains. We name the proposed algorithm "Critical Path with Coloring" (CP/C) algorithm and summarize it in Figure 3.

The advantages of CP/C algorithm are:

1) it can partition a basic block into a set of weakly dependent parallel tasks with $O(n^3)$;
2) it represents a task with two timing values, the earliest finishing time, τ_f and the latest starting time τ_s of the task, and with a color number that represents the task group number;
3) it can simulate the near optimum completion time of a basic block on an infinite number of processors, where the completion time is determined by solely the longest path task cost;
4) in the worst case, a basic block is assumed a task so that it does not allow excessive partitioning.

Consider the example that has several flow dependence relations among statements with arbitrary execution costs, as shown in Figure 4. Flow dependence

relations are identified, such as S_1 S_2 S_3, S_4-S_5 and S_2-S_5. Let dependence relation be replaced with same communication cost, τ_c. If communication cost is enough small to simulate a shared memory multiprocessor architecture, for instance, $\tau_c \leq 400$, task partitioning, such as $T_1 : S_1-S_2-S_3$ and $T_2 : S_4-S_5$ is the optimum way to shorten the completion time. If communication cost simulates a distributed memory architecture, for instance, $\tau_c = 1000$, task partitioning, such as $T_1 : S_1-S_2-S_3-S_5$ and $T_2 : S_4$ is the optimum partitioning. If $\tau_c > 1000$, all statements of the example becomes a single task since any partitionings worsen the result.

---

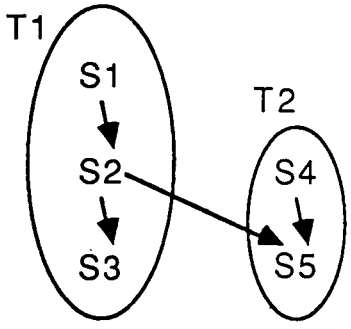
input : Basic Block
output : Parallel Task Set

1. Build Array Dependence Graph, G
2. Append Flow Dependence Arcs of non-array Symbols to G
3. Label Dependence Arcs with Corresponding Communication Cost
4. **while not** empty (G) **do**
4.1. **for** every path of G, G_p **do**
Cost$[p]$ $\sum$*Grain Computation cost* + $\sum$*Communication cost*
4.2. Find the longest path p_{max}
4.3. Eliminate Communication cost from $G_{p_{max}}$
4.4. Put *color* number to every grain of $G_{p_{max}}$
4.5. **while not** finished **do**
Recalculate Cost$[p]$
if task T_i exists, such that T_i mostly affects Cost$[p]$ **then**
merge T_i to $G_{p_{max}}$
else set finished
4.6. Put every grain on $G_{p_{max}}$ into a Task set, T_p
4.7. Compute earliest finishing time and latest starting time of T_p
4.8. Delete $G_{p_{max}}$ from G

Figure 3. Critical Path with Coloring Algorithm.

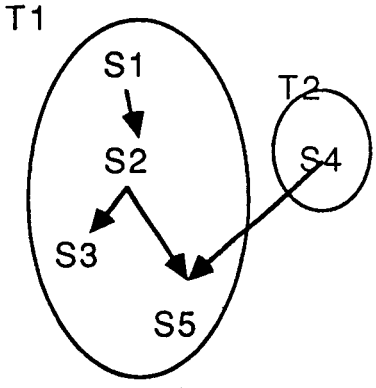
S_1 (1000) X = $\cdots$
S_2 (200) Y = X + $\cdots$
S_3 (500) Z = X * Y * $\cdots$
S_4 (700) V = $\cdots$
S_5 (100) W = Y + V

(a) Basic Block



Task No	Latest Starting Time	Earliest Finishing Time
1	0	1700
2	600	1400

(b) Task Graph ($\tau_c = 100$)



Task No	Latest Starting Time	Earliest Finishing Time
1	0	1800
2	0	700

(c) Task Graph ($\tau_c = 1000$)

Figure 4. Sample Basic block Partitioning based on CP/C Algorithm.

Returning to the allocation problems, we have to allocate a number of task set onto a limited number of processors. Whenever we have enough processors to allocated every task set, we can allocate every task onto different processors as vertical partitioning did. However, if we have a fewer processors than the number of tasks, we have to combine multiple tasks onto a single processor in

an efficient way not to sacrifice the parallelism.

3. Communication Model

The purpose of scheduling is to allocate tasks onto a number of processors so as to achieve the optimum performance of the process. This goal can be accomplished by maximizing the utilization of available processors, while the communication and synchronization activities between processors is kept to a minimum. Intrinsically an efficient computation on a distributed memory multiprocessor is dependent upon not only the degree of parallelism of the programs but the ratio of computation cost over interprocessor communication overhead. The minimizing of interprocessor communication overhead and parallelization are two conflicting objectives in the allocation process. The methods based solely on the first objective tend to utilize only few processors for the sake of reducing communication overhead. On the other hand, methods based solely on parallelization tend to increase the interprocessor communication overhead. One way to compromise these conflicting objectives is to overlap computations and communications in the computation.

The computation model used in [Cam85, CLE80, Lo83, Pat84, SaH86] have assumed that the outgoing precedence data are initiated at the end of the computation, and the incoming data have to be received prior to the computation. We will refer to this view Model A. The assumptions encountered in model A may not maximize the computation / communication overlap when there is more than one operation in the task, as in medium grain tasks. Instead, in the proposed communication model, denoting Model B, all communication activities of each task are accurately represented. The times that a task produces outgoing data and the times it can proceed without the incoming data are labeled in the model. Assuming the communication overhead be lesser than the summation of the computation costs of parallelly executable portions of the tasks, Model B produces better parallel codes than does Model A, as indicated in Figure 5.

Processor 1	Processor 1
X = ...	X = ...
Y = X + ...	Y = X + ...
Z = ...	*send* Y to processor 2
send Y to processor 2	Z = ...
Processor 2	Processor 2
receive Y from processor 1	V = ...
V = ...	*receive* Y from processor 1
W = Y + V	W = Y + V
(a) Model A	(b) Model B

Figure 5. Expected Parallel Codes based on the two Communication models.

In Model *A*, Processor 1 sends Y upon completion of S_3, and Processor 2 receives Y before performing S_4 so that no overlapped operation is allowed. In Model *B*, on the other hand, Y is sent as Processor 1 has updated it and is expected at the moment Processor 2 requests. Thus, Model *B* provides overlapped operations of communication between Processor 1 and Processor 2, and computation of S_3 and S_4.

Possibility of computation / communication overlap is examined by calculation the "benefit factor", which is a cost difference between parallel portions of the task computation costs and the communication costs. If the benefit factor of a pair of tasks is positive, task allocation onto different processors will achieve better performance than that onto the same processor. Overlap operation provides efficient ways of computation not only to a basic block, but also to non-parallelized loop, as DOACR does in [PoB87] and other types of tasks, such as DOALL loops and comparison blocks. We expand overlapped execution to loops along the examples in the next section.

4. Loop Detection and Allocation

Fundamental source of extensive parallelism of programs is the loops. In order to automate parallelism detection of loops, the compiler must look at the data dependence relations in the loops. If a loop contains array data then the data dependencies are somewhat more difficult to solve. If this is the case, an

array dependence relation has to be considered, that is more complex to process than flow dependence relations in the scalar mode. In this chapter we introduce techniques to detect loop types, such as DOALL, DOACR, and DOSEQ. DOALL is a loop such that every iteration has no precedence relations so that every iteration can begin simultaneously. Thus, DOALL loops can be distributed onto processors in an arbitrary manner. DOACR is a loop type such that iterations are weakly dependent so that overlapped operation may shorten the completion time of the loop. DOSEQ type is similar to DOACR type but for in which every iteration is strongly dependent. So distributing iterations of a DOSEQ loop does not necessarily improve the loop performance, and in most cases, they even worsen the performance due to interprocessor communication overhead.

Every loop statement defines loop variable, lower bound, upper bound and step size. Given FORTRAN loop body

```
      DO 10 I = LB, UB, ST
       . . .
10    CONTINUE
```

I,LB,UB and ST be a loop variable, a lower bound, a upper bound, and a step size, respectively. The number of iterations is then given by

$$N = \left\lfloor \frac{UB - LB + 1}{ST} \right\rfloor .$$

Then, the upper bound is normalized as

$$UB_{norm} = \min\left((N-1)\times ST + LB,\ UB\right).$$

We will assume that an upper bound UB, in this discussion, has been normalized for simplicity.

We define "array index vector" be an integer number, that specifies which nested loop directly determines the indices of an array. If a loop index variable is used as an array index within a loop, array index vector of the array index position is set to "1" to indicate the array location is continuously changed per every iteration. We also define "displacement" to be a displacement of the array location in the direction of loop index variable. For the example shown below, D_I and D_J are the displacements for the loop variables I and J, respectively.

```
L1    DO 100 I = 1, 10
L2          DO 100 J = 1, 10
                  A(I+D_I) = B(J+D_J)
```

```
100   CONTINUE
```

Seen in L_2, array index vector of A is set to '0' since the array element of A is unchanged during the execution of L_2. When the loop L_1 is the encountered, the array A's location is dependent upon the loop variable. To identify the location of A is closely related ti the loop variable the array index vector of A is set to '1'. Thus assigning an array index vector to '1' implies that every loop iteration modifies the location of the array element. Similarly, B's index vector to L_1 is '2', that implies the array B's location is modified at the next immediate inner loop bodies. Seen in L_2, B's array vector is set to '1' as is A's array index vector to L_1.

After data dependence relation for the loop body has been constructed and array index vectors and displacements have been computed, the compiler is ready to test the loop types. It may perform some transformations on the source codes to increase parallelism of the loops. So far, the B-HIVE compiler does not include any of these transformations during the compilation, and only can detect loop types for the given structure of the program.

The loop type detection algorithm firstly performs a) DOALL type detection algorithm that isolates DOALL loops, and then in the second pass, b) DOSEQ type detection algorithm that isolates DOSEQ loops from a pool of DOACR loops. Let $Disp(A,k)$ be the displacement of an array A of which k-th dimensional index expression is $expr$, and $Vect(A,Lvar,k)$ be the array index vector of A at k-th dimension for an loop index variable $Lvar$. Using the notations defined above the following relations gives criteria to detect DOALL loops on a distributed memory multiprocessor environment:

1) A loop is DOALL if none of arrays and scalar symbols are used and defined within the loop.
2) A loop is DOALL if $Disp(A_d,k) \leq Disp(A_u,k)$ and $Vect(A_d,Lvar,k) = Vect(A_u,Lvar,k) = 1$ for every pair of A_d and A_u, where A_d is an array element defined and A_u is an array reference.

Loop blocks that do not satisfy above two condition are assumed DOACR, and are left for DOSEQ loop detection. In this report we give an example to test the DOALL conditions instead of verifying the two conditions. The sample loop below is clearly a DOALL according to the condition 1).

```
L1    DO 100 I = 1, 10
            A(I+1) = C(I-1)
100   CONTINUE
```

Consider the following loop body that is known to be parallelizable at a shared memory multiprocessor or pipeline processor environment:

```
L1      DO 100 I = 1, 10
        S1      A(I+1) = C(I)
        S2      B(I) = A(I) × 2
100     CONTINUE
```

L_1 is not DOALL in a distributed memory multiprocessor environment, unsatisfying the condition 2) since

$$Vect(A_d,I,1)=Vect(A_u,I,1)=1$$

and

$$Disp(A_d,1)=1>Disp(A_u,1)=0\ .$$

where A_d is $A(I+1)$ and A_u is $A(I)$. The determination is correct since distributing iterations in a card shuffling fashion encounters communication overhead so as to send $A(I+1)$ to the processor that executes the successive iteration and to receive $A(I)$ from the predecessor.

Loops that do not satisfy the above two conditions are considered as DOACR or DOSEQ type loops. The DOSEQ isolation is performed based on the communication model shown in Section 3. Assuming that every iteration of the loops be distributed evenly onto the processors, a set of variables of an iteration have to be sent to processors that need for successive iterations in them. Let $\tau_u(\Phi)$ be the earliest time that a variable Φ has to be received in processors, and $\tau_d(\Phi)$ be the latest time that a variable Φ is ready to send in a processor. Then the "benefit factor" of distributed computation BF for Φ is

$$BF_\Phi=\tau_u(\Phi)+\tau_{L_i}-\tau_d(\Phi)-\tau_c$$

where τ_{L_i} is the execution cost of the loop body L_i, and τ_c is the communication cost. Thus, in the worst case, the benefit factor, $BF_{\min}$ is

$$BF_{\min}=\min(BF_{\Phi_1},BF_{\Phi_2},\cdots,BF_{\Phi_n})$$

If $BF_{\min}$ is positive, distributing iterations onto processors will provide better performance than allocating the loop on a few processors. Otherwise, allocation on a single processor is preferable. Using the example above, array element $A(*)$ is ready at the time S_2 has done and is received at the time S_1 starts. Thus, $BF_{\min}<0$, and as a result, L_1 is a DOSEQ loop. We can formalize the loop type detection algorithm in Figure 6.

```
input : Loop body
output : Loop Type, either DOALL or DOACR

{ for a given loop body, L_i, split DOALL and DOACR }
Loop_Type (L_i) :=DOALL;
while not done do
 for every symbol Φ, such that Φ_u and Φ_d exist do
 { where Φ_d and Φ_u are a variable Φ defined and used within L_i }
  if Φ is scalar variable then
    Loop_Type(L_i) :=DOACR;
    done := true;
  else { if array then }
    for every dimension, k do
     if ( Vect(Φ_u,L_i var,k)= Vect(Φ_d,L_i var,k)=1)
       and (Disp(Φ_d,k)>Disp(Φ_u,k) then
         Loop_Type(L_i) :=DOACR;
         done := true;
     else { skip }
```

(a) DOALL Type Detection Algorithm

```
input : DOACR loop
output : Loop Type, either DOACR or DOSEQ

{ for a given loop body, L_i, split DOACR and DOSEQ }
while not done do
 for every symbol Φ, such that Φ_d and Φ_u exist do
 { where Φ_d and Φ_u are a variable Φ defined and used within L_i }
   Compute BF(Φ);
   if BF(Φ)≤τ_c then
     Loop_Type(L_i) :=DOSEQ;
     done := true;
   else { continue for next symbol }
```

(b) DOSEQ Type Split Algorithm

Figure 6. Loop Type Detection Algorithm

Parallel loop allocation on a distributed memory multiprocessor should consider a parallelizing overhead caused either by a number of iterations or by a storage allocation. A large number of iterations is preferable as many as possible. If loop bounds are known at compile time, the compiler can determine the possible allocations based on a economic analysis. If a number of iterations of a loop block is quite small, relatively large communication overhead will force the loop block to be allocated to a single processor. If a loop bound is not known at compile time, the economic analysis fails, and one should select possible parallelization in manual or interactively. The B-HIVE compiler assumes unknown loop bounds be enough big to utilize the processors, unless compile directive switch "SEQ" is preceded to a loop block so as to force the loop to a DOSEQ type.

Loop blocks deal with array elements in the loop body, and in most cases, array elements are linearly correspondent to the loop variable. That means the array elements have to be distributed along the loop partitioning strategies before loop execution. In order to prevent unnecessary waiting time to receive the immediately requested variables, sending the variables as soon as they are ready will reduce the waiting time. Similarly after loop execution updated arrays have to be gathered at the compile time known processor for the successive references. We define the location of an array as "origin processor" of the array.

Parallel loop allocations always encounter two communication overheads, before and after a loop execution. For a DOALL loop given

```
L1      DO 100 I = 1, 10
                A(I-1) = B(I+2)
100     CONTINUE
```

B has to be distributed before L_1 begins execution, and A has to be sent back to the origin processor of A after L_1 has finished.

Two iteration distributing strategies are possible. We can distribute every iteration onto processors in a card shuffling fashion, or allocate several contiguous iterations at one processor and then next several onto the others until every iteration is assigned. The first strategy is not powerful in DOALL loop implementation, since the origin processor (assuming all array variables reside at the same processor) and others may have almost same number of iterations although either the origin processor or the other processors do not start at the same time. Assuming that a processor that performs sending variables can do next executions as soon as it initiates send operations, and a processor that initiates receiving operation must wait until the variables are received, the origin processor can share more iterations while the others are receiving variables.

The compensation can be realized under the first strategy, however, the origin processor will have three loops; one for to compensate communication before loop execution, one for after loop execution, and the same number of iterations shared on processors. This will cause difficulty in synthesizing loops.

Another drawback of the first strategy allocating DOALL loops can be found from the array storage usages. For example, consider a sample routine that uses and updates two contiguous array elements, respectively, as indicated below.

```
L1      DO 100 I = 1, 10
              A(I) = B(I)
              A(I-1) = B(I+1)
100     CONTINUE
```

If we use the first strategy, every iteration consumes two disjoint array elements $B(I)$ and $B(I+1)$, and produces two array elements $A(I)$ and $A(I-1)$. Thus, every processor needs 20 array elements. On the other hand, using the latter strategy, each processor needs only 12 array elements since some of array elements are used and updated repeatedly. Consequently, the latter strategy is preferable to DOALL loops. DOACR loops can shorten the completion time of loops to the lowest cost only when every iteration is evenly distributed onto processors, since a loop is DOACR if and only if the loop is not DOALL and the benefit factor of distributed operations is positive. In our implementation the latter strategy is applied to DOALL loops, and the first one is used for DOACR loops.

Consider a DOALL loop L_i, whose loop body execution cost is τ_L. Then the extra iterations that the origin processor has to share for compensation is

$$N_{comp} = \frac{2 \times \tau_c}{\tau_L}$$

and the number of iterations to be shared on processors is

$$N_{dist} = \frac{N - N_{comp}}{Pe}$$

where N is the total number of iterations and Pe is the number of available processors. The total number of iterations that an origin processor should do is $N_{comp} + N_{dist}$. Then the loop bounds at a processor whose identification number is $PEno$ are determined by

$$LB_{PEno} = LB + (N_{comp} + (PEno - PEorg) \times N_{dist}) \times ST$$

and

$$UB_{PEno} = \min(UB, LB_{dist} + (N_{comp} + (PEno - PEorg + 1) \times N_{dist} - 1) \times ST)$$

where $PEorg$ is the identification number of the origin processor, and ST is a step size of the loop.

Communication between a pair of processors is a distinct characteristics of loosely coupled distributed memory multiprocessor architectures. The information is directly passed through either a message passing or a circuit switching. In any cases, a communication channel set up time is relatively expensive compared to the actual communication cost of a unit data transfer. If a large number of data is to transfer, and a new communication channel has to be reinitialized, communication overhead will increase proportional to the number of data to transfer. If we send a block of data through the same channel that was used, we can eliminate the communication channel set up time. To realize block transfer operations, we define two array transfer primitives by

SEND (A($\cdots$ I $\cdots$) I = La, Ua, ST, Pto)

and

RECV (A($\cdots$ I $\cdots$) I = La, Ua, ST, Pfr)

where La and Ua are a lower bound and an upper bound of the block data referring array A, and Pto and Pfr are processor identification numbers to be sent and to be received, respectively. Computing La and Ua are somewhat similar to calculating the loop indices LB, UB. When an origin processor initiates distributing an array A, the two bounds are given by

$$La = LB + \min(Disp(A_{u1}, k), \cdots, Disp(A_{un}, k))$$

and

$$Ua = UB + \max(Disp(A_{u1}, k), \cdots, Disp(A_{un}, k))$$

where A_{ui} is a array referenced in the loop body. When an origin processor collects an array A to restore in it, the two bounds are determined by

$$La = LB + \min(Disp(A_{d1}, k), \cdots, Disp(A_{dm}, k))$$

and

$$Ua = UB + \max(Disp(A_{d1}, k), \cdots, Disp(A_{dm}, k)).$$

Communication primitives must be carefully inserted so as to avoid "dead lock" caused by a non-pair send-receive operations. To prevent dead lock, we add conditional statements to match the pair of communications as

IF (· · ·) THEN
    send or receive primitives
ENDIF

Consider the general format of the expected parallel codes for DOALL loops, as indicated below.

```
        A ( · · · ) = · · ·
                  · · ·
        B ( · · · ) = · · ·
                  · · ·
L_i     DO 100 I = LB, UB, ST
                  · · ·
        S_j       · · · = A ( · · · I · · · )
                  · · ·
        S_k       B( · · · I · · · ) = · · ·
                  · · ·
100     CONTINUE
```

(a) Sequential DOALL Type Loop

```
A ( ·  · ) = · · ·
{ Distribute array A to PE_l through PE_m }
Compute array A's indices, La, Ua for PE_l;
SEND ( A( · · · I · · · ) I = La, Ua, ST, PE_l )
                  · · ·
Compute array A's indices, La, Ua for PE_m;
SEND ( A( · · · I · · · ) I = La, Ua, ST, PE_m )
                  · · ·
B ( · · · ) = · · ·
                  · · ·
{ Start loop }
Calculate LB_org and UB_org;
DO 100 I = LB_org, UB_org, ST
                  · · ·
S_j       · · · = A ( · · · I · · · )
                  · · ·
S_k       B( · · · I · · · ) = · · ·
                  · · ·
```

```
100   CONTINUE
      { Restore Scalar values }
      Calculate PE_last, in which the last iteration is performed;
      RECV ( I, PE_last );
      { Restore Array B by receiving from processors }
      Calculate array B's indices, La, Ua for PE_l;
      RECV ( B( ··· I ··· ) I = La, Ua, ST, PE_l )
           ···
      Calculate array B's indices, La, Ua for PE_m;
      RECV ( B( ··· I ··· ) I = La, Ua, ST, PE_m)
```

(b) Parallel code in an origin processor

```
      Compute LB_dist and UB_dist;
      DO 100 I = LB_dist, UB_dist, ST
           ···
           IF (first iteration) THEN
                Calculate array A's indices, La, Ua;
                RECV ( A( ··· I ··· ) I = La, Ua, ST, PEorg )
           ENDIF
S_j        ··· = A ( ··· I ··· )
           ···
S_k        B( ··· I ··· ) = ···
           ···
100   CONTINUE
      { Pass Updated Scalar to origin processor }
      IF (last iteration) THEN
           SEND ( I, PEorg)
      ENDIF
      { Pass Updated Array to PEorg }
      Calculate array B's indices, La, Ua for origin processor;
      RECV ( B( ··· I ··· ) I = La, Ua, ST, PEorg)
```

(c) Parallel code in non-origin processors

Figure 7. Expected Parallel codes for DOALL Loops.

A boolean expression, *first iteration* ensures one receiving operation during loop iterations by checking whether $I = LB_{dist}$. Similarly, *last iteration* ensures one send operation after completion of the entire loop to restore the scalars that are updated during the iterations.

Several loops sometimes form a nested loops, in which either DOALLs or DOACRs are included. If loop size is not known at compile time, the compiler chooses an outermost DOALL loop as a parallelizable loops. As seen Figure 7, all processors share the iterations for the same amount of time unless the number of iterations is small enough to partition onto few processors. Thus the compiler assumes all processors have their own iterations to do, and this assumption prohibits further parallelization of the loop body that every processor has. Consequently, the loop body of DOALL loops are assumed sequential codes. We will show the actual results of nested DOALL loops in the next chapter.

Iteration distributing strategy in a card shuffling fashion is used for DOACR loop allocation. The expected parallel codes for given DOACR loops are similar to that of DOALL types except the variables that are used at the following iteration are sent to processors they need and that are received from the predecessors before they are needed. During DOACR loop synthesis, the compiler does not need to change the upper bound of the loops. The step size has to be modified to $ST \times Pe$ as does card shuffling, and the lower bound has to be normalized to correct the initial index values in every processor.

Synthesizing the codes for variables that are "reference-only' arrays are the same as DOALL loops, such that reference-only arrays are the ones whose elements are referenced but not redefined in the loop body. Using the notations defined for DOALL loops, we can formalize the expected parallel codes for a given example, as indicated in Figure 8.

```
        A( ··· ) = ···
                ···
L1      DO 100 I = LB, UB, ST
        Sp      ···
        Sn      A( ··· I+DI ··· ) = ···
                ···
        Sm      ··· = A( ··· I ··· )
        Se      ···
100     CONTINUE
```

(a) DOACR loop code

```
        A( ··· ) = ···
                ···
        { Code for reference-only array distribution is identical to
          DOALL loops }
L1      DO 100 I = LB, UB, Pe × ST
        Sp      ···
        Sn      A( ··· I+ DI ··· ) = ···
                IF (NOT last iteration) THEN
                        SEND ( A( ··· I+DI ··· ), mod(PEno + DI))
                END
                ···
                IF (NOT first iteration) THEN
                        RECV ( A( ··· I ··· ), mod(Pe + PEno − DI ))
                ENDIF
        Sm      ··· = A( ··· I ··· )
        Se      ···
100     CONTINUE
        { Restore scalars }
        Calculate PElast, in which the last iteration is performed;
        RECV ( I, PElast )
        { Restore A }
        calculate A 's indices, La, Ua for PEl
        RECV ( A( ··· I ··· ) I=La, Ua ,Pe , PEl )
                ···
        calculate A 's indices, La, Ua for PEm
        RECV ( A( ··· I ··· ) I=La, Ua ,Pe , PEm )
```

(b) Parallel code in an origin processor

```
L1    DO 100 I = LB+ mod (Pe + PEno − PEorg), UB, Pe × ST
Sp    ···
Sn    A( ··· I+DI ··· ) = ···
      IF (NOT last iteration) THEN
            SEND ( A( ··· I ··· ), mod(PEno + DI))
      END
      ···
      IF (first iteration) THEN
            RECV ( A( ··· I ··· ), PEorg )
      ELSE
            RECV ( A( ··· I ··· ), mod(Pe + PEno − DI))
      ENDIF
Sm    ··· = A( ··· I ··· )
Se    ···
100   CONTINUE
      { Restore scalars }
      IF (last iteration) THEN
            SEND ( I, PEorg )
      ENDIF
      { Restore A }
      calculate A's indices, La, Ua for PEno
      SEND ( A( ··· I ··· ) I=La, Ua ,Pe, PEno )
```

(c) Parallel code in non-origin processors

Figure 8. Expected Parallel codes for DOACR Loops.

We will consider an actual DOACR loop routine in the next chapter.

CHAPTER 3

CASE STUDY : *AIR3D*

The AIR3D program [PuS78] is a huge program consisting of a large number of statements with several subroutine calls, numerous variables and many data dependencies between statements ought to be considered. We do not go through the details of routines of the AIR3D program package, since our discussion in this report is to show how a parallelizing compiler can automatically transform sequential codes into parallel versions.

Before doing the actual restructuring or synthesizing the codes, few control flow statements need to be rearranged for simplicity. The AIR3D program extensively uses branch statements in which the condition is determined at run time. Thus, the number of statements to be executed would depend upon the value of the input data. Furthermore, compilers cannot follow the control flows except for simple cases, since in many cases there are cyclic control flows between statements. We therefore replace branch statements into corresponding comparison statements, in which the number of statements within a comparison block is deterministic. This replacement make the speedup analysis easier.

Several branch type blocks exist in the package. Denoting a "forward" branch such that all branch targets always follow branch instructions, and a "backward" branch such that branch targets locate before the branch instruction, we describe a way to restructure branch blocks into corresponding comparison blocks. Common branch structures in AIR3D package are ones having

'IF $(B-expr)$ GOTO S_i'

where S_i is a branch target to jump if $B-expr$ is true. Branch blocks shown in *XXM*, *YYM* and *ZZM* routines use this structure.

A backward branch block, in many cases, simulates a loop, similar to "while" structures in Pascal. However, it is hard to restructure into a loop, since there are many uncertainties to determine the loop types. In the experiments, we do not consider restructuring backward branch blocks. Another branch structure is to use three branch targets as

'IF $(B-expr)$ S_1, S_2, S_3',

where S_1, S_2, S_3 are the branch targets among which one of three locations is active according to $B-expr$ condition. If the instruction is forward branch, it can be restructured similar to 'GOTO' types. We show an example of 'GOTO'

type forward branch restructuring in Figure 9.

```
        IF(B - expr_1) GOTO S_b1
        IF(B - expr_2) GOTO S_b2
        ...
        IF(B - expr_n) GOTO S_bn
S_b0        ...
        GOTO S_bc
S_b1        ...
        GOTO S_bc
S_b2        ...
            .
            .
        GOTO S_bc
S_bn        ...
S_bc    CONTINUE
```

(a) A Branch Block with n Branches

```
        IF(.NOT.(B - expr_1).AND.(.NOT.(B - expr_2).AND. ...
          .AND.(.NOT.(B - expr_n)) THEN
S_b0    ...
        ELSE IF(BS_1) THEN
S_b1    ...
        ELSE IF(BS_2) THEN
S_b2    ...
            .
            .
        ELSE
S_bn    ...
S_bc    ENDIF
```

(b) Corresponding Comparison block restructured from (a)

Figure 9. Forward branch block restructuring.

1. Sample Basic Block : *FLUXVE*

Among several routines of the AIR3D program, *AMATRX*, *DEBUG*, and ***FLUXVE*** are the routines that have few basic blocks (refer to Appendix A). A basic block is a parallelizing source by the flexible granularity model associated with the communication model. We choose ***FLUXVE*** as a sample routine. Applying the vertical partitioning and the associated scheduling algorithm, we can detect parallelism of the routine so that multiprocessor architectures could be utilized.

Subroutine callers pass arguments through a processor that activates callee routines. We assign a processor, say '1' be the origin processor of the caller's arguments. Thus all undefined values referenced are assumed to be in the processor '1'. Consider the actual codes shown in Appendix A-a). J, KL, $R1$, etc of the first statement are are thought to be in PE_1.

Theoretically, all the disjoint medium grain task sets can be allocated to different processors. However, due to a limited number of processors, multiple task set have to be allocated to the same processor. To realize medium grain task allocation, we begin to allocate the longest path task and earliest finishing time task first. During the allocation a "processor time space table" is maintained, which gives the information such as when a processor is ready to accept tasks. The processor finding procedure searches the processor time space table to choose the "best fitting" processor, in which the time space gap becomes minimum. In this way the task can start by the latest starting time of the task or the task can start as soon as possible if it cannot start at its desired time. If there are more than two tasks whose path costs are the same, then the earliest finishing time task is chosen for allocation. If there is a time space, then an unallocated small task can be inserted (and can be finished within the time space), prior to the longest path task allocation.

This strategy simulates a statement reordering, which is a technique to change the order of statements retaining the same result. Since a pair of tasks can be allocated onto different processors until their time boundaries are preserved, allocating small tasks that can finish earlier can be allocated before the task that starts after the small task has been finished. We formalize the basic block allocation algorithm, as below.

1. Sort tasks in descending order of the task cost;
2. **while** task pool is not empty **do**
2.1. Choose the longest task, $T_{p_{max}}$ from the task pool;
2.2. Find the best fitting processor, $P_{p_{max}}$ for $T_{p_{max}}$;
2.3. **while** exist **do**
    Check time space gap of $P_{p_{max}}$;
    Choose the longest task, $T_{p'}$ that can fit in the space gap
        and that can finish within the space gap;
    if $T_{p'}$ exists **then**
        allocate $T_{p'}$ onto $P_{p_{max}}$;
        delete $T_{p'}$ from the task pool;
    else reset exist;
2.4. allocate $T_{p_{max}}$ onto $P_{p_{max}}$;

Figure 10. Basic Block Task Allocation Algorithm.

In the experiments, we use four identical processors to allocate tasks. We can change the number of processors interactively during the synthesis phase. We show the synthesized parallel codes of *FLUXVE* routine in Appendix A-b).

2. Sample DOALL Loop : *XXM*

A parallel loop allocation is a key issues in the loop implementation. As we have described in Chapter 2, DOALL loops are free from finding processors during the allocation since virtually all processors share the loops. Every iteration is shared based on the loop body cost and the communication overhead needed to distribute arrays.

Among several routines in AIR3D package, *XXM*, *YYM*, *ZZM* and *DIFFER* have one or several DOALL loops within the routines. Before restructuring the sample codes, the branch blocks are replaced with comparison blocks, and then loop type detection algorithm determines as is DOALL through the algorithm shown in Figure 6. Using the algorithm of Figure 6, *XXM* is seen to be DOALL type and its loop body cost is 2652 time units.[1] The

[1]Timing results used in this report are the arbitrary values, not of the real machines. The time for one floating point multiplication is assumed *80* unit times.

loop body cost is an average execution cost of one iteration, which realizes all comparison blocks are evenly hit during the execution. Assuming the communication overhead be 1000 time units,[2] the origin processor will execute only one more iteration than non-origin processors since the time space for one iteration provides enough time space for non-origin processors to receive necessary variables and to restore the updated values in the origin processor upon completion of the loop.

The DOALL loop allocation algorithm is given in Figure 11. The algorithm firstly finds the origin processor that will start the first iteration and are responsible for distributing the arrays. The origin processor is easily determined by searching processors that reside the task color number of the loop, where the task color number was assigned during the task partitioning. Then it synthesizes the loop bounds and loop header. Every statement in the loop body is allocated to every processor one by one by adding communication primitives and associated comparison statements. When the loop body encounters the loop label, it begins restoring the arrays and scalars in the origin processor. We show the synthesized parallel code of *XXM*, *YYM*, and *ZZM* in Appendices B, C, and D, respectively.

[2]1000 of communication overhead simulates a hypercube environment, such as the *iPSC* machine. Compared to the cost of multiplication (80), the communication cost is fairly large enough to simulate communications in a loosely coupled distributed multiprocessor environment.

input : DOALL loop
output : Parallel DO Loop

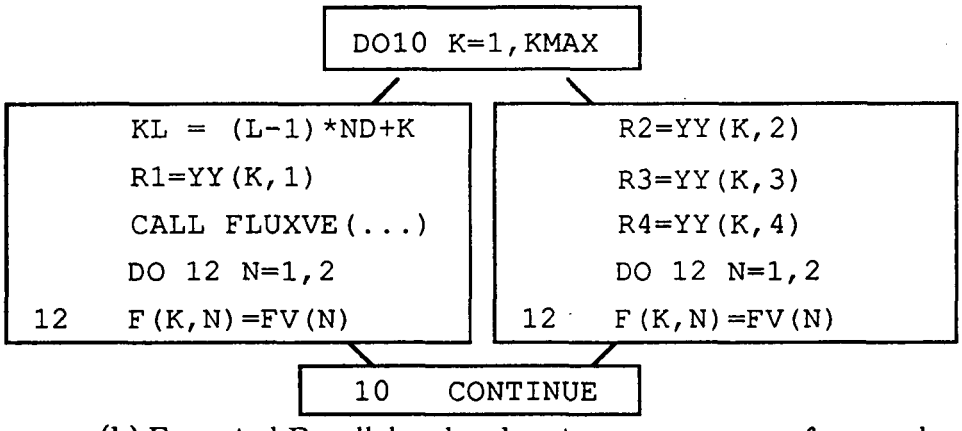
1. Find the origin processor by task *color* number;
2. Append loop bound onto processors;
3. Put 'DO xxx $L_{var} = LB, UB, ST$';
4. **for** every statement, S_i **do**
4.1. **while** *used* variables are not available in a processor *PEno* **do**
if processor is *PEorg* **then**
Put *send* primitives;
if processor is *PEno* **then**
Append 'IF (*first iteration*) THEN';
Put *recv* primitives;
Append 'ENDIF'
4.2. Append S_i;
{ Restore *defined* variables to *PEorg* }
5. **while** *defined* variables exist in a processor *PEno* **do**
5.1 **if** processor is *PEorg* **then**
Put *recv* primitives;
5.2 **if** processor is *PEno* **then**
Append 'IF (*last iteration*) THEN';
Put *send* primitives;
Append 'ENDIF';

Figure 11. DOALL Loop Allocation Algorithm

The drawback of DOALL loop allocation in Figure 11 is that it may not fully utilize the processors if the number of iterations is not large enough to be shared by every processor. If this is the case, then fewer processors will execute iterations although loop body could be parallelized further than that can be detected as per the flexible granularity model). For a partially shown DOALL loop of ***RHS*** routine given below, assume that *KMAX* iterations are distributed onto a ***Pe*** -processor system. With $KMAX = Pe$, each processor performs one iteration. The entire loop body is put into one processor and then, the completion time of the loop would be the loop body cost. However, if $Pe \geq 2 \times KMAX$, every iteration of L_1 can assign *KMAX* processors, and the inner loop body can use another processor to shorten the completion time of the loop body. In this case, better speedup can be achieved than when $KMAX = Pe$.

```
L1      DO 10 K = 1,KMAX
                KL = (L-1)*ND+K
                R1 = YY(K,1)
                R2 = YY(K,2)
                R3 = YY(K,3)
                R4 = YY(K,4)
                CALL FLUXVE(J,KL,R1,R2,R3,R4)
L2              DO 12 N = 1,5
12                    F(K,N) = FV(N)
10      CONTINUE
```

(a) A partial code from *RHS* routine



(b) Expected Parallel code when two processor performs a loop body

Figure 12. More Speedup can be achieved by Loop Body Partitioning.

We can designate processors manually at compile time, allowing a number of processors to execute a loop body together. Several processors then, can share the loop body parallelism. In most cases, however, proposed DOALL loop allocation algorithm is good enough as the number of iterations is generally larger than the number of processors. In this report, we do not consider loop body parallelization.

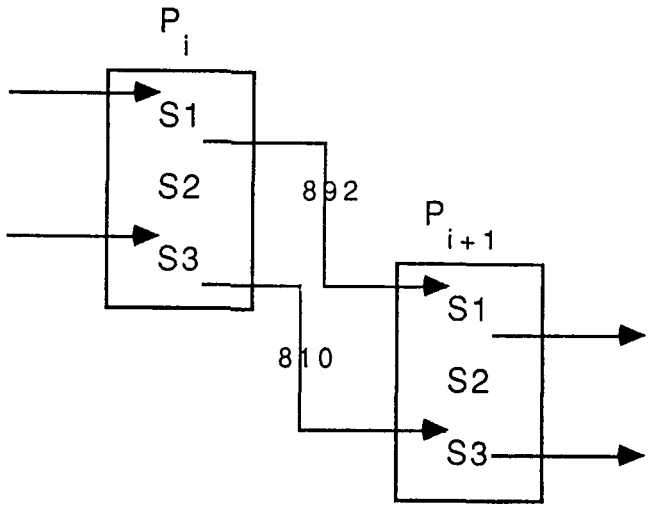
3. Sample DOARC Loop : *GRID*

Let us consider a sample DOACR loop of *GRID* routine. The variables N and array G have to be passed from one processor to the successive processors

that perform the following iterations. Assume the origin processor be PE_1 in the 4-processor system. The processor 1 passes N to the processor 2 upon completion of S_1, and performs S_2 and S_3. Upon completion of S_3 the processor 1 also transfer $G(1,J)$ to the processor 2. While the processor 1 performs S_2 and S_3, the processor 2 has to wait for N to be received from the processor 1. As soon as N has arrived, the processor 2 begins doing S_1. Similarly the processor 2 sends N to the processor 3 upon completion of S_1 and $G(J)$ upon completion of S_3, respectively. By replicating these processes, every iteration is performed in a "software-pipelining" fashion. Software-pipelining operation is advantageous only when the communication overhead is less than the overlapped computation cost. The example given in Figure 13 a) is DOACR while communication cost is less than 810 unit time as seen in Figure 13 b). The expected parallel codes are given in Figure 13 c).

```
L1      DO 912 J = NB11,NB2
( 22)   S1      N = N+1
(788)   S2      GN = (1.0+EPS/SQRT(FLOAT(N+4)))**N
(104)   S3      G(1,J) = G(1,J-1)+DX*GN
912     CONTINUE
```

(a) A Sample DOACR Code from *GRID* routine



(b) Communication between processors

```
C ALLPE IS TOTAL NUMBER OF PROCESSORS
C MYPE IS MY PROCESSOR ID NUMBER
C PEORG IS THE PROCESSOR THAT INITIATES LOOP
C MZZ, LZZ, LQQ ARE TEMP. VALUES
      MZZ = NB2
      LZZ = NB11
      LQQ = MOD (ALLPE + LZZ + MYPE - PEORG)
C BEGIN OF LOOP
      DO 912 J = LQQ,NB2,ALLPE
            IF (J.NE.LZZ) THEN
                  RECV (N, MOD(ALLPE + MYPE -1))
            END
            N = N+1
            IF (J.LT.MZZ) THEN
                  SEND (N, MOD(MYPE + 1))
            ENDIF
            GN = (1.0 +EPS/SQRT(FLOAT(N+4)))**N
            IF (J.NE.LZZ) THEN
                  RECV (G(1,J-1), MOD(ALLPE + MYPE -1))
            END
            G(1,J) = G(1,J-1)+DX*GN
            IF (J.LT.MZZ) THEN
                  SEND (G(1,J), MOD(MYPE + 1))
            ENDIF
912   CONTINUE
```

(c) Expected Parallel codes.

Figure 13. DOACR Loop Allocation and Code Synthesis.

CHAPTER 4

PERFORMANCE EVALUATION

We have implemented the major portion of the B-HIVE parallelizing compiler. The compiler accepts input programs written in FORTRAN, along with the system configuration, produces an allocation scheduling list, and finally produces separate program for each processor. We did not implement a software package to analyze precisely the timing result of the routines. However, the compiler gives timing information during the scheduling phase and the code synthesis phase. Since the static allocation distributes the tasks based on the processor time space table, the time information in the table gives data for precise evaluation. However, it cannot fully support timing issues. For example, DOALL loop allocation algorithm does not consider how many iterations the loop should perform, and it assumes one iteration be assigned onto one processor except the origin processor. Thus, the processor time space table does not project the effect of iteration number a processor should perform. We therefore calculate the timing results in hand, based on the timings of the processor time space table.

We choose a number of AIR3D routines and process as stated in Chapters 1 and 3. The subroutines used in our experiment were *XXM*, *YYM* and *ZZM* for DOALL cases, *FLUXVE* and *AMATRX* for basic blocks, and several other routines are considered for DOACR loops. The parallel codes shown in this report are restructured automatically but for the DOACR routines. Thus, we do not evaluate the timings for the DOACR type routines.

In the experiment, we defined several working environments, setting the number of iterations, the number of processors, and the communication cost. The number of processors is varied, using 4, 16, 32 processors. We put the synthesized parallel codes for the 4-processor system in the Appendices. We assume that the communication cost between any two processors will be the same (disregarding differences due to the computer network and the distance of the two communicating processors). We also vary the communication cost be 1000 and 5000 unit times per word.[1]

[1]Communication cost of 1000 simulates a hypercube MIMD machines, such as the *iPSC* Hypercube. Communication cost of 5000 are used for the comparison purpose.

Parallelizing encounters overhead, such as synchronization, communication overhead, and an extra code execution overhead. Thus the speedup is not always proportional to either the number of processors or the degree of parallelism. The speedup value is calculated from dividing the sequential execution time by the turn around time.

The timing results and the speedup factors for *XXM* are shown in Table I and Table II. The results show that the parallel codes can achieve up to 28 in the 32- processor system. This implies that the transformed DOALL loops are highly parallelized, and as a result, the compiler can transform the DOALL loops effectively. The better speedup achievement is observed when the number of processors and the number of iterations are increased. However, increment of processors does not necessarily increase the speedup when the number of iteration is small. For example, when we increase the number of processors from 16 to 32 for 10 iterations, the speedup values are decreased, as indicated in Table I and II. It is also observed that the speedup values are decreased when the unit communication cost is increased, as expected. We observed the results for *YYM* and *ZZM*. The trends seem quite similar to *XXM*.

The B-HIVE compiler can also improve speedup factors for basic blocks, such as *FLUXVE*, that has several assignment statements only. Normally it is not known to be parallelizable, but the flexible granularity model provides a way to obtain parallelism as we have described. The simulation results when the communication cost be 1000 and 5000 unit times, the speedup factors are 1.1 and 1.0, respectively. The unit speedup at 5000 is expected, since the compiler avoids excessive parallelizing automatically by assigning a whole routine to a single processor. The timing results are given in Table VII.

Table I. The Total Times Needed for Execution of Subroutine *XXM*.

	Number of Iteration (*Comm. cost = 1000*)		
# of PE	10	100	1000
4	11253	69597	666297
16	9365	25277	173789
32	11273	19229	93485

	Number of Iteration (*Comm. cost = 5000*)		
# of PE	10	100	1000
4	16553	74897	671597
16	16822	30082	181246
32	21382	26686	103594

Table II. Speedup Result for *XXM*

	Number of Iteration (*Comm. cost = 1000*)		
# of PE	10	100	1000
4	2.40	3.82	3.98
16	2.88	10.51	15.26
32	2.39	13.82	28.37

	Number of Iteration (*Comm. cost = 5000*)		
# of PE	10	100	1000
4	1.63	3.55	3.95
16	1.60	8.83	14.63
32	1.26	9.96	25.60

Table III. The Total Times Needed for Execution of Subroutine *YYM*.

	Number of Iteration (*Comm. cost = 1000*)		
# of PE	10	100	1000
4	11827	73867	708367
16	9607	26527	184447
32	14167	22627	101587

	Number of Iteration (*Comm. cost = 5000*)		
# of PE	10	100	1000
4	17467	79507	714007
16	18067	32167	192907
32	22627	28267	110047

Table IV. Speedup Result for *YYM*

	Number of Iteration (*Comm. cost = 1000*)		
# of PE	10	100	1000
4	2.40	3.80	3.98
16	2.98	10.65	15.29
32	2.02	12.48	27.76

	Number of Iteration (*Comm. cost = 5000*)		
# of PE	10	100	1000
4	1.64	3.55	3.95
16	1.58	8.78	14.62
32	1.26	9.99	25.63

Table VI. The Total Times Needed for Execution of Subroutine *ZZM*.

	Number of Iteration (*Comm. cost = 1000*)		
# of PE	10	100	1000
4	11557	72057	690807
16	9477	25977	179977
32	14037	22287	99287

	Number of Iteration (*Comm. cost = 5000*)		
# of PE	10	100	1000
4	19807	77557	696307
16	17727	27787	107537
32	22287	27787	107537

Table VI. Speedup Result for *ZZM*

	Number of Iteration (*Comm. cost = 1000*)		
# of PE	10	100	1000
4	2.41	3.82	3.98
16	2.94	10.60	15.28
32	1.99	12.37	27.70

	Number of Iteration (*Comm. cost = 5000*)		
# of PE	10	100	1000
4	1.41	3.55	3.95
16	1.57	8.75	14.61
32	1.25	9.91	25.58

Table VII. The Total Times Needed for Execution of Subroutine *FLUXVE*.

(*Comm. cost = 1000*)		
# of PE	Total Time	Speedup
1	3268	1.0
4	2978	1.1

(*Comm. cost = 5000*)		
# of PE	Total Time	Speedup
1	3268	1.0
4	3268	1.0

CHAPTER 5

CONCLUSION

An approach for parallelizing sequential program has been developed in this work. The vertical partitioning and appropriate scheduling is used in the B-HIVE parallelizing compilers. The performance of the parallelizing models is determined using several routines of AIR3D program package. Especially, the communication overhead is considered to evaluate the task allocation on a distributed memory multiprocessor system. As seen in the parallel code synthesis examples, the B-HIVE compiler can transform the sequential codes into parallel version automatically. The speedup factor is quite close to the number of processors when the number of iterations is large, and the results seems superior to the results in [EBS84]. The current version of the compiler can restructure DOALL loops, basic blocks automatically, and will be updated to restructure the DOACR loops.

Beyond parallelizing, the parallelizing compiler should consider a way to restructure branch instructions, especially the forward branch type codes, automatically. Upon completion of the compiler implementation, we are planning to test the correctness of the compiler for several scientific packages, such as EISPACK and LINPACK. Actual test will also be conducted thereafter.

1. Software Packages Developed

So far, we have implemented several software packages that are used in the various phases of the B-HIVE compiler at North Carolina State University. Following is the list of software packages classified as per the compiling phases.

(1) Phase 1 : FORTRAN Program Syntax Verifier: implementation completed.

(2) Phase 2 : Program Partitioner: implementation completed.

(3) Phase 3 : Basic Block Partitioner: implementation completed.

(4) Phase 4 : Infinite Processor Scheduler.

a) Loop Type Checker has been implemented.
b) Basic Block and DOALL Loop schedulers have been implemented.
c) DOARC and DOSEQ Loop schedulers are under construction.

(5) Phase 5 : Finite Processor Allocator.

a) Basic Block and DOALL Loop Allocator have been implemented.
b) DOACR and DOSEQ Allocators are under construction.

(6) Phase 6 : Code Synthesizer. It has been implemented.

(7) Phase 7 : B-HIVE Coordinator / object code generator. It is under construction based on the communication primitives designed for DOALL loops and Basic blocks.

2. Future Plan

We will continue the implementation of the parallelizing compiler and make it operational on a real machine (B-HIVE). The necessary work to be done during '88 and '89 includes :

(1) Completion of DOARC and DOSEQ loop scheduler associated with allocation strategy development.

(2) Study on subroutine calls and argument passing strategies on a loosely coupled distributed memory multiprocessor environment.

(3) Extensive evaluation of the compiler with testing. Testing will include restructuring of AIR3D package and other scientific packages.

3. Suggestion and Comment

The parallelizing compiler work is a time consuming project. It requires highly advanced techniques in various fields, such as parallel processing, compiler construction, data structure implementation, and programmings, and etc. If additional funds are awarded, we could complete the implementation of the parallel compiler soon, and newer techniques could be developed. With the NASA's support, we have no doubt that our compiler will be the first actual parallel compiler for the loosely coupled distributed memory multiprocessor environment.

REFERENCES

[AAG86] D.P. Agrawal, W.E. Alexander, E.F. Gehringer, R. Mehrotra and J. Mauney, "B-HIVE Project: Present and Future, in Book" *Supercomputers: Algorithms, Architectures and Scientific Computation,* UT Press, Austin, TX 1986, pp. 11-18.

[Ahu86] S. Ahuja, "Linda and Friends," *IEEE Computer,* vol. 19, no. 8, Aug. 1986, pp. 26-34.

[All83] J.R. Allen, "Dependence Analysis For Subscripted Variables and Its Application to Program Transformation," Ph.D. Thesis, 1983, Rice University, Houston, Texas.

[AlK85] J.R. Allen and K. Kennedy, "A Parallel Programming Environment," *IEEE Software,* vol. 2, no. 4, July 1985, pp. 21-29

[Bab84] R.G. Babb II, "Parallel Processing with Large-Grain Data Flow Technique," *IEEE Computer,* vol. 17, no. 7, July 1984, pp. 55-61.

[Cam85] M.L. Campell, "Static Allocation for a Dataflow Multiprocessor," *Proc. of 1985 International Conference on Parallel Processing,* 1985, pp. 511-517.

[Cat87] C.J. Catherasoo, "Separated Flow Simulations using the Vortex method on a Hypercube," *AIAA 8th Computational Fluid Dynamics Conference,* June 9-11, 1987, pp. 81-86.

[CHL80] W.W. Chu, L.J. Holloway, M.T. Lan, and K. Efe, "Task Allocation in distributed Data Processing," *IEEE Computer,* vol. 13, no. 11, Nov. 1980, pp. 57-69.

[CoG72] E.G. Coffman and R.L. Graham "Optimal scheduling for two-processor systems," *Acta Informatica,* Vol.1, No. 3, 1972, pp. 200-213.

[Cve87] Z. Cvetanovic, "The Effects of Problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems," *IEEE Tran. on Computers,* vol. C-36, no. 4, Apr. 1987, pp. 421-432.

[EBS84] D.S. Eberhart, D Baganoff and K.G. Stevens, Jr, "Study of the Mapping of Navier-Stokes Algorithms onto Multiple-Instruction/Multiple-Data Stream Computers," NASA Tech. Memo. 85945, NASA Jul. 1984

[Efe82] Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Computer,* vol. 16, no. 6, June 1982, pp. 50-56.

[FER84] J.A. Fisher, J.R. Ellis, J.C. Ruttenberg, and A. Nicolau, "Parallel Processing: A Smart Compiler and a Dumb Machine," *Proc. of the ACM SIGPLAN '84 Symp. on Compiler Construction,* June 1984, pp. 37-47.

[Fly72] M.J. Flynn, "Some Computer Organizations and their effectiveness," *IEEE Trans. on Computers,* vol. C-21, no. 9, pp. 948-960, September 1972.

[GKL83] D. Grajski, D. Kuck, D. Lawrie, and A. Sameh, "CEDAR: A Large Scale Multiprocessor," *Proc. of the 1983 International Conference on Parallel Processing,* Aug. 1983, pp. 524-529.

[Han77] P.B. Hansen, *The Architecture of Concurrent Programs,* Prentice Hall Inc., 1977.

[Hoa78] C.A.R. Hoare, "Communicating Sequential Processes," *CACM,* Vol. 21, no. 11, Aug. 1978, pp. 666-677.

[KaN84] Hironori Kasahara and Seinosuke Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Trans. on Computers,* Vol. C-33, No. 11, Nov. 1984, pp. 1023-1029.

[KKP81] D.J.Kuck, R.H. Kuhn, D.A., Padua, B. Leaure, and M. Wolfe, "Dependence graphs and Compiler Optimizations," *Proc. of the 8th ACM Symp. on Principles of Programming Languages,* June 1981, pp. 207-218.

[KuS85] J.T. Kuehn and H.J. Siegel, "Extensions to the C Programming Language for SIMD/MIMD Parallelism," *1985 International Conference on Parallel Processing,* August 20-23, pp. 232-235.

[Lo83] V.M. Lo, "Task Assignment in Distributed Systems," Ph.D thesis, Univ. of Illinois, Oct. 1983.

[LAM87] J. Leu, D.P. Agrawal, and J. Mauney, "Modeling of Parallel Software for Efficient Computation-Communication Overlap," *Fall Joint Computer Conference,* Oct. 25-29, 1987, pp. 569-575.

[Pad79] D.A. Padua, "Multiprocessors: Discussion of Some Theoretical and Practical Problems," Ph.D. Thesis, 1979, University of Illinois at Urbana-Champaign.

[PKL80] D.A. Padua, D.J. Kuck, and D.H. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Tran. on Computers,* vol. C-29, no. 9, Sept. 1980, pp. 763-776.

[SwJ85] J.M. Swisshelm and G. M. Johnson, "Numerical Simulation of Three-Dimensional Flowfields Using the Cyber 205," in Supercomputer Application, ed. by R.W. Numrich Plenum Press, New York 1985, pp. 179-195.

[Pat84] G.C. Pathak, "Towards Automated Design of Multicomputer system for Real-time Applications," Ph.D. Thesis, 1984 , North Carolina State University at Raleigh.

[PoB87] C.D. Polychronopoulos and U. Banerjee, "Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds," *IEEE Trans. on Computers,* vol. C-36, no. 4, Apr. 1987, pp. 410-421.

[PoK87] C.D. Polychronopoulos and D.J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Tran. on Computers,* vol. C-36, no. 12, Dec. 1987, pp. 1425-1439.

[PKP86] C.D. Polychronopoulos, D.J. Kuck, and D.A. Padua, "Execution of Parallel Loops on Parallel Processor Systems," *Proc. of 1986 International Conference on Parallel Processing,* Aug. 1986, pp. 519-527.

[Sha86] E. Shapiro, "Concurrent Prolog: A Progress Report," *IEEE Computer,* vol. 19, no. 8, Aug. 1986, pp. 44-58.

[SaH86] V. Sarkar and J. Hennessy, "Compile-time Partitioning and Scheduling of Parallel Programs," *ACM SIGPLAN 86 Symposium on Compiler Construction,* June 23-27 1986, pp. 17-26.

[Sto77] H S Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithm," *IEEE Trans. of Software Eng.,* Vol. SE-3, 1977.

APPENDICES

A: Subroutine *FLUXVE*

B: Subroutine *XXM*

C: Subroutine *YYM*

D: Subroutine *ZZM*

APPENDIX A. SUBROUTINE 'FLUXVE'

a) Sequential Code

```
RR = 1./Q(KL,1,J)
U = Q(KL,2,J)*RR
V = Q(KL,3,J)*RR
W = Q(KL,4,J)*RR
QS = R4+R1*U+R2*V+R3*W
PP = GAMI*(Q(KL,5,J)-.5*Q(KL,1,J)*(U*U+V*V+W*W))
RJ = 1./Q(KL,6,J)
QSINFJ = (R4+R1*UINF+R2*VINF+R3*WINF)*RJ
PINFJ = PINF*RJ
FV(1) = Q(KL,1,J)*QS
FV(2) = Q(KL,2,J)*QS+R1*PP
FV(3) = Q(KL,3,J)*QS+R2*PP
FV(4) = Q(KL,4,J)*QS+R3*PP
FV(5) = (Q(KL,5,J)+PP)*QS-R4*PP
FV(1) = FV(1)-QSINFJ
FV(2) = FV(2)-UINF*QSINFJ-R1*PINFJ
FV(3) = FV(3)-VINF*QSINFJ-R2*PINFJ
FV(4) = FV(4)-WINF*QSINFJ-R3*PINFJ
FV(5) = FV(5)-(EINF+PINF)*QSINFJ+R4*PINFJ
RETURN
END
```

APPENDIX A (continue)

b) Parallel Code

```
FILE     1       {Processor 1}

      CALL SEND (J                                4)
      CALL SEND (KL                               4)
      CALL SEND (R1                               4)
      CALL SEND (R2                               3)
      CALL SEND (J                                3)
      CALL SEND (KL                               3)
      CALL SEND (R3                               2)
      CALL SEND (J                                2)
      CALL SEND (KL                               2)
      RR = 1./Q(KL,1,J)
      CALL SEND (Q                                4)
      CALL SEND (Q                                3)
      CALL SEND (Q                                2)
      CALL SEND (UINF                             4)
      CALL SEND (VINF                             3)
      CALL SEND (WINF                             2)
      U = Q(KL,2,J)*RR
      V = Q(KL,3,J)*RR
      W = Q(KL,4,J)*RR
      QS = R4+R1*U+R2*V+R3*W
      CALL SEND (QS                               4)
      CALL SEND (QS                               3)
      CALL SEND (QS                               2)
      PP = GAMI*(Q(KL,5,J)-.5*Q(KL,1,J)*(U*U+V*V+W*W))
      CALL SEND (PP                               4)
      CALL SEND (PP                               3)
      CALL SEND (PP                               2)
      RJ = 1./Q(KL,6,J)
      QSINFJ = (R4+R1*UINF+R2*VINF+R3*WINF)*RJ
      CALL SEND (QSINFJ                           4)
      CALL SEND (QSINFJ                           3)
      CALL SEND (QSINFJ                           2)
      PINFJ = PINF*RJ
      CALL SEND (PINFJ                            4)
      CALL SEND (PINFJ                            3)
      CALL SEND (PINFJ                            2)
      FV(5) = (Q(KL,5,J)+PP)*QS-R4*PP
      FV(5) = FV(5)-(EINF+PINF)*QSINFJ+R4*PINFJ
      CALL RECV (FV(4)                            2)
      CALL RECV (FV(3)                            3)
      CALL RECV (FV(1)                            4)
      CALL RECV (FV(2)                            4)
      END
```

9

9

```
FILE     2       {Processor 2}

       CALL RECV (PP                                1)
       CALL RECV (R3                                1)
       CALL RECV (QS                                1)
       CALL RECV (J                                 1)
       CALL RECV (KL                                1)
       CALL RECV (Q                                 1)
       FV(4) = Q(KL,4,J)*QS+R3*PP
       CALL RECV (PINFJ                             1)
       CALL RECV (QSINFJ                            1)
       CALL RECV (WINF                              1)
       FV(4) = FV(4)-WINF*QSINFJ-R3*PINFJ
       END
       CALL SEND (FV(4)                             1)
```

```
FILE     3       {processor 3}

       CALL RECV (PP                                1)
       CALL RECV (R2                                1)
       CALL RECV (QS                                1)
       CALL RECV (J                                 1)
       CALL RECV (KL                                1)
       CALL RECV (Q                                 1)
       FV(3) = Q(KL,3,J)*QS+R2*PP
       CALL RECV (PINFJ                             1)
       CALL RECV (QSINFJ                            1)
       CALL RECV (VINF                              1)
       FV(3) = FV(3)-VINF*QSINFJ-R2*PINFJ
       CALL SEND (FV(3)                             1)
       END
```

```
FILE     4      {processor 4}

      CALL RECV (QS                                1)
      CALL RECV (J                                 1)
      CALL RECV (KL                                1)
      CALL RECV (Q                                 1)
      FV(1) = Q(KL,1,J)*QS
      CALL RECV (PP                                1)
      CALL RECV (R1                                1)
      FV(2) = Q(KL,2,J)*QS+R1*PP
      CALL RECV (QSINFJ                            1)
      FV(1) = FV(1)-QSINFJ
      CALL SEND (FV(1)                             1)
      CALL RECV (PINFJ                             1)
      CALL RECV (UINF                              1)
      FV(2) = FV(2)-UINF*QSINFJ-R1*PINFJ
      CALL SEND (FV(2)                             1)
      END
```

APPENDIX B. SUBROUTINE 'XXM'

a) Sequential Code

```
K = M
DY2 = .5/DY1
DZ2 = .5/DZ1
KL = (L-1)*ND+K
KP = KL+1
KR = KL-1
LP = KL+ND
LR = KL-ND
DO 10 J = J1,J2
RJ = Q(KL,6,J)
IF((K.NE.1).AND.(K.NE.KMAX)) THEN
XK = (X(KP,J)-X(KR,J))*DY2
YK = (Y(KP,J)-Y(KR,J))*DY2
ZK = (Z(KP,J)-Z(KR,J))*DY2
ELSE IF(K.EQ.1) THEN
K1 = KL+1
K2 = KL+2
XK = (-3.*X(KL,J)+4.*X(K1,J)-X(K2,J))*DY2
YK = (-3.*Y(KL,J)+4.*Y(K1,J)-Y(K2,J))*DY2
ZK = (-3.*Z(KL,J)+4.*Z(K1,J)-Z(K2,J))*DY2
ELSE
K1 = KL-1
K2 = KL-2
XK = (3.*X(KL,J)-4.*X(K1,J)+X(K2,J))*DY2
YK = (3.*Y(KL,J)-4.*Y(K1,J)+Y(K2,J))*DY2
ZK = (3.*Z(KL,J)-4.*Z(K1,J)+Z(K2,J))*DY2
ENDIF
IF((L.NE.1).AND.(L.NE.LMAX))THEN
XL = (X(LP,J)-X(LR,J))*DZ2
YL = (Y(LP,J)-Y(LR,J))*DZ2
ZL = (Z(LP,J)-Z(LR,J))*DZ2
ELSE IF(L.EQ.1) THEN
L1 = KL+ND
L2 = KL+2*ND
XL = (-3.*X(KL,J)+4.*X(L1,J)-X(L2,J))*DZ2
YL = (-3.*Y(KL,J)+4.*Y(L1,J)-Y(L2,J))*DZ2
ZL = (-3.*Z(KL,J)+4.*Z(L1,J)-Z(L2,J))*DZ2
ELSE
L1 = KL-ND
L2 = KL-2*ND
XL = (3.*X(KL,J)-4.*X(L1,J)+X(L2,J))*DZ2
YL = (3.*Y(KL,J)-4.*Y(L1,J)+Y(L2,J))*DZ2
ZL = (3.*Z(KL,J)-4.*Z(L1,J)+Z(L2,J))*DZ2
ENDIF
XX(J,1) = (YK*ZL-ZK*YL)*RJ
XX(J,2) = (ZK*XL-XK*ZL)*RJ
XX(J,3) = (XK*YL-YK*XL)*RJ
XX(J,4) = -OMEGA*(Z(KL,J)*XX(J,2)-Y(KL,J)*XX(J,3))
```

```
10 CONTINUE
   RETURN
   END
```

APPENDIX B (continue)

b). Parallel Code

```
FILE     1      {Processor 1}

      CALL SEND (J2                          2)
      CALL SEND (J1                          2)
      CALL SEND (J2                          3)
      CALL SEND (J1                          3)
      CALL SEND (J2                          4)
      CALL SEND (J1                          4)
      CALL SEND (L                           2)
      CALL SEND (L                           3)
      CALL SEND (L                           4)
      K = M
      CALL SEND (KMAX                        2)
      CALL SEND (KMAX                        3)
      CALL SEND (KMAX                        4)
      CALL SEND (K                           2)
      CALL SEND (K                           3)
      CALL SEND (K                           4)
      CALL SEND (LMAX                        2)
      CALL SEND (LMAX                        3)
      CALL SEND (LMAX                        4)
      CALL SEND (ND                          2)
      CALL SEND (ND                          3)
      CALL SEND (ND                          4)
      CALL SEND (OMEGA                       2)
      CALL SEND (OMEGA                       3)
      CALL SEND (OMEGA                       4)
      DY2 = .5/DY1
      CALL SEND (DY2                         2)
      CALL SEND (DY2                         3)
      CALL SEND (DY2                         4)
      DZ2 = .5/DZ1
      CALL SEND (DZ2                         2)
      CALL SEND (DZ2                         3)
      CALL SEND (DZ2                         4)
      KL = (L-1)*ND+K
      CALL SEND (KL                          2)
      CALL SEND (KL                          3)
      CALL SEND (KL                          4)
      KP = KL+1
      CALL SEND (KP                          2)
      CALL SEND (KP                          3)
      CALL SEND (KP                          4)
      KR = KL-1
      CALL SEND (KR                          2)
      CALL SEND (KR                          3)
      CALL SEND (KR                          4)
      LP = KL+ND
```

```
CALL SEND (LP                                    2)
CALL SEND (LP                                    3)
CALL SEND (LP                                    4)
LR = KL-ND
CALL SEND (LR                                    2)
CALL SEND (LR                                    3)
CALL SEND (LR                                    4)
NQQ0=(J2-(J1)+1-0) DIV 4+1
LQQ0=J1
MQQ0=MIN(LQQ0+0+1*NQQ0-1,J2)
LZZ=J1+0+NQQ0
MZZ=MIN(LZZ+0+2*NQQ0-1,J2)
CALL SEND(Q(KL,*,KZZ)KZZ=LZZ,MZZ,2)
LZZ=J1+0+2*NQQ0
MZZ=MIN(LZZ+0+3*NQQ0-1,J2)
CALL SEND(Q(KL,*,KZZ)KZZ=LZZ,MZZ,3)
LZZ=J1+0+3*NQQ0
MZZ=MIN(LZZ+0+4*NQQ0-1,J2)
CALL SEND(Q(KL,*,KZZ)KZZ=LZZ,MZZ,4)
LZZ=J1+0+NQQ0
MZZ=MIN(LZZ+0+2*NQQ0-1,J2)
CALL SEND(X(*,KZZ)KZZ=LZZ,MZZ,2)
LZZ=J1+0+2*NQQ0
MZZ=MIN(LZZ+0+3*NQQ0-1,J2)
CALL SEND(X(*,KZZ)KZZ=LZZ,MZZ,3)
LZZ=J1+0+3*NQQ0
MZZ=MIN(LZZ+0+4*NQQ0-1,J2)
CALL SEND(X(*,KZZ)KZZ=LZZ,MZZ,4)
LZZ=J1+0+NQQ0
MZZ=MIN(LZZ+0+2*NQQ0-1,J2)
CALL SEND(Y(*,KZZ)KZZ=LZZ,MZZ,2)
LZZ=J1+0+2*NQQ0
MZZ=MIN(LZZ+0+3*NQQ0-1,J2)
CALL SEND(Y(*,KZZ)KZZ=LZZ,MZZ,3)
LZZ=J1+0+3*NQQ0
MZZ=MIN(LZZ+0+4*NQQ0-1,J2)
CALL SEND(Y(*,KZZ)KZZ=LZZ,MZZ,4)
LZZ=J1+0+NQQ0
MZZ=MIN(LZZ+0+2*NQQ0-1,J2)
CALL SEND(Z(*,KZZ)KZZ=LZZ,MZZ,2)
LZZ=J1+0+2*NQQ0
MZZ=MIN(LZZ+0+3*NQQ0-1,J2)
CALL SEND(Z(*,KZZ)KZZ=LZZ,MZZ,3)
LZZ=J1+0+3*NQQ0
MZZ=MIN(LZZ+0+4*NQQ0-1,J2)
CALL SEND(Z(*,KZZ)KZZ=LZZ,MZZ,4)
DO 10 J=LQQ0,MQQ0
RJ = Q(KL,6,J)
IF((K.NE.1).AND.(K.NE.KMAX)) THEN
XK = (X(KP,J)-X(KR,J))*DY2
YK = (Y(KP,J)-Y(KR,J))*DY2
ZK = (Z(KP,J)-Z(KR,J))*DY2
ELSE IF(K.EQ.1) THEN
```

```
      K1 = KL+1
      K2 = KL+2
      XK = (-3.*X(KL,J)+4.*X(K1,J)-X(K2,J))*DY2
      YK = (-3.*Y(KL,J)+4.*Y(K1,J)-Y(K2,J))*DY2
      ZK = (-3.*Z(KL,J)+4.*Z(K1,J)-Z(K2,J))*DY2
      ELSE
      K1 = KL-1
      K2 = KL-2
      XK = (3.*X(KL,J)-4.*X(K1,J)+X(K2,J))*DY2
      YK = (3.*Y(KL,J)-4.*Y(K1,J)+Y(K2,J))*DY2
      ZK = (3.*Z(KL,J)-4.*Z(K1,J)+Z(K2,J))*DY2
      ENDIF
      IF((L.NE.1).AND.(L.NE.LMAX))THEN
      XL = (X(LP,J)-X(LR,J))*DZ2
      YL = (Y(LP,J)-Y(LR,J))*DZ2
      ZL = (Z(LP,J)-Z(LR,J))*DZ2
      ELSE IF(L.EQ.1) THEN
      L1 = KL+ND
      L2 = KL+2*ND
      XL = (-3.*X(KL,J)+4.*X(L1,J)-X(L2,J))*DZ2
      YL = (-3.*Y(KL,J)+4.*Y(L1,J)-Y(L2,J))*DZ2
      ZL = (-3.*Z(KL,J)+4.*Z(L1,J)-Z(L2,J))*DZ2
      ELSE
      L1 = KL-ND
      L2 = KL-2*ND
      XL = (3.*X(KL,J)-4.*X(L1,J)+X(L2,J))*DZ2
      YL = (3.*Y(KL,J)-4.*Y(L1,J)+Y(L2,J))*DZ2
      ZL = (3.*Z(KL,J)-4.*Z(L1,J)+Z(L2,J))*DZ2
      ENDIF
      XX(J,1) = (YK*ZL-ZK*YL)*RJ
      XX(J,2) = (ZK*XL-XK*ZL)*RJ
      XX(J,3) = (XK*YL-YK*XL)*RJ
      XX(J,4) = -OMEGA*(Z(KL,J)*XX(J,2)-Y(KL,J)*XX(J,3))
   10 CONTINUE
C ----------------------------
      CALL RECV (J                                1)
      CALL RECV (RJ                               1)
      CALL RECV (XK                               1)
      CALL RECV (YK                               1)
      CALL RECV (ZK                               1)
      CALL RECV (K1                               1)
      CALL RECV (K2                               1)
      CALL RECV (XL                               1)
      CALL RECV (YL                               1)
      CALL RECV (ZL                               1)
      CALL RECV (L1                               1)
      CALL RECV (L2                               1)
      KQQ=J2
      LZZ=J1+0+NQQ0
      MZZ=MIN(LZZ+0+2*NQQ0-1,J2)
      CALL RECV (XX(KZZ,*)KZZ=LZZ,MZZ,2)
      LZZ=J1+0+2*NQQ0
      MZZ=MIN(LZZ+0+3*NQQ0-1,J2)
```

```
CALL RECV (XX(KZZ,*)KZZ=LZZ,MZZ,3)
LZZ=J1+0+3*NQQ0
MZZ=MIN(LZZ+0+4*NQQ0-1,J2)
CALL RECV (XX(KZZ,*)KZZ=LZZ,MZZ,4)
END
```

```
FILE     2      {Processor 2}

      CALL RECV (J2                                1)
      CALL RECV (J1                                1)
      NQQ0=(J2-(J1)+1-0) DIV 4+1
      LQQ0=J1+0+NQQ0
      MQQ0=MIN(LQQ0+0+2*NQQ0-1,J2)
      DO 10 J=LQQ0,MQQ0
      IF (J .EQ. LQQ0) THEN
      CALL RECV (KL                                1)
C -----------------------------
      CALL RECV(Q(KL,*,KZZ)KZZ=LQQ0,MQQ0,1)
      ENDIF
      RJ = Q(KL,6,J)
      IF (J .EQ. LQQ0) THEN
      CALL RECV (KMAX                              1)
      CALL RECV (K                                 1)
C -----------------------------
      ENDIF
      IF((K.NE.1).AND.(K.NE.KMAX)) THEN
      IF (J .EQ. LQQ0) THEN
      CALL RECV (DY2                               1)
      CALL RECV (KR                                1)
C -----------------------------
      CALL RECV (KP                                1)
      CALL RECV(X(*,KZZ)KZZ=LQQ0,MQQ0,1)
      ENDIF
      XK = (X(KP,J)-X(KR,J))*DY2
      IF (J .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Y(*,KZZ)KZZ=LQQ0,MQQ0,1)
      ENDIF
      YK = (Y(KP,J)-Y(KR,J))*DY2
      IF (J .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Z(*,KZZ)KZZ=LQQ0,MQQ0,1)
      ENDIF
      ZK = (Z(KP,J)-Z(KR,J))*DY2
      ELSE IF(K.EQ.1) THEN
      K1 = KL+1
      K2 = KL+2
      XK = (-3.*X(KL,J)+4.*X(K1,J)-X(K2,J))*DY2
      YK = (-3.*Y(KL,J)+4.*Y(K1,J)-Y(K2,J))*DY2
      ZK = (-3.*Z(KL,J)+4.*Z(K1,J)-Z(K2,J))*DY2
      ELSE
      K1 = KL-1
      K2 = KL-2
      XK = (3.*X(KL,J)-4.*X(K1,J)+X(K2,J))*DY2
      YK = (3.*Y(KL,J)-4.*Y(K1,J)+Y(K2,J))*DY2
      ZK = (3.*Z(KL,J)-4.*Z(K1,J)+Z(K2,J))*DY2
      ENDIF
      IF (J .EQ. LQQ0) THEN
      CALL RECV (LMAX                              1)
```

```
      CALL RECV (L                                1)
C -----------------------------
      ENDIF
      IF((L.NE.1).AND.(L.NE.LMAX))THEN
      IF (J .EQ. LQQ0) THEN
      CALL RECV (DZ2                              1)
      CALL RECV (LR                               1)
      CALL RECV (LP                               1)
C -----------------------------
      ENDIF
      XL = (X(LP,J)-X(LR,J))*DZ2
      YL = (Y(LP,J)-Y(LR,J))*DZ2
      ZL = (Z(LP,J)-Z(LR,J))*DZ2
      ELSE IF(L.EQ.1) THEN
      IF (J .EQ. LQQ0) THEN
      CALL RECV (ND                               1)
C -----------------------------
      ENDIF
      L1 = KL+ND
      L2 = KL+2*ND
      XL = (-3.*X(KL,J)+4.*X(L1,J)-X(L2,J))*DZ2
      YL = (-3.*Y(KL,J)+4.*Y(L1,J)-Y(L2,J))*DZ2
      ZL = (-3.*Z(KL,J)+4.*Z(L1,J)-Z(L2,J))*DZ2
      ELSE
      L1 = KL-ND
      L2 = KL-2*ND
      XL = (3.*X(KL,J)-4.*X(L1,J)+X(L2,J))*DZ2
      YL = (3.*Y(KL,J)-4.*Y(L1,J)+Y(L2,J))*DZ2
      ZL = (3.*Z(KL,J)-4.*Z(L1,J)+Z(L2,J))*DZ2
      ENDIF
      XX(J,1) = (YK*ZL-ZK*YL)*RJ
      XX(J,2) = (ZK*XL-XK*ZL)*RJ
      XX(J,3) = (XK*YL-YK*XL)*RJ
      IF (J .EQ. LQQ0) THEN
      CALL RECV (OMEGA                            1)
C -----------------------------
      ENDIF
      XX(J,4) = -OMEGA*(Z(KL,J)*XX(J,2)-Y(KL,J)*XX(J,3))
   10 CONTINUE
      KQQ=J2
      IF (J .EQ. KQQ) THEN
      CALL SEND (J                                1)
      CALL SEND (RJ                               1)
      CALL SEND (XK                               1)
      CALL SEND (YK                               1)
      CALL SEND (ZK                               1)
      CALL SEND (K1                               1)
      CALL SEND (K2                               1)
      CALL SEND (XL                               1)
      CALL SEND (YL                               1)
      CALL SEND (ZL                               1)
      CALL SEND (L1                               1)
      CALL SEND (L2                               1)
```

```
      ENDIF
C -----------------------------
      CALL SEND(XX(KZZ,*)KZZ=LQQ0,MQQ0,1)
      END
```

```
FILE     3      {Processor 3}

      CALL RECV (J2                                  1)
      CALL RECV (J1                                  1)
      NQQ0=(J2-(J1)+1-0) DIV 4+1
      LQQ0=J1+0+2*NQQ0
      MQQ0=MIN(LQQ0+0+3*NQQ0-1,J2)
      DO 10 J=LQQ0,MQQ0
      IF (J .EQ. LQQ0) THEN
      CALL RECV (KL                                  1)
C -----------------------------
      CALL RECV(Q(KL,*,KZZ)KZZ=LQQ0,MQQ0,1)
      ENDIF
      RJ = Q(KL,6,J)
      IF (J .EQ. LQQ0) THEN
      CALL RECV (KMAX                                1)
      CALL RECV (K                                   1)
C -----------------------------
      ENDIF
      IF((K.NE.1).AND.(K.NE.KMAX)) THEN
      IF (J .EQ. LQQ0) THEN
      CALL RECV (DY2                                 1)
      CALL RECV (KR                                  1)
C -----------------------------
      CALL RECV (KP                                  1)
      CALL RECV(X(*,KZZ)KZZ=LQQ0,MQQ0,1)
      ENDIF
      XK = (X(KP,J)-X(KR,J))*DY2
      IF (J .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Y(*,KZZ)KZZ=LQQ0,MQQ0,1)
      ENDIF
      YK = (Y(KP,J)-Y(KR,J))*DY2
      IF (J .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Z(*,KZZ)KZZ=LQQ0,MQQ0,1)
      ENDIF
      ZK = (Z(KP,J)-Z(KR,J))*DY2
      ELSE IF(K.EQ.1) THEN
      K1 = KL+1
      K2 = KL+2
      XK = (-3.*X(KL,J)+4.*X(K1,J)-X(K2,J))*DY2
      YK = (-3.*Y(KL,J)+4.*Y(K1,J)-Y(K2,J))*DY2
      ZK = (-3.*Z(KL,J)+4.*Z(K1,J)-Z(K2,J))*DY2
      ELSE
      K1 = KL-1
      K2 = KL-2
      XK = (3.*X(KL,J)-4.*X(K1,J)+X(K2,J))*DY2
      YK = (3.*Y(KL,J)-4.*Y(K1,J)+Y(K2,J))*DY2
      ZK = (3.*Z(KL,J)-4.*Z(K1,J)+Z(K2,J))*DY2
      ENDIF
      IF (J .EQ. LQQ0) THEN
      CALL RECV (LMAX                                1)
```

```
      CALL RECV (L                                1)
C ----------------------------
      ENDIF
      IF((L.NE.1).AND.(L.NE.LMAX))THEN
      IF (J .EQ. LQQ0) THEN
      CALL RECV (DZ2                              1)
      CALL RECV (LR                               1)
      CALL RECV (LP                               1)
C ----------------------------
      ENDIF
      XL = (X(LP,J)-X(LR,J))*DZ2
      YL = (Y(LP,J)-Y(LR,J))*DZ2
      ZL = (Z(LP,J)-Z(LR,J))*DZ2
      ELSE IF(L.EQ.1) THEN
      IF (J .EQ. LQQ0) THEN
      CALL RECV (ND                               1)
C ----------------------------
      ENDIF
      L1 = KL+ND
      L2 = KL+2*ND
      XL = (-3.*X(KL,J)+4.*X(L1,J)-X(L2,J))*DZ2
      YL = (-3.*Y(KL,J)+4.*Y(L1,J)-Y(L2,J))*DZ2
      ZL = (-3.*Z(KL,J)+4.*Z(L1,J)-Z(L2,J))*DZ2
      ELSE
      L1 = KL-ND
      L2 = KL-2*ND
      XL = (3.*X(KL,J)-4.*X(L1,J)+X(L2,J))*DZ2
      YL = (3.*Y(KL,J)-4.*Y(L1,J)+Y(L2,J))*DZ2
      ZL = (3.*Z(KL,J)-4.*Z(L1,J)+Z(L2,J))*DZ2
      ENDIF
      XX(J,1) = (YK*ZL-ZK*YL)*RJ
      XX(J,2) = (ZK*XL-XK*ZL)*RJ
      XX(J,3) = (XK*YL-YK*XL)*RJ
      IF (J .EQ. LQQ0) THEN
      CALL RECV (OMEGA                            1)
C ----------------------------
      ENDIF
      XX(J,4) = -OMEGA*(Z(KL,J)*XX(J,2)-Y(KL,J)*XX(J,3))
   10 CONTINUE
      KQQ=J2
      IF (J .EQ. KQQ) THEN
      CALL SEND (J                                1)
      CALL SEND (RJ                               1)
      CALL SEND (XK                               1)
      CALL SEND (YK                               1)
      CALL SEND (ZK                               1)
      CALL SEND (K1                               1)
      CALL SEND (K2                               1)
      CALL SEND (XL                               1)
      CALL SEND (YL                               1)
      CALL SEND (ZL                               1)
      CALL SEND (L1                               1)
      CALL SEND (L2                               1)
```

```
      ENDIF
C -----------------------------
      CALL SEND(XX(KZZ,*)KZZ=LQQ0,MQQ0,1)
      END
```

```
FILE     4       {Processor 4}

      CALL RECV (J2                                    1)
      CALL RECV (J1                                    1)
      NQQ0=(J2-(J1)+1-0) DIV 4+1
      LQQ0=J1+0+3*NQQ0
      MQQ0=J2
      DO 10 J=LQQ0,MQQ0
      IF (J .EQ. LQQ0) THEN
      CALL RECV (KL                                    1)
C -----------------------------
      CALL RECV(Q(KL,*,KZZ)KZZ=LQQ0,MQQ0,1)
      ENDIF
      RJ = Q(KL,6,J)
      IF (J .EQ. LQQ0) THEN
      CALL RECV (KMAX                                  1)
      CALL RECV (K                                     1)
C -----------------------------
      ENDIF
      IF((K.NE.1).AND.(K.NE.KMAX)) THEN
      IF (J .EQ. LQQ0) THEN
      CALL RECV (DY2                                   1)
      CALL RECV (KR                                    1)
C -----------------------------
      CALL RECV (KP                                    1)
      CALL RECV(X(*,KZZ)KZZ=LQQ0,MQQ0,1)
      ENDIF
      XK = (X(KP,J)-X(KR,J))*DY2
      IF (J .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Y(*,KZZ)KZZ=LQQ0,MQQ0,1)
      ENDIF
      YK = (Y(KP,J)-Y(KR,J))*DY2
      IF (J .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Z(*,KZZ)KZZ=LQQ0,MQQ0,1)
      ENDIF
      ZK = (Z(KP,J)-Z(KR,J))*DY2
      ELSE IF(K.EQ.1) THEN
      K1 = KL+1
      K2 = KL+2
      XK = (-3.*X(KL,J)+4.*X(K1,J)-X(K2,J))*DY2
      YK = (-3.*Y(KL,J)+4.*Y(K1,J)-Y(K2,J))*DY2
      ZK = (-3.*Z(KL,J)+4.*Z(K1,J)-Z(K2,J))*DY2
      ELSE
      K1 = KL-1
      K2 = KL-2
      XK = (3.*X(KL,J)-4.*X(K1,J)+X(K2,J))*DY2
      YK = (3.*Y(KL,J)-4.*Y(K1,J)+Y(K2,J))*DY2
      ZK = (3.*Z(KL,J)-4.*Z(K1,J)+Z(K2,J))*DY2
      ENDIF
      IF (J .EQ. LQQ0) THEN
      CALL RECV (LMAX                                  1)
```

```
      CALL RECV (L                                    1)
C -----------------------------
      ENDIF
      IF((L.NE.1).AND.(L.NE.LMAX))THEN
      IF (J .EQ. LQQ0) THEN
      CALL RECV (DZ2                                  1)
      CALL RECV (LR                                   1)
      CALL RECV (LP                                   1)
C -----------------------------
      ENDIF
      XL = (X(LP,J)-X(LR,J))*DZ2
      YL = (Y(LP,J)-Y(LR,J))*DZ2
      ZL = (Z(LP,J)-Z(LR,J))*DZ2
      ELSE IF(L.EQ.1) THEN
      IF (J .EQ. LQQ0) THEN
      CALL RECV (ND                                   1)
C -----------------------------
      ENDIF
      L1 = KL+ND
      L2 = KL+2*ND
      XL = (-3.*X(KL,J)+4.*X(L1,J)-X(L2,J))*DZ2
      YL = (-3.*Y(KL,J)+4.*Y(L1,J)-Y(L2,J))*DZ2
      ZL = (-3.*Z(KL,J)+4.*Z(L1,J)-Z(L2,J))*DZ2
      ELSE
      L1 = KL-ND
      L2 = KL-2*ND
      XL = (3.*X(KL,J)-4.*X(L1,J)+X(L2,J))*DZ2
      YL = (3.*Y(KL,J)-4.*Y(L1,J)+Y(L2,J))*DZ2
      ZL = (3.*Z(KL,J)-4.*Z(L1,J)+Z(L2,J))*DZ2
      ENDIF
      XX(J,1) = (YK*ZL-ZK*YL)*RJ
      XX(J,2) = (ZK*XL-XK*ZL)*RJ
      XX(J,3) = (XK*YL-YK*XL)*RJ
      IF (J .EQ. LQQ0) THEN
      CALL RECV (OMEGA                                1)
C -----------------------------
      ENDIF
      XX(J,4) = -OMEGA*(Z(KL,J)*XX(J,2)-Y(KL,J)*XX(J,3))
   10 CONTINUE
      KQQ=J2
      IF (J .EQ. KQQ) THEN
      CALL SEND (J                                    1)
      CALL SEND (RJ                                   1)
      CALL SEND (XK                                   1)
      CALL SEND (YK                                   1)
      CALL SEND (ZK                                   1)
      CALL SEND (K1                                   1)
      CALL SEND (K2                                   1)
      CALL SEND (XL                                   1)
      CALL SEND (YL                                   1)
      CALL SEND (ZL                                   1)
      CALL SEND (L1                                   1)
      CALL SEND (L2                                   1)
```

```
      ENDIF
C -----------------------------
      CALL SEND(XX(KZZ,*)KZZ=LQQ0,MQQ0,1)
      END
```

APPENDIX C. SUBROUTINE 'YYM'

a). Sequential Codes

```
      DX2 = .5/DX1
      DZ2 = .5/DZ1
      JP = J+1
      JR = J-1
      DO 10 K = K1,K2
      KL = (L-1)*ND+K
      LP = KL+ND
      LR = KL-ND
      RJ = Q(KL,6,J)
      IF((J.NE.1).AND.(J.EQ.JMAX)) THEN
      XJ = (X(KL,JP)-X(KL,JR))*DX2
      YJ = (Y(KL,JP)-Y(KL,JR))*DX2
      ZJ = (Z(KL,JP)-Z(KL,JR))*DX2
      ELSE IF(J.EQ.1) THEN
      J1 = J+1
      J2 = J+2
      XJ = (-3.*X(KL,J)+4.*X(KL,J1)-X(KL,J2))*DX2
      YJ = (-3.*Y(KL,J)+4.*Y(KL,J1)-Y(KL,J2))*DX2
      ZJ = (-3.*Z(KL,J)+4.*Z(KL,J1)-Z(KL,J2))*DX2
      ELSE IF(J.EQ.JMAX) THEN
      J1 = J-1
      J2 = J-2
      XJ = (3.*X(KL,J)-4.*X(KL,J1)+X(KL,J2))*DX2
      YJ = (3.*Y(KL,J)-4.*Y(KL,J1)+Y(KL,J2))*DX2
      ZJ = (3.*Z(KL,J)-4.*Z(KL,J1)+Z(KL,J2))*DX2
      ENDIF
      IF((L.NE.1).AND.(L.EQ.LMAX))THEN
      XL = (X(LP,J)-X(LR,J))*DZ2
      YL = (Y(LP,J)-Y(LR,J))*DZ2
      ZL = (Z(LP,J)-Z(LR,J))*DZ2
      ELSE IF(L.EQ.1) THEN
      L1 = KL+ND
      L2 = KL+2*ND
      XL = (-3.*X(KL,J)+4.*X(L1,J)-X(L2,J))*DZ2
      YL = (-3.*Y(KL,J)+4.*Y(L1,J)-Y(L2,J))*DZ2
      ZL = (-3.*Z(KL,J)+4.*Z(L1,J)-Z(L2,J))*DZ2
      ELSE IF(L.EQ.LMAX) THEN
      L1 = KL-ND
      L2 = KL-2*ND
      XL = (3.*X(KL,J)-4.*X(L1,J)+X(L2,J))*DZ2
      YL = (3.*Y(KL,J)-4.*Y(L1,J)+Y(L2,J))*DZ2
      ZL = (3.*Z(KL,J)-4.*Z(L1,J)+Z(L2,J))*DZ2
      ENDIF
      YY(K,1) = (ZJ*YL-YJ*ZL)*RJ
      YY(K,2) = (XJ*ZL-XL*ZJ)*RJ
      YY(K,3) = (YJ*XL-XJ*YL)*RJ
      YY(K,4) = -OMEGA*(Z(KL,J)*YY(K,2)-Y(KL,J)*YY(K,3))
   10 CONTINUE
```

```
RETURN
END
```

APPENDIX C (continue)

b). Parallel Code

```
FILE     1       {processor 1}

      CALL SEND (K2                              2)
      CALL SEND (K1                              2)
      CALL SEND (K2                              3)
      CALL SEND (K1                              3)
      CALL SEND (K2                              4)
      CALL SEND (K1                              4)
      CALL SEND (L                               2)
      CALL SEND (L                               3)
      CALL SEND (L                               4)
      CALL SEND (J                               2)
      CALL SEND (J                               3)
      CALL SEND (J                               4)
      DX2 = .5/DX1
      CALL SEND (ND                              2)
      CALL SEND (ND                              3)
      CALL SEND (ND                              4)
      CALL SEND (JMAX                            2)
      CALL SEND (JMAX                            3)
      CALL SEND (JMAX                            4)
      CALL SEND (DX2                             2)
      CALL SEND (DX2                             3)
      CALL SEND (DX2                             4)
      CALL SEND (LMAX                            2)
      CALL SEND (LMAX                            3)
      CALL SEND (LMAX                            4)
      CALL SEND (OMEGA                           2)
      CALL SEND (OMEGA                           3)
      CALL SEND (OMEGA                           4)
      DZ2 = .5/DZ1
      CALL SEND (DZ2                             2)
      CALL SEND (DZ2                             3)
      CALL SEND (DZ2                             4)
      JP = J+1
      CALL SEND (JP                              2)
      CALL SEND (JP                              3)
      CALL SEND (JP                              4)
      JR = J-1
      CALL SEND (JR                              2)
      CALL SEND (JR                              3)
      CALL SEND (JR                              4)
      NQQ0=(K2-(K1)+1-0) DIV 4+1
      LQQ0=K1
      MQQ0=MIN(LQQ0+0+1*NQQ0-1,K2)
      LZZ=K1+0+NQQ0
      MZZ=MIN(LZZ+0+2*NQQ0-1,K2)
      CALL SEND(Q(*,*,J)KZZ=LZZ,MZZ,2)
      LZZ=K1+0+2*NQQ0
```

```
MZZ=MIN(LZZ+0+3*NQQ0-1,K2)
CALL SEND(Q(*,*,J)KZZ=LZZ,MZZ,3)
LZZ=K1+0+3*NQQ0
MZZ=MIN(LZZ+0+4*NQQ0-1,K2)
CALL SEND(Q(*,*,J)KZZ=LZZ,MZZ,4)
LZZ=K1+0+NQQ0
MZZ=MIN(LZZ+0+2*NQQ0-1,K2)
CALL SEND(X(*,*)KZZ=LZZ,MZZ,2)
LZZ=K1+0+2*NQQ0
MZZ=MIN(LZZ+0+3*NQQ0-1,K2)
CALL SEND(X(*,*)KZZ=LZZ,MZZ,3)
LZZ=K1+0+3*NQQ0
MZZ=MIN(LZZ+0+4*NQQ0-1,K2)
CALL SEND(X(*,*)KZZ=LZZ,MZZ,4)
LZZ=K1+0+NQQ0
MZZ=MIN(LZZ+0+2*NQQ0-1,K2)
CALL SEND(Y(*,*)KZZ=LZZ,MZZ,2)
LZZ=K1+0+2*NQQ0
MZZ=MIN(LZZ+0+3*NQQ0-1,K2)
CALL SEND(Y(*,*)KZZ=LZZ,MZZ,3)
LZZ=K1+0+3*NQQ0
MZZ=MIN(LZZ+0+4*NQQ0-1,K2)
CALL SEND(Y(*,*)KZZ=LZZ,MZZ,4)
LZZ=K1+0+NQQ0
MZZ=MIN(LZZ+0+2*NQQ0-1,K2)
CALL SEND(Z(*,*)KZZ=LZZ,MZZ,2)
LZZ=K1+0+2*NQQ0
MZZ=MIN(LZZ+0+3*NQQ0-1,K2)
CALL SEND(Z(*,*)KZZ=LZZ,MZZ,3)
LZZ=K1+0+3*NQQ0
MZZ=MIN(LZZ+0+4*NQQ0-1,K2)
CALL SEND(Z(*,*)KZZ=LZZ,MZZ,4)
DO 10 K=LQQ0,MQQ0
KL = (L-1)*ND+K
LP = KL+ND
LR = KL-ND
RJ = Q(KL,6,J)
IF((J.NE.1).AND.(J.EQ.JMAX)) THEN
XJ = (X(KL,JP)-X(KL,JR))*DX2
YJ = (Y(KL,JP)-Y(KL,JR))*DX2
ZJ = (Z(KL,JP)-Z(KL,JR))*DX2
ELSE IF(J.EQ.1) THEN
J1 = J+1
J2 = J+2
XJ = (-3.*X(KL,J)+4.*X(KL,J1)-X(KL,J2))*DX2
YJ = (-3.*Y(KL,J)+4.*Y(KL,J1)-Y(KL,J2))*DX2
ZJ = (-3.*Z(KL,J)+4.*Z(KL,J1)-Z(KL,J2))*DX2
ELSE IF(J.EQ.JMAX) THEN
J1 = J-1
J2 = J-2
XJ = (3.*X(KL,J)-4.*X(KL,J1)+X(KL,J2))*DX2
YJ = (3.*Y(KL,J)-4.*Y(KL,J1)+Y(KL,J2))*DX2
ZJ = (3.*Z(KL,J)-4.*Z(KL,J1)+Z(KL,J2))*DX2
```

```
      ENDIF
      IF((L.NE.1).AND.(L.EQ.LMAX)) THEN
      XL = (X(LP,J)-X(LR,J))*DZ2
      YL = (Y(LP,J)-Y(LR,J))*DZ2
      ZL = (Z(LP,J)-Z(LR,J))*DZ2
      ELSE IF(L.EQ.1) THEN
      L1 = KL+ND
      L2 = KL+2*ND
      XL = (-3.*X(KL,J)+4.*X(L1,J)-X(L2,J))*DZ2
      YL = (-3.*Y(KL,J)+4.*Y(L1,J)-Y(L2,J))*DZ2
      ZL = (-3.*Z(KL,J)+4.*Z(L1,J)-Z(L2,J))*DZ2
      ELSE IF(L.EQ.LMAX) THEN
      L1 = KL-ND
      L2 = KL-2*ND
      XL = (3.*X(KL,J)-4.*X(L1,J)+X(L2,J))*DZ2
      YL = (3.*Y(KL,J)-4.*Y(L1,J)+Y(L2,J))*DZ2
      ZL = (3.*Z(KL,J)-4.*Z(L1,J)+Z(L2,J))*DZ2
      ENDIF
      YY(K,1) = (ZJ*YL-YJ*ZL)*RJ
      YY(K,2) = (XJ*ZL-XL*ZJ)*RJ
      YY(K,3) = (YJ*XL-XJ*YL)*RJ
      YY(K,4) = -OMEGA*(Z(KL,J)*YY(K,2)-Y(KL,J)*YY(K,3))
   10 CONTINUE
C ----------------------------
      CALL RECV (K                                1)
      CALL RECV (KL                               1)
      CALL RECV (LP                               1)
      CALL RECV (LR                               1)
      CALL RECV (RJ                               1)
      CALL RECV (XJ                               1)
      CALL RECV (YJ                               1)
      CALL RECV (ZJ                               1)
      CALL RECV (J1                               1)
      CALL RECV (J2                               1)
      CALL RECV (XL                               1)
      CALL RECV (YL                               1)
      CALL RECV (ZL                               1)
      CALL RECV (L1                               1)
      CALL RECV (L2                               1)
      KQQ=K2
      LZZ=K1+0+NQQ0
      MZZ=MIN(LZZ+0+2*NQQ0-1,K2)
      CALL RECV (YY(KZZ,*)KZZ=LZZ,MZZ,2)
      LZZ=K1+0+2*NQQ0
      MZZ=MIN(LZZ+0+3*NQQ0-1,K2)
      CALL RECV (YY(KZZ,*)KZZ=LZZ,MZZ,3)
      LZZ=K1+0+3*NQQ0
      MZZ=MIN(LZZ+0+4*NQQ0-1,K2)
      CALL RECV (YY(KZZ,*)KZZ=LZZ,MZZ,4)
      END
```

9

9

```
FILE     2      {Processor 2}

      CALL RECV (K2                                  1)
      CALL RECV (K1                                  1)
      NQQ0=(K2-(K1)+1-0) DIV 4+1
      LQQ0=K1+0+NQQ0
      MQQ0=MIN(LQQ0+0+2*NQQ0-1,K2)
      DO 10 K=LQQ0,MQQ0
      IF (K .EQ. LQQ0) THEN
      CALL RECV (ND                                  1)
      CALL RECV (L                                   1)
C -----------------------------
      ENDIF
      KL = (L-1)*ND+K
      LP = KL+ND
      LR = KL-ND
      IF (K .EQ. LQQ0) THEN
      CALL RECV (J                                   1)
C -----------------------------
      CALL RECV(Q(*,*,J)KZZ=LQQ0,MQQ0,1)
      ENDIF
      RJ = Q(KL,6,J)
      IF (K .EQ. LQQ0) THEN
      CALL RECV (JMAX                                1)
C -----------------------------
      ENDIF
      IF((J.NE.1).AND.(J.EQ.JMAX)) THEN
      IF (K .EQ. LQQ0) THEN
      CALL RECV (DX2                                 1)
      CALL RECV (JR                                  1)
C -----------------------------
      CALL RECV (JP                                  1)
      CALL RECV(X(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      XJ = (X(KL,JP)-X(KL,JR))*DX2
      IF (K .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Y(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      YJ = (Y(KL,JP)-Y(KL,JR))*DX2
      IF (K .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Z(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      ZJ = (Z(KL,JP)-Z(KL,JR))*DX2
      ELSE IF(J.EQ.1) THEN
      J1 = J+1
      J2 = J+2
      XJ = (-3.*X(KL,J)+4.*X(KL,J1)-X(KL,J2))*DX2
      YJ = (-3.*Y(KL,J)+4.*Y(KL,J1)-Y(KL,J2))*DX2
      ZJ = (-3.*Z(KL,J)+4.*Z(KL,J1)-Z(KL,J2))*DX2
      ELSE IF(J.EQ.JMAX) THEN
      J1 = J-1
```

```
      J2 = J-2
      XJ = (3.*X(KL,J)-4.*X(KL,J1)+X(KL,J2))*DX2
      YJ = (3.*Y(KL,J)-4.*Y(KL,J1)+Y(KL,J2))*DX2
      ZJ = (3.*Z(KL,J)-4.*Z(KL,J1)+Z(KL,J2))*DX2
      ENDIF
      IF (K .EQ. LQQ0) THEN
      CALL RECV (LMAX                                1)
C -----------------------------
      ENDIF
      IF((L.NE.1).AND.(L.EQ.LMAX)) THEN
      IF (K .EQ. LQQ0) THEN
      CALL RECV (DZ2                                 1)
C -----------------------------
      ENDIF
      XL = (X(LP,J)-X(LR,J))*DZ2
      YL = (Y(LP,J)-Y(LR,J))*DZ2
      ZL = (Z(LP,J)-Z(LR,J))*DZ2
      ELSE IF(L.EQ.1) THEN
      L1 = KL+ND
      L2 = KL+2*ND
      XL = (-3.*X(KL,J)+4.*X(L1,J)-X(L2,J))*DZ2
      YL = (-3.*Y(KL,J)+4.*Y(L1,J)-Y(L2,J))*DZ2
      ZL = (-3.*Z(KL,J)+4.*Z(L1,J)-Z(L2,J))*DZ2
      ELSE IF(L.EQ.LMAX) THEN
      L1 = KL-ND
      L2 = KL-2*ND
      XL = (3.*X(KL,J)-4.*X(L1,J)+X(L2,J))*DZ2
      YL = (3.*Y(KL,J)-4.*Y(L1,J)+Y(L2,J))*DZ2
      ZL = (3.*Z(KL,J)-4.*Z(L1,J)+Z(L2,J))*DZ2
      ENDIF
      YY(K,1) = (ZJ*YL-YJ*ZL)*RJ
      YY(K,2) = (XJ*ZL-XL*ZJ)*RJ
      YY(K,3) = (YJ*XL-XJ*YL)*RJ
      IF (K .EQ. LQQ0) THEN
      CALL RECV (OMEGA                               1)
C -----------------------------
      ENDIF
      YY(K,4) = -OMEGA*(Z(KL,J)*YY(K,2)-Y(KL,J)*YY(K,3))
   10 CONTINUE
      KQQ=K2
      IF (K .EQ. KQQ) THEN
      CALL SEND (K                                   1)
      CALL SEND (KL                                  1)
      CALL SEND (LP                                  1)
      CALL SEND (LR                                  1)
      CALL SEND (RJ                                  1)
      CALL SEND (XJ                                  1)
      CALL SEND (YJ                                  1)
      CALL SEND (ZJ                                  1)
      CALL SEND (J1                                  1)
      CALL SEND (J2                                  1)
      CALL SEND (XL                                  1)
      CALL SEND (YL                                  1)
```

```
      CALL SEND (ZL                                    1)
      CALL SEND (L1                                    1)
      CALL SEND (L2                                    1)
      ENDIF
C -----------------------------
      CALL SEND(YY(KZZ,*)KZZ=LQQ0,MQQ0,1)
      END
```

```
FILE     3      {Processor 3}

      CALL RECV (K2                                  1)
      CALL RECV (K1                                  1)
      NQQ0=(K2-(K1)+1-0) DIV 4+1
      LQQ0=K1+0+2*NQQ0
      MQQ0=MIN(LQQ0+0+3*NQQ0-1,K2)
      DO 10 K=LQQ0,MQQ0
      IF (K .EQ. LQQ0) THEN
      CALL RECV (ND                                  1)
      CALL RECV (L                                   1)
C -----------------------------
      ENDIF
      KL = (L-1)*ND+K
      LP = KL+ND
      LR = KL-ND
      IF (K .EQ. LQQ0) THEN
      CALL RECV (J                                   1)
C -----------------------------
      CALL RECV(Q(*,*,J)KZZ=LQQ0,MQQ0,1)
      ENDIF
      RJ = Q(KL,6,J)
      IF (K .EQ. LQQ0) THEN
      CALL RECV (JMAX                                1)
C -----------------------------
      ENDIF
      IF((J.NE.1).AND.(J.EQ.JMAX)) THEN
      IF (K .EQ. LQQ0) THEN
      CALL RECV (DX2                                 1)
      CALL RECV (JR                                  1)
C -----------------------------
      CALL RECV (JP                                  1)
      CALL RECV(X(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      XJ = (X(KL,JP)-X(KL,JR))*DX2
      IF (K .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Y(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      YJ = (Y(KL,JP)-Y(KL,JR))*DX2
      IF (K .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Z(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      ZJ = (Z(KL,JP)-Z(KL,JR))*DX2
      ELSE IF(J.EQ.1) THEN
      J1 = J+1
      J2 = J+2
      XJ = (-3.*X(KL,J)+4.*X(KL,J1)-X(KL,J2))*DX2
      YJ = (-3.*Y(KL,J)+4.*Y(KL,J1)-Y(KL,J2))*DX2
      ZJ = (-3.*Z(KL,J)+4.*Z(KL,J1)-Z(KL,J2))*DX2
      ELSE IF(J.EQ.JMAX) THEN
      J1 = J-1
```

```
      J2 = J-2
      XJ = (3.*X(KL,J)-4.*X(KL,J1)+X(KL,J2))*DX2
      YJ = (3.*Y(KL,J)-4.*Y(KL,J1)+Y(KL,J2))*DX2
      ZJ = (3.*Z(KL,J)-4.*Z(KL,J1)+Z(KL,J2))*DX2
      ENDIF
      IF (K .EQ. LQQ0) THEN                                    1)
      CALL RECV (LMAX
-----------------------------
      ENDIF
      IF((L.NE.1).AND.(L.EQ.LMAX)) THEN
      IF (K .EQ. LQQ0) THEN                                    1)
      CALL RECV (DZ2
-----------------------------
      ENDIF
      XL = (X(LP,J)-X(LR,J))*DZ2
      YL = (Y(LP,J)-Y(LR,J))*DZ2
      ZL = (Z(LP,J)-Z(LR,J))*DZ2
      ELSE IF(L.EQ.1) THEN
      L1 = KL+ND
      L2 = KL+2*ND
      XL = (-3.*X(KL,J)+4.*X(L1,J)-X(L2,J))*DZ2
      YL = (-3.*Y(KL,J)+4.*Y(L1,J)-Y(L2,J))*DZ2
      ZL = (-3.*Z(KL,J)+4.*Z(L1,J)-Z(L2,J))*DZ2
      ELSE IF(L.EQ.LMAX) THEN
      L1 = KL-ND
      L2 = KL-2*ND
      XL = (3.*X(KL,J)-4.*X(L1,J)+X(L2,J))*DZ2
      YL = (3.*Y(KL,J)-4.*Y(L1,J)+Y(L2,J))*DZ2
      ZL = (3.*Z(KL,J)-4.*Z(L1,J)+Z(L2,J))*DZ2
      ENDIF
      YY(K,1) = (ZJ*YL-YJ*ZL)*RJ
      YY(K,2) = (XJ*ZL-XL*ZJ)*RJ
      YY(K,3) = (YJ*XL-XJ*YL)*RJ
      IF (K .EQ. LQQ0) THEN                                    1)
      CALL RECV (OMEGA
-----------------------------
      ENDIF
      YY(K,4) = -OMEGA*(Z(KL,J)*YY(K,2)-Y(KL,J)*YY(K,3))
   10 CONTINUE
      KQQ=K2
      IF (K .EQ. KQQ) THEN                                     1)
      CALL SEND (K                                             1)
      CALL SEND (KL                                            1)
      CALL SEND (LP                                            1)
      CALL SEND (LR                                            1)
      CALL SEND (RJ                                            1)
      CALL SEND (XJ                                            1)
      CALL SEND (YJ                                            1)
      CALL SEND (ZJ                                            1)
      CALL SEND (J1                                            1)
      CALL SEND (J2                                            1)
      CALL SEND (XL
      CALL SEND (YL
```

```
      CALL SEND (ZL                                  1)
      CALL SEND (L1                                  1)
      CALL SEND (L2                                  1)
      ENDIF
C -----------------------------
      CALL SEND(YY(KZZ,*)KZZ=LQQ0,MQQ0,1)
      END
```

9

9

```
FILE     4       {Processor 4}

      CALL RECV (K2                                    1)
      CALL RECV (K1                                    1)
      NQQ0=(K2-(K1)+1-0) DIV 4+1
      LQQ0=K1+0+3*NQQ0
      MQQ0=K2
      DO 10 K=LQQ0,MQQ0
      IF (K .EQ. LQQ0) THEN
      CALL RECV (ND                                    1)
      CALL RECV (L                                     1)
C -----------------------------
      ENDIF
      KL = (L-1)*ND+K
      LP = KL+ND
      LR = KL-ND
      IF (K .EQ. LQQ0) THEN
      CALL RECV (J                                     1)
C -----------------------------
      CALL RECV(Q(*,*,J)KZZ=LQQ0,MQQ0,1)
      ENDIF
      RJ = Q(KL,6,J)
      IF (K .EQ. LQQ0) THEN
      CALL RECV (JMAX                                  1)
C -----------------------------
      ENDIF
      IF((J.NE.1).AND.(J.EQ.JMAX)) THEN
      IF (K .EQ. LQQ0) THEN
      CALL RECV (DX2                                   1)
      CALL RECV (JR                                    1)
C -----------------------------
      CALL RECV (JP                                    1)
      CALL RECV(X(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      XJ = (X(KL,JP)-X(KL,JR))*DX2
      IF (K .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Y(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      YJ = (Y(KL,JP)-Y(KL,JR))*DX2
      IF (K .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Z(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      ZJ = (Z(KL,JP)-Z(KL,JR))*DX2
      ELSE IF(J.EQ.1) THEN
      J1 = J+1
      J2 = J+2
      XJ = (-3.*X(KL,J)+4.*X(KL,J1)-X(KL,J2))*DX2
      YJ = (-3.*Y(KL,J)+4.*Y(KL,J1)-Y(KL,J2))*DX2
      ZJ = (-3.*Z(KL,J)+4.*Z(KL,J1)-Z(KL,J2))*DX2
      ELSE IF(J.EQ.JMAX) THEN
      J1 = J-1
```

```
      J2 = J-2
      XJ = (3.*X(KL,J)-4.*X(KL,J1)+X(KL,J2))*DX2
      YJ = (3.*Y(KL,J)-4.*Y(KL,J1)+Y(KL,J2))*DX2
      ZJ = (3.*Z(KL,J)-4.*Z(KL,J1)+Z(KL,J2))*DX2
      ENDIF
      IF (K .EQ. LQQ0) THEN
      CALL RECV (LMAX                              1)
C -----------------------------
      ENDIF
      IF((L.NE.1).AND.(L.EQ.LMAX)) THEN
      IF (K .EQ. LQQ0) THEN
      CALL RECV (DZ2                               1)
C -----------------------------
      ENDIF
      XL = (X(LP,J)-X(LR,J))*DZ2
      YL = (Y(LP,J)-Y(LR,J))*DZ2
      ZL = (Z(LP,J)-Z(LR,J))*DZ2
      ELSE IF(L.EQ.1) THEN
      L1 = KL+ND
      L2 = KL+2*ND
      XL = (-3.*X(KL,J)+4.*X(L1,J)-X(L2,J))*DZ2
      YL = (-3.*Y(KL,J)+4.*Y(L1,J)-Y(L2,J))*DZ2
      ZL = (-3.*Z(KL,J)+4.*Z(L1,J)-Z(L2,J))*DZ2
      ELSE IF(L.EQ.LMAX) THEN
      L1 = KL-ND
      L2 = KL-2*ND
      XL = (3.*X(KL,J)-4.*X(L1,J)+X(L2,J))*DZ2
      YL = (3.*Y(KL,J)-4.*Y(L1,J)+Y(L2,J))*DZ2
      ZL = (3.*Z(KL,J)-4.*Z(L1,J)+Z(L2,J))*DZ2
      ENDIF
      YY(K,1) = (ZJ*YL-YJ*ZL)*RJ
      YY(K,2) = (XJ*ZL-XL*ZJ)*RJ
      YY(K,3) = (YJ*XL-XJ*YL)*RJ
      IF (K .EQ. LQQ0) THEN
      CALL RECV (OMEGA                             1)
C -----------------------------
      ENDIF
      YY(K,4) = -OMEGA*(Z(KL,J)*YY(K,2)-Y(KL,J)*YY(K,3))
   10 CONTINUE
      KQQ=K2
      IF (K .EQ. KQQ) THEN
      CALL SEND (K                                 1)
      CALL SEND (KL                                1)
      CALL SEND (LP                                1)
      CALL SEND (LR                                1)
      CALL SEND (RJ                                1)
      CALL SEND (XJ                                1)
      CALL SEND (YJ                                1)
      CALL SEND (ZJ                                1)
      CALL SEND (J1                                1)
      CALL SEND (J2                                1)
      CALL SEND (XL                                1)
      CALL SEND (YL                                1)
```

```
      CALL SEND (ZL                                  1)
      CALL SEND (L1                                  1)
      CALL SEND (L2                                  1)
      ENDIF
C -----------------------------
      CALL SEND(YY(KZZ,*)KZZ=LQQ0,MQQ0,1)
      END
```

APPENDIX D. SUBROUTINE 'ZZM'

a). Sequential Code

```
K = M
DX2 = .5/DX1
DY2 = .5/DY1
JP = J+1
JR = J-1
DO 10 L = L1,L2
KL = (L-1)*ND+K
KP = KL+1
KR = KL-1
RJ = Q(KL,6,J)
IF((K.NE.1).AND.(K.NE.KMAX)) THEN
XK = (X(KP,J)-X(KR,J))*DY2
YK = (Y(KP,J)-Y(KR,J))*DY2
ZK = (Z(KP,J)-Z(KR,J))*DY2
ELSE IF(K.EQ.1)THEN
K1 = KL+1
K2 = KL+2
XK = (-3.*X(KL,J)+4.*X(K1,J)-X(K2,J))*DY2
YK = (-3.*Y(KL,J)+4.*Y(K1,J)-Y(K2,J))*DY2
ZK = (-3.*Z(KL,J)+4.*Z(K1,J)-Z(K2,J))*DY2
ELSE IF(K.EQ.KMAX) THEN
K1 = KL-1
K2 = KL-2
XK = (3.*X(KL,J)-4.*X(K1,J)+X(K2,J))*DY2
YK = (3.*Y(KL,J)-4.*Y(K1,J)+Y(K2,J))*DY2
ZK = (3.*Z(KL,J)-4.*Z(K1,J)+Z(K2,J))*DY2
ENDIF
IF((J.NE.1).AND.(J.NE.JMAX)) THEN
XJ = (X(KL,JP)-X(KL,JR))*DX2
YJ = (Y(KL,JP)-Y(KL,JR))*DX2
ZJ = (Z(KL,JP)-Z(KL,JR))*DX2
ELSE IF(J.EQ.1) THEN
J1 = J+1
J2 = J+2
XJ = (-3.*X(KL,J)+4.*X(KL,J1)-X(KL,J2))*DX2
YJ = (-3.*Y(KL,J)+4.*Y(KL,J1)-Y(KL,J2))*DX2
ZJ = (-3.*Z(KL,J)+4.*Z(KL,J1)-Z(KL,J2))*DX2
ELSE IF(J.EQ.JMAX) THEN
J1 = J-1
J2 = J-2
XJ = (3.*X(KL,J)-4.*X(KL,J1)+X(KL,J2))*DX2
YJ = (3.*Y(KL,J)-4.*Y(KL,J1)+Y(KL,J2))*DX2
ZJ = (3.*Z(KL,J)-4.*Z(KL,J1)+Z(KL,J2))*DX2
ENDIF
ZZ(L,1) = (YJ*ZK-ZJ*YK)*RJ
ZZ(L,2) = (XK*ZJ-XJ*ZK)*RJ
ZZ(L,3) = (XJ*YK-YJ*XK)*RJ
ZZ(L,4) = -OMEGA*(Z(KL,J)*ZZ(L,2)-Y(KL,J)*ZZ(L,3))
```

```
10 CONTINUE
   RETURN
   END
```

APPENDIX D. (continue)

b). Parallel Code

```
FILE     1       {Processor 1}

      CALL SEND (L2                              2)
      CALL SEND (L1                              2)
      CALL SEND (L2                              3)
      CALL SEND (L1                              3)
      CALL SEND (L2                              4)
      CALL SEND (L1                              4)
      CALL SEND (J                               2)
      CALL SEND (J                               3)
      CALL SEND (J                               4)
      K = M
      CALL SEND (K                               2)
      CALL SEND (K                               3)
      CALL SEND (K                               4)
      CALL SEND (ND                              2)
      CALL SEND (ND                              3)
      CALL SEND (ND                              4)
      CALL SEND (KMAX                            2)
      CALL SEND (KMAX                            3)
      CALL SEND (KMAX                            4)
      CALL SEND (JMAX                            2)
      CALL SEND (JMAX                            3)
      CALL SEND (JMAX                            4)
      CALL SEND (OMEGA                           2)
      CALL SEND (OMEGA                           3)
      CALL SEND (OMEGA                           4)
      DX2 = .5/DX1
      CALL SEND (DX2                             2)
      CALL SEND (DX2                             3)
      CALL SEND (DX2                             4)
      DY2 = .5/DY1
      CALL SEND (DY2                             2)
      CALL SEND (DY2                             3)
      CALL SEND (DY2                             4)
      JP = J+1
      CALL SEND (JP                              2)
      CALL SEND (JP                              3)
      CALL SEND (JP                              4)
      JR = J-1
      CALL SEND (JR                              2)
      CALL SEND (JR                              3)
      CALL SEND (JR                              4)
      NQQ0=(L2-(L1)+1-0) DIV 4+1
      LQQ0=L1
      MQQ0=MIN(LQQ0+0+1*NQQ0-1,L2)
      LZZ=L1+0+NQQ0
      MZZ=MIN(LZZ+0+2*NQQ0-1,L2)
```

```
CALL SEND(Q(*,*,J)KZZ=LZZ,MZZ,2)
LZZ=L1+0+2*NQQ0
MZZ=MIN(LZZ+0+3*NQQ0-1,L2)
CALL SEND(Q(*,*,J)KZZ=LZZ,MZZ,3)
LZZ=L1+0+3*NQQ0
MZZ=MIN(LZZ+0+4*NQQ0-1,L2)
CALL SEND(Q(*,*,J)KZZ=LZZ,MZZ,4)
LZZ=L1+0+NQQ0
MZZ=MIN(LZZ+0+2*NQQ0-1,L2)
CALL SEND(X(*,*)KZZ=LZZ,MZZ,2)
LZZ=L1+0+2*NQQ0
MZZ=MIN(LZZ+0+3*NQQ0-1,L2)
CALL SEND(X(*,*)KZZ=LZZ,MZZ,3)
LZZ=L1+0+3*NQQ0
MZZ=MIN(LZZ+0+4*NQQ0-1,L2)
CALL SEND(X(*,*)KZZ=LZZ,MZZ,4)
LZZ=L1+0+NQQ0
MZZ=MIN(LZZ+0+2*NQQ0-1,L2)
CALL SEND(Y(*,*)KZZ=LZZ,MZZ,2)
LZZ=L1+0+2*NQQ0
MZZ=MIN(LZZ+0+3*NQQ0-1,L2)
CALL SEND(Y(*,*)KZZ=LZZ,MZZ,3)
LZZ=L1+0+3*NQQ0
MZZ=MIN(LZZ+0+4*NQQ0-1,L2)
CALL SEND(Y(*,*)KZZ=LZZ,MZZ,4)
LZZ=L1+0+NQQ0
MZZ=MIN(LZZ+0+2*NQQ0-1,L2)
CALL SEND(Z(*,*)KZZ=LZZ,MZZ,2)
LZZ=L1+0+2*NQQ0
MZZ=MIN(LZZ+0+3*NQQ0-1,L2)
CALL SEND(Z(*,*)KZZ=LZZ,MZZ,3)
LZZ=L1+0+3*NQQ0
MZZ=MIN(LZZ+0+4*NQQ0-1,L2)
CALL SEND(Z(*,*)KZZ=LZZ,MZZ,4)
DO 10 L=LQQ0,MQQ0
KL = (L-1)*ND+K
KP = KL+1
KR = KL-1
RJ = Q(KL,6,J)
IF((K.NE.1).AND.(K.NE.KMAX)) THEN
XK = (X(KP,J)-X(KR,J))*DY2
YK = (Y(KP,J)-Y(KR,J))*DY2
ZK = (Z(KP,J)-Z(KR,J))*DY2
ELSE IF(K.EQ.1)THEN
K1 = KL+1
K2 = KL+2
XK = (-3.*X(KL,J)+4.*X(K1,J)-X(K2,J))*DY2
YK = (-3.*Y(KL,J)+4.*Y(K1,J)-Y(K2,J))*DY2
ZK = (-3.*Z(KL,J)+4.*Z(K1,J)-Z(K2,J))*DY2
ELSE IF(K.EQ.KMAX) THEN
K1 = KL-1
K2 = KL-2
XK = (3.*X(KL,J)-4.*X(K1,J)+X(K2,J))*DY2
```

```
      YK = (3.*Y(KL,J)-4.*Y(K1,J)+Y(K2,J))*DY2
      ZK = (3.*Z(KL,J)-4.*Z(K1,J)+Z(K2,J))*DY2
      ENDIF
      IF((J.NE.1).AND.(J.NE.JMAX)) THEN
      XJ = (X(KL,JP)-X(KL,JR))*DX2
      YJ = (Y(KL,JP)-Y(KL,JR))*DX2
      ZJ = (Z(KL,JP)-Z(KL,JR))*DX2
      ELSE IF(J.EQ.1) THEN
      J1 = J+1
      J2 = J+2
      XJ = (-3.*X(KL,J)+4.*X(KL,J1)-X(KL,J2))*DX2
      YJ = (-3.*Y(KL,J)+4.*Y(KL,J1)-Y(KL,J2))*DX2
      ZJ = (-3.*Z(KL,J)+4.*Z(KL,J1)-Z(KL,J2))*DX2
      ELSE IF(J.EQ.JMAX) THEN
      J1 = J-1
      J2 = J-2
      XJ = (3.*X(KL,J)-4.*X(KL,J1)+X(KL,J2))*DX2
      YJ = (3.*Y(KL,J)-4.*Y(KL,J1)+Y(KL,J2))*DX2
      ZJ = (3.*Z(KL,J)-4.*Z(KL,J1)+Z(KL,J2))*DX2
      ENDIF
      ZZ(L,1) = (YJ*ZK-ZJ*YK)*RJ
      ZZ(L,2) = (XK*ZJ-XJ*ZK)*RJ
      ZZ(L,3) = (XJ*YK-YJ*XK)*RJ
      ZZ(L,4) = -OMEGA*(Z(KL,J)*ZZ(L,2)-Y(KL,J)*ZZ(L,3))
   10 CONTINUE
C ----------------------------
      CALL RECV (L                                1)
      CALL RECV (KL                               1)
      CALL RECV (KP                               1)
      CALL RECV (KR                               1)
      CALL RECV (RJ                               1)
      CALL RECV (XK                               1)
      CALL RECV (YK                               1)
      CALL RECV (ZK                               1)
      CALL RECV (K1                               1)
      CALL RECV (K2                               1)
      CALL RECV (XJ                               1)
      CALL RECV (YJ                               1)
      CALL RECV (ZJ                               1)
      CALL RECV (J1                               1)
      CALL RECV (J2                               1)
      KQQ=L2
      LZZ=L1+0+NQQ0
      MZZ=MIN(LZZ+0+2*NQQ0-1,L2)
      CALL RECV (ZZ(KZZ,*)KZZ=LZZ,MZZ,2)
      LZZ=L1+0+2*NQQ0
      MZZ=MIN(LZZ+0+3*NQQ0-1,L2)
      CALL RECV (ZZ(KZZ,*)KZZ=LZZ,MZZ,3)
      LZZ=L1+0+3*NQQ0
      MZZ=MIN(LZZ+0+4*NQQ0-1,L2)
      CALL RECV (ZZ(KZZ,*)KZZ=LZZ,MZZ,4)
      END
```

9

9

```
FILE     2      {Processor 2}

      CALL RECV (L2                                  1)
      CALL RECV (L1                                  1)
      NQQ0=(L2-(L1)+1-0) DIV 4+1
      LQQ0=L1+0+NQQ0
      MQQ0=MIN(LQQ0+0+2*NQQ0-1,L2)
      DO 10 L=LQQ0,MQQ0
      IF (L .EQ. LQQ0) THEN
      CALL RECV (K                                   1)
      CALL RECV (ND                                  1)
C -----------------------------
      ENDIF
      KL = (L-1)*ND+K
      KP = KL+1
      KR = KL-1
      IF (L .EQ. LQQ0) THEN
      CALL RECV (J                                   1)
C -----------------------------
      CALL RECV(Q(*,*,J)KZZ=LQQ0,MQQ0,1)
      ENDIF
      RJ = Q(KL,6,J)
      IF (L .EQ. LQQ0) THEN
      CALL RECV (KMAX                                1)
C -----------------------------
      ENDIF
      IF((K.NE.1).AND.(K.NE.KMAX)) THEN
      IF (L .EQ. LQQ0) THEN
      CALL RECV (DY2                                 1)
C -----------------------------
      CALL RECV(X(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      XK = (X(KP,J)-X(KR,J))*DY2
      IF (L .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Y(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      YK = (Y(KP,J)-Y(KR,J))*DY2
      IF (L .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Z(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      ZK = (Z(KP,J)-Z(KR,J))*DY2
      ELSE IF(K.EQ.1)THEN
      K1 = KL+1
      K2 = KL+2
      XK = (-3.*X(KL,J)+4.*X(K1,J)-X(K2,J))*DY2
      YK = (-3.*Y(KL,J)+4.*Y(K1,J)-Y(K2,J))*DY2
      ZK = (-3.*Z(KL,J)+4.*Z(K1,J)-Z(K2,J))*DY2
      ELSE IF(K.EQ.KMAX) THEN
      K1 = KL-1
      K2 = KL-2
      XK = (3.*X(KL,J)-4.*X(K1,J)+X(K2,J))*DY2
```

```
      YK = (3.*Y(KL,J)-4.*Y(K1,J)+Y(K2,J))*DY2
      ZK = (3.*Z(KL,J)-4.*Z(K1,J)+Z(K2,J))*DY2
      ENDIF
      IF (L .EQ. LQQ0) THEN
      CALL RECV (JMAX                              1)
C -----------------------------
      ENDIF
      IF((J.NE.1).AND.(J.NE.JMAX)) THEN
      IF (L .EQ. LQQ0) THEN
      CALL RECV (DX2                               1)
      CALL RECV (JR                                1)
      CALL RECV (JP                                1)
C -----------------------------
      ENDIF
      XJ = (X(KL,JP)-X(KL,JR))*DX2
      YJ = (Y(KL,JP)-Y(KL,JR))*DX2
      ZJ = (Z(KL,JP)-Z(KL,JR))*DX2
      ELSE IF(J.EQ.1) THEN
      J1 = J+1
      J2 = J+2
      XJ = (-3.*X(KL,J)+4.*X(KL,J1)-X(KL,J2))*DX2
      YJ = (-3.*Y(KL,J)+4.*Y(KL,J1)-Y(KL,J2))*DX2
      ZJ = (-3.*Z(KL,J)+4.*Z(KL,J1)-Z(KL,J2))*DX2
      ELSE IF(J.EQ.JMAX) THEN
      J1 = J-1
      J2 = J-2
      XJ = (3.*X(KL,J)-4.*X(KL,J1)+X(KL,J2))*DX2
      YJ = (3.*Y(KL,J)-4.*Y(KL,J1)+Y(KL,J2))*DX2
      ZJ = (3.*Z(KL,J)-4.*Z(KL,J1)+Z(KL,J2))*DX2
      ENDIF
      ZZ(L,1) = (YJ*ZK-ZJ*YK)*RJ
      ZZ(L,2) = (XK*ZJ-XJ*ZK)*RJ
      ZZ(L,3) = (XJ*YK-YJ*XK)*RJ
      IF (L .EQ. LQQ0) THEN
      CALL RECV (OMEGA                             1)
C -----------------------------
      ENDIF
      ZZ(L,4) = -OMEGA*(Z(KL,J)*ZZ(L,2)-Y(KL,J)*ZZ(L,3))
   10 CONTINUE
      KQQ=L2
      IF (L .EQ. KQQ) THEN
      CALL SEND (L                                 1)
      CALL SEND (KL                                1)
      CALL SEND (KP                                1)
      CALL SEND (KR                                1)
      CALL SEND (RJ                                1)
      CALL SEND (XK                                1)
      CALL SEND (YK                                1)
      CALL SEND (ZK                                1)
      CALL SEND (K1                                1)
      CALL SEND (K2                                1)
      CALL SEND (XJ                                1)
      CALL SEND (YJ                                1)
```

```
      CALL SEND (ZJ                                  1)
      CALL SEND (J1                                  1)
      CALL SEND (J2                                  1)
      ENDIF
C -----------------------------
      CALL SEND(ZZ(KZZ,*)KZZ=LQQ0,MQQ0,1)
      END
```

```
FILE     3      {Processor 3}

      CALL RECV (L2                                  1)
      CALL RECV (L1                                  1)
      NQQ0=(L2-(L1)+1-0) DIV 4+1
      LQQ0=L1+0+2*NQQ0
      MQQ0=MIN(LQQ0+0+3*NQQ0-1,L2)
      DO 10 L=LQQ0,MQQ0
      IF (L .EQ. LQQ0) THEN
      CALL RECV (K                                   1)
      CALL RECV (ND                                  1)
C -----------------------------
      ENDIF
      KL = (L-1)*ND+K
      KP = KL+1
      KR = KL-1
      IF (L .EQ. LQQ0) THEN
      CALL RECV (J                                   1)
C -----------------------------
      CALL RECV(Q(*,*,J)KZZ=LQQ0,MQQ0,1)
      ENDIF
      RJ = Q(KL,6,J)
      IF (L .EQ. LQQ0) THEN
      CALL RECV (KMAX                                1)
C -----------------------------
      ENDIF
      IF((K.NE.1).AND.(K.NE.KMAX)) THEN
      IF (L .EQ. LQQ0) THEN
      CALL RECV (DY2                                 1)
C -----------------------------
      CALL RECV(X(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      XK = (X(KP,J)-X(KR,J))*DY2
      IF (L .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Y(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      YK = (Y(KP,J)-Y(KR,J))*DY2
      IF (L .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Z(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      ZK = (Z(KP,J)-Z(KR,J))*DY2
      ELSE IF(K.EQ.1)THEN
      K1 = KL+1
      K2 = KL+2
      XK = (-3.*X(KL,J)+4.*X(K1,J)-X(K2,J))*DY2
      YK = (-3.*Y(KL,J)+4.*Y(K1,J)-Y(K2,J))*DY2
      ZK = (-3.*Z(KL,J)+4.*Z(K1,J)-Z(K2,J))*DY2
      ELSE IF(K.EQ.KMAX) THEN
      K1 = KL-1
      K2 = KL-2
      XK = (3.*X(KL,J)-4.*X(K1,J)+X(K2,J))*DY2
```

```
      YK = (3.*Y(KL,J)-4.*Y(K1,J)+Y(K2,J))*DY2
      ZK = (3.*Z(KL,J)-4.*Z(K1,J)+Z(K2,J))*DY2
      ENDIF
      IF (L .EQ. LQQ0) THEN
      CALL RECV (JMAX                              1)
C ----------------------------
      ENDIF
      IF((J.NE.1).AND.(J.NE.JMAX)) THEN
      IF (L .EQ. LQQ0) THEN
      CALL RECV (DX2                               1)
      CALL RECV (JR                                1)
      CALL RECV (JP                                1)
C ----------------------------
      ENDIF
      XJ = (X(KL,JP)-X(KL,JR))*DX2
      YJ = (Y(KL,JP)-Y(KL,JR))*DX2
      ZJ = (Z(KL,JP)-Z(KL,JR))*DX2
      ELSE IF(J.EQ.1) THEN
      J1 = J+1
      J2 = J+2
      XJ = (-3.*X(KL,J)+4.*X(KL,J1)-X(KL,J2))*DX2
      YJ = (-3.*Y(KL,J)+4.*Y(KL,J1)-Y(KL,J2))*DX2
      ZJ = (-3.*Z(KL,J)+4.*Z(KL,J1)-Z(KL,J2))*DX2
      ELSE IF(J.EQ.JMAX) THEN
      J1 = J-1
      J2 = J-2
      XJ = (3.*X(KL,J)-4.*X(KL,J1)+X(KL,J2))*DX2
      YJ = (3.*Y(KL,J)-4.*Y(KL,J1)+Y(KL,J2))*DX2
      ZJ = (3.*Z(KL,J)-4.*Z(KL,J1)+Z(KL,J2))*DX2
      ENDIF
      ZZ(L,1) = (YJ*ZK-ZJ*YK)*RJ
      ZZ(L,2) = (XK*ZJ-XJ*ZK)*RJ
      ZZ(L,3) = (XJ*YK-YJ*XK)*RJ
      IF (L .EQ. LQQ0) THEN
      CALL RECV (OMEGA                             1)
C ----------------------------
      ENDIF
      ZZ(L,4) = -OMEGA*(Z(KL,J)*ZZ(L,2)-Y(KL,J)*ZZ(L,3))
   10 CONTINUE
      KQQ=L2
      IF (L .EQ. KQQ) THEN
      CALL SEND (L                                 1)
      CALL SEND (KL                                1)
      CALL SEND (KP                                1)
      CALL SEND (KR                                1)
      CALL SEND (RJ                                1)
      CALL SEND (XK                                1)
      CALL SEND (YK                                1)
      CALL SEND (ZK                                1)
      CALL SEND (K1                                1)
      CALL SEND (K2                                1)
      CALL SEND (XJ                                1)
      CALL SEND (YJ                                1)
```

```
      CALL SEND (ZJ                                    1)
      CALL SEND (J1                                    1)
      CALL SEND (J2                                    1)
      ENDIF
C -----------------------------
      CALL SEND(ZZ(KZZ,*)KZZ=LQQ0,MQQ0,1)
      END
```

```
FILE     4      {Processor 4}

      CALL RECV (L2                                  1)
      CALL RECV (L1                                  1)
      NQQ0=(L2-(L1)+1-0) DIV 4+1
      LQQ0=L1+0+3*NQQ0
      MQQ0=L2
      DO 10 L=LQQ0,MQQ0
      IF (L .EQ. LQQ0) THEN
      CALL RECV (K                                   1)
      CALL RECV (ND                                  1)
C -----------------------------
      ENDIF
      KL = (L-1)*ND+K
      KP = KL+1
      KR = KL-1
      IF (L .EQ. LQQ0) THEN
      CALL RECV (J                                   1)
C -----------------------------
      CALL RECV(Q(*,*,J)KZZ=LQQ0,MQQ0,1)
      ENDIF
      RJ = Q(KL,6,J)
      IF (L .EQ. LQQ0) THEN
      CALL RECV (KMAX                                1)
C -----------------------------
      ENDIF
      IF((K.NE.1).AND.(K.NE.KMAX)) THEN
      IF (L .EQ. LQQ0) THEN
      CALL RECV (DY2                                 1)
C -----------------------------
      CALL RECV(X(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      XK = (X(KP,J)-X(KR,J))*DY2
      IF (L .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Y(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      YK = (Y(KP,J)-Y(KR,J))*DY2
      IF (L .EQ. LQQ0) THEN
C -----------------------------
      CALL RECV(Z(*,*)KZZ=LQQ0,MQQ0,1)
      ENDIF
      ZK = (Z(KP,J)-Z(KR,J))*DY2
      ELSE IF(K.EQ.1)THEN
      K1 = KL+1
      K2 = KL+2
      XK = (-3.*X(KL,J)+4.*X(K1,J)-X(K2,J))*DY2
      YK = (-3.*Y(KL,J)+4.*Y(K1,J)-Y(K2,J))*DY2
      ZK = (-3.*Z(KL,J)+4.*Z(K1,J)-Z(K2,J))*DY2
      ELSE IF(K.EQ.KMAX) THEN
      K1 = KL-1
      K2 = KL-2
      XK = (3.*X(KL,J)-4.*X(K1,J)+X(K2,J))*DY2
```

```
      YK = (3.*Y(KL,J)-4.*Y(K1,J)+Y(K2,J))*DY2
      ZK = (3.*Z(KL,J)-4.*Z(K1,J)+Z(K2,J))*DY2
      ENDIF
      IF (L .EQ. LQQ0) THEN
      CALL RECV (JMAX                              1)
C -----------------------------
      ENDIF
      IF((J.NE.1).AND.(J.NE.JMAX)) THEN
      IF (L .EQ. LQQ0) THEN
      CALL RECV (DX2                               1)
      CALL RECV (JR                                1)
      CALL RECV (JP                                1)
C -----------------------------
      ENDIF
      XJ = (X(KL,JP)-X(KL,JR))*DX2
      YJ = (Y(KL,JP)-Y(KL,JR))*DX2
      ZJ = (Z(KL,JP)-Z(KL,JR))*DX2
      ELSE IF(J.EQ.1) THEN
      J1 = J+1
      J2 = J+2
      XJ = (-3.*X(KL,J)+4.*X(KL,J1)-X(KL,J2))*DX2
      YJ = (-3.*Y(KL,J)+4.*Y(KL,J1)-Y(KL,J2))*DX2
      ZJ = (-3.*Z(KL,J)+4.*Z(KL,J1)-Z(KL,J2))*DX2
      ELSE IF(J.EQ.JMAX) THEN
      J1 = J-1
      J2 = J-2
      XJ = (3.*X(KL,J)-4.*X(KL,J1)+X(KL,J2))*DX2
      YJ = (3.*Y(KL,J)-4.*Y(KL,J1)+Y(KL,J2))*DX2
      ZJ = (3.*Z(KL,J)-4.*Z(KL,J1)+Z(KL,J2))*DX2
      ENDIF
      ZZ(L,1) = (YJ*ZK-ZJ*YK)*RJ
      ZZ(L,2) = (XK*ZJ-XJ*ZK)*RJ
      ZZ(L,3) = (XJ*YK-YJ*XK)*RJ
      IF (L .EQ. LQQ0) THEN
      CALL RECV (OMEGA                             1)
C -----------------------------
      ENDIF
      ZZ(L,4) = -OMEGA*(Z(KL,J)*ZZ(L,2)-Y(KL,J)*ZZ(L,3))
   10 CONTINUE
      KQQ=L2
      IF (L .EQ. KQQ) THEN
      CALL SEND (L                                 1)
      CALL SEND (KL                                1)
      CALL SEND (KP                                1)
      CALL SEND (KR                                1)
      CALL SEND (RJ                                1)
      CALL SEND (XK                                1)
      CALL SEND (YK                                1)
      CALL SEND (ZK                                1)
      CALL SEND (K1                                1)
      CALL SEND (K2                                1)
      CALL SEND (XJ                                1)
      CALL SEND (YJ                                1)
```

```
      CALL SEND (ZJ                                    1)
      CALL SEND (J1                                    1)
      CALL SEND (J2                                    1)
      ENDIF
C -----------------------------
      CALL SEND(ZZ(KZZ,*)KZZ=LQQ0,MQQ0,1)
      END
```

END

DATE

April 11, 1989