Object-Oriented Implementation of the NAS Parallel Benchmarks using Charm++*

Sanjeev Krishnan, Milind Bhandarkar and Laxmikant V. Kalé
Dept of Computer Science, University of Illinois, Urbana, IL 61801
E-mail: {sanjeev,milind,kale}@cs.uiuc.edu

1 Introduction

This report describes experiences with implementing the NAS Computational Fluid Dynamics benchmarks using a parallel object-oriented language, Charm++. Our main objective in implementing the NAS CFD kernel benchmarks was to develop a code that could be used to easily experiment with different domain decomposition strategies and dynamic load balancing. We also wished to leverage the object-orientation provided by the Charm++ parallel object-oriented language [7, 8], to develop reusable abstractions that would simplify the process of developing parallel applications.

We first describe the Charm++ parallel programming model and the parallel object array abstraction, then go into detail about each of the Scalar Pentadiagonal (SP) and Lower/Upper Triangular (LU) benchmarks, along with performance results. Finally we conclude with an evaluation of the methodology used.

2 The Charm++ parallel object-oriented programming model

This work is based on the parallel object-oriented language *Charm++*[7, 8], an extension of C++. Charm++ is an explicitly parallel language, whose parallel constructs are modeled after the Charm parallel programming system [4]. Its innovative features include *message driven execution* for latency tolerance and modularity, *dynamic creation and load balancing* of concurrent objects, *branched objects* which have a representative on every processor, and multiple *specific information sharing abstractions*. This section describes the essential features, syntax, and implementation of Charm++.

Charm++ was designed to address the issues of portability, need to deal with communication latencies, support for irregular and dynamic computation structures, and reuse of parallel software modules.

2.1 Message Driven Execution

Charm++ uses message driven execution to overcome the problem of communication latency. In message driven execution, computation is initiated in response to the availability of a message. In Charm++, messages are directed to a method inside an object. Messages received from the network are kept in a queue, from which the system scheduler picks a message, and invokes the specified method within the object at which the incoming message is directed.

*This research was supported by the NASA grant NAG 2-897.

Message-driven execution, combined with an asynchronous (non-blocking) model of communication, exhibits latency tolerance by overlapping computation and communication adaptively and automatically. Each processor typically has multiple objects waiting to be scheduled based on availability of messages directed at them. A remote operation (such as fetching remote data), is initiated by an object by sending a message asynchronously to an object on the remote processor and returning control to the runtime system. The runtime system schedules pending computations in any other objects on the processor. When the remote data finally arrives in the form of a message, the runtime system can schedule the requesting object again. Multiple remote operations could be initiated by a single object and could be processed in the order these operations finish. Thus message-driven execution has several advantages over the traditional "blocking-receive" based communication, offering better performance through adaptive scheduling of computations. Message-driven execution also helps to promote modularity and reuse in parallel programs without losing efficiency, by allowing the overlap of computations across modules.

2.2 Dynamic Object Creation: chares and messages

In order to support irregular computations in which the amount of work on a processor changes dynamically and unpredictably, Charm++ allows dynamic creation of parallel objects (chares), which can then be mapped to different processors to balance loads. A chare is identified by a *handle*, which is a global pointer.

Chares communicate using messages. Sending a message to an object corresponds to an asynchronous method invocation. Message definitions have the form :

```
message class MessageType {
    // List of data and function members as in C++
} ;
```

Chare definitions have the form

```
chare class ChareType {
    // Data and member functions as in C++.
    // One or more entry functions of the form :
entry:
    void FunctionName(MessageType *MsgPointer)
        { C++ code block }
} ;
```

The entry function definition specifies code that is executed atomically when a message is received and scheduled for processing. Only one message per chare is executed at a time. Thus a chare object defines a boundary between sequential and parallel execution : actions within a chare are sequential, while those across chares may happen in parallel. Entry functions are public object methods with message as a parameter and no return value. The handle of a chare is of type "**ChareType handle**", and is unique across all processors. While multiple inheritance, dynamic binding, and overloading are supported for sequential objects by C++, Charm++ extends these concepts for chares (concurrent objects), thus permitting inheritance hierarchies of chare classes.

Every Charm++ program must have a chare type named **main**, which must have the function **main**. There can be only one instance of the **main** chare type, which usually executes on processor 0. Execution of a Charm++ program begins with the system creating an instance of the **main** chare and invoking its **main** function. Typically, this function is used by the programmer to create chares and branched chares and initialize shared objects.

Chares are created using the operator **newchare**, similar to the **new** in C++ : `newchare ChareType(MsgPointer)`, where `ChareType` is the name of a chare class. This operator deposits the *seed* for a new chare in a pool of seeds and returns immediately. Later, the runtime system will actually create the chare on some processor, as determined by the dynamic load balancing strategy. When the chare is created, it is initialized by executing its constructor entry function with the message contained in `MsgPointer` as parameter. The user can also specify a processor number as an optional argument to create the chare on specific processor, thereby overriding the dynamic load balancing strategy. A chare can obtain its own handle once it has been created and pass it to other objects in messages.

Messages are allocated using the C++ **new** operator. Messages can be sent to chares using the notation `ChareHandle=>EF(MsgPointer)`[1] This sends the message pointed to by `MsgPointer` to the chare having handle `ChareHandle` at the entry function `EF`, which must be a valid entry function of that chare type.

2.3 Dynamic load balancing

Charm++ supports dynamic object creation with dynamic load balancing libraries which help to map newly created chares to processors so that the work is balanced. Since the patterns of object creation vary widely among applications, and the characteristics of the underlying parallel machine also vary, different dynamic load balancing strategies become suitable in different circumstances. Charm++ provides many generic libraries for dynamic load balancing, which can be selected by the user at link time, depending on the requirements of the application. These libraries are implemented as modules on top of the basic runtime system.

2.4 Branched chares

A branched chare is a group of chares with a single name. A branched chare has one representative (branch) chare on each processor, and has a single global handle. One can asynchronously invoke a method on (i.e. send a message to) a representative of a branched chare by specifying its handle as well as the processor. In addition, one can synchronously (i.e. just like a sequential function call) invoke a method within the *local* representative of a branched chare.

Branched chares can be used to implement distributed services such as distributed data structures, global operations and high level information sharing abstractions, thereby encapsulating concurrency. They can be used for static load-balancing in object-parallel computations (each representative performs the same computation on the data owned by it). They also provide a convenient mechanism for distributed data exchange between modules : the representatives of a branched chare in one module hand over the data to the representatives in the other module on their own processors, without the need for centralized transfer. Finally, branched chares can also be used to encapsulate processor-specific information. (Indeed branched chares are used to implement many dynamic load balancing strategies.)

It is important to underscore that branched chares are objects. In particular, there may be multiple instances of the same branched chare class. So, simple local function calls do not provide the same service as the invocation of a method in a local representative branch.

Branched chares are also created with the **newchare** operator : `newchare ChareType(MsgPointer)`. This causes the runtime system to create a branch on every processor and initialize it by executing

[1](Note the syntactic difference between *asynchronous* message sending and sequential method invocation as in C++.)

its constructor entry function. Branched chares are usually created in the **main** function of the **main** chare, in which case this operator returns the handle of the newly created branched chare.
`ChareHandle[LOCAL]->DataMember` and `ChareHandle[LOCAL]->FunctionMember()` are used to access public members of the local branch of a branched chare.
`ChareHandle[P]=>EF(MsgPointer)` sends a message to the function EF in the branch of a branched chare on the processor P.
`ChareHandle[ALL]=>EF(MsgPointer)` results in a message being broadcast to all branches of a branched chare (i.e. to all processors).

2.5 Specific information sharing abstractions

Charm++ provides specific abstract object types for sharing information. Each abstraction for information sharing may be thought of as a template of an abstract object, with methods whose code is to be provided by the user. These shared objects have a global handle (name), and can be accessed on all processors, but only through their specific methods. These abstractions may be implemented differently on different architectures by the Charm++ runtime system, for efficiency. Some of the abstractions provided by Charm++ are: read-only variables, distributed tables, accumulators, and monotonic variables. Additional abstractions may be added as libraries, as the need for them arises.

2.6 Other Charm++ features

Prioritized Execution : Charm++ provides many strategies that the user can select for managing queues of messages waiting to be processed. Some of them (FIFO, LIFO, etc) are based solely on the temporal order of arrival of messages. However, in many applications (such as algorithms with a critical path, search-based algorithms, and discrete event simulations), it is necessary to allow the application to influence the order of processing of messages by assigning message priorities. Charm++ supports integer priorities as well as bit-vector priorities (with lexicographical comparison of bit-vectors determining order of processing), which are especially useful for prioritizing combinatorial search algorithms.
Conditional Message Packing : Charm++ allows arbitrarily complex data structures in messages. On private memory systems, pointers are not valid across processors, hence it is necessary to copy (*pack*) the pointer-linked structure into a contiguous block of memory before sending the message. However, packing is wasteful if the message is sent to an object on the same processor, or on shared memory systems. To allow optimal performance in this context, for messages involving pointers, the user is required to specify the methods *pack* and *unpack* in the message class for packing and unpacking messages that are called *by the system* just before sending and after receiving a message, respectively. Thus only messages that are actually sent to other processors are packed.
Quiescence Detection : Since the Charm++ model provides independently executing parallel objects, there is no single global thread of control, hence detecting quiescence (termination) of a program is difficult. Charm++ provides a quiescence detection library for this purpose, which detects quiescence (when no object is executing any computation and all messages sent have been processed). The programmer may then choose to simply exit, or start the next phase of the parallel program.

2.7 Implementation

Charm++ has been implemented as a translator and a runtime system. The translator converts Charm++ constructs into C++ constructs and calls to the runtime system. The runtime system is

layered into a language independent portable layer *Converse*, on top of which is the *Chare Kernel* layer.

2.7.1 Converse : Portability and interoperability

The Converse layer provides a portable machine interface which supports the essential parallel operations on MIMD machines. These includes synchronous and asynchronous sends and receives, global operations such as broadcast, atomic terminal I/O, and other advanced features. Some important principles that guided the development of Converse include *need-based cost* (e.g. Charm++ should not need to pay the overhead of a tag-based receive mechanism provided by an underlying layer, since Charm++ uses message-driven execution; also, a system such as PVM should not have to pay the cost of prioritized scheduling that Charm++ needs), *efficiency* (the performance of programs developed on top of Converse should be comparable to native implementations) and *component based design* (the Converse layer is divided into components with well-defined interfaces and possibly multiple implementations which can be plugged in as required by higher layers).

Converse is designed to help modules from different parallel programming paradigms to interoperate in a single application. In addition to common components such as the portable machine-interface, it provides paradigm-specific components such as message managers and thread objects, that can be customized and used to implement individual language runtime layers. Converse supports both SPMD style programs (which have no concurrency within a processor and explicit, static flow of control) as well as message-driven objects and threads (which have concurrency within a processor and implicit, adaptive scheduling).

The Converse machine interface has been ported to most parallel machines. Languages implemented on the Converse framework include Charm, Charm++, PVM (messaging), threaded PVM, SM (a simple messaging layer), and DP (a data parallel language).

2.7.2 Chare Kernel

The Chare Kernel layer was developed originally to support Charm, but was modified to support C++ interfaces required for Charm++ too. It implements various functions such as system initialization, chare creation, message processing (to identify the target object and deliver the message to it), performance measurements, quiescence detection, etc.

One important function of the Chare Kernel is to map parallel class and function names into consistent integer ids which can be passed to other processors. This is required because function and method pointers may not be identical across processors, especially in a heterogenous execution environment. The Charm++ translator cannot assign unique ids to classes and methods at compile time, because Charm++ supports separate compilation, and the translator does not know about the existence of other modules. Also, while passing ids for methods across processors, this mapping must be implemented so as to support inheritance and dynamic binding : when a sender sends a message to a chare C at an entry function E defined in C's base class, C must call its own definition of E if it has been redefined, otherwise it must call its base class' definition of E.

To meet these requirements the Chare Kernel provides a function registration facility, which maintains the mapping from ids to pointers. The translator-generated code uses this registration facility during initialization at run-time to assign globally unique indices to chare and entry function names. This unique id can be passed in messages across modules. The translator also generates stub functions for every entry function in every chare class. When a message is received and scheduled for processing, the Chare Kernel uses this stub function to invoke the correct method in the correct chare object. For dynamic binding to work, the stub function invoked is the one corresponding

to the static type of the chare handle at the call site; the C++ virtual function mechanism then invokes the correct method depending on the actual type of the chare object.

The Chare Kernel uses a scheduler (defined as a component of Converse) which is essentially a "pick and process" loop. It picks up incoming messages from the Converse message buffer, enqueues them by priority according to a user-selected queueing strategy, and then picks the highest priority message from the queue for processing.

Finally, the Chare Kernel also manages chare handles (which are essentially global pointers), and does the mapping from local object pointers to chare handles and vice versa. Branched chare handles need to be managed slightly differently, since they have a single global handle for a group of chares : the Chare Kernel needs to ensure that a consistent handle is used on all processors.

3 Parallel Array Abstraction

Since the NAS parallel benchmarks involved computations on a three dimensional data space, the natural parallelization scheme was to divide these arrays in many smaller *cubes* and perform computations on these cubes in parallel while preserving data dependencies. In order to represent a multi-dimensional array in parallel, we developed a parallel object array abstraction for Charm++ [9]. This abstraction allows the programmer to create an array of parallel objects, map it to processors according to the parallel algorithm requirements, send messages to selected elements, perform global operations such as multicasts, and specify new mappings for dynamic decompositions.

A parallel array is a group of objects (the array elements) with a common global name (id), which are organized in a multidimensional, distributed array, with each array element identified by its coordinates. The mapping of array elements to processors is specified by a user-provided mapping function. A default mapping is also provided for cases when the mapping is not significant. The data space of the problem could be partitioned into contiguous blocks and could be assigned to parallel objects that are elements of the parallel array.

3.1 Parallel Array Definition

A parallel array is defined as a normal parallel object (*chare*) class in Charm++, except that it must inherit from the system-defined base class *array*. This base class provides the following data fields :

- `thishandle` : this gives the unique handle (global pointer) of the array element.
- `thisgroup` : this gives the global id by which the whole array is known.
- `thisi`, `thisj`, `thisk` : these give the coordinates of the array element[2].

Messages that are sent between array elements must inherit from the system-defined message class *arraymsg*. The following code gives an example of an array definition.

```
message class MessageType : public arraymsg {
        // list of data fields to be sent
} ;
```

[2]Currently, only 1, 2, or 3-dimensional arrays are supported, although this can be easily extended to higher dimensions. For brevity, all the examples in this section assume a 2-dimensional array.

```
chare class MyArray : public array {
        // list of private and public data and function members
    entry:
        // list of "entry functions" where messages are received
        MyArray(MessageType *m) ;  // constructor
        void EntryFunction(MessageType *m) ;
} ;
```

3.2 Parallel Array Creation

A parallel array is created using the operator *newgroup*, which has the following syntax :

```
MapFunctionType mymapfn ;
MessageType *msgptr ;
MyArray  group arrayid1 =  newgroup MyArray[XSize][Ysize](msgptr) ;
MyArray  group arrayid2 =  newgroup (mymapfn) MyArray[XSize][Ysize](msgptr) ;
```

The code above creates two-dimensional parallel arrays with sizes `XSize` and `YSize` in X and Y dimensions. The `newgroup` operator causes all the array element objects to be created (and their constructors invoked) on their respective processors. The parameter `msgptr` is sent to all processors as the parameter to the constructor for each array element. The first array above uses the default mapping function. The second array has a user-specified mapping function `mymapfn`, which takes the coordinates of an element as input and returns the processor where the element is located. `newgroup` is a non-blocking operator that immediately returns the id of the newly created array, which has the type `MyArray group`, and is analogous to a global pointer to an array. Because of its non-blocking nature, the elements of an array might not have been created when *newgroup* returns the array id. If necessary, the programmer may explicitly synchronize after initialization of all array elements on all processors by using a suitable reduction or synchronization operation. Currently, parallel arrays may be created only from processor 0.

3.3 Asynchronous messaging: remote method invocation

The parallel array library provides both point-to-point as well as multicast messaging. All messaging is asynchronous (no reply value is allowed), in keeping with the non-blocking communication paradigm of Charm++. If a reply is desired, the receiving object must send a reply message back to the sender object.

The syntax for point-to-point asynchronous messaging is :

```
arrayid[i][j]=>EntryFunction(msgptr) ;
```

where arrayid is the "global pointer" to the parallel array, i, j are the coordinates of the recipient array element, `EntryFunction` is the function to be invoked in the receiving object, and `msgptr` is the message to be sent across, which is passed as the sole parameter to the function.

The syntax for multicast asynchronous messaging is :

```
arrayid[i1..i2][j1..j2]=>EntryFunction(msgptr) ; // multicast to sub-array
arrayid[ALL][j]=>EntryFunction(msgptr) ;         // multicast to column
arrayid[i][ALL]=>EntryFunction(msgptr) ;         // multicast to row
arrayid[ALL][ALL]=>EntryFunction(msgptr) ;       // multicast to whole array
```

If an array element is known to be on the local processor, its data and function members may be accessed as in sequential C++ :

```
arrayid[i][j]->datamember
arrayid[i][j]->functionmember(...)
```

3.4 Remapping and migration

The parallel array library supports both synchronous remapping and asynchronous object migration. Synchronous remapping must be initiated from processor 0 as follows :

```
arrayid->remap((MapFunctionType)newmapfn, return_chare_handle,
                                &(ReturnChareType::ReturnFunction)) ;
```

`newmapfn` is the new mapping function. All array elements will be moved from their original locations to their new locations as specified by the new mapping function. After all elements have been installed on their new locations, a message is sent to the function `ReturnFunction` in the chare object specified by `return_chare_handle`. This provides a synchronization point after remapping. The user program must ensure that no messages are sent to any elements of the array being re-mapped.

Sometimes such synchronization is impossible or inefficient. Asynchronous remapping or "migration" is activated by each array element independently, by calling the function
`migrate((MapFunctionType)newmapfn)`
on the array element to be moved. The `newmapfn` parameter specifies the new mapping function, which tells the run-time library the destination processor for the array element. The call results in only the specified object being moved to its destination processor. The run-time library will correctly forward messages directed to the migrating array element to its new location.

The actual steps performed by the runtime system while migrating an object are :

1. Before migrating an object, the runtime library calls a user-provided *pack* function on the object, which copies the object's data area into a contiguous message buffer. The programmer must provide a pack function for every object type that needs migration. (In future, we plan to automatically generate such pack and unpack functions based on the interface specification for array elements.)

2. Send the message to the object's destination processor

3. Create the new object

4. Initialize the object's data area using the message buffer. This is done by another user-provided *unpack* function. (Note: the pack and unpack functions are virtual functions defined in the base class `array`).

5. Forward messages directed to the object from the old processor to the new processor.

3.5 Implementation

The parallel array library is implemented on top of the Converse interoperable run-time framework [6]. The library can thus be used in conjunction with modules written in other programming systems such as PVM and MPI. Although the parallel array concepts we developed were implemented in the context of the Charm++ parallel object-oriented language, the essential features are language-independent. Currently we are in the process of modifying the Charm++ translator to translate the parallel array syntax into calls to C++ functions in the runtime library. The runtime library provides functions to create an array, send message to all elements of an array or to a subset of array, and various utility functions.

3.6 Typical Usage of Array Abstraction

In this section we describe how to use the Array abstraction in the current form (that is, without the translator modifications supporting the syntax described in earlier section.)

`CreateArray` function creates a parallel array of objects. The programmer needs to provide the `CreateArray` function with a mapping function. The mapping function takes group id and the coordinates as input parameters and returns a processor number on which the array element will be placed. An example of a simple mapping function is given below.

```
int Grid3D(int gid, int x, int y, int z)
{
    return (x%NX + (y%NY)*NX + (z%NZ)*NX*NY);
}
```

Where total number of processors is $NX \times NY \times NZ$.

An array is typically created in the main function of the main chare. One needs to allocate a message and fill in the appropriate fields in the message that will be sent to each element of the array for initialization. An example of array creation is given below:

```
msgptr = new CreateMessage;
msgptr->m = thishandle;
msgptr->dt = dt;
msgptr->omega = omega;
msgptr->itmax = itmax;

//cubearray = CreateArray(Chare(cube),EP(cube,cube),msgptr,
                          Grid3D,ncx,ncy,ncz);
cubearray = CreateArray(_CK_chare_cube,_CK_ep_cube_cube,msgptr,
                          Grid3D,ncx,ncy,ncz);
```

Here, a message of type `CreateMessage` is being sent to the newly created array of `cube` chares at the constructor entry function of each chare. The name mangling will be handled by the translator once it supports the array abstraction. However, currently the programmer needs to take care of it. (One can use macros `Chare` and `EP` to achieve this as shown in comments in the above example.) Chare names are translated to integers of the form `_CK_chare_`*charename* and the entry functions are translated as `_CK_ep_`*charename*`_`*entryname*. `ncx`, `ncy`, and `ncz` are the sizes of the array in X, Y and Z directions respectively.

The synchronization requirement after the creation of array demands that the newly created chares in the array perform a global reduction. This synchronization code needs to be provided by the programmer. Typically, this could be achieved by each chare sending a message to the main chare and awaiting a message from the main chare to trigger computation. The main chare keeps track of how many synchronization messages it receives and then sends a message to all the elements of the array to start computation. Messages could be sent to a particular element of an array using `SendArray` function or multicast to the entire array (or its subset) using `SendArrayRange` function.

4 The NAS Scalar Pentadiagonal (SP) benchmark

The NAS Scalar Pentadiagonal (SP) benchmark [1] is one of three simulated Computational Fluid Dynamics benchmarks in the NAS benchmark suite. It is intended to represent the principal computation and communication requirements of CFD applications in use today.

The SP benchmark involves the solution of multiple independent systems of scalar pentadiagonal equations which are not diagonally dominant. The computational space is a three-dimensional

structured mesh consisting of 64 x 64 x 64 grid points. The method used is an iterative Alternating Direction Implicit (ADI) method. In each iteration there are three "sweeps" successively along each of the three coordinate axes. Thus the method involves global spatial data dependences.

Our main objective in implementing the NAS SP benchmark was to develop a code that could be used to easily experiment with different domain decomposition strategies. We also wished to leverage the object-orientation provided by the Charm++ parallel object-oriented language [7, 8], to develop reusable abstractions that would simplify the process of developing parallel applications.

4.1 Parallelization schemes

The steps in the the numerical algorithm [3] which are significant for parallelization are :

- Computation of the RIIS vector of the partial differential equation. Each grid point in the cubical mesh needs values of the U matrix from two neighboring grid points on either side, in each of the three dimensions. This corresponds to six "parallel-shift" operations.

- Solution of a system of linear equations in the x-direction. Each grid point initially needs values from two succeeding grid points in the x-direction (corresponding to a shift operation in the negative-x direction). Then there is a sweep along the positive-x direction in which each grid point computes values that are needed by the next two points.

- Solution of a system of linear equations in the y-direction. This is similar to the previous step, except that communication is along the y-direction.

- Solution of a system of linear equations in the z-direction. This is similar to the previous step, except that communication is along the z-direction.

Parallelizing these steps requires decomposition of the three-dimensional computational array among processors. This decomposition must be done so as to balance computational load across processors as well as reduce inter-processor data communication.

Three of the most common methods used to parallelize ADI methods are [11] :

- Pipelined static block decomposition : each processor is statically allocated a contiguous three-dimensional block of grid points for the entire length of the computation. The block is made as close to cubical as possible to minimize the amount of communication (which is proportional to surface-area of the block). During the sweeps, each processor receives boundary data from the previous processor in the sweep direction, computes its data, and sends its boundary data on to the next processor. In order to reduce idle times while processors wait for data from previous processors, the computation is pipelined : each processor works on a slice of its grid points, sends the resulting boundary on to the next processor, and then goes on to the next slice. The disadvantage of this decomposition is that many processors idle at the beginning and end of the sweeps; moreover, there are many small messages sent between processors corresponding to the boundary data for each slice, which could cause significant overhead on machines with large message latencies.

- Transpose-based dynamic block decomposition : the three-dimensional mesh is divided into slabs oriented along the X direction first. After the X-direction sweep completes a transpose operation is done to orient the slabs along the Y-direction, in preparation for the Y-sweep. Finally, a third transpose operation is needed before the X-sweep of the next iteration. Thus there are a total of three transpose operations needed per iteration. The advantage of this

method is that computations within each sweep are completely local to a processor. However, the transpose operations between sweeps can result in significant overhead on bandwidth-limited machines.

- The multi-partition or Bruno-Capello decomposition [10, 2] : this is a static decomposition where the computational mesh is divided into cubes, and each cube is assigned to a processor such that all processors are active at all stages in each of the three sweeps. In other words, each coordinate plane in the computational space contains cubes on all processors. Thus processor loads are balanced during all stages of all sweeps, and also no transpose operations are needed. The minimum number of cubes needed for this decomposition is $P^{3/2}$ (where P is the number of processors), so that each processor has $\sqrt{P}$ cubes. The cube with coordinate (i, j, k) is allocated to processor $(i-k)\%s + s((j-k)\%s) + 1$, where $s = \sqrt[3]{P}$ and $1 <= i, j, k <= s$. The tradeoff in the multi-partition method is that computations within a sweep involve cross-processor messaging.

4.2 Implementing the SP benchmark using parallel arrays

We developed the following abstraction for the NAS SP benchmark code : The computational space is represented as a three-dimensional array of cubical sub-spaces. Each cube is represented by a parallel object in Charm++, which communicates with other cubes by sending and receiving messages. Thus the parallel program consists of a network of communicating objects.

The different decomposition/mapping strategies are expressed by simply specifying a different mapping function for the parallel array. E.g. the mapping function for the multipartition (Bruno-Capello) decomposition is :

```
int MultiPartitionMapFn(int arrayid, int i, int j, int k)
{       // return processor number owning object (i,j,k)
        return ( XArraySize*((i-k+XArraySize)%XArraySize) +
                   (j-k+YArraySize)%YArraySize ) ;
}
```

For the transpose method, all adjacent cubes along the direction of the sweep are mapped to the same processor. E.g. the mapping function for the sweep along the X-axis is :

```
int XSweepMapFn(int arrayid, int i, int j, int k)
{
        return ( ZArraySize * j + k ) ;
}
```

The transpose is effected by simply doing a remap operation on the parallel array between sweeps, with the mapping function corresponding to the orientation of the next sweep. Thus we have a very flexible, elegant code which allows us to concentrate on experiments with the application, instead of getting involved in the details of implementing the decomposition.

The asynchronous migration facility provided with parallel arrays allows us to further optimize the transpose method by overlapping communication and computation. Each cube object migrates itself as soon as it has completed its work along one sweep. Thus the communication overhead of transferring its data to another processor is overlapped with the computation performed by other cubes. This overlap gives significant performance advantages over the traditional loosely-synchronous (separate phases of computation and communication) implementations.

4.3 Performance results

Table 1 presents performance results for the different decompositions in the Charm++ implementation of the NAS SP benchmark. Sync-Transpose is the transpose-based dynamic block decomposition. Async-Transpose is the dynamic block decomposition with asynchronous migration of parallel objects for moving data between sweeps. Note that the only modification needed to change the decompositions for the different runs was to change the mapping function.

Processors	4	16	64	256
Sync-Transpose	-	8.08	3.01	1.98
Async-Transpose	-	7.81	2.54	1.40
Multipartition	24.63	7.60	1.98	1.00

Table 1: Time (in milliseconds) for different decompositions for the NAS SP benchmark (size A) on the Intel Paragon.

The results show that the multipartition (Bruno-Capello) decomposition is the best overall, with the Async-Transpose and Sync-Transpose decompositions being successively worse. The absolute performance of our program does not compare well with performance numbers quoted by vendors for the NAS benchmarks. This is mainly because our focus was on flexible parallelization issues, and not on tuning the algorithm or code for sequential or absolute performance.

5 The NAS Lower/Upper Triangular (LU) benchmark

The LU benchmark is one of the three CFD kernels in NAS benchmarks. It solves a regular-sparse, block (5 X 5) lower and upper triangular system. This represents the computations associated with the implicit operator of a newer class of implicit CFD algorithms, and has a lower degree of parallelism compared with other benchmarks in this suite. All the data-dependencies in this benchmark are local (nearest neighbor.) The system of linear equations obtained by replacing the spatial derivatives by second-order accurate, central finite difference operators is solved using the symmetric successive over-relaxation scheme. Each iteration of this algorithm consists of

- Computing the Right Hand Side explicitly.
- Forming and solving the regular, sparse, block lower triangular system.
- Forming and solving the regular, sparse, block upper triangular system.
- Updating the solution.

We used a spatial domain decomposition strategy to split the computational domain into a number of cubes. Different communication patterns develop as a result of data dependences in the above steps in each iteration. Computing RHS explicitly in the first step requires domain boundaries to be communicated between neighboring cubes (in both positive and negative X, Y and Z directions.) In the second step, forming the lower triangular system is a completely local operation and does not need any data from neighbors. However, solving the system requires wave-like communication pattern in the direction of diagonal of the computation domain. This is a

result of data-dependence where an element (i, j, k) needs elements $(i, j-1, k-1)$,$(i-1, j, k-1)$ and $(i-1, j-1, k)$. Computation in step 3 requires a wave-like communication similar to step 2 but in opposite direction. Thus, data-dependence mandates that there be three different communication patterns in each iteration. Also, the local data-dependence requires that these steps cannot be executed parallelly within the same iteration. This reduces the degree of parallelism in this benchmark significantly. Figure 1 shows that the available degree of parallelism in our algorithm in initial stages of the wave is very small. In the middle stages, where the wave reaches the principal diagonal of the 3-D data space, the degree of parallelism is high and then it reduces again in the later stages of the wave.

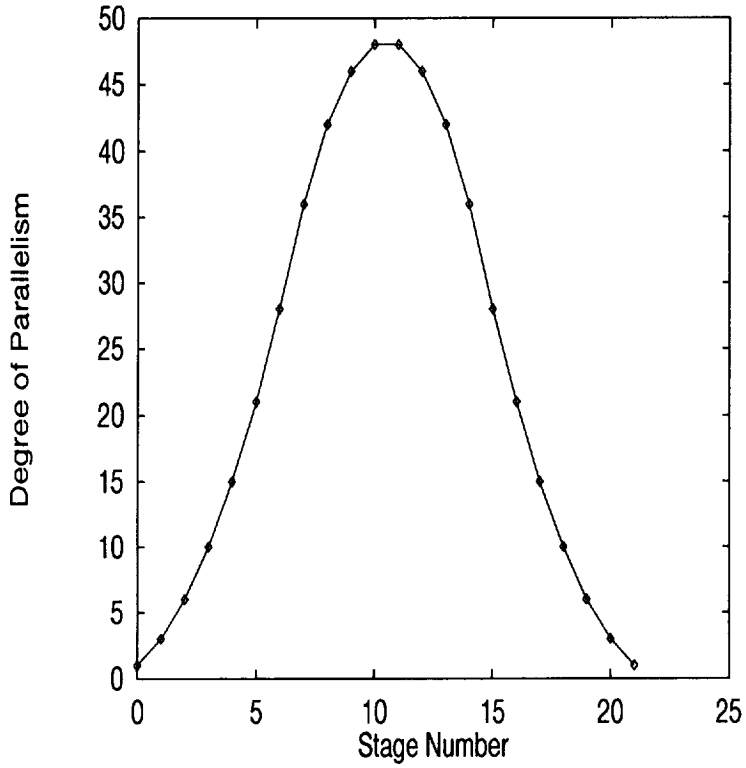

Figure 1: Degree of Parallelism in LU

In algorithms such as these, the only sources of enhancing performance are proper scheduling of work and overlapping communication and computation. One of the advantages of programming this application in a message-driven language such as Charm++ is that, the programmer only has to code the data-dependencies and leave the scheduling and overlapping communication and computation to the run-time system.

The implementation of this benchmark using the parallel array abstraction in Charm++ for domain decomposition is similar to the implementation of SP benchmark described in earlier section. This substantiates one of our main claims in this work that, using parallel object-oriented languages such as Charm++, one could develop reusable abstractions which could be used in development of several applications. However, the approach taken to develop the parallel code for LU differs from that of SP. In developing LU benchmark, we first converted the implementation of sequential LU algorithm from FORTRAN to C++, forming private methods for individual cubes in the process. Using the array abstraction to parallelize the C++ code was a very trivial task then. One of the

main advantage of this approach was that we were able to eliminate errors due to base-language variation early in the process using tools for sequential programs, which are more advanced than their counterparts in parallel programming.

We observed that the LU benchmark showed dependence to some extent on the actual placement of the cubes and the results of our experiments with different placement strategies reflect this observation. Our use of the parallel array abstraction in Charm++ allowed us to efficiently experiment with different placement strategies as well as different domain decomposition strategies.

One of the placement strategies we used is shown in figure 2. For simplicity, we have shown a wave-parallel placement pattern for a 2-dimensional 4×4 grid, placed using this strategy on 4 processors. The advantage of this placement strategy is that as the wave advances from one diagonal to the another, all the objects executing methods concurrently are placed on different processors. Therefore, we utilize all the processors optimally.

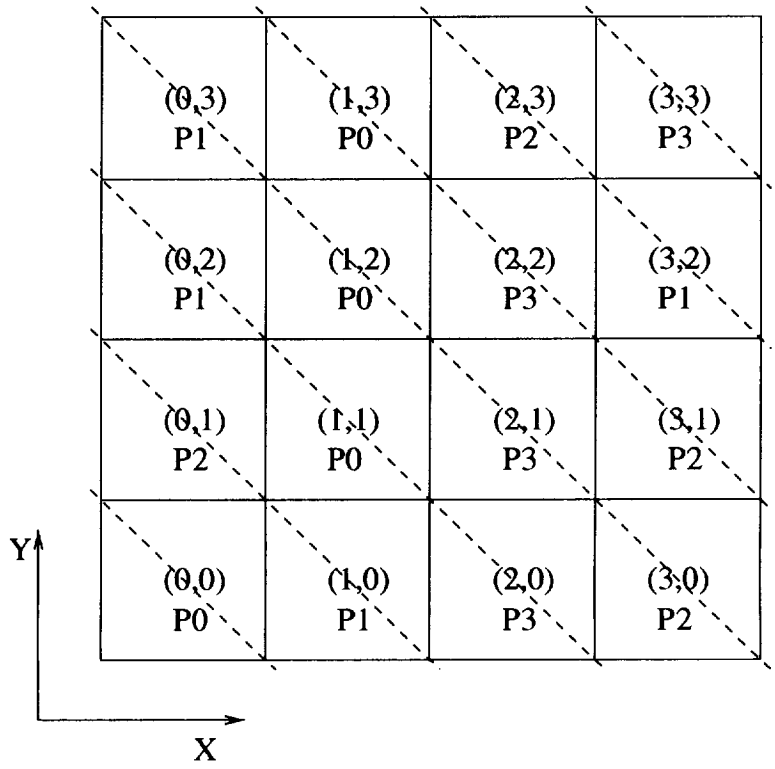

Figure 2: Wave-Parallel Placement Strategy

5.1 Performance Results

Our experiments were carried out on IBM-SP at Argonne National Laboratories. Charm++ is implemented on top of the native MPL communication library on IBM-SP systems. We conducted our experiments using 4 different decomposition strategies with 4, 8 and 16 processors and four different placement strategies (using different mapping functions during array creation.) The times are given for 25 iteration in seconds (The complete benchmark requires 250 iterations.) The different decompositions indicate the number of divisions of the computational domain in each direction. For example, a $8 \times 8 \times 1$ decomposition means the computational domain was split into 8 parts in

X and Y direction, but was left untouched in the Z direction, thus forming 64 *cubes*. Our results are comparable to numbers reported in [3] considering the experience of other users of IBM-SP at Argonne National Labs that it is slower than other installations of IBM-SP by almost factor of 2 for most programs. Also, we have not optimized the sequential part of computations that are coded in C++ rather than in FORTRAN which has a better set of optimization tools available for such scientific computations.

Processors	$4 \times 4 \times 1$	$4 \times 4 \times 4$	$8 \times 8 \times 1$	$8 \times 8 \times 8$
4	370.580	308.072	322.450	-
8	231.840	151.379	179.884	413.638
16	210.931	96.273	119.704	81.930

Table 2: Time (in seconds) for different decompositions for the NAS LU benchmark (size A) on the IBM SP using Wave-Parallel Mapping function.

Processors	$4 \times 4 \times 1$	$4 \times 4 \times 4$	$8 \times 8 \times 1$	$8 \times 8 \times 8$
4	453.809	298.159	300.545	-
8	430.003	289.487	220.567	343.630
16	423.968	289.297	219.503	320.542

Table 3: Time (in seconds) for different decompositions for the NAS LU benchmark (size A) on the IBM SP using 1D Grid Mapping function.

Processors	$4 \times 4 \times 1$	$4 \times 4 \times 4$	$8 \times 8 \times 1$	$8 \times 8 \times 8$
4	389.677	267.771	296.191	-
8	237.368	171.202	182.077	335.665
16	208.387	131.432	137.044	66.602

Table 4: Time (in seconds) for different decompositions for the NAS LU benchmark (size A) on the IBM SP using 2D Grid Mapping function.

5.2 Performance Analysis of LU

This section presents our work on the performance analysis of the LU code performed using a performance analysis tool for Charm and Charm++ programs, called Projections. Projections is available as a trace generation facility for Chare kernel, the run-time system of Charm and Charm++ languages; and as a performance visualization and analysis tool. It includes an expert system that works with the trace data generated by the program and analyzes for critical paths, phases and degree of parallelism within the code. For enabling projections trace generations, a

Processors	$4 \times 4 \times 1$	$4 \times 4 \times 4$	$8 \times 8 \times 1$	$8 \times 8 \times 8$
4	384.666	287.692	297.734	-
8	358.427	163.656	294.130	255.330
16	233.877	108.904	185.241	65.016

Table 5: Time (in seconds) for different decompositions for the NAS LU benchmark (size A) on the IBM SP using 3D-Grid Mapping function.

Charm++ program should be linked by specifying `-execmode projections` on the Charm linker command line. When this program is run, it produces trace files, one per processor. These files are then used as input for the X-windows based Projections visualizations tool. This section presents some of the analysis we did using Projections on the LU code.

One of the main reasons for performance degradation of parallel programs is the improper load-balancing. Proper load-balancing is characterized by equal amount of computation on all nodes. We checked the performance of LU for the amount of processing on each node. The processing time on each processor is shown in figure 3. It is in the range of 48 to 56 percent of the total time on each processor. This busy time is calculated based on the entire run of the program that includes the Charm initialization time during which most processors are idle. However, during the SSOR iterations, the busy time was 70 to 85 percent. Therefore, we concluded that LU is properly load balanced.

Though the total load across all nodes was determined to be similar, the regular structure of our application demanded that each type of computations should be distributed in equal amounts on all nodes for proper load balance. The main types of computations in LU are building the matrices (`setiv`), solving the lower and upper triangular systems (`blts` and `buts` respectively), and computing the RHS (`rhs`). We have made each of these computations into entry methods of the chare `cube`, therefore, determining the amount of each of these computations across all processors amounts to finding out how many times each of these entry methods were invoked on each chare. Figure 4 shows this in a graphical format. We can conclude from figure 4 that the individual computations were load balanced as well.

Another reason for the performance degradation especially in the light of our finding of low busy time per processor is the overhead imposed by the run-time system in the form of message sending, message processing and internal copying etc. However, the log files generated by Projections indicated that a total of 4424 messages were processed for each iteration of LU by each processor. From table 2, each iteration of LU takes 3.24 seconds. Thus the average grainsize of computation during the iteration is 732.368 μseconds. Another experiment was run on SP2, which calculated the overhead per message creation and processing. This involved running a simple pingpong program written in Charm++ that transferred messages back and forth between two processors. This experiment indicated that the average overhead per message processing was 126 μseconds. This amounts to 17 % overhead per message.

Next we viewed the aggregate work on all processors as a function of time. And noticed the distinct peaks for each of the iterations of LU. An iteration is characterized by forming RHS, solving the lower triangular system and solving the upper triangular system. We noticed that the amount of work done is at its peak in the middle of an iteration. This was expected since the degree of parallelism is at its highest in a wave-parallel distribution when the wave reaches the diagonal of the cube. Thus we figured out that this dependence is the main cause of low performance. We analyzed

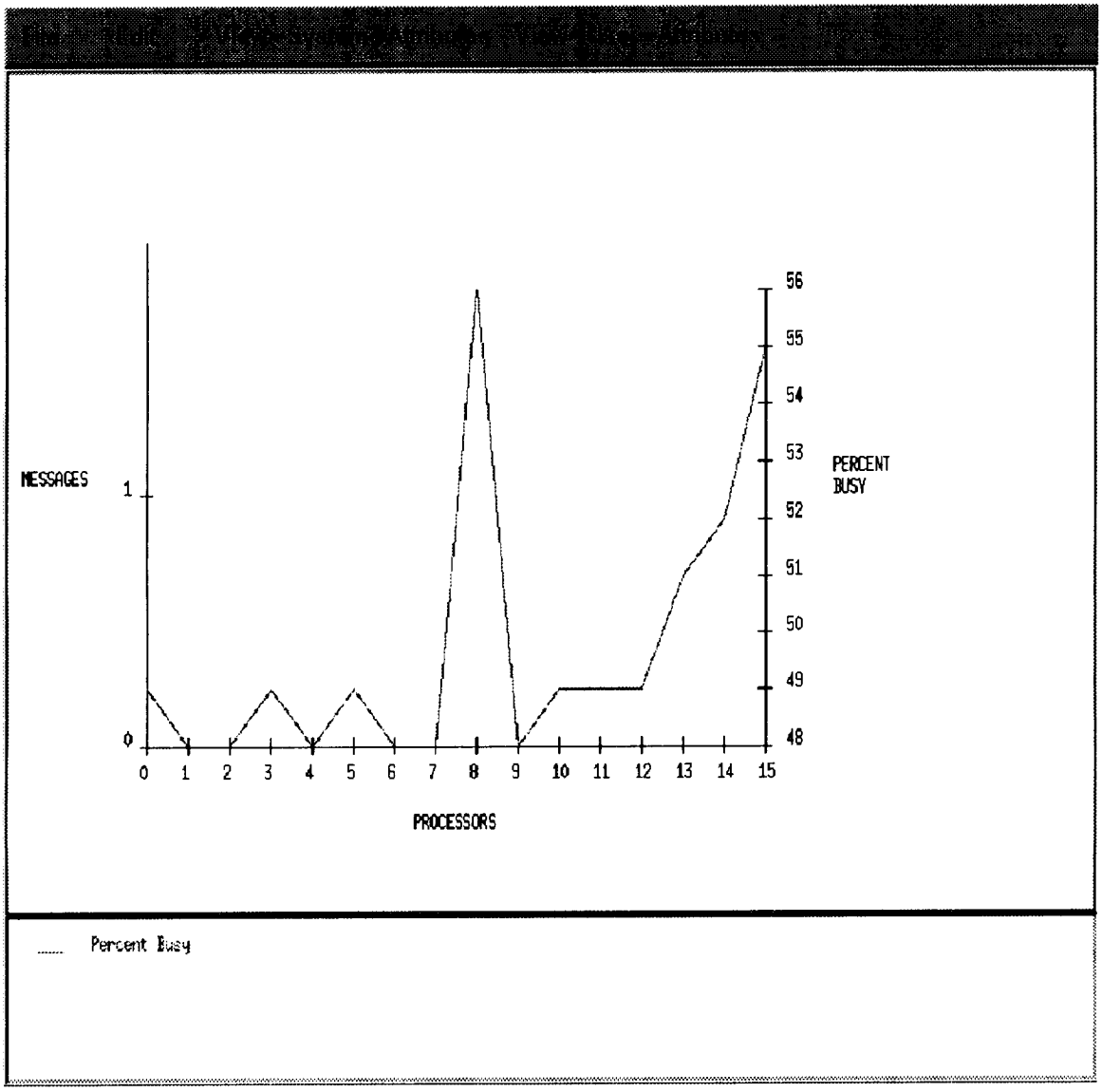

Figure 3: Busy Time Per Processor

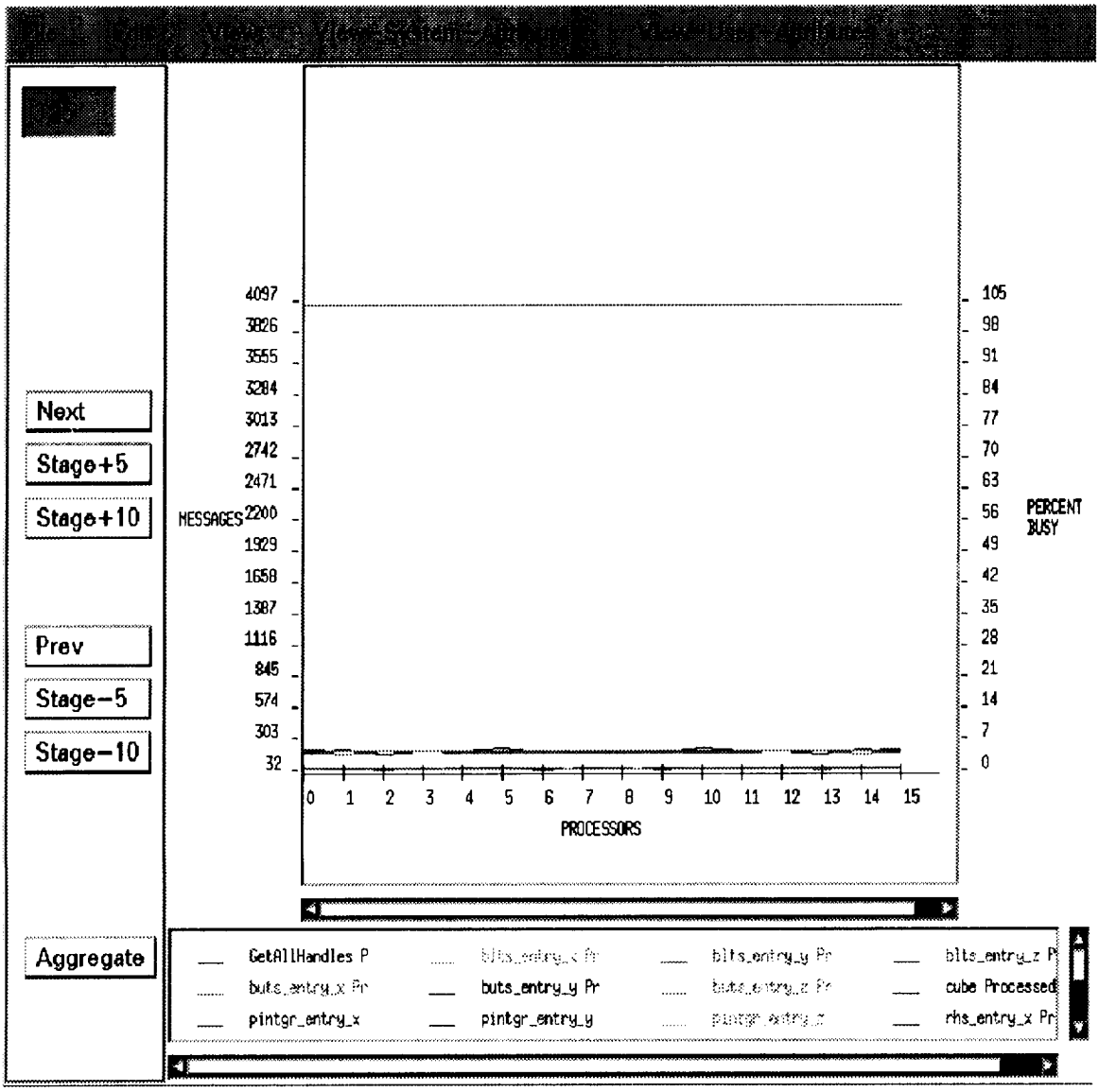

Figure 4: Individual Computations Per Processor

the trace data using the expert analysis tool of projections. Projections performed a critical path analysis and showed that almost all the computations for each cube are on critical path and that the average degree of parallelism is low. This degree of parallelism can be usually increased, as suggested by the expert system, by breaking the entry points into multiple entry points which could be executed in any order. However, this did not seem to be possible because of the dependences present in the problem. We have already split the dependences within each direction into different entry methods. However, the computations could be performed upon the arrival of these messages from all direction. Therefore, all of those entry methods will have to be on the critical path. Thus we concluded that the available degree of parallelism within the problem was very low and further optimizations were infeasible.

6 Conclusions

We have implemented the CFD kernels in the NAS Parallel Benchmarks using Charm++, a parallel object-oriented language. In order to simplify expression of multi-dimensional parallel arrays in Charm++, we implemented a parallel array abstractions using facilities provided by Charm++ and its runtime system, Converse. We have shown that the abstractions developed using Charm++ are indeed reusable by implementing both the Scalar Pentadiagonal (SP) and Lower-Upper Triangulation (LU) benchmarks without any modifications to the abstractions. The higher level array abstraction allowed us to experiment with many placement strategies and load-balancing without any significant programming overhead. Also, the benchmark code was developed such that the communication harness could be reused efficiently by plugging in different code for local computations. We have presented the performance results for both the codes using different placement and decomposition strategies. However, the main objective of this project was not to demonstrate the performance of the code but to demonstrate the ease of programming and experimentation using abstractions in parallel object-oriented languages such as Charm++. The later was demonstrated by the parallel array abstraction that made it possible to switch between different placement and domain decomposition strategies by merely providing a different mapping function at array creation time.

A Pseudo-Code for LU Benchmark

Figure 5 shows pseudo-code for the cube chare in the LU benchmark. This pseudo-code is written using a notation called Structured Dagger [5], which is a coordination language built on top on Charm++.

References

[1] D. Bailey et al. The NAS Parallel Benchmarks. *Intl. Journal of Supercomputer Applications*, 5(3), 1991.

[2] J. Bruno and P. Capello. Implementing the Beam and Warming method on the hypercube. In *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, Jan. 1988.

[3] D. Bailey, J. Barton, T. Lasinski and H. Simon, editors. The NAS Parallel Benchmarks. *NASA Technical Memorandum 103863, NASA Ames Research Center*, July 1993.

```
chare cube
{
   //chare-local variables declarations

   structentry iterations:  (InitMessage *message)
   {
      atomic {
         Initialization();
         // Send Startup Messages Containing Boundary Elements
         // To Neighbors' rhs_entry points
      }
      while(iter<maxiter) {
         atomic { rhs_init(); }
         overlap {
            when rhs_entry_xm1(Bdry *xm1),rhs_entry_xp1(Bdry *xp1) {
               rhs_x(xm1,xp1);}
            when rhs_entry_ym1(Bdry *ym1),rhs_entry_yp1(Bdry *yp1) {
               rhs_y(ym1,yp1);}
            when rhs_entry_zm1(Bdry *zm1),rhs_entry_zp1(Bdry *zp1) {
               rhs_z(zm1,zp1);}
         if(x==0 && y==0 && z==0) {
            // Start the first sweep By sending messages to +1 neighbors
         }
         when XmBdry(Bdry *xmmsg),YmBdry(Bdry *ymmsg),ZmBdry(Bdry *zmmsg) {
            atomic {
               blts(); jacld();
               // Continue the sweep by sending messages to +1 neighbors'
               // XmBdry, YmBdry, ZmBdry entry-points
            }
         }
         if(x==maxx && y==maxy && z==maxz) {
            // Start the reverse sweep by sending messages to -1 neighbors
         }
         when XpBdry(Bdry *xmsg),YpBdry(Bdry *ymsg),ZpBdry(Bdry *zmsg) {
            atomic {
               buts(); jacu();
               //Continue reverse sweep by sending messages to -1 neighbors'
               // XpBdry, YpBdry, ZpBdry entry-points
               updateu();
               // Send updated boundaries to neighbors' rhs_entry points
               iter++;
            }
         }
}
```

Figure 5: LU Benchmark Program

[4] L. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Proces sing*, Aug. 1990.

[5] L. Kale and M. Bhandarkar. Structured dagger: A coordination language for message-driven programming. In *EUROPAR, '96*, August 1996.

[6] L. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[7] L. Kale and S. Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993.

[8] L. V. Kale and S. Krishnan. Charm++ : Parallel Programming with Message-Driven Objects. *in Parallel Programming using C++*, MIT Press, 1995. To be published.

[9] S. Krishnan and L. V. Kale. A parallel array abstraction for data-driven objects. In *Proceedings of the Parallel Object-Oriented Methods and Applications Conference*, Feb. 1996.

[10] N. H. Naik, V. K. Naik, and M. Nicoules. Parallelization of a class of implicit finite difference schemes in computational fluid dynamics. *International Journal of High Speed Computing*, 5(1), 1993.

[11] R. van der Wijngaart. Efficient Implementation of a 3-Dimensional ADI Method on the iPSC/860. In *Proceedings of Supercomputing 1993*, Nov. 1993.

NAS SP Benchmark Source Code

Sanjeev Krishnan, Milind Bhandarkar and Laxmikant V. Kalé
Dept of Computer Science, University of Illinois, Urbana, IL 61801
E-mail: {sanjeev,milind,kale}@cs.uiuc.edu

```
#include <stdio.h>
#include <string.h>

#include "parray.h"
#include "cube.h"
#include "controlboc.h"
#include "main.h"

extern readonly Cube group cubearray ;
extern readonly int NumCubes ;

char str[1024] ;

extern readonly ControlBoc group controlboc ;

extern "C" double sqrt(double) ;

extern "C" void getns_(int *,int *,int *, int *) ;
extern "C" void initsp_(int *) ;
extern "C" void cpphix_(double *,double *, int *, int *) ;
extern "C" void cpphiy_(double *,double *, int *, int *) ;
extern "C" void cpphiz_(double *,double *, int *, int *) ;
extern "C" void gphi1x_(double *) ;
extern "C" void gphi2x_(double *) ;
extern "C" void gphi1y_(double *) ;
extern "C" void gphi2y_(double *) ;
extern "C" void gphi1z_(double *) ;
extern "C" void gphi2z_(double *) ;
extern "C" void gintgr_() ;
extern "C" void callv_(double *, double *) ;


extern int IsweepMapfn(int gid, int i, int j, int k) ;
extern int JsweepMapfn(int gid, int i, int j, int k) ;
extern int KsweepMapfn(int gid, int i, int j, int k) ;




ControlBoc::ControlBoc(SomeMsg *m)
{
  cubearray = 0 ;                          /* this is because cubearray is created AFTER
        main::main to avoid initialization problems */

  myhandle = thisgroup ;

  MYPE = CMyPeNum() ;
  NUMPROCS = CMaxPeNum() ;
  istep = 1 ;

  initsp_(&MYPE) ;

  getns_(&nx,&ny,&nz,&itmax) ;
```



```
  NumRSDNorms = 0 ;
  for ( int i=0; i<5; i++ )
    TotRSDNorm[i] = 0.0 ;

  NumCubesDone = 0 ;
  NumProcsDone = 0 ;

  NumPhi1x = 0 ;NumPhi1y = 0 ;NumPhi1z = 0 ;
  NumPhi2x = 0 ;NumPhi2y = 0 ;NumPhi2z = 0 ;
  NumPhis = 0 ;

  phi1x = new double [ny*nz] ;
  phi1y = new double [nx*nz] ;
  phi1z = new double [ny*nx] ;
  phi2x = new double [ny*nz] ;
  phi2y = new double [nx*nz] ;
  phi2z = new double [ny*nx] ;

  lasttime = 0 ;

  mainhandle=>InitialSync(m) ;
}



void
ControlBoc::InitialSync(SomeMsg *m)
{
  NumCubesDone++ ;

  if ( NumCubesDone < NumCubes/CNumPes() ) {
    delete m ;
    return ;
  }
  NumCubesDone = 0 ;

  myhandle[0]=>InitialSync0(m) ;
}


void
ControlBoc::InitialSync0(SomeMsg *m)
{
  NumProcsDone++ ;

  if ( NumProcsDone < CNumPes() ) {
    delete m ;
    return ;
  }
  NumProcsDone = 0 ;

  /* Start Timer */
  begintime = CTimer() ;
```

```
  /* Start first iteration */
  SendArrayRange(cubearray, _CK_ep_Cube_StartRHS_I, m, -1, -1, -1, -1 , -1, -1) ;
}



/****** CODE FOR RHS ******/

void
ControlBoc::EndRHS_ISync(SomeMsg *m)
{
  NumCubesDone++ ;
  if ( NumCubesDone < NumCubes ) {
    delete m ;
    return ;
  }
  NumCubesDone = 0 ;


  SendArrayRange(cubearray, _CK_ep_Cube_StartRHS_J, m, -1, -1, -1, -1 , -1, -1) ;
}


void
ControlBoc::EndRHS_JSync(SomeMsg *m)
{
  NumCubesDone++ ;
  if ( NumCubesDone < NumCubes ) {
    delete m ;
    return ;
  }
  NumCubesDone = 0 ;

  SendArrayRange(cubearray, _CK_ep_Cube_StartRHS_K, m, -1, -1, -1, -1 , -1, -1) ;
}


/******* CODE FOR L2Norm **********/

void
ControlBoc::GlobalRSDNorms(NormsMsg *m)
{
  /* compute global sum and actual norm and print it */
  for ( int i=0; i<5; i++ )
    TotRSDNorm[i] += m->sum[i] ;
  delete m ;
  NumRSDNorms++ ;

  if ( NumRSDNorms < NumCubes ) {
    return ;
  }
```



```
  for ( i=0; i<5; i++ )
    TotRSDNorm[i] = sqrt( TotRSDNorm[i] /
          ( (nx-2) * (ny-2) * (nz-2) ) ) ;

  memcpy(LatestRSDNorm, TotRSDNorm, sizeof(double)*5) ;

  NumRSDNorms = 0 ;
  for ( i=0; i<5; i++ )
    TotRSDNorm[i] = 0.0 ;


  SomeMsg *m2 = new SomeMsg ;
  SendArrayRange(cubearray, _CK_ep_Cube_StartADI_Isweep, m2, -1, -1, -1, -1 , -1, -1) ;
}


/******* CODE FOR ADI LOOP **********/

void
ControlBoc::EndADI_ISync(SomeMsg *m)
{
  NumCubesDone++ ;
  if ( NumCubesDone < YArraySize*ZArraySize ) {
    // Only cubes for which myx==0 send a sync msg here
    delete m ;
    return ;
  }
  NumCubesDone = 0 ;

#ifdef TRANSPOSE
  ArrayRemap(cubearray, JsweepMapfn, &(ControlBoc::FinishedJsweepRemap).
      thishandle) ;
#else
  SendArrayRange(cubearray, _CK_ep_Cube_StartADI_Jsweep, m, -1, -1, -1, -1 , -1, -1) ;
#endif
}

void
ControlBoc::FinishedJsweepRemap(SomeMsg *m)
{

  SendArrayRange(cubearray, _CK_ep_Cube_StartADI_Jsweep, m, -1, -1, -1, -1 , -1, -1) ;
}


void
ControlBoc::EndADI_JSync(SomeMsg *m)
{
  NumCubesDone++ ;
  if ( NumCubesDone < XArraySize*ZArraySize ) {
    // Only cubes for which myy==0 send a sync msg here
    delete m ;
    return ;
  }
```

```
  NumCubesDone = 0 ;

#ifdef TRANSPOSE
  ArrayRemap(cubearray, KsweepMapfn, &(ControlBoc::FinishedKsweepRemap),
        thishandle) ;
#else
  SendArrayRange(cubearray, _CK_ep_Cube_StartADI_Ksweep, m, -1, -1, -1, -1 , -1, -1) ;
#endif
}

void
ControlBoc::FinishedKsweepRemap(SomeMsg *m)
{

  SendArrayRange(cubearray, _CK_ep_Cube_StartADI_Ksweep, m, -1, -1, -1, -1 , -1, -1) ;
}


void
ControlBoc::EndADI_KSync(SomeMsg *m)
{
  NumCubesDone++ ;
  if ( NumCubesDone < XArraySize*YArraySize ) {
    // Only cubes for which myz==0 send a sync msg here
    delete m ;
    return ;
  }
  NumCubesDone = 0 ;

#ifdef TRANSPOSE
  ArrayRemap(cubearray, IsweepMapfn, &(ControlBoc::FinishedIsweepRemap),
        thishandle) ;
#else
  SendArrayRange(cubearray, _CK_ep_Cube_StartRHS_I, m, -1, -1, -1, -1 , -1, -1) ;
#endif
}


void
ControlBoc::FinishedIsweepRemap(SomeMsg *m)
{

  SendArrayRange(cubearray, _CK_ep_Cube_StartRHS_I, m, -1, -1, -1, -1 , -1, -1) ;
}



/****  CODE FOR computing the norms of pseudo-time iteration corrections ***/

void
ControlBoc::GlobalCNorms(NormsMsg *m)
{
  /* compute global sum and actual norm and print it */
  for ( int i=0; i<5; i++ )
```



```
    TotRSDNorm[i] += m->sum[i] ;
  NumRSDNorms++ ;
  delete m ;

  if ( NumRSDNorms < NumCubes )
    return ;

  for ( i=0; i<5; i++ )
    TotRSDNorm[i] = sqrt( TotRSDNorm[i] /
          ( (nx-2) * (ny-2) * (nz-2) ) ) ;

  NumRSDNorms = 0 ;
  for ( i=0; i<5; i++ )
    TotRSDNorm[i] = 0.0 ;

  SomeMsg *m2 = new SomeMsg ;
  SendArrayRange(cubearray, _CK_ep_Cube_StartRHS_I, m2, -1, -1, -1, -1 , -1, -1) ;

}



/***** CODE FOR STUFF AFTER FINISHING ADI LOOP ****/

void
ControlBoc::FinalSync(SomeMsg *m)
{
  delete m ;
  NumCubesDone++ ;

  if ( NumCubesDone < NumCubes )
    return ;

  /* End Timer */
  int endtime = CTimer() ;
  CPrintf("\n\nTime for %d iterations:  %d millisec, begin %d end %d\n\n",
        itmax, endtime-begintime, begintime, endtime) ;

  NumCubesDone = 0 ;

  SomeMsg *m2 = new SomeMsg ;
  SendArrayRange(cubearray, _CK_ep_Cube_FinishedADILoop, m2, -1, -1, -1, -1 , -1, -1) ;
}


void
ControlBoc::GlobalError(NormsMsg *m)
{
  /* compute global sum and error norm and print it */
  for ( int i=0; i<5; i++ )
    TotRSDNorm[i] += m->sum[i] ;
  delete m ;
  NumRSDNorms++ ;
```

```
  if ( NumRSDNorms < NumCubes )
    return ;

  for ( i=0; i<5; i++ )
    TotRSDNorm[i] = sqrt( TotRSDNorm[i] /
          ( (nx-2) * (ny-2) * (nz-2) ) ) ;

  CPrintf("RMS-norm of error in soln.  to \n") ;
  sprintf(str,"first pde = %.5e\n",TotRSDNorm[0]) ;
  CPrintf("%s",str) ;
  sprintf(str,"second pde = %.5e\n",TotRSDNorm[1]) ;
  CPrintf("%s",str) ;
  sprintf(str,"third pde = %.5e\n",TotRSDNorm[2]) ;
  CPrintf(str,"%s",str) ;
  sprintf(str,"fourth pde = %.5e\n",TotRSDNorm[3]) ;
  CPrintf("%s",str) ;
  sprintf(str,"fifth pde = %.5e\n",TotRSDNorm[4]) ;
  CPrintf("%s",str) ;

  NumRSDNorms = 0 ;


  CharmExit() ;

}



void
ControlBoc::ReceivePhi1x(PhiMsg *m)
{
  /* Store the phis */
  int begin1 = m->myy * (ny/YArraySize) ;
  int begin2 = m->myz * (nz/ZArraySize) ;
  cpphix_(phi1x,m->phi,&begin1,&begin2) ;

  delete m ;
  NumPhi1x++ ;
  if ( NumPhi1x < YArraySize*ZArraySize )
    return ;

  gphi1x_(phi1x) ;

  GlobalPintgr() ;
}

void
ControlBoc::ReceivePhi2x(PhiMsg *m)
{
  /* Store the phis */
  int begin1 = m->myy * (ny/YArraySize) ;
  int begin2 = m->myz * (nz/ZArraySize) ;
  cpphix_(phi2x,m->phi,&begin1,&begin2) ;
```



```
  delete m ;
  NumPhi2x++ ;
  if ( NumPhi2x < YArraySize*ZArraySize )
    return ;

  gphi2x_(phi2x) ;

  GlobalPintgr() ;
}

void
ControlBoc::ReceivePhi1y(PhiMsg *m)
{
  /* Store the phis */
  int begin1 = m->myx * (nx/XArraySize) ;
  int begin2 = m->myz * (nz/ZArraySize) ;
  cpphiy_(phi1y,m->phi,&begin1,&begin2) ;

  delete m ;
  NumPhi1y++ ;
  if ( NumPhi1y < XArraySize*ZArraySize )
    return ;

  gphi1y_(phi1y) ;

  GlobalPintgr() ;
}

void
ControlBoc::ReceivePhi2y(PhiMsg *m)
{
  /* Store the phis */
  int begin1 = m->myx * (nx/XArraySize) ;
  int begin2 = m->myz * (nz/ZArraySize) ;
  cpphiy_(phi2y,m->phi,&begin1,&begin2) ;

  delete m ;
  NumPhi2y++ ;
  if ( NumPhi2y < XArraySize*ZArraySize )
    return ;

  gphi2y_(phi2y) ;

  GlobalPintgr() ;
}

void
ControlBoc::ReceivePhi1z(PhiMsg *m)
{
  /* Store the phis */
  int begin1 = m->myx * (nx/XArraySize) ;
  int begin2 = m->myy * (ny/YArraySize) ;
  cpphiz_(phi1z,m->phi,&begin1,&begin2) ;
```

```
  delete m ;
  NumPhi1z++ ;
  if ( NumPhi1z < XArraySize*YArraySize )
    return ;

  gphi1z_(phi1z) ;

  GlobalPintgr() ;
}

void
ControlBoc::ReceivePhi2z(PhiMsg *m)
{
  /* Store the phis */
  int begin1 = m->myx * (nx/XArraySize) ;
  int begin2 = m->myy * (ny/YArraySize) ;
  cpphiz_(phi2z,m->phi,&begin1,&begin2) ;

  delete m ;
  NumPhi2z++ ;
  if ( NumPhi2z < XArraySize*YArraySize )
    return ;

  gphi2z_(phi2z) ;

  GlobalPintgr() ;
}


void
ControlBoc::GlobalPintgr()
{
  NumPhis++ ;
  if ( NumPhis == 6 ) {
    CPrintf("[%d] In GlobalPintgr\n",MYPE) ;
    gintgr_() ;

    callv_(LatestRSDNorm,TotRSDNorm) ;

    CharmExit() ;
  }

}
```

```
message class NormsMsg {
public: double sum[5] ;
} ;

chare class ControlBoc : public groupmember {

  ControlBoc group myhandle ;

  int istep ;
  int itmax ;                                          // number of ADI iterations

  int MYPE ;
  int NUMPROCS ;

  int nx, ny, nz ;

  int NumRSDNorms ;
  double TotRSDNorm[5] ;
  double LatestRSDNorm[5] ;

  double *phi1x, *phi2x, *phi1y, *phi2y, *phi1z, *phi2z ;
  int NumPhi1x, NumPhi2x, NumPhi1y, NumPhi2y, NumPhi1z, NumPhi2z ;
  int NumPhis ;

  int NumCubesDone ;
  int NumProcsDone ;

  int lasttime ;

  int begintime ;

entry:
  ControlBoc(SomeMsg *m) ;

  void InitialSync(SomeMsg *m) ;
  void InitialSync0(SomeMsg *m) ;

/***** CODE FOR RHS *****/
entry:
  void EndRHS_ISync(SomeMsg *m) ;

  void EndRHS_JSync(SomeMsg *m) ;

  void GlobalRSDNorms(NormsMsg *m) ;


/******* CODE FOR ADI LOOP **********/

entry:
  void EndADI_ISync(SomeMsg *m) ;

  void EndADI_JSync(SomeMsg *m) ;
```



```
  void EndADI_KSync(SomeMsg *m) ;

  void FinishedJsweepRemap(SomeMsg *m) ;

  void FinishedKsweepRemap(SomeMsg *m) ;

  void FinishedIsweepRemap(SomeMsg *m) ;


/****  CODE FOR computing the norms of pseudo-time iteration corrections ***/
entry:
  void GlobalCNorms(NormsMsg *m) ;


/***** CODE FOR STUFF AFTER FINISHING ADI LOOP ****/
entry:
  void FinalSync(SomeMsg *m) ;

  void GlobalError(NormsMsg *m) ;

  void ReceivePhi1x(PhiMsg *m) ;
  void ReceivePhi2x(PhiMsg *m) ;
  void ReceivePhi1y(PhiMsg *m) ;
  void ReceivePhi2y(PhiMsg *m) ;
  void ReceivePhi1z(PhiMsg *m) ;
  void ReceivePhi2z(PhiMsg *m) ;

public:
  void GlobalPintgr() ;
} ;
```

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "parray.h"
#include "cube.h"
#include "controlboc.h"

int nx, ny, nz ;

extern readonly ControlBoc group controlboc ;
extern readonly Cube group cubearray ;


extern "C" void pval_(double *) ;
extern "C" void psval_(int *, double *) ;

extern "C" prsd_(double *) ;
extern "C" pfrct_(double *) ;
extern "C" getns_(int *, int *, int*, int * ) ;
extern "C" setglobals_(int *, int *, int*, int *, int *, int*, int *, int *, int* ) ;
extern "C" setb_(double *, double *, int *) ;
extern "C" setcd_(double *, double *, int *) ;
extern "C" getb_(double *, double *, int *, int *) ;
extern "C" getcd_(double *, double *, int *, int *) ;
extern "C" setbv_(double *,int *, int *, int*, int *, int *, int* ) ;
extern "C" setiv_(double *,int *, int *, int* ) ;
extern "C" erhs_(double *,int *, int *, int*, int *, int *, int* ) ;
extern "C" rhsx_(double *,double *,double *,int *, int *, int*, int *, int *, int* ) ;
extern "C" rhsy_(double *,double *,double *,int *, int *, int*, int *, int *, int* ) ;
extern "C" rhsz_(double *,double *,double *,int *, int *, int*, int *, int *, int* ) ;
extern "C" rnorms_(double *,double *) ;
extern "C" adix_(double *,double *,double *,double *,double *,double *,double *,double *,int
*, int *, int*, int *, int *, int* );
extern "C" badix_(double *,double *,double *,double *,double *,double *,double *,double *,int
*, int *, int*, int *, int *, int* );
extern "C" adiy_(double *,double *,double *,double *,double *,double *,double *,double *,int
*, int *, int*, int *, int *, int* );
extern "C" badiy_(double *,double *,double *,double *,double *,double *,double *,double *,int
*, int *, int*, int *, int *, int* );
extern "C" adiz_(double *,double *,double *,double *,double *,double *,double *,double *,int
*, int *, int*, int *, int *, int* );
extern "C" badiz_(double *,double *,double *,double *,double *,double *,double *,double *,int
*, int *, int*, int *, int *, int* );
extern "C" mvmul_(double *,double *,int *, int *, int*, int *, int *, int* );
extern "C" cnorms_(double *,double *) ;
extern "C" error_(double *,double *,int *, int *, int*, int *, int *, int* ) ;
extern "C" get1x_(double *, double *) ;
extern "C" get2x_(double *, double *) ;
extern "C" get1y_(double *, double *) ;
extern "C" get2y_(double *, double *) ;
extern "C" get1z_(double *, double *) ;
extern "C" get2z_(double *, double *) ;
```



```
extern int IsweepMapfn(int gid, int i, int j, int k) ;
extern int JsweepMapfn(int gid, int i, int j, int k) ;
extern int KsweepMapfn(int gid, int i, int j, int k) ;


void
pval_(double *val)
{
  if ( CMyPe() == 10 )
    CPrintf("[%d] Val is %.12e\n",CMyPe(),*val) ;
}


void
psval_(int *i, double *val)
{
  if ( CMyPe() == 10 )
    CPrintf("[%d] %d %.12e\n",CMyPe(),*i,*val) ;
}


Cube::Cube(SomeMsg *m)
{
  istep = 0 ;
  getns_(&nx,&ny,&nz,&itmax) ;

  /* Set all my local variables */
  myx = thisi ;
  myy = thisj ;
  myz = thisk ;


  eachx = nx/XArraySize ;
  eachy = ny/YArraySize ;
  eachz = nz/ZArraySize ;

  int allocx = eachx + 4 ;                          // two boundary rows on either side
  int allocy = eachy + 4 ;
  int allocz = eachz + 4 ;


  /* Allocate u,rsd,frct in message to prevent copying while migrating */
  int cubesize = (sizeof(Cube)/8 + 1) * 8 ;          // make double word boundary
  int totsize = cubesize ;
  int eacharray = 5 * allocx * allocy * allocz * sizeof(double) ;
  totsize += 3*eacharray ;
  dataarea = new (&totsize) PackMsg ;
  char *ptr = dataarea->data + cubesize ;
  u = (double *)ptr ;
  ptr += eacharray ;
  rsd = (double *)ptr ;
  ptr += eacharray ;
  frct = (double *)ptr ;
```

```
  /* Allocate space for all local arrays */
  a = new double[3*allocx*allocy*allocz] ;
  b = new double[3*allocx*allocy*allocz] ;
  c = new double[3*allocx*allocy*allocz] ;
  d = new double[3*allocx*allocy*allocz] ;
  e = new double[3*allocx*allocy*allocz] ;

  memset(u, 0, 5*allocx*allocy*allocz*sizeof(double)) ;
  memset(rsd, 0, 5*allocx*allocy*allocz*sizeof(double)) ;
  memset(frct, 0, 5*allocx*allocy*allocz*sizeof(double)) ;
  memset(a, 0, 3*allocx*allocy*allocz*sizeof(double)) ;
  memset(b, 0, 3*allocx*allocy*allocz*sizeof(double)) ;
  memset(c, 0, 3*allocx*allocy*allocz*sizeof(double)) ;
  memset(d, 0, 3*allocx*allocy*allocz*sizeof(double)) ;
  memset(e, 0, 3*allocx*allocy*allocz*sizeof(double)) ;

  /* set global indices of my array section */
  gstx = eachx * myx + 1 ;
  gendx = gstx + eachx - 1 ;
  gsty = eachy * myy + 1 ;
  gendy = gsty + eachy - 1 ;
  gstz = eachz * myz + 1 ;
  gendz = gstz + eachz - 1 ;

  igstx = &gstx ;
  igendx = &gendx ;
  igsty = &gsty ;
  igendy = &gendy ;
  igstz = &gstz ;
  igendz = &gendz ;

  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;

  NumBoundaries = 0 ;


  BvIvErhs() ;                                                    // NEW

  controlboc[LOCAL]->InitialSync(m) ;

  DoneRHSI = DoneRHSJ = DoneADII = DoneADIJ = 0 ;
  InitializedIsweep = InitializedJsweep = InitializedKsweep = 0 ;
  InitializedRHS_I = InitializedRHS_J = InitializedRHS_K = 0 ;
}

void PrintArray(char *, double *, int) ;



void
Cube::SetBoundary(BoundaryMsg *m1)
{
  if ( m1 == NULL )
```



```
    CPrintf("[%d] ERROR: m1 == NULL\n",CMyPe()) ;

  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;

  switch( m1->whicharrays ) {
      case U_ONLY :
      {
    UBoundaryMsg *m = (UBoundaryMsg *) m1 ;
    setb_(u, m->u, &(m->type)) ;
      }
      break ;

      case RSD_ONLY :
      {
    RSDBoundaryMsg *m = (RSDBoundaryMsg *) m1 ;
    setb_(rsd, m->rsd, &(m->type)) ;
      }
      break ;

      case ALL_ARRAYS :
      {
    AllBoundaryMsg *m = (AllBoundaryMsg *) m1 ;

    setb_(u, m->u, &(m->type)) ;
    setb_(rsd, m->rsd, &(m->type)) ;

    setcd_(c, m->c, &(m->type)) ;
    setcd_(d, m->d, &(m->type)) ;
      }
      break ;
  }

      delete m1 ;
}

BoundaryMsg *
Cube::GetBoundary(int type, int whicharrays)
{
  int size ;

  switch ( type ) {
    case IPREV :
    case INEXT :
      size = eachy * eachz * 2 * 5 ;                              // 5 arrays
      break ;
    case JPREV :
    case JNEXT :
      size = eachx * eachz * 2 * 5 ;
      break ;
    case KPREV :
    case KNEXT :
      size = eachy * eachx * 2 * 5 ;
      break ;
  }
```

```
  BoundaryMsg *m1 ;

  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;

  switch( whicharrays ) {
      case U_ONLY :
      {
    UBoundaryMsg *m = new (&size) UBoundaryMsg ;
    int count=1 ;
    getb_(u, m->u, &type, &count) ;
    m1 = m ;
      }
      break ;

      case RSD_ONLY :
      {
    RSDBoundaryMsg *m = new (&size) RSDBoundaryMsg ;
    int count=1 ;
    getb_(rsd, m->rsd, &type, &count) ;
    m1 = m ;
      }
      break ;

      case ALL_ARRAYS :
      {
    int sizes[4] ;
    sizes[0] = sizes[1] = size ;
    sizes[2] = sizes[3] = size/5*3 ;                    // c, d have only 3 subarrays

    AllBoundaryMsg *m = new (sizes) AllBoundaryMsg ;
    int count=1 ;
    getb_(u, m->u, &type, &count) ;
    count = 1 ;
    getb_(rsd, m->rsd, &type, &count) ;

    count = 1 ;
    getcd_(c, m->c, &type, &count) ;
    count = 1 ;
    getcd_(d, m->d, &type, &count) ;
    m1 = m ;
      }
      break ;
  }

  m1->type = type ;
  m1->whicharrays = whicharrays ;

  return m1 ;
}
```



```
void
Cube::BvIvErhs()
{
  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;

  setbv_(u,igstx, igendx, igsty, igendy, igstz, igendz) ;
  setiv_(u,igstx,igsty,igstz) ;

  erhs_(frct,igstx, igendx, igsty, igendy, igstz, igendz) ;
}


/****************** THESE ROUTINES ARE FOR SENDING BOUNDARIES ************/

void
Cube::SendPrevI(int type)
{
  if ( myx == 0 ) {
    NumBoundaries++ ;
    return ;
  }

        BoundaryMsg *m = GetBoundary(IPREV,U_ONLY) ;

  if ( type == ADI )
    SendArray(cubearray, _CK_ep_Cube_DoADI_Isweep, m, myx-1, myy, myz) ;
  else
    SendArray(cubearray, _CK_ep_Cube_DoRHS_I, m, myx-1, myy, myz) ;
}

void
Cube::SendNextI()
{
  if ( myx == XArraySize-1 ) {
    NumBoundaries++ ;
    return ;
  }

        BoundaryMsg *m = GetBoundary(INEXT,U_ONLY) ;
  SendArray(cubearray, _CK_ep_Cube_DoRHS_I, m, myx+1, myy, myz) ;
}


void
Cube::SendPrevJ(int type)
{
  if ( myy == 0 ) {
    NumBoundaries++ ;
    return ;
  }

        BoundaryMsg *m = GetBoundary(JPREV,U_ONLY) ;
```

```
  if ( type == ADI )
    SendArray(cubearray, _CK_ep_Cube_DoADI_Jsweep, m, myx, myy-1, myz) ;
  else
    SendArray(cubearray, _CK_ep_Cube_DoRHS_J, m, myx, myy-1, myz) ;
}

void
Cube::SendNextJ()
{
  if ( myy == YArraySize-1 ) {
    NumBoundaries++ ;
    return ;
  }

        BoundaryMsg *m = GetBoundary(JNEXT,U_ONLY) ;
  SendArray(cubearray, _CK_ep_Cube_DoRHS_J, m, myx, myy+1, myz) ;
}


void
Cube::SendPrevK(int type)
{
  if ( myz == 0 ) {
    NumBoundaries++ ;
    return ;
  }

        BoundaryMsg *m = GetBoundary(KPREV,U_ONLY) ;

  if ( type == ADI )
    SendArray(cubearray, _CK_ep_Cube_DoADI_Ksweep, m, myx, myy, myz-1) ;
  else
    SendArray(cubearray, _CK_ep_Cube_DoRHS_K, m, myx, myy, myz-1) ;
}

void
Cube::SendNextK()
{
  if ( myz == ZArraySize-1 ) {
    NumBoundaries++ ;
    return ;
  }

        BoundaryMsg *m = GetBoundary(KNEXT,U_ONLY) ;
  SendArray(cubearray, _CK_ep_Cube_DoRHS_K, m, myx, myy, myz+1) ;
}


/****************** THESE ROUTINES ARE FOR THE RHS SWEEPS ************/

void
Cube::StartRHS_I(SomeMsg *m)
{
```



```
  istep++ ;
  if ( istep > itmax ) {
    SomeMsg *m2 = new SomeMsg ;
    controlboc[0]=>FinalSync(m2) ;
    return ;
  }


  SendPrevI(RHS) ;
  SendNextI() ;

  InitializedRHS_I = 1 ;

        if ( XArraySize == 1 )
                DoRHS_I(NULL) ;

}

void
Cube::DoRHS_I(BoundaryMsg *m)
{
  if ( InitializedRHS_I == 0 ) {
    SendArray(thisgroup, _CK_ep_Cube_DoRHS_I, m, myx, myy, myz) ;
    return ;
  }
  if ( NumBoundaries < 2 ) {
    SetBoundary(m) ;

    NumBoundaries++ ;

    if ( NumBoundaries < 2 )
      return ;
  }

  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;

  rhsx_(u,rsd,frct,igstx, igendx, igsty, igendy, igstz, igendz) ;

  DoneRHSI = 1 ;
  NumBoundaries = 0 ;
  InitializedRHS_I = 0 ;

  StartRHS_J(NULL) ;
}

void
Cube::StartRHS_J(SomeMsg *m)
{
  SendPrevJ(RHS) ;
  SendNextJ() ;

  InitializedRHS_J = 1 ;

        if ( YArraySize == 1 )
```

```
                DoRHS_J(NULL) ;
}

void
Cube::DoRHS_J(BoundaryMsg *m)
{
  if ( DoneRHSI == 0 || InitializedRHS_J == 0 ) {
    SendArray(thisgroup, _CK_ep_Cube_DoRHS_J, m, myx, myy, myz) ;
    return ;
  }
  if ( NumBoundaries < 2 ) {
    SetBoundary(m) ;

    NumBoundaries++ ;

    if ( NumBoundaries < 2 )
      return ;
  }

  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;

  rhsy_(u,rsd,frct,igstx, igendx, igsty, igendy, igstz, igendz) ;

  DoneRHSJ = 1 ;
  DoneRHSI = 0 ;                                          // reset for next itn
  NumBoundaries = 0 ;
  InitializedRHS_J = 0 ;

  StartRHS_K(NULL) ;
}


void
Cube::StartRHS_K(SomeMsg *m)
{
  SendPrevK(RHS) ;
  SendNextK() ;

  InitializedRHS_K = 1 ;

        if ( ZArraySize == 1 )
                DoRHS_K(NULL) ;
}


void
Cube::DoRHS_K(BoundaryMsg *m)
{
  if ( DoneRHSJ == 0 || InitializedRHS_K == 0 ) {
    SendArray(thisgroup, _CK_ep_Cube_DoRHS_K, m, myx, myy, myz) ;
    return ;
  }
  if ( NumBoundaries < 2 ) {
    SetBoundary(m) ;
```



```
    NumBoundaries++ ;

    if ( NumBoundaries < 2 )
      return ;
  }
  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;

  rhsz_(u,rsd,frct,igstx, igendx, igsty, igendy, igstz, igendz) ;

  DoneRHSJ = 0 ;                                          // reset for next itn
  NumBoundaries = 0 ;
  InitializedRHS_K = 0 ;

  StartADI_Isweep(NULL) ;


}


/****************** THIS ROUTINE IS FOR LOCAL RSD NORMS *************/
void
Cube::LocalRSDNorms()
{
  NormsMsg *m = new NormsMsg ;
  for ( int i=0; i<5; i++ )
                m->sum[i] = 0.0 ;


  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;
  rnorms_(rsd,m->sum) ;

  controlboc[0]=>GlobalRSDNorms(m) ;
}


/****************** THESE 3 ROUTINES ARE FOR THE ADI SWEEPS *************/

void
Cube::StartADI_Isweep(SomeMsg *msg)
{
  SendPrevI(ADI) ;

  if ( myx == XArraySize-1 )
    NumBoundaries++ ;

  InitializedIsweep = 1 ;

  if ( XArraySize == 1 )
    DoADI_Isweep(NULL) ;
}


void
```

```
Cube::DoADI_Isweep(BoundaryMsg *m)
{
  if ( InitializedIsweep == 0 ) {
    SendArray(thisgroup, _CK_ep_Cube_DoADI_Isweep, m, myx, myy, myz) ;
    return ;
  }

  if ( NumBoundaries < 2 && m ≠ NULL ) {
    SetBoundary(m) ;

    NumBoundaries++ ;

    if ( NumBoundaries < 2 )
      return ;
  }
  NumBoundaries = 0 ;                                          // reinit
  InitializedIsweep = 0 ;                                  // reset for next itn

  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;
  adix_(u,rsd,frct,a,b,c,d,e,igstx, igendx, igsty, igendy, igstz, igendz) ;

  if ( myx ≠ XArraySize-1 ) {
    BoundaryMsg *m2 = GetBoundary(INEXT,ALL_ARRAYS) ;
    SendArray(cubearray, _CK_ep_Cube_DoADI_Isweep, m2, myx+1, myy, myz) ;
  }
  else {
    ADIback_Isweep(NULL) ;
  }
}


void
Cube::ADIback_Isweep(BoundaryMsg *m)
{
  if ( myx ≠ XArraySize - 1 ) {
    SetBoundary(m) ;
  }

  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;
  badix_(u,rsd,frct,a,b,c,d,e,igstx, igendx, igsty, igendy, igstz, igendz) ;

  if ( myx ≠ 0 ) {
    BoundaryMsg *m2 = GetBoundary(IPREV,RSD_ONLY) ;
    SendArray(cubearray, _CK_ep_Cube_ADIback_Isweep, m2, myx-1, myy, myz) ;
  }
#ifndef NOSYNCS
  else {
    SomeMsg *m2 = new SomeMsg ;
    controlboc[0]=>EndADI_ISync(m2) ;
  }
#endif

  DoneADII = 1 ;
```



```
#ifdef NOSYNCS
  SomeMsg *mm = new SomeMsg ;
  SendArray(thisgroup, _CK_ep_Cube_StartADI_Jsweep, mm, myx, myy, myz) ;

#ifdef TRANSPOSE
  migrate(JsweepMapfn) ;
#endif
#endif
}



void
Cube::StartADI_Jsweep(SomeMsg *msg)
{
  SendPrevJ(ADI) ;

  if ( myy == YArraySize-1 )
    NumBoundaries++ ;

  InitializedJsweep = 1 ;

  if ( YArraySize == 1 )
    DoADI_Jsweep(NULL) ;
}



void
Cube::DoADI_Jsweep(BoundaryMsg *m)
{

  if ( DoneADII == 0 || InitializedJsweep == 0 ) {
    SendArray(thisgroup, _CK_ep_Cube_DoADI_Jsweep, m, myx, myy, myz) ;
    return ;
  }

  if ( NumBoundaries < 2 && m ≠ NULL ) {
    SetBoundary(m) ;

    NumBoundaries++ ;

    if ( NumBoundaries < 2 )
      return ;
  }
  NumBoundaries = 0 ;                                          // reinit
  DoneADII = 0 ;                                           // reset for next itn
  InitializedJsweep = 0 ;                                  // reset for next itn

  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;
  adiy_(u,rsd,frct,a,b,c,d,e,igstx, igendx, igsty, igendy, igstz, igendz) ;

  if ( myy ≠ YArraySize-1 ) {
    BoundaryMsg *m2 = GetBoundary(JNEXT,ALL_ARRAYS) ;
```

```
    SendArray(cubearray, _CK_ep_Cube_DoADI_Jsweep, m2, myx, myy+1, myz) ;
  }
  else {
    ADIback_Jsweep(NULL) ;
  }
}


void
Cube::ADIback_Jsweep(BoundaryMsg *m)
{
  if ( myy ≠ YArraySize - 1 ) {
    SetBoundary(m) ;
  }

  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;
  badiy_(u,rsd,frct,a,b,c,d,e,igstx, igendx, igsty, igendy, igstz, igendz) ;

  if ( myy ≠ 0 ) {
    BoundaryMsg *m2 = GetBoundary(JPREV,RSD_ONLY) ;
    SendArray(cubearray, _CK_ep_Cube_ADIback_Jsweep, m2, myx, myy-1, myz) ;
  }
#ifndef NOSYNCS
  else {
    SomeMsg *m2 = new SomeMsg ;
    controlboc[0]=>EndADI_JSync(m2) ;
  }
#endif

  DoneADIJ = 1 ;

#ifdef NOSYNCS
  SomeMsg *mm = new SomeMsg ;
  SendArray(thisgroup, _CK_ep_Cube_StartADI_Ksweep, mm, myx, myy, myz) ;

#ifdef TRANSPOSE
  migrate(KsweepMapfn) ;
#endif
#endif
}



void
Cube::StartADI_Ksweep(SomeMsg *msg)
{
  SendPrevK(ADI) ;

  if ( myz == ZArraySize-1 )
    NumBoundaries++ ;

  InitializedKsweep = 1 ;

  if ( ZArraySize == 1 )
```



```
    DoADI_Ksweep(NULL) ;
}


void
Cube::DoADI_Ksweep(BoundaryMsg *m)
{
  if ( DoneADIJ == 0 || InitializedKsweep == 0 ) {
    SendArray(thisgroup, _CK_ep_Cube_DoADI_Ksweep, m, myx, myy, myz) ;
    return ;
  }

  if ( NumBoundaries < 2 && m ≠ NULL ) {
    SetBoundary(m) ;

    NumBoundaries++ ;

    if ( NumBoundaries < 2 )
      return ;
  }
  NumBoundaries = 0 ;                                          // reinit
  DoneADIJ = 0 ;                                          // reset for next itn
  InitializedKsweep = 0 ;                                 // reset for next itn


  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;
  adiz_(u,rsd,frct,a,b,c,d,e,igstx, igendx, igsty, igendy, igstz, igendz) ;

  if ( myz ≠ ZArraySize-1 ) {
    BoundaryMsg *m2 = GetBoundary(KNEXT,ALL_ARRAYS) ;
    SendArray(cubearray, _CK_ep_Cube_DoADI_Ksweep, m2, myx, myy, myz+1) ;
  }
  else
    ADIback_Ksweep(NULL) ;
}


void
Cube::ADIback_Ksweep(BoundaryMsg *m)
{
  if ( myz ≠ ZArraySize - 1 ) {
    SetBoundary(m) ;
  }

  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;
  badiz_(u,rsd,frct,a,b,c,d,e,igstx, igendx, igsty, igendy, igstz, igendz) ;


  if ( myz ≠ 0 ) {
    BoundaryMsg *m2 = GetBoundary(KPREV,RSD_ONLY) ;
    SendArray(cubearray, _CK_ep_Cube_ADIback_Ksweep, m2, myx, myy, myz-1) ;
  }
#ifndef NOSYNCS
  else {
```

```
    SomeMsg *m2 = new SomeMsg ;
    controlboc[0]=>EndADI_KSync(m2) ;
  }
#endif


  mvmul_(u,rsd,igstx, igendx, igsty, igendy, igstz, igendz) ;


#ifdef NOSYNCS
  SomeMsg *mm = new SomeMsg ;
  SendArray(thisgroup, _CK_ep_Cube_StartRHS_I, mm, myx, myy, myz) ;

#ifdef TRANSPOSE
  migrate(IsweepMapfn) ;
#endif
#endif
}



/*** THIS ROUTINE IS FOR LOCAL NORMS of pseudo-time iteration corrections****/
void
Cube::LocalCNorms(SomeMsg *m2)
{
  NormsMsg *m = new NormsMsg ;
  for ( int i=0; i<5; i++ )
              m->sum[i] = 0.0 ;

  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;
  cnorms_(rsd,m->sum) ;

  controlboc[0]=>GlobalCNorms(m) ;
}



/***** CODE FOR STUFF AFTER FINISHING ADI LOOP ****/
void
Cube::FinishedADILoop(SomeMsg *m2)
{
        /* This is for the local error */

        NormsMsg *m = new NormsMsg ;
        for ( int i=0; i<5; i++ )
                m->sum[i] = 0.0 ;

  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;
  error_(u,m->sum,igstx, igendx, igsty, igendy, igstz, igendz) ;

  controlboc[0]=>GlobalError(m) ;
}
```



```
/*** THIS ROUTINE IS FOR local pintgr() ****/
void
Cube::LocalPintgr(SomeMsg *m2)
{
/* If Im on a boundary, send phi's to proc 0 */

  setglobals_(&eachx, &eachy, &eachz, igstx, igendx, igsty, igendy, igstz, igendz) ;


  if ( myx == 0 ) {                                   /* Im on lower boundary in x direction */
    int size = (eachz+4) * (eachy+4) ;
    PhiMsg *m = new (&size) PhiMsg;
    m->myx = myx ; m->myy = myy ; m->myz = myz ;
    memset(m->phi,0,sizeof(double)*size) ;
    get1x_(u, m->phi) ;
    controlboc[0]=>ReceivePhi1x(m) ;
  }
  if ( myx == XArraySize-1 ) {
    int size = (eachz+4) * (eachy+4) ;
    PhiMsg *m = new (&size) PhiMsg;
    m->myx = myx ; m->myy = myy ; m->myz = myz ;
    memset(m->phi,0,sizeof(double)*size) ;
    get2x_(u, m->phi) ;
    controlboc[0]=>ReceivePhi2x(m) ;
  }
  if ( myy == 0 ) {                                   /* Im on lower boundary in y direction */
    int size = (eachx+4) * (eachz+4) ;
    PhiMsg *m = new (&size) PhiMsg;
    m->myx = myx ; m->myy = myy ; m->myz = myz ;
    memset(m->phi,0,sizeof(double)*size) ;
    get1y_(u, m->phi) ;
    controlboc[0]=>ReceivePhi1y(m) ;
  }
  if ( myy == YArraySize-1 ) {
    int size = (eachx+4) * (eachz+4) ;
    PhiMsg *m = new (&size) PhiMsg;
    m->myx = myx ; m->myy = myy ; m->myz = myz ;
    memset(m->phi,0,sizeof(double)*size) ;
    get2y_(u, m->phi) ;
    controlboc[0]=>ReceivePhi2y(m) ;
  }
  if ( myz == 0 ) {                                   /* Im on lower boundary in z direction */
    int size = (eachx+4) * (eachy+4) ;
    PhiMsg *m = new (&size) PhiMsg;
    m->myx = myx ; m->myy = myy ; m->myz = myz ;
    memset(m->phi,0,sizeof(double)*size) ;
    get1z_(u, m->phi) ;
    controlboc[0]=>ReceivePhi1z(m) ;
  }
  if ( myz == ZArraySize-1 ) {
    int size = (eachx+4) * (eachy+4) ;
```

```
    PhiMsg *m = new (&size) PhiMsg;
    m→myx = myx ; m→myy = myy ; m→myz = myz ;
    memset(m→phi,0,sizeof(double)*size) ;
    get2z_(u, m→phi) ;
    controlboc[0]=>ReceivePhi2z(m) ;
  }
}



/* These are the pack and unpack functions for the cube chare */

ArrayMsg *
Cube::pack(EntryPointType *unpackep)
{

  *unpackep = &(Cube::Unpack) ;

  memcpy(dataarea→data, this, sizeof(Cube)) ;

  delete a ;
  delete b ;
  delete c ;
  delete d ;
  delete e ;


  return dataarea ;
}


void
Cube::Unpack(PackMsg *p)
{
  char *ptr = p→data ;
  memcpy(this, ptr, sizeof(Cube)) ;


  int allocx = nx/XArraySize + 4 ;                    // two boundary rows on either side
      int allocy = ny/YArraySize + 4 ;
      int allocz = nz/ZArraySize + 4 ;

  /* Allocate space for local arrays */
      a = new double[3*allocx*allocy*allocz] ;
      b = new double[3*allocx*allocy*allocz] ;
      c = new double[3*allocx*allocy*allocz] ;
      d = new double[3*allocx*allocy*allocz] ;
      e = new double[3*allocx*allocy*allocz] ;

      igstx = &gstx ;
      igendx = &gendx ;
      igsty = &gsty ;
      igendy = &gendy ;
      igstz = &gstz ;
```



```
      igendz = &gendz ;

  int cubesize = (sizeof(Cube)/8 + 1) * 8 ;           // make double word boundary
  int eacharray = 5 * allocx * allocy * allocz * sizeof(double) ;
  ptr += cubesize ;
  u = (double *)ptr ;                                 // prevent copying
  ptr += eacharray ;
  rsd = (double *)ptr ;
  ptr += eacharray ;
  frct = (double *)ptr ;

  dataarea = p ;
}



void PrintArray(char *file, double *a, int size)
{
  FILE *fp = fopen(file,"w") ;
  for ( int i=0; i<size; i++ )
    fprintf(fp,"%lf\n",a[i]) ;

  fprintf(fp,"Done !!!\n") ;
  fclose(fp) ;
}
```

```
#define XArraySize 1
#define YArraySize 1
#define ZArraySize 1

#define IPREV 1
#define INEXT 2
#define JPREV 3
#define JNEXT 4
#define KPREV 5
#define KNEXT 6

#define U_ONLY 1
#define RSD_ONLY 2
#define ALL_ARRAYS 3

#define ADI 123
#define RHS 456


message class SomeMsg : public ArrayMsg {
public: int x ;
} ;

message class BoundaryMsg : public ArrayMsg {
public: int type ;
  int whicharrays ;
} ;

message class UBoundaryMsg : public BoundaryMsg {
public: double u[VARSIZE] ;                    // MUST be aligned at 8 byte boundary !
} ;

message class RSDBoundaryMsg : public BoundaryMsg {
public: double rsd[VARSIZE] ;
} ;

message class AllBoundaryMsg : public BoundaryMsg {
public: double u[VARSIZE] ;
  double rsd[VARSIZE] ;
  double c[VARSIZE] ;
  double d[VARSIZE] ;
} ;

message class PhiMsg {
public: int myx, myy, myz ;
  int type ;
  double phi[VARSIZE] ;
} ;


message class PackMsg : public ArrayMsg {
public: char data[VARSIZE] ;
} ;
```



```
chare class Cube : public array {
public:
  int istep, itmax ;                                   // number of iterations
  int myx, myy, myz ;                                  /* my coordinates */

  int gstx, gsty, gstz ;
  int gendx, gendy, gendz ;

  int *igstx, *igendx, *igsty, *igendy, *igstz, *igendz ;

  int eachx, eachy, eachz ;

  int NumBoundaries ;

  PackMsg *dataarea ;

  double *u, *rsd, *frct ;
  double *a, *b,*c,*d,*e;

  int DoneRHSI, DoneRHSJ, DoneADII, DoneADIJ ;
  int InitializedIsweep, InitializedJsweep, InitializedKsweep ;
  int InitializedRHS_I, InitializedRHS_J, InitializedRHS_K ;

entry:
  Cube(SomeMsg *m) ;

public:
  void SetBoundary(BoundaryMsg *m1) ;

  BoundaryMsg * GetBoundary(int type, int whicharrays) ;

  void BvIvErhs() ;

  void SendPrevI(int) ;

  void SendNextI() ;

  void SendPrevJ(int) ;

  void SendNextJ() ;

  void SendPrevK(int) ;

  void SendNextK() ;

  void LocalRSDNorms() ;


/****************** THESE 3 EPs ARE FOR THE RHS SWEEPS ************/
entry:
  void StartRHS_I(SomeMsg *m) ;
```

```
    void DoRHS_I(BoundaryMsg *m) ;

    void StartRHS_J(SomeMsg *m) ;

    void DoRHS_J(BoundaryMsg *m) ;

    void StartRHS_K(SomeMsg *m) ;

    void DoRHS_K(BoundaryMsg *m) ;

/****************** THESE 3 EPs ARE FOR THE ADI SWEEPS *************/
entry:
    void StartADI_Isweep(SomeMsg *msg) ;

    void DoADI_Isweep(BoundaryMsg *m) ;

    void ADIback_Isweep(BoundaryMsg *m) ;

    void StartADI_Jsweep(SomeMsg *msg) ;

    void DoADI_Jsweep(BoundaryMsg *m) ;

    void ADIback_Jsweep(BoundaryMsg *m) ;

    void StartADI_Ksweep(SomeMsg *msg) ;

    void DoADI_Ksweep(BoundaryMsg *m) ;

    void ADIback_Ksweep(BoundaryMsg *m) ;

/*** THIS ROUTINE IS FOR LOCAL NORMS of pseudo-time iteration corrections****/
    void LocalCNorms(SomeMsg *m) ;

    void FinishedADILoop(SomeMsg *) ;

/*** THIS ROUTINE IS FOR local pintgr() ****/
entry:
    void LocalPintgr(SomeMsg *m) ;

/*** These are the pack and unpack functions for the cube chare ****/
public:
    ArrayMsg *pack(EntryPointType *unpackep) ;

entry:
    void Unpack(PackMsg *p) ;
} ;
```

```
#include <stdio.h>
#include "parray.h"
#include "cube.h"
#include "controlboc.h"
#include "main.h"


extern "C" MAIN_() ;

MAIN_()
{}


extern "C" double sqrt(double) ;


readonly ControlBoc group controlboc ;
readonly Cube group cubearray ;
readonly int NumCubes ;


int BrunoCapelloMapFn(int gid, int i, int j, int k)
{
  if ( XArraySize ≠ YArraySize && YArraySize ≠ ZArraySize )
    CPrintf("ERROR: ArraySize is not same in all dims\n") ;

  /* General expression for 3-D Bruno-Capello is :
     Map(i,j,k) = s[(i-k)%s] + (j-k)%s
     where s is the side of the cube
     This is ONLY for the case where ArraySize is sqrt(NumProcs).
     E.g. 64 cubes on 16 procs or 8 cubes on 4 procs */
       return ( XArraySize*((i-k+XArraySize)%XArraySize) +
      (j-k+XArraySize)%XArraySize ) ;


}

int NaiveMapFn(int gid, int i, int j, int k)
{                                                  /* Long blocks given to one proc */

  if ( XArraySize ≠ YArraySize && YArraySize ≠ ZArraySize )
    CPrintf("ERROR: ArraySize is not same in all dims\n") ;

  return ( 2*(i%2) + j%2 ) ;
}



int IsweepMapfn(int gid, int i, int j, int k)
{                                                  /* Long blocks given to one proc */

  return ( ZArraySize * j + k ) ;
}
```



```
int JsweepMapfn(int gid, int i, int j, int k)
{                                                  /* Long blocks given to one proc */

  // This is ONLY for the case where ArraySize is sqrt(NumProcs)
  return ( XArraySize * k + i ) ;
}

int KsweepMapfn(int gid, int i, int j, int k)
{                                                  /* Long blocks given to one proc */

  return ( YArraySize * i + j ) ;
}


/** MAIN CHARE **/

        main::main()
        {
    CPrintf("Started Charm++ pgm, creating ControlBoc\n") ;

    NumCubes = XArraySize * YArraySize * ZArraySize ;

    SomeMsg *m = new SomeMsg ;
    controlboc = newgroup ControlBoc(m) ;

    NumCubesDone = 0 ;
        }

  void main::InitialSync(SomeMsg *m)
  {
          NumCubesDone++ ;
          if ( NumCubesDone < CNumPes() ) {
                  delete m ;
                  return ;
          }
          NumCubesDone = 0 ;

    cubearray = CreateArray(_CK_chare_Cube, _CK_ep_Cube_Cube, m,
                                    BrunoCapelloMapFn,
          XArraySize,YArraySize,ZArraySize);

  }
```

```
chare class main {

        int NumCubesDone ;

    public:
        main() ;

    entry:
        void InitialSync(SomeMsg *m) ;
} ;
```

```
      implicit real*8 (a-h,o-z)
c
      parameter ( isiz1 = 64, isiz2 = 64, isiz3 = 64 )
      parameter ( xsize = 36, ysize = 36, zsize = 36 )
      parameter ( eachx = 32, eachy = 32, eachz = 32 )
      parameter ( c1 = 1.40d+00, c2 = 0.40d+00,
     $            c3 = 1.00d-01, c4 = 1.00d+00,
     $            c5 = 1.40d+00 )
c     real timer, tstart, tend
c
      integer ex, ey, ez, ex1, ey1, ez1, ex2, ey2, ez2,
     $                 sx, sy, sz,
     $                 sx1, sy1, sz1, sx2, sy2, sz2,
     $                 sx3, ex3, sy3, ey3, sz3, ez3,
     $                 gstx, gendx, gsty, gendy, gstz, gendz,
     $                 sxm2, sym2, szm2, exm2, eym2, ezm2,
     $                 sxm1, sym1, szm1, exm1, eym1, ezm1
c
c***for parallelization
c
      common/parallel/ ex, ey, ez, ex1, ey1, ez1, ex2, ey2, ez2,
     $                 sx, sy, sz,
     $                 sx1, sy1, sz1, sx2, sy2, sz2,
     $                 sx3, ex3, sy3, ey3, sz3, ez3,
     $                 gstx, gendx, gsty, gendy, gstz, gendz,
     $                 sxm2, sym2, szm2, exm2, eym2, ezm2,
     $                 sxm1, sym1, szm1, exm1, eym1, ezm1
c
c***boundaries for use in setiv - setbv :
c
      common/boundaries/ bx1(5,isiz2,isiz3), bxnx(5,isiz2,isiz3),
     $                   by1(5,isiz1,isiz3), byny(5,isiz1,isiz3),
     $                   bz1(5,isiz1,isiz2), bznz(5,isiz1,isiz2)
c
c***grid
c
      common/cgcon/ nx, ny, nz,
     $              ii1, ii2, ji1, ji2, ki1, ki2, idum1,
     $              dxi, deta, dzeta,
     $              tx1, tx2, tx3,
     $              ty1, ty2, ty3,
     $              tz1, tz2, tz3
c
c***dissipation
c
      common/disp/ dx1,dx2,dx3,dx4,dx5,
     $             dy1,dy2,dy3,dy4,dy5,
     $             dz1,dz2,dz3,dz4,dz5,
     $             dssp
c
c***field variables and residuals
c
c     common/cvar/ u(5,isiz1,isiz2,isiz3),
c    $             rsd(5,isiz1,isiz2,isiz3),
c    $             frct(5,isiz1,isiz2,isiz3)
c
```



```
c***output control parameters
c
      common/cprcon/ ipr, iout, inorm
c
c***newton-raphson iteration control parameters
c
      common/ctscon/ itmax, invert,
     $               dt, omega, tolrsd(5),
     $               rsdnm(5), errnm(5), frc, ttotal,
     $               frc1, frc2, frc3
c
c***global jacobian matrix
c
c     common/cjac/ a(isiz1,isiz2,isiz3),
c    $             b(isiz1,isiz2,isiz3),
c    $             c(isiz1,isiz2,isiz3),
c    $             d(isiz1,isiz2,isiz3),
c    $             e(isiz1,isiz2,isiz3)
c
c***coefficients of the exact solution
c
      common/cexact/ ce(5,13)
c
```

```
      subroutine initsp(ipe)
c
c***driver for the performance evaluation of the solver for
c   five coupled, nonlinear partial differential equations.
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
c***open file for input data
c
      open (unit=5,file='appsp.input',status='old',
     *      access='sequential',form='formatted')
      rewind 5
c
c***read the unit number for output data
c
      read (5,*)
      read (5,*)
      read (5,*) iout
c
c***flag that controls printing of the progress of iterations
c
      read (5,*)
      read (5,*)
      read (5,*) ipr, inorm
c
c***set the maximum number of pseudo-time steps to be taken
c
      read (5,*)
      read (5,*)
      read (5,*) itmax
c
c***set the magnitude of the time step
c
      read (5,*)
      read (5,*)
      read (5,*) dt
c
c***set the value of over-relaxation factor for SSOR iterations
c
      read (5,*)
      read (5,*)
      read (5,*) omega
c
c***set the steady-state residual tolerance levels
c
      read (5,*)
      read (5,*)
      read (5,*) tolrsd(1),tolrsd(2),
     $           tolrsd(3),tolrsd(4),tolrsd(5)
c
c***read problem specification parameters
c
```



```
c
c***specify the number of grid points in xi, eta and zeta directions
C
      read (5,*)
      read (5,*)
      read (5,*) nx, ny, nz
c
c***open the file for output data
c
      if ( iout .eq. 7 ) then
c
         open (unit=7,file='output.data',status='unknown',
     $         access='sequential',form='formatted')
         rewind 7
c
      end if
c
      if ( ( nx .lt. 5 ) .or.
     $     ( ny .lt. 5 ) .or.
     $     ( nz .lt. 5 ) ) then
c
         write (*,2001)
 2001    format (5x,'PROBLEM SIZE IS TOO SMALL - ',
     $        /5x,'SET EACH OF NX, NY AND NZ AT LEAST EQUAL TO 5')
         stop
c
      end if
c
      if ( ( nx .gt. isiz1 ) .or.
     $     ( ny .gt. isiz2 ) .or.
     $     ( nz .gt. isiz3 ) ) then
c
         write (*,2002)
 2002    format (5x,'PROBLEM SIZE IS TOO LARGE - ',
     $        /5x,'NX, NY AND NZ SHOULD BE LESS THAN OR EQUAL TO ',
     $        /5x,'ISIZ1, ISIZ2 AND ISIZ3 RESPECTIVELY')
c
      end if
c
      dxi = 1.0d+00 / ( nx - 1 )
      deta = 1.0d+00 / ( ny - 1 )
      dzeta = 1.0d+00 / ( nz - 1 )
c
      tx1 = 1.0d+00 / ( dxi * dxi )
      tx2 = 1.0d+00 / ( 2.0d+00 * dxi )
      tx3 = 1.0d+00 / dxi
c
      ty1 = 1.0d+00 / ( deta * deta )
      ty2 = 1.0d+00 / ( 2.0d+00 * deta )
      ty3 = 1.0d+00 / deta
c
      tz1 = 1.0d+00 / ( dzeta * dzeta )
      tz2 = 1.0d+00 / ( 2.0d+00 * dzeta )
      tz3 = 1.0d+00 / dzeta
c
      ii1 = 2
```

```
      ii2 = nx - 1
      ji1 = 2
      ji2 = ny - 2
      ki1 = 3
      ki2 = nz - 1
c
      frc1 = 0.0d+00
      frc2 = 0.0d+00
      frc3 = 0.0d+00
c
c
c***diffusion coefficients
c
      dx1 = 0.75d+00
      dx2 = dx1
      dx3 = dx1
      dx4 = dx1
      dx5 = dx1
c
      dy1 = 0.75d+00
      dy2 = dy1
      dy3 = dy1
      dy4 = dy1
      dy5 = dy1
c
      dz1 = 1.00d+00
      dz2 = dz1
      dz3 = dz1
      dz4 = dz1
      dz5 = dz1
c
c***fourth difference dissipation
c
      dssp = ( max (dx1, dy1, dz1 ) ) / 4.0d+00
c
c***coefficients of the exact solution to the first pde
c
      ce(1,1) = 2.0d+00
      ce(1,2) = 0.0d+00
      ce(1,3) = 0.0d+00
      ce(1,4) = 4.0d+00
      ce(1,5) = 5.0d+00
      ce(1,6) = 3.0d+00
      ce(1,7) = 5.0d-01
      ce(1,8) = 2.0d-02
      ce(1,9) = 1.0d-02
      ce(1,10) = 3.0d-02
      ce(1,11) = 5.0d-01
      ce(1,12) = 4.0d-01
      ce(1,13) = 3.0d-01
c
c***coefficients of the exact solution to the second pde
c
      ce(2,1) = 1.0d+00
      ce(2,2) = 0.0d+00
      ce(2,3) = 0.0d+00
```



```
      ce(2,4) = 0.0d+00
      ce(2,5) = 1.0d+00
      ce(2,6) = 2.0d+00
      ce(2,7) = 3.0d+00
      ce(2,8) = 1.0d-02
      ce(2,9) = 3.0d-02
      ce(2,10) = 2.0d-02
      ce(2,11) = 4.0d-01
      ce(2,12) = 3.0d-01
      ce(2,13) = 5.0d-01
c
c***coefficients of the exact solution to the third pde
c
      ce(3,1) = 2.0d+00
      ce(3,2) = 2.0d+00
      ce(3,3) = 0.0d+00
      ce(3,4) = 0.0d+00
      ce(3,5) = 0.0d+00
      ce(3,6) = 2.0d+00
      ce(3,7) = 3.0d+00
      ce(3,8) = 4.0d-02
      ce(3,9) = 3.0d-02
      ce(3,10) = 5.0d-02
      ce(3,11) = 3.0d-01
      ce(3,12) = 5.0d-01
      ce(3,13) = 4.0d-01
c
c***coefficients of the exact solution to the fourth pde
c
      ce(4,1) = 2.0d+00
      ce(4,2) = 2.0d+00
      ce(4,3) = 0.0d+00
      ce(4,4) = 0.0d+00
      ce(4,5) = 0.0d+00
      ce(4,6) = 2.0d+00
      ce(4,7) = 3.0d+00
      ce(4,8) = 3.0d-02
      ce(4,9) = 5.0d-02
      ce(4,10) = 4.0d-02
      ce(4,11) = 2.0d-01
      ce(4,12) = 1.0d-01
      ce(4,13) = 3.0d-01
c
c***coefficients of the exact solution to the fifth pde
c
      ce(5,1) = 5.0d+00
      ce(5,2) = 4.0d+00
      ce(5,3) = 3.0d+00
      ce(5,4) = 2.0d+00
      ce(5,5) = 1.0d-01
      ce(5,6) = 4.0d-01
      ce(5,7) = 3.0d-01
      ce(5,8) = 5.0d-02
      ce(5,9) = 4.0d-02
      ce(5,10) = 3.0d-02
      ce(5,11) = 1.0d-01
```

```
      ce(5,12) = 3.0d-01
      ce(5,13) = 2.0d-01
c
      return
      end
c
c
c****************** this control code is now in C++ ***************
c
c***set the boundary values for dependent variables
c
c     call setbv
c
c***set the initial values for dependent variables
c
c     call setiv
c
c***compute the forcing term based on prescribed exact solution
c
c     call erhs
c
c***perform scalar approximate factorization iterations
c
c     call adi
c
c***compute the solution error
c
c     call error
c
c***compute the surface integral
c
c     call pintgr
c
c***verification test
c
c     call verify ( rsdnm, errnm, frc )
c
c***print the CPU time
c
c     write (iout,1001) ttotal
c1001 format (//5x,'Total CPU time = ',1pe12.4,' Sec. ')
c
c     return
c     end
c
c*****************************************************************
c
c
c*************************************************
c
      subroutine getns ( inx, iny, inz, iitmax )
c
c***return isiz parameters
c
c Author: Sanjeev Krishnan
c
```



```
      include 'appsp.incl'
c
      integer inx, iny, inz
c
      inx = nx
      iny = ny
      inz = nz
c
      iitmax = itmax
c
      return
      end
c
c
c
      subroutine setglobals ( iex, iey, iez, igstx, igendx, igsty,
     $                                        igendy, igstz, igendz)
c
c***set global variables for parallelization
c
c Author: Sanjeev Krishnan
c
      include 'appsp.incl'
c
      integer iex, iey, iez, igstx, igendx, igsty,igendy, igstz,
     $        igendz
c
c NOTE : sx1, ex1, sx2, ex2 etc
c change from cube to cube. So they have to be passed as parameters
c or set in setglobals.
c
c


      ex = iex + 2
      ey = iey + 2
      ez = iez + 2
      sx = 3
      sy = 3
      sz = 3
c
c     xsize = iex + 4
c     ysize = iey + 4
c     zsize = iez + 4
c
      gstx = igstx
      gsty = igsty
      gstz = igstz
      gendx = igendx
      gendy = igendy
      gendz = igendz
c
      sx1 = sx
      sy1 = sy
      sz1 = sz
      if ( gstx .eq. 1 ) sx1 = sx + 1
      if ( gsty .eq. 1 ) sy1 = sy + 1
```

```
      if ( gstz .eq. 1 ) sz1 = sz + 1
c
      ex1 = ex
      ey1 = ey
      ez1 = ez
      if ( gendx .eq. nx ) ex1 = ex - 1
      if ( gendy .eq. ny ) ey1 = ey - 1
      if ( gendz .eq. nz ) ez1 = ez - 1
c
      sx2 = sx
      sy2 = sy
      sz2 = sz
      if ( gstx .eq. 1 ) sx2 = sx + 2
      if ( gsty .eq. 1 ) sy2 = sy + 2
      if ( gstz .eq. 1 ) sz2 = sz + 2
c
      ex2 = ex
      ey2 = ey
      ez2 = ez
      if ( gendx .eq. nx ) ex2 = ex - 2
      if ( gendy .eq. ny ) ey2 = ey - 2
      if ( gendz .eq. nz ) ez2 = ez - 2
c
      sx3 = sx
      sy3 = sy
      sz3 = sz
      if ( gstx .eq. 1 ) sx3 = sx + 3
      if ( gsty .eq. 1 ) sy3 = sy + 3
      if ( gstz .eq. 1 ) sz3 = sz + 3
c
      ex3 = ex
      ey3 = ey
      ez3 = ez
      if ( gendx .eq. nx ) ex3 = ex - 3
      if ( gendy .eq. ny ) ey3 = ey - 3
      if ( gendz .eq. nz ) ez3 = ez - 3
c
c some more bounds : Sanjeev
      sxm1 = sx - 1
      sym1 = sy - 1
      szm1 = sz - 1
      if ( gstx .eq. 1 ) sxm1 = sx
      if ( gsty .eq. 1 ) sym1 = sy
      if ( gstz .eq. 1 ) szm1 = sz
      exm1 = ex + 1
      eym1 = ey + 1
      ezm1 = ez + 1
      if ( gendx .eq. nx ) exm1 = ex
      if ( gendy .eq. ny ) eym1 = ey
      if ( gendz .eq. nz ) ezm1 = ez
c
      sxm2 = sx - 2
      sym2 = sy - 2
      szm2 = sz - 2
      if ( gstx .eq. 1 ) sxm2 = sx
      if ( gsty .eq. 1 ) sym2 = sy
```



```
      if ( gstz .eq. 1 ) szm2 = sz
      exm2 = ex + 2
      eym2 = ey + 2
      ezm2 = ez + 2
      if ( gendx .eq. nx ) exm2 = ex
      if ( gendy .eq. ny ) eym2 = ey
      if ( gendz .eq. nz ) ezm2 = ez
c
      return
      end
c
c
c
c
      function iglob (i)
c
      include 'appsp.incl'
      integer i,iglob
c
      itemp = gstx + i - 3
      iglob = itemp
c
      return
      end
c
c
      function jglob (j)
c
      include 'appsp.incl'
      integer j,jglob
c
      jtemp = gsty + j - 3
      jglob = jtemp
c
      return
      end
c
c
      function kglob (k)
c
      include 'appsp.incl'
      integer k,kglob
c
      ktemp = gstz + k - 3
      kglob = ktemp
c
      return
      end
c
c
c
c
      subroutine setb(b, data, type)
c
      include 'appsp.incl'
c
```

```
      real *8 b(5,xsize,ysize,zsize)
      real *8 data(*)
      integer type
      integer count
c
      count = 1
c
      if ( type .eq. 2 ) then
         do i = sx-2,sx-1
            do j = sy, ey
               do k = sz, ez
                  do m = 1,5
                     b(m,i,j,k) = data(count)
                     count = count + 1
                  end do
               end do
            end do
         end do
      else if ( type .eq. 1) then
         do i = ex+1, ex+2
            do j = sy, ey
               do k = sz, ez
                  do m = 1,5
                     b(m,i,j,k) = data(count)
                     count = count + 1
                  end do
               end do
            end do
         end do
      else if ( type .eq. 4 ) then
         do i = sx, ex
            do j = sy-2,sy-1
               do k = sz, ez
                  do m = 1,5
                     b(m,i,j,k) = data(count)
                     count = count + 1
                  end do
               end do
            end do
         end do
      else if ( type .eq. 3 ) then
         do i = sx, ex
            do j = ey+1, ey+2
               do k = sz, ez
                  do m = 1,5
                     b(m,i,j,k) = data(count)
                     count = count + 1
                  end do
               end do
            end do
         end do
      else if ( type .eq. 6 ) then
         do i = sx, ex
            do j = sy, ey
               do k = sz-2,sz-1
                  do m = 1,5
```



```
                     b(m,i,j,k) = data(count)
                     count = count + 1
                  end do
               end do
            end do
         end do
      else if ( type .eq. 5 ) then
         do i = sx, ex
            do j = sy, ey
               do k = ez+1, ez+2
                  do m = 1,5
                     b(m,i,j,k) = data(count)
                     count = count + 1
                  end do
               end do
            end do
         end do
      endif
c
      return
      end
c
c
c
c
      subroutine setcd(cd, data, type)
c
      include 'appsp.incl'
c
      real *8 cd(xsize,ysize,zsize,3)
      real *8 data(*)
      integer type
      integer count
c
      count = 1
c
      if ( type .eq. 2 ) then
         do i = sx-2,sx-1
            do j = sy, ey
               do k = sz, ez
                  do m = 1,3
                     cd(i,j,k,m) = data(count)
                     count = count + 1
                  end do
               end do
            end do
         end do
      else if ( type .eq. 1) then
         do i = ex+1, ex+2
            do j = sy, ey
               do k = sz, ez
                  do m = 1,3
                     cd(i,j,k,m) = data(count)
                     count = count + 1
                  end do
               end do
```

```
            end do
         end do
      else if ( type .eq. 4 ) then
         do i = sx, ex
            do j = sy-2,sy-1
               do k = sz, ez
                  do m = 1,3
                     cd(i,j,k,m) = data(count)
                     count = count + 1
                  end do
               end do
            end do
         end do
      else if ( type .eq. 3 ) then
         do i = sx, ex
            do j = ey+1, ey+2
               do k = sz, ez
                  do m = 1,3
                     cd(i,j,k,m) = data(count)
                     count = count + 1
                  end do
               end do
            end do
         end do
      else if ( type .eq. 6 ) then
         do i = sx, ex
            do j = sy, ey
               do k = sz-2,sz-1
                  do m = 1,3
                     cd(i,j,k,m) = data(count)
                     count = count + 1
                  end do
               end do
            end do
         end do
      else if ( type .eq. 5 ) then
         do i = sx, ex
            do j = sy, ey
               do k = ez+1, ez+2
                  do m = 1,3
                     cd(i,j,k,m) = data(count)
                     count = count + 1
                  end do
               end do
            end do
         end do
      endif
c
      return
      end
c
c
c
c
      subroutine getb(b, data, type, count)
c
```



```
      include 'appsp.incl'
c
      real *8 b(5,xsize,ysize,zsize)
      real *8 data(*)
      integer type
      integer count

      if ( type .eq. 1 ) then
         do i = sx, sx+1
            do j = sy, ey
               do k = sz, ez
                  do m = 1,5
                     data(count) = b(m,i,j,k)
                     count = count + 1
                  end do
               end do
            end do
         end do
      else if ( type .eq. 2 ) then
         do i = ex-1, ex
            do j = sy, ey
               do k = sz, ez
                  do m = 1,5
                     data(count) = b(m,i,j,k)
                     count = count + 1
                  end do
               end do
            end do
         end do
      else if ( type .eq. 3 ) then
         do i = sx, ex
            do j = sy, sy+1
               do k = sz, ez
                  do m = 1,5
                     data(count) = b(m,i,j,k)
                     count = count + 1
                  end do
               end do
            end do
         end do
      else if ( type .eq. 4 ) then
         do i = sx, ex
            do j = ey-1, ey
               do k = sz, ez
                  do m = 1,5
                     data(count) = b(m,i,j,k)
                     count = count + 1
                  end do
               end do
            end do
         end do
      else if ( type .eq. 5 ) then
         do i = sx, ex
            do j = sy, ey
               do k = sz, sz+1
                  do m = 1,5
```

```
                  data(count) = b(m,i,j,k)
                  count = count + 1
               end do
            end do
         end do
      end do
   else if ( type .eq. 6 ) then
      do i = sx, ex
         do j = sy, ey
            do k = ez-1, ez
               do m = 1,5
                  data(count) = b(m,i,j,k)
                  count = count + 1
               end do
            end do
         end do
      end do
   endif
c
   return
   end
c
c
c
c
c
   subroutine getcd(cd, data, type, count)
c
   include 'appsp.incl'
c
   real *8 cd(xsize,ysize,zsize,3)
   real *8 data(*)
   integer type
   integer count

   if ( type .eq. 1 ) then
      do i = sx, sx+1
         do j = sy, ey
            do k = sz, ez
               do m = 1,3
                  data(count) = cd(i,j,k,m)
                  count = count + 1
               end do
            end do
         end do
      end do
   else if ( type .eq. 2 ) then
      do i = ex-1, ex
         do j = sy, ey
            do k = sz, ez
               do m = 1,3
                  data(count) = cd(i,j,k,m)
                  count = count + 1
               end do
            end do
         end do
```



```
      end do
   else if ( type .eq. 3 ) then
      do i = sx, ex
         do j = sy, sy+1
            do k = sz, ez
               do m = 1,3
                  data(count) = cd(i,j,k,m)
                  count = count + 1
               end do
            end do
         end do
      end do
   else if ( type .eq. 4 ) then
      do i = sx, ex
         do j = ey-1, ey
            do k = sz, ez
               do m = 1,3
                  data(count) = cd(i,j,k,m)
                  count = count + 1
               end do
            end do
         end do
      end do
   else if ( type .eq. 5 ) then
      do i = sx, ex
         do j = sy, ey
            do k = sz, sz+1
               do m = 1,3
                  data(count) = cd(i,j,k,m)
                  count = count + 1
               end do
            end do
         end do
      end do
   else if ( type .eq. 6 ) then
      do i = sx, ex
         do j = sy, ey
            do k = ez-1, ez
               do m = 1,3
                  data(count) = cd(i,j,k,m)
                  count = count + 1
               end do
            end do
         end do
      end do
   endif
c
   return
   end
c
c
c
c
c**************************************************
c
```

```
c
      subroutine setbv(u,igstx, igendx, igsty, igendy, igstz, igendz)
c
c***set the boundary values of dependent variables
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize)
      integer igstx, igendx, igsty, igendy, igstz, igendz
c
      gstx = igstx
      gendx = igendx
      gsty = igsty
      gendy = igendy
      gstz = igstz
      gendz = igendz
c
c***set the dependent variable values along the top and bottom faces
c
      do j = sy, ey
         do i = sx, ex
c
            call exact ( iglob(i), jglob(j), 1, bz1( 1, i, j ) )
            call exact ( iglob(i), jglob(j), nz, bznz( 1, i, j ) )
c
         end do
      end do
c
      if ( gstz .eq. 1 ) then
      do j = sy, ey
         do i = sx, ex
c
            call exact ( iglob(i), jglob(j), 1, u( 1, i, j, sz ) )
c
         end do
      end do
      endif
      if ( gendz .eq. nz ) then
      do j = sy, ey
         do i = sx, ex
c
            call exact ( iglob(i), jglob(j), nz, u( 1, i, j, ez ) )
c
         end do
      end do
      endif
c
c***set the dependent variable values along north and south faces
c
      do k = sz, ez
         do i = sx, ex
c
```



```
            call exact ( iglob(i), 1, kglob(k), by1( 1, i, k ) )
            call exact ( iglob(i), ny, kglob(k), byny( 1, i, k ) )
c
         end do
      end do
c
      if ( gsty .eq. 1 ) then
      do k = sz, ez
         do i = sx, ex
c
            call exact ( iglob(i), 1, kglob(k), u( 1, i, sy, k ) )
c
         end do
      end do
      endif
      if ( gendy .eq. ny ) then
      do k = sz, ez
         do i = sx, ex
c
            call exact ( iglob(i), ny, kglob(k), u( 1, i, ey, k ) )
c
         end do
      end do
      endif
c
c***set the dependent variable values along east and west faces
c
      do k = sz, ez
         do j = sy, ey
c
            call exact ( 1, jglob(j), kglob(k), bx1( 1, j, k ) )
            call exact ( nx, jglob(j), kglob(k), bxnx( 1, j, k ) )
c
         end do
      end do
c
      if ( gstx .eq. 1 ) then
      do k = sz, ez
         do j = sy, ey
c
            call exact ( 1, jglob(j), kglob(k), u( 1, sx, j, k ) )
c
         end do
      end do
      endif
      if ( gendx .eq. nx ) then
      do k = sz, ez
         do j = sy, ey
c
            call exact ( nx, jglob(j), kglob(k), u( 1, ex, j, k ) )
c
         end do
      end do
      endif
c
      return
```

```
      end
c
c
c
c
      subroutine setiv(u,igstx,igsty,igstz)
c
c***set the initial values of independent variables based on tri-linear
c   interpolation of boundary values in the computational space.
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize)
      integer igstx, igsty, igstz
c
      gstx = igstx
      gsty = igsty
      gstz = igstz
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      do k = sz1, ez1
c
         kg = kglob(k)
         zeta = ( dfloat(kg-1) ) / ( nz - 1 )
c
         do j = sy1, ey1
c
            jg = jglob(j)
            eta = ( dfloat(jg-1) ) / ( ny - 1 )
c
            do i = sx1, ex1
c
               ig = iglob(i)
               xi = ( dfloat(ig-1) ) / ( nx - 1 )
c
               do m = 1, 5
c
                  pxi =   ( 1.0d+00 - xi ) * bx1(m,j,k)
     $                          + xi   * bxnx(m,j,k)
c
                  peta =  ( 1.0d+00 - eta ) * by1(m,i,k)
     $                          + eta   * byny(m,i,k)
c
                  pzeta = ( 1.0d+00 - zeta ) * bz1(m,i,j)
     $                          + zeta   * bznz(m,i,j)
c
                  u( m, i, j, k ) = pxi + peta + pzeta
     $                 - pxi * peta - peta * pzeta - pzeta * pxi
     $                 + pxi * peta * pzeta
c
               end do
c
            end do
```



```
c
         end do
c
      end do
c
      return
      end
c
c
c
c
c**************** THIS IS OLD CODE FOR BEGINNING OF ADI *********
c
c     subroutine adi
c
c***to perform pseudo-time stepping iterations
c   for five coupled, nonlinear pde's.
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
c     include 'appsp.incl'
c
c     dimension idmax(5), jdmax(5), kdmax(5),
c    $          imax(5), jmax(5), kmax(5),
c    $          delunm(5)
c     parameter ( one = 1.0d+00 )
c
c     lnorm = 2
c
c***compute the steady-state residuals
c
c     call rhs
c
c***compute the norms of the residuals
c
c     if ( lnorm .eq. 1 ) then
c
c        call maxnorm ( isiz1, isiz2, isiz3,
c    $                  nx, ny, nz,
c    $                  imax, jmax, kmax,
c    $                  rsd, rsdnm )
c
c        if ( ipr .eq. 1 )  then
c
c           write (iout,*) '          Initial Residual norms'
c           write (iout,*)
c           write (iout,1003) ( rsdnm(m),
c    $           imax(m), jmax(m), kmax(m), m = 1, 5 )
c
c        end if
c
c     else if ( lnorm .eq. 2 ) then
c
c        call l2norm ( rsd, rsdnm )
```

```
c
c        if ( ipr .eq. 1 )  then
c
c           write (iout,*) '          Initial Residual norms'
c           write (iout,*)
c           write (iout,1007) ( rsdnm(m), m = 1, 5 )
c
c        end if
c
c     end if
c
c***begin pseudo-time stepping iterations
c
c     tstart = timer( )
c
c     do istep = 1, itmax
c
c
c********************** END OF OLD CODE AT BEGINNING OF ADI *******
c
c
c
c
      subroutine adix(u,rsd,frct,a,b,c,d,e,igstx, igendx, igsty,
     $                                     igendy, igstz, igendz)
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize),
     $        frct(5,xsize,ysize,zsize), a(xsize,ysize,zsize,3),
     $        b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3),
     $        d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
c
c     dimension idmax(5), jdmax(5), kdmax(5),
c    $          imax(5), jmax(5), kmax(5),
c    $          delunm(5)
      parameter ( one = 1.0d+00 )
c
      integer igstx, igendx, igsty, igendy, igstz, igendz
c
      gstx = igstx
      gendx = igendx
      gsty = igsty
      gendy = igendy
      gstz = igstz
      gendz = igendz
c
c**** This code was formerly at the beginning of the ADI loop ********
c
c        if ( ( mod ( istep, inorm ) .eq. 0 ) .and.
c    $        ( ipr .eq. 1 ) ) then
c
c
c           write ( iout, 1001 ) istep
c
c        end if
```



```
c
         do k = sz, ez
            do j = sy, ey
               do i = sx, ex
                  do m = 1, 5
c
                     rsd(m,i,j,k) = dt * rsd(m,i,j,k)
c
                  end do
               end do
            end do
         end do
c
c***perform 3-factor, scalar ADI iterations
c
c***perform the block diagonal inversion
c
         call txinvr(u,rsd)
c
c***perform the xsi-direction sweep
c
         call jacx ( 3, u, a(1,1,1,1), b(1,1,1,1), c(1,1,1,1),
     $                  d(1,1,1,1), e(1,1,1,1) )
c
         call spentax3 ( a(1,1,1,1), b(1,1,1,1), c(1,1,1,1),
     $                  d(1,1,1,1), e(1,1,1,1),
     $                  rsd )
c
         call jacx ( 4, u, a(1,1,1,2), b(1,1,1,2), c(1,1,1,2),
     $                d(1,1,1,2), e(1,1,1,2) )
c
         call spentax ( 4, a(1,1,1,2), b(1,1,1,2), c(1,1,1,2),
     $                  d(1,1,1,2), e(1,1,1,2),
     $                  rsd )
c
         call jacx ( 5, u, a(1,1,1,3), b(1,1,1,3), c(1,1,1,3),
     $                d(1,1,1,3), e(1,1,1,3) )
c
         call spentax ( 5, a(1,1,1,3), b(1,1,1,3), c(1,1,1,3),
     $                  d(1,1,1,3), e(1,1,1,3),
     $                  rsd )
c
c
      return
c
 1001 format (1x/5x,'pseudo-time Scalar ADI iteration no.=',i4/)
c
      end
c
c
c
c
      subroutine badix(u,rsd,frct,a,b,c,d,e,igstx, igendx, igsty,
     $                                     igendy, igstz, igendz)
c
      include 'appsp.incl'
```

```
c
      real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize),
     $        frct(5,xsize,ysize,zsize,3), a(xsize,ysize,zsize,3),
     $        b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3),
     $        d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
c
c     dimension idmax(5), jdmax(5), kdmax(5),
c    $          imax(5), jmax(5), kmax(5),
c    $          delunm(5)
      parameter ( one = 1.0d+00 )
c
      integer igstx, igendx, igsty, igendy, igstz, igendz
c
      gstx = igstx
      gendx = igendx
      gsty = igsty
      gendy = igendy
      gstz = igstz
      gendz = igendz
c
c Backward substitution
c
         call bspentax3 ( a(1,1,1,1), b(1,1,1,1), c(1,1,1,1),
     $                    d(1,1,1,1), e(1,1,1,1),
     $                   rsd )
c
         call bspentax ( 4, a(1,1,1,2), b(1,1,1,2), c(1,1,1,2),
     $                   d(1,1,1,2), e(1,1,1,2),
     $                  rsd )
c
         call bspentax ( 5, a(1,1,1,3), b(1,1,1,3), c(1,1,1,3),
     $                   d(1,1,1,3), e(1,1,1,3),
     $                  rsd )
c
      return
      end
c
c
c
c
      subroutine adiy(u,rsd,frct,a,b,c,d,e,igstx, igendx, igsty,
     $                                     igendy, igstz, igendz)
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize),
     $        frct(5,xsize,ysize,zsize), a(xsize,ysize,zsize,3),
     $        b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3),
     $        d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
c
c     dimension idmax(5), jdmax(5), kdmax(5),
c    $          imax(5), jmax(5), kmax(5),
c    $          delunm(5)
      parameter ( one = 1.0d+00 )
c
      integer igstx, igendx, igsty, igendy, igstz, igendz
```



```
c
      gstx = igstx
      gendx = igendx
      gsty = igsty
      gendy = igendy
      gstz = igstz
      gendz = igendz
c
c
c**** This code was formerly in the middle of the ADI loop ********
c
c***perform the block diagonal inversion
c
         call ninvr(rsd)
c
c***perform the eta-direction sweep
         call jacy ( 3, u, a(1,1,1,1), b(1,1,1,1), c(1,1,1,1),
     $                  d(1,1,1,1), e(1,1,1,1) )
c
         call spentay3 ( a(1,1,1,1), b(1,1,1,1), c(1,1,1,1),
     $                   d(1,1,1,1), e(1,1,1,1),
     $                   rsd )
c
         call jacy ( 4, u, a(1,1,1,2), b(1,1,1,2), c(1,1,1,2),
     $                d(1,1,1,2), e(1,1,1,2) )
c
         call spentay ( 4, a(1,1,1,2), b(1,1,1,2), c(1,1,1,2),
     $                   d(1,1,1,2), e(1,1,1,2),
     $                   rsd )
c
         call jacy ( 5, u, a(1,1,1,3), b(1,1,1,3), c(1,1,1,3),
     $                d(1,1,1,3), e(1,1,1,3) )
c
         call spentay ( 5, a(1,1,1,3), b(1,1,1,3), c(1,1,1,3),
     $                   d(1,1,1,3), e(1,1,1,3),
     $                   rsd )
c
c
      return
      end
c
c
c
c
      subroutine badiy(u,rsd,frct,a,b,c,d,e,igstx, igendx, igsty,
     $                                     igendy, igstz, igendz)
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize),
     $        frct(5,xsize,ysize,zsize), a(xsize,ysize,zsize,3),
     $        b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3),
     $        d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
c
c     dimension idmax(5), jdmax(5), kdmax(5),
c    $          imax(5), jmax(5), kmax(5),
```

```
c     $          delunm(5)
      parameter ( one = 1.0d+00 )
c
      integer igstx, igendx, igsty, igendy, igstz, igendz
c
      gstx = igstx
      gendx = igendx
      gsty = igsty
      gendy = igendy
      gstz = igstz
      gendz = igendz
c
c
c***backward sweep
c
          call bspentay3 ( a(1,1,1,1), b(1,1,1,1), c(1,1,1,1),
     $                     d(1,1,1,1), e(1,1,1,1),
     $                    rsd )
c
          call bspentay ( 4, a(1,1,1,2), b(1,1,1,2), c(1,1,1,2),
     $                   d(1,1,1,2), e(1,1,1,2),
     $                   rsd )
c
          call bspentay ( 5, a(1,1,1,3), b(1,1,1,3), c(1,1,1,3),
     $                   d(1,1,1,3), e(1,1,1,3),
     $                   rsd )
c
c
      return
      end
c
c
c
c
c
c
      subroutine adiz(u,rsd,frct,a,b,c,d,e,igstx, igendx, igsty,
     $                                     igendy, igstz, igendz)
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize),
     $        frct(5,xsize,ysize,zsize), a(xsize,ysize,zsize,3),
     $        b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3),
     $        d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
c
c     dimension idmax(5), jdmax(5), kdmax(5),
c     $          imax(5), jmax(5), kmax(5),
c     $          delunm(5)
      parameter ( one = 1.0d+00 )
c
      integer igstx, igendx, igsty, igendy, igstz, igendz
c
      gstx = igstx
      gendx = igendx
      gsty = igsty
```



```
      gendy = igendy
      gstz = igstz
      gendz = igendz
c
c**** This code was formerly near the end of the ADI loop ********
c
c
c***perform the block diagonal inversion
c
          call pinvr(rsd)
c
c***perform the zeta-direction sweep
c
          call jacz ( 3, u, a(1,1,1,1), b(1,1,1,1), c(1,1,1,1),
     $                 d(1,1,1,1), e(1,1,1,1) )
c
          call spentaz3 ( a(1,1,1,1), b(1,1,1,1), c(1,1,1,1),
     $                   d(1,1,1,1), e(1,1,1,1),
     $                   rsd )
c
          call jacz ( 4, u, a(1,1,1,2), b(1,1,1,2), c(1,1,1,2),
     $               d(1,1,1,2), e(1,1,1,2) )
c
          call spentaz ( 4, a(1,1,1,2), b(1,1,1,2), c(1,1,1,2),
     $                   d(1,1,1,2), e(1,1,1,2),
     $                   rsd )
c
          call jacz ( 5, u, a(1,1,1,3), b(1,1,1,3), c(1,1,1,3),
     $               d(1,1,1,3), e(1,1,1,3) )
c
          call spentaz ( 5, a(1,1,1,3), b(1,1,1,3), c(1,1,1,3),
     $                  d(1,1,1,3), e(1,1,1,3),
     $                  rsd )
c
      return
      end
c
c
c
c
c
c
      subroutine badiz(u,rsd,frct,a,b,c,d,e,igstx, igendx, igsty,
     $                                     igendy, igstz, igendz)
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize),
     $        frct(5,xsize,ysize,zsize), a(xsize,ysize,zsize,3),
     $        b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3),
     $        d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
c
c     dimension idmax(5), jdmax(5), kdmax(5),
c     $          imax(5), jmax(5), kmax(5),
c     $          delunm(5)
      parameter ( one = 1.0d+00 )
```

```
c
      integer igstx, igendx, igsty, igendy, igstz, igendz
c
      gstx = igstx
      gendx = igendx
      gsty = igsty
      gendy = igendy
      gstz = igstz
      gendz = igendz
c
c**** This code was formerly near the end of the ADI loop ********
c
c        call prsd(rsd)
c
         call bspentaz3 ( a(1,1,1,1), b(1,1,1,1), c(1,1,1,1),
     $                    d(1,1,1,1), e(1,1,1,1),
     $                   rsd )
c
         call bspentaz ( 4, a(1,1,1,2), b(1,1,1,2), c(1,1,1,2),
     $                  d(1,1,1,2), e(1,1,1,2),
     $                  rsd )
c
         call bspentaz ( 5, a(1,1,1,3), b(1,1,1,3), c(1,1,1,3),
     $                  d(1,1,1,3), e(1,1,1,3),
     $                  rsd )
c
      return
      end
c
c
c
c
c
c
      subroutine mvmul(u,rsd,igstx, igendx, igsty, igendy,
     $                                       igstz, igendz)
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize)
      parameter ( one = 1.0d+00 )
c
      integer igstx, igendx, igsty, igendy, igstz, igendz
c
      gstx = igstx
      gendx = igendx
      gsty = igsty
      gendy = igendy
      gstz = igstz
      gendz = igendz
c
c***perform the block diagonal matrix-vector multiplication
c
         call tzetar(u,rsd)
c
c***update the variables
```



```
c
         do k = sz1, ez1
            do j = sy1, ey1
               do i = sx1, ex1
                  do m = 1, 5
c
                     u( m, i, j, k ) =  u( m, i, j, k )
     $                                  + rsd( m, i, j, k )
c
                  end do
               end do
            end do
         end do
c
c
      return
      end
c
c
c
c
c
c
      subroutine cnorms(rsd,delunm)
c
      include 'appsp.incl'
c
c     implicit real *8 (a-h,o-z)
c
      real *8 rsd(5,xsize,ysize,zsize)
c
      dimension idmax(5), jdmax(5), kdmax(5),
     $          imax(5), jmax(5), kmax(5),
     $          delunm(5)
      parameter ( one = 1.0d+00 )
      real *8 tmp(5)
c
c
c***compute the norms of pseudo-time iteration corrections
c
      lnorm = 2
c
         if ( mod ( istep, inorm ) .eq. 0 ) then
c
            if ( lnorm .eq. 1 ) then
c
               call maxnorm ( isiz1, isiz2, isiz3,
     $                        nx, ny, nz,
     $                        idmax, jdmax, kdmax,
     $                        rsd, delunm )
c
            if ( ipr .eq. 1 ) then
c
               write (iout,1002) ( delunm(m),
     $                idmax(m), jdmax(m), kdmax(m), m = 1, 5 )
c
```

```
               end if
c
            else if ( lnorm .eq. 2 ) then
c
               call l2norm ( rsd, tmp )
c
               if ( ipr .eq. 1 ) then
c
                  write (iout,1006) ( tmp(m), m = 1, 5 )
c
               end if
c
            end if
c
         end if
         do m = 1,5
            delunm(m) = delunm(m) + tmp(m)
c           call psval(m,tmp(m))
         end do


c
c***compute the steady-state residuals
c
c This is called from the C++ code
c        call rhs
c
c
      return
c
 1002 format (1x/1x,'max-norm of Scalar ADI-iteration correction ',
     $ 'for first pde  = ',1pe12.5/,
     $ 59x,'(',i4,',',i4,',',i4,')'/,
     $ 1x,'max-norm of Scalar ADI-iteration correction ',
     $ 'for second pde = ',1pe12.5/,
     $ 59x,'(',i4,',',i4,',',i4,')'/,
     $ 1x,'max-norm of Scalar ADI-iteration correction ',
     $ 'for third pde  = ',1pe12.5/,
     $ 59x,'(',i4,',',i4,',',i4,')'/,
     $ 1x,'max-norm of Scalar ADI-iteration correction ',
     $ 'for fourth pde = ',1pe12.5/,
     $ 59x,'(',i4,',',i4,',',i4,')'/,
     $ 1x,'max-norm of Scalar ADI-iteration correction ',
     $ 'for fifth pde  = ',1pe12.5/,
     $ 59x,'(',i4,',',i4,',',i4,')' )
c
 1006 format (1x/1x,'RMS-norm of scalar adi-iteration correction ',
     $ 'for first pde  = ',1pe12.5/,
     $ 1x,'RMS-norm of scalar adi-iteration correction ',
     $ 'for second pde = ',1pe12.5/,
     $ 1x,'RMS-norm of scalar adi-iteration correction ',
     $ 'for third pde  = ',1pe12.5/,
     $ 1x,'RMS-norm of scalar adi-iteration correction ',
     $ 'for fourth pde = ',1pe12.5/,
     $ 1x,'RMS-norm of scalar adi-iteration correction ',
     $ 'for fifth pde  = ',1pe12.5)
```



```
c
      end
c
c
c
c
c
      subroutine rnorms(rsd,rnms)
c
      include 'appsp.incl'
c
c     implicit real *8 (a-h,o-z)
c
      real *8 rsd(5,xsize,ysize,zsize)
c
      dimension idmax(5), jdmax(5), kdmax(5),
     $          imax(5), jmax(5), kmax(5),
     $          delunm(5), rnms(5)
      parameter ( one = 1.0d+00 )
c
c
c**** This code was formerly near the end of the ADI loop ********
c
c***compute the norms of the residuals
c
      lnorm = 2

c
         if ( ( mod ( istep, inorm ) .eq. 0 ) .or.
     $        ( istep .eq. itmax ) ) then
c
            if ( lnorm .eq. 1 ) then
c
               call maxnorm ( isiz1, isiz2, isiz3,
     $                        nx, ny, nz,
     $                        imax, jmax, kmax,
     $                        rsd, rsdnm )
c
               if ( ipr .eq. 1 )  then
c
                  write (iout,1003) ( rsdnm(m),
     $                        imax(m), jmax(m), kmax(m), m = 1, 5 )
c
               end if
c
            else if ( lnorm .eq. 2 ) then
c
               call l2norm ( rsd, rsdnm )
c
               if ( ipr .eq. 1 )  then
c
                  write (iout,1007) ( rsdnm(m), m = 1, 5 )
c
               end if
c
            end if
```

```
c
        end if
c
        do m = 1,5
           rnms(m) = rnms(m) + rsdnm(m)
        end do
c
c***check the pseudo-time iteration residuals against the tolerance levels
c
c       if ( ( rsdnm(1) .lt. tolrsd(1) ) .and.
c    $       ( rsdnm(2) .lt. tolrsd(2) ) .and.
c    $       ( rsdnm(3) .lt. tolrsd(3) ) .and.
c    $       ( rsdnm(4) .lt. tolrsd(4) ) .and.
c    $       ( rsdnm(5) .lt. tolrsd(5) ) ) then
c
c          write (iout,1004) istep
cSanjeev   return
c
c       end if
c
c
     return
c

 1003 format (1x/1x,'max-norm of steady-state residual for ',
    $ 'first pde  = ',1pe12.5/,
    $ 49x,'(',i4,',',i4,',',i4,')'/,
    $ 1x,'max-norm of steady-state residual for ',
    $ 'second pde = ',1pe12.5/,
    $ 49x,'(',i4,',',i4,',',i4,')'/,
    $ 1x,'max-norm of steady-state residual for ',
    $ 'third pde  = ',1pe12.5/,
    $ 49x,'(',i4,',',i4,',',i4,')'/,
    $ 1x,'max-norm of steady-state residual for ',
    $ 'fourth pde = ',1pe12.5/,
    $ 49x,'(',i4,',',i4,',',i4,')'/,
    $ 1x,'max-norm of steady-state residual for ',
    $ 'fifth pde  = ',1pe12.5/,
    $ 49x,'(',i4,',',i4,',',i4,')' )
c
 1004 format (1x/1x,'convergence was achieved after ',i4,
    $   ' pseudo-time steps' )
c
 1007 format (1x/1x,'RMS-norm of steady-state residual for ',
    $ 'first pde  = ',1pe12.5/,
    $ 1x,'RMS-norm of steady-state residual for ',
    $ 'second pde = ',1pe12.5/,
    $ 1x,'RMS-norm of steady-state residual for ',
    $ 'third pde  = ',1pe12.5/,
    $ 1x,'RMS-norm of steady-state residual for ',
    $ 'fourth pde = ',1pe12.5/,
    $ 1x,'RMS-norm of steady-state residual for ',
    $ 'fifth pde  = ',1pe12.5)
c
     end
c
```



```
c
c
c************* THIS CODE WAS FORMERLY AT THE END OF THE ADI LOOP ******
c
c     end do
c
c     tend = timer( )
c
c     ttotal = tend - tstart
c
c     return
c
c1005 format (/5x,'CPU time = ',1pe12.4,' Sec.')
c
c     end
c
c************* END OF CODE FORMERLY AT THE END OF THE ADI LOOP ********
c
c
c
c
      subroutine maxnorm ( ldmi, ldmj, ldmk,
     $                     nx, ny, nz,
     $                     imax, jmax, kmax, v, vnm )
c
c***compute the max-norm of vector v.
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
c     include 'appsp.incl'
c
      implicit real *8 (a-h,o-z)
      dimension v(5,ldmi,ldmj,*),
     $          vnm(*), imax(*), jmax(*), kmax(*)
c
      do m = 1, 5
c
         vnm(m) = -1.0d+10
c
      end do
c
      do k = sz, ez
         do j = sy, ey
            do i = sx, ex
c
               do m = 1, 5
c
                  t1 = abs ( v( m, i, j, k ) )
c
                  if ( vnm(m) .lt. t1 ) then
                     vnm(m) = t1
                     imax(m) = iglob(i)
                     jmax(m) = jglob(j)
                     kmax(m) = kglob(k)
```

```
                  end if
c
               end do
c
            end do
c
         end do
c
      end do
c
      return
      end
c
c
c
c
c
      subroutine l2norm ( v, sum )
c
c***compute the l2-norm of vector v.
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
c     implicit real *8 (a-h,o-z)
c     dimension v(5,ldx,ldy,*),
      real *8 v(5,xsize,ysize,zsize), sum(5)
c
c
      do m = 1, 5
         sum(m) = 0.0d+00
      end do
c
      do k = sz1, ez1
         do j = sy1, ey1
            do i = sx1, ex1
c
               do m = 1, 5
c
                  sum(m) = sum(m) + v(m,i,j,k) * v(m,i,j,k)
c
               end do
c
            end do
         end do
      end do

c
c     do m = 1, 5
c
c        sum(m) = sqrt ( sum(m) / ( (nx-2) * (ny-2) * (nz-2) ) )
c
c     end do
```



```
c
      return
      end
c
c
c
c
c
      subroutine erhs(frct,igstx, igendx, igsty, igendy, igstz,
     $                igendz)
c
c***compute the right hand side based on exact solution
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 frct(5,xsize,ysize,zsize)
c
      dimension flux(5,zsize), ue(5,zsize)
      integer igstx, igsty, igstz
c
      gstx = igstx
      gendx = igendx
      gsty = igsty
      gendy = igendy
      gstz = igstz
      gendz = igendz
c
      dsspm = dssp
c
      do k = sz, ez
         do j = sy, ey
            do i = sx, ex
               do m = 1, 5
                  frct( m, i, j, k ) = 0.0d+00
               end do
            end do
         end do
      end do
c
c***xi-direction flux differences
c
      do k = sz1, ez1
c
         kg = kglob(k)
         zeta = ( dfloat(kg-1) ) / ( nz - 1 )
c
         do j = sy1, ey1
c
            jg = jglob(j)
            eta = ( dfloat(jg-1) ) / ( ny - 1 )
c
            do i = sxm2, exm2
```

```
c
               ig = iglob(i)
               xi = ( dfloat(ig-1) ) / ( nx - 1 )
c
               do m = 1, 5
c
                  ue(m,i) =  ce(m,1)
     $                 + ce(m,2) * xi
     $                 + ce(m,3) * eta
     $                 + ce(m,4) * zeta
     $                 + ce(m,5) * xi * xi
     $                 + ce(m,6) * eta * eta
     $                 + ce(m,7) * zeta * zeta
     $                 + ce(m,8) * xi * xi * xi
     $                 + ce(m,9) * eta * eta * eta
     $                 + ce(m,10) * zeta * zeta * zeta
     $                 + ce(m,11) * xi * xi * xi * xi
     $                 + ce(m,12) * eta * eta * eta * eta
     $                 + ce(m,13) * zeta * zeta * zeta * zeta
c
               end do
c
               flux(1,i) = ue(2,i)
c
               u21 = ue(2,i) / ue(1,i)
c
               q = 0.50d+00 * (  ue(2,i) * ue(2,i)
     $                         + ue(3,i) * ue(3,i)
     $                         + ue(4,i) * ue(4,i) )
     $                       / ue(1,i)
c
               flux(2,i) = ue(2,i) * u21 + c2 * ( ue(5,i) - q )
c
               flux(3,i) = ue(3,i) * u21
c
               flux(4,i) = ue(4,i) * u21
c
               flux(5,i) = ( c1 * ue(5,i) - c2 * q ) * u21
c
            end do
c
            do i = sx1, ex1
c
               do m = 1, 5
c
                  frct(m,i,j,k) =  frct(m,i,j,k)
     $                          - tx2 * ( flux(m,i+1) - flux(m,i-1) )
c
               end do
c
            end do
c
            do i = sx1, exm1
c
               tmp = 1.0d+00 / ue(1,i)
c
```



```
               u21i = tmp * ue(2,i)
               u31i = tmp * ue(3,i)
               u41i = tmp * ue(4,i)
               u51i = tmp * ue(5,i)
c
               tmp = 1.0d+00 / ue(1,i-1)
c
               u21im1 = tmp * ue(2,i-1)
               u31im1 = tmp * ue(3,i-1)
               u41im1 = tmp * ue(4,i-1)
               u51im1 = tmp * ue(5,i-1)
c
               flux(2,i) = (4.0d+00/3.0d+00) * tx3 * ( u21i - u21im1 )
               flux(3,i) = tx3 * ( u31i - u31im1 )
               flux(4,i) = tx3 * ( u41i - u41im1 )
               flux(5,i) = 0.50d+00 * ( 1.0d+00 - c1*c5 )
     $              * tx3 * ( ( u21i  **2 + u31i  **2 + u41i  **2 )
     $                      - ( u21im1**2 + u31im1**2 + u41im1**2 ) )
     $              + (1.0d+00/6.0d+00)
     $              * tx3 * ( u21i**2 - u21im1**2 )
     $              + c1 * c5 * tx3 * ( u51i - u51im1 )
c
            end do
c
            do i = sx1, ex1
c
               frct(1,i,j,k) = frct(1,i,j,k)
     $              + dx1 * tx1 * (            ue(1,i-1)
     $                             - 2.0d+00 * ue(1,i)
     $                             +           ue(1,i+1) )
c
               frct(2,i,j,k) = frct(2,i,j,k)
     $              + tx3 * c3 * c4 * ( flux(2,i+1) - flux(2,i) )
     $              + dx2 * tx1 * (            ue(2,i-1)
     $                             - 2.0d+00 * ue(2,i)
     $                             +           ue(2,i+1) )
c
               frct(3,i,j,k) = frct(3,i,j,k)
     $              + tx3 * c3 * c4 * ( flux(3,i+1) - flux(3,i) )
     $              + dx3 * tx1 * (            ue(3,i-1)
     $                             - 2.0d+00 * ue(3,i)
     $                             +           ue(3,i+1) )
c
               frct(4,i,j,k) = frct(4,i,j,k)
     $              + tx3 * c3 * c4 * ( flux(4,i+1) - flux(4,i) )
     $              + dx4 * tx1 * (            ue(4,i-1)
     $                             - 2.0d+00 * ue(4,i)
     $                             +           ue(4,i+1) )
c
               frct(5,i,j,k) = frct(5,i,j,k)
     $              + tx3 * c3 * c4 * ( flux(5,i+1) - flux(5,i) )
     $              + dx5 * tx1 * (            ue(5,i-1)
     $                             - 2.0d+00 * ue(5,i)
     $                             +           ue(5,i+1) )
c
            end do
```

```
c
c***Fourth-order dissipation
c
            if ( gstx .eq. 1 ) then
            do m = 1, 5
c
               frct(m,sx1,j,k) = frct(m,sx1,j,k)
     $           - dsspm * ( + 5.0d+00 * ue(m,sx1)
     $                       - 4.0d+00 * ue(m,sx2)
     $                       +           ue(m,sx3) )
c
               frct(m,sx2,j,k) = frct(m,sx2,j,k)
     $           - dsspm * ( - 4.0d+00 * ue(m,sx1)
     $                       + 6.0d+00 * ue(m,sx2)
     $                       - 4.0d+00 * ue(m,sx3)
     $                       +           ue(m,sx+4) )
c
            end do
            endif
c
            do i = sx3, ex3
c
               do m = 1, 5
c
                  frct(m,i,j,k) = frct(m,i,j,k)
     $              - dsspm * (           ue(m,i-2)
     $                          - 4.0d+00 * ue(m,i-1)
     $                          + 6.0d+00 * ue(m,i)
     $                          - 4.0d+00 * ue(m,i+1)
     $                          +           ue(m,i+2) )
c
               end do
c
            end do
c
            if ( gendx .eq. nx ) then
            do m = 1, 5
c
               frct(m,ex2,j,k) = frct(m,ex2,j,k)
     $           - dsspm * (           ue(m,ex-4)
     $                       - 4.0d+00 * ue(m,ex3)
     $                       + 6.0d+00 * ue(m,ex2)
     $                       - 4.0d+00 * ue(m,ex1)  )
c
               frct(m,ex1,j,k) = frct(m,ex1,j,k)
     $           - dsspm * (           ue(m,ex3)
     $                       - 4.0d+00 * ue(m,ex2)
     $                       + 5.0d+00 * ue(m,ex1) )
c
            end do
            endif
c
         end do
c
      end do
c
```



```
c
c***eta-direction flux differences
c
      do k = sz1, ez1
c
         kg = kglob(k)
         zeta = ( dfloat(kg-1) ) / ( nz - 1 )
c
         do i = sx1, ex1
c
            ig = iglob(i)
            xi = ( dfloat(ig-1) ) / ( nx - 1 )
c
            do j = sym2, eym2
c
               jg = jglob(j)
               eta = ( dfloat(jg-1) ) / ( ny - 1 )
c
               do m = 1, 5
c
                  ue(m,j) =  ce(m,1)
     $                 + ce(m,2) * xi
     $                 + ce(m,3) * eta
     $                 + ce(m,4) * zeta
     $                 + ce(m,5) * xi * xi
     $                 + ce(m,6) * eta * eta
     $                 + ce(m,7) * zeta * zeta
     $                 + ce(m,8) * xi * xi * xi
     $                 + ce(m,9) * eta * eta * eta
     $                 + ce(m,10) * zeta * zeta * zeta
     $                 + ce(m,11) * xi * xi * xi * xi
     $                 + ce(m,12) * eta * eta * eta * eta
     $                 + ce(m,13) * zeta * zeta * zeta * zeta
c
               end do
c
               flux(1,j) = ue(3,j)
c
               u31 = ue(3,j) / ue(1,j)
c
               q = 0.50d+00 * (  ue(2,j) * ue(2,j)
     $                         + ue(3,j) * ue(3,j)
     $                         + ue(4,j) * ue(4,j) )
     $                       / ue(1,j)
c
               flux(2,j) = ue(2,j) * u31
c
               flux(3,j) = ue(3,j) * u31 + c2 * ( ue(5,j) - q )
c
               flux(4,j) = ue(4,j) * u31
c
               flux(5,j) = ( c1 * ue(5,j) - c2 * q ) * u31
c
            end do
c
            do j = sy1, ey1
```

```
c
               do m = 1, 5
c
                  frct(m,i,j,k) =  frct(m,i,j,k)
     $                        - ty2 * ( flux(m,j+1) - flux(m,j-1) )
c
               end do
c
            end do
c
            do j = sy1, eym1
c
               tmp = 1.0d+00 / ue(1,j)
c
               u21j = tmp * ue(2,j)
               u31j = tmp * ue(3,j)
               u41j = tmp * ue(4,j)
               u51j = tmp * ue(5,j)
c
               tmp = 1.0d+00 / ue(1,j-1)
c
               u21jm1 = tmp * ue(2,j-1)
               u31jm1 = tmp * ue(3,j-1)
               u41jm1 = tmp * ue(4,j-1)
               u51jm1 = tmp * ue(5,j-1)
c
               flux(2,j) = ty3 * ( u21j - u21jm1 )
               flux(3,j) = (4.0d+00/3.0d+00) * ty3 * ( u31j - u31jm1 )
               flux(4,j) = ty3 * ( u41j - u41jm1 )
               flux(5,j) = 0.50d+00 * ( 1.0d+00 - c1*c5 )
     $              * ty3 * ( ( u21j  **2 + u31j  **2 + u41j  **2 )
     $                      - ( u21jm1**2 + u31jm1**2 + u41jm1**2 ) )
     $              + (1.0d+00/6.0d+00)
     $              * ty3 * ( u31j**2 - u31jm1**2 )
     $              + c1 * c5 * ty3 * ( u51j - u51jm1 )
c
            end do
c
            do j = sy1, ey1
c
               frct(1,i,j,k) = frct(1,i,j,k)
     $              + dy1 * ty1 * (           ue(1,j-1)
     $                            - 2.0d+00 * ue(1,j)
     $                            +           ue(1,j+1) )
c
               frct(2,i,j,k) = frct(2,i,j,k)
     $              + ty3 * c3 * c4 * ( flux(2,j+1) - flux(2,j) )
     $              + dy2 * ty1 * (           ue(2,j-1)
     $                            - 2.0d+00 * ue(2,j)
     $                            +           ue(2,j+1) )
c
               frct(3,i,j,k) = frct(3,i,j,k)
     $              + ty3 * c3 * c4 * ( flux(3,j+1) - flux(3,j) )
     $              + dy3 * ty1 * (           ue(3,j-1)
     $                            - 2.0d+00 * ue(3,j)
     $                            +           ue(3,j+1) )
```



```
c
               frct(4,i,j,k) = frct(4,i,j,k)
     $              + ty3 * c3 * c4 * ( flux(4,j+1) - flux(4,j) )
     $              + dy4 * ty1 * (           ue(4,j-1)
     $                            - 2.0d+00 * ue(4,j)
     $                            +           ue(4,j+1) )
c
               frct(5,i,j,k) = frct(5,i,j,k)
     $              + ty3 * c3 * c4 * ( flux(5,j+1) - flux(5,j) )
     $              + dy5 * ty1 * (           ue(5,j-1)
     $                            - 2.0d+00 * ue(5,j)
     $                            +           ue(5,j+1) )
c
            end do
c
c***fourth-order dissipation
c
            if ( gsty .eq. 1 ) then
            do m = 1, 5
c
               frct(m,i,sy1,k) = frct(m,i,sy1,k)
     $           - dsspm * ( + 5.0d+00 * ue(m,sy1)
     $                       - 4.0d+00 * ue(m,sy2)
     $                       +           ue(m,sy3) )
c
               frct(m,i,sy2,k) = frct(m,i,sy2,k)
     $           - dsspm * ( - 4.0d+00 * ue(m,sy1)
     $                       + 6.0d+00 * ue(m,sy2)
     $                       - 4.0d+00 * ue(m,sy3)
     $                       +           ue(m,sy+4) )
c
            end do
            endif
c
            do j = sy3, ey3
c
               do m = 1, 5
c
                  frct(m,i,j,k) = frct(m,i,j,k)
     $              - dsspm * (           ue(m,j-2)
     $                        - 4.0d+00 * ue(m,j-1)
     $                        + 6.0d+00 * ue(m,j)
     $                        - 4.0d+00 * ue(m,j+1)
     $                        +           ue(m,j+2) )
c
               end do
c
            end do
c
            if ( gendy .eq. ny ) then
            do m = 1, 5
c
               frct(m,i,ey2,k) = frct(m,i,ey2,k)
     $           - dsspm * (           ue(m,ey-4)
     $                     - 4.0d+00 * ue(m,ey3)
     $                     + 6.0d+00 * ue(m,ey2)
```

```
     $                        - 4.0d+00 * ue(m,ey1)   )
c
               frct(m,i,ey1,k) = frct(m,i,ey1,k)
     $           - dsspm * (             ue(m,ey3)
     $                        - 4.0d+00 * ue(m,ey2)
     $                        + 5.0d+00 * ue(m,ey1)   )
c
            end do
            endif
c
         end do
c
      end do
c
c
c***zeta-direction flux differences
c
      do j = sy1, ey1
c
         jg = jglob(j)
         eta = ( dfloat(jg-1) ) / ( ny - 1 )
c
         do i = sx1, ex1
c
            ig = iglob(i)
            xi = ( dfloat(ig-1) ) / ( nx - 1 )
c
            do k = szm2, ezm2
c
               kg = kglob(k)
               zeta = ( dfloat(kg-1) ) / ( nz - 1 )
c
               do m = 1, 5
c
                  ue(m,k) =  ce(m,1)
     $                 + ce(m,2) * xi
     $                 + ce(m,3) * eta
     $                 + ce(m,4) * zeta
     $                 + ce(m,5) * xi * xi
     $                 + ce(m,6) * eta * eta
     $                 + ce(m,7) * zeta * zeta
     $                 + ce(m,8) * xi * xi * xi
     $                 + ce(m,9) * eta * eta * eta
     $                 + ce(m,10) * zeta * zeta * zeta
     $                 + ce(m,11) * xi * xi * xi * xi
     $                 + ce(m,12) * eta * eta * eta * eta
     $                 + ce(m,13) * zeta * zeta * zeta * zeta
c
               end do
c
               flux(1,k) = ue(4,k)
c
               u41 = ue(4,k) / ue(1,k)
c
               q = 0.50d+00 * (  ue(2,k) * ue(2,k)
     $                         + ue(3,k) * ue(3,k)
```



```
     $                         + ue(4,k) * ue(4,k) )
     $                       / ue(1,k)
c
               flux(2,k) = ue(2,k) * u41
c
               flux(3,k) = ue(3,k) * u41
c
               flux(4,k) = ue(4,k) * u41 + c2 * ( ue(5,k) - q )
c
               flux(5,k) = ( c1 * ue(5,k) - c2 * q ) * u41
c
            end do
c
            do k = sz1, ez1
c
               do m = 1, 5
c
                  frct(m,i,j,k) =  frct(m,i,j,k)
     $                          - tz2 * ( flux(m,k+1) - flux(m,k-1) )
c
               end do
c
            end do
c
            do k = sz1, ezm1
c
               tmp = 1.0d+00 / ue(1,k)
c
               u21k = tmp * ue(2,k)
               u31k = tmp * ue(3,k)
               u41k = tmp * ue(4,k)
               u51k = tmp * ue(5,k)
c
               tmp = 1.0d+00 / ue(1,k-1)
c
               u21km1 = tmp * ue(2,k-1)
               u31km1 = tmp * ue(3,k-1)
               u41km1 = tmp * ue(4,k-1)
               u51km1 = tmp * ue(5,k-1)
c
               flux(2,k) = tz3 * ( u21k - u21km1 )
               flux(3,k) = tz3 * ( u31k - u31km1 )
               flux(4,k) = (4.0d+00/3.0d+00) * tz3 * ( u41k - u41km1 )
               flux(5,k) = 0.50d+00 * ( 1.0d+00 - c1*c5 )
     $              * tz3 * ( ( u21k  **2 + u31k  **2 + u41k  **2 )
     $                      - ( u21km1**2 + u31km1**2 + u41km1**2 ) )
     $              + (1.0d+00/6.0d+00)
     $              * tz3 * ( u41k**2 - u41km1**2 )
     $              + c1 * c5 * tz3 * ( u51k - u51km1 )
c
            end do
c
            do k = sz1, ez1
c
               frct(1,i,j,k) = frct(1,i,j,k)
     $              + dz1 * tz1 * (            ue(1,k+1)
```

```
     $                            - 2.0d+00 * ue(1,k)
     $                            +           ue(1,k-1) )
c
               frct(2,i,j,k) = frct(2,i,j,k)
     $              + tz3 * c3 * c4 * ( flux(2,k+1) - flux(2,k) )
     $              + dz2 * tz1 * (           ue(2,k+1)
     $                            - 2.0d+00 * ue(2,k)
     $                            +           ue(2,k-1) )
c
               frct(3,i,j,k) = frct(3,i,j,k)
     $              + tz3 * c3 * c4 * ( flux(3,k+1) - flux(3,k) )
     $              + dz3 * tz1 * (           ue(3,k+1)
     $                            - 2.0d+00 * ue(3,k)
     $                            +           ue(3,k-1) )
c
               frct(4,i,j,k) = frct(4,i,j,k)
     $              + tz3 * c3 * c4 * ( flux(4,k+1) - flux(4,k) )
     $              + dz4 * tz1 * (           ue(4,k+1)
     $                            - 2.0d+00 * ue(4,k)
     $                            +           ue(4,k-1) )
c
               frct(5,i,j,k) = frct(5,i,j,k)
     $              + tz3 * c3 * c4 * ( flux(5,k+1) - flux(5,k) )
     $              + dz5 * tz1 * (           ue(5,k+1)
     $                            - 2.0d+00 * ue(5,k)
     $                            +           ue(5,k-1) )
c
            end do
c
c***fourth-order dissipation
c
c
            if ( gstz .eq. 1 ) then
            do m = 1, 5
c
               frct(m,i,j,sz1) = frct(m,i,j,sz1)
     $           - dsspm * ( + 5.0d+00 * ue(m,sz1)
     $                       - 4.0d+00 * ue(m,sz2)
     $                       +           ue(m,sz3) )
c
               frct(m,i,j,sz2) = frct(m,i,j,sz2)
     $           - dsspm * (- 4.0d+00 * ue(m,sz1)
     $                      + 6.0d+00 * ue(m,sz2)
     $                      - 4.0d+00 * ue(m,sz3)
     $                      +           ue(m,sz+4) )
c
            end do
            endif
c
            do k = sz3, ez3
c
               do m = 1, 5
c
                  frct(m,i,j,k) = frct(m,i,j,k)
     $              - dsspm * (           ue(m,k-2)
     $                        - 4.0d+00 * ue(m,k-1)
```



```
     $                        + 6.0d+00 * ue(m,k)
     $                        - 4.0d+00 * ue(m,k+1)
     $                        +           ue(m,k+2) )
c
                  end do
c
               end do
c
               if ( gendz .eq. nz ) then
               do m = 1, 5
c
                  frct(m,i,j,ez2) = frct(m,i,j,ez2)
     $              - dsspm * (            ue(m,ez-4)
     $                         - 4.0d+00 * ue(m,ez3)
     $                         + 6.0d+00 * ue(m,ez2)
     $                         - 4.0d+00 * ue(m,ez1)  )
c
                  frct(m,i,j,ez1) = frct(m,i,j,ez1)
     $              - dsspm * (             ue(m,ez3)
     $                          - 4.0d+00 * ue(m,ez2)
     $                          + 5.0d+00 * ue(m,ez1)  )
c
               end do
               endif
c
            end do
c
         end do
c
c     write(iout,*) igstx, igsty, igstz, 'frct3', frct(2,4,4,8)
c
         return
         end
c
c
c
c
c
         subroutine exact ( i, j, k, u000ijk )
c
c***compute the exact solution at (i,j,k)
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
         include 'appsp.incl'
c
         dimension u000ijk(*)
c
c Sanjeev : DONT change these i,j,k to global values ! the caller does it
         xi  = ( dfloat ( i - 1 ) ) / ( nx - 1 )
         eta  = ( dfloat ( j - 1 ) ) / ( ny - 1 )
         zeta = ( dfloat ( k - 1 ) ) / ( nz - 1 )
c
         do m = 1, 5
```

```
c
          u000ijk(m) =  ce(m,1)
     $         + ce(m,2) * xi
     $         + ce(m,3) * eta
     $         + ce(m,4) * zeta
     $         + ce(m,5) * xi * xi
     $         + ce(m,6) * eta * eta
     $         + ce(m,7) * zeta * zeta
     $         + ce(m,8) * xi * xi * xi
     $         + ce(m,9) * eta * eta * eta
     $         + ce(m,10) * zeta * zeta * zeta
     $         + ce(m,11) * xi * xi * xi * xi
     $         + ce(m,12) * eta * eta * eta * eta
     $         + ce(m,13) * zeta * zeta * zeta * zeta
c
      end do
c
      return
      end
c
c
c
c
c
      subroutine error(u,enorms,igstx, igendx, igsty, igendy, igstz,
     $                 igendz)
c
c***compute the solution error
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize)
      real *8 enorms(5)
c
      dimension imax(5), jmax(5), kmax(5),
     $          u000ijk(5), errmax(5)
c
      integer igstx, igendx, igsty, igendy, igstz, igendz
c
      gstx = igstx
      gendx = igendx
      gsty = igsty
      gendy = igendy
      gstz = igstz
      gendz = igendz
c
      lnorm = 2
c
      if ( lnorm .eq. 1 ) then
c
         do m = 1, 5
            errmax(m) = - 1.0d+20
```



```
         end do
c
         do k = sz1, ez1
            do j = sy1, ey1
               do i = sx1, ex1
c
                  call exact ( iglob(i), jglob(j), kglob(k), u000ijk )
c
                  do m = 1, 5
c
                     tmp = abs ( u000ijk(m) - u(m,i,j,k) )
c
                     if ( tmp .gt. errmax(m) ) then
c
                        errmax(m) = tmp
                        imax(m) = iglob(i)
                        jmax(m) = jglob(j)
                        kmax(m) = kglob(k)
c
                     end if
c
                  end do
c
               end do
            end do
         end do
c
c        write (iout,1001) ( errmax(m),
c     $        imax(m), jmax(m), kmax(m), m = 1, 5 )
c
      else if ( lnorm .eq. 2 ) then
c
         do m = 1, 5
            errnm(m) = 0.0d+00
         end do
c
         do k = sz1, ez1
            do j = sy1, ey1
               do i = sx1, ex1
c
                  call exact ( iglob(i), jglob(j), kglob(k), u000ijk )
c
                  do m = 1, 5
c
                     tmp = ( u000ijk(m) - u(m,i,j,k) )
c
                     errnm(m) = errnm(m) + tmp ** 2
c
                  end do
c
               end do
            end do
         end do
c
c        do m = 1, 5
c           errnm(m) = sqrt ( errnm(m) / ( (nx-2)*(ny-2)*(nz-2) ) )
```

```
c        end do
c
c        write (iout,1002) ( errnm(m), m = 1, 5 )
c
         do m = 1, 5
            enorms(m) = enorms(m) + errnm(m)
         end do
c
      end if
c
 1001 format (/5x,'max. error in soln. to first pde  =',1pe12.4/,
     $5x,'and its location                  = (',i4',',i4,',',i4,' )'/,
     $/5x,'max. error in soln. to second pde =',1pe12.4/,
     $5x,'and its location                  = (',i4',',i4,',',i4,' )'/,
     $/5x,'max. error in soln. to third pde  =',1pe12.4/,
     $5x,'and its location                  = (',i4',',i4,',',i4,' )'/,
     $/5x,'max. error in soln. to fourth pde =',1pe12.4/,
     $5x,'and its location                  = (',i4',',i4,',',i4,' )'/,
     $/5x,'max. error in soln. to fifth pde  =',1pe12.4/,
     $5x,'and its location                  = (',i4',',i4,',',i4,' )' )
c
 1002 format (1x/1x,'RMS-norm of error in soln. to ',
     $ 'first pde  = ',1pe12.5/,
     $ 1x,'RMS-norm of error in soln. to ',
     $ 'second pde = ',1pe12.5/,
     $ 1x,'RMS-norm of error in soln. to ',
     $ 'third pde  = ',1pe12.5/,
     $ 1x,'RMS-norm of error in soln. to ',
     $ 'fourth pde = ',1pe12.5/,
     $ 1x,'RMS-norm of error in soln. to ',
     $ 'fifth pde  = ',1pe12.5)
c
      return
      end
c
c
c
c
c
      subroutine jacx ( m, u,a,b,c,d,e )
c
c***form the xi-direction pentadiagonal system
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize), a(xsize,ysize,zsize),
     $        b(xsize,ysize,zsize), c(xsize,ysize,zsize),
     $        d(xsize,ysize,zsize), e(xsize,ysize,zsize)
c
      dimension cv(xsize), aa(xsize), rhon(xsize)
c
      r43 = 4.0d+00 / 3.0d+00
```



```
      c34 = c3 * c4
      c1345 = c1 * c3 * c4 * c5
c
      if ( m .eq. 3 ) then
c
         sn = 0.0d+00
c
      else if ( m .eq. 4 ) then
c
         sn = 1.0d+00
c
      else if ( m .eq. 5 ) then
c
         sn = - 1.0d+00
c
      end if
c
      do k = sz1, ez1
c
         do j = sy1, ey1
c
            do i = sxm1, exm1
c
               ru1 = 1.0d+00 / u(1,i,j,k)
               uu = ru1 * u(2,i,j,k)
               vv = ru1 * u(3,i,j,k)
               ww = ru1 * u(4,i,j,k)
c
               q = 0.50d+00 * (  uu ** 2
     $                         + vv ** 2
     $                         + ww ** 2 )
c
               cv(i) = uu
               aa(i) = sqrt ( c1 * c2 * ( ru1 * u(5,i,j,k) - q ) )
c
               rhon(i) = max ( dx1,
     $                         dx2 + r43 * c34 * ru1,
     $                         dx3 + c34 * ru1,
     $                         dx4 + c34 * ru1,
     $                         dx5 + c1345 * ru1 )
c
            end do
c
            if ( gstx .eq. 1 ) then
               a(sx,j,k) = 0.0d+00
               b(sx,j,k) = 0.0d+00
               c(sx,j,k) = 1.0d+00
               d(sx,j,k) = 0.0d+00
               e(sx,j,k) = 0.0d+00
            else
               e(sx-1,j,k) = 0.0d+00
               e(sx-2,j,k) = 0.0d+00
            endif
c
            do i = sx1, ex1
c
```

```
               a(i,j,k) =   0.0d+00
               b(i,j,k) = - dt * tx2 * ( cv(i-1) + sn * aa(i-1) )
     $                    - dt * rhon(i-1) * tx1
               c(i,j,k) =   1.0d+00
     $                    + dt * rhon(i) * tx1 * 2.0d+00
               d(i,j,k) =   dt * tx2 * ( cv(i+1) + sn * aa(i+1) )
     $                    - dt * rhon(i+1) * tx1
               e(i,j,k) =   0.0d+00
c
            end do
c
            if ( gendx .eq. nx ) then
               a(ex,j,k) = 0.0d+00
               b(ex,j,k) = 0.0d+00
               c(ex,j,k) = 1.0d+00
               d(ex,j,k) = 0.0d+00
               e(ex,j,k) = 0.0d+00
            endif
c
c***fourth order dissipation
c
            if ( gstx .eq. 1 ) then
               c(sx1,j,k) = c(sx1,j,k) + dt * dssp * ( + 5.0d+00 )
               d(sx1,j,k) = d(sx1,j,k) + dt * dssp * ( - 4.0d+00 )
               e(sx1,j,k) = e(sx1,j,k) + dt * dssp * ( + 1.0d+00 )
c
               b(sx2,j,k) = b(sx2,j,k) + dt * dssp * ( - 4.0d+00 )
               c(sx2,j,k) = c(sx2,j,k) + dt * dssp * ( + 6.0d+00 )
               d(sx2,j,k) = d(sx2,j,k) + dt * dssp * ( - 4.0d+00 )
               e(sx2,j,k) = e(sx2,j,k) + dt * dssp * ( + 1.0d+00 )
            else
               e(sx-1,j,k) = e(sx-1,j,k) + dt * dssp * ( + 1.0d+00 )
               e(sx-2,j,k) = e(sx-2,j,k) + dt * dssp * ( + 1.0d+00 )
            endif
c
c           begin = sxm2
c           if ( gstx .eq. 1 ) begin = sx + 3
c           end = exm2
c           if ( gendx .eq. nx ) end = ex - 3
c
            do i = sx3, ex3
c
               a(i,j,k) = a(i,j,k) + dt * dssp * ( + 1.0d+00 )
               b(i,j,k) = b(i,j,k) + dt * dssp * ( - 4.0d+00 )
               c(i,j,k) = c(i,j,k) + dt * dssp * ( + 6.0d+00 )
               d(i,j,k) = d(i,j,k) + dt * dssp * ( - 4.0d+00 )
               e(i,j,k) = e(i,j,k) + dt * dssp * ( + 1.0d+00 )
c
            end do
c
            if ( gendx .eq. nx ) then
               a(ex2,j,k) = a(ex2,j,k) + dt * dssp * ( + 1.0d+00 )
               b(ex2,j,k) = b(ex2,j,k) + dt * dssp * ( - 4.0d+00 )
               c(ex2,j,k) = c(ex2,j,k) + dt * dssp * ( + 6.0d+00 )
               d(ex2,j,k) = d(ex2,j,k) + dt * dssp * ( - 4.0d+00 )
c
```



```
               a(ex1,j,k) = a(ex1,j,k) + dt * dssp * ( + 1.0d+00 )
               b(ex1,j,k) = b(ex1,j,k) + dt * dssp * ( - 4.0d+00 )
               c(ex1,j,k) = c(ex1,j,k) + dt * dssp * ( + 5.0d+00 )
            endif
c
         end do
c
      end do
c
      return
      end
c
c
c
c
c
      subroutine jacy ( m, u,a,b,c,d,e )
c
c***form the eta-direction pentadiagonal system.
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize), a(xsize,ysize,zsize),
     $        b(xsize,ysize,zsize), c(xsize,ysize,zsize),
     $        d(xsize,ysize,zsize), e(xsize,ysize,zsize)
c
      dimension cv(ysize), aa(ysize), rhoq(ysize)
c
      r43 = 4.0d+00 / 3.0d+00
      c34 = c3 * c4
      c1345 = c1 * c3 * c4 * c5
c
      if ( m .eq. 3 ) then
c
         sn = 0.0d+00
c
      else if ( m .eq. 4 ) then
c
         sn = 1.0d+00
c
      else if ( m .eq. 5 ) then
c
         sn = - 1.0d+00
c
      end if
c
      do k = sz1, ez1
c
         do i = sx1, ex1
c
            do j = sym1, eym1
c
```

```
            ru1 = 1.0d+00 / u(1,i,j,k)
            uu = ru1 * u(2,i,j,k)
            vv = ru1 * u(3,i,j,k)
            ww = ru1 * u(4,i,j,k)
c
            q = 0.50d+00 * (  uu ** 2
     $                      + vv ** 2
     $                      + ww ** 2 )
c
            cv(j) = vv
            aa(j) = sqrt ( c1 * c2 * ( ru1 * u(5,i,j,k) - q ) )
c
            rhoq(j) = max ( dy1,
     $                      dy2 + c34 * ru1,
     $                      dy3 + r43 * c34 * ru1,
     $                      dy4 + c34 * ru1,
     $                      dy5 + c1345 * ru1 )
c
         end do
c
         if ( gsty .eq. 1 ) then
            a(i,sy,k) = 0.0d+00
            b(i,sy,k) = 0.0d+00
            c(i,sy,k) = 1.0d+00
            d(i,sy,k) = 0.0d+00
            e(i,sy,k) = 0.0d+00
         else
            e(i,sy-1,k) = 0.0d+00
            e(i,sy-2,k) = 0.0d+00
         endif
c
c        begin = sym2
c        if ( gsty .eq. 1 ) begin = sy + 1
c        end = eym2
c        if ( gendy .eq. ny ) end = ey - 1
c
         do j = sy1, ey1
c
            a(i,j,k) = 0.0d+00
            b(i,j,k) = - dt * ty2 * ( cv(j-1) + sn * aa(j-1) )
     $                 - dt * rhoq(j-1) * ty1
            c(i,j,k) =   1.0d+00
     $                 + dt * rhoq(j) * ty1 * 2.0d+00
            d(i,j,k) =   dt * ty2 * ( cv(j+1) + sn * aa(j+1) )
     $                 - dt * rhoq(j+1) * ty1
            e(i,j,k) = 0.0d+00
c
         end do
c
         if ( gendy .eq. ny ) then
            a(i,ey,k) = 0.0d+00
            b(i,ey,k) = 0.0d+00
            c(i,ey,k) = 1.0d+00
            d(i,ey,k) = 0.0d+00
            e(i,ey,k) = 0.0d+00
         endif
```



```
c
c***fourth order dissipation
c
            if ( gsty .eq. 1 ) then
               c(i,sy1,k) = c(i,sy1,k) + dt * dssp * ( + 5.0d+00 )
               d(i,sy1,k) = d(i,sy1,k) + dt * dssp * ( - 4.0d+00 )
               e(i,sy1,k) = e(i,sy1,k) + dt * dssp * ( + 1.0d+00 )
c
               b(i,sy2,k) = b(i,sy2,k) + dt * dssp * ( - 4.0d+00 )
               c(i,sy2,k) = c(i,sy2,k) + dt * dssp * ( + 6.0d+00 )
               d(i,sy2,k) = d(i,sy2,k) + dt * dssp * ( - 4.0d+00 )
               e(i,sy2,k) = e(i,sy2,k) + dt * dssp * ( + 1.0d+00 )
            else
               e(i,sy-1,k) = e(i,sy-1,k) + dt * dssp * ( + 1.0d+00 )
               e(i,sy-2,k) = e(i,sy-2,k) + dt * dssp * ( + 1.0d+00 )
            endif
c
c           begin = sym2
c           if ( gsty .eq. 1 ) begin = sy + 3
c           end = eym2
c           if ( gendy .eq. ny ) end = ey - 3
c
            do j = sy3, ey3
c
               a(i,j,k) = a(i,j,k) + dt * dssp * ( + 1.0d+00 )
               b(i,j,k) = b(i,j,k) + dt * dssp * ( - 4.0d+00 )
               c(i,j,k) = c(i,j,k) + dt * dssp * ( + 6.0d+00 )
               d(i,j,k) = d(i,j,k) + dt * dssp * ( - 4.0d+00 )
               e(i,j,k) = e(i,j,k) + dt * dssp * ( + 1.0d+00 )
c
            end do
c
            if ( gendy .eq. ny ) then
               a(i,ey2,k) = a(i,ey2,k) + dt * dssp * ( + 1.0d+00 )
               b(i,ey2,k) = b(i,ey2,k) + dt * dssp * ( - 4.0d+00 )
               c(i,ey2,k) = c(i,ey2,k) + dt * dssp * ( + 6.0d+00 )
               d(i,ey2,k) = d(i,ey2,k) + dt * dssp * ( - 4.0d+00 )
c
               a(i,ey1,k) = a(i,ey1,k) + dt * dssp * ( + 1.0d+00 )
               b(i,ey1,k) = b(i,ey1,k) + dt * dssp * ( - 4.0d+00 )
               c(i,ey1,k) = c(i,ey1,k) + dt * dssp * ( + 5.0d+00 )
            endif
c
         end do
c
      end do
c
      return
      end
c
c
c
c
c
      subroutine jacz ( m, u,a,b,c,d,e )
c
```

```
c***form the zeta-direction pentadiagonal system.
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize), a(xsize,ysize,zsize),
     $        b(xsize,ysize,zsize), c(xsize,ysize,zsize),
     $        d(xsize,ysize,zsize), e(xsize,ysize,zsize)
c
      dimension cv(zsize), aa(zsize), rhos(zsize)
c
      r43 = 4.0d+00 / 3.0d+00
      c34 = c3 * c4
      c1345 = c1 * c3 * c4 * c5
c
      if ( m .eq. 3 ) then
c
         sn = 0.0d+00
c
      else if ( m .eq. 4 ) then
c
         sn = 1.0d+00
c
      else if ( m .eq. 5 ) then
c
         sn = - 1.0d+00
c
      end if
c
      do j = sy1, ey1
c
         do i = sx1, ex1
c
            do k = szm1, ezm1
c
               ru1 = 1.0d+00 / u(1,i,j,k)
               uu = ru1 * u(2,i,j,k)
               vv = ru1 * u(3,i,j,k)
               ww = ru1 * u(4,i,j,k)
c
               q = 0.50d+00 * (  uu ** 2
     $                         + vv ** 2
     $                         + ww ** 2 )
c
               cv(k) = ww
               aa(k) = sqrt ( c1 * c2 * ( ru1 * u(5,i,j,k) - q ) )
c
               rhos(k) = max ( dz1,
     $                         dz2 + c34 * ru1 ,
     $                         dz3 + c34 * ru1 ,
     $                         dz4 + r43 * c34 * ru1,
     $                         dz5 + c1345 * ru1 )
c
```



```
            end do
c
            if ( gstz .eq. 1 ) then
               a(i,j,sz) = 0.0d+00
               b(i,j,sz) = 0.0d+00
               c(i,j,sz) = 1.0d+00
               d(i,j,sz) = 0.0d+00
               e(i,j,sz) = 0.0d+00
            else
               e(i,j,sz-1) = 0.0d+00
               e(i,j,sz-2) = 0.0d+00
            endif
c
c           begin = szm2
c           if ( gstz .eq. 1 ) begin = sz + 1
c           end = ezm2
c           if ( gendz .eq. nz ) end = ez - 1
c
            do k = sz1, ez1
c
               a(i,j,k) = 0.0d+00
               b(i,j,k) = - dt * tz2 * ( cv(k-1) + sn * aa(k-1) )
     $                    - dt * rhos(k-1) * tz1
               c(i,j,k) =   1.0d+00
     $                    + dt * rhos(k) * tz1 * 2.0d+00
               d(i,j,k) =   dt * tz2 * ( cv(k+1) + sn * aa(k+1) )
     $                    - dt * rhos(k+1) * tz1
               e(i,j,k) = 0.0d+00
c
            end do
c
            if ( gendz .eq. nz ) then
               a(i,j,ez) = 0.0d+00
               b(i,j,ez) = 0.0d+00
               c(i,j,ez) = 1.0d+00
               d(i,j,ez) = 0.0d+00
               e(i,j,ez) = 0.0d+00
            endif
c
c***fourth order dissipation
c
            if ( gstz .eq. 1 ) then
               c(i,j,sz1) = c(i,j,sz1) + dt * dssp * ( + 5.0d+00 )
               d(i,j,sz1) = d(i,j,sz1) + dt * dssp * ( - 4.0d+00 )
               e(i,j,sz1) = e(i,j,sz1) + dt * dssp * ( + 1.0d+00 )
c
               b(i,j,sz2) = b(i,j,sz2) + dt * dssp * ( - 4.0d+00 )
               c(i,j,sz2) = c(i,j,sz2) + dt * dssp * ( + 6.0d+00 )
               d(i,j,sz2) = d(i,j,sz2) + dt * dssp * ( - 4.0d+00 )
               e(i,j,sz2) = e(i,j,sz2) + dt * dssp * ( + 1.0d+00 )
            else
               e(i,j,sz-1) = e(i,j,sz-1) + dt * dssp * ( + 1.0d+00 )
               e(i,j,sz-2) = e(i,j,sz-2) + dt * dssp * ( + 1.0d+00 )
            endif
c
c           begin = szm2
```

```
c           if ( gstz .eq. 1 ) begin = sz + 3
c           end = ezm2
c           if ( gendz .eq. nz ) end = ez - 3
c
            do k = sz3, ez3
c
               a(i,j,k) = a(i,j,k) + dt * dssp * ( + 1.0d+00 )
               b(i,j,k) = b(i,j,k) + dt * dssp * ( - 4.0d+00 )
               c(i,j,k) = c(i,j,k) + dt * dssp * ( + 6.0d+00 )
               d(i,j,k) = d(i,j,k) + dt * dssp * ( - 4.0d+00 )
               e(i,j,k) = e(i,j,k) + dt * dssp * ( + 1.0d+00 )
c
            end do
c
            if ( gendz .eq. nz ) then
               a(i,j,ez2) = a(i,j,ez2) + dt * dssp * ( + 1.0d+00 )
               b(i,j,ez2) = b(i,j,ez2) + dt * dssp * ( - 4.0d+00 )
               c(i,j,ez2) = c(i,j,ez2) + dt * dssp * ( + 6.0d+00 )
               d(i,j,ez2) = d(i,j,ez2) + dt * dssp * ( - 4.0d+00 )
c
               a(i,j,ez1) = a(i,j,ez1) + dt * dssp * ( + 1.0d+00 )
               b(i,j,ez1) = b(i,j,ez1) + dt * dssp * ( - 4.0d+00 )
               c(i,j,ez1) = c(i,j,ez1) + dt * dssp * ( + 5.0d+00 )
            endif
c
         end do
c
      end do
c
      return
      end
c
c
c
c
c
      subroutine ninvr(rsd)
c
c***block-diagonal matrix-vector multiply
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 rsd(5,xsize,ysize,zsize)
c
      bt = sqrt ( 0.50d+00 )
c
      do k = sz1, ez1
c
         do j = sy1, ey1
c
            do i = sx1, ex1
c
```



```
               r1 = rsd(1,i,j,k)
               r2 = rsd(2,i,j,k)
               r3 = rsd(3,i,j,k)
               r4 = rsd(4,i,j,k)
               r5 = rsd(5,i,j,k)
c
               rsd(1,i,j,k) = - r2
c
               rsd(2,i,j,k) =   r1
c
               rsd(3,i,j,k) = bt * ( r4 - r5 )
c
               t1 = bt * r3
c
               t2 = 0.50d+00 * ( r4 + r5 )
c
               rsd(4,i,j,k) = - t1 + t2
c
               rsd(5,i,j,k) =   t1 + t2
c
            end do
c
         end do
c
      end do
c
      return
      end
c
c
c
c
      subroutine get1z(u, phi1z)
c
c***compute the surface integral
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize)
c
      dimension phi1z(xsize,ysize)
c
      do j = sy1, ey2
c
         do i = sx1, ex1
c
            phi1z(i,j) = c2*(  u(5,i,j,sz2)
     $           - 0.50d+00 * (  u(2,i,j,sz2) ** 2
     $                         + u(3,i,j,sz2) ** 2
     $                         + u(4,i,j,sz2) ** 2 )
     $                        / u(1,i,j,sz2) )
c
         end do
c
      end do
c
```

```
      return
      end
c
c
c
      subroutine get2z(u, phi2z)
c
c***compute the surface integral
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize)
c
      dimension phi2z(xsize,ysize)
c
      do j = sy1, ey2
c
         do i = sx1, ex1
c
            phi2z(i,j) = c2*(  u(5,i,j,ez1)
     $           - 0.50d+00 * (  u(2,i,j,ez1) ** 2
     $                         + u(3,i,j,ez1) ** 2
     $                         + u(4,i,j,ez1) ** 2 )
     $                        / u(1,i,j,ez1) )
c
         end do
c
      end do
c
      return
      end
c
c
      subroutine get1y(u, phi1y)
c
c***compute the surface integral
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize)
c
      dimension phi1y(xsize,zsize)
c
      do k = sz2, ez1
c
         do i = sx1, ex1
c
            phi1y(i,k) = c2*(  u(5,i,sy1,k)
     $           - 0.50d+00 * (  u(2,i,sy1,k) ** 2
     $                         + u(3,i,sy1,k) ** 2
     $                         + u(4,i,sy1,k) ** 2 )
     $                        / u(1,i,sy1,k) )
c
         end do
c
      end do
```



```
c
      return
      end
c
c
      subroutine get2y(u, phi2y)
c
c***compute the surface integral
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize)
c
      dimension phi2y(xsize,zsize)
c
      do k = sz2, ez1
c
         do i = sx1, ex1
c
            phi2y(i,k) = c2*(  u(5,i,ey2,k)
     $           - 0.50d+00 * (  u(2,i,ey2,k) ** 2
     $                         + u(3,i,ey2,k) ** 2
     $                         + u(4,i,ey2,k) ** 2 )
     $                        / u(1,i,ey2,k) )
c
         end do
c
      end do
c
      return
      end
c
c
      subroutine get1x(u, phi1x)
c
c***compute the surface integral
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize)
c
      dimension phi1x(ysize,zsize)
c
      do k = sz2, ez1
c
         do j = sy1, ey2
c
            phi1x(j,k) = c2*(  u(5,sx1,j,k)
     $           - 0.50d+00 * (  u(2,sx1,j,k) ** 2
     $                         + u(3,sx1,j,k) ** 2
     $                         + u(4,sx1,j,k) ** 2 )
     $                        / u(1,sx1,j,k) )
c
         end do
c
      end do
```

```
c
      return
      end
c
c
      subroutine get2x(u, phi2x)
c
c***compute the surface integral
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize)
c
      dimension phi2x(ysize,zsize)
c
      do k = sz2, ez1
c
         do j = sy1, ey2
c
            phi2x(j,k) = c2*(  u(5,ex1,j,k)
     $           - 0.50d+00 * (  u(2,ex1,j,k) ** 2
     $                         + u(3,ex1,j,k) ** 2
     $                         + u(4,ex1,j,k) ** 2 )
     $                        / u(1,ex1,j,k) )
c
         end do
c
      end do
c
      return
      end
c
c
c
c
      subroutine cpphix(phi, data, jjj, kkk)
c
c***compute the surface integral
c
      include 'appsp.incl'
c
      real *8 phi(isiz2,isiz3), data(ysize,zsize)
c
      do k = 1, zsize-4
c
         do j = 1, ysize-4
c
            phi(j+jjj,k+kkk) = data(j+2,k+2)
c
         end do
c
      end do
c
      return
      end
c
```



```
c
c
c
      subroutine cpphiy(phi, data, iii, kkk)
c
c***compute the surface integral
c
      include 'appsp.incl'
c
      real *8 phi(isiz1,isiz3), data(xsize,zsize)
c
      do k = 1, zsize-4
c
         do i = 1, xsize-4
c
            phi(i+iii,k+kkk) = data(i+2,k+2)
c
         end do
c
      end do
c
      return
      end
c
c
c
c
      subroutine cpphiz(phi, data, iii, jjj)
c
c***compute the surface integral
c
      include 'appsp.incl'
c
      real *8 phi(isiz1,isiz2), data(xsize,ysize)
c
      do j = 1, ysize-4
c
         do i = 1, xsize-4
c
            phi(i+iii,j+jjj) = data(i+2,j+2)
c
         end do
c
      end do
c
      return
      end
c
c
c
c
      subroutine gphilz(philz)
c
c***compute the global surface integral
c
      include 'appsp.incl'
```

```
c
      real *8 phi1z(isiz1,isiz2)
c
      do j = ji1, ji2-1
c
         do i = ii1, ii2-1
c
            frc1 = frc1 + (  phi1z(i,j)
     $                     + phi1z(i+1,j)
     $                     + phi1z(i,j+1)
     $                     + phi1z(i+1,j+1) )
c
         end do
c
      end do
c
      return
      end
c
c
c
      subroutine gphi2z(phi2z)
c
c***compute the global surface integral
c
      include 'appsp.incl'
c
      real *8 phi2z(isiz1,isiz2)
c
      do j = ji1, ji2-1
c
         do i = ii1, ii2-1
c
            frc1 = frc1 + (  phi2z(i,j)
     $                     + phi2z(i+1,j)
     $                     + phi2z(i,j+1)
     $                     + phi2z(i+1,j+1) )
c
         end do
c
      end do
c
      return
      end
c
c
c
c
      subroutine gphi1y(phi1y)
c
c***compute the global surface integral
c
      include 'appsp.incl'
c
      real *8 phi1y(isiz1,isiz3)
c
```



```
      do k = ki1, ki2-1
c
         do i = ii1, ii2-1
c
            frc2 = frc2 + (  phi1y(i,k)
     $                     + phi1y(i+1,k)
     $                     + phi1y(i,k+1)
     $                     + phi1y(i+1,k+1) )
c
         end do
c
      end do
c
      return
      end
c
c
c
      subroutine gphi2y(phi2y)
c
c***compute the global surface integral
c
      include 'appsp.incl'
c
      real *8 phi2y(isiz1,isiz3)
c
      do k = ki1, ki2-1
c
         do i = ii1, ii2-1
c
            frc2 = frc2 + (  phi2y(i,k)
     $                     + phi2y(i+1,k)
     $                     + phi2y(i,k+1)
     $                     + phi2y(i+1,k+1) )
c
         end do
c
      end do
c
      return
      end
c
c
c
c
      subroutine gphi1x(phi1x)
c
c***compute the global surface integral
c
      include 'appsp.incl'
c
      real *8 phi1x(isiz2,isiz3)
c
      do k = ki1, ki2-1
c
         do j = ji1, ji2-1
```

```
c
            frc3 = frc3 + (  phi1x(j,k)
     $                     + phi1x(j+1,k)
     $                     + phi1x(j,k+1)
     $                     + phi1x(j+1,k+1) )
c
         end do
c
      end do
c
      return
      end
c
c
c
      subroutine gphi2x(phi2x)
c
c***compute the global surface integral
c
      include 'appsp.incl'
c
      real *8 phi2x(isiz2,isiz3)
c
      do k = ki1, ki2-1
c
         do j = ji1, ji2-1
c
            frc3 = frc3 + (  phi2x(j,k)
     $                     + phi2x(j+1,k)
     $                     + phi2x(j,k+1)
     $                     + phi2x(j+1,k+1) )
c
         end do
c
      end do
c
      return
      end
c
c
c
      subroutine gintgr()
c
c***compute the global surface integral
c
      include 'appsp.incl'
c
      frc1 = dxi * deta * frc1
c
      frc2 = dxi * dzeta * frc2
c
      frc3 = deta * dzeta * frc3
c
      frc = 0.25d+00 * ( frc1 + frc2 + frc3 )
c
c     write (iout,1001) frc
```



```
c
      return
c
 1001 format (//5x,'surface integral = ',1pe12.5//)
c
      end
c
c
c
c
c
      subroutine pinvr(rsd)
c
c***block-diagonal matrix-vector multiply
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 rsd(5,xsize,ysize,zsize)
c
      bt = sqrt ( 0.50d+00 )
c
      do k = sz1, ez1
c
         do j = sy1, ey1
c
            do i = sx1, ex1
c
               r1 = rsd(1,i,j,k)
               r2 = rsd(2,i,j,k)
               r3 = rsd(3,i,j,k)
               r4 = rsd(4,i,j,k)
               r5 = rsd(5,i,j,k)
c
               rsd(1,i,j,k) = bt * ( r4 - r5 )
c
               rsd(2,i,j,k) = - r3
c
               rsd(3,i,j,k) = r2
c
               t1 = bt * r1
c
               t2 = 0.50d+00 * ( r4 + r5 )
c
               rsd(4,i,j,k) = - t1 + t2
c
               rsd(5,i,j,k) =   t1 + t2
c
            end do
c
         end do
c
      end do
```

```
c
      return
      end
c
c
c
c
      subroutine pfrct(frct)
c
      include 'appsp.incl'
c
      real *8 frct(5,xsize,ysize,zsize)
c
c     write(iout,*) 'FRCT', frct(2,4,4,8)
c
      return
      end
c
c
      subroutine prsd(rsd)
c
      include 'appsp.incl'
c
      real *8 rsd(5,xsize,ysize,zsize)
c
         do i = sx, ex
            do j = sy, ey
               do k = sz, ez
                  do m = 1,5
                     b = rsd(m,i,j,k)
                     write(iout,*) i,j,k,b
                  end do
               end do
            end do
         end do

      write(iout, *) 'Done writing rsd'

      return
      end
c
c
c
c
c
      subroutine rhsx(u,rsd,frct,igstx, igendx, igsty, igendy, igstz,
     $                igendz)
c
c***compute the right hand sides
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
```



```
      real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize),
     $        frct(5,xsize,ysize,zsize)
c
      dimension flux(5,xsize)
      integer igstx, igendx, igsty, igendy, igstz, igendz
c
      gstx = igstx
      gendx = igendx
      gsty = igsty
      gendy = igendy
      gstz = igstz
      gendz = igendz
c
      do k = sz, ez
         do j = sy, ey
            do i = sx, ex
               do m = 1, 5
                  rsd(m,i,j,k) = - frct(m,i,j,k)
               end do
            end do
         end do
      end do
c
c
c     write(iout,*) 'Rsd', rsd(2,4,4,8)
c
c     call prsd(rsd)
c
c***xi-direction flux differences
c
      do k = sz1, ez1
c
         do j = sy1, ey1
c
            do i = sxm1, exm1
c
               flux(1,i) = u(2,i,j,k)
c
               u21 = u(2,i,j,k) / u(1,i,j,k)
c
               q = 0.50d+00 * (  u(2,i,j,k) * u(2,i,j,k)
     $                         + u(3,i,j,k) * u(3,i,j,k)
     $                         + u(4,i,j,k) * u(4,i,j,k) )
     $                       / u(1,i,j,k)
c
               flux(2,i) = u(2,i,j,k) * u21 + c2 * ( u(5,i,j,k) - q )
c
               flux(3,i) = u(3,i,j,k) * u21
c
               flux(4,i) = u(4,i,j,k) * u21
c
               flux(5,i) = ( c1 * u(5,i,j,k) - c2 * q ) * u21
c
            end do
c
            do i = sx1, ex1
```

```
c
               do m = 1, 5
c
                  rsd(m,i,j,k) =  rsd(m,i,j,k)
     $                       - tx2 * ( flux(m,i+1) - flux(m,i-1) )
c
               end do
c
            end do
c
            do i = sx1, exm1
c
               tmp = 1.0d+00 / u(1,i,j,k)
c
               u21i = tmp * u(2,i,j,k)
               u31i = tmp * u(3,i,j,k)
               u41i = tmp * u(4,i,j,k)
               u51i = tmp * u(5,i,j,k)
c
               tmp = 1.0d+00 / u(1,i-1,j,k)
c
               u21im1 = tmp * u(2,i-1,j,k)
               u31im1 = tmp * u(3,i-1,j,k)
               u41im1 = tmp * u(4,i-1,j,k)
               u51im1 = tmp * u(5,i-1,j,k)
c
               flux(2,i) = (4.0d+00/3.0d+00) * tx3 * ( u21i - u21im1 )
               flux(3,i) = tx3 * ( u31i - u31im1 )
               flux(4,i) = tx3 * ( u41i - u41im1 )
               flux(5,i) = 0.50d+00 * ( 1.0d+00 - c1*c5 )
     $              * tx3 * ( ( u21i  **2 + u31i  **2 + u41i  **2 )
     $                      - ( u21im1**2 + u31im1**2 + u41im1**2 ) )
     $              + (1.0d+00/6.0d+00)
     $              * tx3 * ( u21i**2 - u21im1**2 )
     $              + c1 * c5 * tx3 * ( u51i - u51im1 )
c
            end do
c
            do i = sx1, ex1
c
               rsd(1,i,j,k) = rsd(1,i,j,k)
     $              + dx1 * tx1 * (           u(1,i-1,j,k)
     $                            - 2.0d+00 * u(1,i,j,k)
     $                            +           u(1,i+1,j,k) )
c
               rsd(2,i,j,k) = rsd(2,i,j,k)
     $              + tx3 * c3 * c4 * ( flux(2,i+1) - flux(2,i) )
     $              + dx2 * tx1 * (           u(2,i-1,j,k)
     $                            - 2.0d+00 * u(2,i,j,k)
     $                            +           u(2,i+1,j,k) )
c
               rsd(3,i,j,k) = rsd(3,i,j,k)
     $              + tx3 * c3 * c4 * ( flux(3,i+1) - flux(3,i) )
     $              + dx3 * tx1 * (           u(3,i-1,j,k)
     $                            - 2.0d+00 * u(3,i,j,k)
     $                            +           u(3,i+1,j,k) )
```



```
c
               rsd(4,i,j,k) = rsd(4,i,j,k)
     $              + tx3 * c3 * c4 * ( flux(4,i+1) - flux(4,i) )
     $              + dx4 * tx1 * (           u(4,i-1,j,k)
     $                            - 2.0d+00 * u(4,i,j,k)
     $                            +           u(4,i+1,j,k) )
c
               rsd(5,i,j,k) = rsd(5,i,j,k)
     $              + tx3 * c3 * c4 * ( flux(5,i+1) - flux(5,i) )
     $              + dx5 * tx1 * (           u(5,i-1,j,k)
     $                            - 2.0d+00 * u(5,i,j,k)
     $                            +           u(5,i+1,j,k) )
c
            end do
c
      write(iout,*) 'RSD', rsd(2,4,4,8)
c
c***Fourth-order dissipation
c
            if ( gstx .eq. 1 ) then
            do m = 1, 5
c
               rsd(m,sx1,j,k) = rsd(m,sx1,j,k)
     $           - dssp * ( + 5.0d+00 * u(m,sx1,j,k)
     $                      - 4.0d+00 * u(m,sx2,j,k)
     $                      +           u(m,sx3,j,k) )
c
               rsd(m,sx2,j,k) = rsd(m,sx2,j,k)
     $           - dssp * ( - 4.0d+00 * u(m,sx1,j,k)
     $                      + 6.0d+00 * u(m,sx2,j,k)
     $                      - 4.0d+00 * u(m,sx3,j,k)
     $                      +           u(m,sx+4,j,k) )
c
            end do
            endif
c
            do i = sx3, ex3
c
               do m = 1, 5
c
                  rsd(m,i,j,k) = rsd(m,i,j,k)
     $              - dssp * (           u(m,i-2,j,k)
     $                         - 4.0d+00 * u(m,i-1,j,k)
     $                         + 6.0d+00 * u(m,i,j,k)
     $                         - 4.0d+00 * u(m,i+1,j,k)
     $                         +           u(m,i+2,j,k) )
c
               end do
c
            end do
c
            if ( gendx .eq. nx ) then
            do m = 1, 5
c
               rsd(m,ex2,j,k) = rsd(m,ex2,j,k)
     $           - dssp * (           u(m,ex-4,j,k)
```

```
     $                     - 4.0d+00 * u(m,ex3,j,k)
     $                     + 6.0d+00 * u(m,ex2,j,k)
     $                     - 4.0d+00 * u(m,ex1,j,k)  )
c
            rsd(m,ex1,j,k) = rsd(m,ex1,j,k)
     $        - dssp * (             u(m,ex3,j,k)
     $                     - 4.0d+00 * u(m,ex2,j,k)
     $                     + 5.0d+00 * u(m,ex1,j,k) )
c
          end do
          endif
c
        end do
c
      end do
c
      return
      end
c
c
c
c
c
      subroutine rhsy(u,rsd,frct,igstx, igendx, igsty, igendy, igstz,
     $                igendz)
c
c***compute the right hand sides
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize),
     $        frct(5,xsize,ysize,zsize)
c
      dimension flux(5,ysize)
      integer igstx, igendx, igsty, igendy, igstz, igendz
c
      gstx = igstx
      gendx = igendx
      gsty = igsty
      gendy = igendy
      gstz = igstz
      gendz = igendz
c
c***eta-direction flux differences
c
      do k = sz1, ez1
c
        do i = sx1, ex1
c
          do j = sym1, eym1
c
            flux(1,j) = u(3,i,j,k)
```



```
c
            u31 = u(3,i,j,k) / u(1,i,j,k)
c
            q = 0.50d+00 * (  u(2,i,j,k) * u(2,i,j,k)
     $                      + u(3,i,j,k) * u(3,i,j,k)
     $                      + u(4,i,j,k) * u(4,i,j,k) )
     $                    / u(1,i,j,k)
c
            flux(2,j) = u(2,i,j,k) * u31
c
            flux(3,j) = u(3,i,j,k) * u31 + c2 * ( u(5,i,j,k) - q )
c
            flux(4,j) = u(4,i,j,k) * u31
c
            flux(5,j) = ( c1 * u(5,i,j,k) - c2 * q ) * u31
c
          end do
c
          do j = sy1, ey1
c
            do m = 1, 5
c
              rsd(m,i,j,k) =  rsd(m,i,j,k)
     $                      - ty2 * ( flux(m,j+1) - flux(m,j-1) )
c
            end do
c
          end do
c
          do j = sy1, eym1
c
            tmp = 1.0d+00 / u(1,i,j,k)
c
            u21j = tmp * u(2,i,j,k)
            u31j = tmp * u(3,i,j,k)
            u41j = tmp * u(4,i,j,k)
            u51j = tmp * u(5,i,j,k)
c
            tmp = 1.0d+00 / u(1,i,j-1,k)
c
            u21jm1 = tmp * u(2,i,j-1,k)
            u31jm1 = tmp * u(3,i,j-1,k)
            u41jm1 = tmp * u(4,i,j-1,k)
            u51jm1 = tmp * u(5,i,j-1,k)
c
            flux(2,j) = ty3 * ( u21j - u21jm1 )
            flux(3,j) = (4.0d+00/3.0d+00) * ty3 * ( u31j - u31jm1 )
            flux(4,j) = ty3 * ( u41j - u41jm1 )
            flux(5,j) = 0.50d+00 * ( 1.0d+00 - c1*c5 )
     $           * ty3 * ( ( u21j  **2 + u31j  **2 + u41j  **2 )
     $                   - ( u21jm1**2 + u31jm1**2 + u41jm1**2 ) )
     $           + (1.0d+00/6.0d+00)
     $           * ty3 * ( u31j**2 - u31jm1**2 )
     $           + c1 * c5 * ty3 * ( u51j - u51jm1 )
c
          end do
```

```
c
            do j = sy1, ey1
c
               rsd(1,i,j,k) = rsd(1,i,j,k)
     $              + dy1 * ty1 * (            u(1,i,j-1,k)
     $                             - 2.0d+00 * u(1,i,j,k)
     $                             +           u(1,i,j+1,k) )
c
               rsd(2,i,j,k) = rsd(2,i,j,k)
     $              + ty3 * c3 * c4 * ( flux(2,j+1) - flux(2,j) )
     $              + dy2 * ty1 * (            u(2,i,j-1,k)
     $                             - 2.0d+00 * u(2,i,j,k)
     $                             +           u(2,i,j+1,k) )
c
               rsd(3,i,j,k) = rsd(3,i,j,k)
     $              + ty3 * c3 * c4 * ( flux(3,j+1) - flux(3,j) )
     $              + dy3 * ty1 * (            u(3,i,j-1,k)
     $                             - 2.0d+00 * u(3,i,j,k)
     $                             +           u(3,i,j+1,k) )
c
               rsd(4,i,j,k) = rsd(4,i,j,k)
     $              + ty3 * c3 * c4 * ( flux(4,j+1) - flux(4,j) )
     $              + dy4 * ty1 * (            u(4,i,j-1,k)
     $                             - 2.0d+00 * u(4,i,j,k)
     $                             +           u(4,i,j+1,k) )
c
               rsd(5,i,j,k) = rsd(5,i,j,k)
     $              + ty3 * c3 * c4 * ( flux(5,j+1) - flux(5,j) )
     $              + dy5 * ty1 * (            u(5,i,j-1,k)
     $                             - 2.0d+00 * u(5,i,j,k)
     $                             +           u(5,i,j+1,k) )
c
            end do
c
c***fourth-order dissipation
c
            if ( gsty .eq. 1 ) then
            do m = 1, 5
c
               rsd(m,i,sy1,k) = rsd(m,i,sy1,k)
     $           - dssp * ( + 5.0d+00 * u(m,i,sy1,k)
     $                      - 4.0d+00 * u(m,i,sy2,k)
     $                      +           u(m,i,sy3,k) )
c
               rsd(m,i,sy2,k) = rsd(m,i,sy2,k)
     $           - dssp * ( - 4.0d+00 * u(m,i,sy1,k)
     $                      + 6.0d+00 * u(m,i,sy2,k)
     $                      - 4.0d+00 * u(m,i,sy3,k)
     $                      +           u(m,i,sy+4,k) )
c
            end do
            endif
c
            do j = sy3, ey3
c
               do m = 1, 5
```



```
c
                  rsd(m,i,j,k) = rsd(m,i,j,k)
     $              - dssp * (            u(m,i,j-2,k)
     $                        - 4.0d+00 * u(m,i,j-1,k)
     $                        + 6.0d+00 * u(m,i,j,k)
     $                        - 4.0d+00 * u(m,i,j+1,k)
     $                        +           u(m,i,j+2,k) )
c
               end do
c
            end do
c
            if ( gendy .eq. ny ) then
            do m = 1, 5
c
               rsd(m,i,ey2,k) = rsd(m,i,ey2,k)
     $           - dssp * (             u(m,i,ey-4,k)
     $                    - 4.0d+00 * u(m,i,ey3,k)
     $                    + 6.0d+00 * u(m,i,ey2,k)
     $                    - 4.0d+00 * u(m,i,ey1,k)  )
c
               rsd(m,i,ey1,k) = rsd(m,i,ey1,k)
     $           - dssp * (             u(m,i,ey3,k)
     $                    - 4.0d+00 * u(m,i,ey2,k)
     $                    + 5.0d+00 * u(m,i,ey1,k) )
c
            end do
            endif
c
         end do
c
      end do
c
      return
      end
c
c
c
c
c
      subroutine rhsz(u,rsd,frct,igstx, igendx, igsty, igendy, igstz,
     $                igendz)
c
c***compute the right hand sides
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize),
     $        frct(5,xsize,ysize,zsize)
c
      dimension flux(5,zsize)
      integer igstx, igendx, igsty, igendy, igstz, igendz
```

```
c
      gstx = igstx
      gendx = igendx
      gsty = igsty
      gendy = igendy
      gstz = igstz
      gendz = igendz
c
c***zeta-direction flux differences
c
      do j = sy1, ey1
c
         do i = sx1, ex1
c
            do k = szm1, ezm1
c
               flux(1,k) = u(4,i,j,k)
c
               u41 = u(4,i,j,k) / u(1,i,j,k)
c
               q = 0.50d+00 * (  u(2,i,j,k) * u(2,i,j,k)
     $                         + u(3,i,j,k) * u(3,i,j,k)
     $                         + u(4,i,j,k) * u(4,i,j,k) )
     $                       / u(1,i,j,k)
c
               flux(2,k) = u(2,i,j,k) * u41
c
               flux(3,k) = u(3,i,j,k) * u41
c
               flux(4,k) = u(4,i,j,k) * u41 + c2 * ( u(5,i,j,k) - q )
c
               flux(5,k) = ( c1 * u(5,i,j,k) - c2 * q ) * u41
c
            end do
c
            do k = sz1, ez1
c
               do m = 1, 5
c
                  rsd(m,i,j,k) =  rsd(m,i,j,k)
     $                           - tz2 * ( flux(m,k+1) - flux(m,k-1) )
c
               end do
c
            end do
c
            do k = sz1, ezm1
c
               tmp = 1.0d+00 / u(1,i,j,k)
c
               u21k = tmp * u(2,i,j,k)
               u31k = tmp * u(3,i,j,k)
               u41k = tmp * u(4,i,j,k)
               u51k = tmp * u(5,i,j,k)
c
               tmp = 1.0d+00 / u(1,i,j,k-1)
```



```
c
               u21km1 = tmp * u(2,i,j,k-1)
               u31km1 = tmp * u(3,i,j,k-1)
               u41km1 = tmp * u(4,i,j,k-1)
               u51km1 = tmp * u(5,i,j,k-1)
c
               flux(2,k) = tz3 * ( u21k - u21km1 )
               flux(3,k) = tz3 * ( u31k - u31km1 )
               flux(4,k) = (4.0d+00/3.0d+00) * tz3 * ( u41k - u41km1 )
               flux(5,k) = 0.50d+00 * ( 1.0d+00 - c1*c5 )
     $                   * tz3 * ( ( u21k  **2 + u31k  **2 + u41k  **2 )
     $                           - ( u21km1**2 + u31km1**2 + u41km1**2 ) )
     $                   + (1.0d+00/6.0d+00)
     $                   * tz3 * ( u41k**2 - u41km1**2 )
     $                   + c1 * c5 * tz3 * ( u51k - u51km1 )
c
            end do
c
            do k = sz1, ez1
c
               rsd(1,i,j,k) = rsd(1,i,j,k)
     $                   + dz1 * tz1 * (           u(1,i,j,k-1)
     $                               - 2.0d+00 * u(1,i,j,k)
     $                               +           u(1,i,j,k+1) )
c
               rsd(2,i,j,k) = rsd(2,i,j,k)
     $                   + tz3 * c3 * c4 * ( flux(2,k+1) - flux(2,k) )
     $                   + dz2 * tz1 * (           u(2,i,j,k-1)
     $                               - 2.0d+00 * u(2,i,j,k)
     $                               +           u(2,i,j,k+1) )
c
               rsd(3,i,j,k) = rsd(3,i,j,k)
     $                   + tz3 * c3 * c4 * ( flux(3,k+1) - flux(3,k) )
     $                   + dz3 * tz1 * (           u(3,i,j,k-1)
     $                               - 2.0d+00 * u(3,i,j,k)
     $                               +           u(3,i,j,k+1) )
c
               rsd(4,i,j,k) = rsd(4,i,j,k)
     $                   + tz3 * c3 * c4 * ( flux(4,k+1) - flux(4,k) )
     $                   + dz4 * tz1 * (           u(4,i,j,k-1)
     $                               - 2.0d+00 * u(4,i,j,k)
     $                               +           u(4,i,j,k+1) )
c
               rsd(5,i,j,k) = rsd(5,i,j,k)
     $                   + tz3 * c3 * c4 * ( flux(5,k+1) - flux(5,k) )
     $                   + dz5 * tz1 * (           u(5,i,j,k-1)
     $                               - 2.0d+00 * u(5,i,j,k)
     $                               +           u(5,i,j,k+1) )
c
            end do
c
c***fourth-order dissipation
c
            if ( gstz .eq. 1 ) then
            do m = 1, 5
c
```

```
               rsd(m,i,j,sz1) = rsd(m,i,j,sz1)
     $           - dssp * ( + 5.0d+00 * u(m,i,j,sz1)
     $                      - 4.0d+00 * u(m,i,j,sz2)
     $                      +           u(m,i,j,sz3) )
c
               rsd(m,i,j,sz2) = rsd(m,i,j,sz2)
     $           - dssp * ( - 4.0d+00 * u(m,i,j,sz1)
     $                      + 6.0d+00 * u(m,i,j,sz2)
     $                      - 4.0d+00 * u(m,i,j,sz3)
     $                      +           u(m,i,j,sz+4) )
c
            end do
            endif
c
            do k = sz3, ez3
c
               do m = 1, 5
c
                  rsd(m,i,j,k) = rsd(m,i,j,k)
     $              - dssp * (           u(m,i,j,k-2)
     $                         - 4.0d+00 * u(m,i,j,k-1)
     $                         + 6.0d+00 * u(m,i,j,k)
     $                         - 4.0d+00 * u(m,i,j,k+1)
     $                         +           u(m,i,j,k+2) )
c
               end do
c
            end do
c
            if ( gendz .eq. nz ) then
            do m = 1, 5
c
               rsd(m,i,j,ez2) = rsd(m,i,j,ez2)
     $           - dssp * (             u(m,i,j,ez-4)
     $                      - 4.0d+00 * u(m,i,j,ez3)
     $                      + 6.0d+00 * u(m,i,j,ez2)
     $                      - 4.0d+00 * u(m,i,j,ez1)  )
c
               rsd(m,i,j,ez1) = rsd(m,i,j,ez1)
     $           - dssp * (             u(m,i,j,ez3)
     $                      - 4.0d+00 * u(m,i,j,ez2)
     $                      + 5.0d+00 * u(m,i,j,ez1) )
c
            end do
            endif
c
         end do
c
      end do
c
      return
      end
c
c
c
c
```



```
c
      subroutine spentax ( m, a, b, c, d, e, f )
c
c***solution of multiple, independent systems of penta-diagonal systems
c   using Gaussian elimination (without pivoting) algorithm
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 f(5,xsize,ysize,zsize), a(xsize,ysize,zsize),
     $        b(xsize,ysize,zsize), c(xsize,ysize,zsize),
     $        d(xsize,ysize,zsize), e(xsize,ysize,zsize)
c
c***forward elimination
c
      do k = sz1, ez1
c
         do j = sy1, ey1
c
            if ( gstx .eq. 1 ) then
               tmp1 = b(sx1,j,k) / c(sx,j,k)
c
               c(sx1,j,k) = c(sx1,j,k) - tmp1 * d(sx,j,k)
               d(sx1,j,k) = d(sx1,j,k) - tmp1 * e(sx,j,k)
               f(m,sx1,j,k) = f(m,sx1,j,k) - tmp1 * f(m,sx,j,k)
            endif
c
            do i = sx2, ex
c
               tmp2 = a(i,j,k) / c(i-2,j,k)
c
               b(i,j,k) = b(i,j,k) - tmp2 * d(i-2,j,k)
               c(i,j,k) = c(i,j,k) - tmp2 * e(i-2,j,k)
               f(m,i,j,k) = f(m,i,j,k) - tmp2 * f(m,i-2,j,k)
c
               tmp1 = b(i,j,k) / c(i-1,j,k)
c
               c(i,j,k) = c(i,j,k) - tmp1 * d(i-1,j,k)
               d(i,j,k) = d(i,j,k) - tmp1 * e(i-1,j,k)
               f(m,i,j,k) = f(m,i,j,k) - tmp1 * f(m,i-1,j,k)
c
            end do
c
         end do
c
      end do
c
      return
      end
c
c
c
c
```

```
      subroutine bspentax (m, a, b, c, d, e, f )
c
c***solution of multiple, independent systems of penta-diagonal systems
c   using Gaussian elimination (without pivoting) algorithm
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 f(5,xsize,ysize,zsize), a(xsize,ysize,zsize),
     $        b(xsize,ysize,zsize), c(xsize,ysize,zsize),
     $        d(xsize,ysize,zsize), e(xsize,ysize,zsize)
c
c***back-substitution phase
c
      do k = sz1, ez1
c
         do j = sy1, ey1
c
            if ( gendx .eq. nx ) then
            f(m,ex,j,k) = f(m,ex,j,k) / c(ex,j,k)
            f(m,ex1,j,k) = ( f(m,ex1,j,k) - d(ex1,j,k)*f(m,ex,j,k) )
     $           / c(ex1,j,k)
            endif
c
            do i = ex2, sx, -1
c
               f(m,i,j,k) = ( f(m,i,j,k) - d(i,j,k)*f(m,i+1,j,k)
     $              - e(i,j,k)*f(m,i+2,j,k) ) / c(i,j,k)
c
            end do
c
         end do
c
      end do
c
      return
      end
c
c
c
c
c
      subroutine spentax3 ( a, b, c, d, e, f )
c
c***solution of multiple, independent systems of penta-diagonal systems
c   using Gaussian elimination algorithm
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
```



```
      real *8 f(5,xsize,ysize,zsize), a(xsize,ysize,zsize),
     $        b(xsize,ysize,zsize), c(xsize,ysize,zsize),
     $        d(xsize,ysize,zsize), e(xsize,ysize,zsize)
c
c***forward elimination
c
      do k = sz1, ez1
c
         do j = sy1, ey1
c
            if ( gstx .eq. 1 ) then
               tmp1 = b(sx1,j,k) / c(sx,j,k)
c
               c(sx1,j,k) = c(sx1,j,k) - tmp1 * d(sx,j,k)
               d(sx1,j,k) = d(sx1,j,k) - tmp1 * e(sx,j,k)
               f(1,sx1,j,k) = f(1,sx1,j,k) - tmp1 * f(1,sx,j,k)
               f(2,sx1,j,k) = f(2,sx1,j,k) - tmp1 * f(2,sx,j,k)
               f(3,sx1,j,k) = f(3,sx1,j,k) - tmp1 * f(3,sx,j,k)
            endif
c
            do i = sx2, ex
c
               tmp2 = a(i,j,k) / c(i-2,j,k)
c
               b(i,j,k) = b(i,j,k) - tmp2 * d(i-2,j,k)
               c(i,j,k) = c(i,j,k) - tmp2 * e(i-2,j,k)
               f(1,i,j,k) = f(1,i,j,k) - tmp2 * f(1,i-2,j,k)
               f(2,i,j,k) = f(2,i,j,k) - tmp2 * f(2,i-2,j,k)
               f(3,i,j,k) = f(3,i,j,k) - tmp2 * f(3,i-2,j,k)
c
               tmp1 = b(i,j,k) / c(i-1,j,k)
c
               c(i,j,k) = c(i,j,k) - tmp1 * d(i-1,j,k)
               d(i,j,k) = d(i,j,k) - tmp1 * e(i-1,j,k)
               f(1,i,j,k) = f(1,i,j,k) - tmp1 * f(1,i-1,j,k)
               f(2,i,j,k) = f(2,i,j,k) - tmp1 * f(2,i-1,j,k)
               f(3,i,j,k) = f(3,i,j,k) - tmp1 * f(3,i-1,j,k)
c
            end do
c
         end do
c
      end do
c
      return
      end
c
c
c
c
      subroutine bspentax3 ( a, b, c, d, e, f )
c
c***solution of multiple, independent systems of penta-diagonal systems
c   using Gaussian elimination algorithm
c
c Author: Sisira Weeratunga
```

```
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 f(5,xsize,ysize,zsize), a(xsize,ysize,zsize),
     $        b(xsize,ysize,zsize), c(xsize,ysize,zsize),
     $        d(xsize,ysize,zsize), e(xsize,ysize,zsize)
c
c***back-substitution phase
c
      do k = sz1, ez1
c
         do j = sy1, ey1
c
            if ( gendx .eq. nx ) then
            f(1,ex,j,k) = f(1,ex,j,k) / c(ex,j,k)
            f(1,ex-1,j,k) = ( f(1,ex-1,j,k) - d(ex-1,j,k)*f(1,ex,j,k) )
     $           / c(ex-1,j,k)
c
            f(2,ex,j,k) = f(2,ex,j,k) / c(ex,j,k)
            f(2,ex-1,j,k) = ( f(2,ex-1,j,k) - d(ex-1,j,k)*f(2,ex,j,k) )
     $           / c(ex-1,j,k)
c
            f(3,ex,j,k) = f(3,ex,j,k) / c(ex,j,k)
            f(3,ex-1,j,k) = ( f(3,ex-1,j,k) - d(ex-1,j,k)*f(3,ex,j,k) )
     $           / c(ex-1,j,k)
            endif
c
            do i = ex2, sx, -1
c
               f(1,i,j,k) = ( f(1,i,j,k) - d(i,j,k)*f(1,i+1,j,k)
     $              - e(i,j,k)*f(1,i+2,j,k) ) / c(i,j,k)
c
               f(2,i,j,k) = ( f(2,i,j,k) - d(i,j,k)*f(2,i+1,j,k)
     $              - e(i,j,k)*f(2,i+2,j,k) ) / c(i,j,k)
c
               f(3,i,j,k) = ( f(3,i,j,k) - d(i,j,k)*f(3,i+1,j,k)
     $              - e(i,j,k)*f(3,i+2,j,k) ) / c(i,j,k)
c
            end do
c
         end do
c
      end do
c
      return
      end
c
c
c
c
c
      subroutine spentay ( m, a, b, c, d, e, f )
c
c***solution of multiple, independent systems of penta-diagonal systems
```



```
c   using Gaussian elimination algorithm
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 f(5,xsize,ysize,zsize), a(xsize,ysize,zsize),
     $        b(xsize,ysize,zsize), c(xsize,ysize,zsize),
     $        d(xsize,ysize,zsize), e(xsize,ysize,zsize)
c
c***forward elimination
c
      do k = sz1, ez1
c
         do i = sx1, ex1
c
            if ( gsty .eq. 1 ) then
               tmp1 = b(i,sy1,k) / c(i,sy,k)
c
               c(i,sy1,k) = c(i,sy1,k) - tmp1 * d(i,sy,k)
               d(i,sy1,k) = d(i,sy1,k) - tmp1 * e(i,sy,k)
               f(m,i,sy1,k) = f(m,i,sy1,k) - tmp1 * f(m,i,sy,k)
            endif
c
            do j = sy2, ey
c
               tmp2 = a(i,j,k) / c(i,j-2,k)
c
               b(i,j,k) = b(i,j,k) - tmp2 * d(i,j-2,k)
               c(i,j,k) = c(i,j,k) - tmp2 * e(i,j-2,k)
               f(m,i,j,k) = f(m,i,j,k) - tmp2 * f(m,i,j-2,k)
c
               tmp1 = b(i,j,k) / c(i,j-1,k)
c
               c(i,j,k) = c(i,j,k) - tmp1 * d(i,j-1,k)
               d(i,j,k) = d(i,j,k) - tmp1 * e(i,j-1,k)
               f(m,i,j,k) = f(m,i,j,k) - tmp1 * f(m,i,j-1,k)
c
            end do
c
         end do
c
      end do
c
      return
      end
c
c
c
c
      subroutine bspentay ( m, a, b, c, d, e, f )
c
c***solution of multiple, independent systems of penta-diagonal systems
c   using Gaussian elimination algorithm
```

```
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 f(5,xsize,ysize,zsize), a(xsize,ysize,zsize),
     $        b(xsize,ysize,zsize), c(xsize,ysize,zsize),
     $        d(xsize,ysize,zsize), e(xsize,ysize,zsize)
c
c***back-substitution phase
c
      do k = sz1, ez1
c
         do i = sx1, ex1
c
            if ( gendy .eq. ny ) then
            f(m,i,ey,k) = f(m,i,ey,k) / c(i,ey,k)
            f(m,i,ey-1,k) = ( f(m,i,ey-1,k) - d(i,ey-1,k)*f(m,i,ey,k) )
     $           / c(i,ey-1,k)
            endif
c
            do j = ey2, sy, -1
c
               f(m,i,j,k) = ( f(m,i,j,k) - d(i,j,k)*f(m,i,j+1,k)
     $              - e(i,j,k)*f(m,i,j+2,k) ) / c(i,j,k)
c
            end do
c
         end do
c
      end do
c
      return
      end
c
c
c
c
c
      subroutine spentay3 ( a, b, c, d, e, f )
c
c***solution of multiple, independent systems of penta-diagonal systems
c   using Gaussian elimination algorithm
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 f(5,xsize,ysize,zsize), a(xsize,ysize,zsize),
     $        b(xsize,ysize,zsize), c(xsize,ysize,zsize),
     $        d(xsize,ysize,zsize), e(xsize,ysize,zsize)
c
```



```
c***forward elimination
c
      do k = sz1, ez1
c
         do i = sx1, ex1
c
            if ( gsty .eq. 1 ) then
               tmp1 = b(i,sy1,k) / c(i,sy,k)
c
               c(i,sy1,k) = c(i,sy1,k) - tmp1 * d(i,sy,k)
               d(i,sy1,k) = d(i,sy1,k) - tmp1 * e(i,sy,k)
               f(1,i,sy1,k) = f(1,i,sy1,k) - tmp1 * f(1,i,sy,k)
               f(2,i,sy1,k) = f(2,i,sy1,k) - tmp1 * f(2,i,sy,k)
               f(3,i,sy1,k) = f(3,i,sy1,k) - tmp1 * f(3,i,sy,k)
            endif
c
            do j = sy2, ey
c
               tmp2 = a(i,j,k) / c(i,j-2,k)
c
               b(i,j,k) = b(i,j,k) - tmp2 * d(i,j-2,k)
               c(i,j,k) = c(i,j,k) - tmp2 * e(i,j-2,k)
               f(1,i,j,k) = f(1,i,j,k) - tmp2 * f(1,i,j-2,k)
               f(2,i,j,k) = f(2,i,j,k) - tmp2 * f(2,i,j-2,k)
               f(3,i,j,k) = f(3,i,j,k) - tmp2 * f(3,i,j-2,k)
c
               tmp1 = b(i,j,k) / c(i,j-1,k)
c
               c(i,j,k) = c(i,j,k) - tmp1 * d(i,j-1,k)
               d(i,j,k) = d(i,j,k) - tmp1 * e(i,j-1,k)
               f(1,i,j,k) = f(1,i,j,k) - tmp1 * f(1,i,j-1,k)
               f(2,i,j,k) = f(2,i,j,k) - tmp1 * f(2,i,j-1,k)
               f(3,i,j,k) = f(3,i,j,k) - tmp1 * f(3,i,j-1,k)
c
            end do
c
         end do
c
      end do
c
      return
      end
c
c
c
c
      subroutine bspentay3 ( a, b, c, d, e, f )
c
c***solution of multiple, independent systems of penta-diagonal systems
c   using Gaussian elimination algorithm
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
```

```
c
      real *8 f(5,xsize,ysize,zsize), a(xsize,ysize,zsize),
     $        b(xsize,ysize,zsize), c(xsize,ysize,zsize),
     $        d(xsize,ysize,zsize), e(xsize,ysize,zsize)
c
c***back-substitution phase
c
      do k = sz1, ez1
c
         do i = sx1, ex1
c
            if ( gendy .eq. ny ) then
            f(1,i,ey,k) = f(1,i,ey,k) / c(i,ey,k)
            f(1,i,ey-1,k) = ( f(1,i,ey-1,k) - d(i,ey-1,k)*f(1,i,ey,k) )
     $           / c(i,ey-1,k)
c
            f(2,i,ey,k) = f(2,i,ey,k) / c(i,ey,k)
            f(2,i,ey-1,k) = ( f(2,i,ey-1,k) - d(i,ey-1,k)*f(2,i,ey,k) )
     $           / c(i,ey-1,k)
c
            f(3,i,ey,k) = f(3,i,ey,k) / c(i,ey,k)
            f(3,i,ey-1,k) = ( f(3,i,ey-1,k) - d(i,ey-1,k)*f(3,i,ey,k) )
     $           / c(i,ey-1,k)
            endif
c
            do j = ey2, sy, -1
c
               f(1,i,j,k) = ( f(1,i,j,k) - d(i,j,k)*f(1,i,j+1,k)
     $              - e(i,j,k)*f(1,i,j+2,k) ) / c(i,j,k)
c
               f(2,i,j,k) = ( f(2,i,j,k) - d(i,j,k)*f(2,i,j+1,k)
     $              - e(i,j,k)*f(2,i,j+2,k) ) / c(i,j,k)
c
               f(3,i,j,k) = ( f(3,i,j,k) - d(i,j,k)*f(3,i,j+1,k)
     $              - e(i,j,k)*f(3,i,j+2,k) ) / c(i,j,k)
c
            end do
c
         end do
c
      end do
c
      return
      end
c
c
c
c
c
      subroutine spentaz ( m, a, b, c, d, e, f )
c
c***solution of multiple, independent systems of penta-diagonal systems
c   using Gaussian elimination algorithm
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
```



```
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 f(5,xsize,ysize,zsize), a(xsize,ysize,zsize),
     $        b(xsize,ysize,zsize), c(xsize,ysize,zsize),
     $        d(xsize,ysize,zsize), e(xsize,ysize,zsize)
c
c***forward elimination
c
      do j = sy1, ey1
c
         do i = sx1, ex1
c
            if ( gstz .eq. 1 ) then
               tmp1 = b(i,j,sz1) / c(i,j,sz)
c
               c(i,j,sz1) = c(i,j,sz1) - tmp1 * d(i,j,sz)
               d(i,j,sz1) = d(i,j,sz1) - tmp1 * e(i,j,sz)
               f(m,i,j,sz1) = f(m,i,j,sz1) - tmp1 * f(m,i,j,sz)
            endif
c
            do k = sz2, ez
c
               tmp2 = a(i,j,k) / c(i,j,k-2)
c
               b(i,j,k) = b(i,j,k) - tmp2 * d(i,j,k-2)
               c(i,j,k) = c(i,j,k) - tmp2 * e(i,j,k-2)
               f(m,i,j,k) = f(m,i,j,k) - tmp2 * f(m,i,j,k-2)
c
               tmp1 = b(i,j,k) / c(i,j,k-1)
c
               c(i,j,k) = c(i,j,k) - tmp1 * d(i,j,k-1)
               d(i,j,k) = d(i,j,k) - tmp1 * e(i,j,k-1)
               f(m,i,j,k) = f(m,i,j,k) - tmp1 * f(m,i,j,k-1)
c
            end do
c
         end do
c
      end do
c
      return
      end
c
c
c
c
      subroutine bspentaz ( m, a, b, c, d, e, f )
c
c***solution of multiple, independent systems of penta-diagonal systems
c   using Gaussian elimination algorithm
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
```

```
c
      include 'appsp.incl'
c
      real *8 f(5,xsize,ysize,zsize), a(xsize,ysize,zsize),
     $        b(xsize,ysize,zsize), c(xsize,ysize,zsize),
     $        d(xsize,ysize,zsize), e(xsize,ysize,zsize)
c
c***back-substitution phase
c
      do j = sy1, ey1
c
         do i = sx1, ex1
c
            if ( gendz .eq. nz ) then
            f(m,i,j,ez) = f(m,i,j,ez) / c(i,j,ez)
            f(m,i,j,ez-1) = ( f(m,i,j,ez-1) - d(i,j,ez-1)*f(m,i,j,ez) )
     $           / c(i,j,ez-1)
            endif
c
            do k = ez2, sz, -1
c
               f(m,i,j,k) = ( f(m,i,j,k) - d(i,j,k)*f(m,i,j,k+1)
     $              - e(i,j,k)*f(m,i,j,k+2) ) / c(i,j,k)
c
            end do
c
         end do
c
      end do
c
      return
      end
c
c
c
c
c
      subroutine spentaz3 ( a, b, c, d, e, f )
c
c***solution of multiple, independent systems of penta-diagonal systems
c   using Gaussian elimination algorithm
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 f(5,xsize,ysize,zsize), a(xsize,ysize,zsize),
     $        b(xsize,ysize,zsize), c(xsize,ysize,zsize),
     $        d(xsize,ysize,zsize), e(xsize,ysize,zsize)
c
c***forward elimination
c
c EVERYTHINGS OK HERE
      do j = sy1, ey1
```



```
c
         do i = sx1, ex1
c
            if ( gstz .eq. 1 ) then
               tmp1 = b(i,j,sz1) / c(i,j,sz)
c
               c(i,j,sz1) = c(i,j,sz1) - tmp1 * d(i,j,sz)
               d(i,j,sz1) = d(i,j,sz1) - tmp1 * e(i,j,sz)
               f(1,i,j,sz1) = f(1,i,j,sz1) - tmp1 * f(1,i,j,sz)
               f(2,i,j,sz1) = f(2,i,j,sz1) - tmp1 * f(2,i,j,sz)
               f(3,i,j,sz1) = f(3,i,j,sz1) - tmp1 * f(3,i,j,sz)
            endif
c
            do k = sz2, ez
c
               tmp2 = a(i,j,k) / c(i,j,k-2)
c
               b(i,j,k) = b(i,j,k) - tmp2 * d(i,j,k-2)
               c(i,j,k) = c(i,j,k) - tmp2 * e(i,j,k-2)
               f(1,i,j,k) = f(1,i,j,k) - tmp2 * f(1,i,j,k-2)
               f(2,i,j,k) = f(2,i,j,k) - tmp2 * f(2,i,j,k-2)
               f(3,i,j,k) = f(3,i,j,k) - tmp2 * f(3,i,j,k-2)
c
               tmp1 = b(i,j,k) / c(i,j,k-1)
c
               c(i,j,k) = c(i,j,k) - tmp1 * d(i,j,k-1)
               d(i,j,k) = d(i,j,k) - tmp1 * e(i,j,k-1)
               f(1,i,j,k) = f(1,i,j,k) - tmp1 * f(1,i,j,k-1)
               f(2,i,j,k) = f(2,i,j,k) - tmp1 * f(2,i,j,k-1)
               f(3,i,j,k) = f(3,i,j,k) - tmp1 * f(3,i,j,k-1)
c
            end do
c
         end do
c
      end do
c
c EVERYTHINGS OK HERE
      return
      end
c
c
c
c
      subroutine bspentaz3 ( a, b, c, d, e, f )
c
c***solution of multiple, independent systems of penta-diagonal systems
c   using Gaussian elimination algorithm
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 f(5,xsize,ysize,zsize), a(xsize,ysize,zsize),
```

```
     $        b(xsize,ysize,zsize), c(xsize,ysize,zsize),
     $        d(xsize,ysize,zsize), e(xsize,ysize,zsize)
c
c***back-substitution phase
c
      do j = sy1, ey1
c
         do i = sx1, ex1
c
            if ( gendz .eq. nz ) then
            f(1,i,j,ez) = f(1,i,j,ez) / c(i,j,ez)
            f(1,i,j,ez-1) = ( f(1,i,j,ez-1) - d(i,j,ez-1)*f(1,i,j,ez) )
     $           / c(i,j,ez-1)
c
            f(2,i,j,ez) = f(2,i,j,ez) / c(i,j,ez)
            f(2,i,j,ez-1) = ( f(2,i,j,ez-1) - d(i,j,ez-1)*f(2,i,j,ez) )
     $           / c(i,j,ez-1)
c
            f(3,i,j,ez) = f(3,i,j,ez) / c(i,j,ez)
            f(3,i,j,ez-1) = ( f(3,i,j,ez-1) - d(i,j,ez-1)*f(3,i,j,ez) )
     $           / c(i,j,ez-1)
            endif
c
            do k = ez2, sz, -1
c
               f(1,i,j,k) = ( f(1,i,j,k) - d(i,j,k)*f(1,i,j,k+1)
     $              - e(i,j,k)*f(1,i,j,k+2) ) / c(i,j,k)
c
               f(2,i,j,k) = ( f(2,i,j,k) - d(i,j,k)*f(2,i,j,k+1)
     $              - e(i,j,k)*f(2,i,j,k+2) ) / c(i,j,k)
c
               f(3,i,j,k) = ( f(3,i,j,k) - d(i,j,k)*f(3,i,j,k+1)
     $              - e(i,j,k)*f(3,i,j,k+2) ) / c(i,j,k)
c
            end do
c
         end do
c
      end do
c
c     call psval(4,f(1,4,4,4))
c
      return
      end
c
c
c
c
c
      subroutine txinvr(u,rsd)
c
c***block-diagonal matrix-vector multiplication
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
```



```
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize)
c
      bt = sqrt ( 0.50d+00 )
c
      do k = sz1, ez1
c
         do j = sy1, ey1
c
            do i = sx1, ex1
c
               ru1 = 1.0d+00 / u(1,i,j,k)
c
               uu = ru1 * u(2,i,j,k)
               vv = ru1 * u(3,i,j,k)
               ww = ru1 * u(4,i,j,k)
c
               q = 0.50d+00 * (  uu ** 2
     $                         + vv ** 2
     $                         + ww ** 2 )
c
               ac2 = c1 * c2 * ( ru1 * u(5,i,j,k) - q )
c
               if ( ac2 .le. 0.0d+00 ) then
                  write (*,*) 'Speed of sound is zero'
                  stop
               end if
c
               ac = sqrt ( ac2 )
c
               r1 = rsd(1,i,j,k)
               r2 = rsd(2,i,j,k)
               r3 = rsd(3,i,j,k)
               r4 = rsd(4,i,j,k)
               r5 = rsd(5,i,j,k)
c
               t1 = ( c2 / ac2 ) * ( q * r1 - uu * r2
     $                                    - vv * r3
     $                                    - ww * r4
     $                                    +      r5 )
c
               rsd(1,i,j,k) = r1 - t1
c
               rsd(2,i,j,k) = - ru1 * ( ww * r1 - r4 )
c
               rsd(3,i,j,k) =   ru1 * ( vv * r1 - r3 )
c
               t2 = bt * ru1 * ( uu * r1 - r2 )
               t3 = ( bt * ru1 * ac ) * t1
c
               rsd(4,i,j,k) = - t2 + t3
c
               rsd(5,i,j,k) =   t2 + t3
c
```

```
            end do
c
         end do
c
      end do
c
      return
      end
c
c
c
c
c
      subroutine tzetar(u,rsd)
c
c***block-diagonal matrix-vector multipication
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize)
c
      bt = sqrt ( 0.50d+00 )
c
      do k = sz1, ez1
c
         do j = sy1, ey1
c
            do i = sx1, ex1
c
               ru1 = 1.0d+00 / u(1,i,j,k)
c
               uu = ru1 * u(2,i,j,k)
               vv = ru1 * u(3,i,j,k)
               ww = ru1 * u(4,i,j,k)
c
               q = 0.50d+00 * (  uu ** 2
     $                         + vv ** 2
     $                         + ww ** 2 )
c
               ac2 = c1 * c2 * ( ru1 * u(5,i,j,k) - q )
c
               ac = sqrt ( ac2 )
c
               alph = ( bt * u(1,i,j,k) ) / ac
c
               r1 = rsd(1,i,j,k)
               r2 = rsd(2,i,j,k)
               r3 = rsd(3,i,j,k)
               r4 = rsd(4,i,j,k)
               r5 = rsd(5,i,j,k)
c
               t1 = alph * ( r4 + r5 )
```



```
               t2 = r3 + t1
               t3 = bt * u(1,i,j,k) * ( r4 - r5 )
c
               rsd(1,i,j,k) = t2
c
               rsd(2,i,j,k) = - u(1,i,j,k) * r2 + uu * t2
c
               rsd(3,i,j,k) =   u(1,i,j,k) * r1 + vv * t2
c
               rsd(4,i,j,k) =   ww * t2 + t3
c
               rsd(5,i,j,k) =  u(1,i,j,k) * ( - uu * r2 + vv * r1 )
     $                       + q * t2
     $                       + ( ac2 / c2 ) * t1
     $                       + ww * t3
c
            end do
c
         end do
c
      end do
c
      return
      end
c
c
c
c
c
c
c
c
      subroutine callv (finalrsdnm,finalerror)
c
      include 'appsp.incl'
c
c***verification test
c
      real *8 finalrsdnm(*), finalerror(*)
c
      call verify ( finalrsdnm, finalerror, frc )
c
c***print the CPU time
c
c     write (iout,1001) ttotal
 1001 format (//5x,'Total CPU time = ',1pe12.4,' Sec. ')
c
      return
      end
c
c
c
c
c
c
      subroutine verify ( xcr, xce, xci )
```

```
c
c***verification routine
c
c Author: Sisira Weeratunga
c         NASA Ames Research Center
c         (10/25/90)
c
      include 'appsp.incl'
c
      dimension xcr(5), xce(5),
     $          xrr(5), xre(5)
c
c***tolerance level
c
      epsilon = 1.0e-08
c
      if ( ( nx .eq. 12 ) .and.
     $     ( ny .eq. 12 ) .and.
     $     ( nz .eq. 12 ) ) then
c
c***Reference values of RMS-norms of residual, for the (12X12X12) grid,
c   after 100 time steps, with  DT = 1.50e-02
c
         xrr(1) = 2.7470315451339479d-02
         xrr(2) = 1.0360746705285417d-02
         xrr(3) = 1.6235745065095532d-02
         xrr(4) = 1.5840557224455615d-02
         xrr(5) = 3.4849040609362460d-02
c
c***Reference values of RMS-norms of solution error, for the (12X12X12) grid,
c   after 100 time steps, with  DT = 1.50e-02
c
         xre(1) = 2.7289258557377227d-05
         xre(2) = 1.0364446640837285d-05
         xre(3) = 1.6154798287166471d-05
         xre(4) = 1.5750704994480102d-05
         xre(5) = 3.4177666183390531d-05
c
c***Reference value of surface integral, for the (12X12X12) grid,
c   after 100 time steps, with  DT = 1.50e-02
c
         xri = 7.8406293309126962e+00
c
c***verification test for residuals
c
         do m = 1, 5
c
            tmp = abs ( ( xcr(m) - xrr(m) ) / xrr(m) )
c
            if ( tmp .gt. epsilon ) then
c
               write (iout,1001)
 1001          format(/5x,'VERIFICATION TEST FOR RESIDUALS FAILED')
c
               go to 100
c
```



```
            end if
c
         end do
c
         write (iout,1002)
 1002    format (/5x,'VERIFICATION TEST FOR RESIDUALS ',
     $        'IS SUCCESSFUL')
c
c***verification test for solution error
c
 100     continue
c
         do m = 1, 5
c
            tmp = abs ( ( xce(m) - xre(m) ) / xre(m) )
c
            if ( tmp .gt. epsilon ) then
c
               write (iout,1003)
 1003          format(/5x,'VERIFICATION TEST FOR SOLUTION ',
     $             'ERRORS FAILED')
c
               go to 200
c
            end if
c
         end do
c
         write (iout,1004)
 1004    format (/5x,'VERIFICATION TEST FOR SOLUTION ERRORS ',
     $        'IS SUCCESSFUL')
c
c***verification test for surface integral
c
 200     continue
c
         tmp = abs ( ( xci - xri ) / xri )
c
         if ( tmp .gt. epsilon ) then
c
            write (iout,1005)
 1005       format (/5x,'VERIFICATION TEST FOR SURFACE INTEGRAL FAILED')
c
         else
c
            write (iout,1006)
 1006       format (/5x,'VERIFICATION TEST FOR SURFACE INTEGRAL ',
     $           'IS SUCCESSFUL')
c
         end if
c
         write (iout,1007)
 1007    format(//10x,'CAUTION',
     $ //5x,'REFERENCE VALUES CURRENTLY IN THIS VERIFICATION ',
     $    'ROUTINE ',
     $ /5x,'ARE VALID ONLY FOR RUNS WITH THE FOLLOWING PARAMETER ',
```

```
     $     'VALUES:',
     $ //5x,'NX = 12;  NY = 12;  NZ = 12 ',
     $ //5x,'ITMAX = 100',
     $ //5x,'DT = 1.5e-02',
     $ //5x,'CHANGE IN ANY OF THE ABOVE VALUES RENDER THE REFERENCE ',
     $     'VALUES ',
     $ /5x,'INVALID AND CAUSES A FAILURE OF THE VERIFICATION TEST.')
c
      else if ( ( nx .eq. 64 ) .and.
     $          ( ny .eq. 64 ) .and.
     $          ( nz .eq. 64 ) ) then
c
c***Reference values of RMS-norms of residual, for the (64X64X64) grid,
c   after 400 time steps, with  DT = 1.5e-03
c
         xrr(1) = 2.4799822399300195d+00
         xrr(2) = 1.1276337964368832d+00
         xrr(3) = 1.5028977888770491d+00
         xrr(4) = 1.4217816211695179d+00
         xrr(5) = 2.1292113035138280d+00
c
c***Reference values of RMS-norms of solution error, for the (64X64X64) grid,
c   after 400 time steps, with  DT = 1.5e-03
c
         xre(1) = 1.0900140297820550d-04
         xre(2) = 3.7343951769282091d-05
         xre(3) = 5.0092785406541633d-05
         xre(4) = 4.7671093939528255d-05
         xre(5) = 1.3621613399213001d-04
c
c***Reference value of surface integral, for the (64X64X64) grid,
c   after 400 time steps, with DT = 1.5e-03
c
         xri = 1.2080439685038630e+01
c
c***verification test for residuals
c
         do m = 1, 5
c
            tmp = abs ( ( xcr(m) - xrr(m) ) / xrr(m) )
c
            if ( tmp .gt. epsilon ) then
c
               write (iout,1001)
               go to 400
c
            end if
c
         end do
c
         write (iout,1002)
c
c***verification test for solution error
c
 400     continue
c
```



```
         do m = 1, 5
c
            tmp = abs ( ( xce(m) - xre(m) ) / xre(m) )
c
            if ( tmp .gt. epsilon ) then
c
               write (iout,1003)
               go to 500
c
            end if
c
         end do
c
         write (iout,1004)
c
c***verification test for surface integral
c
 500     continue
c
         tmp = abs ( ( xci - xri ) / xri )
c
         if ( tmp .gt. epsilon ) then
c
            write (iout,1005)
c
         else
c
            write (iout,1006)
c
         end if
c
         write (iout,1008)
 1008    format(//10x,'CAUTION',
     $ //5x,'REFERENCE VALUES CURRENTLY IN THIS VERIFICATION ',
     $     'ROUTINE ',
     $ /5x,'ARE VALID ONLY FOR RUNS WITH THE FOLLOWING PARAMETER ',
     $     'VALUES:',
     $ //5x,'NX = 64;  NY = 64;  NZ = 64 ',
     $ //5x,'ITMAX = 400',
     $ //5x,'DT = 1.5e-03',
     $ //5x,'CHANGE IN ANY OF THE ABOVE VALUES RENDER THE REFERENCE ',
     $     'VALUES ',
     $ /5x,'INVALID AND CAUSES A FAILURE OF THE VERIFICATION TEST.')
c
      else
c
         write (iout,1009)
 1009    format (//1x,'FOR THE PROBLEM PARAMETERS IN USE ',
     $        'NO REFERENCE VALUES ARE PROVIDED'
     $        /1x,'IN THE CURRENT VERIFICATION ROUTINE - ',
     $        'NO VERIFIACTION TEST WAS PERFORMED')
c
      end if
c
      return
      end
```

```
c
c
c
c
c
c
c     function timer(x)
c     real x, timer
c     real secs,tarray(2)
c
c
c     sun version of timing routine
ccccccccccccccccccccccccccccccccccc
c     secs = etime(tarray)
c     timer = secs
ccccccccccccccccccccccccccccccccccc
c
c       cray version of timing routine
ccccccccccccccccccccccccccccccccccc
c      timer = second ()
ccccccccccccccccccccccccccccccccccc
c
c       sgi version of timing routine
ccccccccccccccccccccccccccccccccccc
c      secs = secnds(0.0)
c     timer = secs
ccccccccccccccccccccccccccccccccccc
c     return
c     end
```

```
      implicit real*8 (a-h,o-z)
c
      parameter ( isiz1 = 64, isiz2 = 64, isiz3 = 64 )
      parameter ( xsize = 68, ysize = 68, zsize = 68 )
      parameter ( eachx = 64, eachy = 64, eachz = 64 )
      parameter ( c1 = 1.40d+00, c2 = 0.40d+00,
     $            c3 = 1.00d-01, c4 = 1.00d+00,
     $            c5 = 1.40d+00 )
c     real timer, tstart, tend
      integer ex, ey, ez, ex1, ey1, ez1, ex2, ey2, ez2,
     $                sx, sy, sz,
     $                sx1, sy1, sz1, sx2, sy2, sz2,
     $                sx3, ex3, sy3, ey3, sz3, ez3,
     $                gstx, gendx, gsty, gendy, gstz, gendz,
     $                sxm2, sym2, szm2, exm2, eym2, ezm2,
     $                sxm1, sym1, szm1, exm1, eym1, ezm1
c
c***for parallelization
c
      common/parallel/ ex, ey, ez, ex1, ey1, ez1, ex2, ey2, ez2,
     $                 sx, sy, sz,
     $                 sx1, sy1, sz1, sx2, sy2, sz2,
     $                 sx3, ex3, sy3, ey3, sz3, ez3,
     $                 gstx, gendx, gsty, gendy, gstz, gendz,
     $                 sxm2, sym2, szm2, exm2, eym2, ezm2,
     $                 sxm1, sym1, szm1, exm1, eym1, ezm1
c
c***boundaries for use in setiv - setbv
c
      common/boundaries/ bx1(5,ysize,zsize), bxnx(5,ysize,zsize),
     $                   by1(5,xsize,zsize), byny(5,xsize,zsize),
     $                   bz1(5,xsize,ysize), bznz(5,xsize,ysize)
c
c***grid
c
      common/cgcon/ nx, ny, nz,
     $              ii1, ii2, ji1, ji2, ki1, ki2, idum1,
     $              dxi, deta, dzeta,
     $              tx1, tx2, tx3,
     $              ty1, ty2, ty3,
     $              tz1, tz2, tz3
c
c***dissipation
c
      common/disp/ dx1,dx2,dx3,dx4,dx5,
     $             dy1,dy2,dy3,dy4,dy5,
     $             dz1,dz2,dz3,dz4,dz5,
     $             dssp
c
c***field variables and residuals
c
c     common/cvar/ u(5,isiz1,isiz2,isiz3),
c    $             rsd(5,isiz1,isiz2,isiz3),
c    $             frct(5,isiz1,isiz2,isiz3)
c
c***output control parameters
```



```
c
      common/cprcon/ ipr, iout, inorm
c
c***newton-raphson iteration control parameters
c
      common/ctscon/ itmax, invert,
     $               dt, omega, tolrsd(5),
     $               rsdnm(5), errnm(5), frc, ttotal,
     $               frc1, frc2, frc3
c
c***global jacobian matrix
c
c     common/cjac/ a(isiz1,isiz2,isiz3),
c    $             b(isiz1,isiz2,isiz3),
c    $             c(isiz1,isiz2,isiz3),
c    $             d(isiz1,isiz2,isiz3),
c    $             e(isiz1,isiz2,isiz3)
c
c***coefficients of the exact solution
c
      common/cexact/ ce(5,13)
c
```

```
      implicit real*8 (a-h,o-z)
c
      parameter ( isiz1 = 12, isiz2 = 12, isiz3 = 12 )
      parameter ( xsize = 10, ysize = 10, zsize = 10 )
      parameter ( eachx = 6, eachy = 6, eachz = 6 )
      parameter ( c1 = 1.40d+00, c2 = 0.40d+00,
     $            c3 = 1.00d-01, c4 = 1.00d+00,
     $            c5 = 1.40d+00 )
c     real timer, tstart, tend
      integer ex, ey, ez, ex1, ey1, ez1, ex2, ey2, ez2,
     $                 sx, sy, sz,
     $                 sx1, sy1, sz1, sx2, sy2, sz2,
     $                 sx3, ex3, sy3, ey3, sz3, ez3,
     $                 gstx, gendx, gsty, gendy, gstz, gendz,
     $                 sxm2, sym2, szm2, exm2, eym2, ezm2,
     $                 sxm1, sym1, szm1, exm1, eym1, ezm1
c
c***for parallelization
c
      common/parallel/ ex, ey, ez, ex1, ey1, ez1, ex2, ey2, ez2,
     $                 sx, sy, sz,
     $                 sx1, sy1, sz1, sx2, sy2, sz2,
     $                 sx3, ex3, sy3, ey3, sz3, ez3,
     $                 gstx, gendx, gsty, gendy, gstz, gendz,
     $                 sxm2, sym2, szm2, exm2, eym2, ezm2,
     $                 sxm1, sym1, szm1, exm1, eym1, ezm1
c
c***boundaries for use in setiv - setbv
c
      common/boundaries/ bx1(5,isiz2,isiz3), bxnx(5,isiz2,isiz3),
     $                   by1(5,isiz1,isiz3), byny(5,isiz1,isiz3),
     $                   bz1(5,isiz1,isiz2), bznz(5,isiz1,isiz2)
c
c***grid
c
      common/cgcon/ nx, ny, nz,
     $              ii1, ii2, ji1, ji2, ki1, ki2, idum1,
     $              dxi, deta, dzeta,
     $              tx1, tx2, tx3,
     $              ty1, ty2, ty3,
     $              tz1, tz2, tz3
c
c***dissipation
c
      common/disp/ dx1,dx2,dx3,dx4,dx5,
     $             dy1,dy2,dy3,dy4,dy5,
     $             dz1,dz2,dz3,dz4,dz5,
     $             dssp
c
c***field variables and residuals
c
c     common/cvar/ u(5,isiz1,isiz2,isiz3),
c    $             rsd(5,isiz1,isiz2,isiz3),
c    $             frct(5,isiz1,isiz2,isiz3)
c
c***output control parameters
```



```
c
      common/cprcon/ ipr, iout, inorm
c
c***newton-raphson iteration control parameters
c
      common/ctscon/ itmax, invert,
     $               dt, omega, tolrsd(5),
     $               rsdnm(5), errnm(5), frc, ttotal,
     $               frc1, frc2, frc3
c
c***global jacobian matrix
c
c     common/cjac/ a(isiz1,isiz2,isiz3),
c    $             b(isiz1,isiz2,isiz3),
c    $             c(isiz1,isiz2,isiz3),
c    $             d(isiz1,isiz2,isiz3),
c    $             e(isiz1,isiz2,isiz3)
c
c***coefficients of the exact solution
c
      common/cexact/ ce(5,13)
c
```