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 imple-
menting 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 lan-
guage [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 Tri-
angular (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 ab-
stractions. 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:

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

Chare definitions have the form

```cpp
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)`.

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. 

\texttt{ChareHandle[LOCAL]->DataMember} and \texttt{ChareHandle[LOCAL]->FunctionMember()} are used to access public members of the local branch of a branched chare.

\texttt{ChareHandle[P]=>EF(MsgPointer)} sends a message to the function \texttt{EF} in the branch of a branched chare on the processor \texttt{P}.

\texttt{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

\textbf{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.

\textbf{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 (\textit{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 \textit{pack} and \textit{unpack} in the message class for packing and unpacking messages that are called \textit{by the system} just before sending and after receiving a message, respectively. Thus only messages that are actually sent to other processors are packed.

\textbf{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 heterogeneous 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 computa-
tions 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 proces-
sors 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.

```cpp
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.
```cpp
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][] => EntryFunction(msgptr) ;          // multicast to column
arrayid[][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:

```c
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

```c
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.

```c
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 (hare. 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:

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

//cubearray = CreateArray(Chare(cube), EP(cube, cube), msgptr,
Grid3D, ncy, ncz);

cubearray = CreateArray(_CK_chare_cube, _CK_ep_cube, msgptr,
Grid3D, ncy, ncz);
```

Here, a message of type CreateMessage is being sent to the newly created array of cube (hare at the constructor entry function of each (hare. 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_charenname and the entry functions are translated as _CK_ep_charenname_entryname. ncy, 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 (hare in the array perform a global reduction. This synchronization code needs to be provided by the programmer. Typically, this could be achieved by each (hare sending a message to the main (hare and awaiting a message from the main (hare to trigger computation. The main (hare 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 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 \( s = \sqrt{P} \) and \( 1 \leq i, j, k \leq s \). 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:

```c
int MultiPartitionMapFn(int arrayid, int i, int j, int k) {
    // return processor number owning object (i,j,k)
    return (ZArraySize*(i-k+ZArraySize)%ZArraySize) +
            (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:

```c
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.

<table>
<thead>
<tr>
<th>Processors</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync-Transpose</td>
<td>-</td>
<td>8.98</td>
<td>3.01</td>
<td>1.98</td>
</tr>
<tr>
<td>Async-Transpose</td>
<td>-</td>
<td>7.81</td>
<td>2.54</td>
<td>1.40</td>
</tr>
<tr>
<td>Multipartition</td>
<td>24.63</td>
<td>7.60</td>
<td>1.98</td>
<td>1.00</td>
</tr>
</tbody>
</table>

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.

![Degree of Parallelism in LU](image)

**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 $4 \times 4$ 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.

![Wave-Parallel Placement Strategy](image)

**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.

<table>
<thead>
<tr>
<th>Processors</th>
<th>$4 \times 4 \times 1$</th>
<th>$4 \times 4 \times 4$</th>
<th>$8 \times 8 \times 1$</th>
<th>$8 \times 8 \times 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>370.580</td>
<td>308.072</td>
<td>322.450</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>231.840</td>
<td>151.379</td>
<td>179.884</td>
<td>413.638</td>
</tr>
<tr>
<td>16</td>
<td>210.931</td>
<td>96.273</td>
<td>119.704</td>
<td>81.930</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Processors</th>
<th>$4 \times 4 \times 1$</th>
<th>$4 \times 4 \times 4$</th>
<th>$8 \times 8 \times 1$</th>
<th>$8 \times 8 \times 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>453.809</td>
<td>298.159</td>
<td>300.545</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>430.003</td>
<td>289.487</td>
<td>220.567</td>
<td>343.630</td>
</tr>
<tr>
<td>16</td>
<td>423.968</td>
<td>289.297</td>
<td>219.503</td>
<td>320.542</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Processors</th>
<th>$4 \times 4 \times 1$</th>
<th>$4 \times 4 \times 4$</th>
<th>$8 \times 8 \times 1$</th>
<th>$8 \times 8 \times 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>389.677</td>
<td>267.771</td>
<td>296.191</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>237.368</td>
<td>171.202</td>
<td>182.077</td>
<td>335.665</td>
</tr>
<tr>
<td>16</td>
<td>208.387</td>
<td>131.432</td>
<td>137.044</td>
<td>66.602</td>
</tr>
</tbody>
</table>

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
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.

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

<table>
<thead>
<tr>
<th>Processors</th>
<th>4 x 4 x 1</th>
<th>4 x 4 x 4</th>
<th>8 x 8 x 1</th>
<th>8 x 8 x 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>384.666</td>
<td>287.692</td>
<td>297.734</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>358.427</td>
<td>163.656</td>
<td>294.130</td>
<td>255.330</td>
</tr>
<tr>
<td>16</td>
<td>233.877</td>
<td>108.904</td>
<td>185.241</td>
<td>65.016</td>
</tr>
</tbody>
</table>

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


char cube
{
  // char-local variables declarations

  struct entry 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 {
          bits(); 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


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 "array.h"
#include "cube.h"
#include "controlboc.h"
#include "misc.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 getsn(int *, int *, int *, int *);
extern "C" void initp(int *);
extern "C" void copyf(double *, double *, int *, int *);
extern "C" void copyf(double *, double *, int *, int *);
extern "C" void ghlxf(double *);
extern "C" void ghlxz(double *);
extern "C" void ghlx(double *);
extern "C" void gxv(double *, double *);
extern "C" void gxv(double *, double *);

extern int lswepMap(int gid, int i, int j, int k);
extern int lswepMap(int gid, int i, int j, int k);
extern int lswepMap(int gid, int i, int j, int k);

ControlBoc::ControlBoc(SomeMsg *m)
{
  cubearray = 0;
  /* that is because cubearray is created AFTER
     main to avoid initialization problems */
  myhandle = thisgroup;
  MYPE = CMypEnum();
  NCMPROCS = CMaxPEnum();
  isetp = 1;
  initp(&MYPE);
  getsn(&nx, &ny, &nx, &nx);
}

NumRSDNorms = 0;
for (int i = 0; i < 5; i++)
  TolRSDNorms[i] = 0;

NumCubesDone = 0;
NumProcsDone = 0;

NumPhiX = 0; NumPhiY = 0; NumPhiZ = 0;
NumPhi2X = 0; NumPhi2Y = 0; NumPhi2Z = 0;
NumPhi = 0;

phiX = new double [nx * ny];
phiY = new double [nx * ny];
phiZ = new double [nx * ny];
phi2X = new double [nx * ny];
phi2Y = new double [nx * ny];
phi2Z = new double [nx * ny];

lasttime = 0;
myhandle->InitialSync(m);
}

void ControlBoc::InitialSync(SomeMsg *m)
{
  NumCubesDone++;
  if (NumCubesDone < NumCubes/CNumPens())
    delete m;
  return;
  NumCubesDone = 0;

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

void ControlBoc::InitialSync0(SomeMsg *m)
{
  NumProcsDone++;
  if (NumProcsDone < CNumPens())
    delete m;
  return;
  NumProcsDone = 0;

  /* Start Timer */
  beginTime = CTimer();
}
// Start first iteration */
SendArrayRange(cubarray,Ck.sp.Cube.StartRHS,j, m,-1,-1,-1,-1,-1,-1);
}

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

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

    SendArrayRange(cubarray,Ck.sp.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(cubarray,Ck.sp.Cube.StartRHS,k, m,-1,-1,-1,-1,-1,-1);
}

****** CODE FOR LNNorm **********/

void ControlBoc::GlobalBNorm(NormsMsg *m)
{    
    /* compute global sum and actual norm and print it */
    for ( int i=0; i<5; i++ )
        TotBNSum[i] += aSum[i];
    delete m;
    NumBNSums++;
    if ( NumBNSums < NumCubes ) { return; }
}

for ( i=0; i<5; i++ )
    TotBNSum[i] = sqrt( TotBNSum[i] /
        ( (x^2) * (y^2) * (z^2) ) );

memcpy(LatestBNSum, TotBNSum, sizeof(double)*5);

NumBNSums = 0;
for ( i=0; i<5; i++ )
    TotBNSum[i] = 0.0;

SomeMsg *m2 = new SomeMsg;
SendArrayRange(cubarray,Ck.sp.Cube.StartADI NSWep, m2,-1,-1,-1,-1,-1,-1,-1);
}

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

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

    #ifdef TRANSPOSE
    ArrayRemap(cubarray, JswepMapfs, &(ControlBoc::FinishedJswepRemap),
        thishandle);
    #else
    SendArrayRange(cubarray,Ck.sp.Cube.StartADI NSWep, m,-1,-1,-1,-1,-1,-1,-1);
    #endif
}

void ControlBoc::FinishedJswepRemap(SomeMsg *m)
{    
    SendArrayRange(cubarray,Ck.sp.Cube.StartADI NSWep, m,-1,-1,-1,-1,-1,-1,-1);
}

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

#define TRANSPOSE
ArrayRemap(cubarray, KsweepMapfn, &((ControlBoc::FinishedKsweepRemap),
    thishandle));
#else
    SendArrayRange(cubarray, CK.ep.Cube.StartADI.Ksweep, m. -1, -1, -1, -1, -1);
#endif

void
ControlBoc::FinishedKsweepRemap(SomeMsg *m)
{
    SendArrayRange(cubarray, CK.ep.Cube.StartADI.Ksweep, m. -1, -1, -1, -1, -1);
}

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

#define TRANSPOSE
    ArrayRemap(cubarray, IrwMapfn, &((ControlBoc::FinishedIrwRemap),
        thishandle));
#else
    SendArrayRange(cubarray, CK.ep.Cube.StartRHS.I, m. -1, -1, -1, -1, -1);
#endif

void
ControlBoc::FinishedIrwRemap(SomeMsg *m)
{
    SendArrayRange(cubarray, CK.ep.Cube.StartRHS.I, m. -1, -1, -1, -1, -1);
}

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

void
ControlBoc::GlobalCNorms(NormsMsg *m)
{
    /* compute global sum and error norm and print it */
    for (int i = 0; i < 2; i++)
        ToRSDNorm[i] += m->sum[i];
    NumRSDNorms++;
    delete m;
    if (NumRSDNorms < NumCubes)
        return;
    for (int i = 0; i < 2; i++)
        ToRSDNorm[i] = sqrt(ToRSDNorm[i] /
            (N - 2) * (N - 2) * (N - 2));
    NumRSDNorms = 0;
    for (int i = 0; i < 2; i++)
        ToRSDNorm[i] = 0.0;
    SomeMsg *m2 = new SomeMsg;
    SendArrayRange(cubarray, 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();
    CPInfo("%s time for %d iterations: %d millisecs, begin %d end %d",
        Name, endtime - begintime, begintime, endtime);
    NumCubesDone = 0;
    SomeMsg *m2 = new SomeMsg;
    SendArrayRange(cubarray, 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 < 2; i++)
        ToRSDNorm[i] += m->sum[i];
    delete m;
    NumRSDNorms++;
}
void ControlBoc::ReceivePhi1x(PhiMag *m) {
  /* Store the phi */
  int begin1 = m->myy * (mx/XArraySize);
  int begin2 = m->myy * (mx/ZArraySize);
  cphi1x(phi1x,m->phi,&begin1,&begin2);
  delete m;
  NumPhi1x++;
  if (NumPhi1x < YArraySize*ZArraySize) return;
  gphi1x(phi1x);
  GlobalPingr();
}

void ControlBoc::ReceivePhi2x(PhiMag *m) {
  /* Store the phi */
  int begin1 = m->myy * (mx/XArraySize);
  int begin2 = m->myy * (mx/ZArraySize);
  cphi2x(phi2x,m->phi,&begin1,&begin2);
  delete m;
  NumPhi2x++;
  if (NumPhi2x < YArraySize*ZArraySize) return;
  gphi2x(phi2x);
  GlobalPingr();
}

void ControlBoc::ReceivePhi1y(PhiMag *m) {
  /* Store the phi */
  int begin1 = m->myx * (my/XArraySize);
  int begin2 = m->myx * (my/ZArraySize);
  cphi1y(phi1y,m->phi,&begin1,&begin2);
  delete m;
  NumPhi1y++;
  if (NumPhi1y < XArraySize*ZArraySize) return;
  gphi1y(phi1y);
  GlobalPingr();
}

void ControlBoc::ReceivePhi2y(PhiMag *m) {
  /* Store the phi */
  int begin1 = m->myx * (my/XArraySize);
  int begin2 = m->myx * (my/ZArraySize);
  cphi2y(phi2y,m->phi,&begin1,&begin2);
  delete m;
  NumPhi2y++;
  if (NumPhi2y < XArraySize*ZArraySize) return;
  gphi2y(phi2y);
  GlobalPingr();
}
delete m;
NumPhi1++;
if ( NumPhi1 < XArraySize*YArraySize )
    return;

gphi1z(phi1z);

GlobalPintgr();

} void ControlBloc::ReceivePhi2c(PhiMsg *m)

{ /* Store the phi */
  int begin1 = m->myx * (ax/XArraySize);
  int begin2 = m->myy * (ay/YArraySize);
cppphi2z(phi2z,m->phi,&begin1,&begin2);

delete m;
NumPhi2++;
if ( NumPhi2 < XArraySize*YArraySize )
    return;

gphi2z(phi2z);

GlobalPintgr();

} void ControlBloc::GlobalPintgr()

{ NumPhi++;
if ( NumPhi == 6 ) { 
    CPrintf("[d1] In GlobalPintgr(x*,MYPE):
    gintgr();

callv(LatestRSDNorm,TotRSDNorm);

    CharmExit();
    }
    }
}
message class NormsMsg {
public: double num[5];
};

class ControlBoc : public groupmember {

ControlBoc group myhandle;

int istep;
int itmax;  // number of ADI iterations
int MYPE;
int NUMROCS;
int nx, ny, nz;
int NumRSDNorns;
double TotRSDNorn[5];
double LastRSDNorn[5];
double *phi1x, *phi1z, *phi2y, *phi2z;
int NumPhi1x, NumPhi1z, NumPhi2y, NumPhi2z, NumPhi1z;
int NumPhi1z;
int NumProcDone;
int NumCommDone;
int lasttime;
int begintime;

entry:
ControlBoc(SomeMsg *m);

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

/******** CODE FOR RHS ******/
entry:
void EndRHSSync(SomeMsg *m);
void EndRHSSyncX(SomeMsg *m);
void GlobalRSDNorns(NormsMsg *m);

/******** CODE FOR ADI LOOP ******/
entry:
void EndADLSync(SomeMsg *m);
void EndADLSyncX(SomeMsg *m);

void EndADLSync(SomeMsg *m);
void FinishedUwrsRemap(SomeMsg *m);
void FinishedKsweepRemap(SomeMsg *m);
void FinishedUwrsRemap(SomeMsg *m);

/**** CODE FOR computing the norms of pseudo-time iterations corrections ****/
entry:
void GlobalCNorm(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 GlobalPrint();  
}
extern int JswepMatr(int gid, int i, int j, int k);
extern int JswepMatr(int gid, int i, int j, int k);
extern int JswepMatr(int gid, int i, int j, int k);

void
pval_(double *vnl)
{
    if (CMYPe() == 10)
        CPrint("'Eda Val is: X.12e:x",CMYPe(),*vnl);
}

void
pval_(int *i, double *vnl)
{
    if (CMYPe() == 10)
        CPrint("'Eda X.12e:x",CMYPe(),*vnl);
}

void
Cube::Cube(SomeMsg *em)
{
    int ep = 0;
    gena(cx, cx, cmin, kitmax);
    /* Set all my local variables */
    myx = thix;
    myy = thix;
    myz = thix;

eachx = x/NArraySize;
eachy = y/NArraySize;
eachz = z/NArraySize;

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

    /* Allocate u, v, fsrc in message to prevent copying while migrating */
    int cubizex = (sizeof(Cube) / 8 + 1) * 8; // make double word boundary
    int tosize = cubicize;
    int eacharray = 5 * allocx * allocy * allocz * sizeof(double);
    tosize += eacharray;
daarea = new (totize) PackMsg;
char *ptr = daarea->data + cubizex;
    u = (double *)ptr;
ptr += eacharray;
    rd = (double *)ptr;
ptr += eacharray;
    fsrc = (double *)ptr;
cube.P

/* Allocate space for all local arrays */
a = new double[AllocSize+allocSize];
b = new double[AllocSize+allocSize];
c = new double[AllocSize+allocSize];
d = new double[AllocSize+allocSize];
e = new double[AllocSize+allocSize];

memset(u, 0, allocSize+allocSize+sizeof(double));
memset(t, 0, allocSize+allocSize+sizeof(double));
memset(f, 0, allocSize+allocSize+sizeof(double));
memset(a, 0, allocSize+allocSize+sizeof(double));
memset(b, 0, allocSize+allocSize+sizeof(double));
memset(c, 0, allocSize+allocSize+sizeof(double));
memset(d, 0, allocSize+allocSize+sizeof(double));
memset(e, 0, allocSize+allocSize+sizeof(double));

/* set global indices for my array section */
gtx = eaxX + myX + 1;
gtxy = eaxX + eaxY + 1;
gtxz = eaxX + eaxY + eaxZ + 1;
gtx = eaxX + eaxY + eaxZ + 1;
gtx = &gtax;
igtx = &igtx;
igtxy = &igtxy;
igtxy = &igtxy;
igtxz = &igtxz;
igtxz = &igtxz;

setglobals(&$eaxX, &$eaxY, &$eaxZ, $igtx, $igtxy, $igtxy, $igtxy, $igtxz, $igtxz);
NumBoundaries = 0;

BvArMx();

control[LOCAL] = InitialSync(m);
DoneRHSI = DoneRHSJ = DoneADJ = DoneADJ = 0;
InitializedK = InitializedK = InitializedK = 0;
void PrintArray(char *, double *, int);

void Cube::SetBoundary(Bound cube = m);
{
  if ( m1 == NULL )
    
CPrintf("[E] Error: m == NULL\n\n);
cube P

BoundaryMsg *m1;
setglobals(kszachx, &ksachy, &ksachz, igstx, igstdx, igstry, igstyd, igstdy, igstdz);

switch( whicharrays ){
    case C.ONLY:
        UBoundaryMsg *m = new (ksize) UBoundaryMsg;
        int count=1;
        getb(u, m->u, &type, &count);
        m1 = m;
    }
    break;

case RSD.ONLY:
    RSBoundaryMsg *m = new (ksize) RSBoundaryMsg;
    int count=1;
    getb(rd, m->rd, &type, &count);
    m1 = m;
    break;

case ALL.ARRAYS:
    int sizes[4];
    sizes[0] = sizes[1] = size;
    AllBoundaryMsg *m = new (sizes) AllBoundaryMsg;
    int count=1;
    getb(u, m->u, &type, &count);
    count = 1;
    getb(rd, m->rd, &type, &count);
    count = 1;
    getb(c, m->c, &type, &count);
    count = 1;
    getb(d, m->d, &type, &count);
    m1 = m;
    break;
}

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


void Cube::BvSets()
{
    setglobals(kszachx, &ksachy, &ksachz, igstx, igstdx, igstry, igstyd, igstdy, igstdz);
    setrv.i(u,igstx, igstdx, igstry, igstyd, igstdy, igstdz);
    setrv.i(u,igstry,igstdy);
    erhs.(frct,igstx, igstdx, igstry, igstyd, igstdy, igstdz);
}

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

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

    BoundaryMsg *m = GetBoundary(IPREV.U.ONLY);
    if ( type == ADI )
        SendArray(cubearray, .CK.ap.Cube.DoADILsweep, m, myx-1, myy, myz);
    else
        SendArray(cubearray, .CK.ap.Cube.DoRHS.I, m, myx-1, myy, myz);
}

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

    BoundaryMsg *m = GetBoundary(INEXT.U.ONLY);
    SendArray(cubearray, .CK.ap.Cube.DoRHS.I, m, myx+1, myy, myz);
}

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

    BoundaryMsg *m = GetBoundary(IPREV.U.ONLY);
if ( type == ADI )
SendArray(cubearray, CKEp.Cube.DoADI, Xsweep, m, myx, myy-1, myz);
else
SendArray(cubearray, CKEp.Cube.DoRHS, J, m, myx, myy-1, myz);
}

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

BoundaryMsg *m = GetBoundary(INEXT,U,ONLY);
SendArray(cubearray, CKEp.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, CKEp.Cube.DoADI, Xsweep, m, myx, myy, myz-1);
else
SendArray(cubearray, CKEp.Cube.DoRHS, K, m, myx, myy-1);
}

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

BoundaryMsg *m = GetBoundary(KNEXT,U,ONLY);
SendArray(cubearray, CKEp.Cube.DoRHS, K, m, myx, myy+1);

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

void Cube::StartRHS(J(SomeMsg *m)
{
istep++;
if ( istep > itmax )
{
SomeMsg *m2 = new SomeMsg;
controlMsg[j] = FinalSync(m2);
return;
}

SendPrevRHS();
SendNextJ();
InitializedRHSJ = 1;
if ( XArraySize == 1 )
{ DoRHSJ(NULL);
}

void Cube::DoRHS(J(BoundaryMsg *m)
{
if ( InitializedRHSJ == 0 )
{ SendArray(thisgroup, CKEp.Cube.DoRHS, J, m, myx, myy, myz);
return;
}
if ( NumBoundaries < 2 )
{ SetBoundary(m);
NumBoundaries++;
if ( NumBoundaries < 2 )
return;
}
setglobals(&eachx, &eachy, &eachz, igtx, igdtx, igty, igdty, igtz, igd tz);
rhx(u, u, 1, rft, igtx, igdtx, igty, igdty, igtz, igd tz);
DoneRHSJ = 1;
NumBoundaries = 0;
InitializedRHSJ = 0;
StartRHSJ(NULL);
}

void Cube::StartRHS(J(SomeMsg *m)
{
SendPrevRHS();
SendNextJ();
InitializedRHSJ = 1;
if ( YArraySize == 1 )
DoRHSJ(NULL);
}

void Cube::DoRHSJ(BoundaryMsg *m)
{
    if (DoneRHSJ == 0 || InitializedRHSJ == 0) {
        SendArray(thisgroup, CK, ap, Cube, DoRHSJ, s, m, myx, myy, myz);
        return;
    }
    if (NumBoundaries < 2) {
        SetBoundary(m);
        NumBoundaries++;
        if (NumBoundaries < 2) return;
    }
    setglobals(kexachX, kexachY, kexachZ, igatx, igatnx, igaty, igatny, igatz, igatnz);
    rhs_x, rdx, ftxgtx, igatnx, igatny, igatnz, igatnz);
    doneRHSJ = 1;
    doneRHSJ = 0;
    if (DoneRHSJ == 1) return;  // reset for next im
}

void Cube::StartRHSJ(K(SomeMsg *m)
{
    SendPre(V(RHS));
    SendNextK(K);
    InitializedRHSJ = 1;
    if (ZArraySize == 1) {
        DoRHSJ(NULL);
    }
}

void Cube::DoRHSJ(K(BoundaryMsg *m)
{
    if (DoneRHSJ == 0 || InitializedRHSJ == 0) {
        SendArray(thisgroup, CK, ap, Cube, DoRHSJ, s, m, myx, myy, myz);
        return;
    }
    if (NumBoundaries < 2) {
        SetBoundary(m);
        NumBoundaries);
        if (NumBoundaries < 2)
        return;
    }
    setglobals(kexachX, kexachY, kexachZ, igatx, igatnx, igaty, igatny, igatz, igatnz);
    rhs_x, rdx, ftxgtx, igatnx, igatny, igatnz, igatnz);
    doneRHSJ = 1;
    doneRHSJ = 0;
    if (DoneRHSJ == 1) return;  // reset for next im
}

/******* THIS ROUTINE IS FOR LOCAL RSD NORMS *******

void Cube::LocalRSDNorm()
{
    NormaMag += new NormaMag;
    for (int i = 0; i < 3; i++)
        m += sumj = 0.0;
    setglobals(kexachX, kexachY, kexachZ, igatx, igatnx, igaty, igatny, igatz, igatnz);
    norma O = GlobalRSDNorm(m);
}

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

void Cube::StartADI(leave)(SomeMsg *m)
{
    SendPreV(ADI);
    if (myx == XArraySize-1)
        NumBoundaries++;
    Initialized = 1;
    if (XArraySize == 1)
        DoADI(leave)(NULL);
}

void
Cube: DoADLsweep (BoundaryMsg em)
{
    if (Initializedsweep == 0) {
        SendArray(thigroup, CK.ep.Cube.DoADLsweep, m, myx, myy, myz);
        return;
    }
    NumBoundaries ++;
    if (NumBoundaries < 2)
        return;
    NumBoundaries = 0;
    Initializedsweep = 0; // reset for next stmt
    setglobals (knachx, knachy, knachz, igtx, igtix, igty, igtix, igtz, igtdx);
    adjx (a.red, fret, a, b, c, d, e, igtx, igtix, igty, igtix, igtz, igtdx);
    if (myx != XArraySize - 1) {
        BoundaryMsg m2 = GetBoundary (NEXT.ALL.ARRAYS);
        SendArray (cubearray, CK.ep.Cube.DoADLsweep, m2, myx + 1, myy, myz);
    } else {
        ADLback.sweep (NULL);
    }
}

void Cube: ADLback.sweep (BoundaryMsg em)
{
    if (myx != XArraySize - 1) {
        SetBoundary (m);
    }
    setglobals (knachx, knachy, knachz, igtx, igtix, igty, igtix, igtz, igtdx);
    badjx (a.red, fret, a, b, c, d, e, igtx, igtix, igty, igtix, igtz, igtdx);
    if (myx != 0) {
        BoundaryMsg m2 = GetBoundary (PREV.BSDONLY);
        SendArray (cubearray, CK.ep.Cube.ADlback.sweep, m2, myx - 1, myy, myz);
    }
    #ifdef NOSYNC
    else {
        SomeMsg m2 = new SomeMsg;
        controlbox[0] = EndADLSync (m2);
    }
    #endif
    DoneADI = 1;
}

CUBE: StartADLsweep (SomeMsg *msg)
{
    SendPrev (ADI);
    if (myy == YArraySize - 1)
        NumBoundaries ++;
    Initializedsweep = 1;
    if (YArraySize == 1)
        DoADLsweep (NULL);
}

void Cube: DoADLsweep (BoundaryMsg em)
{
    if (DoneADI == 0 || Initializedsweep == 0) {
        SendArray (thigroup, CK.ep.Cube.DoADLsweep, m, myx, myy, myz);
        return;
    }
    if (NumBoundaries < 2 && m != NULL) {
        SetBoundary (m);
        NumBoundaries ++;
        if (NumBoundaries < 2)
            return;
    }
    NumBoundaries = 0;
    Initializedsweep = 0; // reset for next stmt
    setglobals (knachx, knachy, knachz, igtx, igtix, igty, igtix, igtz, igtdx);
    ady (a.red, fret, a, b, c, d, e, igtx, igtix, igty, igtix, igtz, igtdx);
    if (myy != YArraySize - 1) {
        BoundaryMsg m2 = GetBoundary (NEXT.ALL.ARRAYS);
    }
SendArray(cubearray, CK.ep.Cube.DoADI.Ksweep, m2, myx, myy+1, myz);
} else { 
    ADBack.Ksweep(NULL);
}

void Cube::ADBack.Ksweep(BoundaryMsg *m)
{
    if (myy != YArraySize - 1) {
        SetBoundary(m);
    }

    setglobals(xonachs, &onachs, &onact, igtx, igondx, igpty, igondy, igonst, igondz);
    badtx.(a,rd,frct,a,b,c,d,e,igtx, igondx, igpty, igondy, igonst, igondz);
    if (myy != 0) {
        BoundaryMsg *m2 = GetBoundary(KPREW.RSD.ONLY);
        SendArray(cubearray, CK.ep.Cube.ADBack.Ksweep, m2, myx, myy-1, myz);
    } #ifdef NOSYNCS
    else {
        SomeMsg *n2 = new SomeMsg;
        controlbox[i++]=EndADLSyns(m2);  
    } #endif

    DoneADI = 1;
    #ifdef NOSYNCS
    SomeMsg *mans = new SomeMsg;
    SendArray(thingroup, CK.ep.Cube.StartADI.Ksweep, mans, myx, myy, myz);
    #ifdef TRANSPOSE
    migrate(KsweepMapfn);
    #endif
    #endif

void Cube::StartADI.Ksweep(SomeMsg *msg)
{
    SendPrevK(ADI);
    if (myy == ZArraySize-1)
        NumBoundaries++;
    InitializedKsweep = 1;
    if (ZArraySize == 1)
    {
        if (msg) msg = CubeArray;
    } else {
        ADBack.Ksweep(NULL);
    }

void Cube::DoADI.Ksweep(BoundaryMsg *m)
{
    if (DoneADI == 0) {
        SendArray(thingroup, CK.ep.Cube.DoADI.Ksweep, m, myx, myy, myz);
        return;
    }

    if (NumBoundaries < 2 && m != NULL) {
        SetBoundary(m);
        NumBoundaries++;
    } else if (NumBoundaries < 2) {
        return;
    } NumBoundaries = 0;
    DoneADI = 0;
    InitializedKsweep = 0;
}

setglobals(xonachs, &onachs, &onact, igtx, igondx, igpty, igondy, igonst, igondz);
badtx.(a,rd,frct,a,b,c,d,e,igtx, igondx, igpty, igondy, igonst, igondz);
if (myy != ZArraySize-1) {
    BoundaryMsg *m2 = GetBoundary(KPREW.ALL.ARRAYS);
    SendArray(cubearray, CK.ep.Cube.DoADI.Ksweep, m2, myx, myy, myy+1);
} else {
    ADBack.Ksweep(NULL);
}

void Cube::ADBack.Ksweep(BoundaryMsg *m)
{
    if (myy != ZArraySize - 1) {
        SetBoundary(m);
    }

    setglobals(xonachs, &onachs, &onact, igtx, igondx, igpty, igondy, igonst, igondz);
    badtx.(a,rd,frct,a,b,c,d,e,igtx, igondx, igpty, igondy, igonst, igondz);
    if (myy != 0) {
        BoundaryMsg *m2 = GetBoundary(KPREW.RSD.ONLY);
        SendArray(cubearray, CK.ep.Cube.ADBack.Ksweep, m2, myx, myy-1);
    } #ifdef NOSYNCS
    else {
    }
SomeMsg m2 = new SomeMsg;
        controlbo[0] = EndADILSync(m2);

#endif

mynul_.x, y, z, igax, igay, igaz, igay, igaz, igax);

#endif NOSYNCS
        SomeMsg sum = new SomeMsg;
        SendArray(thegroup, CKsp, Cube, StarRHS_L, mm, myy, mxy, mxy);
#endif TRANSPOSE
        migrate(lswepMapfn);
#endif
#endif

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

    setglobals(lkoch, loandy, lsecond, igax, igay, igaz, igay, igaz, igax);

    cnorms(rdm.m = sum);

    controlbo[0] = GlobalCNorms(m);
}

/**** CODE FOR STUFF AFTER FINISHING ADI LOOP ****/
void Cube::FinishADILLoop(SomeMsg m2)
{
    /* This is for the local error */
    NormsMsg sm = new NormsMsg;
    for (int i=0; i<5; i++)
        m-sum[i] = 0.0;

    setglobals(lkoch, loandy, lsecond, igax, igay, igaz, igay, igaz, igax);
    error(x, m-sum, igax, igay, igaz, igay, igaz, igax);

    controlbo[0] = GlobalError(m);
}
cube.P

PhivMsg *m = new (ksize) PhivMsg;
int m->nx = m->ny = m->nxy = m->nxyz = m->nxyz = m->nxyz;
memset(m->phi0), sizeof(double)*size);  // make double word boundary
get2s(*a, m->phi);  // present copying
controlbo[0] = ReceivePhiv(m);
}
}

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

ArrayMsg *
Cube::pack(EntryPointType *unpackp)
{
    *unpackp = L(Cube::Unpack);
    memcpy(dataarea->data, this, sizeof(Cube));

dataarea = p;
}

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

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

    /* Allocate space for local arrays */
    a = new double[alloc*alloc*alloc*alloc];
    b = new double[alloc*alloc*alloc*alloc];
    c = new double[alloc*alloc*alloc*alloc];
    d = new double[alloc*alloc*alloc*alloc];
    e = new double[alloc*alloc*alloc*alloc];

tgx = &gxz;
     gy = &gdx;
     gz = &gdy;

    igedx = &gedx;
    int cubearea = (sizeof(Cube)/8 + 1) * 8;
    int eacharray = 5 * alloc * alloc * alloc * sizeof(double);
    ptr += cubearea;
    u = (double *)ptr;
    ptr += eacharray;
    rd = (double *)ptr;
    ptr += eacharray;
    rct = (double *)ptr;
    PrintArray(*file, double *a, int size)
{
    FILE *fp = fopen(*file, "w")
    for (int i = 0; i < size; i++)
        printf(fp, "%f", a[i]);
    printf(fp, "Done !!!!!\n*");
    fclose(fp);

}
#define XArraySize 1
#define YArraySize 1
#define ZArraySize 1
#define JPREV 1
#define JNEXT 2
#define JPREV 3
#define JNEXT 4
#define KPREV 5
#define KNEXT 6
#define U_ONLY 1
#define RSD_ONLY 2
#define ALLARRAYS 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 ro[VARSIZE];
};

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

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

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

char class Cube : public array {
  public:
    int istep, itmax;
    int myx, myy, myz;
    int gtx, gty, gtz;
    int gnx, gny, gnz;
    int nx, ny, nz;
    int sx, sy, sz;
    int NumBoundaries;
    PackMsg *dataarea;
    double sv, svd, svft;
    double sa, b, c, d, e;
    int DoneRHI, DoneRHSJ, DoneADI, DoneACJ;
    int InitializedRHS, InitializedJrigh, InitializedKsweep;
    int InitializedRHS, InitializedRHSJ, InitializedRHSK;
  
  entry:
    Cube(SomeMsg *m);
  
  public:
    void SetBoundary(BoundaryMsg *m);
    BoundaryMsg * GetBoundary(int type, int whicharrays);
    void BvLvExn();
    void SendPrev(int);
    void SendNext();
    void SendPrevJ(int);
    void SendNextJ();
    void SendPrevK();
    void SendNextK();
    void LocalRSDNorm();

  //*************** THESE 3 EPs ARE FOR THE RHS SWEEPS **************/
  entry:
    void StartRHSJ(SomeMsg *m);
void DoRHS(J(BoundaryMsg *m));
void StartRHS(J(SomeMsg *m));
void DoRHS(J(BoundaryMsg *m));
void StartRHS(K(SomeMsg *m));
void DoRHS(K(BoundaryMsg *m));

/************ THESE 3 Eqs ARE FOR THE ADI SWEEPS ************/
entry:
void StartADI.Sweep(SomeMsg *msg);
void DoADI.Sweep(BoundaryMsg *m);
void ADBackward.Sweep(BoundaryMsg *m);
void StartADI.Sweep(SomeMsg *msg);
void DoADI.Sweep(BoundaryMsg *m);
void ADBackward.Sweep(BoundaryMsg *m);
void StartADI.K.Sweep(SomeMsg *msg);
void DoADI.K.Sweep(BoundaryMsg *m);
void ADBackward.K.Sweep(BoundaryMsg *m);

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

void FinishedADI.Sweep(SomeMsg *);

/*** THIS ROUTINE IS FOR local print() ****/
entry:
void LocalPrint(SomeMsg *m);

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

entry:
void UnPack(PackMsg *p);
} ;
```c
#include <stdio.h>
#include "parray.h"
#include "cube.h"
#include "controloc.h"
#include "main.h"

extern "C" MAIN();
MAIN()
{
}

extern "C" double sqrt(double);

readonly Ctrlloc group controloc;
readonly Cube group cubearray;
readonly int NumCubes;

int BrunoCapelloMapFn(int gid, int i, int j, int k)
{
    if ( XArraySize \neq YArraySize && YArraySize \neq ZArraySize )
        CPrintf("ERROR: ArraySize is not same in all dimen\n");
    /* General expression for 3-D Bruno-Capello is:
       Map(i,j,k) = i(i-k)*a + j(j-k)*b
       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)
{
    if ( XArraySize \neq YArraySize && YArraySize \neq ZArraySize )
        CPrintf("ERROR: ArraySize is not same in all dimen\n");
    return ( 2(i*(i) + j*(j) ) );
}

int JsonMapFn(int gid, int i, int j, int k)
{
    return ( ZArraySize * j + k );
}

int JsonMapFn(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 JsonMapFn(int gid, int i, int j, int k)
{
    return ( YArraySize * j + i );
}

/** MAIN CHARE **/

main:main()
{
    printf("Starting Charm++ pgs, creating Controloc\n");
    NumCubes = XArraySize * YArraySize * ZArraySize;
    SomeMsg *m = new SomeMsg;
    controloc = newgroup Controloc(m);
    NumCubesDone = 0;
}

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

cubearray = CreateArray(.CK.charm.Cube. .CK.snp.Cube.Cube, m,
    BrunoCapelloMapFn,
    XArraySize,YArraySize,ZArraySize);
}
```
char class main {
    int NumCubesDone;

    public:
    main();

    entry:
    void InitialSync(SomeMsg *m);
};
implicit real*8 (a-h,o-z)

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 )

real timer, tstart, tend

integer ex, ey, ez, ex1, ey1, ez1, ex2, ey2, ez2,
  $ sx, sy, sz,
  $ sx1, syl, sz1, sx2, sy2, sz2,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
  $ sx2, syl, sz2, sx1, syl, sz1,
  $ sx3, ex3, sy3, ey3, sz3, ez3,
  $ gtx, gdx, gty, gdx, gty, gtx, gdx,
subroutine initsp(ipe)

***driver for the performance evaluation of the solver for
five coupled, nonlinear partial differential equations.

Author: Sisira Weeratunga
NASA Ames Research Center
(10/25/90)

include 'appsp.incl'

***open file for input data

open (unit=5, file='appsp.input', status='old',
      access='sequential', form='formatted')
rewind 5

***read the unit number for output data

read (5,*)
read (5,*) iout

***flag that controls printing of the progress of iterations

read (5,*)
read (5,*) ipr, inorm

***set the maximum number of pseudo-time steps to be taken

read (5,*)
read (5,*) itmax

***set the magnitude of the time step

read (5,*)
dt = read (5,*)

***set the value of over-relaxation factor for SSOR iterations

read (5,*)
read (5,*) omega

***set the steady-state residual tolerance levels

read (5,*) tolrsd(1),tolrsd(2),
  tolrsd(3),tolrsd(4),tolrsd(5)

***read problem specification parameters

****specify the number of grid points in xi, eta and zeta directions

read (5,*)
read (5,*) nx, ny, nz

***open the file for output data

if (iout .eq. 7) then
  open (unit=7, file='output.data', status='unknown',
       access='sequential', form='formatted')
  rewind 7
end if

if ((nx .lt. 5) .or. 
    (ny .lt. 5) .or. 
    (nz .lt. 5)) then
  write (*,2001)
  2001 format (5x,'PROBLEM SIZE IS TOO SMALL - ',
       5x,'SET EACH OF NX, NY AND NZ AT LEAST EQUAL TO 5')
  stop
end if

if ((nx .gt. isize1) .or. 
    (ny .gt. isize2) .or. 
    (nz .gt. isize3)) then
  write (*,2002)
  2002 format (5x,'PROBLEM SIZE IS TOO LARGE - ',
       5x,'NX, NY AND NZ SHOULD BE LESS THAN OR EQUAL TO ',
       5x,'ISIZE1, ISIZE2 AND ISIZE3 RESPECTIVELY')
end if

dxi = 1.d0 + 0 / (nx - 1)
deta = 1.d0 + 0 / (ny - 1)
dzeta = 1.d0 + 0 / (nz - 1)
tx1 = 1.d0 + 0 / (dxi * dxi)
tx2 = 1.d0 + 0 / (2.0d0 + 0 * dxi)
tx3 = 1.d0 + 0 / dxi
ty1 = 1.d0 + 0 / (deta * deta)
ty2 = 1.d0 + 0 / (2.0d0 + 0 * deta)
ty3 = 1.d0 + 0 / deta
tz1 = 1.d0 + 0 / (dzeta * dzeta)
tz2 = 1.d0 + 0 / (2.0d0 + 0 * dzeta)
tz3 = 1.d0 + 0 / dzeta

ii1 = 2
ii2 = nx - 1
ji2 = ny - 2
ki2 = nz - 1

c
frc1 = 0.0d+00
frc2 = 0.0d+00
frc3 = 0.0d+00

c
***diffusion coefficients

dx1 = 0.75d+00
dx2 = dx1
dx3 = dx1
dx4 = dx1
dx5 = dx1

dy1 = 0.75d+00
dy2 = dy1
dy3 = dy1
dy4 = dy1
dy5 = dy1

dz1 = 1.00d+00
dz2 = dz1
dz3 = dz1
dz4 = dz1
dz5 = dz1

c
***fourth difference dissipation

dssp = ( max (dx1, dy1, dz1 ) ) / 4.0d+00

c
***coefficients of the exact solution to the first pde

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
***coefficients of the exact solution to the second pde

ce(2,1) = 1.0d+00
ce(2,2) = 0.0d+00
ce(2,3) = 0.0d+00

c
***coefficients of the exact solution to the third pde

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
***coefficients of the exact solution to the fourth pde

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
***coefficients of the exact solution to the fifth pde

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
**Aug 27 11:59 1996 appsp.f Page 5**

```fortran
ce(5,12) = 3.0d-01
ce(5,13) = 2.0d-01

return
end

*************** this control code is now in C++ ***************

***set the boundary values for dependent variables

call setbv

***set the initial values for dependent variables

call setiv

***compute the forcing term based on prescribed exact solution

call erhs

***perform scalar approximate factorization iterations

call adi

***compute the solution error

call error

***compute the surface integral

call pintgr

***verification test

call verify ( rsdum, errmm, frc )

***print the CPU time

c write ( iout,1001 ) ttotal
c1001 format ( //5x,'Total CPU time = ',1pe12.4,' Sec. ' )
return
end

***************

```

**Aug 27 11:59 1996 appsp.f Page 6**

```fortran
include 'appsp.incl'

integer inx, iny, inz

inx = nx
iny = ny
inz = nz

itmax = itmax

return
end

subroutine setglobals ( iex, iey, iez, igstx, igendx, igsty, igeny, igstz, igenz, igendx, igstx, igeny, igstz, igenz)

***set global variables for parallelization

Author: Sanjeev Krishnan

include 'appsp.incl'

integer iex, iey, iez, igstx, igendx, igsty, igendy, igstz, igenz

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
ex = iex + 2
ey = iey + 2
ez = iez + 2
sx = 3
sy = 3
sz = 3

xs = iex + 4
ys = iey + 4
zs = iez + 4

gstx = igstx
gstz = igstz
genx = igenx
gendy = igeny
gendz = igenz

sx1 = sx
syl = sy
szl = sz
if ( gstx .eq. 1 ) sx1 = sx + 1
if ( gstz .eq. 1 ) syl = sy + 1
```

---

The above text contains Fortran code for a scientific computing program. The code includes routines for setting boundary values, initializing dependent variables, computing forcing terms, performing factorization iterations, computing solution errors, and calculating surface integrals. It also includes a subroutine for setting global variables for parallelization. The code is written in a clear and structured manner, with comments explaining the purpose of each section.
if ( gstz .eq. 1 ) sz1 = sz + 1

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

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

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

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

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

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
ey1 = ey + 1
ezm1 = ez + 1
if ( gendx .eq. nx ) exm1 = ex
if ( gendy .eq. ny ) eym1 = ey
if ( gendz .eq. nz ) ezm1 = ez

sx2 = sx - 2
sy2 = sy - 2
sz2 = sz - 2
if ( gstx .eq. 1 ) sx2 = sx
if ( gsty .eq. 1 ) sym2 = sy

function iglob (i)

include 'appsp.incl'
integer i,iglob

itmp = gatx + i - 3
iglob = itmp

return
end

function jglob (j)

include 'appsp.incl'
integer j,jglob

jtemp = gatx + j - 3
iglob = jtemp

return
end

function kglob (k)

include 'appsp.incl'
integer k,kglob

ktemp = gatx + k - 3
kglob = ktemp

return
end

subroutine setb(b, data, type)

include 'appsp.incl'

real *8 b(5,xsize,yzize.zsize)
real *8 data(*)
integer type
integer count

  count = 1

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
endif

return

end

subroutine setcd(cd, data, type)
include 'appsp.incl'
real *8 cd(xsize,yzize.zsize,3)eal *8 data(*)
integer type
integer count

  count = 1

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
endif

return

end
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
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
return
c
c
 subroutine getb(b, data, type, count)

c
data(count) = b(m, i, j, k)
count = count + 1
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
c

subroutine getcd(cd, data, type, count)
c
include 'appsp.incl'
c
real *8 cd(xsize, ysize, zsize, 3)
real *8 data(*)
integer type
inger 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
delse 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
delse 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
delse 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
c
c
c
******************************************************************************
subroutine setbv(u, igstx, igndx, igsty, igendy, igstz, igendz)

*** set the boundary values of dependent variables

Author: Sisira Weeratunga
NASA Ames Research Center
(10/25/90)

include 'appsp.incl'
real *8 u(s.xsize, ysize, zsize)
integer igstx, igndx, igsty, igendy, igstz, igendz

gstx = igstx
gndx = igndx
gsty = igsty
gendy = igendy
gstz = igstz
gendz = igendz

*** set the dependent variable values along the top and bottom faces

do j = sy, ey
   do i = sx, ex
      call exact (iglob(i), jglob(j), 1, bzl(1, i, j))
      call exact (iglob(i), jglob(j), nz, bznz(1, i, j))
   end do
   end do

if (gstz.eq.1) then
   do j = sy, ey
      do i = sx, ex
         call exact (iglob(i), jglob(j), 1, u(1, i, j, sz))
      end do
   end do
endif

*** set the dependent variable values along east and west faces

do k = sz, ez
   do j = sy, ey
      do i = sx, ex
         call exact (1, jglob(j), kglob(k), bxl(1, j, k))
      end do
   end do
endif

*** set the dependent variable values along north and south faces

do k = sz, ez
   do j = sy, ey
      do i = sx, ex
         call exact (1, jglob(j), kglob(k), u(1, sx, j, k))
      end do
   end do
endif

return
end

subroutine setiv(u, igx, igstx, igstz)
***set the initial values of independent variables based on tri-linear interpolation of boundary values in the computational space.

include 'appsp.incl'

real *8 u(5, xsize, ysize, zsize)
integer igx, igstx, igstz

gstx = igstx
gsty = igsty
gstz = igstz

Author: Sisira Weeratunga
NASA Ames Research Center
10/25/90

do k = szl, ezl

kg = kglob(k)
zeta = ( dfloat(kg-1) ) / ( nz - 1 )
do j = syl, eyl

jg = jglob(j)
et = ( dfloat(jg-1) ) / ( ny - 1 )
do i = sxl, ex

ig = iglob(i)
xi = ( dfloat(ig-1) ) / ( nx - 1 )
do m = 1, 5

pxi = ( 1.0d+00 - xi ) * bx1(m, j, k)
+ xi * bxn(m, j, k)

peta = ( 1.0d+00 - et ) * by1(m, i, k)
+ et * bym(m, i, k)

pzeta = ( 1.0d+00 - zeta ) * bz1(m, i, j)
+ zeta * bznm(m, i, j)

u(m, i, j, k) = pxi + peta + pzeta
- pxi * peta + peta * pzeta - pxi * pxi
+ pxi * peta * pzeta

end do
end do

end do
**Aug 27 11:59 1996 appsp.f Page 19**

```fortran
        if ( ipr .eq. 1 ) then
            write (iout,*) ' Initial Residual norms'
            write (iout,1007) ( rsdnum(m), m = 1, 5 )
        end if
        end if

**begin pseudo-time stepping iterations**

tstart = timer( )
do istep = 1, itmax

*************** END OF OLD CODE AT BEGINNING OF ADI **********

**end of pseudo-time stepping iterations**

```fortran
        subroutine adix(u,rxd,frct,a,b,c,d,e,igstx, igendx, igsty, igendz)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
          real *8 u(5,xsize,ysize,zsize), rxd(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)
          dimension idmax(5), jmax(5), kmax(5)
          include 'appsp.incl'

          real *8 rxd(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize), frct(5,xsize,ysize,zsize)
          real *8 a(xsize,ysize,zsize,3), b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3)
          real *8 d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
          real *8 u(5,xsize,ysize,zsize), rxd(5,xsize,ysize,zsize), frct(5,xsize,ysize,zsize)
          real *8 a(xsize,ysize,zsize,3), b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3)
          real *8 d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
          real *8 u(5,xsize,ysize,zsize), rxd(5,xsize,ysize,zsize), frct(5,xsize,ysize,zsize)
          real *8 a(xsize,ysize,zsize,3), b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3)
          real *8 d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
        subroutine adix(u,rxd,frct,a,b,c,d,e,igstx, igendx, igsty, igendz)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
          real *8 u(5,xsize,ysize,zsize), rxd(5,xsize,ysize,zsize), frct(5,xsize,ysize,zsize)
          real *8 a(xsize,ysize,zsize,3), b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3)
          real *8 d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
          real *8 u(5,xsize,ysize,zsize), rxd(5,xsize,ysize,zsize), frct(5,xsize,ysize,zsize)
          real *8 a(xsize,ysize,zsize,3), b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3)
          real *8 d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
        subroutine adix(u,rxd,frct,a,b,c,d,e,igstx, igendx, igsty, igendz)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
          real *8 u(5,xsize,ysize,zsize), rxd(5,xsize,ysize,zsize), frct(5,xsize,ysize,zsize)
          real *8 a(xsize,ysize,zsize,3), b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3)
          real *8 d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
          real *8 u(5,xsize,ysize,zsize), rxd(5,xsize,ysize,zsize), frct(5,xsize,ysize,zsize)
          real *8 a(xsize,ysize,zsize,3), b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3)
          real *8 d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
        subroutine adix(u,rxd,frct,a,b,c,d,e,igstx, igendx, igsty, igendz)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
          real *8 u(5,xsize,ysize,zsize), rxd(5,xsize,ysize,zsize), frct(5,xsize,ysize,zsize)
          real *8 a(xsize,ysize,zsize,3), b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3)
          real *8 d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
          real *8 u(5,xsize,ysize,zsize), rxd(5,xsize,ysize,zsize), frct(5,xsize,ysize,zsize)
          real *8 a(xsize,ysize,zsize,3), b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3)
          real *8 d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
          real *8 u(5,xsize,ysize,zsize), rxd(5,xsize,ysize,zsize), frct(5,xsize,ysize,zsize)
          real *8 a(xsize,ysize,zsize,3), b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3)
          real *8 d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
          real *8 u(5,xsize,ysize,zsize), rxd(5,xsize,ysize,zsize), frct(5,xsize,ysize,zsize)
          real *8 a(xsize,ysize,zsize,3), b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3)
          real *8 d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
          real *8 u(5,xsize,ysize,zsize), rxd(5,xsize,ysize,zsize), frct(5,xsize,ysize,zsize)
          real *8 a(xsize,ysize,zsize,3), b(xsize,ysize,zsize,3), c(xsize,ysize,zsize,3)
          real *8 d(xsize,ysize,zsize,3), e(xsize,ysize,zsize,3)
          integer igstx, igendx, igsty, igendz
          integer idmax(5), jmax(5), kmax(5)
          parameter ( one = 1.0d+00 )
```
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)
$ delurn(5)
$ imax(5), jmax(5), kmax(5),
$ idmax(5), jmax(5), kmax(5)
$ parameter ( one = 1.0d+00 )
integer igstx, igendx, igsty, igendency, igstz, igendz

$$
gstx = igstx$$
$gendx = igendx$
$gsty = igsty$
$gendy = igendency$
$gstz = igstz$
$gendz = igendz$

$\text{Backward substitution}$

$\text{call bspentax3 ( a(1,1,1,1), b(1,1,1,1), c(1,1,1,1),}$
$\text{d(1,1,1,1), e(1,1,1,1),}$
$\text{rsd )}$
$\text{call bspentax ( 4, a(1,1,1,2), b(1,1,1,2), c(1,1,1,2),}$
$\text{d(1,1,1,2), e(1,1,1,2),}$
$\text{rsd )}$
$\text{call bspentax ( 5, a(1,1,1,3), b(1,1,1,3), c(1,1,1,3),}$
$\text{d(1,1,1,3), e(1,1,1,3),}$
$\text{rsd )}$

$\text{return}$
$\text{end}$
$\text{subroutine adiy(u, rsd, frct, a, b, c, d, e, igstx, igendx, igsty,}$
$\text{igendency, igstz, igendz)}$

$\text{include 'appsp.incl'}$

$\text{real *8 u(5, xsize, ysize, zsize), rsd(5, xsize, ysize, zsize, 3),}$
$\text{frct(5, xsize, ysize, zsize), a(xsize, ysize, zsize, 3),}$
$\text{b(xsize, ysize, zsize, 3), c(xsize, ysize, zsize, 3),}$
$\text{d(xsize, ysize, zsize, 3), e(xsize, ysize, zsize, 3),}$
$\text{delurn(5) parameter ( one = 1.0d+00 )}$
integer igstx, igendx, igsty, igendency, igstz, igendz

$\text{gstx = igstx}$
$\text{gendx = igendx}$
$\text{gsty = igsty}$
$\text{gendy = igendency}$
$\text{gstz = igstz}$
$\text{gendz = igendz}$

$\text{**** This code was formerly in the middle of the ADI loop *****}$
$\text{****perform the block diagonal inversion}$
$\text{call ninvr(rsd)}$
$\text{****perform the eta-direction sweep}$
$\text{call jacy ( 3, u, a(1,1,1,1), b(1,1,1,1), c(1,1,1,1),}$
$\text{d(1,1,1,1), e(1,1,1,1),}$
$\text{rsd )}$
$\text{call spentay3 ( a(1,1,1,1), b(1,1,1,1), c(1,1,1,1),}$
$\text{d(1,1,1,1), e(1,1,1,1),}$
$\text{rsd )}$
$\text{call jacy ( 4, u, a(1,1,1,2), b(1,1,1,2), c(1,1,1,2),}$
$\text{d(1,1,1,2), e(1,1,1,2),}$
$\text{rsd )}$
$\text{call spentay ( 4, a(1,1,1,2), b(1,1,1,2), c(1,1,1,2),}$
$\text{d(1,1,1,2), e(1,1,1,2),}$
$\text{rsd )}$
$\text{call jacy ( 5, u, a(1,1,1,3), b(1,1,1,3), c(1,1,1,3),}$
$\text{d(1,1,1,3), e(1,1,1,3),}$
$\text{rsd )}$
$\text{call spentay ( 5, a(1,1,1,3), b(1,1,1,3), c(1,1,1,3),}$
$\text{d(1,1,1,3), e(1,1,1,3),}$
$\text{rsd )}$

$\text{return}$
$\text{end}$
$\text{subroutine badiy(u, rsd, frct, a, b, c, d, e, igstx, igendx, igsty,}$
$\text{igendency, igstz, igendz)}$

$\text{include 'appsp.incl'}$

$\text{real *8 u(5, xsize, ysize, zsize), rsd(5, xsize, ysize, zsize, 3),}$
$\text{frct(5, xsize, ysize, zsize, 3), a(xsize, ysize, zsize, 3),}$
$\text{b(xsize, ysize, zsize, 3), c(xsize, ysize, zsize, 3),}$
$\text{d(xsize, ysize, zsize, 3), e(xsize, ysize, zsize, 3),}$
$\text{delurn(5) parameter ( one = 1.0d+00 )}$
integer igstx, igendx, igsty, igendency, igstz, igendz

$\text{gstx = igstx}$
$\text{gendx = igendx}$
$\text{gsty = igsty}$
$\text{gendy = igendency}$
$\text{gstz = igstz}$
$\text{gendz = igendz}$
$\text{delunm(5)}$
parameter ( one = 1.0d+00 )

integer igstx, igndx, igsty, igdeny, igstz, igendz

gstx = igstx
gendx = igendx
gsty = igsty
gendy = igdeny
gsz = igstz
gendz = igendz

***backward sweep

call bsptay3 ( a(1,1,1), b(1,1,1), c(1,1,1),
$ \text{d(1,1,1), e(1,1,1),}$
$ \text{rsd} )$

call bsptay ( 4, a(1,1,1), b(1,1,1), c(1,1,1),
$ \text{d(1,1,1), e(1,1,1),}$
$ \text{rsd} )$

call bsptay ( 5, a(1,1,1), b(1,1,1), c(1,1,1),
$ \text{d(1,1,1), e(1,1,1),}$
$ \text{rsd} )$

return
end

subroutine adiz(u,rsd,frct,a,b,c,d,e,igstx, igendx, igsty, igdeny, igstz, igendz)
$ \text{include 'appsp.incl'}$

real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize),
$ \text{frct(5,xsize,ysize,zsize), a(xsize,ysize,zsize),}$
$ \text{b(xsize,ysize,zsize), c(xsize,ysize,zsize),}$
$ \text{d(xsize,ysize,zsize), e(xsize,ysize,zsize),}$

dimension idmax(5), jmax(5), kmax(5),
$ \text{delunm(5)}$

integer igstx, igendx, igsty, igdeny, igstz, igendz

gstx = igstx
gendx = igendx
gsty = igsty

subroutine badiz(u,rsd,frct,a,b,c,d,e,igstx, igendx, igsty, igdeny, igstz, igendz)
$ \text{include 'appsp.incl'}$

real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize),
$ \text{frct(5,xsize,ysize,zsize), a(xsize,ysize,zsize),}$
$ \text{b(xsize,ysize,zsize), c(xsize,ysize,zsize),}$
$ \text{d(xsize,ysize,zsize), e(xsize,ysize,zsize),}$

dimension idmax(5), jmax(5), kmax(5),
$ \text{delunm(5)}$

parameter ( one = 1.0d+00 )
integer igstx, igendx, igsty, igendy, igstz, igendz

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

c**** This code was formerly near the end of the ADI loop ********

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 )

return
end

subroutine mvmul(u,rsd,igstx, igendx, igsty, igendy,
$        igstz, igendz)

include 'appsp.incl'
real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize)
parameter ( one = 1.0d+00 )

integer igstx, igendx, igsty, igendy, igstz, igendz

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

c****perform the block diagonal matrix-vector multiplication

c    call tzeta(u,rsd)

c****update the variables

do k = szl, es1
  do j = syl, ey1
    do i = sx1, ex1
      do m = 1, 5
        u( m, i, j, k ) = u( m, i, j, k )
        * rsd( m, i, j, k )
      end do
    end do
  end do
end do

return
end

subroutine cnorms(rsd,delumn)
include 'appsp.incl'
implicit real *8 ( a-h,o-z)
real *8 rsd(5,xsize,ysize,zsize)
dimension idmax(5), jdxmax(5), kdxmax(5),
$        imax(5), jmax(5), kmax(5),
$        delumn(5)
parameter ( one = 1.0d+00 )
real *8 tmp(5)

llnorm = 2

if ( mod ( istep, inorm ) .eq. 0 ) then
  if ( lnorm .eq. 1 ) then
    call maxnorm ( isiz1, isiz2, isiz3,
$        nx, ny, nz,
$        idmax, jdxmax, kdxmax,
$        rsd, delumn )

  if ( ipr .eq. 1 ) then
    write (iout,1002) ( delumn(m),
$        idmax(m), jdxmax(m), kdxmax(m), m = 1, 5 )

end if
end if
Aug 27 11:59 1996 appsp.f Page 27

end if

else if ( lnorm .eq. 2 ) then
  call l2norm ( rsd, tmp )
  if ( impr .eq. 1 ) then
    write (iout,1006) ( tmp(m), m = 1, 5 )
  end if
end if

do m = 1, 5
  delumn(m) = delumn(m) + tmp(m)
  call psval(m,tmp(m))
end do

***compute the steady-state residuals

This is called from the C++ code

return

1002 format (1x/1x,'max-norm of scalar ADI-iteration correction ',
$ ' for first pde = ',1pe12.5/,
$ ' for second pde = ',1pe12.5/,
$ ' for third pde = ',1pe12.5/,
$ ' for fourth pde = ',1pe12.5/,
$ ' for fifth pde = ',1pe12.5/,
$ ' for sixth pde = ',1pe12.5/)

1006 format (1x/1x,'RMS-norm of scalar ADI-iteration correction ',
$ ' for first pde = ',1pe12.5/,
$ ' for second pde = ',1pe12.5/,
$ ' for third pde = ',1pe12.5/,
$ ' for fourth pde = ',1pe12.5/,
$ ' for fifth pde = ',1pe12.5/)
end if

dm = 1.5

rms(m) = rms(m) + rsdm(m)

end do

check the pseudo-time iteration residuals against the tolerance levels

if ( (rsd1(1).lt.to1rs1) .and.
  (rsd1(2).lt.to1rs2) .and.
  (rsd1(3).lt.to1rs3) .and.
  (rsd1(4).lt.to1rs4) .and.
  (rsd1(5).lt.to1rs5) ) then

write (iout,1004) istep

return

end if

return

1003 format (1x/1x,'max-norm of steady-state residual for ', $  'first pde = ',1pe12.5/,
  ' second pde = ',1pe12.5/, $  'third pde = ',1pe12.5/, $  'fourth pde = ',1pe12.5/, $  'fifth pde = ',1pe12.5/, $  'pseudo-time steps ')' $)

1004 format (1x/1x,'convergence was achieved after ',i4, $  'pseudo-time steps')

1007 format (1x/1x,'RMS-norm of steady-state residual for ', $  'first pde = ',1pe12.5/, $  'second pde = ',1pe12.5/, $  'third pde = ',1pe12.5/, $  'fourth pde = ',1pe12.5/, $  'fifth pde = ',1pe12.5)
end if
end do
end do
end do

return
end

subroutine l2norm( v, sum )

***compute the 12-norm of vector v.

Author: Sisira Weeratunga
NASA Ames Research Center
(10/25/90)

include 'appsp.incl'

implicit real *8 (a-h,o-z)

dimension v(5,ldx,ldy,*),
real *8 v(5,xsize,ysize,zsize), sum(5)

do m = 1, 5
sum(m) = 0.0d+00
end do

do k = szl, ezl
  do j = sy1, ey1
    do i = sx1, ex1
      sum(m) = sum(m) * v(m,i,j,k) * v(m,i,j,k)
    end do
  end do
end do

***xi-direction flux differences

do k = szl, ez1
  do j = sy1, ey1
    do i = sx1, ex1
      sum(m) = sqrt( sum(m) / (nx-2) * (ny-2) * (nz-2) )
    end do
  end do
end do

return
end

subroutine erhs(frcx, igstx, igendx, igsty, igendy, igstz, igendz)

***compute the right hand side based on exact solution

Author: Sisira Weeratunga
NASA Ames Research Center
(10/25/90)

include 'appsp.incl'

real *8 frcx(5,xsize,ysize,zsize)
dimension flux(5,zsize), ue(5,zsize)
integer igstx, igsty, igstz
igstx = igstx
igendx = igendx
igsty = igsty
igendy = igendy
igstz = igstz
igendz = igendz
dssp = dssp

do k = sz, ez
  do j = sy, ey
    do i = sx, ex
      frcx( m, i, j, k ) = 0.0d+00
      end do
    end do
  end do
end do

kg = kglob(k)
seta = ( dfloat(kg-1) ) / ( nz - 1 )

do j = sy1, ey1
  do i = sx1, ex1
    jg = jglob(j)
teta = ( dfloat(jg-1) ) / ( ny - 1 )
    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) * u21 + u21 * u21 )
/ (ue(1,i))
c
flux(2,i) = u21 * ( u21 + c2 * ( (ue(5,i) - q) )
c
flux(3,i) = u21 * (u21) 
c
flux(4,i) = u21 * (u21) 
c
flux(5,i) = (c1 * u21 - c2 * q) * u21
c
end do
c
do i = sx1, ex1
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
c
c
e nd do
c
d o i = sx1, ex1

tmp = 1.0d+00 / ue(1,i)
c
 Aug 27 11:59 1996 appsp.f Page 34

u2li = tmp * ue(2,i)
u3li = tmp * ue(3,i)
u4li = tmp * ue(4,i)
u5li = tmp * ue(5,i)
c
tmp = 1.0d+00 / ue(1,i-1)
c
u2liml = tmp * ue(2,i-1)
u3liml = tmp * ue(3,i-1)
u4liml = tmp * ue(4,i-1)
u5liml = tmp * ue(5,i-1)
c
flux(2,i) = (4.0d+00/3.0d+00) * tx3 * ( u2li - u2liml )
flux(3,i) = tx3 * ( u3li - u3liml )
flux(4,i) = tx3 * ( u4li - u4liml )
flux(5,i) = 0.50d+00 * ( 1.0d+00 - cl*c5 )
$ * tx3 * ( ( u2li **2 + u3li **2 + u4li **2 )$
$ + ( u2liml**2 + u3liml**2 + u4liml**2 ) )
$ + (1.0d+00/6.0d+00)
$ * tx3 * ( u2li**2 - u2liml**2 )
$ + cl * c5 * tx3 * ( u5li - u5liml )
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-1) )$
$ + (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-1) )$
$ + (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-1) )$
$ + (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-1) )$
$ + (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-1) )$
$ + (ue(5,i-1) )$
$c
end do
***Fourth-order dissipation

if ( gstx .eq. 1 ) then
    do m = 1, 5
        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) )
    end do
    do i = sx1, ex3
        do m = 1, 5
            frct(m,i,j,k) = frct(m,i,j,k)
                - dsspm * ( - 4.0d+00 * ue(m,i-1) + 6.0d+00 * ue(m,i) )
                    - 4.0d+00 * ue(m,i+1) + ue(m,i+2) )
        end do
    end do
end if

***eta-direction flux differences

do k = sz1, ez1
    kg = kglob(k)
    zeta = ( dfloat(kg-1) ) / ( nz - 1 )
    do i = sz1, ex1
        ig = iglob(i)
        xi = ( dfloat(ig-1) ) / ( nx - 1 )
        do j = sym2, eym2
            jg = iglob(j)
            eta = ( dfloat(jg-1) ) / ( ny - 1 )
            do m = 1, 5
                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
            end do
        end do
    end do
end do

flux(1,j) = ue(3,j)

u31 = ue(3,j) / ue(1,j)

q = 0.50d+00 * ( ue(2,j) * ue(2,j) + u31 * u31 + ue(4,j) * ue(4,j) )

flux(2,j) = ue(2,j) * u31

flux(3,j) = ue(3,j) * u31 + c2 * ( ue(5,j) - q )

flux(4,j) = ue(4,j) * u31

flux(5,j) = ( c1 * ue(5,j) - c2 * q ) * u31

end do

do j = sy1, ey1
Aug 27 11:59 1996 appsp.f Page 37

do m = 1, 5
     frct(m,i,j,k) = frct(m,i,j,k)
         - ty2 * ( flux(m,j+1) - flux(m,j-1) )
end do

end do

do j = syl, eyml
     tmp = 1.0d+00 / ue(1,j)
     u2lj = tmp * ue(2,j)
     u3lj = tmp * ue(3,j)
     u4lj = tmp * ue(4,j)
     u5lj = tmp * ue(5,j)
     tmp = 1.0d+00 / ue(1,j-1)
     u2ljml = tmp * ue(2,j-1)
     u3ljml = tmp * ue(3,j-1)
     u4ljml = tmp * ue(4,j-1)
     u5ljml = tmp * ue(5,j-1)

     flux(2,j) = ty3 * ( u2lj - u2ljml )
     flux(3,j) = ( 4.0d+00/3.0d+00 ) * ty3 * ( u3lj - u3ljml )
     flux(4,j) = ty3 * ( u4lj - u4ljml )
     flux(5,j) = 0.50d+00 * ( 1.0d+00 - cl*c5 )
         * ty3 * ( ( u2lj **2 + u3lj **2 + u4lj **2 )
               + (1.0d+00/6.0d+00) )
         + cl * c5 * ty3 * ( u5lj - u5ljml )
end do

end do

do j = syl, ey3
     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) )
end do

frct(2,i,j,k) = frct(2,i,j,k)
     + dy2 * ty1 * ( ue(2,j-1)
                         - 2.0d+00 * ue(2,j)
                         + ue(2,j+1) )

frct(3,i,j,k) = frct(3,i,j,k)
     + dy3 * c3 * c4 * ( flux(2,j+1) - flux(2,j) )
     + dy4 * ty1 * ( flux(4,j+1) - flux(4,j) )
     + 2.0d+00 * ue(4,j)
     + ue(4,j+1)

frct(5,i,j,k) = frct(5,i,j,k)
     + ty3 * c3 * c4 * ( flux(5,j+1) - flux(5,j) )
     + 2.0d+00 * ue(5,j)
     + ue(5,j+1)

end do

***fourth-order dissipation

if ( gsty .eq. 1 ) then
     do m = 1, 5
         frct(m,i,syl,k) = frct(m,i,syl,k)
         - dsspm * ( 5.0d+00 * ue(m,sy1)
                        - 4.0d+00 * ue(m,sy2)
                        + ue(m,sy3) )
     end do
endif

do j = syl, ey3
     frct(1,i,j,k) = frct(1,i,j,k)
     - dsspm * ( 4.0d+00 * ue(m,j-1)
                  + 6.0d+00 * ue(m,j)
                  - 4.0d+00 * ue(m,j+1)
                  + ue(m,j+2) )
end do

end do

if ( gendy .eq. ny ) then
     do m = 1, 5
         frct(m,i,ey2,k) = frct(m,i,ey2,k)
         - dsspm * ( 4.0d+00 * ue(m,ey3)
                        + 6.0d+00 * ue(m,ey2)
                        + 6.0d+00 * ue(m,ey2) )
     end do
endif
$ \quad - 4.0d+00 * \text{ue}(m,\text{eyl})$

$\quad \text{frct}(m,i,\text{eyl},k) = \text{frct}(m,i,\text{eyl},k)$

$\quad - \text{daspm} * \{ \text{ue}(m,\text{eyl}) \}$

$\quad - 4.0d+00 * \text{ue}(m,\text{eyl})$

$\quad + 5.0d+00 * \text{ue}(m,\text{eyl})$

end do

endif

end do

do i = sx1, ex1

g = iglob(i)

xi = ( \text{dfloat}(ig-1) ) / ( \text{ny} - 1 )

do j = sy1, ey1

gj = jglob(j)

\text{eta} = ( \text{dfloat}(jg-1) ) / ( \text{ny} - 1 )

do k = sm2, ezm2

kg = kglob(k)

\text{zeta} = ( \text{dfloat}(kg-1) ) / ( \text{nz} - 1 )

do m = 1, 5

\text{ue}(m,k) = \text{ce}(m,1)

\quad + \text{ce}(m,2) * \text{xi}

\quad + \text{ce}(m,3) * \text{eta}

\quad + \text{ce}(m,4) * \text{zeta}

\quad + \text{ce}(m,5) * \text{xi} * \text{xi}

\quad + \text{ce}(m,6) * \text{eta} * \text{eta}

\quad + \text{ce}(m,7) * \text{zeta} * \text{zeta}

\quad + \text{ce}(m,8) * \text{xi} * \text{xi} * \text{xi}

\quad + \text{ce}(m,9) * \text{eta} * \text{eta} * \text{eta}

\quad + \text{ce}(m,10) * \text{zeta} * \text{zeta} * \text{zeta}

\quad + \text{ce}(m,11) * \text{xi} * \text{xi} * \text{xi}

\quad + \text{ce}(m,12) * \text{eta} * \text{eta} * \text{eta}

\quad + \text{ce}(m,13) * \text{zeta} * \text{zeta} * \text{zeta}

\text{end do}

\text{flux}(1,k) = \text{ue}(4,k)

\text{u41} = \text{ue}(4,k) / \text{ue}(1,k)

q = 0.50d+00 * ( \text{ue}(2,k) * \text{ue}(2,k)

\quad + \text{ue}(3,k) * \text{ue}(3,k)

\quad + \text{ue}(4,k) * \text{ue}(4,k)

\quad / \text{ue}(1,k)

\text{flux}(2,k) = \text{ue}(2,k) * \text{u41}

\text{flux}(3,k) = \text{ue}(3,k) * \text{u41}

\text{flux}(4,k) = \text{ue}(4,k) * \text{u41} + c2 * ( \text{ue}(5,k) - q )

\text{flux}(5,k) = ( c1 * \text{ue}(5,k) - c2 * q ) * \text{u41}

end do

do k = sz1, ez1

m = 1, 5

\text{frct}(m,i,j,k) = \text{frct}(m,i,j,k)

- tz2 * ( \text{flux}(m+1) - \text{flux}(m-1) )

\text{end do}

do k = sz1, ezm1

\text{tmp} = 1.0d+00 / \text{ue}(1,k)

\text{u1k} = \text{tmp} * \text{ue}(2,k)

\text{u3k} = \text{tmp} * \text{ue}(3,k)

\text{u4k} = \text{tmp} * \text{ue}(4,k)

\text{u5k} = \text{tmp} * \text{ue}(5,k)

\text{tmp} = 1.0d+00 / \text{ue}(1,k-1)

\text{u1km1} = \text{tmp} * \text{ue}(2,k-1)

\text{u3km1} = \text{tmp} * \text{ue}(3,k-1)

\text{u4km1} = \text{tmp} * \text{ue}(4,k-1)

\text{u5km1} = \text{tmp} * \text{ue}(5,k-1)

\text{flux}(2,k) = \text{tz3} * ( \text{u1k} - \text{u1km1} )

\text{flux}(3,k) = \text{tz3} * ( \text{u3k} - \text{u3km1} )

\text{flux}(4,k) = ( 4.0d+00/3.0d+00 ) * \text{tz3} * ( \text{u4k} - \text{u4km1} )

\text{flux}(5,k) = 0.50d+00 * ( 1.0d+00 - c1*c5 )

* \text{tz3} * ( \text{u1k} **2 + \text{u3k} **2 + \text{u4k} **2 )

* ( \text{u1km1}**2 + \text{u3km1}**2 + \text{u4km1}**2 )

+ (1.0d+00/6.0d+00)

* \text{tz3} * ( \text{u4km1}**2 - \text{u4km1}**2 )

* c1 * c5 * \text{tz3} * ( \text{u5k} - \text{u5km1} )

end do

do k = sz1, ez1

\text{frct}(1,i,j,k) = \text{frct}(1,i,j,k)

- \text{dz1} * \text{tz1} * ( \text{ue}(1,k+1) )
- 2.0d+00 * ue(1,k)
+ 2.0d+00 * ue(1,k-1)

frct(2,i,j,k) = frct(2,i,j,k)
+ tz3 * c3 * c4 * ( flux(2,k+1) - flux(2,k) )
+ dz2 * tz1 * ( u1e(2,k+1) - u1e(2,k) )
+ 2.0d+00 * u1e(2,k)
+ 2.0d+00 * u1e(2,k-1)

frct(3,i,j,k) = frct(3,i,j,k)
+ tz3 * c3 * c4 * ( flux(3,k+1) - flux(3,k) )
+ dz3 * tz1 * ( u2e(3,k+1) - u2e(3,k) )
+ 2.0d+00 * u2e(3,k)
+ 2.0d+00 * u2e(3,k-1)

frct(4,i,j,k) = frct(4,i,j,k)
+ tz4 * c3 * c4 * ( flux(4,k+1) - flux(4,k) )
+ dz4 * tz1 * ( u3e(4,k+1) - u3e(4,k) )
+ 2.0d+00 * u3e(4,k)
+ 2.0d+00 * u3e(4,k-1)

frct(5,i,j,k) = frct(5,i,j,k)
+ tz5 * c3 * c4 * ( flux(5,k+1) - flux(5,k) )
+ dz5 * tz1 * ( u4e(5,k+1) - u4e(5,k) )
+ 2.0d+00 * u4e(5,k)
+ 2.0d+00 * u4e(5,k-1)

end do

c***fourth-order dissipation

c
if ( gstz .eq. 1 ) then
  do m = 1, 5
    frct(m,i,j,szl) = frct(m,i,j,szl)
    - dspxm * ( + 5.0d+00 * ue(m,szl) - 4.0d+00 * ue(m,szl) )
    + u2e(m,szl)
  end do

c***compute the exact solution at (i,j,k)

c Author: Sisira Weeratunga
NASA Ames Research Center
(10/25/90)

include 'appsp.incl'
dimension u000i(jk)(*)

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 )
do m = 1, 5
c    u000ij(m) = ce(m,1) * xi + ce(m,2) * eta + ce(m,3) * xi * xi + ce(m,4) * zeta + ce(m,5) * eta * eta + ce(m,6) * eta * eta * eta + ce(m,7) * zeta + ce(m,8) * xi * xi * xi + ce(m,9) * eta * eta * eta + ce(m,10) * zeta * zeta + ce(m,11) * eta * xi * xi * xi + ce(m,12) * eta * eta * eta * eta + ce(m,13) * zeta * zeta * zeta * zeta
end do
return end

subroutine error(u, enorme, igstx, igendx, igsty, igendy, igstz, igendz)

*** compute the solution error

Author: Sisira Weeratunga
NASA Ames Research Center
(10/25/90)

include 'appsp.incl'

real *8 u(5, xsize, ysize, zsize)
real *8 enorms(5)

dimension imax(5), jmax(5), kmax(5), u000ijk(5), errmax(5)

integer igstx, igendx, igsty, igendy, igstz, igendz

gstx = igstx
gendx = igendx
gstY = igsty
gendy = igendy
gstz = igstz
gendz = igendz

lnorm = 2

if ( lnorm .eq. 1 ) then
    do m = 1, 5
        errmax(m) = 1.0d+20
    end do
end do

else if ( lnorm .eq. 2 ) then
    do m = 1, 5
        errrn(m) = 0.0d+00
    end do

    do k = szl, ezl
    do j = syl, eyl
    do i = szl, exl

        call exact ( iglob(i), jglob(j), kglob(k), u000ijk )
        do m = 1, 5
            tmp = abs ( u000ijk(m) - u(m,i,j,k) )
            if ( tmp .gt. errmax(m) ) then
                errmax(m) = tmp
                imax(m) = iglob(i)
                jmax(m) = jglob(j)
                kmax(m) = kglob(k)
            end if
        end do
    end do
end do

write (iout,1001) ( errmax(m), nmax(m) )
delete (nmax, m = 1, 5 )

end do

end do
end do

end do

end do

end do

write (iout,1001) ( errrn(m), nmax(m) )
delete (nmax, m = 1, 5 )

else if ( lnorm .eq. 2 ) then
    do m = 1, 5
        errrn(m) = 0.0d+00
    end do

    do k = szl, ezl
    do j = syl, eyl
    do i = szl, exl

        call exact ( iglob(i), jglob(j), kglob(k), u000ijk )
        do m = 1, 5
            tmp = ( u000ijk(m) - u(m,i,j,k) )
            errrn(m) = errrn(m) + tmp ** 2
        end do
    end do
end do

end do

end do

end do

errrn(m) = sqrt ( errrn(m) / ( nx-2 )*( ny-2 )*( nz-2 ) )
c end do

  c write (iout,1002) ( errnm(m), m = 1, 5 )
  c do m = 1, 5
    errnm(m) = errnm(m) + errnm(m)
  c end do
  c end if

1001 format (/5x,'max. error in soln. to first pde =',1pe12.4/, 
  $/5x,' and its location = (',i4,'i4','i4','i4,')'/, 
  $/5x,'max. error in soln. to second pde =',1pe12.4/, 
  $/5x,' and its location = (',i4,'i4','i4','i4,')'/, 
  $/5x,'max. error in soln. to third pde =',1pe12.4/, 
  $/5x,' and its location = (',i4,'i4','i4','i4,')'/, 
  $/5x,'max. error in soln. to fifth pde =',1pe12.4/, 
  $/5x,' and its location = (',i4,'i4','i4','i4,')'/)

1002 format (1x/1x,'RMS-norm of error in soln. to ', 
  $ 'first pde = ',1pe12.5/, 
  $ 'second pde = ',1pe12.5/, 
  $ 'third pde = ',1pe12.5/, 
  $ 'fourth pde = ',1pe12.5/, 
  $ 'fifth pde = ',1pe12.5)
  return

end

subroutine jacc ( m, u,a,b,c,d,e )
***form the xi-direction pentadiagonal system

Author: Sisira Weeratunga
NASA Ames Research Center
(10/25/90)

include 'appsp.incl'

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)

dimension cv(xsize), aa(xsize), rhon(xsize)

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
  else if ( m .eq. 4 ) then
    c sn = 1.0d+00
  else if ( m .eq. 5 ) then
    c sn = - 1.0d+00
  c end if
  c do k = szl, ezl
    c do j = syl, eyl
      c do i = sxm, exm
        c
    c rul = 1.0d+00 / u(1,i,j,k)
    u = rul * u(2,i,j,k)
    v = rul * u(3,i,j,k)
    w = rul * u(4,i,j,k)
    q = 0.50d+00 * ( uu ** 2 + vv ** 2 + ww ** 2 )
    c
    cv(i) = uu
    aa(i) = sqrt ( c1 * c2 * ( rul * u(5,i,j,k) - q ) )
    c
    rhon(i) = max ( dx1, 
      $ dx2 + r43 * c34 * rul, 
      $ dx3 + c34 * rul, 
      $ dx4 + c34 * rul, 
      $ dx5 + c1345 * rul )
  c end do
  c

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
a(i,j,k) = 0.0d+00
b(i,j,k) = - dt * tx2 + ( cv(i-1) + sn * aa(i-1) )
c(i,j,k) = 1.0d+00
$ + dt * rhon(i-1) * tx1$
d(i,j,k) = - dt * tx2 + ( cv(i+1) + sn * aa(i+1) )
e(i,j,k) = 0.0d+00
end do

if ( gendx .eq. nx ) then
a(ex1,j,k) = 0.0d+00
b(ex1,j,k) = 0.0d+00
c(ex1,j,k) = 1.0d+00
d(ex1,j,k) = 0.0d+00
e(ex1,j,k) = 0.0d+00
endif

**** fourth order dissipation

if ( gstdx .eq. 1 ) then
b(sx1,j,k) = b(sx1,j,k) + dt * dssp * ( + 5.0d+00 )
c(sx1,j,k) = c(sx1,j,k) + dt * dssp * ( + 1.0d+00 )
d(sx1,j,k) = d(sx1,j,k) + dt * dssp * ( + 4.0d+00 )
e(sx1,j,k) = e(sx1,j,k) + dt * dssp * ( + 4.0d+00 )
else
b(sx-1,j,k) = b(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

begin = sx2
if ( gstdx .eq. 1 ) begin = sx + 3
end = exn2
if ( gendx .eq. nx ) end = ex - 3

do i = sx3, ex3

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 * ( + 5.0d+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 )
end do

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 )
end if

subroutine jacy ( m, u,a,b,c,d,e )

**form the eta-direction pentadiagonal system.

Author: Sisira Weeratunga

NASA Ames Research Center

(10/25/90)

include 'appsp.incl'

real *8 u(5,xsize,ysize), a(xsize,ysize,ysize),
$ b(xsize,ysize,ysize), c(xsize,ysize,ysize),$
$ d(xsize,ysize,ysize), e(xsize,ysize,ysize)

dimension cv(ysize), aa(ysize), rhoq(ysize)

r43 = 4.0d+00 / 3.0d+00
c4 = c3 * c4
c1345 = c1 * c3 * c4 * c5

if ( m .eq. 3 ) then
sn = 0.0d+00
else if ( m .eq. 4 ) then
sn = 1.0d+00
else if ( m .eq. 5 ) then
sn = - 1.0d+00
end if

do k = szl, ezl

do i = sx1, exl

do j = sym1, eyml
```c
rul = 1.0d+00 / u(1,i,j,k)
uu = rul * u(2,i,j,k)
vv = rul * u(3,i,j,k)
ww = rul * u(4,i,j,k)

c
q = 0.50d+00 * ( uu ** 2 + vv ** 2 + ww ** 2 )

c
rvj = vv
aa(j) = sqrt ( c1 * c2 * ( rul * u(5,i,j,k) - q ) )
rhoq(j) = max ( dy1,
  dy2 + c34 * rul,
  dy3 + r43 * c34 * rul,
  dy4 + c34 * rul,
  dy5 + c1345 * rul )

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
end if

c
begin = sym2
if ( gsty .eq. 1 ) begin = sy + 1
endif

c
dj = sy1, ey1

a(i, j, k) = 0.0d+00
b(i, j, k) = - dt * ty2 * ( cv(j-1) + sn * aa(j-1) )
c(i, j, k) = 1.0d+00
d(i, j, k) = dt * ty2 * ( cv(j+1) + sn * aa(j+1) )
e(i, j, k) = 0.0d+00

 amort do
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

end do

subroutine jac ( m, u,a,b,c,d,e )
```
c***form the zeta-direction pentadiagonal system.
c
c Author: Sisira Weeratunga
c NASA Ames Research Center
c (10/25/90)
c
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)

dimension cv(zsize), aa(zsize), rhos(zsize)
c
r43 = 4.0d+00 / 3.0d+00
c34 = c3 * c4
cl345 = cl1 * c3 * c4 * c5

c if (m .eq. 3) then
sn = 0.0d+00
c
else if (m .eq. 4) then
sn = 1.0d+00

c else if (m .eq. 5) then
sn = -1.0d+00
c
c end if

do j = syl, eyl
do i = sxl, exl
do k = szl1, ezl1
do

r1 = 1.0d+00 / u(1,1,1,1,1)
uu = r1 * u(2,1,1,1,1)
v = r1 * u(3,1,1,1,1)
w = r1 * u(4,1,1,1,1)
c
q = 0.50d+00 * ( uu ** 2
$ $ + vv ** 2
$ $ + ww ** 2 )
c
cv(k) = w
aa(k) = sqrt ( c1 * c2 * ( r1 * u(5,1,1,1,1) - q ) )
c
rhos(k) = max ( dz1,
$ $ dz2 + c34 * r1,
$ $ dz3 + c34 * r1,
$ $ dz4 + r43 * c34 * r1,
$ $ dz5 + cl345 * r1 )

c if (gtsz .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
f(i,j,sz-1) = 0.0d+00
e(i,j,sz-2) = 0.0d+00
endif
c
c begin = szm2
c if (gtsz .eq. 1) begin = sz + 1
c end = ezm2
c if (gendz .eq. nz) end = ez - 1
c
do k = szl, ezl
do

a(i,j,k) = 0.0d+00
b(i,j,k) = -dt * tz1 * ( cv(k-1) + sn * aa(k-1) )
$ c(i,j,k) = 1.0d+00
$ d(i,j,k) = dt * tz1 * ( cv(k-1) + sn * aa(k-1) )
$ c e(i,j,k) = 0.0d+00

do c elseif

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

***fourth order dissipation
c

c if (gtsz .eq. 1) then
a(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
f(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
if ( gestz .eq. 1 ) begin = sz + 3
end = ezm2
if ( gendz .eq. nz ) end = ez - 3
do k = sz3, ez3
        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 )
end do
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 )
e(i,j,ez2) = e(i,j,ez2) + dt * dssp * ( + 5.0d+00 )
endif
end do
return
end

subroutine ninvr(rsd)

***block-diagonal matrix-vector multiply

Author: Sisira Weeratunga
NASA Ames Research Center
(10/25/90)

include 'appsp.incl'

real *8 rsd(5,xsize,ysize,zsize)
bt = sqrt ( 0.50d+00 )
do k = szl, ezl
do j = syl, eyl
do i = sxl, exl

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)

rsd(1,i,j,k) = - r2
rsd(2,i,j,k) = r1
rsd(3,i,j,k) = bt * ( r4 - r5 )
t1 = bt * r3
t2 = 0.50d+00 * ( r4 + r5 )
rsd(4,i,j,k) = - t1 * t2
rsd(5,i,j,k) = t1 * t2
end do
end do
return
end

subroutine getlz(u, philz)

***compute the surface integral

include 'appsp.incl'

real *8 u(5,xsize,ysize,zsize)
dimension philz(xsize,ysize)
do j = syl, ey2
do i = sxl, ex1

philz(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) )
end do
end do
subroutine get2z(u, phi2z)

***compute the surface integral

include 'appsp.inc'

real *8 u5,xsize,ysize,zsize)

dimension phi2z(xsize,ysize)

do j = sy1, ey2

do i = sx1, ex1

phi2z(i,j) = c2*( u5(i,j,ez1) ** 2
$ - 0.50d+00 * ( u2(i,j,ezl) ** 2
$ + u5(i,j,ezl) ** 2 )
$ + u4(i,j,ezl) ** 2 )
$ / (u1,i,j,ezl) )

end do

end do

return

end

subroutine get2y(u, phi2y)

***compute the surface integral

include 'appsp.inc'

real *8 u5,xsize,ysize,zsize)

dimension phi2y(xsize,ysize)

do k = sz2, ez1

do i = sx1, ex1

phi2y(i,k) = c2*( u5(i,ey2,k)
$ - 0.50d+00 * ( u2(i,ey2,k) ** 2
$ + u3(i,ey2,k) ** 2
$ + u5(i,ey2,k) ** 2 )
$ / (u1,i,ey2,k) )

end do

end do

return

end

subroutine get1x(u, philx)

***compute the surface integral

include 'appsp.inc'

real *8 u5,xsize,ysize,zsize)

dimension philx(ysize,zsize)

do k = sz2, ez1

do j = sy1, ey2

philx(j,k) = c2*( u5(sx1,j,k)
$ - 0.50d+00 * ( u2(sx1,j,k) ** 2
$ + u3(sx1,j,k) ** 2
$ + u5(sx1,j,k) ** 2 )
$ / (u1,sx1,j,k) )

end do

end do
c
return
cend
c
subroutine get2x(u, phi2x)
c***compute the surface integral
cinclude 'appsp.incl'
creal *8 u(5,xsize,ysize,zsize)
cdimension phi2x(ysize,zsize)
do k = sz2, ez1
do j = syl, ey2

phi2x(j,k) = c2*( u(5,exl,j,k) - 0.50d+00 * ( u(2,exl,j,k) ** 2 + u(3,exl,j,k) ** 2 + u(4,exl,j,k) ** 2 ) / u(1,exl,j,k) )
end do
doen
return
cend
c
subroutine cpphiy(phi, data, iii, kkk)
c***compute the surface integral
cinclude 'appsp.incl'
creal *8 phi(isiz1,isiz3), data(xsize,zsize)
do k = 1, zsize-4
do i = 1, xsize-4
phi(i+iii,k+kkk) = data(i+2,k+2)
end do
doen
return
cend

subroutine cpphiy(phi, data, iii, jjj)
c***compute the surface integral
cinclude 'appsp.incl'
creal *8 phi(isiz1,isiz3), data(xsize,ysize)
do j = 1, ysize-4
do i = 1, xsize-4
phi(i+iii,j+jjj) = data(i+2,j+2)
end do
doen
return
cend

subroutine cpphiy(phi, data, jjj, kkk)
c***compute the surface integral
cinclude 'appsp.incl'
creal *8 phi(isiz2,isiz3), data(ysize,zsize)
do k = 1, zsize-4
do j = 1, ysize-4
phi(j+jjj,k+kkk) = data(j+2,k+2)
end do
doen
return
cend

subroutine gphiy(phi)
c***compute the global surface integral
cinclude 'appsp.incl'
real *8 phi2z(isiz1,isiz2)
do j = jil, ji2-1
do i = il1, ii2-1
    frc1 = frc1 + ( phi2z(i,j) $ + phi2z(i+1,j) $ + phi2z(i,j+1) $ + phi2z(i+1,j+1) )
end do
end do
return
end

subroutine gphi2z(phi2z)

***compute the global surface integral
include 'appsp.incl'
real *8 phi2z(isiz1,isiz2)
do j = jil, ji2-1
do i = il1, ii2-1
    frc1 = frc1 + ( phi2z(i,j) $ + phi2z(i+1,j) $ + phi2z(i,j+1) $ + phi2z(i+1,j+1) )
end do
end do
return
end

subroutine gphi2y(phi2y)

***compute the global surface integral
include 'appsp.incl'
real *8 phi2y(isiz1,isiz3)
do k = k11, k12-1
do i = il1, ii2-1
    frc2 = frc2 + ( phi2y(i,k) $ + phi2y(i+1,k) $ + phi2y(i,k+1) $ + phi2y(i+1,k+1) )
end do
end do
return
end

subroutine gphi2x(phi2x)

***compute the global surface integral
include 'appsp.incl'
real *8 phi2x(isiz2,isiz3)
do j = jil, jil2-1
do k = k11, k12-1
SUBROUTINE PHI2X PHI2X

SUBROUTINE GPHI2X PHI2X

***Compute the global surface integral

INCLUDE 'APPS.INCL'

REAL *8 PHI2X(ISIZ2,ISIZ3)

DO K = K11, K12-1
   DO J = J11, J12-1
      FRC3 = FRC3 + ( PHI2X(J,K)
                    + PHI2X(J+1,K)
                    + PHI2X(J,K+1)
                    + PHI2X(J+1,K+1) )
   END DO
   END DO
RETURN
END

SUBROUTINE GINTGR()

***Compute the global surface integral

INCLUDE 'APPS.INCL'

FRC1 = DXI * DETA * FRC1
FRC2 = DXI * DZETA * FRC2
FRC3 = DETA * DZETA * FRC3

FRC = 0.25D00 * ( FRC1 + FRC2 + FRC3 )

WRITE (IOUT,1001) FRC

RETURN
END

1001 FORMAT ('/X, 'SURFACE INTEGRAL = ',LPE12.5//)
SUBROUTINE PRED(FLX)
  REAL *8 FLUX(5,XSIZE,YSIZE,SIZE), RED(5,XSIZE,YSIZE,SIZE)
  INTEGER I,J,K,NI,NIK
  REAL *8 FCRX(5,XSIZE,YSIZE), FCRY(1,1,1,1)

  CALL PRD(FLX,FLX)
  CALL PRD(FLX,FLY)
  CALL PRD(FLX,FLZ)
  CALL PRD(FLX,FLX)
  CALL PRD(FLX,FLX)

  RETURN
END

SUBROUTINE PRD(FLX,RED)
  REAL *8 FLX(5,XSIZE,YSIZE,SIZE), RED(5,XSIZE,YSIZE,SIZE)

  DO I = 1, XSIZE
    DO J = 1, YSIZE
      DO K = 1, SIZE
        RED(1,I,J,K) = 0.0
        RED(2,I,J,K) = 0.0
        RED(3,I,J,K) = 0.0
        RED(4,I,J,K) = 0.0
        RED(5,I,J,K) = 0.0
      END DO
    END DO
  END DO

  RETURN
END
do m = 1, 5
    rsd(m,i,j,k) = rsd(m,i,j,k)
    $ + dx1 * tx1 * ( u(1,i,j,k) - 2.0d+00 * u(1,i,j,k) )
end do

end do

do i = sx1, ex1
    $ + c1 * c5 * tx3 * ( u51 - u51im )
end do

rsd(1,i,j,k) = rsd(1,i,j,k)
$ + dx1 * tx1 * ( u(1,i-1,j,k) - 2.0d+00 * u(1,i-1,j,k) )

rsd(2,i,j,k) = rsd(2,i,j,k)
$ + dx2 * tx1 * ( u(2,i,j,k) - 2.0d+00 * u(2,i,j,k) )

rsd(3,i,j,k) = rsd(3,i,j,k)
$ + dx3 * tx1 * ( u(3,i,j,k) - 2.0d+00 * u(3,i,j,k) )
end do

c

rsd(4,i,j,k) = rsd(4,i,j,k)
$ + tx1 * c3 * c4 * ( flux(4,i+1) - flux(4,i) )
$ + tx1 * c3 * c4 * ( flux(4,i+1) - flux(4,i) )

rsd(5,i,j,k) = rsd(5,i,j,k)
$ + dx5 * tx1 * ( u(5,i-1,j,k) - 2.0d+00 * u(5,i-1,j,k) )

write(iout,*'RSD', rsd(2,4,4,8)

***Fourth-order dissipation

if ( qstx .eq. 1 ) then
    do m = 1, 5
        rsd(m,sx1,j,k) = rsd(m,sx1,j,k)
        $ - dsx * ( + 5.0d+00 * u(m,sx1,j,k) - 4.0d+00 * u(m,sx2,j,k) + u(m,sx3,j,k) )
    end do
endif

do i = sx3, ex3
    rsd(m,i,j,k) = rsd(m,i,j,k)
$ - dsx * ( + 6.0d+00 * u(m,i-1,j,k) + u(m,i+1,j,k) )
end do

if ( gendx .eq. nx ) then
    do m = 1, 5
        rsd(m,ex2,j,k) = rsd(m,ex2,j,k)
$ - dsx * ( + u(m,ex-4,j,k) )
end do

$- 4.0d+00 * u(m,ex3,j,k) \\
+ 6.0d+00 * u(m,ex2,j,k) \\
- 4.0d+00 * u(m,ex1,j,k) \\
\$

end do
endif
end do
end do
return
end

subroutine rhsy(u,rsd,frct,igstx, igendx, igsty, igendy, igstz, igendz)

***compute the right hand sides

C Author: Sisira Weeratunga
C NASA Ames Research Center
C (10/25/90)

C include 'appsp.incl'

real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize), 
frct(5,xsize,ysize,zsize)

dimension flux(5,ysize)

integer igstx, igendx, igsty, igendy, igstz, igendz

gstx = igstx
gendx = igendx
gstv = igstv
gendy = igendy
gstz = igstz
gendz = igendz

***eta-direction flux differences

do k = szl, exl

do i = sxl, exl

do j = sym1, eyml

flux(1,j) = u(3,i,j,k)

u31 = u(3,i,j,k) / u(1,i,j,k)

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) \\

flux(2,j) = u(2,i,j,k) * u31

flux(3,j) = u(3,i,j,k) * u31 + c2 * ( u(5,i,j,k) - q )

flux(4,j) = u(4,i,j,k) * u31

flux(5,j) = ( c1 * u(5,i,j,k) - c2 * q ) * u31

end do

end do

do j = sym1, eyml

do m = 1, 5

rsd(m,i,j,k) = rsd(m,i,j,k) \\
- ty2 * ( flux(m,j+1) - flux(m,j-1) )

end do

end do

end do
do j = syl, eyl
   rsd(1,i,j,k) = rsd(1,i,j,k)
   + dy1 * tyl * ( u(1,i,j-1,k) - 2.0d+00 * u(1,i,j,k) + u(1,i,j+1,k) )
   rsd(2,i,j,k) = rsd(2,i,j,k)
   + ty3 * c3 * c4 * ( flux(2,j+1) - flux(2,j) )
   + dy2 * tyl * ( u(2,i,j-1,k) - 2.0d+00 * u(2,i,j,k) + u(2,i,j+1,k) )
   rsd(3,i,j,k) = rsd(3,i,j,k)
   + ty3 * c3 * c4 * ( flux(3,j+1) - flux(3,j) )
   + dy3 * tyl * ( u(3,i,j-1,k) - 2.0d+00 * u(3,i,j,k) + u(3,i,j+1,k) )
   rsd(4,i,j,k) = rsd(4,i,j,k)
   + ty3 * c3 * c4 * ( flux(4,j+1) - flux(4,j) )
   + dy4 * tyl * ( u(4,i,j-1,k) - 2.0d+00 * u(4,i,j,k) + u(4,i,j+1,k) )
   rsd(5,i,j,k) = rsd(5,i,j,k)
   + ty3 * c3 * c4 * ( flux(5,j+1) - flux(5,j) )
   + dy5 * tyl * ( u(5,i,j-1,k) - 2.0d+00 * u(5,i,j,k) + u(5,i,j+1,k) )
end do

!!fourth-order dissipation
if ( gsty .eq. 1 ) then
   do m = 1, 5
      rsd(m,i,j,k) = rsd(m,i,j,k)
      - dssp * ( + 5.0d+00 * u(m,i,syl,k) - 4.0d+00 * u(m,i,sy2,k) + u(m,i,sy3,k) )
   end do
endif

!!compute the right hand sides

subroutine rhsz(u,rsd,frct,igtx, igndx, igsty, igendy, igstz, igendz)
   ! 
   ! Author: Sisira Weeratunga
   ! NASA Ames Research Center
   ! (10/25/90)
   !
   ! include 'appsp.incl'
   !
   ! real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize),
   ! frct(5,xsize,ysize,zsize)
   !
   ! dimension flux(5,zsize)
   ! integer igtx, igndx, igsty, igendy, igstz, igendz

end subroutine rhsz
c gatx = igatx
gendx = igendx
gsty = igsty
gendy = igendy
gatx = igatx
gendz = igendz

**zeta-direction flux differences**
c do j = syl, eyl
  do i = sx1, ex1
    do k = sz1, ezml
      flux(1,k) = u(4,i,j,k)
      u41 = u(4,i,j,k) / u(1,i,j,k)
      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) )
      flux(2,k) = u(2,i,j,k) * u41
      flux(3,k) = u(3,i,j,k) * u41
      flux(4,k) = u(4,i,j,k) * u41 + c2 * ( u(5,i,j,k) - q )
      flux(5,k) = ( c1 * u(5,i,j,k) - c2 * q ) * u41
    end do
  end do k = sz1, ezml
  do m = 1, 5
    rsd(m,i,j,k) = rsd(m,i,j,k) - tz2 * ( flux(m,k+1) - flux(m,k-1) )
  end do
end do

end do

c do k = sz1, ezml
  tmp = 1.0d+00 / u(1,i,j,k)
  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)
  tmp = 1.0d+00 / u(1,i,j,k-1)
end do

c****fourth-order dissipation
c if ( gatx .eq. 1 ) then
  do m = 1, 5
    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)
    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 )
    flux(5,k) = ( u21k **2 + u31k **2 + u41k **2 )
    - ( u21km1 **2 + u31km1 **2 + u41km1 **2 )
    + (1.0d+00/6.0d+00)
    + ( u21km1 **2 + u31km1 **2 + u41km1 **2 )
    + c1 * c5 * tz3 * ( u51k - u51km1 )
  end do
end if
$$\text{rsd}(m,i,j,\text{sz1}) = \text{rsd}(m,i,j,\text{sz1})$$
$$\text{rsd}(m,i,j,\text{sz2}) = \text{rsd}(m,i,j,\text{sz2})$$
$$\text{rsd}(m,i,j,k) = \text{rsd}(m,i,j,k)$$
$$\text{rsd}(m,i,j,\text{ez2}) = \text{rsd}(m,i,j,\text{ez2})$$
$$\text{rsd}(m,i,j,\text{ez1}) = \text{rsd}(m,i,j,\text{ez1})$$

```c
rsd(m,i,j,sz1) = rsd(m,i,j,sz1)
    - dssp * ( + 5.0d00 * u(m,i,j,sz1)
        - 4.0d00 * u(m,i,j,sz2)
        + u(m,i,j,sz3) )
rsd(m,i,j,sz2) = rsd(m,i,j,sz2)
    - dssp * ( - 4.0d00 * u(m,i,j,sz1)
        + 6.0d00 * u(m,i,j,sz2)
        - 4.0d00 * u(m,i,j,sz3)
        + u(m,i,j,sz+4) )
end do
endif
do k = sz3, ez3
do m = 1, 5
    rsd(m,i,j,k) = rsd(m,i,j,k)
        - dssp * ( - 4.0d00 * u(m,i,j,k-2)
            - 6.0d00 * u(m,i,j,k-1)
            + 4.0d00 * u(m,i,j,k+1)
            + u(m,i,j,k+2) )
end do
do k = sz1, ez1
do j = syl, ey1
    if ( gxst .eq. 1 ) then
        tmp1 = b(sxl,j,k) / c(sx,j,k)
        c(sxl,j,k) = c(sxl,j,k) - tmp1 * d(sx,j,k)
        d(sxl,j,k) = d(sxl,j,k) - tmp1 * e(sx,j,k)
        f(m,sxl,j,k) = f(m,sxl,j,k) - tmp1 * f(m,sx,j,k)
    endif
do i = sx2, ex
    tmp2 = a(i,j,k) / c(i-2,j,k)
    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)
    tmp1 = b(i,j,k) / c(i-1,j,k)
    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)
end do
end do
end do
return
end
```
subroutine bspentax (m, a, b, c, d, e, f)

***solution of multiple, independent systems of penta-diagonal systems
using Gaussian elimination (without pivoting) algorithm

Author: Sisira Weeratunga
NASA Ames Research Center
(10/25/90)

include 'appsp.incl'

real *8 f(m.xsize,ysize,zi.size), a(xsize,ysize,zi.size),
$ b(xsize,ysize,zi.size), c(xsize,ysize,zi.size),
$ d(xsize,ysize,zi.size), e(xsize,ysize,zi.size)

***forward elimination

do k = 1, z1

do j = 1, y1

if (gendj .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

do i = ex2, nx - 1

f(m.i.j,k) = ( f(m.i.j,k) - d(i,j,k)*f(m.i+1,j,k) ) / c(i,j,k)
end do

end do

return
end

subroutine bspentax3 ( a, b, c, d, e, f)

***solution of multiple, independent systems of penta-diagonal systems
using Gaussian elimination algorithm

Author: Sisira Weeratunga
NASA Ames Research Center
(10/25/90)

include 'appsp.incl'

real *8 f(5,xsize,ysize,zi.size), a(5,xsize,ysize,zi.size),
$ b(5,xsize,ysize,zi.size), c(5,xsize,ysize,zi.size),
$ d(5,xsize,ysize,zi.size), e(5,xsize,ysize,zi.size)

***forward elimination

do k = 1, z1

do j = 1, y1

if (gstx .eq. 1) then
tmpl1 = b(sx1,j,k) / c(sx.j,k)

c(sx1.j,k) = c(sx1.j,k) - tmpl1 * d(sx.j,k)
d(sx1.j,k) = d(sx1.j,k) - tmpl1 * e(sx.j,k)
f(1,sx1.j,k) = f(1,sx1.j,k) - tmpl1 * f(1,sx.j,k)
f(2,sx1.j,k) = f(2,sx1.j,k) - tmpl1 * f(2,sx.j,k)
f(3,sx1.j,k) = f(3,sx1.j,k) - tmpl1 * f(3,sx.j,k)
endif

do i = sx2, ex

tmpl2 = b(i,j,k) / c(i-2,j,k)

b(i,j,k) = b(i,j,k) - tmpl2 * d(i-2,j,k)
c(i,j,k) = c(i,j,k) - tmpl2 * e(i-2,j,k)
f(1,i,j,k) = f(1,i,j,k) - tmpl2 * f(1,i-2,j,k)
f(2,i,j,k) = f(2,i,j,k) - tmpl2 * f(2,i-2,j,k)
f(3,i,j,k) = f(3,i,j,k) - tmpl2 * f(3,i-2,j,k)

c(i,j,k) = c(i,j,k) - tmpl1 * d(i-1,j,k)
d(i,j,k) = d(i,j,k) - tmpl1 * e(i-1,j,k)
f(1,i,j,k) = f(1,i,j,k) - tmpl1 * f(1,i-1,j,k)
f(2,i,j,k) = f(2,i,j,k) - tmpl1 * f(2,i-1,j,k)
f(3,i,j,k) = f(3,i,j,k) - tmpl1 * f(3,i-1,j,k)

end do

end do

return
end

subroutine bspentax3 ( a, b, c, d, e, f)

***solution of multiple, independent systems of penta-diagonal systems
using Gaussian elimination algorithm

Author: Sisira Weeratunga
**using Gaussian elimination algorithm**

Author: Sisira Weeratunga
NASA Ames Research Center

```
**include 'appsp.incl'
**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)

***back-substitution phase

do k = szl, ex1

do j = syl, ey1
    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) )
    enddo
    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) )
    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) )
    end i = ex2, sx, -1
    do i = 1, i+1, j, k
        f(1,i,j,k) = ( f(1,i,j,k) - d(i,j,k)*f(1,i+1,j,k) ) / c(i,j,k)
        f(2,i,j,k) = ( f(2,i,j,k) - d(i,j,k)*f(2,i+1,j,k) ) / c(i,j,k)
        f(3,i,j,k) = ( f(3,i,j,k) - d(i,j,k)*f(3,i+1,j,k) ) / c(i,j,k)
    enddo
    end j = sy2, ey
    tmp1 = b(i,syl,k) / c(i,sy,k)
    c(i,syl,k) = c(i,syl,k) - tmp1 * d(i, sy,k)
    d(i, syl,k) = d(i, syl,k) - tmp1 * e(i, sy,k)
    f(m, i, syl,k) = f(m, i, syl,k) - tmp1 * f(m, i, sy,k)
    endif
    do j = sy2, ey
        tmp2 = a(i,j,k) / c(i,j-2,k)
        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)
        tmpl = b(i,j,k) / c(i,j-1,k)
        c(i,j,k) = c(i,j,k) - tmpl * d(i,j-1,k)
        d(i,j,k) = d(i,j,k) - tmpl * e(i,j-1,k)
        f(m, i, j,k) = f(m, i, j,k) - tmpl * f(m, i, j-1,k)
    enddo
    enddo
    return
```

**solution of multiple, independent systems of penta-diagonal systems**

```
**using Gaussian elimination algorithm**

```

**subroutine bspentay ( m, a, b, c, d, e, f )

```
 Aug 27 11:59 1996 appart.f Page 79

c Author: Sisira Weeratunga
 c NASA Ames Research Center
 (10/25/90)

c include 'appsp.incl'

real *8 f5(xsize,ysize,zsize), a(xsize,ysize,zsize),
 b(xsize,ysize,zsize), c(xsize,ysize,zsize),
 d(xsize,ysize,zsize), e(xsize,ysize,zsize)

c***back-substitution phase

do k = szl, ezl

do i = sxl, exl

if (gndy.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

do j = ey2, sy, -1
f(m,i,j,k) = ( f(m,i,j,k) - d(i,j,k)*f(m,i,j+1,k) ) / c(i,j,k)
end do

return
end

subroutine spentay3 (a, b, c, d, e, f)

c***solution of multiple, independent systems of penta-diagonal systems
 using Gaussian elimination algorithm

c Author: Sisira Weeratunga
 NASA Ames Research Center
 (10/25/90)

include 'appsp.incl'

real *8 f5(xsize,ysize,zsize), a(xsize,ysize,zsize),
 b(xsize,ysize,zsize), c(xsize,ysize,zsize),
 d(xsize,ysize,zsize), e(xsize,ysize,zsize)

c***forward elimination

do k = szl, ezl

do i = sxl, exl

if (gndy.eq. 1) then
	tmp1 = b(i,syl,k) / c(i,sy,k)
	d(i,syl,k) = d(i,syl,k) - tmp1 * e(i,sy,k)
	f(1,i,syl,k) = f(1,i,syl,k) - tmp1 * f(1,i,sy,k)
	f(2,i,syl,k) = f(2,i,syl,k) - tmp1 * f(2,i,sy,k)
	f(3,i,syl,k) = f(3,i,syl,k) - tmp1 * f(3,i,sy,k)
endif

do j = sy2, ey

tmp2 = a(i,j,k) / c(i,j-2,k)

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)

tmp1 = b(i,j,k) / c(i,j-1,k)

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)

end do

return
end

subroutine spentay3 (a, b, c, d, e, f)

c***solution of multiple, independent systems of penta-diagonal systems
 using Gaussian elimination algorithm

c Author: Sisira Weeratunga
 NASA Ames Research Center
 (10/25/90)

include 'appsp.incl'
Aug 27 11:59 1996  appsp.f Page 81

c real *8 f5(xsize,ysize,zsize), a(xsize,ysize,zsize),
$ b(xsize,ysize,zsize), c(xsize,ysize,zsize),
$ d(xsize,ysize,zsize), e(xsize,ysize,zsize)

**back-substitution phase**

do k = szl, ex1

do i = sx1, ex1
    if ( gendy.eq. ny ) then
        f1(i,i,ey,k) = f1(i,i,ey,k) / c(i,i,k)
        f1(i,ey-1,k) = ( f1(i,ey-1,k) - d(i,ey-1,k)*f1(i,i,ey,k) ) / c(i,ey-1,k)
    endif
    f2(i,ey,k) = f2(i,ey,k) / c(i,i,k)
    f2(i,ey-1,k) = ( f2(i,ey-1,k) - d(i,ey-1,k)*f2(i,i,ey,k) ) / c(i,ey-1,k)
    f3(i,i,ey,k) = f3(i,i,ey,k) / c(i,i,k)
    f3(i,ey-1,k) = ( f3(i,ey-1,k) - d(i,ey-1,k)*f3(i,i,ey,k) ) / c(i,ey-1,k)
    do j = ey2, sy, -1
        f1(i,i,j,k) = ( f1(i,i,j,k) - d(i,j,k)*f1(i,i,j+1,k)
            - e(i,j,k)*f1(i,i,j+2,k) ) / c(i,i,j)
        f2(i,i,j,k) = ( f2(i,i,j,k) - d(i,j,k)*f2(i,i,j+1,k)
            - e(i,j,k)*f2(i,i,j+2,k) ) / c(i,i,j)
        f3(i,i,j,k) = ( f3(i,i,j,k) - d(i,j,k)*f3(i,i,j+1,k)
            - e(i,j,k)*f3(i,i,j+2,k) ) / c(i,i,j)
    enddo
    enddo
return

**solution of multiple, independent systems of pentadiagonal systems**

use Gaussian elimination algorithm

Author: Sisira Weeratunga
NASA Ames Research Center

Aug 27 11:59 1996  appsp.f Page 82

(10/25/90)

include 'appsp.incl'

real *8 f5(xsize,ysize,zsize), a(xsize,ysize,zsize),
$ b(xsize,ysize,zsize), c(xsize,ysize,zsize),
$ d(xsize,ysize,zsize), e(xsize,ysize,zsize)

**forward elimination**

do j = sy1, ey1
    do i = sx1, ex1
        if ( gatz.eq. 1 ) then
            tmp1 = b(i,j,sz1) / c(i,j,sz)
            c(i,j,sz) = c(i,j,sz) - tmp1 * d(i,j,sz)
            d(i,j,sz) = d(i,j,sz) - tmp1 * e(i,j,sz)
            f(m1,i,j,sz) = f(m1,i,j,sz) - tmp1 * f(m1,i,j,sz)
        endif
        tmp2 = a(i,j,k) / c(i,j,k-2)
        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(m1,i,j,k) = f(m1,i,j,k) - tmp2 * f(m1,i,j,k-2)
        tmp1 = b(i,j,k) / c(i,j,k-1)
        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(m1,i,j,k) = f(m1,i,j,k) - tmp1 * f(m1,i,j,k-1)
    enddo
    enddo
return

subroutine bspentaz ( m, a, b, c, d, e, f )
Aug 27 11:59 1996  appsp.f  Page 83

```c
include 'appsp.incl'

real *8 f5,xsize,ysize,zsize), a(xsize,ysize,zsize),
  b(xsize,ysize,zsize), c(xsize,ysize,zsize),
  d(xsize,ysize,zsize), e(xsize,ysize,zsize)

***back-substitution phase

do j = syl, ey1
  do i = sx1, ex1
    if ( gendz .eq. nz ) then
      f(m,i,j) = f(m,i,j) / c(i,j)
      f(m,i,j) = (f(m,i,j) - d(i,j) * f(m,i,j) + c(i,j)) / c(i,j)
    end if
    do k = e2, sz, -1
      f(m,i,j,k) = (f(m,i,j,k) - d(i,j,k) * f(m,i,j,k) + c(i,j,k)) / c(i,j,k)
    end do
  end do

return
```

Aug 27 11:59 1996  appsp.f  Page 84

```c
do i = sx1, ex1

if ( gstz .eq. 1 ) then
  tmpl = b(i,j,sz) / c(i,j)
  c(i,j,sz) = c(i,j,sz) - tmpl * d(i,j,sz)
  d(i,j,sz) = d(i,j,sz) - tmpl * e(i,j,sz)
  f(1,i,j,sz) = f(1,i,j,sz) - tmpl * f(1,i,j,sz)
  f(2,i,j,sz) = f(2,i,j,sz) - tmpl * f(2,i,j,sz)
  f(3,i,j,sz) = f(3,i,j,sz) - tmpl * f(3,i,j,sz)
endif

do k = sz2, e3
  tmp2 = a(i,j,k) / c(i,j,k-1)
  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)
  tmpl = b(i,j,k) / c(i,j,k-1)
  c(i,j,k) = c(i,j,k) - tmpl * d(i,j,k-1)
  d(i,j,k) = d(i,j,k) - tmpl * e(i,j,k-1)
  f(1,i,j,k) = f(1,i,j,k) - tmpl * f(1,i,j,k-1)
  f(2,i,j,k) = f(2,i,j,k) - tmpl * f(2,i,j,k-1)
  f(3,i,j,k) = f(3,i,j,k) - tmpl * f(3,i,j,k-1)
end do
```

Aug 27 11:59 1996  appsp.f  Page 85

```c
EVERYTHINGS OK HERE

return
```

```c
EVERYTHINGS OK HERE

return
```

```c
EVERYTHINGS OK HERE

return
```

```c
EVERYTHINGS OK HERE

return
```
Aug 27 11:59 1996  appsp.f  Page 85

$ b(xsize,ysize,zsize), c(xsize,ysize,zsize),
$ d(xsize,ysize,zsize), e(xsize,ysize,zsize)

****back-substitution phase

do j = syl, ey1

do i = sx1, ex1

if ( gendz .eq. nz ) then

f(i,i,j,ez) = ( f(i,i,j,ez-1) - e(i,j,ez)*f(i,i,j,ez+1) ) / c(i,i,ez)

endif

do k = ez2, sz, -1

f(i,i,k) = ( f(i,i,k-1) - e(i,k)*f(i,i,k+1) ) / c(i,i,k)

end do

do do

return

end

subroutine txinvr(u,rsd)

****block-diagonal matrix-vector multiplication

Author:  Sisira Weeratunga
NASA Ames Research Center
(10/25/90)
end do
end do
end do
return
end

subroutine tzeta(u,rsd)
***block-diagonal matrix-vector multiplication
Author: Sisira Weeratunga
NASA Ames Research Center
(10/25/90)
include 'appsp.incl'
real *8 u(5,xsize,ysize,zsize), rsd(5,xsize,ysize,zsize)
b = sqrt( 0.50d+00 )
do k = szl, ezl
   do j = syl, eyl
      do i = sx1, ex1
         rul = 1.0d+00 / u(1,i,j,k)
         uu = rul * u(2,i,j,k)
         vv = rul * u(3,i,j,k)
         ww = rul * u(4,i,j,k)
         q = 0.50d+00 * ( uu ** 2
         $              + vv ** 2
         $              + ww ** 2 )
         ac2 = cl * c2 * ( rul * u(5,i,j,k) - q )
         ac = sqrt ( ac2 )
         alph = ( b * u(1,i,j,k) ) / ac
         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)
         t1 = alph * ( r4 + r5 )
         t2 = r3 + t1
         t3 = bt * u(1,i,j,k) * ( r4 - r5 )
         rsd(1,i,j,k) = t2
         rsd(2,i,j,k) = - u(1,i,j,k) * r2 + uu * t2
         rsd(3,i,j,k) = u(1,i,j,k) * r1 + vv * t2
         rsd(4,i,j,k) = ww * t2 + t3
         rsd(5,i,j,k) = u(1,i,j,k) * ( - uu * r2 + vv * r1 )
         $ + q * t2
         $ + ( ac2 / c2 ) * t1
         $ + ww * t3
      end do
   end do
end do

return
end

subroutine callv (finalrsdnm,finalerror)
include 'appsp.incl'
***verification test
real *8 finalrsdnm(*), finalerror(*)
call verify ( finalrsdnm, finalerror, frc )
c***print the CPU time
cwrite (lout,1001) ttotal
1001 format ('/5x,'Total CPU time = ',1pe12.4,' Sec. ')
return
end

subroutine verify ( xcr, xce, xci )
c

**verification routine**

c
**Author:** Sisira Weeratunga  
nasa Ames Research Center  
(10/25/90)

include 'appsp.incl'

dimension xcr(5), xce(5),  
$ xrr(5), xre(5)$

c
***tolerance level***

epsilon = 1.0e-08

if ( ( nx .eq. 12 ) .and.  
$ ( ny .eq. 12 ) .and.  
$ ( nz .eq. 12 ) ) then

c
***Reference values of RMS-norms of residual, for the (12X12X12) grid,  
after 100 time steps, with DT = 1.50e-02

c
xrr(1) = 2.7470315451339479d-02  
xrr(2) = 1.0360746705285417d-02  
xrr(3) = 1.5235745065095532d-02  
xrr(4) = 1.5840557224455615d-02  
xrr(5) = 3.4849040693624606d-02

c
***Reference values of RMS-norms of solution error, for the (12X12X12) grid,  
after 100 time steps, with DT = 1.50e-02

c
xre(1) = 2.7289258557377227d-05  
xre(2) = 1.0364464640837285d-05  
xre(3) = 1.654792827166471d-05  
xre(4) = 1.5750704994480102d-05  
xre(5) = 3.4177661618390531d-05

c
***Reference value of surface integral, for the (12X12X12) grid,  
after 100 time steps, with DT = 1.50e-02

c
xri = 7.840629309126962e+00

c
***verification test for residuals

do m = 1, 5

c
tmp = abs ( ( xcr(m) - xrr(m) ) / xrr(m) )

c
if ( tmp .gt. epsilon ) then

write (iout,1001)

format ('/5x,'VERIFICATION TEST FOR RESIDUALS FAILED')

c
go to 100

c
else

write (iout,1005)

format ('/5x,'VERIFICATION TEST FOR SURFACE INTEGRAL FAILED')

c
end if

c
end do

write (iout,1002)

1002 format ('/5x,'VERIFICATION TEST FOR RESIDUALS ',  
$ 'IS SUCCESSFUL')

c
***verification test for solution error

do m = 1, 5

c
tmp = abs ( ( xce(m) - xre(m) ) / xre(m) )

c
if ( tmp .gt. epsilon ) then

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
***verification test for surface integral

do m = 1, 5

c
tmp = abs ( ( xci - xri ) / xri )

c
if ( tmp .gt. epsilon ) then

write (iout,1005)

format ('/5x,'VERIFICATION TEST FOR SURFACE INTEGRAL FAILED')

c
else

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 ',
do m = 1, 5
   tmp = abs ( ( xcr(m) - xrr(m) ) / xrr(m) )
   if ( tmp .gt. epsilon ) then
      write (iout,1001)
      go to 400
   end if
   end do
   write (iout,1002)
end if

format('///x','CAUTION'.
$ 'REFERENCE VALUES CURRENTLY IN THIS VERIFICATION'.
$ 'ROUTINE'.
$ 'ARE VALID ONLY FOR RUNS WITH THE FOLLOWING PARAMETER'.
$ 'VALUES'.
$ 'NX = 64; NY = 64; NZ = 64'.
$ 'ITMAX = 400'.
$ 'DT = 1.5e-03'.
$ 'CHANGE IN ANY OF THE ABOVE VALUES RENDER THE REFERENCE'.
$ 'VALUES'.
$ 'INVALID AND CAUSES A FAILURE OF THE VERIFICATION TEST'.
)
else
   write (iout,1009)
end if
return
function timer(x)
real x, timer
real secs.tarray(2)

sun version of timing routine
csecs = etime(tarray)
timer = secs
cray version of timing routine
timer = second()
csgl version of timing routine
csecs = secnds(0.0)
timer = secs
creturn
cend
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, exl, eyl, ezl, ex2, ey2, ez2,
  sx, sy, sz,
  sx1, myl, sl1, sx2, sy2, sz2,
  sx3, ex3, my3, ey3, sz3, ez3,
  gtx, gtnx, gty, gtny, gtxz, gtnz,
  sxm2, sym2, smz2, exm2, eym2, emz2,
  sxm1, sym1, szm1, exm1, eym1, emz1
c***for parallelization
c common/parallel/ ex, ey, ez, exl, eyl, ezl, ex2, ey2, ez2,
  sx, sy, sz,
  sx1, myl, sl1, sx2, sy2, sz2,
  sx3, ex3, my3, ey3, sz3, ez3,
  gtx, gtnx, gty, gtny, gtxz, gtnz,
  sxm2, sym2, smz2, exm2, eym2, emz2,
  sxm1, sym1, szm1, exm1, eym1, emz1
c***boundaries for use in setiv - setbv
c common/boundaries/ bx(5,xsize,ysize), bmx(5,xsize,ysize),
  by(5,xsize,ysize), byn(5,xsize,ysize),
  bz(5,xsize,ysize), bzn(5,xsize,ysize)
c***grid
c common/cgcon/ nx, ny, nz,
  i11, i12, j11, j12, k1, k2, idum1,
  dx1, delta, dzeta,
  tx1, tx2, tx3,
  ty1, ty2, ty3,
  tz1, tz2, tz3
c***dissipation
c common/disp/ dx1, dx2, dx3, dx4, dx5,
  dy1, dy2, dy3, dy4, dy5,
  dz1, dz2, dz3, dz4, dz5,
  dssp
c***field variables and residuals
c common/cvar/ u(5,isiz1,isiz2,isiz3),
  r(5,isiz1,isiz2,isiz3),
  frc(5,isiz1,isiz2,isiz3)
c***output control parameters
Aug 27 11:59 1996  appsp.sample.incl Page 1

implicit real*8 (a-h,o-z)

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 )

real timer, tstart, tend
integer ex, ey, ez, exl, eyl, ezl, ex2, ey2, ez2,
$ sx, sy, sz,
$ sxl, syl, szl, sx2, sy2, sz2,
$ sx3, ex3, sy3, ez3, ez3,
$ gtx, gndx, gty, gndy, gtz, gndx,
$ sxm2, sym2, szm2, sxm2, sym2, szm2,
$ sxm1, sym1, szm1, sxm1, sym1, szm1

***for parallelization

common/parallel/ ex, ey, ez, exl, eyl, ezl, ex2, ey2, ez2,
$ sx, sy, sz,
$ sxl, syl, szl, sx2, sy2, sz2,
$ sx3, ex3, sy3, ez3, ez3,
$ gtx, gndx, gty, gndy, gtz, gndx,
$ sxm2, sym2, szm2, sxm2, sym2, szm2,
$ sxm1, sym1, szm1, sxm1, sym1, szm1

***boundaries for use in setiv - setbv

common/boundaries/ bxl(5,isiz2,isiz3), bxn(5,isiz2,isiz3),
$ byl(5,isiz1,isiz3), byn(5,isiz1,isiz3),
$ bz1(5,isiz1,isiz3), bzn(5,isiz1,isiz3)

***grid

common/cgcon/ nx, ny, nz,
$ il1, il2, jl1, jl2, kl1, kl2, idum1,
$ dx1, deta, dzeta,
$ tx1, tx2, tx3,
$ ty1, ty2, ty3,
$ tz1, tz2, tz3

***dissipation

common/diss/ dx1,dx2,dx3,dx4,dx5,
$ dy1,dy2,dy3,dy4,dy5,
$ dz1,dz2,dz3,dz4,dz5,
$ dis

***field variables and residuals

common/cvar/ u(5,isiz1,isiz2,isiz3),
$ rsd(5,isiz1,isiz2,isiz3),
$ frct(5,isiz1,isiz2,isiz3)

***output control parameters

Aug 27 11:59 1996  appsp.sample.incl Page 2

C common/cprcon/ ipr, iout, inorm
C c***newton-raphson iteration control parameters
C common/ctscon/ itmax, invert,
$ dt, omega, tolrd(5),
$ rdsrm(5), errmn(5), frc, ttotan,
$ frc1, frc2, frc3
C c***global jacobian matrix
C c common/cjac/ a(isizl,isiz2,isiz3),
C $ b(isizl,isiz2,isiz3),
C $ c(isizl,isiz2,isiz3),
C $ d(isizl,isiz2,isiz3),
C $ e(isizl,isiz2,isiz3)
C c***coefficients of the exact solution
C common/cexact/ ce(5,13)
C