NASA/CR—1999–209483

PARAMESH: A Parallel Adaptive Mesh Refinement Community Toolkit

Peter MacNeice
Drexel University, Philadelphia, Pennsylvania

Kevin M. Olson
Enrico Fermi Institute, University of Chicago, Chicago, Illinois

Clark Mobarry
NASA Goddard Space Flight Center, Greenbelt, Maryland

Rosalinda de Fainchtein and Charles Packer
Raytheon ITSS, Lanham, Maryland

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

October 1999

Email for Peter MacNeice:
macneice@alfven.gsfc.nasa.gov

Available from:

NASA Center for AeroSpace Information
7121 Standard Drive
Hanover, MD 21076–1320
Price Code: A17

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Price Code: A10

Abstract

In this paper we describe a community toolkit which is designed to provide parallel support with adaptive mesh capability for a large and important class of computational models, those using structured, logically cartesian meshes. The package of Fortran 90 subroutines, called PARAMESH, is designed to provide an application developer with an easy route to extend an existing serial code which uses a logically cartesian structured mesh into a parallel code with adaptive mesh refinement. Alternatively, in its simplest use, and with minimal effort, it can operate as a domain decomposition tool for users who want to parallelize their serial codes, but who do not wish to use adaptivity. The package can provide them with an incremental evolutionary path for their code, converting it first to uniformly refined parallel code, and then later if they so desire, adding adaptivity.

1 Introduction

Many scientific modeling challenges today attempt to simulate processes which span very large ranges in spatial scale. They have reached a point where the finest uniform meshes which can be run on the largest computers do not provide sufficient resolution. Larger dynamic ranges in spatial resolution are required, and for this researchers are looking to adaptive mesh refinement(AMR) techniques.

At the same time the largest computers are now highly parallel distributed memory machines which provide a challenging programming environment. Few genuinely shared memory machines exist, and those which do, with the exception of the Cray YMP architecture, perform inefficiently unless the programmer takes aggressive control of decomposing their computational domain. The principal reason is that most shared memory machines are actually distributed memory machines with globally addressable memory. Data locality is often critical for good performance, because memory access times are not uniform, and fetching data from more remote memory can be relatively expensive. For example the HP-Convex Exemplar series has a sophisticated NUMA(non-uniform memory access) architecture, but un-cached non-local shared data takes more than 100 times longer to retrieve than does un-cached local shared data.

Ideally it should not be necessary for the developers of these models to have to become experts in AMR techniques and parallel computing. It should be possible to make these techniques available and competitive by providing an appropriate toolkit which can be used to extend their existing codes.

In this paper we describe just such a portable community toolkit which is designed to provide parallel support with adaptive mesh capability for a large class of models on distributed memory machines.

Our package of Fortran 90 subroutines, called PARAMESH is designed to provide an application developer with an easy route to extend an existing serial code which uses a logically cartesian structured mesh into a parallel code with AMR.

Alternatively, in its simplest use, and with minimal effort, it can operate as a domain decomposition tool for users who want to parallelize their serial codes, but who do not wish to use adaptivity. The package can provide them with an incremental evolutionary path for their code, converting it first to uniformly refined parallel code, and then later if they so desire, adding adaptivity.

The package is distributed as source code which will enable users to extend it to cover any unusual requirements.

This paper is intended to serve as an introduction to the PARAMESH package, not as a Users manual. A comprehensive Users manual is included with the software distribution.

2 Literature Review

There are a number of different approaches to AMR in the literature. Most AMR treatments have been in support of finite element models on unstructured meshes (i.e. Löhner [1]). These have the advantage that they can be shaped most easily to fit awkward boundary geometries. However unstructured meshes require a large degree of indirect memory referencing which leads to relatively poor performance on cache-based processors.

Berger and co-workers [2, 3, 4] have pioneered AMR for more structured grids. They use a hierarchy of logically cartesian grids and sub-grids to cover the computational domain. They allow for logically rectangular sub-grids, which can overlap, be rotated relative to the coordinates axes, have arbitrary shapes and which can be merged with other sub-grids at the same refinement level whenever appropriate. This is a flexible and memory-efficient strategy, but the resulting code is very complex and has proven to be very difficult to parallelize. Quirk [6] has developed a somewhat simplified variant of this approach. De Zeeuw and Powell [5] implemented a still more simplified variant which develops the hierarchy of sub-grids by bisecting grid blocks in each coordinate direction when refinement is required, and linking the hierarchy of sub-grids developed in this way as the nodes of a data-tree. The package which we describe in this paper is similar to this in all the essential details. All the AMR schemes in this class use guard cells at sub-grid boundaries as a means of providing needed information to the sub-grids which surround it. This can add a significant memory overhead and in most cases a computational overhead also.

The AMR approaches for 'structured' grids which we cited above refine blocks of grid cells. Khokhlov [7] has developed a strategy which refines individual grid cells instead. These cells are again managed as elements of a tree data-structure. This approach has the advantage that it can produce much more flexible adaptivity, in much the same way that the finite-element AMR does. It also avoids the guard cell overhead associated with the sub-grid approaches. However, just as with the unstructured finite element AMR, it requires a large degree of irregular memory referencing and so can be expected to produce slowly executing code. Also, the code which updates the solution at a grid cell is more labor intensive, and in some cases much more so, than in the sub-grid approach. This is because it must constantly use an expensive general interpolation formula to evaluate the terms in the difference equations, from the data in the neighboring grid cells which can be arranged in many different spatial patterns.

Most of these AMR examples have been developed within application codes to which they are tightly coupled. Some have been distributed as packages to enable other users to develop their own applications. Serial examples include HAMR [9], and AMRCLAW [12]. However we are currently aware of only one other package which supports the sub-grid class of AMR on parallel machines. This is a toolkit called DAGH [8]. It is written in

C and C++, but can interface with a user's Fortran routines. It executes in parallel using MPI. An object-oriented AMR library called AMR++ is currently under development [10]. A third object-oriented package is known at SAMRAI [11].

The PARAMESH, DAGH, AMR++, and SAMRAI have some differences. PARAMESH and SAMRAI have additional support routines for conservation laws and the solenoidal condition in MHD, and allow the integration timestep to vary with spatial resolution. DAGH enables error estimation by comparison of the solution at two different refinement levels at each spatial grid point, a feature not (currently) supported by PARAMESH. Perhaps the most significant difference is that DAGH and SAMRAI are constructed and are described in terms of highly abstracted data and control structures. PARAMESH was designed and is described with much less abstraction. This difference will have some impact on the speed with which a user can learn to use each package, though we make no claims here as to which is the easier to learn.

3 Basic Package Design and Application

The PARAMESH package builds a hierarchy of sub-grids to cover the computational domain, with spatial resolution varying to satisfy the demands of the application. These sub-grid blocks form the nodes of a tree data-structure (quad-tree in 2D or oct-tree in 3D).

All the grid blocks have an identical logical structure. Thus, in 2D, if we begin, for example, with a 6 x 4 grid on one block covering the entire domain, the first refinement step would produce 4 child blocks, each with its own 6 x 4 mesh, but now with mesh spacing one-half that of its parent. Any or all of these children can themselves be refined, in the same manner. This process continues, until the domain is covered with a quilt-like pattern of blocks with the desired spatial resolution everywhere.

The grid blocks are assumed to be logically cartesian (or structured). By this we mean that within a block the grid cells can be indexed as though they were cartesian. If a cell's first dimension index is i, then it lies between cells i-1 and i+1. The actual physical grid geometry can be cartesian, cylindrical, spherical, polar(in 2D), or any other metric which enables the physical grid to be mapped to a cartesian grid. The metric coefficients which define quantities such as cell volumes are assumed to be built into the user's algorithm.

Each grid block has a user prescribed number of guard cell layers at each of its boundaries. These guard cells are filled with data from the appropriate neighbor blocks, or by evaluating user prescribed boundary conditions, if the block boundary is part of a boundary to the computational domain.

The package supports 1D, 2D, 2.5D (such as is used frequently in Magneto-Hydrodynamics applications where a magnetic field pointing out of the 2-D plane is kept), and 3D models.

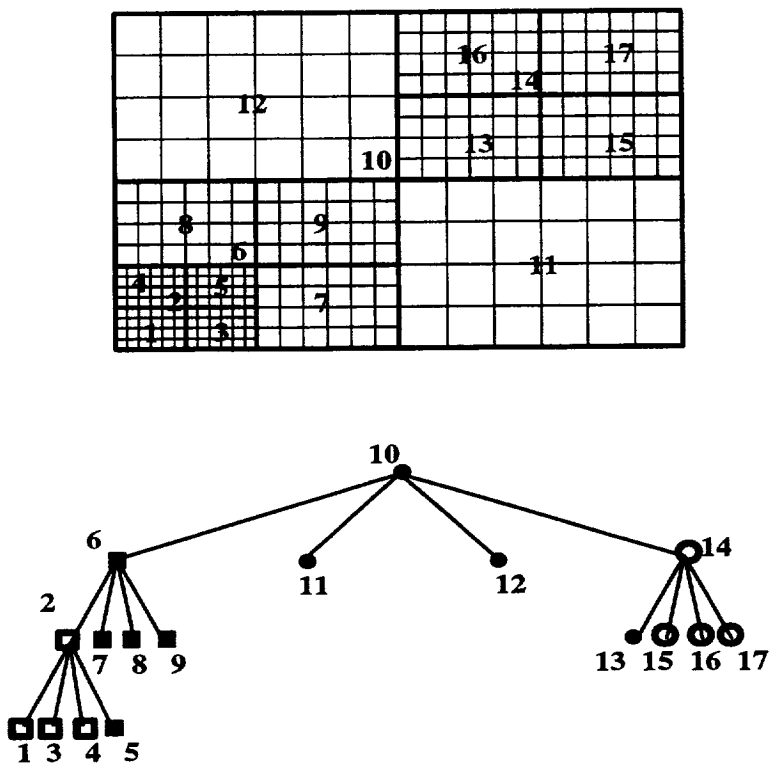

Figure 1: A simple 2D example of a grid block tree covering a rectangular domain. The tree node shapes indicate how this tree might be distributed on a 4 processor machine to balance the workload. The heavy lines in the grid diagram indicate the boundaries of sub-grid blocks, and the lighter lines indicate individual grid cells.

Requiring that all grid blocks have identical logical structure, may, at first sight seem inflexible and therefore inefficient. In terms of memory use this is certainly true, although, even in extreme cases the associated memory overhead is rarely more than about 30%. However this has two significant advantages. The first and most important is that the logical structure of the package is considerably simplified, which is a major advantage in developing robust parallel software. The second is that the data-structures are defined at compile time which gives modern optimizing compilers a better opportunity to manage cache use and extract superior performance.

A simple example is shown in Figure 1 in which a 6 x 4 grid is created on each block. The numbers assigned to each block designate the block's location in the quad-tree below. The different shapes assigned to the nodes of the tree indicate one possible distribution of the blocks during a 4 processor calculation. The leaves of the tree are the active sub-grid blocks.

There are a some restrictions placed on the refinement process. For example, during the refinement process the refinement level is not allowed to jump by more than 1 refinement level at any location in the spatial

domain.

The package manages the creation of the grid blocks, builds and maintains the tree-structure which tracks the spatial relationships between blocks, distributes the blocks amongst the available processors and handles all inter-block and inter-processor communication. It can distribute the blocks in ways which maximize block locality and so minimize inter-processor communications. It also keeps track of physical boundaries on which particular boundary conditions are to be enforced, ensuring that child blocks inherit this information when appropriate.

The philosophy we have adopted in constructing PARAMESH, is to remove from the application developer as much of the burden of inter-block and inter-processor communication as we possibly can. Hopefully, the result is that the application developer can focus on writing code to advance their solution on one generic structured grid-block which is not split across processors.

The parallel structure which PARAMESH assumes is a SPMD (Single Program Multiple Data) approach. In other words the same code executes on all the processors being used, but the local data content modifies the program flow on each processor. It is the message-passing paradigm, but with the burden of message-passing removed from the application developer by the package.

The programming task facing the user can be broken down into a series of straightforward steps.

1. Edit a few lines in the header files provided with the package which define the model's spatial dimensionality, the properties of a typical grid block, the storage limits of the block-tree, and the number of data words required in each grid cell for each of the packages data-structures. The header files are extensively commented to make this step easy.

2. Construct a main program. Most time-dependent fluid models will be able to use an example provided as a template for their main program. This can be easily modified to suit the user's requirements. The sequence of calls to the 'upper level' routines in the PARAMESH package should not need to be altered. The user will need to customize the construction of an initial grid, establish a valid initial solution on this grid, set the number of timesteps and limits on the allowed range of refinement, and add any I/O required. Sample code for all these tasks is provided.

3. Provide a routine which advances the model solution on all the 'leaf' grid blocks through a timestep (or iteration). This step is much simpler than it appears. The routine can be constructed by taking the equivalent code from the user's existing application which advances the solution on a single grid, and inserting it inside a loop over the leaf

blocks on the local processor. Inside this loop, the solution data for the current grid block must be copied from the package's data-structures into the equivalent local variables in the user's code segment. Then the user's code segment executes to update the solution on that block. Finally, the solution is copied back from the user variables to the package's data-structures, before the loop moves on to repeat the same sequence for the next leaf block. If conservation constraints must be satisfied, a few extra lines must be added inside this routine to capture fluxes and/or cell edge data at block boundaries.

4. Provide a routine to compute the model's timestep. Again this can be straightforwardly constructed from the existing template by inserting the appropriate code segment from the user's existing application, in the manner described in step 3. The existing timestep routine template has all the control and inter-processor communications required to properly compute the global minimum of the maximum timesteps calculated for each block, or to enable longer timesteps on coarser blocks if the user chooses that option.

5. Provide a routine to establish the initial state on the initial grid. A template has been provided which can be tailored to suit the user's model.

6. Provide a routine to set data values in guard cells at physical boundaries in order to implement the user's choices of boundary conditions. Once again a template routine has been provided which can be very easily modified.

7. Provide a function to test a single block to determine if any refinement or de-refinement is appropriate. This function is called during the refinement testing operation. Again, a template exists which can be modified by the user.

Detailed 'How To' instruction and illustration is provided in the User's manual which comes bundled with the software distribution.

Templates and worked examples are provided with the package for all of these tasks. The Fortran 90 pointers mechanism can be used to connect the PARAMESH data structures with those of the user's application so that they do not need to edit the variable names in their code segments.[1]

The design philosophy while developing this package has been to present the user with a clean well commented Fortran 90 source code, sufficiently simple in structure that the user would not be afraid to customize routines for their own particular use. We also strove for efficiency on cache-based multiprocessors.

[1] Use of the Fortran 90 pointers mechanism may degrade performance significantly. If computational speed is critical it may be better to explicitly copy data between the user defined variables and the PARAMESH data-structures, as necessary.

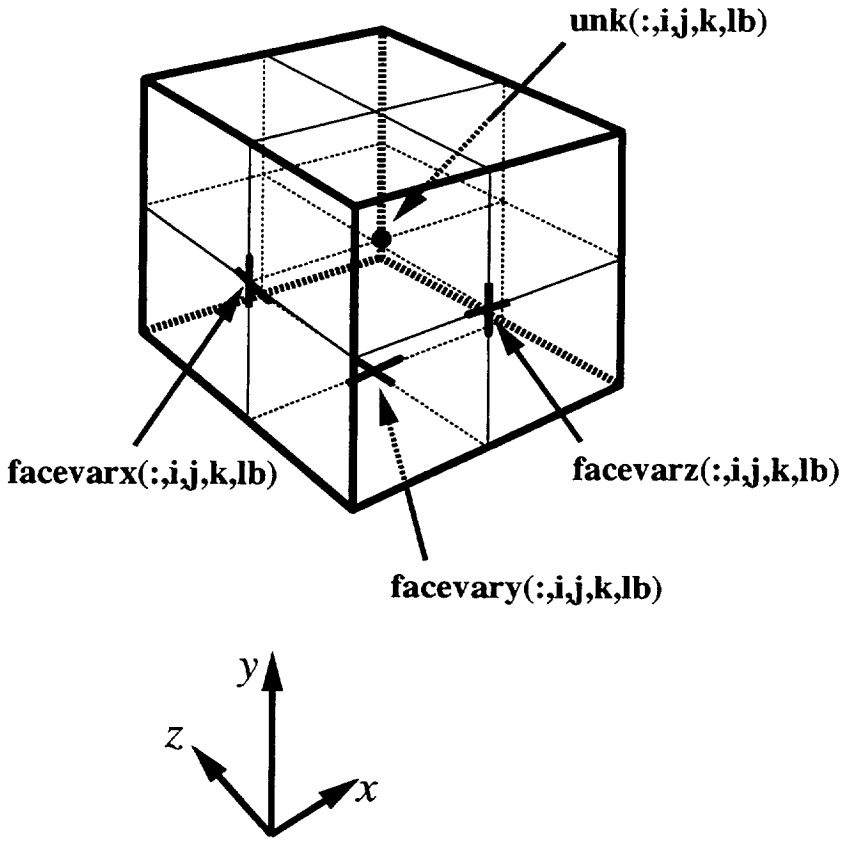

Figure 2: The physical relationship of the basic solution data words to their grid cell shown for grid cell indices (i,j,k) in a 3D model.

4 Data-structures in PARAMESH

There are two critical data-structures maintained by the package, one to store the model solution, and the other to store the tree information describing the numerical grid.

The data which constitutes the solution can include data located at the center point of grid cells, and data located at the centers of the grid cell faces. On each processor the cell-centered data for the grid blocks located in that processors memory are stored in an array called **unk**. The cell face-centered data are stored in arrays called `facevarx`, `facevary` and `facevarz`. This is illustrated in Figure 2.

This datastructure is record based, which means that within each of the arrays all the data words for a given grid cell are stored in contiguous memory words. This should help give the package superior cache-line reuse characteristics. For example in a 3D hydro code with mass density, momentum and energy density all specified at grid cell center, only **unk** would be used and its data for cell (i,j,k) of grid sub-block `lb` could be set up like this :

`unk(1,i,j,k,lb)`	- mass density
`unk(2,i,j,k,lb)`	- x-component of momentum
`unk(3,i,j,k,lb)`	- y-component of momentum
`unk(4,i,j,k,lb)`	- z-component of momentum
`unk(5,i,j,k,lb)`	- energy density.

A 3D MHD code with mass, momentum and energy densities specified at grid cell center, and the x-component of magnetic field on the x-face, y-component of magnetic field on the y-face and z-component of magnetic field on the z-face could use :

`unk(1,i,j,k,lb)`	- mass density
`unk(2,i,j,k,lb)`	- x-component of momentum
`unk(3,i,j,k,lb)`	- y-component of momentum
`unk(4,i,j,k,lb)`	- z-component of momentum
`unk(5,i,j,k,lb)`	- energy density
`facevarx(1,i,j,k,lb)`	- x-component of magnetic field
`facevary(1,i,j,k,lb)`	- y-component of magnetic field
`facevarz(1,i,j,k,lb)`	- z-component of magnetic field.

Each node in the tree structure stores the tree location of its parent as well as any child nodes which might exist. Each node also stores the location in space of the sub-grid block that it represents as well as a vector which describes the physical size in each dimension of the bounding box containing that sub-grid. The tree data structure is fully linked, meaning that each node in the tree has stored at its location the tree locations of its neighboring sub-grids at its level in the tree. Each link in the tree (i.e. stored locations of parents, children, and neighboring sub-grids) are stored as two integers: the first being the memory location within a remote processor and the second being the processor to which that link points.

5 Additional Features

5.1 Conservation Laws and Solenoidal Constraint

Many applications will require consistent data use at the boundaries between grid blocks at different refinement levels. For example, conservative hydrodynamics codes will require that the fluxes entering or leaving a grid cell through a common cell face shared with 4 cells of a more refined neighbor block, equal the sum of the fluxes across the appropriate faces of the 4 smaller cells. The package provides routines which enforce these constraints.

In MHD codes, a similar consistency issue can arise with electric field values which are sometimes known at the centers of grid cell edges. The magnetic field update evaluates a circulation integral along the cell edges which bound a grid cell face, in order to compute the change in magnetic induction through that face. This change must be consistent at the shared

boundary of grid blocks of different refinement levels. We provide routines which can be used to enforce this constraint also.

5.2 Variable timestep support

When there is significant variation in spatial resolution within the computational domain, the timestep computed using a fixed Courant number will probably also vary significantly. With PARAMESH, the user can choose to use a uniform timestep, or can vary the timestep from grid block to grid block provided certain restrictions are satisfied. These restrictions are that any two blocks with the same refinement level must use the same timestep, that a block cannot use a longer timestep than any more coarsely resolved blocks, and that all timesteps are integer multiples of the timestep used for the finest active refinement level.

There are two reasons why we might want to allow the solution on coarser blocks to be advanced with longer timesteps, but it is not clear that these reasons are compelling for all cases.

If we use a large number of very fine timesteps to advance the solution on the more coarsely refined grid blocks, we introduce the possibility that the accumulated effects of numerical diffusion will become significant. A counter argument to this, suggests that this can never be too serious because the reason these grid blocks were coarsely refined was that there was not very much structure there anyway.

The second reason to use variable timesteps is to save the computational effort associated with advancing the solution on the coarser blocks with unnecessarily fine timesteps. However, to enable the use of variable timesteps, extra memory must be allocated to store temporary copies of the solution. Also, because most real applications have restrictive synchronization constraints, enabling variable timesteps tends to force ordering of the way the solution must be advanced on the different grid blocks, and this can have a damaging effect on load balance.

5.3 Load Balancing

PARAMESH aggressively manages the distribution of grid blocks amongst the processors in an effort to achieve load balance, and by improving data locality to lower communication costs. It uses a Peano-Hilbert space filling curve to define the ordering of the grid blocks. Different work weighting factors are assigned to the different types of blocks in the tree. For example leaf blocks are assigned the highest value because they do the most work. Parents of leaf blocks also receive a non-zero weighting. The package then sums the work associated with all the blocks and tries to segment the list in such a way as to maximize the load balance. The work weight assigned to different types of blocks can be adjusted to suit the user's needs. This feature is very similar to methods developed for distributing tree data structures used for particle applications on multi-processor machines [13].

5.4 Interpolation functions.

When a child block is spawned during a refinement step, the solution arrays on this new grid block must be initialized, by interpolating from the solution on its parent. We call this operation 'prolongation' following the convention used in Multigrid solvers. The interpolation used by default in PARAMESH during prolongation is linear interpolation. This can be easily up-graded if necessary for use with higher order algorithms such as the Piecewise Parabolic Method(PPM). Some alternative interpolation operators are included with the package distribution.

6 Portability

The code uses the SGI/Cray SHMEM (shared memory) library to perform all the necessary interprocessor communications. The SHMEM library is available on the Cray T3E and on SGI machines. For machines which do not support the SHMEM library, the package uses MPI. Replacement SHMEM routines are provided which mimic the SHMEM calls but actually use MPI inside. The MPI version uses an additional supplementary library called MMPI (Managed MPI), developed by Harold Carter Edwards at the University of Texas, which supports one-sided communications and blocking gets.

7 The Structure of an Application

In this section we illustrate the basic structure of a typical application which uses PARAMESH.

The final application should be thought of as having a basic skeleton provided by the PARAMESH package, into which the user inserts appropriate snippets of serial (i.e. single processor) code, each of which is designed to perform a particular computational sub-task on a generic sub-block.

Templates are provided in PARAMESH for these sub-tasks, which the application developer can edit. These define the interface required between the package and the user's code snippet. We will illustrate this process below.

However, first let us describe the basic skeleton, defined by the applications main program. The typical flowchart is shown in Figure 3.

7.1 Template for Main Program

This begins with a sequence of include statements, which make the tree and solution data structures visible to the main program. The next step is to initialize PARAMESH, which is done by calling the routine `amr_initialize`.

```
! amr package initialization
      call amr_initialize
```

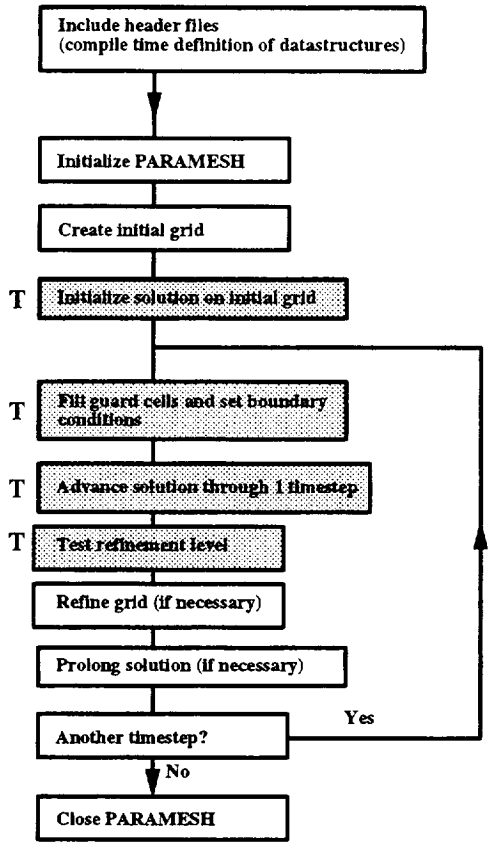

Figure 3: A typical flowchart for a PARAMESH application. Those elements of the flowchart which the user may need to modify to some degree from the templates provided are labeled with a 'T'.

We are now ready to generate the initial grid. The simplest way to do this is to place a single grid block with refinement level 1 covering the entire computational domain. The boundary condition flags are set on this block and it is marked for refinement. Then we refine this block using the machinery of the amr package until we have a mesh which has an acceptable refinement pattern for the initial solution.

```
! Setup up initial grid-block tree

! set limits on the range of refinement levels to be allowed.
! level 1 is a single block covering the entire domain, level
! 2 is refined by a factor 2, level 3 by a factor 4, etc.
      lrefine_max = 10          ! finest refinement level allowed
      lrefine_min = 6           ! coarsest refinement level allowed

! set the no of blocks required initially to cover the
! computational domain
```

```
      no_of_blocks = 2**(lrefine_min-1)

! initialize the counter for the number of blocks currently
! on this processor
      lnblocks = 0


! begin by setting up a single block on processor 0, covering
! the whole domain which ranges from 0 to 1.0 along each axis
! in this example.
      if(mype.eq.0.) then
            lnblocks      = 1
            size(:,1)     = 1.0
            bnd_box(1,:,1) = 0.0
            bnd_box(2,:,1) = bnd_box(1,:,1) +size(:,1)
            coord(:,1)    = .5*(bnd_box(1,:,1)+ bnd_box(2,:,1))
            nodetype(1)   = 1
            lrefine(1)    = 1

            neigh(:,:,1)  = -21 ! initial block is not its own
                                ! neighbor. signifies external
                                ! boundaries.

            refine(1)=.true.    ! mark this block for refinement
      endif
```

At this point we have a single block covering the whole domain. In this example the block size is set to 1.0 and the block center is located at 0.5 along each axis. This first block is placed on processor 0, and **lnblocks** which stores the number of blocks on the local processor is set to 1 on processor 0. This first block is assigned **nodetype = 1** which indicates that at this point it is recognized as a leaf block. Also note that the addresses of this block's neighbors are all set to values less than -20. This is the way external boundaries are identified to the package. If we wished to use periodic boundaries we could set the **neigh** array so that the initial block identifies itself as it's own neighbors.

Now we are ready to refine this first block.

```
! Now cycle over blocks until 'no_of_blocks' leaf blocks have
! been created.
      loop_count_max = int(log(real(no_of_blocks ))/log(2.)+.1)
      do loop_count=1,loop_count_max
            refine(1:lnblocks)=.true.
            call shmem_barrier_all()
! refine grid and apply Peano-Hilbert reordering to grid blocks if
! necessary
```

```
            call amr_refine_derefine
        enddo
```

In this example we loop over the list of blocks marking all the existing blocks for refinement and then implementing the refinement with a call to the routine **amr_refine_derefine**. We continue looping until we have reached a pre-selected refinement level. The routine **amr_refine_derefine** creates the necessary child blocks, identifying them as leaf blocks and modifying the nodetype of their parents to indicate that they are no longer leaf blocks. It also manages the inheritance of the neighbor addresses, which in this case means that the correct boundary conditions will be applied to children which are next to the external boundaries of the computational domain. This simple example sets up an initial grid-block tree which covers the computational domain with uniform refinement. However it is easy to see how the process can be modified to create more complex initial grids. This topic is discussed in more detail below.

Now we need to set up the initial solution on these initial grid blocks.

```
! set up initial solution on the grid blocks
        time = 0.
        call initial_soln(mype)

! exchange guardcell information - the call to guardcell also
! causes the guard cells at external boundaries to be filled
! using the user defined boundary conditions which the user
! must code into the routine amr_bc_block.
        nlayers = nguard
        call amr_guardcell(mype,1,nlayers)
```

This is done here in a user supplied routine which we have called **initial_soln**. This routine sets the initial solution on the interior grid cells on the leaf blocks. Then the call to **amr_guardcell** causes **nlayers** of guard cells at each block boundary to be filled with the correct data from their neighboring blocks. The call to **amr_guardcell** also causes the guard cells at external boundaries to be filled according to the boundary conditions which the user has coded into the routine **amr_bc_block**. A template for **amr_bc_block** is provided with the package.

Finally we are ready to advance the solution in time. The loop over timesteps is very simple. First we call the user's routine which advances the solution. In this case a user supplied routine called **advect** is being used to advance the fluid equations through one timestep. The routine **advect** includes the computation of the timestep as well as the integration of the fluid equations. When the solution has been advanced it is tested to see if the current refinement pattern is still appropriate. The arguments **lrefine_max** and **lrefine_min** to **amr_test_refinement** set bounds on the refinement levels which the code is permitted to use. Any refinements or de-refinements which

have been requested are then executed by the call to amr_refine_derefine. Of course the user will need to modify amr_test_refinement to perform the test which they deem appropriate for their application. We demonstrate how this can be done below. The call to amr_prolong fills any newly created child blocks with data by interpolating from their parents. Finally the guard cells are updated and boundary conditions established by calling amr_guardcell.

```
! set the no of timesteps
      ntsteps = 100

! Begin time integration
      do loop=1,ntsteps

! perform a single timestep integration
            call advect(mype,dt,time,loop)

! test to see if refinement or de-refinement is necessary
            call amr_test_refinement(mype, lrefine_min,lrefine_max)

! refine grid and apply Peano-Hilbert reordering to grid
! blocks if necessary
            call amr_refine_derefine

! prolong solution to any new leaf blocks if necessary
            call amr_prolong(mype,iopt,nlayers)

! exchange guardcell information and boundary conditions.
            call amr_guardcell(mype,1,nlayers)

      enddo
! end timestep integration loop
```

Before we exit, we must close the amr package. The principal task performed here is to properly close the MPI package, if the application has been built to use MPI.

```
! Now close the amr package
      call amr_close()
```

That is the basic structure of the main program for a typical parallel AMR fluid code which would work in 1, 2 or 3 spatial dimensions. It is essentially independent of the algorithm which is being used, since these details are submerged in the routine which the user supplies to advance the solution through a single timestep. It should also be clear that this design for the main program could be used for any calculation on this type

of grid which approaches a solution through an iterative loop - it need not necessarily be a time dependent fluid code. Finally we should note that a working code would have some I/O routines, which we have left out here in the interests of simplicity.

7.2 User Modified Routines

One reason that the main program has such a straightforward and robust template is because all the details associated with the actual applications algorithm have been placed inside the few routines to be supplied by the user. However these routines can also have a very straightforward structure. In essence they amount to a 'do loop' over the grid-blocks stored on the local processor. Inside the 'do loop' the user inserts the serial code to perform the appropriate computation on a generic grid-block. For all the necessary routines, templates are provided which already contain the loop over grid blocks which makes the task of integrating these code segments with the package relatively painless.

We use the example of the routine `amr_test_refinement` to illustrate this process.

The `amr_test_refinement` routine has the responsibility of setting values to the logical arrays `refine` and `derefine`, which belong to the tree datastructure and which control whether a grid-block is refined or removed by the subsequent call to `amr_refine_derefine`. It also guarantees that no refinements or de-refinements are requested which exceed the refinement limits `lrefine_min` and `lrefine_max`. Assume that we wish to compute a local error measure from the data stored in `unk(1,:,:,:,:)`. Refinement is selected for any leaf grid block if this error exceeds some predefined threshold anywhere in that block. Similarly if the error measure is less than some small value everywhere on that block the block can be marked for de-refinement. Here is a version of `amr_test_refinement` which does this. It amounts to a loop over all the leaf grid blocks on the local processor, and for each iteration we call a routine called `error_measure` which computes the local error estimate for that block.

```
      subroutine amr_test_refinement( mype,lrefine_min,lrefine_max)
#include "physicaldata.fh"
      include 'tree.fh'
      real error(1:nxb+2*nguard, 1:nyb+2*nguard,1:nzb+2*nguard)
      real error_max

! Re-initialize the refinement and de-refinement flag arrays
      refine(:) = .false.
      derefine(:) = .false.
! Error limits which control the refinement and de-refinement
! requests below.
      ref = .35
```

```
      deref = .05

! Loop over all leaf blocks and all parents of leaf blocks
      if(lnblocks.gt.0) then
      do lb=1,lnblocks
      if(nodetype(lb).eq.1.or.nodetype (lb).eq.2) then

! User provided routine which returns an array error, which has
! some error measure computed for each grid cell, based on some
! computation on the input array unk(1,:,:,:,lb).
            call error_measure( error, unk(1,1,1,1,lb) )
            error_max = maxval( error )

! Does the error measure on this block anywhere exceed the limit
! which should trigger refinement?
            if( ( lrefine(lb) .lt. lrefine_max) .and.
1             ( error_max .ge. ref ) ) refine (lb) = .true.
! Can we derefine this block?
            if( ( lrefine(lb) .gt. lrefine_min ) .and.
1             ( .not. lrefine(lb) ) .and.
2             ( error_max .lt. deref ) ) derefine(lb) = .true.

      endif
      end do              ! end of loop over blocks
      endif

return
end
```

That completes the construction of `amr_test_refinement`. From this it should be obvious how it can be customized to handle more complicated error measures.

All the other routines which the user is required to provide (in this case `initial_soln`, `advect` and `amr_bc_block`) have essentially the same structure. Obviously some will be more complicated than others. However they all consist of a 'do loop' or sequence of 'do loop's over the leaf grid blocks, with each do loop executing a particular code segment on data entirely local to the current grid block.

7.3 Conservation Laws

PARAMESH provides some support for conservation laws and the solenoidal condition associated with MHD models. This is implemented in the most obvious way. However, because this requires the insertion of a few lines of code in the heart of the routine which the user supplies for advancing the solution, we take the time to outline the steps here.

When neighboring grid blocks have different levels of refinement, the fluxes at the common boundary used to update the solutions on the two blocks may not be completely consistent. This will lead to a loss of conservation if it is not remedied. PARAMESH provides some routines which will update the fluxes on the coarser of the two blocks with data provided by the finer block. This data should be captured while the user's algorithm is computing the appropriate fluxes and advancing the solution. The strategy here is to record the inconsistent fluxes used at the block boundaries as the solution is advanced, modify then to achieve consistency, and then correct the advanced solution for the differences between the original and modified fluxes.

The flux data must be stored in the PARAMESH arrays called `flux_x`, `flux_y` and `flux_z`. Once this has been done on all the leaf blocks, the routine `amr_flux_conserve` is called. The first thing this does is to copy the contents of these flux arrays into temporary copies, called `tflux_x`, `tflux_y` and `tflux_z`. Then when a block borders a finer neighbor the flux array for that direction (`flux_x` on x faces, etc). is updated with the appropriate sum or average of fluxes from the neighbor. Finally, after the call to `amr_flux_conserve`, each block is considered in turn and the solution in cells immediately inside each block boundary are corrected for any differences between the arrays `flux_x`, `flux_y`, `flux_z` and `tflux_x`, `tflux_y`, `tflux_z`.

A variant of this problem can also occur in MHD codes which use circulation integrals of electric field values defined on grid cell edges to update magnetic field components. If these circulation integrals are not evaluated consistently at the block boundaries where the refinement level jumps, then the magnetic field will not remain divergence-free. We have provided some routines which use the same approach as used with the fluxes, to ensure consistency.

8 Examples

We discuss four examples of the application of PARAMESH to complex fluid codes. The first is a 1D fluid model of the development of a Solar Coronal condensation, using a MUSCL advection solver. The second is a 2D MUSCL simulation of a strong plane shock wave propagating past a square obstacle. The third is a 3D MHD code which uses Flux Corrected Transport(FCT). The fourth is another 2D hydrodynamics code based on the Piecewise Parabolic algorithm. The purpose here is not to explain how these pre-existing codes were modified to use PARAMESH, rather to illustrate with physical examples the type of solutions which the package enables and the range of real numerical algorithms that have already been supported, and to introduce the codes which we used to produce performance results described in section 9.

8.1 A 1D Fluid Model

The serial uniform mesh 1D code [17] from which our first example was developed, solves the fluid equations using a MUSCL-type scheme, as described in Balsara [16]. A MUSCL algorithm is a second order variant of the Godunov approach to hydrodynamic modeling.

In a classical Godunov scheme, each interface between grid cells is considered to represent a separate Riemann problem. Uniform states are assumed to exist on either side of each cell interface at the beginning of the timestep. An analytic solution to the Riemann problem exists for the 1D Euler equations, and this is applied over the timestep to compute the flow of mass, momentum and energy across each cell interface. These fluxes are used to updated the amount of mass, momentum and energy in each grid cell and the new time advanced solution at cell centers is computed assuming that the densities are piece-wise constant within each grid cell. The timestep is limited by a Courant-Friedrichs-Lewy condition which ensures that disturbances propagating from any cell interface in the analytic solution do not interfere with similar disturbances propagating from its neighboring cell interfaces during a timestep.

The classical Godunov scheme applies piece-wise constant interpolation to define the mass, momentum and energy densities across a grid cell. A MUSCL scheme is a second order variant because it uses a linear interpolant with a 'limiter' algorithm to reconstruct the profiles inside each grid cell from the cell centered values. In this case the code also includes a steepening algorithm which is applied to the linearly degenerate entropy wave, as described by Yang [14].

The code uses a one pass time integration scheme which is second order accurate [15] in time.

For the solar simulation the code included a spatially dependent gravitational acceleration term, optically thin radiative energy loss, a spatially varying heating term and non-linear Spitzer-Harm thermal conduction. All the solution variables were cell centered.

A snapshot of the development of a coronal condensation calculation is shown in Figure 4. This shows the variation of electron number density, velocity, temperature, and spatial resolution along the axis of a coronal flux tube. It is not our purpose here to discuss the physical implications of this calculation which will appear elsewhere [17]. The key points we wish to emphasize are that the calculation was performed using grid blocks with 20 interior grid points and 2 guard cells at each block boundary. The refinement criterion we used tested for variations in electron number density, triggering refinement if a variation of more than 25% was seen between grid cells and allowing de-refinement if no variation greater than 5% was detected in a pair of sibling leaf blocks. At the time the snapshot was made, there were 55 grid blocks distributed along the flux tube, with 7 different refinement levels (i.e. a factor of $2^6 = 64$ variation in resolution between coarsest and finest

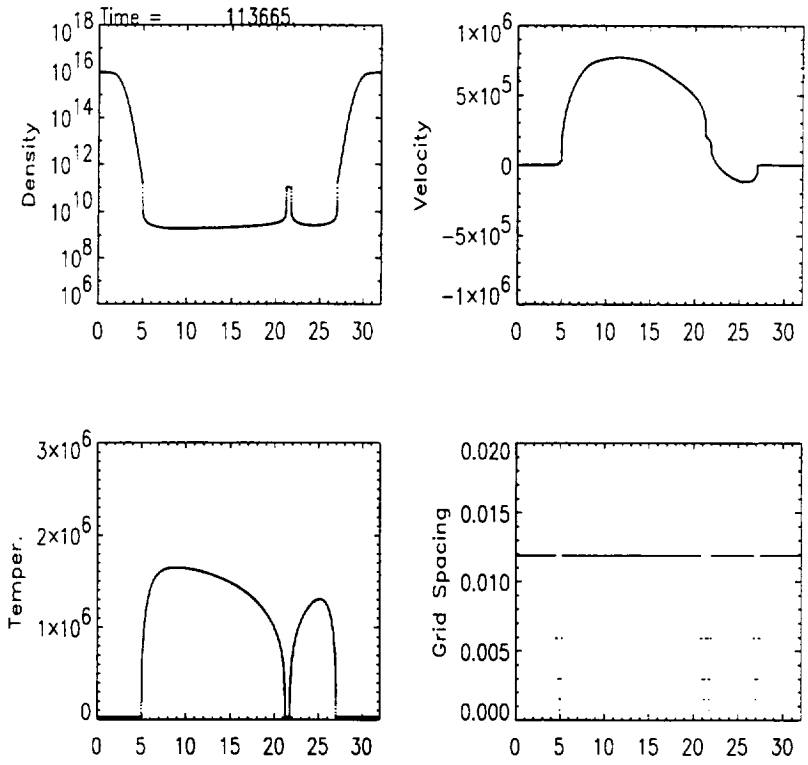

Figure 4: A snapshot during the development of a Coronal Condensation modeled with a 1D MUSCL type fluid algorithm.

grid cell). The concentration of refined blocks in 2 distinct narrow regions illustrates the power of AMR.

This calculation enforced conservation constraints at the block boundaries where the refinement varied. This was done by saving the gas dynamic fluxes computed at grid block boundaries. These fluxes were then modified by calling the routine `amr_flux_conserve` which, at the shared grid block boundary at a refinement jump, replaces the fluxes on the coarser block with the fluxes used on the finer neighbor. The solution was then corrected to be consistent with these flux modifications.

8.2 A 2D Fluid Model

The second example is a 2D gas dynamic simulation of a Mach 10 plane shock wave propagating past a square obstacle. For this we used a MUSCL scheme, with an approximate Riemann solver. We used 8 x 8 sized grid blocks with 2 layers of guard cells at each block boundary. A snapshot of the density variation in a section of the computational domain near the obstacle is shown in Figure 5.

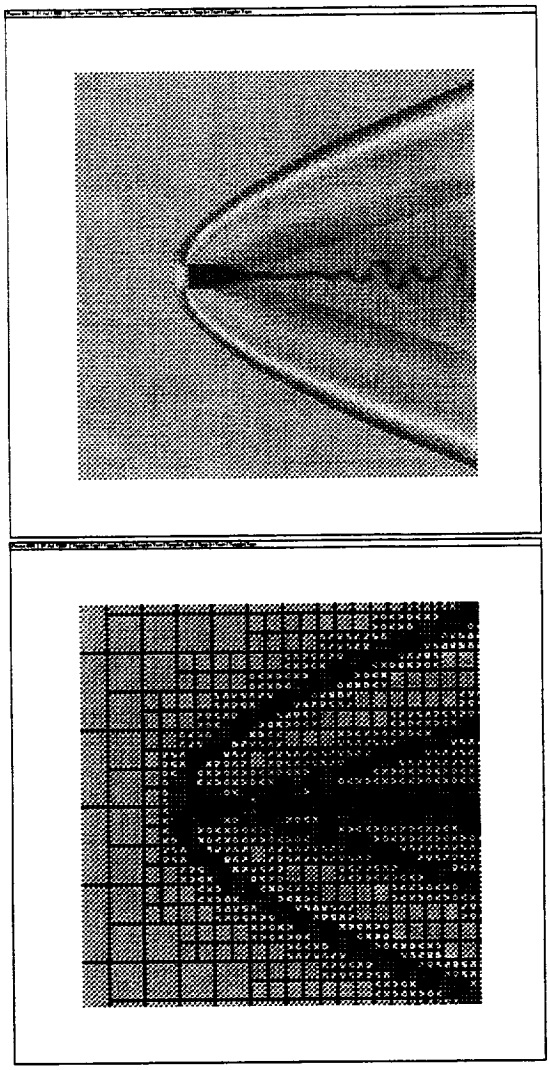

Figure 5: The top half of the frame shows a snapshot of the density from a 2D hydro simulation of a Mach 10 shock flowing past a square obstacle. Only the central portion of the computational domain is shown here. The bottom half frame shows the same image but with the boundaries of the grid blocks superimposed.

In Figure 5 you can see that there are 5 distinct 'ray'-like regions extending downstream from the obstacle. These are the bow shock above and below the object, a second pair of shock fronts which extend from behind the obstacle, and a shear layer extending straight back from the obstacle. In the bottom frame of Figure 5 we show the same snapshot but with an outline of the grid blocks superimposed on it. This shows how the refinement algorithm has placed high resolution along all these features, and has even adapted to follow the shape of the oscillation which has developed in the unstable shear layer.

In the later stages of this calculation we used almost 25000 grid blocks distributed across 128 processors of an SGI/Cray T3E. The computation involved 10 different refinement levels. In the snapshot there are 5 different

levels visible, a dynamic range of $2^5 = 32$.

8.3 A 3D MHD Model

Our third example is an AMR implementation of the FCTMHD3D code, a 3D MHD code which uses Flux Corrected Transport(FCT) [18].

This application exercises still more of the functions built into PARAMESH. The time advance in this code achieves second order by using a 2 step predictor-corrector approach. The grid blocks are 8 x 8 x 8 with 3 guard cell layers at each block boundary. The code uses a staggered mesh approach in which the mass, momentum and energy densities are specified at cell centers, but the magnetic field components are specified at cell face centers. So the x-component of the magnetic field is stored in our `facevarx` array, the y-component in `facevary` and the z-component in `facevarz` (see Figure 2). The routine `amr_flux_conserve` was used to ensure that mass, momentum and energy conservation were maintained at block boundaries where the refinement level jumped.

The time advance of the magnetic field requires the computation of circulation integrals of the electric field about each grid cell face. These electric field values are known at the centers of the cell edges. To ensure that the constraint $\nabla \cdot \vec{B} = 0$ on the magnetic field $\vec{B}$, was maintained, it is necessary that the circulation integrals on shared block faces at refinement jumps be consistent. This is handled by a routine called `amr_edge_average` which replaces the edge values on the coarser block face with appropriate averages from the edge values on the finer block face.

In section 9 we report performance measurements for this code on the T3E.

8.4 A PPM Model

Our final example is a hydrodynamics code, based on the Piecewise Parabolic method [19].

This application is being developed at the University of Chicago as part of the ASCI project [20]. The code (known as the FLASH code) is being used to study Astrophysical X-ray bursts, Novae, and Supernovae. FLASH employs the piecewise parabolic method, allows for equations of state which vary as a function of space, can follow an arbitrary number of nuclear species, and allows for energy generation via nuclear reactions. The overall FLASH code is based upon an original uniform mesh code known as PROMETHEUS [21] with extensions for nuclear burning and equations of state appropriate for stellar interiors [22], [23].

FLASH uses PARAMESH blocks which are 8 x 8 x 8 cells with a 4 guardcell region around each block. To make the code multidimensional, each timestep is vector split and so is divided up into 1, 2 or 3 directional sweeps through the mesh. As a result, a flux conservation and guardcell filling operation must be performed after each directional sweep. In three

dimensions the guard-cells must be filled 3 times per time step and this places a considerable demand on the PARAMESH software to fulfill these guardcell filling requests. For the FLASH code several of the main PARAMESH routines have been rewritten directly using MPI rather than the MMPI library as described above. We regard these changes as a step along the way to producing the next release of PARAMESH which will include versions of the communication intensive routines written in MPI.

In section 9 we report performance measurements for the FLASH code on a SGI Origin 2000 and the ASCI RED (Intel) machine.

9 Performance

What performance results can we provide to a potential user of PARAMESH which would enable them to decide whether they can use it to develop an efficient application ? They need to be convinced that with AMR they can achieve the desired resolution with faster, hopefully much faster, time to solution than they can achieve with a uniformly refined calculation, and that this will remain true when using a large number of processors.

Low AMR-related overhead, good scaling, and load balancing are important factors in enabling this. In this section we report some performance results, with the following caveat.

It is difficult to define a useful and objective measure of the performance of PARAMESH (or any other AMR package) which is not highly application dependent. Fine details of the applications algorithm, its balance of computation and communication when expressed in parallel, the refinement criteria used in controlling refinement, the frequency with which the grid resolution is tested and modified, the size of grid blocks used, and many other design choices will all modify the performance. Likewise, the performance figures will be highly dependent on the capabilities of the hardware in use. Processor speeds, memory latencies, inter-processor communication bandwidth and message latencies will all influence performance.

The best we can hope to do is to show that for some specific cases, good scaling and/or performance were achieved.

To illustrate the performance which can be achieved with PARAMESH, we have measured aspects of the AMR overhead, scaling, load balance and time to solution of the AMR version of the FCTMHD3D code on the T3E, and of the FLASH code on the ASCI RED machine.

9.1 Performance with Shared Memory

PARAMESH was originally written for the low latency, high bandwidth environment of the T3E. We would therefore expect it to perform well on any machine with those characteristics. Our first performance test describes the use of the FCTMHD3D code on a 512 processor T3E using the SHMEM communication library.

We report performance from two scaling tests, the first with a problem constructed to keep the amount of work per processor fixed, and the second with a fixed total amount of work. In each case we show the total time to solution and the percentage of the time to solution which was contributed by those routines performing the AMR and parallelization tasks.

The tasks considered to represent AMR and parallelization overhead were guardcell filling, building the grid-block tree, refinement and de-refinement, data prolongation and restriction, and enforcement of conservation laws at refinement jumps. The tasks considered part of the application were local computation of the timestep, local time advance of the solution, and local computation of the error measure to be used in testing the local spatial resolution.

The performance reported was for 20 timesteps, with testing and modification of the grid enabled for every second timestep, and with the refinement testing routine testing out to 3 grid cells beyond the physical boundary of each grid block.

Figure 6 shows the execution time for these different components, for both tests, as the number of processors is varied. The left frame refers to the scale-up test, and the right frame to the fixed size problem.

For the scale-up problem, the time to solution is almost constant as the processor number is varied. The scaling for the fixed size problem is also good, although it deteriorates at the larger processor numbers when each processor has very little work to do. For example, when using 256 processors there is an average of only 3 leaf blocks on each processor.

Figure 7 shows the typical load balance achieved during the transport phase of the same calculation on 32 processors of the SGI/Cray T3E. In this case the load balance shows that the 32 processors used 99.1% of the available cpu time during this phase of the calculation.

9.2 Performance without Shared Memory

We set up several test calculations using the FLASH code to explore PARAMESH performance in the absence of shared memory.

In the first calculation a constant amount of work was assigned to each processor. A shock tube was modeled in which all disturbances propagate in the x direction and the solution does not vary along the y and z coordinate axes. As processors are added the domain size of the total problem was increased proportionately in the y direction. The total execution time for this test calculation is shown in figure 8. These curves clearly show that the code scales well for this test calculation out to 1024 processors. They also show that the AMR overhead amounts to only 25% of the total execution time. These test calculations were run using the ASCI-RED machine located at Sandia National Laboratory which is a 'one-of-a-kind' machine constucted using Intel Pentium-pro processors connected by a fast network (for more information see http://www.sandia.gov/ASCI/Red).

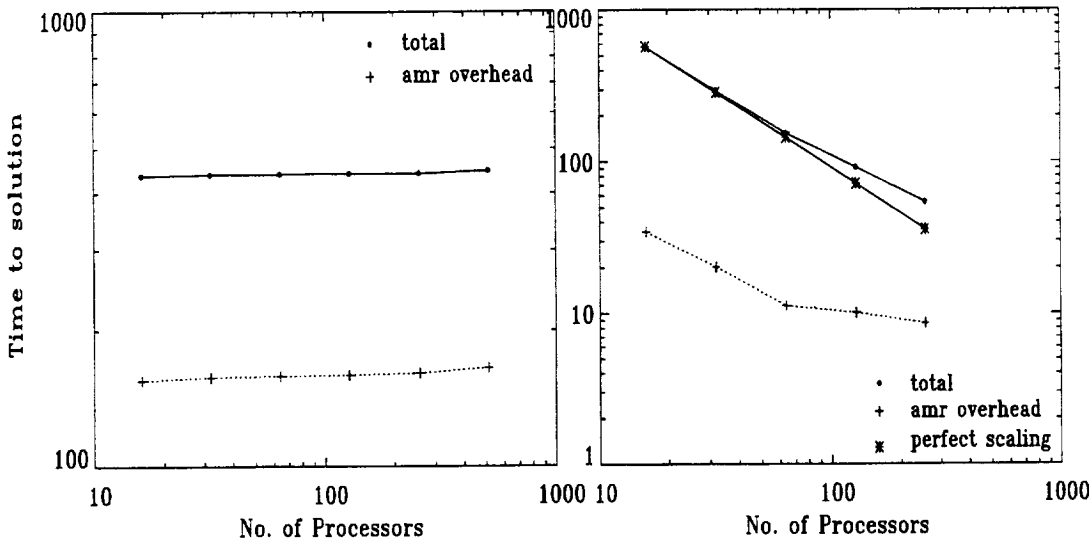

Figure 6: The execution time for the different components of the FCTMHD3D code on the T3E, for the scale-up test (left frame), and the fixed size test (right frame). The ideal scaling curve for the fixed size problem is shown for reference.

The next test we performed was designed to measure scaling for a problem with a fixed total amount of work. This calculation was done in two dimensions and with a maximum refinement level of 7 (i.e. the finest resolution is the same as the resolution of a uniform 512x512 mesh). The scaling of total execution time is shown in Figure 9. The scaling is poor, principally because the AMR overhead does not scale well, particularly at large processor numbers. For comparative purposes we also show in figure 9 the scaling curve for a uniform mesh version of the FLASH code run for exactly the same problem and which was run at the same effective resolution as the version using PARAMESH. This curve clearly shows that the uniform mesh code scales well. This code achieved 40 Mflops per processor using double precision arithmetic. Note, however that in spite of its scaling deficiencies the time to solution for the FLASH code using PARAMESH is still better than the uniform mesh code for ALL processor numbers for which we collected timing results.

9.3 Time to Solution

To test whether the time to solution achieved using PARAMESH is indeed improved when compared to a uniform mesh code running the same problem, we constructed a test where the initial condition had a uniform density and zero velocity in a square computational domain of size 1 unit on a side. The

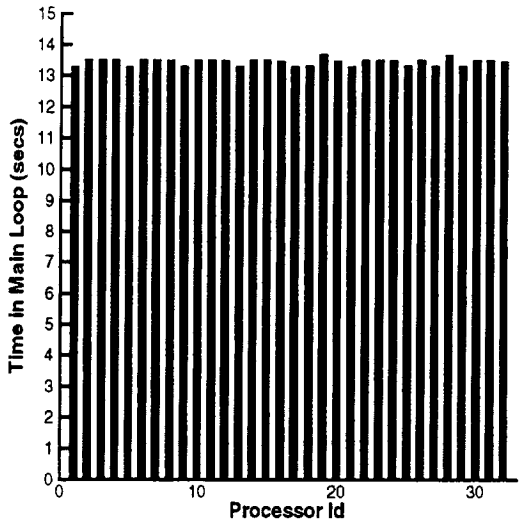

Figure 7: The execution time of the corrector step of the transport phase of the FCTMHD3D code during a typical timestep on each processor for a 32 processor run. The ratio of the average height of these histogram bars to the maximum height is a measure of the load balance.

pressure was set such that it had a high value within a radius of 0.3 and a low value everywhere else. This state was then evolved for a fixed number of time steps using both the uniform version of the FLASH code and the AMR version. Timing results were obtained for different resolutions and are shown in Figures 10 and 11. Here we plot the highest level of refinement used vs. the time to solution for both the AMR and uniform versions of the FLASH code. For reference a refinement level of 5 corresponds to a uniform mesh with a resolution of 128 points on a side. Each increase in refinement by one level corresponds to a factor of 2 in linear resolution. For these tests an identical two dimensional calculation was run on two different computer architectures using different numbers of processors in each case. The computers used were 16 processors of an SGI Origin 2000 and 256 processors of the INTEL based ASCI RED machine located at Sandia National Laboratories.

These plots show that the AMR version of the FLASH code gets better times to solution except for the coarsest resolutions. These plots also show that the time to solution using AMR becomes much better relative to the uniform version of the FLASH code as more and more levels of refinement are added. This is due to the fact that refinement is only placed in areas of the domain where it is needed and the fraction of the area of the mesh which is refined to the finest level decreases with increasing resolution. One can also see by comparing these plots that the curves representing the time

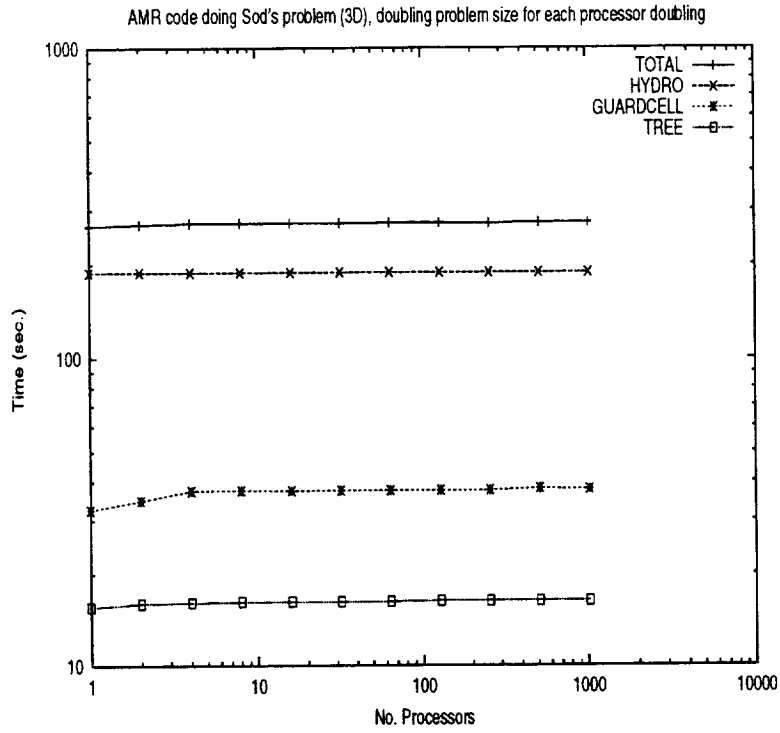

Figure 8: Time to Solution as a function of the number of processors for a problem which has a fixed amount of work per processor.

to solution using the AMR FLASH code and the uniform FLASH code cross at a finer resolution for the cases run on 256 processors. This is due to the fact that for the more coarsely refined cases the total number of blocks is small relative to the number of processors and some processors have only a small amount of work (see discussion below).

9.4 Discussion of Performance

We have run two types of scaling tests, fixed problem size and for fixed work per processor.

For problems with fixed work per processor we see excellent scaling, both with and without shared memory. Our tests illustrate that PARAMESH adds overhead which is small compared with the cost of the core algorithm for typical fluid and MHD codes.

The scaling for problems with fixed size is not as good. It is reasonably good for the small processor range, but deteriorates at large numbers of processors. This behavior is not surprising. It is due in part to the nature of block adaptive AMR, and in the case of the FLASH results, also due to inefficiencies in the way PARAMESH uses MPI.

Block adaptive schemes will scale reasonably well as long as each proces-

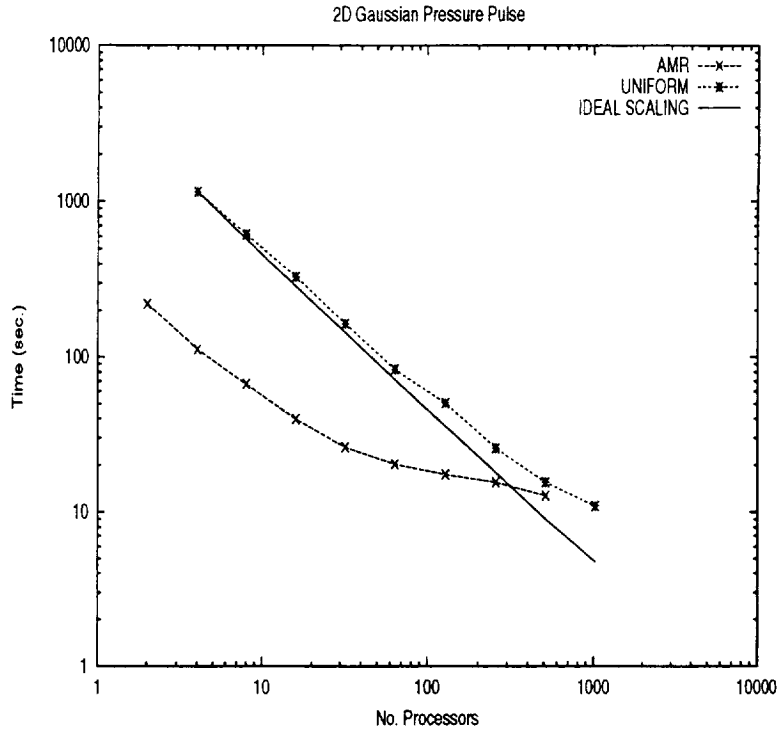

Figure 9: Time to Solution as a function of the number of processors for FLASH for a fixed problem size.

sor has a large enough number (i.e. ≥ 10) of grid blocks on which to work. As the number of blocks per processor decreases we expect the scaling to deteriorate. The reasons for this are,

- it is harder to balance the communication and computational work load when each processor has very few blocks
- the probability that a given processor's neighbors are off processor, increases as the number of blocks per processor decreases, altering the balance of communication and computation.

When performing scaling measurements on problems with fixed size, a problem which fills memory on one processor will fill only 0.1% of memory on one thousand processors. A typical user would never consider running this problem on one thousand processors. Rather, they would increase the size of their model, or run on fewer processors. We contend therefore that the only part of these fixed size scaling curves which we show which relate to the way a typical user would operate are at the lower processor number range.

In the absence of shared memory, we expect much poorer performance with the current release of PARAMESH, because MMPI relies on high la-

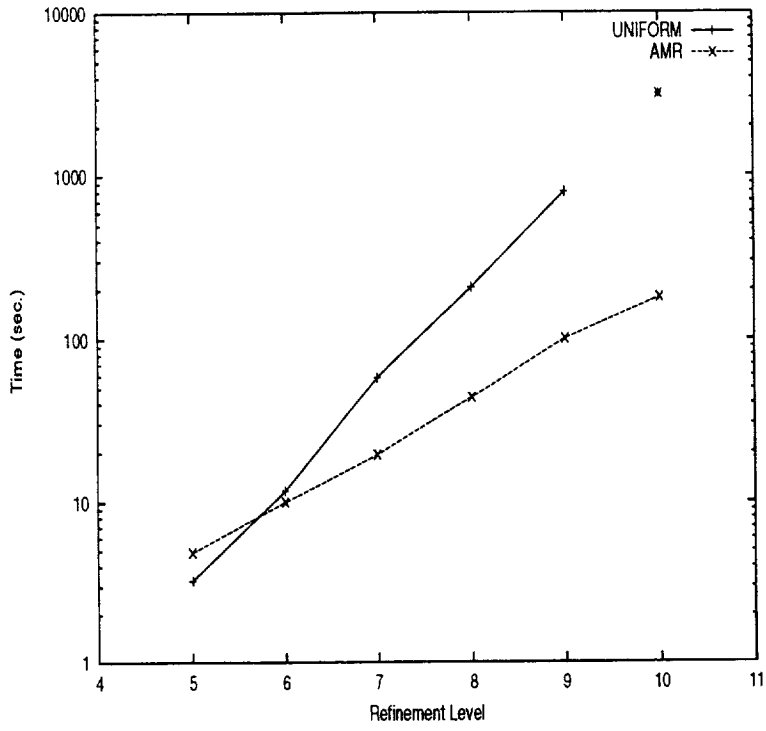

Figure 10: Time to Solution as a function of resolution for the AMR FLASH code and the uniform FLASH code. The starred point represents an estimate of the running time for the uniform code since this case did not fit in memory. The curves shown were run in two dimensions using 16 processors of an SGI Origin 2000.

tency system interrupts to manage the many one-sided communications, blocking gets, and barriers in our code.[2] This is borne out in the scaling curves for the FLASH code from the ASCI RED machine.

Given all these considerations with regard to scaling of fixed sized problems, our results still show that the performance is good enough to substantially improve on the time to solution achieved by uniformly refined calculations when high resolution is required.

10 Installation and Test Runs

PARAMESH will run on any parallel UNIX system which has a Fortran 90 compiler, with a preprocessor, and the SGI/Cray SHMEM library or MPI. The SHMEM and MPI libraries are not required when running on a single processor.

[2]This situation should improve significantly with the release of MPI2. We are currently engaged in rewriting critical routines to improve performance in this regard.

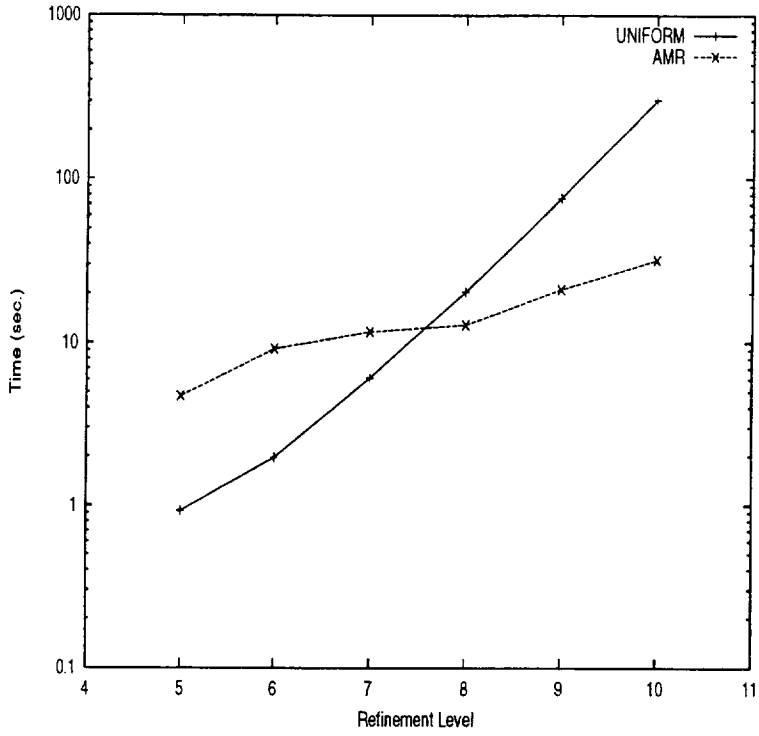

Figure 11: Time to Solution as a function of resolution for the AMR FLASH code and the uniform FLASH code. The curves shown were run in two dimensions using 256 processors of the ASCI RED machine.

The PARAMESH package is distributed as a UNIX tar file. To install the package, simply untar it. When this is done a hierarchy of sub-directories is created below the current directory. This is illustrated in Figure 12. The current directory will then contain a README file. A comprehensive user's manual is included in the 'Users_manual' sub-directory, which includes detailed instruction on how to build applications. Makefile templates are provided.

In the subdirectory Tests we have included a number of test programs designed to verify that the package has been installed correctly. A README file in this sub-directory gives instruction on how to build these test programs.

Directory Structure of the Package

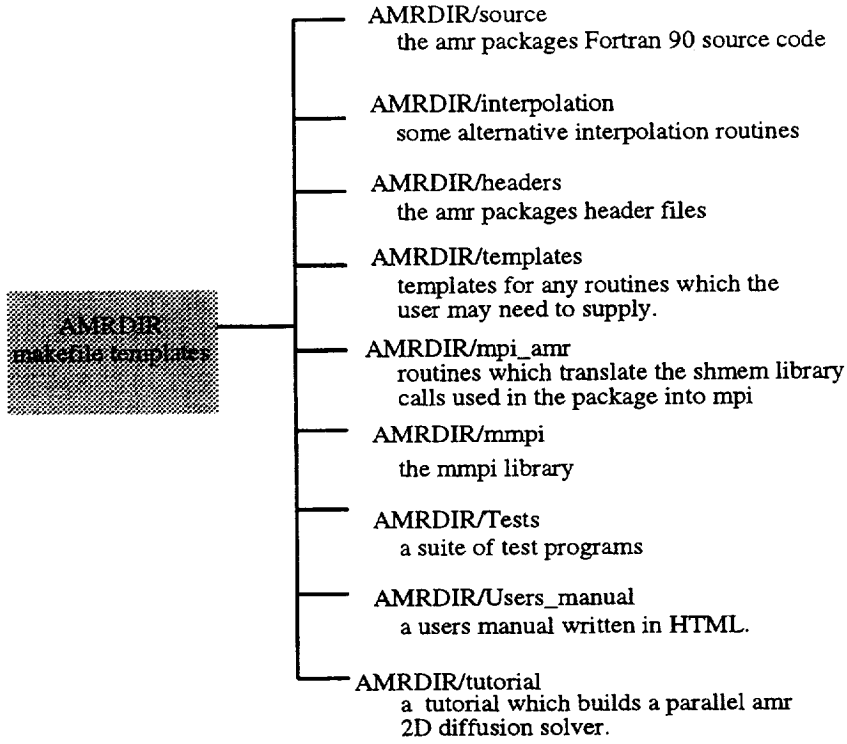

Figure 12: The organization of the PARAMESH package distribution file system.

TEST RUN OUTPUT

In the subdirectory Tests we have included a number of test programs designed to verify that the package has been installed correctly. A README file in this sub-directory gives instruction on how to build these test programs. Makefiles are provided to build these test programs, for Silicon Graphics machines running IRIX, and for the SGI/Cray T3E. For any other combination of machine and operating system these makefiles may require slight modification. The corresponding author can be contacted for assistance with these modifications.

To verify that the package has been installed properly run one or more of the test programs. Check the last line of the output. It reports the number of errors detected during the automatic testing process. If the test is successful it will say

No errors detected - Test Successful.

The test programs require 15 MBytes of memory.

Acknowledgements

This work was supported by the NASA Earth and Space Sciences High Performance Computing and Communications program. K. Olson acknowleges support from the ASCI Flash Center at the University of Chicago under DOE contract B341495. We would like to thank Dr. Daniel S. Spicer for his assistance with the 2D MUSCL gas dynamics code, and Dr. Rick DeVore who provided the original FCTMHD3D code. Dr. Bruce Fryxell, Dr. Andrea Malagoli, and Dr. Bob Rosner are also gratefully acknowledged for many useful discussions.

References

[1] R. Löhner, *Computer Methods in Applied Mechanics and Engineering* **61** (1987) 323.

[2] M.J. Berger, Ph.D Thesis, Stanford University, (1982).

[3] M.J. Berger and J. Oliger, *J. Comput. Phys.* **53** (1984) 484.

[4] M.J. Berger and P. Colella, *J. Comput. Phys.*, **82** (1989) 64.

[5] D. De Zeeuw and K.G. Powell, *J. Comput. Phys.* **104** (1993) 56.

[6] J.J. Quirk, Ph.D Thesis, Cranfield Institute of Technology, (1991).

[7] A.M. Khokhlov, NRL Memo 6404-97-7950, (1997).

[8] S. Mitra, M. Parashar and J.C. Browne, Dept. of Computer Sciences, Univ of Texas at Austin, 1997. at http://www.caip.rutgers.edu/~parashar/DAGH/.

[9] H. Neeman, at http://zeus.ncsa.uiuc.edu:8080/~hneeman/hamr.html.

[10] D. Quinlan, at http://www.llnl.gov/casc/people/quinlan.

[11] S. Kohn, X. Garaizar, R. Hornung, and S. Smith, at http://www.llnl.gov/CASC/SAMRAI

[12] R. LeVeque and M. Berger,
at http://www.amath.washington.edu:80/~rjl/amrclaw/.

[13] M. S. Warren and J. K. Salmon, in: *Proc. Supercomputing '93* (IEEE: Computer Society, Washington D.C., 1993) 12.

[14] H. Yang, *J. Comput. Phys.* **89** (1990) 125.

[15] P. Colella, *J. Comput. Phys.* **87** (1990) 200.

[16] D. Balsara, *Astrophys. J.Supp.* **116** (1998) 133.

[17] S.K. Antiochos, P. MacNeice, D.S. Spicer, and J.A. Klimchuk, *Astrophys. J.* **512** (1999) 985.

[18] C.R. DeVore, at http://www.lcp.nrl.navy.mil/hpcc-ess/index.html (1997).

[19] P. Colella and P. Woodward, *J. Comput. Phys.* **54** (1984) 174.

[20] F. X. Timmes , K. Olson, P. Ricker, M. Zingale, B. Fryxell, P. MacNeice, H. Tufo, D. Q. Lamb, and R. Rosner, In preparation (1999).

[21] B. Fryxell, E. Muller, and D. Arnett, *Hydrodynamics and Nuclear Burning*, Max-Plank-Institute fur Astrophysik Report 449 (1989).

[22] F. X. Timmes and D. Arnett, *Astrophys. J. Suppl.* in press (1999).

[23] F. X. Timmes and D. Swesty, *Astrophys. J. Suppl.* in press (1999).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1999	Contractor Report

4. TITLE AND SUBTITLE
PARAMESH: A Parallel Adaptive Mesh Refinement Community Toolkit

5. FUNDING NUMBERS
NAG5-6029

6. AUTHOR(S)
P. MacNeice, K.M. Olson, C. Mobarry, R. de Fainchtein, and C. Parker

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS (ES)
Goddard Space Flight Center
Greenbelt, Maryland 20771

8. PEFORMING ORGANIZATION REPORT NUMBER
2000–00298–0

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS (ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
CR—1999–209483

11. SUPPLEMENTARY NOTES
P. MacNeice: Drexel University, Philadelphia, PA; K.M. Olson: Enrico Fermi Institute, University of Chicago, Chicago, IL; R. de Fainchtein: Raytheon ITSS, Lanham, Maryland; C. Packer: (formerly of Raytheon ITSS, Lanham, Maryland)

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified–Unlimited
Subject Category: 61
Report available from the NASA Center for AeroSpace Information,
7121 Standard Drive, Hanover, MD 21076-1320. (301) 621-0390.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

In this paper, we describe a community toolkit which is designed to provide parallel support with adaptive mesh capability for a large and important class of computational models, those using structured, logically cartesian meshes. The package of Fortran 90 subroutines, called PARAMESH, is designed to provide an application developer with an easy route to extend an existing serial code which uses a logically cartesian structured mesh into a parallel code with adaptive mesh refinement. Alternatively, in its simplest use, and with minimal effort, it can operate as a domain decomposition tool for users who want to parallelize their serial codes, but who do not wish to use adaptivity. The package can provide them with an incremental evolutionary path for their code, converting it first to uniformly refined parallel code, and then later if they so desire, adding adaptivity.

14. SUBJECT TERMS
adaptive mesh, PARAMESH, cartesian mesh, parallel code, Fortran 90, computational models

15. NUMBER OF PAGES
35

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL