NV-64
31P

NASA Contractor Report 191530
ICASE Report No. 93-63

ICASE

A PARALLEL ADAPTIVE MESH REFINEMENT ALGORITHM

(NASA-CR-191530) A PARALLEL ADAPTIVE MESH REFINEMENT ALGORITHM (ICASE) 31 p
N94-15724
Unclas
63/64 0190904

James J. Quirk
Ulf R. Hanebutte

NASA Contract No. NAS1-19480
August 1993

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23681-0001

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-0001

A Parallel Adaptive Mesh Refinement Algorithm

James J. Quirk[1]
and
Ulf R. Hanebutte[1]

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23681, USA.

Abstract

Over recent years, Adaptive Mesh Refinement (AMR) algorithms which dynamically match the local resolution of the computational grid to the numerical solution being sought have emerged as powerful tools for solving problems that contain disparate length and time scales. In particular, several workers have demonstrated the effectiveness of employing an adaptive, block-structured hierarchical grid system for simulations of complex shock wave phenomena. Unfortunately, from the parallel algorithm developer's viewpoint, this class of scheme is quite involved; these schemes cannot be distilled down to a small kernel upon which various parallelizing strategies may be tested. However, because of their block-structured nature such schemes are inherently parallel, so all is not lost. In this paper we describe the method by which Quirk's AMR algorithm has been parallelized. This method is built upon just a few simple message passing routines and so it may be implemented across a broad class of MIMD machines. Moreover, the method of parallelization is such that the original serial code is left virtually intact, and so we are left with just a single product to support. The importance of this fact should not be underestimated given the size and complexity of the original algorithm.

While the parallel version currently lacks some of the advanced features of the serial version, it is sufficiently mature that it can be used routinely to perform very large scale simulations of detonation phenomena using workstation clusters. Hence the parallel algorithm has progressed beyond the level of being solely an exercise in computer science to become a powerful research tool for investigating fluid phenomena. Finally, although it will be seen that we have produced a fair amount of paraphernalia to parallelize just a single algorithm, it should be appreciated that the AMR algorithm is itself sufficiently general to be applicable to a large class of problems. And so the method described here could be legitimately construed as being a template for parallelizing block-structured, adaptive grid algorithms.

[1]This research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681.

1 Introduction

Despite the enormous potential power offered by parallel computers, it is worth illustrating that brute force calculations are unlikely to be of much use for solving problems that contain disparate physical length scales. Consider the following example taken from the study of detonation waves.

The usefulness of explosive materials stems from their ability to rapidly convert chemical energy into heat energy. For example, a good solid explosive converts energy at a rate of the order 10^{10} watts per square centimetre of its detonation front. Thus, as noted by Fickett and Davis[7], a 20 m square detonation wave operates at a power level equal to all the power the earth receives from the sun! For a given explosive, the rate of energy release essentially depends on the speed with which a detonation wave is propagated. Traditionally, detonation speeds are determined from experiment. For solid explosives, a cylindrical charge known as a rate-stick is ignited at one end, and the propagation speed is measured at the other end. It is assumed that the length of the stick is sufficient to allow the detonation front to reach its nominally steady speed. Note that the leading part of a detonation front is a strong shock wave. As this wave propagates, so the explosive material is compressed and thus heats up. This raise in temperature triggers a chemical reaction which releases large amounts of energy in the form of heat. This energy release provides motive force for the shock, and a balance is reached such that the chemical reaction supports a nominally steady speed of shock propagation.

The simulation of a rate-stick experiment represents a formidable challenge. Since the chemical reaction drives the shock wave, the simulation must be able to resolve the reaction zone accurately. Results for model problems suggest that at least 10 mesh cells are needed across the width of the reaction zone. Now for certain types of solid explosive the reaction zone may be only 0.02 mm in thickness, in which case the mesh spacing within the reaction zone must be no larger than 0.002 mm. Given that a rate-stick might be 100 mm in length and 100 mm in diameter, some 1.25×10^9 cells would be required for an axisymmetric flow calculation on a uniform mesh. From the point of view of numerical accuracy, it is unlikely that the detonation front could be propagated by more than one mesh cell per time step. Consequently it would take some 5×10^4 time steps for the detonation to travel the full length of the rate-stick. Therefore the total workload for the simulation would be of the order of 6.25×10^{13} cell updates. Such a calculation would be absurd. A 1 Gflop computer might be capable of 10^6 mesh updates per second, in which case the calculation would take 723 days to run! Clearly, to make such a simulation viable, something other than a large computer is required, hence the need for adaptive mesh refinement.

Adaptive mesh schemes attempt to match dynamically the local resolution of the computational grid to the requirements of the evolving flow solution. Thus very fine mesh cells

are restricted to those regions where they are needed, and elsewhere the computational grid may be quite coarse. Such a strategy can dramatically reduce the computational effort required to perform simulations of problems that contain disparate scales. Returning to our detonation simulation, if the fine mesh cells were restricted to the vicinity of the reaction zone, only about 2.5×10^5 cells would be required for the simulation. In which case the simulation would only require of the order of 1.25×10^{10} cell updates. Therefore, whereas the uniform mesh simulation might take 723 days to run, the adaptive mesh simulation would take just 208 minutes!

Because the potential savings are so large, the adoption of almost any form of mesh adaption, no matter how naïve, will pay some dividend. Consequently, a wide variety of strategies have been utilized[11]. For the simulation of complex shock wave phenomena, however, the AMR algorithm first developed by Berger[1, 2], and later by Quirk[13], and by Fischer[8], has proved to be particularly effective. Despite its effectiveness, it is clear that the long term usefulness of the AMR algorithm will depend on the extent to which it can exploit parallel computing engines. The aim of this paper is to show that although the AMR algorithm is quite involved, and as such it may be thought to be an unsuitable candidate for parallelization, in actuality the algorithm has sufficient inherent parallelism so as to be a good candidate for running on MIMD machines. Before proceeding further it should be acknowledged that Berger and Saltzman[3] have already had some success using the AMR algorithm on a SIMD machine. But since this architecture necessitates an approach very different from the one described here their work will not be considered further.

The rest of this paper is as organized follows. In Section 2 we present some background information for the AMR algorithm together with some of the implementation details for the original serial algorithm so that the reader might better understand the method of parallelization which is described in Section 3. In Section 4 we present two set of results which demonstrate the worth of the new parallel version of AMR. The first set of results will be of more interest to the computer scientist, for it deals with issues such as how well does the algorithm scale. The second set of results will be of more interest to the applications scientist, since it shows how the parallel algorithm might be used in earnest; results are presented from a study of the Mach reflection of a detonation wave by a ramp. Lastly, in Section 5 we present some conclusions that we have drawn from this work.

2 The AMR Algorithm – A Primer

The AMR algorithm is a general purpose mesh refinement scheme that has primarily been used for studying shock wave phenomena[2, 6, 14]. The purpose of this section is to provide a primer for those readers who are unfamiliar with the algorithm in order that they might appreciate the issues that shaped our method of parallelization. It will soon become obvious

that the AMR algorithm is quite involved, and so here we can do little more than describe what constitutes the algorithm. No real attempt is made to describe why the algorithm chooses to do things a certain way, nor how it actually performs certain tasks. Those interested in the full details are strongly recommended to read Quirk's thesis[13]. The exposition given here is broken down into four parts. First, we describe the grid structure used by the algorithm, for all other aspects of the scheme stem from this structure. Second, we outline the process which integrates the discretized flow solution contained by the grid structure. Third, we outline the process whereby the grid dynamically adapts to the evolving flow solution, and finally, we close this primer by presenting some pseudo-code fragments which show how the AMR algorithm is organized.

2.1 Grid Structure

The AMR algorithm employs a hierarchical structure of embedded meshes to discretize the flow domain. The bottom of the hierarchy, level 0, consists of a set of coarse mesh patches which delineate the flow domain. Each of these mesh patches forms a logically rectangular unit of cells. These patches are restricted such that it is possible to reference all their cells by a single (i, j) co-ordinate system, $\mathbf{C}_0^2$, as shown in Figure 1. This restriction ensures that there is continuity of grid lines between neighbouring patches and that if two patches overlap one another then the regions of overlap are identical. These mesh patches form the effective grid at level 0, G_0, and we identify the k^{th} patch by $G_{0,k}$. Usually the terms mesh and grid are synonymous, but throughout this work we reserve the term mesh for a single logically rectangular patch and the term grid for a collection of such patches.

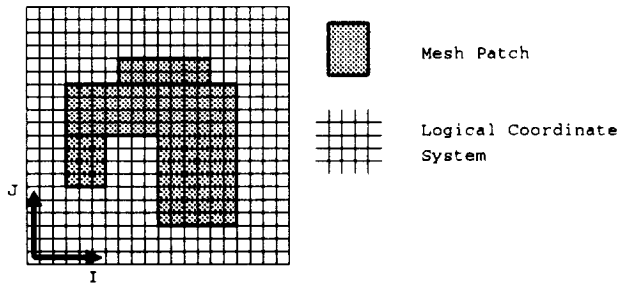
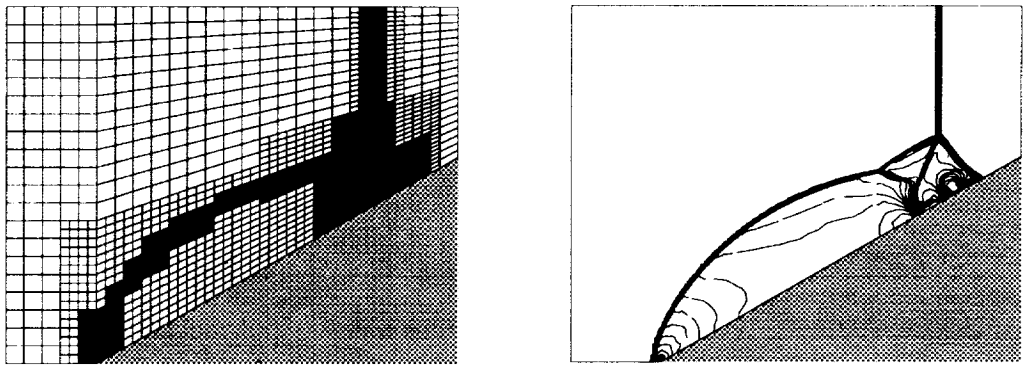


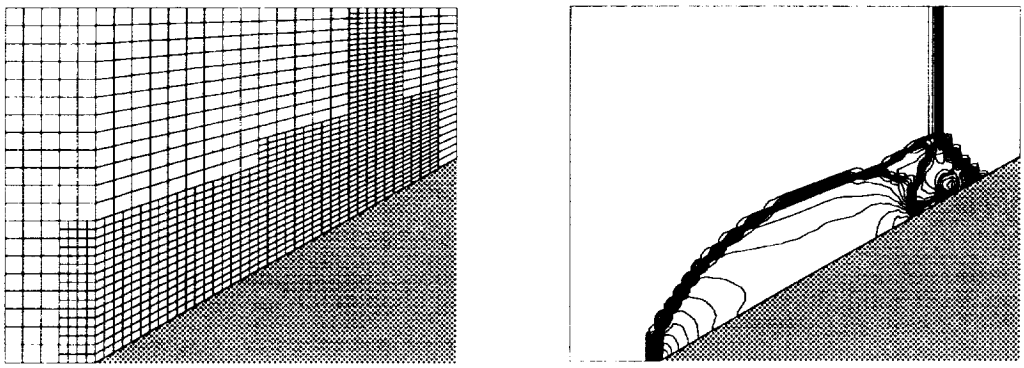
Figure 1: All meshes are fixed relative to a logical co-ordinate system.

The flow domain may be refined locally by embedding finer mesh patches into the coarse grid at level 0 to form the next grid level within the hierarchy, G_1. These embedded patches are formed by linearly sub-dividing rectangular groups of coarse cells. The choice for the number of sub-divisions along the edges of a coarse cell is arbitrary, but it must be the same for every coarse cell that is refined. This restriction enables every mesh cell contained at level 1 to be referenced by its own (i, j) co-ordinate system, $\mathbf{C}_1^2$. In their turn, these

embedded patches may contain even finer embedded patches which form the grid G_2. This process of refinement may be repeated as desired up to some level l_{max}. The grids at different levels within the hierarchy co-exist, for underlying an embedded fine grid there is a complete coarse grid and a complete coarse field solution, see Figure 2. Note that the discretized flow solution is taken to be a cell-centred projection of the true solution.



Grid and density contours for all 3 grid levels



Grid and density contours for just the lower 2 grid levels

Figure 2: Coarse grids exist beneath fine grids.

In order to specify an arbitrary grid structure, G, it is necessary to supply the spatial refinement factors rI_l and rJ_l for each grid level, l, together with the extent of each mesh patch, $G_{l,k}$, using $\mathbf{C}_l^2$ co-ordinates. The extent of a patch is just given by the co-ordinates of its lower-left and upper-right corners. All other details for the grid structure can be gleaned from this basic data. For example, since a simple relationship exists between the co-ordinate systems $\mathbf{C}_l^2$ and $\mathbf{C}_{l-1}^2$,

$$i_{l-1} = \frac{(i_l - 1)}{rI_l} + 1 \qquad \text{and} \qquad j_{l-1} = \frac{(j_l - 1)}{rJ_l} + 1,$$

it is possible to determine which patches at level $l-1$ underpin a given patch at level l.

For convenience we chose to store the following variables for the original serial implementation of the grid structure.

LMAX	l_{max}, maximum grid level.

For each grid G_l:

NGA(L)	nGl, number of grids contained by G_l.
GP(L)	Gp_l, grid index pointer.
rI(L)	rI_l, Spatial refinement factor in I direction.
rJ(L)	rJ_l, Spatial refinement factor in J direction.

For each mesh patch $G_{l,k}$:

GRD = GP(L)+K	
LGRID(GRD)	Grid level for the mesh $G_{l,k}$.
IMX(GRD)	$IM_{l,k}$, Width of mesh $G_{l,k}$.
JMX(GRD)	$JM_{l,k}$, Height of mesh $G_{l,k}$.
JNf(GRD)	$\square_{l,k}$, extent of mesh $G_{l,k}$ using $\mathbf{C}_l^2$ co-ordinates.
JSf(GRD)	
IEf(GRD)	
IWf(GRD)	
JNc(GRD)	$\square_{l,k}^c$, extent of mesh $G_{l,k}$ using $\mathbf{C}_{l+1}^2$ co-ordinates.
JSc(GRD)	
IEc(GRD)	
IWc(GRD)	

Note that the storage overheads are quite small, just 11 variables are used for each patch. Also note that we have assigned a unique index to each mesh patch; the grid index pointer Gp_l satisfies the recurrence relation, $Gp_{l+1} = Gp_l + nG_l$, with Gp_0 set to zero. Considering that an average patch might contain upwards of 1000 cells, the overhead per cell is negligible. A pair of nested DO loops is all that is required to run through the grid structure and operate on every mesh.

```
DO L=0,LMAX
    DO K=1,NGA(L)
        GRD = GP(L)+K
        ...
        Operate on mesh G_{l,k}
        ...
    END DO
END DO
```

The discretized flow solution is stored in a series of large lists or heaps; a separate heap is used for each variable. Each heap contains a contiguous set of blocks, one block for each

mesh, and each block consists of a contiguous set of storage elements with one element for each cell of the mesh. It is convenient to view a mesh patch as being surrounded by a border of dummy cells, two cells deep. Thus $(IM + 4) \times (JM + 4)$ storage elements are required for each mesh patch. The head of each block is found by indirection through the list, H2PTR, thus the information for the ij^{th} cell of the grid, GRD, would be located at,

$$\text{H2PTR(GRD)} + (\text{I} + 2) + (\text{J} + 1) * (\text{IMX(GRD)} + 4).$$

Note that the cells within a mesh patch are stored by rows. The following code fragment would access every mesh cell in the data structure; the subroutine UNPACK_GRID computes the location pointer, IJ, for the cell $(1, 1)$ and returns the stride lengths Ibmp and Jbmp for the specified mesh.

```
DO L=0,LMAX
    DO K=1,NGA(L)
        GRD = GP(L)+K
        CALL UNPACK_GRID(GRD,1,1,IJ,Ibmp,Jbmp)
        DO I=1,IMX(GRD)
            IJo = IJ
            DO J=1,JMX(GRD)

                IJ contains a pointer to the data stored
                for the ij^th cell of G_{l,k}.

                IJ = IJ +Jbmp
            END DO
            IJ = IJo+Ibmp
        END DO
    END DO
END DO
```

Connectivity information which is needed along mesh boundaries, such as which mesh patches abut a given patch, is also stored using linked lists, but owing to a lack of space no details can be given here.

2.2 Flow Integration

In principle any cell-centred, flux-based scheme developed for a single topologically regular mesh can form the basis for the flow integration process. The dummy cells which surround each mesh patch are the key to this flexibility. They effectively turn cell interfaces along mesh boundaries into internal interfaces. Prior to integrating a grid, the dummy cells for every mesh patch contained by the grid are primed with data. Each mesh patch is

then processed independently of every other mesh patch by some user supplied, black-box integrator that never actually sees a boundary. This is possible because the data used to prime the dummy cells is chosen such that the resultant numerical fluxes along mesh boundaries are consistent with the various boundary conditions that have to be met.

Consider the *fine-coarse* boundary shown in Figure 3. The AMR algorithm refines in time as well as space. So for a refinement ratio of 4 say, a fine mesh patch will be integrated 4 times with $\frac{1}{4}$ the size of the time step of its underlying coarse patch. The order of integration is always from coarse to fine, thus the coarse patch flow solution may be interpolated in space and time to provide Dirichlet boundary conditions for the fine patch. For multi-level calculations, the integrations of the various separate grid levels are recursively interleaved so as to minimize the time span over which interpolation is required.

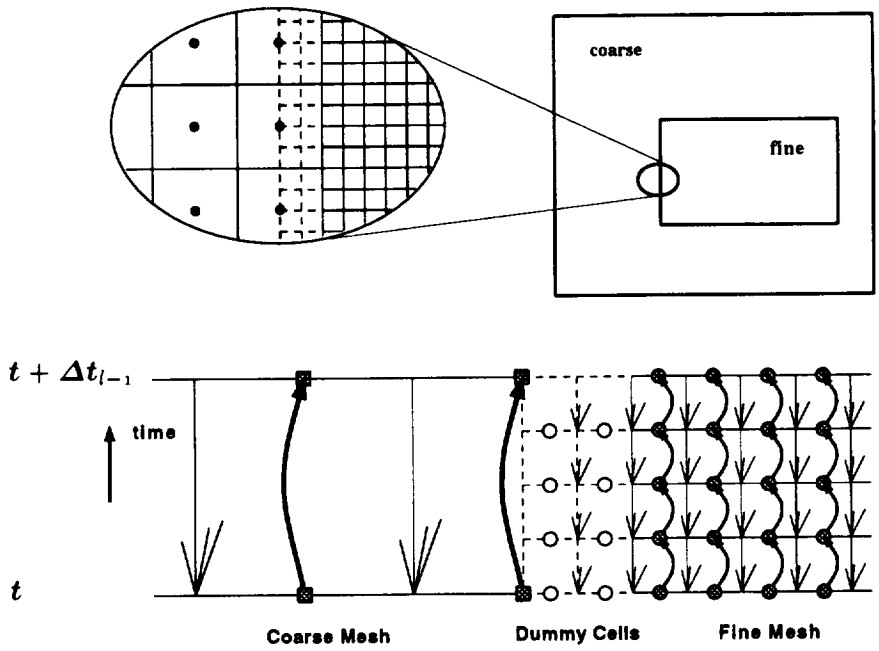


Figure 3: A *fine-coarse* internal boundary.

Only two other types of mesh boundary exist: *fine-fine* boundaries, where the dummy cells of one mesh patch exactly overlap the mesh cells of another patch at the same grid level, in which case the appropriate data for the dummy cells is simply shovelled directly from the relevant mesh cells; *external* boundaries, where the data for the dummy cells can be inferred locally, for example, at a solid wall a simple reflection procedure is applied in the usual manner.

The only other operators that form part of the AMR flow integration process are a restriction operator which projects the flow solution contained by a fine mesh patch on to its underlying coarse mesh patch; this is required for consistency purposes. And a fixup

operator which is applied along a *fine-coarse* boundary whenever a conservative integration process is required. Essentially the fixup operator modifies the updated coarse cell solutions along a *fine-coarse* boundary using the cumulative fluxes across the boundary as seen by the fine patch during its sub-cycle of smaller integration time steps.

2.3 Elements of the Adaption Process

Consider a planar shock wave travelling down a duct, from left to right. Suppose a coarse grid, G_0, is used to discretize the duct and further suppose that an embedded grid, G_1, which is finer than G_0 covers the vicinity of the shock, see Figure 4 (a). Now, if the flow solution on this grid structure is integrated forward in time, sooner or later the shock will move to within one mesh cell of the right-hand edge of the grid G_1, see Figure 4 (b). The shock is about to run off the edge of the embedded mesh. At the very least this act will cause the shock to smear to its natural width on the coarse grid thus lowering the resolution of the simulation. But if the shock is strong, it will also introduce spurious oscillations into the flow solution. The AMR algorithm avoids such problems by dynamically adapting the grid structure to the evolving flow. Here, the adaption process results in the embedded grid, G_1, gliding along the coarse grid, G_0, so as to keep pace with the moving shock front.

The adaption process may be split loosely into three tasks. First, given a grid structure and flow solution, regions of interest are identified. These regions will be refined, that is they will be covered by an embedded grid. For our duct example the adaption process looks at the solution on the grid G_0. Using some *ad hoc* monitor function such as the local density gradient, the cells in the vicinity of the shock front are flagged for refinement as shown in Figure 4 (c). Second, the flagged cells are grouped into clusters using a recursive, area subdivide algorithm. Each of the clusters so produced is then covered by a single embedded mesh patch. This grouping process results in a new grid structure, $\tilde{G}$, see Figure 4 (d). Temporarily, as shown in Figure 4 (e), there are now two grid structures, but the new grid structure does not have a flow solution. Finally, the solution from the old structure is transferred to the new one, see Figure 4 (f). If this sequence of tasks is performed repeatedly, the embedded grid will shadow the moving shock front.

For multi-level calculations, a grid is adapted whenever it has completed its sub-cycle of integrations relative to the coarse grid which underpins it. Thus the order of adaption dovetails with the recursive order of integrating the different grid levels. Whenever it is necessary to adapt more than one grid level at once, the adaption process always proceeds from fine to coarse. Following the adaption of a fine grid, the adaption process for the coarse grid at the next level down must ensure that any new coarse grid that may be produced fully supports the finer grid. This job of ensuring "proper nesting" can only be done if the order of adaption is from fine to coarse.

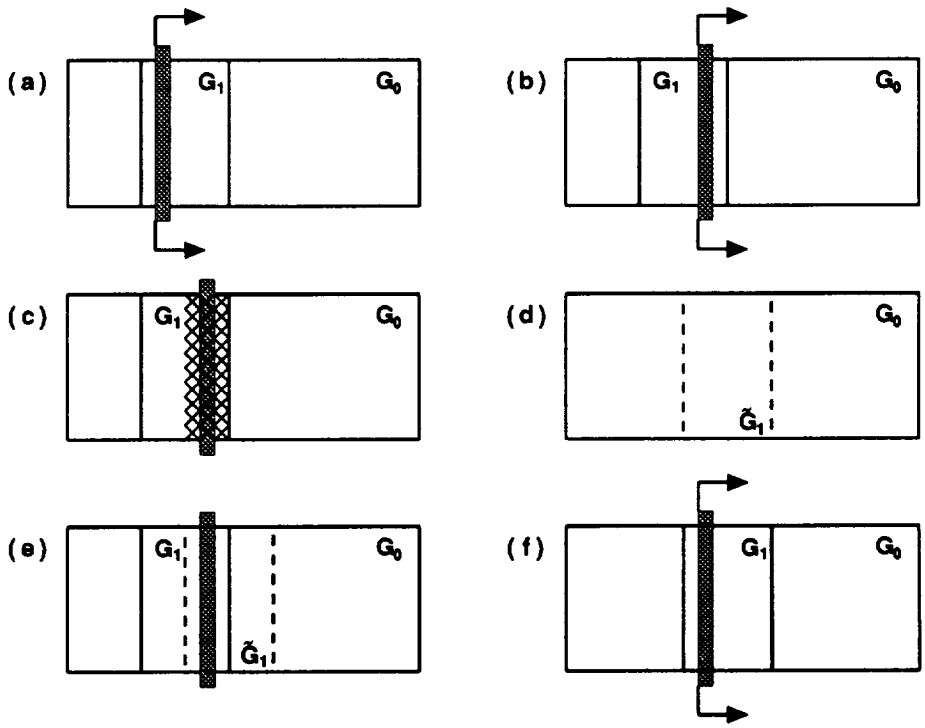


Figure 4: The adaption process for a shock moving down a duct.

2.4 Pseudo-Code Fragments

The following fragments of pseudo-code give an overview of the complete AMR algorithm. We have chosen to present these fragments since they provide a succinct means of describing the closely coupled nature of the AMR algorithm. It is not our intention that they be cribbed verbatim in order to build a working code. Indeed, this would not be possible since for reasons of clarity much clutter has been omitted.

First a simple harness is required to drive the algorithm.

```
repeat {
  Find_Stable_Time_Step
  AMR(0,1)
} until The_Calculation_Is_Finished
```

This harness simply iterates until the computation is finished. For each iteration a set of stable time steps is computed just prior to calling the algorithm proper; a separate time step Δt_l is required for each grid G_l. The procedure **AMR** orchestrates the recursive interleaving of the integrations and adaptions of the various grid levels.

```
Procedure AMR(l,Nt)
  Integrate_Grid(l,Nt)
  if l < l_max {
    Initialise_Conservative_Fixup(l)
    for Nft = 1 to rt_l {
      AMR(l + 1,Nft)
    }
    Apply_Conservative_Fixup(l + 1)
    Project_Solution(l + 1)
    Set_bc(l,Nt)
    if Nt < rt_l Adapt(l)
  }
End Procedure
```

The parameters l and N_t specify the grid level to be operated on and the integration, sub-cycling index, respectively. The procedure **Integrate_grid** advances the flow solution held on the grid G_l forward by time Δt_l. This is done by first priming the dummy cells for each mesh patch contained in the grid via a call to the procedure **Set_BC**. Note that the sub-cycling index N_t is required so as to be able to perform the correct interpolation in time at a *fine-coarse* boundary. Once all the dummy cells have been primed, a call to the user-supplied routine **Integrate_Mesh** is invoked for all nG_l patches contained by the grid G_l. The remaining two routines form part of the conservative fixup procedure, the details of which need not concern us here.

```
Procedure Integrate_Grid(l,Nt)
  Set_BC(l,Nt)
  for k = 1 to nG_l {
    Integrate_Mesh(Gp_l + k)
    Save_Coarse_Fluxes(Gp_l + k)
    Integrate_Fine_Fluxes(Gp_l + k)
  }
End Procedure
```

Following the integration of the grid G_l it may be necessary to recursively call the procedure **AMR** so as to integrate the next finer grid, G_{l+1}, for its rt_{l+1} sub-cycles. If not, a call to the routine **Apply_Conservative_Fixup** uses the cumulative flux totals to correct those coarse cells in G_{l-1} which abut the grid G_l. Following which, for consistency, the solution held on G_l is projected on to the coarse grid G_{l-1}. Once the subcycling is complete for a particular grid level, it must be adapted; this is done via a call to the procedure **Adapt**, the innards of which should be clear from having read Section 2.3.

```
Procedure Adapt(l_1)
  Initialise_G̃
  for l = l_max − 1 down to l_1 {
    Set_Refine_Flags(l)
    Ensure_Proper_Nesting(l)
    Cluster(l)
  }
  Transfer_Solution
  Transfer_Data_Structure
  Build_Connectivity(l_1)
End Procedure
```

3 Method of Parallelization

It should now be clear that the AMR algorithm cannot be distilled down to a small kernel upon which one can simply try out various parallelizing strategies. And so it might appear that the algorithm is an unsuitable candidate for parallelization. Fortunately, however, the algorithm exhibits a natural coarse grain parallelism in that individual mesh patches may be processed largely in isolation from one another. We have concentrated our efforts on exploiting this coarse grain parallelism using a message passing paradigm. Given this strategy, one might assume that the resultant parallel implementation would not scale well for large numbers of processors, but before reaching such a conclusion the following two points should be considered carefully.

Firstly, by far the bulk of the computational effort for the AMR algorithm is spent not on complex tasks such as the adaption process but on the simple task, logically speaking, of integrating an isolated mesh patch. For the explicit integration methods that are currently in vogue for simulating shock wave phenomena, this computationally intensive task exhibits much fine grain parallelism that could reasonably be expected to utilize additional processors. In view of this, the ideal parallel computing engine for the AMR algorithm would probably be a hybrid MIMD/SIMD machine. The orchestration of the algorithm would be performed using our message passing paradigm on MIMD processors, and the low level "number crunching" would be performed locally on shared memory, SIMD processors which executed code produced by a "smart" compiler. Interestingly, such hybrid machines are already beginning to appear in the market place, an example being the CM-5 which is produced by Thinking Machines Corporation[16].

Secondly, it should be appreciated that the AMR algorithm is designed for performing very detailed numerical simulations; thus it is safe to assume that there are always a large number of mesh patches to be processed. For example, the not overly large problem shown in Figure 8 contains 318 patches. All in all, we feel that for practical purposes our scheme

will not run into scaling difficulties, especially since economic strictures will, more often than not, severely limit the numbers of processors that are available.

All the basic components of the AMR algorithm stem from its choice of grid system. In turn, most of the implementation details for the algorithm stem from the way in which the grid system is coded. Therefore at the outset of designing the parallel implementation, we decided that it was necessary to try and preserve, as far as was possible, the grid description employed by the original serial version so as to be able to re-cycle large portions of the existing code. It later transpired that given a layer of "parallel machinery" we could actually re-use all the old serial code in a new SPMD (Single Program Multiple Data) parallel code. We now describe this machinery in some detail. First we give some of the implementation details for the parallel grid structure and its associated data storage mechanism, since these fixed the ease with which the parallelization was accomplished. Next we give an abstract description for the message passing paradigm which underpins our parallel implementation. Finally, we describe how the new parallel, message passing AMR algorithm is organized.

3.1 Parallel Grid Structure

In essence, the serial code maintained a unique identifier for each mesh patch; thus the grid structure could be described using a set of tables which were simply indexed using the mesh identifiers. For the parallel implementation we have adopted the same approach. Firstly, since we restrict ourselves to a coarse grain parallelism, we can postulate that a mesh patch need only ever reside on a single processor. After all, as was already the case with the serial implementation, a large patch can always be split up into two or more smaller patches. Given the number of mesh patches on each processing node, it is possible to construct a set of unique identifiers for the mesh patches in a grid structure distributed across one or more processors. Thus it is possible to maintain a reversible mapping between some global mesh identifier and the information pair consisting of: the identity of the node which owns the patch referenced by the global identifier and the local index by which this node references the patch;

$$\textit{global mesh identifier} \Longleftrightarrow \textit{(processor id, local mesh identifier)}.$$

Secondly, given the small amount of information that is required to define the grid structure, it is not unreasonable to store a local copy of the above mapping, together with the global grid tables, on each processing node. For our implementation this storage overhead amounts to just 52 bytes per mesh patch; although small, this overhead could be reduced to about 20 bytes, since for convenience we store certain derived variables. Now a larged sized simulation might consist of 1024 mesh patches with an average sized patch of 50×50 cells. If this problem were spread across 128 nodes, the local copy of the grid structure would amount to less than 6% of the memory required to store the flow solution contained on a specific node

(a typical user-supplied flow solver might store between 8 and 10 double precision variables for each mesh cell). Obviously having either a smaller average grid size or a larger number of processors increases this storage overhead, but the strategy of keeping a local copy of the global grid structure remains attractive. However, if at some future date the overhead ever became prohibitively large, it would be possible to make use of the information contained in the mesh extents so as to partition each of the global grid tables into some form of distributed description table.

Since the grid data structure for the parallel implementation is effectively the same as that for the old serial implementation, all of the original coding may be reused. For example, the code fragment for accessing each cell of the grid structure, given on page 6, now has the form:

```
DO L=0,LMAX
   DO K=1,NGA(L)
      GRD = GP(L)+K
      if_node_owns_grid(GRD)
         CALL UNPACK_GRID(GRD,1,1,IJ,Ibmp,Jbmp)
         DO I=1,IMX(GRD)
            IJo = IJ
            DO J=1,JMX(GRD)

               IJ contains a pointer to the data stored
               for the ij^th cell of G_{l,k}.

               IJ = IJ +Jbmp
            END DO
            IJ = IJo+Ibmp
         END DO
      end_node_if
   END DO
END DO
```

The directives *if_node_owns_grid* and *end_node_if* have simply been inserted into the original code so as to ensure that an individual node only processes those mesh patches for which it owns the flow solution. Note that the variables LMAX, NGA, GP, IMX and JMX form part of the global grid tables and are therefore stored locally on each node. It should also be noted that we only maintain one code for both the serial and the parallel versions of the AMR algorithm. For the serial code, the above directives are simply mapped to blank lines during a pre-processing phase of the compilation, while for the parallel code they are substituted by a FORTRAN "IF ...THEN" construct which tests to see if the mesh patch GRD is located on the node executing the code.

Another reason as to why the above code fragment may be so easily parallelized lies buried within the subroutine UNPACK_GRID. As was described in Section 2.1, all flow variables are stored using a number of heaps, one heap for each variable, with the data associated with the ij^{th} cell of the grid, GRD, being stored at the location

$$\text{H2PTR(GRD)} + (\text{I} + 2) + (\text{J} + 1) * (\text{IMX(GRD)} + 4).$$

In the parallel implementation this location is now given by

$$\text{H2PTR}(\textit{lindex}(\text{GRD})) + (\text{I} + 2) + (\text{J} + 1) * (\text{IMX(GRD)} + 4),$$

where the macro *lindex* returns the local mesh index corresponding to the global mesh index, GRD. Thus each node maintains its own heap data storage in exactly the same way as the serial code, but it only stores data for those patches for which it is deemed responsible.

Lastly, as before, we can recover the old serial code from the new parallel implementation by simply making the appropriate substitutions for the macro *lindex* at compile time.

3.2 A Message Passing Paradigm

Our parallel implementation for the basic AMR algorithm is based upon a message passing paradigm. At various junctures during run time, the access of data is identified as being either local or non-local. For any access which is non-local, that is the required data lies off-processor, the appropriate inter node communication tasks are first scheduled and then later executed as a series of sends and receives. In this section we describe how these inter node communications are orchestrated. Our message passing machinery relies only on a limited functionality being available at the system level; the high level procedures described here are supplemented by a small user supplied library (typically less than two hundred lines of code) which is dependent on the target platform. Thus we have been able to ensure the portability of the AMR algorithm. To date, we have ported the algorithm to a dedicated parallel computer (a 32 node iPSC/860 machine) using its native message passing routines [9], and to the following workstation cluster environments: PVM (Parallel Virtual Machine) [4], and APPL (Application Portable Parallel Library) [12].

Given our assumption that a mesh patch resides on just one processor, there are only a few key tasks within the AMR algorithm that might necessitate accessing non-local data. The most visible of these tasks is the job of priming the dummy cells which surround each mesh patch. In the original serial implementation, whenever the grid structure changed, the inter mesh connectivity was recomputed so as to build a schedule of the individual data accesses required for priming the dummy cells of the grid. Since we store a local copy of the grid description on each node, this procedure remains the same as that for the serial implementation. Basically, the connectivity information is found by comparing the mesh

extents between patches on consecutive grid levels. Where appropriate, *external* boundary information takes precedence over *fine-fine* information which in turn takes precedence over *fine-coarse* information. Once this data access schedule has been produced it may be parsed so as to find which of its entries involve non-local data. These entries are collected and stored in a separate schedule. The following pseudo-code illustrates this procedure; it builds the connectivity information for grid levels l_1 to l_{max}.

```
Procedure Build_Connectivity(l_1)
  for l = l_1 to l_max{
     Reset_Bdy_Types(l)
     Flag_Coarse_Bdy(l)
     Flag_Fine_Bdy(l)
     Set_External_Bdy(l)
     Check_Nesting(l)
  }
  Build_Off_Processor_BC_Schedule(l_1)
End Procedure
```

The innards of the five procedures within the "for" loop are too involved to be described here, but they are identical to those in the serial implementation except for the addition of some *if_node_owns_grid* directives which ensure that a processor only works on those patches which it owns; we remind the reader that full details of the AMR algorithm have been given elsewhere[13].

The procedure **Build_Off_Processor_BC_Schedule** oversees the production of the schedule of tasks which involve accessing non-local data. The resulting schedule simply consists of a set of requests for information; each request identifies a processing node together with a portion of a mesh owned by that processor which contains the desired data.

```
Procedure Build_Off_Processor_BC_Schedule(l_1)
  for l = l_1 to l_max {
     Reset_Requests(l)
     for k = 1 to nG_l {
        if_node_owns_grid(Gp_l + k)
           Build_Requests(Gp_l + k)
        end_node_if
     }
  }
  for l = l_1 to l_max {
     Transmit_Requests(l)
  }
End Procedure
```

A global synchronization is performed in the procedure **Transmit_Requests**, following which, each separate request is sent to the specific processing node that will eventually supply the off-processor data. When all the requests have been transmitted, each processing node has two local lists. For each grid level, l, the first list contains the details for nS_l items of data that the node must send out when the dummy cells for G_l are being primed, while the second list contains the details for the nR_l items of data that the node is expecting to receive during the priming process. Given these send and receive schedules, the procedure which oversees the priming of the dummy cells for the grid, G_l, has the form:

```
Procedure Set_BC(l, Nt)
  for k = 1 to nG_l {
    if_node_owns_grid(Gp_l + k)
      Set_On_Processor_BC(Nt, Gp_l + k)
    end_node_if
  }
    Set_Off_Processor_BC(l, Nt)
End Procedure
```

The call to **Set_On_Processor_BC** performs all the data movements that involve just local accesses, therefore this routine is the same as that for the serial implementation, while the procedure **Set_Off_Processor_BC** performs all the movements that involve non-local data. Note that the integration, sub-cycling index, N_t, is required so that the correct interpolation in time can be done at a *fine-coarse* boundary, see Section 2.2.

```
Procedure Set_Off_Processor_BC(l, Nt)
  Synchronize_Nodes
  for item = 1 to nS_l {
    Pack_MsgBuf(item, Nt)
    Node = Get_Node(item)
    Snd_BC_Msg(Node)
  }
  for item = 1 to nR_l {
    Wait_For_BC_Msg
    Rcv_BC_Msg
    Unpack_MsgBuf(Nt)
  }
End Procedure
```

The non-local data movements take place as follows. First, all the processing nodes are synchronized, then each node works through its list of messages to send. Each separate message is packed into a buffer and then sent to the appropriate node using some low-level, system dependent routine. Once a node has sent out all its messages, it is ready to receive

any incoming messages that it might be expecting. The order in which the messages arrive is unimportant. A node knows that it will be sent nR_l messages, so it simply waits for that number of messages to arrive. Having received a message, again via some low-level, system dependent routine, it simply decodes an identifier from within the message so as to determine where the incoming, off-processor data should be stored. Note that a similar process of sends and receives may be used in the other tasks (principally the procedures **Project_Solution** and **Transfer_Solution**) that might want to access non-local data.

3.3 The Parallel AMR Algorithm

Apart from a couple of additions made to the adaption process, the basic organization of the parallel AMR algorithm remains the same as that for the serial implementation. We now describe these additions, but first we simply present the new version of the procedure **Adapt**, c.f. the version given on page 11.

```
Procedure Adapt(l_1)
  Initialize_G̃
  for l = l_max − 1 down to l_1 {
    Set_Refine_Flags(l)
    Ensure_Proper_Nesting(l)
    Cluster(l)
    Exchange_New_Extents(l)
  }
  Distribute_Grids(l_1)
  Transfer_Solution
  Transfer_Data_Structure
  Build_Connectivity(l_1)
End Procedure
```

As has been described previously, the clustering process produces a set of mesh extents which describe the newly adapted grid $\tilde{G}_l$. However, for the parallel implementation, the call **Cluster**(l) will only produce a subset of the mesh extents for $\tilde{G}_l$, since a node only processes those patches which it owns. Therefore following the call to **Cluster**, it is necessary to perform a global exchange, amongst the processors, of the mesh extents found locally. This is done with the call to **Exchange_New_Extents**. This global exchange allows each node to assemble the global grid description table for the newly adapted grid, $\tilde{G}_l$.

Once each processor has the global description for the grid, sufficient information is available to determine how the new mesh patches should be distributed so as to achieve a satisfactory load-balance amongst the different processing nodes; this is done via the call **Distribute_Grids**. As yet, we do not have any universal, distribution strategy that will work well in all cases. Instead, we have simply chosen one of several heuristic strategies

depending upon the particular flow problem that we are solving. The question of, how should one distribute the mesh patches, remains an active area of research. However, our algorithm is structured such that different distribution strategies may be swapped in and out at will. Therefore, it matters little at this stage that we have no universal answer.

4 Numerical Results and Discussion

We now present two sets of results which demonstrate the efficacy of our parallel implementation of the AMR algorithm. The first set of results concentrates on computer science issues, such as how well does the algorithm scale, and the second set shows how the algorithm can be used as a tool to provide insight into basic fluid phenomena.

4.1 Performance Issues

In order to determine the performance of a parallel algorithm, it is common practice to carry out at least one of the following two studies. For varying numbers of processors, one monitors the time taken to solve, either a fixed-sized problem, or a problem that is scaled in such a way that the workload per processor remains constant. For reasons which will follow, neither study is entirely satisfactory for determining the performance of our parallel AMR algorithm. Nevertheless, we have carried out both studies, the results of which we present below. But first, we make the following observations.

Assuming a perfect load-balance amongst processors, effectively, it is only the time spent on inter node communications that impacts on the performance of our algorithm. On the scale of things, the overhead for looping over every mesh patch in the grid structure as is done by the code fragment on page 13, rather than looping over just those patches that a node owns, is negligible. Consequently, for a given number of processors, the efficiency of the algorithm will be strongly related to the ratio of the amount of computation to the amount of communication. Therein lies the first difficulty for assessing the performance of our algorithm. The AMR algorithm is a general purpose scheme which is not tied to any one flow solver. Thus for a fixed grid structure, that is a fixed amount of communication, the amount of computation can vary tremendously depending upon the application. Even for a fixed application, one might have a choice of several different numerical schemes to perform the "number crunching". For example, for the application shown in the next section, depending on the circumstances, we have in the past used schemes that are 4–5 times more expensive than the one used for this paper. Despite such uncertainties, of one thing we can be sure, using an expensive scheme will flatter any performance figures that are measured. For that reason, all the performance figures given in this section are for the flow solver used in the next section. Since this solver is one of the least expensive we use, the performance figures given below may be considered as conservative estimates.

Returning to our two performance studies, it is well known that a study in which the flow problem is scaled, so as to keep the computational load per processor constant, can hide poor performance caused by either a large serial component in the algorithm under test, or a large system overhead[10]. Of the two studies, however, this one more accurately reflects the way in which our algorithm is used in practice. Again, it is worth emphasizing that the AMR algorithm is designed for performing very detailed simulations. Indeed, to this end, the serial implementation has been quite successful for investigating inert shock wave phenomena; a typical simulation might take a day to run on a high-end workstation and require around 96 Mbytes of storage. Therein lies a second difficulty in assessing the performance of our algorithm. The only dedicated parallel machine that is available to us has a paltry 8 Mbytes of memory per node, of which maybe only half is available for storage once the operating system and load module have had their share. Consequently it is impossible to run a fixed-sized test problem of any consequence on such a machine. After all, a problem that would fit into 4 Mbytes might take only 30 minutes to run on a workstation. What is the merit in solving this in under 30 seconds on a massively parallel machine, if the results are going to take a day to analyze!

When investigating fluid phenomena our *modus operandi*, in common with many of our colleagues, is to scale the problem size such that, given the available computing resources, the results are delivered within some acceptable time limit. Since we utilize colleagues' machines during off-hours for our workstation clusters, typically our problems are scaled to match one of two time slots, either the 12 hours available during a week night, or the 40 odd hours available over a weekend; the simulation shown in Figure 8 was performed in one such weekend time slot.

Despite our misgivings, we have performed the following fixed-sized performance study. Thirty two horizontal stripes, which are stacked one above the other, are used to discretize a duct along which a detonation wave is propagated. Each stripe consists of 50 by 4 coarse cells and contains one level of adaption, with a refinement ratio of 4, so as to improve the resolution of the detonation wave. Tests were run on a 32 node iPSC/860 hypercube machine, and two workstation clusters: one of 16 SUN ELCs, and one of 8 SUN SPARC10s; both clusters used an ethernet ring. For the workstation tests we utilized the APPL[12] message passing library. If there were fewer nodes than stripes, the stripes were distributed in blocks. For example, four neighbouring stripes would be distributed on each node of an 8 node cluster.

For each of the parallel environments tested, Figure 5 shows a plot of the measured efficiency against the numbers of nodes used to run the fixed-sized problem. Here the efficiency is given by the ratio of the wall clock time taken to execute the problem on a single node, to the wall clock time taken to execute the problem on n processors scaled n times. Note that for the workstation clusters there is a sharp drop in efficiency across

the test range, while for the hypercube the efficiency only drops below 94% for 32 nodes. Since the identical, load-balanced problem was executed in each case, in all probability, the low communications bandwidth of the ethernet ring became saturated whereas the scalable, high bandwidth network used by the hypercube did not. To test this hypothesis, a simple test program was run which attempted to pass a large number of varying sized messages around a ring of workstations, as quickly as was possible. This test represents a crude approximation to the communication process during the priming of the dummy cells. The results from this test are shown in Figure 6, they indicate that the bandwidth of our ethernet ring is just 1 Mbyte/sec. The two vertical lines show the range of typical message sizes used during the priming process, from which it can be seen that the ethernet ring would indeed become saturated. Note that it takes just two machines to saturate the network! Now we have observed a ten-fold increase in bandwidth when switching from ethernet to fibre optic FDDI. This increase in bandwidth would significantly delay the point at which network saturation takes place. Unfortunately, our workstation clusters have not yet been upgraded to take advantage of this improved technology.

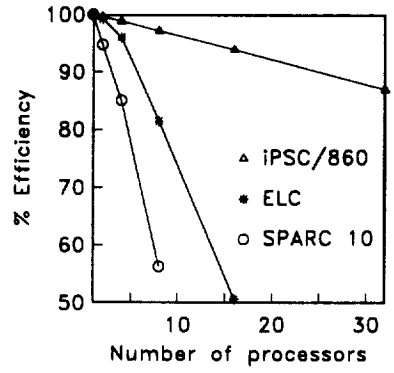
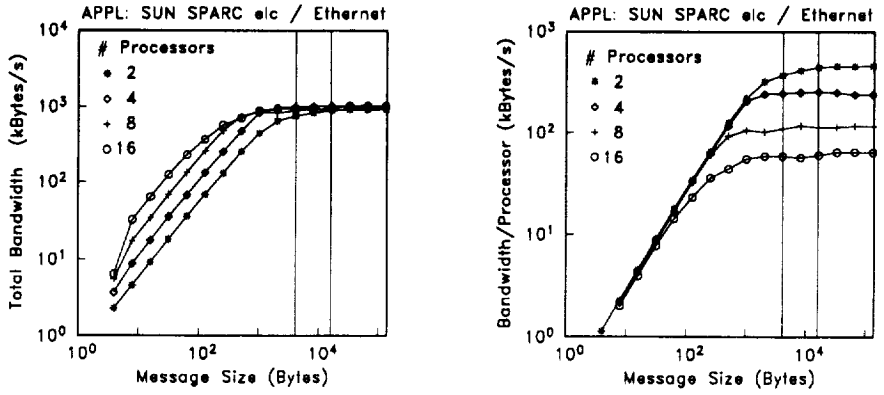


Figure 5: Measured efficiencies for the fixed-sized test problem.



(a) Total Bandwidth (b) Bandwidth per Processor

Figure 6: Ethernet bandwidth as a function of both message size and cluster size.

As an aside, it should be noted that every effort was made to find the best set of compile options for a given machine so as not to artificially improve the performance figures by needlessly increasing the computation time relative to the communication time. For example, on a SUN SPARC10 our selection of compile switches gives a CPU saving of 26% compared to a naïve optimization using just the "-O3" compilation switch.

Given the available bandwidth, the poor performance of the workstation clusters merely suggests that the ratio of computation to communication was too small for the fixed-sized problem. But this ratio was set artificially small just so the problem would fit on a single node of the hypercube. A more realistic ratio can be obtained by running a scaled problem were each node processes at most one stripe; the more nodes, the more stripes that are used for the test. We have carried out a performance study for our algorithm on such a scaled problem using a stripe which consisted of one coarse mesh of 100 by 8 cells that contained two levels of refinement, each with a refinement ratio of 4. Figure 7 shows the results of this study. Here the efficiency is defined as the ratio of the serial wall clock time to the parallel wall clock time. Once again, the algorithm performs extremely well on the hypercube. Note that the efficiency is still above 97% for 32 nodes. This time, however, the drop-off in efficiency for the workstation clusters has reached acceptable levels. For example, either 16 ELCs or 8 SPARC 10s may be used at over 80% efficiency. Again we would like to emphasize that this scaled problem is more representative of how we use the AMR algorithm than is the fixed-sized problem. Therefore, as will be shown in the next section, the above efficiencies are readily obtainable on real problems.

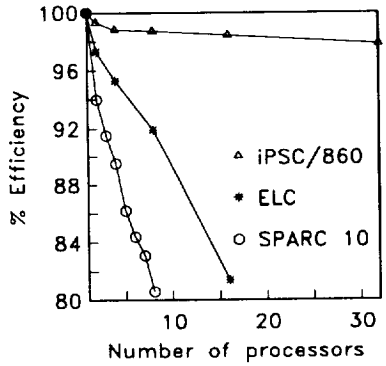


Figure 7: Measured efficiencies for the scaled test problem.

In summary, the above results show that the AMR algorithm scales extremely well on a dedicated parallel machine that offers a scalable data network. But more importantly, perhaps, the algorithm can deliver a good performance on workstation clusters, even when communications are via ethernet. With the tenfold increase in bandwidth observed with

fibre optic connections, it would not be unreasonable to expect to be able to use more than 20 machines at well over 80% efficiency, provided that the problem was reasonably well load-balanced. Given that there is a trend towards vendors offering multi-processor workstations, the effective number of nodes that could be used efficiently would reach three figures. Further advances in network technology will increase this figure still further. Certainly, for the sorts of application for which the AMR algorithm is intended, there is little danger in the foreseeable future of it performing poorly simply because of a surfeit of processing nodes.

Before moving on to the next section it is worth mentioning one practical advantage of our strategy of attacking the coarse grain parallelism within the AMR algorithm. Given the complexity of the algorithm, one can never be sure that a specific implementation is entirely free of bugs. However, through several years of use, it is clear that our serial implementation has reached a stage whereby it might be considered safe to assume that any remaining bugs are benign. Since our parallel implementation performs the same set of operations on the same set of data as the serial version, there is no reason why it should not give the same results, even down to round-off errors. After some vigorous debugging we can claim that this is indeed the case. If we had tackled a finer grain parallelism from the outset, we probably would not have been able to utilize such a stringent quality control test as checking for *zero* differences in round-off. Admittedly, one expects a good algorithm to be insensitive to round-off errors, and therefore we might well have been content to settle for having just very small discrepancies between the results produced by two nominally equivalent implementations of our algorithm. But, based upon previous experiences, there would always remain a nagging doubt that some subtle bug had been left uncovered. We are now well placed to extend our work by tackling the fine grain parallelism within a mesh patch. Given the relative simplicity, logically speaking, of the tasks performed within a patch, there would be no merit in trying to maintain the same stringent control over round-off errors for this future work.

4.2 A "Real World" Application

Although classically modelled as a quasi one-dimensional structure consisting of an inert shock front followed by a reaction zone, a detonation wave, in marked contrast to an ordinary shock wave, can be highly two-dimensional in nature[7]; a detonation front rarely remains planar. Over the years, experiments have revealed that a detonation front often supports a complex system of transverse waves which trace out a "fish-scale" pattern, thus giving rise to a characteristic length (the height of one scale) known as the detonation cell size. Recently, modern numerical schemes have been utilized in an attempt to improve our general understanding of this phenomena[5, 14]. Although such numerical simulations have been reasonably successful, it is clear that they have a number of shortcomings. For example,

whereas most calculations have been for fronts which are just one or two cells in length, a realistically sized problem would necessitate using a front that contained tens of cells. This shortcoming is an indication of the expense of such simulations. Today, however, using our parallel AMR algorithm running on a relatively small number of workstations, we are able to perform routine calculations that involve ten or more detonation cells.

Figure 8 shows results from one such simulation, a detonation wave reflecting from a 20° ramp. The left-hand plate shows a single Schlieren-type snapshot which clearly shows the expected Mach reflection flow pattern, while the right-hand plate shows the characteristic "fish-scale" pattern traced out by the transverse waves. This last plate is essentially a smear image of the vorticity field as the detonation front interacts with the ramp, and it is therefore comparable to the soot trace records that are produced experimentally[7]; the details for how we produce numerical soot traces are given elsewhere[15]. Note that the lighter a point within the image, the higher the peak vorticity that has passed that point in space, and so the edges of the "fish-scales" correspond to the tracks traced out by the numerous triple points along the length of the detonation front. Also note that the detonation cell size behind the Mach stem is considerably smaller than that behind the incident front. Again this is in accordance with experimental observation. Since the Mach stem moves normal to the ramp surface, it must be travelling faster than the incident front; thus the ratio of its speed to the Chapman-Jouguet velocity (the minimum sustainable detonation speed) is also higher than that of the incident front. Generally speaking, the higher this ratio, the more stable a detonation front becomes and so its cell size decreases.

Given the performance results that were presented in the previous section, it would serve no purpose here to present further detailed figures for the ramp simulation. Suffice it to say, such simulations have been run for up to 40 consecutive hours on eight SUN SPARC10 workstations, during which sustained efficiencies of over 80% were achieved, despite the fact that all communications were via ethernet rather than fibre optic links. While we do note a few specific details as to how the ramp calculation was performed, it should be recognized that this calculation forms part of a careful investigation into the Mach reflection process of detonation waves, and therefore a full account will appear in the literature in due course.

The ramp calculation employed a four level adaptive grid. The coarsest grid contained 400 by 18 cells, and a further three levels, each with a refinement ratio of 4, were used to resolve the details of the flow. Thus the finest grid level is equivalent to a uniform mesh of 25600 by 1152 cells. The so-called reactive Euler equations were used to model the flow[14]. In essence, a single reactant A is converted to a single product B by a one-step irreversible chemical reaction which is governed by Arrhenius kinetics. Four parameters dictate the basic behaviour of the detonation wave. These are: a non-dimensional activation energy, E, a non-dimensional heat release, Q, the ratio of specific heats, γ, and the degree of overdrive, f, for the detonation wave. For the case presented here, these parameters took the values

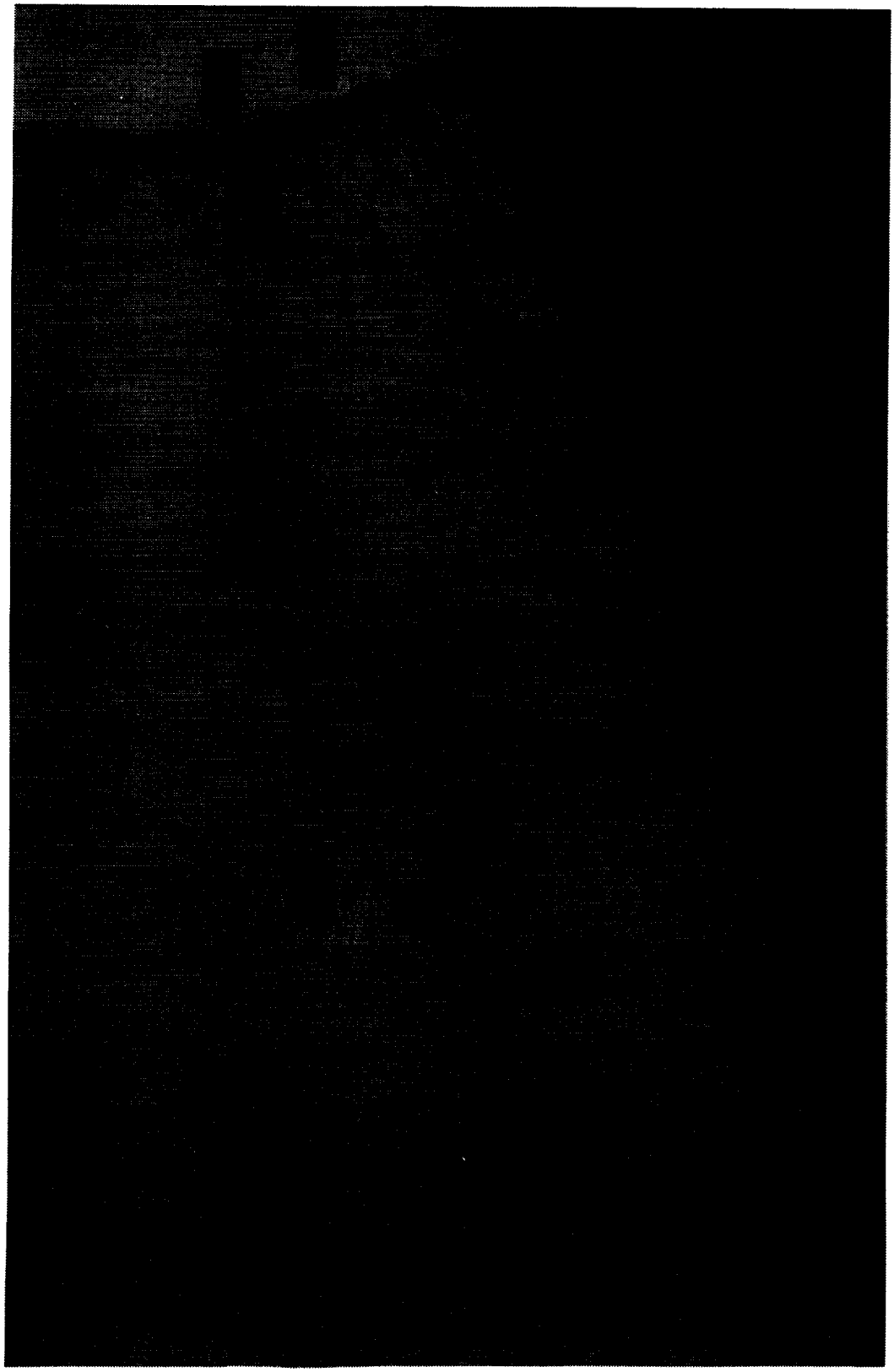
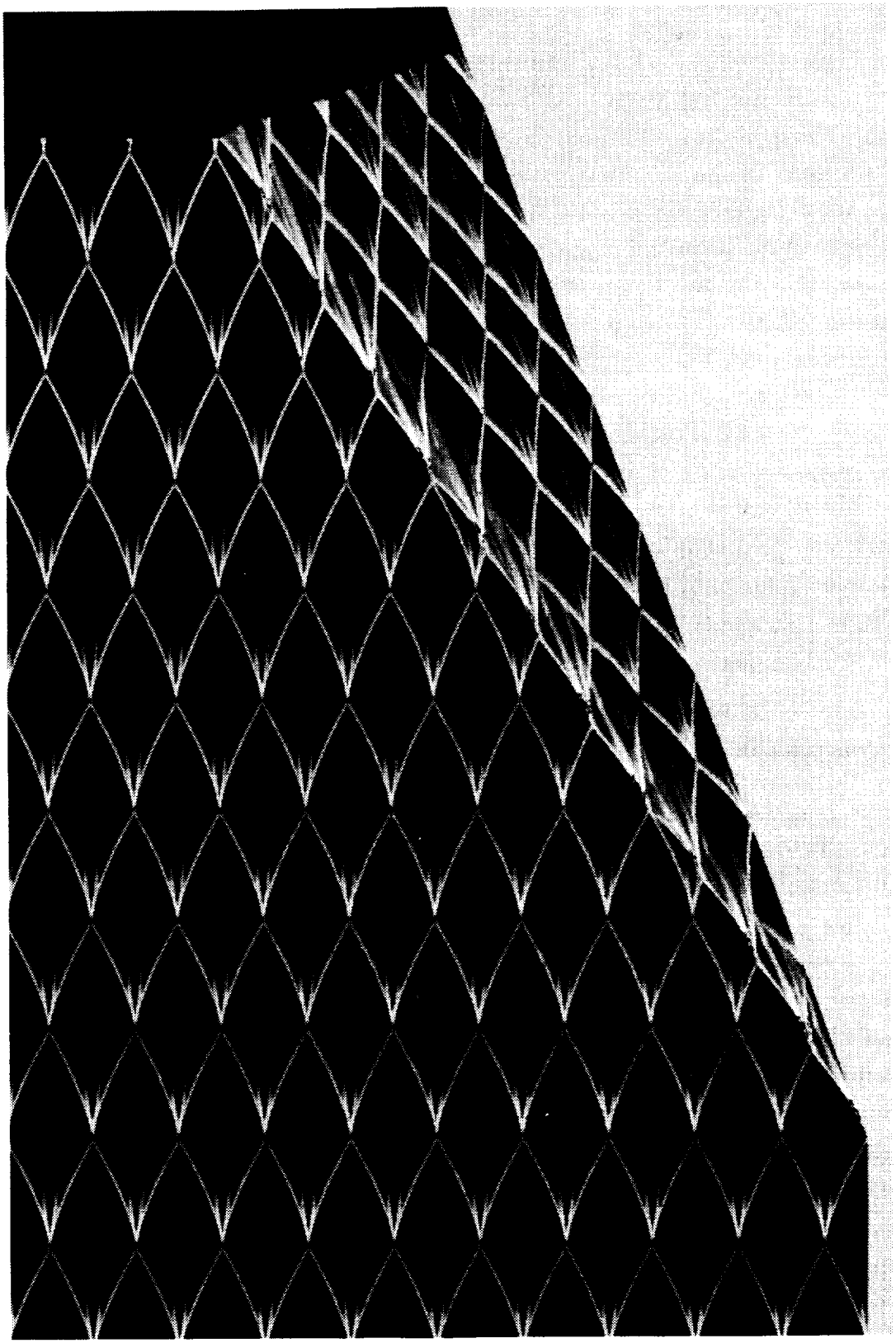


Figure 8: Schlieren-type image and numerical soot trace for a detonation wave reflecting from a 20° ramp. Key: I.S., Incident Shock; R.S., Reflected Shock; M.S., Mach Stem; T.P., Triple Point.



For caption, see page opposite.

10, 50, 1.2 and 1.2, respectively. The computation was started using the exact, steady ZND wave solution[7]. The resolution of the computational mesh may be gauged by the fact that the finest grid level provides 15 mesh cells per reaction half-length, $L_{\frac{1}{2}}$, the distance measured from the detonation front by which half the reactants are consumed in the steady ZND solution. To seed the transverse wave structure, a sinusoidal perturbation was added to the pre-exponential factor in the Arrhenius rate term for the first few time steps. The wavelength of the perturbation was chosen to be close to the transverse spacing predicted by linear theory[5]; thus to get 10 detonation cells along the length of the incident front the channel width was taken to be 76.8 reaction half-lengths. The operator split version of Roe's scheme which has been described by Clarke *et al.*[6] was used to integrate the flow, albeit with the inclusion of some additional dissipation so to avoid the failings reported by Quirk[14]. The bulk of the simulation was spent propagating the incident wave to the foot of the ramp, a distance of some 15 channel widths, so as to be sure that the detonation front was well settled before it interacted with the ramp.

We close this section with a reminder that the AMR algorithm is not tied to any one application. Admittedly the theme throughout this paper has been one of detonation flows, but, there is no reason why our AMR machinery cannot be brought to bear on an entirely different application.

5 Conclusions

A method has been presented which describes how Quirk's adaptive mesh refinement algorithm (AMR) may be parallelized. This algorithm is sufficiently general that the method of parallelization can be taken as a template for a whole class of block-structured, mesh refinement schemes. The method of parallelization exploits the natural coarse grain parallelism found in the AMR algorithm so as to leave the original serial algorithm virtually intact. Moreover, since it makes no demands on the target parallel hardware other than assuming some simple message passing capability, portability across a range of platforms is ensured. To date we have demonstrated that the parallel algorithm runs on both workstation clusters and on an Intel hypercube system.

The robustness of the parallel AMR algorithm is such that it is now run routinely on workstation clusters for large scale simulations of detonation phenomena. A typical calculation might utilize 8 SUN SPARC10s for up to forty hours. Efficiencies of over 80% are reached for these computations even when inter-workstation communication is via ethernet rather than fibre optic links. Such high efficiencies stem from the block-structured nature of the AMR algorithm; by dealing with blocks of cells rather than individual cells one markedly improves the ratio of computation to communication, thus alleviating some of the performance problems associated with using low speed networks.

Thus far we have circumvented the thorny issue of load balancing. For our detonation simulations, although the computational grid is adapting to the flow solution, it does so in a fashion which allows the load balancing to be fixed as a one-off at the start of the calculation. At this juncture, considering the large number of adaptions that take place in a typical simulation, we are not hopeful that a general purpose load balancing procedure will be found that is cheap enough to be used during the course of a calculation. Instead, we envisage employing heuristic procedures much as is done with mesh refinement monitor functions. Experience shows that such refinement functions work well in practice, so there is no reason to believe that heuristic load balancing functions won't also work well. Further work is planned to see whether or not this optimism is misplaced.

Lastly, it is worth noting that workstation clusters provide a relatively robust and cheap environment in which to develop parallel algorithms. While large purpose-built parallel machines may offer unrivaled computational power, they do not always come complete with stable operating systems! We were able to design and test the parallel AMR algorithm on a reliable system of workstations. Then, safe in the knowledge that the algorithm was working satisfactorily, it took less than one mornings work to port the code to the Intel hypercube. It would doubtless have been a more traumatic exercise to develop the parallel AMR algorithm directly on the hypercube.

Acknowledgements

We would like to thank the Internal Fluid Mechanics Division of NASA Lewis Research Center for providing us with their APPL message passing library.

References

[1] M. J. BERGER, *Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations.* Ph.D. thesis, Computer Science Dept., Stanford University (1982).

[2] M. J. BERGER AND P. COLELLA, *Local Adaptive Mesh Refinement for Shock Hydrodynamics.* J. Comput. Phys., **82**(1989), pp. 67–84.

[3] M. J. BERGER AND J. S. SALTZMAN, *Structured Adaptive Mesh Refinement on the Connection Machine.* Proceedings of the sixth SIAM conference on "Parallel Processing for Scientific Computing," Edited by R. F. Sicovec, D. E. Keyes, M. R. Leuze, L. R. Petzold and D. A. Reed. Vol II, pp. 903–906, 1993.

[4] A. BEQUELIN, J. DONGARRA, A. GEIST, R. MANCHEK AND V. SUNDERAM, *A User's Guide to PVM Parallel Virtual Machine.* Oak Ridge National Laboratory report, ORNL/TM-11826, July 1991.

[5] A. BOURLIOUX AND A. J. MAJDA, *Theoretical and Numerical Structure for Unstable Two-Dimensional Detonations.* Combustion and Flame, **90**(1992), pp. 211–229.

[6] J. F. CLARKE, S. KARNI, J. J. QUIRK, P. L. ROE, L. G. SIMMONDS AND E. F. TORO, *Numerical Computation of Two-Dimensional Unsteady Detonation Waves in High Energy Solids.* J. Comput. Phys., **106**(1993), pp. 215–233.

[7] W. FICKETT AND W. DAVIS, *Detonation.* University of California Press, Berkeley, 1979.

[8] J. FISCHER, *Selbstadaptive, lokale Netzverfeinerungen für die numerische Simulation kompressibler, reibungsbehafteter Strömungen.* Ph.D. thesis, Institut für Aerodynamik und Gasdynamik, Universität Stuttgart (1993).

[9] INTEL CORPORATION, *iPSC/2 and iPSC/860 Programmers Reference Manual.* April 1991.

[10] V.KUMAR AND A.GUPTA, *Analyzing the Scalability of Parallel Algorithms and Architectures: A Survey.* University of Minnesota, Computer Science Dept. Tech. Report TR 91-18, 1991. To appear in, Journal of Parallel and Distributed Computing.

[11] K. G. POWELL, P. L. ROE AND J. J. QUIRK, *Adaptive-mesh Algorithms for Computational Fluid Dynamics.* pp. 303–337, "Algorithmic Trends in Computational Fluid Dynamics," Edited by M. Y. Hussaini, A. Kumar and M. D. Salas. Springer-Verlag, New York, 1993.

[12] A. QUEALY, G. L. COLE AND R. A. BLECH, *Portable Programming on Parallel/Networked Computers using the Application Portable Parallel Library (APPL).* NASA TM-106238, July 1993.

[13] J. J. QUIRK, *An Adaptive Grid Algorithm for Computational Shock Hydrodynamics.* Ph.D. thesis, College of Aeronautics, Cranfield Institute of technology (1991).

[14] J. J. QUIRK, *Godunov-Type Schemes Applied to Detonation Flows.* To appear in the proceedings of the 2nd ICASE/NASA LaRC Combustion Workshop, Hampton VA, 1992. Kluwer publishing.

[15] J. J. QUIRK, *Numerical Soot Traces: "The Writing's on The Wall".* To appear in the proceedings of the Transition, Turbulence and Combustion workshop held by ICASE/NASA LaRC, Hampton VA, summer 1993 .

[16] THINKING MACHINES CORPORATION, "The Connection Machine CM-5 Technical Summary," October 1991.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1993	Contractor Report

4. TITLE AND SUBTITLE
A PARALLEL ADAPTIVE MESH REFINEMENT ALGORITHM

5. FUNDING NUMBERS
C NAS1-19480

WU 505-90-52-01

6. AUTHOR(S)
James J. Quirk
Ulf R. Hanebutte

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 93-63

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-191530
ICASE Report No. 93-63

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report

Submitted to Journal of Computational Physics

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 02, 64

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Over recent years, Adaptive Mesh Refinement (AMR) algorithms which dynamically match the local resolution of the computational grid to the numerical solution being sought have emerged as powerful tools for solving problems that contain disparate length and time scales. In particular, several workers have demonstrated the effectiveness of employing an adaptive, block-structured hierarchical grid system for simulations of complex shock wave phenomena. Unfortunately, from the parallel algorithm developer's viewpoint, this class of scheme is quite involved; these schemes cannot be distilled down to a small kernel upon which various parallelizing strategies may be tested. However, because of their block-structured nature such schemes are inherently parallel, so all is not lost. In this paper we describe the method by which Quirk's AMR algorithm has been parallelized. This method is built upon just a few simple message passing routines and so it may be implemented across a broad class of MIMD machines. Moreover, the method of parallelization is such that the original serial code is left virtually intact, and so we are left with just a single product to support. The importance of this fact should not be underestimated given the size and complexity of the original algorithm.

14. SUBJECT TERMS
adaptive mesh refinement, distributed computing, message passing paradigm

15. NUMBER OF PAGES
30

16. PRICE CODE
A03

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

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

☆U.S. GOVERNMENT PRINTING OFFICE: 1993 - 728-064/86067