PARALLEL ADAPTIVE NUMERICAL SCHEMES FOR HYPERBOLIC SYSTEMS OF CONSERVATION LAWS*

BRADLEY J. LUCIER† AND ROSS OVERBEEK‡

Abstract. We generalize the first author's adaptive numerical scheme for scalar first order conservation laws to systems of equations. The resulting numerical methods generate highly non-uniform, time-dependent grids, and hence are difficult to execute efficiently on vector computers such as the Cray or Cyber 205. In contrast, we show that these algorithms may be executed in parallel on alternate computer architectures. We describe a parallel implementation of the algorithm on the Denelcor HEP, a multiple-instruction, multiple-data (MIMD) shared memory parallel computer.

Introduction

In [22] the first author presented an adaptive numerical scheme for the numerical approximation of the scalar hyperbolic conservation law

$$\text{(C)} \qquad \begin{aligned} u_t + f(u)_x &= 0, \qquad x \in \mathbb{R}, t > 0, \\ u(x,0) &= u_0(x), \qquad x \in \mathbb{R}. \end{aligned}$$

The novelty of the method lay in the criteria used to adapt the grid and the experimental demonstration that asymptotic speedup resulted for some problems when compared with a method using the same finite difference scheme on a fixed, uniform grid. Although the adaptive method is unsuitable for execution on vector computers such as the Cray or Cyber 205 because it uses a highly non-uniform, time-dependent computation grid, the algorithm may be executed in parallel on alternate computer architectures. In this paper we extend the algorithm for scalar conservation laws to hyperbolic systems of conservation laws, and we describe a parallel implementation of the algorithm on the Denelcor HEP, a multiple-instruction, multiple-data (MIMD) shared memory parallel computer [16]. Because our implementation uses a macro-preprocessor to avoid mention of the specific hardware synchronization primitives provided on the HEP, our programs may be transported to any shared-memory multiprocessor on which the synchronization macro package has been installed.

There has been much recent interest in adaptive or moving grid methods applied to evolution equations, and specifically to conservation laws. Papers describing numerical schemes or analyses may be found in [2–5], [9], [13–14], [15–23], [25–27], [29], [33], [35–36]. Papers making specific mention of data structures for adaptive computation include [1], [4], [22], and [32].

1. Background: The Sequential Algorithm for Scalar Equations

We first describe the method for scalar conservation laws presented in [22]. While no further reference will be made in this section to that paper, the interested reader is referred there for more details about the algorithm. The numerical method is based on the following algorithm, similar to well-known algorithms for adaptive linear approximation [6–7], for choosing a grid on which to approximate a function u defined on an interval $[a, b]$.

* Submitted to the SIAM Journal on Scientific and Statistical Computing.

† Department of Mathematics, Purdue University, West Lafayette, Indiana 47907. The work of the first author was partially supported by the National Science Foundation under Grant No. DMS–8403219.

‡ Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois 60439. The work of the second author was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W–31–109–Eng–38.

ALGORITHM M: This algorithm chooses grid points at which to approximate a bounded function u defined on $[a,b]$ that is constant outside $[a,b]$. Let ϵ be a small parameter.

(1) The grid points consist only of the points a and b and the centers of admissible intervals. Admissible intervals are defined by (2) and (3) below.
(2) The interval $[a,b]$ is an admissible interval.
(3) For any admissible interval I, let $3I = \{x | \text{dist}\,(x,I) = \inf_{y\in I} |x-y| < |I|\}$. If $|I| \geq \epsilon$ and

$$|I| \int_{3I} [|u_{xx}| + |f''(u)| u_x^2]\, dx \geq \epsilon, \tag{1}$$

then the left and right halves of I are admissible intervals. The above integral is finite if u_{xx} is a finite measure; otherwise, it is to be interpreted as infinite. Note that $3I$ is an *open* interval. The approximation to u is chosen to be the piecewise linear interpolant of u at the given grid points.

This algorithm chooses a grid that satisfies three criteria that we feel are important for adaptive numerical methods that use piecewise linear functions as approximations to solutions of evolution equations:

(1) Under plausible assumptions about the structure of the solution u of (C) (that $u \in BV(\mathbb{R})$, and that between discontinuity curves $u_x \in BV(\mathbb{R})$), it may be shown that u can be approximated well in $L^1(\mathbb{R})$ by continuous piecewise linear functions on this grid.
(2) It may be shown formally that the spatial operator $f(u)_x$ may be approximated well in $L^1(\mathbb{R})$ by piecewise constant functions on this grid.
(3) One observes empirically that the error incurred in the regridding process from one time-step to the next is small on this grid; there are heuristic reasons, based on the assumed local smoothness of u away from discontinuity curves, for believing this.

The numerical scheme solves (C) on an interval $[a,b]$ instead of $\mathbb{R}$. To ensure that interactions with the boundaries of the interval $[a,b]$ may be neglected, we assume that the widths of the leftmost and rightmost minimal interval are greater than ϵ. We also assume that $f \in C^2$ and that $\|f'\|_{L^\infty(\mathbb{R})} \leq 1$; this may always be achieved by a change in the time scale. The numerical scheme is as follows:

(1) A grid $\{x_i^0\} = \mathbf{M}^0$ and initial approximation U^0 is chosen, based on the initial data $u_0(x)$ ($U^n(x_i^n) = U_i^n$).
(2) For each $n \geq 0$:
   (a) The approximate solution is advanced from time t^n to t^{n+1} at all the grid-points $x_i^n \in \mathbf{M}^n$ using, in our case, the Engquist-Osher finite difference operator ([10]):

$$\frac{\bar{U}_i^{n+1} - U_i^n}{\Delta t} + \frac{f^+(U_i^n) - f^+(U_{i-1}^n)}{h_i^n} + \frac{f^-(U_{i+1}^n) - f^-(U_i^n)}{h_{i+1}^n} = 0. \tag{F}$$

   The flux f has been separated into its increasing (f^+) and decreasing (f^-) parts, $f = f^+ + f^-$. To maintain the stability of the scheme, the time-step for the finite difference scheme (F), Δt, is chosen to be $\epsilon/4$; $h_i^n = x_i^n - x_{i-1}^n$.
   (b) The grid-selection algorithm is applied to the piecewise linear function $\bar{U}^{n+1}$ with values $\bar{U}_i^{n+1}$ at the points $x_i^n \in \mathbf{M}^n$ to yield a new grid $\mathbf{M}^{n+1}$ and approximation U^{n+1}.

The greatest difficulty in a sequential implementation of the adaptive numerical method is the organization of the data structures and the implementation of Algorithm M (which, strictly speaking, is not algorithmic at all, but definitional). Because the grid selection is based on recursive subdivision of the basic interval $[a,b]$, the natural data structure through which to organize the grid information is a binary tree. Each point in the grid, or equivalently, each admissible interval, corresponds to a node in the tree; therefore,

we will speak of nodes, grid points and intervals interchangeably. The left and right *children* of a node are the nodes corresponding to the left and right halves of the interval associated with that node; the *parent* relation is the inverse of the child relation. Two other pairs of pointers are needed to calculate the integral in (1). Links to the left and right boundary points of the interval associated with each node are stored in that node. As well, if a node is not a leaf node (if it has been divided by Step 3 in Algorithm M), pointers are stored to the adjacent intervals of the same width to the left and right of the node's corresponding interval, which we choose to call *sibling* links. (Because these links correspond to adjacent nodes at the same depth in the tree representing the grid, *cousins* may be a more appropriate term!) It may be shown that interior nodes *always* have adjacent left and right siblings; this will prove important in the parallel version of the algorithm.

After an initialization phase that calculates the grid $\mathbf{M}^0$, the calculation is organized as follows. As outlined above, the finite difference scheme calculates $\bar{U}^{n+1}$ defined on $\mathbf{M}^n$; a new grid must then be chosen based on the new solution $\bar{U}^{n+1}$. The grid selection algorithm is divided into two phases. First, a standard recursive algorithm, similar to algorithms for adaptive quadrature, calculates for each grid point $x_i^n \in \mathbf{M}$ centered in the interval $I = (x_l, x_r)$, *left int*, *right int*, and *int*, the integral (1) over the intervals (x_l, x_i^n), (x_i^n, x_r) and (x_l, x_r). This process is relatively simple—because $\bar{U}^{n+1}$ is piecewise linear, $\bar{U}_{xx}^{n+1}$ consists solely of multiples of delta measures located at each grid point x_i^n; the absolute value of each measure will be denoted by $x_i^n \to uxx$. For the same reason, $\bar{U}_x^{n+1}$ is constant between grid points, further simplifying the calculation. After these integrals are calculated, the tree associated with $\mathbf{M}^n$ is then traversed in a breadth first, top down, ordering, examining each node in turn. If a node x_i^n is a left child, it is not difficult to see that with the definitions of the various links, the integral in (1) may be calculated as

$$x_i^n \to \mathit{parent} \to \mathit{int} + x_i^n \to \mathit{parent} \to \mathit{left\ sibling} \to \mathit{right\ int} + x_i^n \to \mathit{left\ boundary} \to uxx. \tag{2}$$

(Links are represented with a right arrow.) A similar calculation applies to nodes that are right children. Note that the calculation depends on the sibling links of the parent. Step 3 of Algorithm M is applied to decide whether the node x_i^n should have children, and the tree is updated accordingly. It may be shown that at every stage in the calculation all interior nodes have adjacent left and right siblings, so that (2) may be calculated.

2. Adaptive Methods For Hyperbolic Systems

We now consider when u in (C) is a vector of unknowns in $\mathbb{R}^m$, and the equation takes the form

$$\begin{aligned} &u_t + F(u)_x = 0, \qquad x \in \mathbb{R}, t > 0, \\ &u(x,0) = u_0(x) \in \mathbb{R}^m, \qquad x \in \mathbb{R}. \end{aligned} \tag{S}$$

We assume that for all u the Jacobian matrix $A(u) = \partial F(u)$ has m real and distinct eigenvalues $\lambda_1(u) < \lambda_2(u) < \cdots < \lambda_m(u)$, with associated right eigenvectors $r_1(u), r_2(u), \ldots, r_m(u)$. Furthermore we assume that each eigenvalue is either *genuinely nonlinear* in the sense of Lax or is linearly degenerate, that is, the eigenvectors $r_k(u)$ may be normalized so that $\nabla_u \lambda_k(u) \cdot r_k(u)$ is either identically 1 (which is a convexity condition) or is identically 0 (see [17]). Again we assume a time scaling such that $|\lambda_i(u)| \le 1$ for all i and all u that will arise in the computation.

Several finite difference schemes (see [12] [28] [31]) have been devised for such problems that satisfy certain auxiliary conditions, such as having Riemann invariants. In our algorithm we chose a generalization of the Engquist-Osher scheme, introduced by Osher and Solomon in [30], which is based on methods described in [17] for solving the Riemann problem for (S). The Riemann problem has the special initial data,

$$u_0(x) = \begin{cases} u_l, & \text{for } x \le 0, \\ u_r, & \text{for } x > 0. \end{cases}$$

Because the structure of Osher and Solomon's finite difference operator determines the grid selection algorithm (as we believe it should), their scheme is described below.

Near each point $u \in \mathbb{R}^m$ one may find a local covering $\sigma(\,\cdot\,, u) : [-S, S]^m \to \mathbb{R}^m$, with $\sigma(0, u) = u$, by solving successively the differential equations:

$$\frac{d\Gamma_k}{d\tau} = r_k(\Gamma_k), \qquad \text{for } \tau \text{ between 0 and } s_k, k = m, \ldots, 1,$$

with $\Gamma_m(0) = u$ and $\Gamma_k(0) = \Gamma_{k+1}(s_{k+1})$. Set $\sigma(s, u) = \Gamma_1(s_1)$. (The paths $\Gamma_k(\tau)$ are related to the solution of the Riemann problem with $u_l = u$ and $u_r = \sigma(s, u)$.) Because the matrix whose columns consist of the eigenvectors $r_k(u)$ is nonsingular, for small enough S the mapping σ exists, is one-to-one, and covers a local neighborhood of u. In other words, any state v near u may be connected to u by a unique path $\{\Gamma_k\}$. For many physical problems, and for the Euler equations for gas dynamics in particular, it may be shown that any two physically possible states may be connected by such a path. This property is necessary for the finite difference scheme of Osher and Solomon to make sense, and we restrict our attention to systems (S) that have this property.

Note that for the conservation law (C), the spatial difference operator $(f(U_i^n) - f(U_{i-1}^n))/h_i^n$ may be written as:

$$\begin{aligned} \frac{f(U_i^n) - f(U_{i-1}^n)}{h_i^n} &= \frac{f^+(U_i^n) - f^+(U_{i-1}^n)}{h_i^n} + \frac{f^-(U_i^n) - f^-(U_{i-1}^n)}{h_i^n} \\ &= \frac{1}{h_i^n} \int_{U_{i-1}^n}^{U_i^n} (f'(u) \vee 0)\, du + \frac{1}{h_i^n} \int_{U_{i-1}^n}^{U_i^n} (f'(u) \wedge 0)\, du, \end{aligned} \tag{3}$$

where $a \vee b = \max(a, b)$ and $a \wedge b = \min(a, b)$. The scheme (F) may be thought of as distributing the first term on the right of (3) to the right endpoint of the interval (x_{i-1}^n, x_i^n) and the second term to the left. The basic idea of the Osher-Solomon finite difference method for systems is to apply the difference scheme (F) along each curve Γ_k^i connecting U_{i-1}^n to U_i^n; i.e., $U_i^n = \sigma(s, U_{i-1}^n)$. For the system (S), the wave speeds $\lambda_k(u)$ correspond to $f'(u)$ and the paths $\{\Gamma_k^i\}$ replace the simple path of integration, so that

$$\frac{F(U_i^n) - F(U_{i-1}^n)}{h_i^n} = \frac{1}{h_i^n} \sum_{k=1}^m \int_{\Gamma_k^i} (\lambda_k(u) \vee 0) r_k(u)\, du + \frac{1}{h_i^n} \sum_{k=1}^m \int_{\Gamma_k^i} (\lambda_k(u) \wedge 0) r_k(u)\, du. \tag{4}$$

The finite difference scheme for systems, then, consists of distributing the first sum to the right endpoint of the interval (x_{i-1}^n, x_i^n) and the second sum to the left. Osher and Solomon showed that these integrals may be calculated simply and effectively for many problems of interest.

One may instantly devise an adaptive method for systems, based on Algorithm M, once one finds a suitable substitute for the condition (1). The condition we choose, although heuristic, reduces to (1) in the special case of a scalar conservation law, and may be shown to be effective when solving constant coefficient, linear problems.

We first note that when the flux f of (C) is convex or linear and u is a piecewise linear function defined on a grid $\{x_i\}$, we may write

$$I_1(i) = \int_{x_{i-1}}^{x_i} |f''(u)| u_x^2\, dx = |f'(u(x_i)) - f'(u(x_{i-1}))| \frac{|u(x_i) - u(x_{i-1})|}{h_i}; \tag{5}$$

similarly,

$$I_2(i) = \int_{x_i - \delta}^{x_i + \delta} |u_{xx}|\, dx = \left| \frac{u(x_{i+1}) - u(x_i)}{h_{i+1}} - \frac{u(x_i) - u(x_{i-1})}{h_i} \right|, \tag{6}$$

in the limit of small δ. We generalize (5) and (6) for systems (with the assumption of genuinely nonlinear or linearly degenerate eigenvalues) by noting that the eigenvalues $\lambda_k(u)$ play the part of the derivative $f'(u)$, and so for systems we set

$$I_1(i) = \sum_{k=0}^{m-1} |\lambda_k(\Gamma_k^i(0)) - \lambda_k(\Gamma_{k+1}^i(0))| \frac{\|\Gamma_k^i(0) - \Gamma_{k+1}^i(0)\|_1}{h_i}, \tag{7}$$

and

$$I_2(i) = \| \frac{u_{i+1} - u_i}{h_{i+1}} - \frac{u_i - u_{i-1}}{h_i} \|_1, \tag{8}$$

where $\|v\|_1 = \sum_{k=1}^m |v_k|$ for $v \in \mathbb{R}^m$. When $m = 1$ definitions (5) (6) and (7) (8) coincide. The adaptive scheme for systems uses Algorithm M to choose the grid, substituting (7) and (8) for (5) and (6) whenever the latter two quantities arise in the scalar computation of (1).

A complete analysis of the adaptive scheme using these substitutes for the integral (1) may be carried out for the constant coefficient, linear case, that is, for

$$\begin{aligned} u_t + Au_x &= 0, \qquad x \in \mathbb{R}, t > 0, \\ u(x,0) &= u_0(x) \in \mathbb{R}^m, \qquad x \in \mathbb{R}, \end{aligned} \tag{9}$$

where A is a constant matrix. We may write $A = R\Lambda R^{-1}$, where Λ is the diagonal matrix with nontrivial entries λ_k, and R is the matrix whose columns consist of the right eigenvectors of A. The finite difference scheme that we use may now be written as

$$\frac{\bar{U}_i^{n+1} - U_i^n}{\Delta t} + \frac{A^+(U_i^n - U_{i-1}^n)}{h_i^n} + \frac{A^-(U_{i+1}^n - U_i^n)}{h_{i+1}^n} = 0, \tag{10}$$

where $A^+ = R\Lambda^+ R^{-1}$ and Λ^+ is the diagonal matrix with diagonal entries $\max(\lambda_k, 0)$; A^- is defined similarly. To analyze the system, we introduce the change of dependent variables $u = Rv$; by this device, the system (9) is transformed to the decoupled system

$$\begin{aligned} v_t + \Lambda v_x &= 0, \qquad x \in \mathbb{R}, t > 0, \\ v(x,0) &= R^{-1}u_0(x); \end{aligned}$$

the finite difference scheme (4) is equivalent to

$$\frac{\bar{V}_i^{n+1} - V_i^n}{\Delta t} + \frac{\Lambda^+(V_i^n - V_{i-1}^n)}{h_i^n} + \frac{\Lambda^-(V_{i+1}^n - V_i^n)}{h_{i+1}^n} = 0.$$

Because the components of this system are uncoupled, and the system is linear (so that $I_1(i) = 0$), the analysis presented in Theorem 5.2 of [22] may be applied separately to each component of v to prove the following theorem, which we state here without proof.

THEOREM. *Let each component of u_0 have a first derivative that is of bounded variation, and let $u(x,t)$ be the solution of (9). Assume, moreover, that Algorithm M is modified so that if an interval is smaller than $\epsilon^{1/2}$, then it may no longer be divided by Step 3; let $\Delta t = \epsilon^{1/2}/4$. If $n\Delta t = T$, and U^n is the solution of the adaptive grid algorithm above, then*

$$\|u(T) - U^n\|_{(L^1(\mathbb{R}))^m} \le 3(T+1)m\Delta t \|R\|_1 \|R^{-1}\|_1 \|u_0'\|_{(BV(\mathbb{R}))^m}.$$

$\|R\|_1$ is the operator norm of R when applied to $\mathbb{R}^m$ with the $\|\cdot\|_1$ norm.

The computer implementation for systems of equations is organized somewhat differently than the implementation for a scalar equation. Each time-step begins with $\bar{U}^n$ defined on $\mathbf{M}^{n-1}$. The first pass through the tree calculates (1) for every interval I in $\mathbf{M}^{n-1}$; in doing so it calculates $\Gamma_k^i(s_k^i)$ and $\Gamma_k^{i+1}(s_k^{i+1})$ for each leaf x_i^{n-1} in $\mathbf{M}^{n-1}$. For systems such as the Euler equations, calculating the intersection of the curves Γ_k takes much longer than calculating (7) and (8), so that the supplementary calculations to choose the mesh are very cheap. Next, the tree is traversed in a breadth first ordering, as before, to add and remove nodes to obtain $\mathbf{M}^n$. Whenever one discovers a leaf x_i^n in $\mathbf{M}^n$, then one knows that the left and right neighbors of x_i^n cannot be changed by further modifications to the tree, and one can calculate the contributions of the sums in (4) on (x_{i-1}^n, x_i^n) and (x_i^n, x_{i+1}^n) to $\bar{U}_{i-1}^{n+1}$, $\bar{U}_i^{n+1}$, and $\bar{U}_{i+1}^{n+1}$. Thus, the tree modification phase and the finite difference phase of the computation are fused into the second pass over the tree.

3. Parallel Implementation

Several changes had to be made to the original data structure and computational sequence found in [22] to implement an efficient parallel version of the adaptive algorithm. We will begin our discussion by describing in more detail the recursive calculation of the integral (1) over each interval (x_l, x_i), (x_i, x_r), and $(x_l, x_r) = I$ associated with a node x_i in the grid.

The basic algorithm is very simple:

```
Calculate Integrals (I) {
    do calculations common to all nodes x_i
    if (I is a leaf) {
        calculate left int, right int, and uxx from basic formulae
        set int := left int + right int + uxx
    } else {
        Calculate Integrals (I→left child)
        Calculate Integrals (I→right child)
        set left int := I→left child→int
        set right int := I→right child→int
        calculate uxx from basic formula
        set int := left int + right int + uxx
    }
}
```

By maintaining a stack of nodes waiting to be processed and introducing a counter in each node that keeps track of the number of children of that node that have been processed, the recursion may be removed as follows:

```
initialize the stack
push the root of the tree onto the stack
while (the stack is not empty) {
    pop I from the stack
    while (I is not a leaf) {
        do calculations common to all intervals
        set I→count := 0
        push I→right child onto the stack
        set I := I→left child
    }
    /* at this point I is a leaf */
    do calculations common to all intervals
    calculate left int, right int, and uxx from basic formulae
    set int := left int + right int + uxx
    set done := false
    while (I is not the root node and not done) {
        set I := I→parent
        if (I→count = 0) {
            /* only one child of I has been processed */
            set I→count := 1
            set done := true
        } else {
            /* both children of I have been processed */
            set left int := I→left child→int
            set right int := I→right child→int
            calculate uxx from basic formula
            set int := left int + right int + uxx
        }
    }
}
```

To implement this algorithm in parallel we follow the strategy of Lusk and Overbeek [24] (and, obviously, Brinch Hansen [8] and others), and set up a *monitor* to control access to the stack by a fixed number of processes executing identical copies of the above code in parallel. A monitor is a shared data structure that can be accessed by only one process at a time, together with routines that access and modify that structure. In our case, the stack of nodes waiting to be processed has associated with it two routines, *push a node onto the stack*, and *request a node from the stack*. When a process requests a node from the stack, either (a) it is successful, or (b) all processes are waiting to get a node from the stack and the stack is empty; in the latter case, the *request* monitor routine returns a special code to all processes that indicates that the integral calculation stage of execution is finished because there are no other processes executing that can add a node to the stack. When the special termination code is sent out, one process, the *organizer*, returns to set up the next phase of the computation, while the *worker* processes wait until the stack is non-empty and there is more work to be done. The monitor routines are implemented as macros that are expanded into in-line code by a macro preprocessor, so that little overhead is expended in the synchronization code. It is true, however, that the code in the monitor is executed sequentially, so whenever one routine is pushing a node onto the stack, for example, no other process may enter the monitor.

Because processes may not access the global (shared) stack in parallel, it is important to minimize contention among processes using the stack. This can be accomplished by having each process complete the

computation for a subtree if this subtree is smaller than a maximum size. This requires one to keep track of the size of the subtree headed by each node in the tree; this information is stored in a field that we call *subtree size*. The integral calculation algorithm may be further abstracted as follows:

```
while (the global stack is not empty) {
    pop I from the global stack
    while (I→subtree size > max size) {
        do preliminary calculations
        push I→right child onto the global stack
        set I := I→left child
    }
    /* at this point I heads a subtree with max size or fewer nodes */
    process the subtree headed by I
}
```

This strategy works well when the tree is balanced, that is, when the two subtrees headed by the children of a node are roughly of the same size. In this case, each process computes results for about *max size* to *(max size) / 2* nodes. However, the trees that are generated by our grid algorithm are very unbalanced, because the tree is deep and narrow near discontinuities in the solution. This causes some processes to have little work to do between trips to the monitor, thereby decreasing the amount of inherent parallelism. Thus, a dynamic tree partitioning algorithm that breaks up the tree into parts of roughly the same size is necessary to ensure that all processes have about the same workload between trips to the monitor. We developed the scheme presented in Figure 1, which is similar to a bottom-up scheme noted in [11]. In the tree-modification phase of the computation, complete subtrees may be removed by the algorithm, thereby making a bottom-up scheme impractical—it is useless to examine nodes that should have already been removed by a previous part of the calculation. This algorithm ensures that each process works on between $(max\ size) + 1$ and $3(max\ size)$, inclusive, nodes before returning to the monitor. It also ensures that every node on the global stack heads a subtree of size at least $(max\ size) + 1$. The partitioning algorithm employs a local stack on which to hold unprocessed subtrees. In practice, this code is executed by each worker process imbedded in a loop that repeats endlessly, with the worker processes waiting between timesteps when there are no nodes on the shared stack. At the end of the program a special signal is sent to all worker processes that allows them to terminate gracefully.

```
Tree Partition Algorithm {
    Let stack size denote the number of nodes in the
        subtrees stored temporarily on the local stack
    pop I from global stack
    set stack size := 0
    while (stack size ≤ max size and stack size + I→tree size > 3 (max size)) {
        process I as an interior node
        let min tree be the smaller of the subtrees of the two children of I
        let max tree be the larger of the subtrees of the two children of I
        if (min tree→tree size + stack size > 3 (max size)) {
            push min tree onto the global stack
        } else {
            push min tree onto the local stack
            set stack size := stack size + min tree→tree size
        }
        set I := max tree
    }
    if (I→tree size + stack size > 3 (max size)) {
        push I onto the global stack
    } else {
        push I onto the local stack
    }
    Process all subtrees on the local stack
}
```

Figure 1. Tree partitioning algorithm.

There are two conflicting requirements when considering the value of the parameter *max size*. *Max size* should be small enough so that the tree is partitioned into many pieces and all processes end work at about the same time; it should be large enough so that little time is spent by processes waiting to enter the monitor. One might find that only for larger trees can these conflicting requirements be met—there may just not be enough inherent parallelism available in the computation of a small tree.

The tree partitioning algorithm could not be applied directly to the second phase of the algorithm, in which we modify the tree and advance the solution using the finite difference operator. As noted in the discussion of the sequential algorithm, the decision whether to add or remove subtrees from a node depends on the left and right sibling links of that node's parent. Thus a breadth first traversal of the tree is strongly suggested. This means that the maximum parallelism available at one stage of the computation is restricted to the number of nodes at a given depth in the tree. Near shocks the tree is very narrow (cf. Figure 3 of [22]), and very little parallelism is available—our first implementation used a breadth first traversal of the tree and could achieve a speedup of at most a factor of 1.5 independently of the number of processes invoked in the computation. Clearly another strategy is in order.

It is at this point that we use the fact that the regridding process may be interpreted as a tree transformation process, where the tree with which we are starting, $\mathbf{M}^n$, has been developed by the same process at the previous time step. It is known that every node in $\mathbf{M}^n$ has a parent with adjacent left and right siblings, so that every node in $\mathbf{M}^n$ may be examined in parallel (subject to the condition that the tree be traversed top-down, so that nodes that will be removed during this time-step *are* removed before their children are examined). Thus, if we allow ourselves to remove subtrees at will, but to add only a layer of nodes one node thick to the edge of the tree, the complete tree associated with $\mathbf{M}^n$ may be executed in parallel. This requires that we postpone the addition of sibling links to new nodes until the tree has been processed completely.

One may show that children may be added to leaf nodes in parallel with the processing of the rest of the tree if the sibling links are not added immediately. The tree partitioning algorithm in Figure 1 may now be used to balance the execution load among the processes.

After the nodes in $\mathbf{M}^n$ have been processed, it remains to add the new sibling links and to process recursively the nodes that have been added in this step. This part of the program is done sequentially, using a breadth first ordering. Because it is rare that a subtree of height two or more is added to a node in one step (it was never observed in any computer simulation), this part of the computation takes very little time.

4. Computational Results and Discussion

Our method was applied to approximate the one-dimensional Euler equations of gas dynamics. The physical quantities in this system are the density ρ, the momentum m and the specific energy e, so that

$$u = \begin{pmatrix} \rho \\ m \\ e \end{pmatrix}, \text{and } F(u) = \begin{pmatrix} m \\ \frac{m}{\rho}\left(m + (\gamma-1)\left(e - \frac{1}{2}\frac{m^2}{\rho}\right)\right) \\ \frac{m}{\rho}\left(e + (\gamma-1)\left(e - \frac{1}{2}\frac{m^2}{\rho}\right)\right) \end{pmatrix}.$$

To compare the results with experiments by Sod [34], we set $\gamma = 1.4$; the initial values of $\rho(x)$ were chosen to be 1 for $x \le 0$ and $1/8$ for $x > 0$; the momentum was initially 0 everywhere; and $e(x) = 2.5$ for $x \le 0$ and $e(x) = 0.2$ for $x > 0$. The mesh refinement algorithm was modified slightly for this problem. To make the mesh slightly less uniform, Step 3 of Algorithm M was changed for m-dimensional systems so that if $|I| \ge \epsilon$ and

$$|I| \int_{3I} [|u_{xx}| + |f''(u)|u_x^2]\, dx \ge m\epsilon,$$

(with the appropriate substitute for the integral when $m > 1$) then the left and right halves of I are admissible intervals. To avoid an artificial time scaling, the time-step Δt was set to $\epsilon/16$. Figures 2a through 2c show the numerical approximations to the density ρ, the pressure $p = (\gamma-1)\left(e - \frac{m^2}{2\rho}\right)$, and the internal energy $\left(\frac{e}{\rho} - \frac{1}{2}\frac{m^2}{\rho^2}\right)$ at time 0.15 when solved on the interval $[-1, 1]$ with $\epsilon = 1/256$. A total of 193 grid points were used to approximate the solution at the final time; the minimum grid spacing was $1/1024$, while the largest spacing was $1/8$. Figure 3 shows the node placement superimposed upon a graph of the density ρ.

The algorithm was implemented in C on the Denelcor HEP using the synchronization macro package described in [24]. The architecture of the HEP is described in [16]; for our purposes each Process Execution Module (PEM) may be seen as a computer with an execution pipeline of length eight that executes instructions from processes in a common execution queue. One instruction from the queue is started every cycle that an instruction is waiting to be executed. When an instruction is completed, the next instruction from the same process is entered into the instruction queue. Processes waiting on semaphores busy-wait in a separate queue. In a one PEM system, all main memory accesses except for accesses to asynchronous variables and indexed stores go through a local memory interface that has a queue length of eight; all other memory accesses go through the general memory switch, which has an effective queue length of 24–30 (see [15]). One HEP may have several PEMs.

The speedups resulting from the parallel algorithm are presented in Table 1 and Figure 4. Table 1 presents the clock times for the execution of the the program for the Euler equations with $\epsilon = 1/256$. Execution times were not directly measurable, and the clock times were partially vitiated by the periodic execution of a system disk management routine for a fraction of a second. Figure 4 presents the speed-up versus the number of processes for the same problem.

TABLE 1
EXECUTION TIMES

Number of Processes	Time (in seconds) for first pass	Time (in seconds) for second pass
1	165.589419	51.099661
2	86.486981	27.329106
3	60.209457	19.657619
4	48.109058	16.012172
5	40.244957	13.783819
6	35.079278	12.341126
7	32.348853	11.445739
8	29.046026	10.726242
9	27.370288	10.190968
10	25.556640	9.809485
11	24.225887	9.481623
12	23.330546	9.264341
13	22.942822	9.215509
14	22.728492	9.123390
15	22.503358	9.183468

Because of the various queues in the HEP architecture, with their different lengths, it is difficult to estimate the maximum possible speed-up for this algorithm independent of the task synchronization time. However, some indication may be given by an examination of the assembly code for the program. Ignoring synchronization code, fewer than 13 percent of all non-synchronization instructions in the main execution path of the program are indexed store instructions. Furthermore, much of the time is spent in the library routines *pow()* and *sqrt()* (used to calculate the position of the points $\{\Gamma_k^i(s_k^i)\}$), which have mainly register to register instructions. If one assumes that the *sqrt* routine takes about 15 instructions and the *pow* routine takes about 40 instructions (conservative estimates), this further lowers the fraction of instructions that use the general memory switch to an estimated 9 percent. A crude estimate of the maximum speedup possible in a program may be obtained by taking a average of the queue lengths of the instructions executed, weighted by the number of instructions that go through each queue. In our case this would be less than $0.91 \times 8 + 0.09 \times 24$, or about 9.44. The final speedup of 7.36 achieved for the first phase of the computation, on a problem of relatively small size, is very good.

The second computational phase of each time-step did not speed up as much as the first because of the spatial dimension of the underlying physical problem. When a shock moves along the x-axis, most often only two or four nodes are added in front of the shock in one time-step. These nodes are added at the deepest part of the tree, after most of the tree has been processed. There is therefore at most a factor of two or four parallelism available during the latter part of the tree-modification and finite difference step. This effect would disappear for two dimensional problems, for which approximately $O(N^2)$ nodes would be added along a moving front at each time-step, where the average mesh size where the solution is smooth is $O(N^{-1})$. Thus, a much greater amount of parallelism would be available in the two dimensional problem.

5. Conclusions

We present a programming methodology and tree partitioning algorithm that allows us to efficiently compute the solution of a nonlinear partial differential equation with moving discontinuities in parallel on an currently available machine. It has been shown empirically in [22] that in some cases the present adaptive method achieves asymptotic speed-up over uniform fixed grid methods with the same difference scheme.

This work shows that even though the fixed grid scheme may be simple to execute on vector machines, parallel processing offers hope of executing adaptive grid methods at supercomputer speeds. The use of these techniques with better difference schemes may improve the approximation schemes for the problems by an order of magnitude. Because the amount of parallelism available depends on tree size, two dimensional problems, with larger and more complex grids, offer the promise of even more parallel efficiency.

Acknowledgments. Danny Sorensen offered much help and many suggestions during the course of this work. The computer coding was done on the Denelcor HEP at Argonne National Laboratory.

REFERENCES

[1] R. E. Bank, *The efficient implementation of local mesh refinement algorithms*, Adaptive Computational Methods for Partial Differential Equations, I. Babuska, J. Chandra, J. Flaherty, ed., SIAM, Philadelphia, 1983, pp. 74–84.

[2] J. B. Bell and G. R. Shubin, *An adaptive grid finite difference method for conservation laws*, J. Comp. Phys., 52 (1983), 569–591.

[3] M. Berger and J. Oliger, *Adaptive mesh refinement for hyperbolic partial differential equations*, J. Comp. Phys., 54 (1984), 484–512.

[4] M. Berger, *Data structures for adaptive mesh refinement*, Adaptive Computational Methods for Partial Differential Equations, I. Babuska, J. Chandra, J. Flaherty, ed., SIAM, Philadelphia, 1983, pp. 237–251.

[5] J. H. Bolstad, *An adaptive finite difference method for hyperbolic systems in one space dimension*, Lawrence Berkeley Lab. LBL-13287 (STAN-CS-82-899) (dissertation).

[6] C. de Boor, *Good approximation by splines with variable knots*, Spline functions and approximation theory, A. Meir and A. Sharma ed., ISNM v. 21, Birkhauser Verlag, 1973, pp. 57–72.

[7] ———, *Good approximation by splines with variable knots II*, Lecture Notes in Mathematics 363, Springer Verlag, 1974, pp. 12–20.

[8] P. Brinch Hansen, *Operating System Principles*, Prentice Hall, Englewood Cliffs, NJ, 1973.

[9] J. Douglas Jr. and M. F. Wheeler, *Implicit, time-dependent variable grid finite difference methods for the approximation of a linear waterflood*, Math. Comp., 40 (1983), 107–122.

[10] B. Engquist and S. Osher, *Stable and entropy satisfying approximations for transonic flow calculations*, Math. Comp., 34 (1980), 45–75.

[11] G. N. Frederickson, *Data structures for on-line updating of minimum spanning trees, with applications*, SIAM J. on Computing, 14 (1985), 781–798.

[12] A. Harten, *High resolution schemes for hyperbolic conservation laws*, J. Comp. Physics, 49 (1983), 357–393.

[13] A. Harten and J. M. Hyman, *Self-adjusting grid methods for one-dimensional hyperbolic conservation laws*, J. Comp. Phys., 50 (1983), 235–269.

[14] G. W. Hedstrom and G. H. Rodrigue, *Adaptive-grid methods for time-dependent partial differential equations*, Multigrid Methods, W. Hackbusch and U. Trottenberg, ed., Springer Verlag, 1982, pp. 474–484.

[15] H. F. Jordan, *HEP architecture, programming, and performance*, Parallel MIMD Computation: the HEP Supercomputer and its Applications, J. S. Kowalik, ed., MIT Press, Cambridge, 1985, pp. 1–41.

[16] J. S. Kowalik, ed., *Parallel MIMD Computation: the HEP Supercomputer and its Applications*, MIT Press, Cambridge, 1985.

[17] P. D. Lax, *Hyperbolic systems of conservation laws and the mathematical theory of shock waves*, SIAM Regional Conference Lectures in Applied Mathematics, Number 11, 1972.

[18] R. J. LeVeque, *A large time step generalization of Godunov's method for systems of conservations laws* (to appear).

[19] ———, *Convergence of a large time step generalization of Godunov's method for conservation laws*, Comm. Pure Appl. Math., 37 (1984), 463–478.

[20] ———, *Large time step shock-capturing techniques for scalar conservation laws*, SIAM J. Numer. Anal., 29, 1091–1109.

[21] B. J. Lucier, *A moving mesh numerical method for hyperbolic conservation laws*, Math. Comp., Jan. 1986 (to appear).

[22] ———, *A stable adaptive numerical scheme for hyperbolic conservation laws*, SIAM J. Numer. Anal., 22 (1985), 180–203.

[23] ———, *Error bounds for the methods of Glimm, Godunov, and LeVeque*, SIAM J. Numer. Anal., Dec. 1985 (to appear).

[24] E. L. Lusk and R. A. Overbeek,, *Use of monitors in FORTRAN: a tutorial on the barrier, self-scheduling DO-loop, and askfor monitors*, Parallel MIMD Computation: the HEP Supercomputer and its Applications, J. S. Kowalik, ed., MIT Press, Cambridge, 1985, pp. 367–411.

[25] K. Miller, *Alternate modes to control the nodes in the moving finite element method*, Adaptive Computational Methods for Partial Differential Equations, I. Babuska, J. Chandra, J. Flaherty, ed., SIAM, Philadelphia, 1983, pp. 165–184.

[26] ———, *Moving finite elements, parts II*, SIAM J. Numer. Anal., 18 (1981), 1019–1057.

[27] J. Oliger, *Approximate Methods for Atmospheric and Oceanographic Circulation Problems*, Lecture Notes in Physics 91, R. Glowinski and J. Lions, ed., Springer Verlag, 1979, pp. 171–184.

[28] S. Osher and S. Chakravarthy, *High resolution schemes and the entropy condition*, SIAM J. Numer. Anal., 21 (1984), 955–984.

[29] S. Osher and R. Sanders, *Numerical approximations to nonlinear conservation laws with locally varying time and space grids*, Math. Comp., 41 (1983), 321–336.
[30] S. Osher and F. Solomon, *Upwind difference schemes for hyperbolic systems of conservation laws*, Math. Comp., 38 (1982), 339–374.
[31] J. Pike, *Grid adaptive algorithms for the solution of the Euler equations on irregular grids* (to appear).
[32] W. C. Rheinboldt and C. K. Mesztenyi, *On a data structure for adaptive finite element mesh refinements*, ACM TOMS, 6 (1980), 166–187.
[33] R. Sanders, *The moving grid method for nonlinear hyperbolic conservation laws*, SIAM J. Numer. Anal., 22 (1985), 713–728.
[34] G. A. Sod, *A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws*, J. Comp. Phys., 27 (1978), 1–31.
[35] A. J. Wathen, *Mesh-independent spectrum in the moving finite element equations* (to appear).
[36] A. J. Wathen and M. J. Baines, *On the structure of the moving finite element equations*, IMA J. Numer. Anal. (to appear).