PERFORMANCE EVALUATION FOR MULTIPROCESSORS PROGRAMMED USING MONITORS

BRADLEY J. LUCIER

DEPARTMENT OF MATHEMATICS
PURDUE UNIVERSITY
WEST LAFAYETTE, IN 47907

Abstract. We present a classification of synchronization delays inherent in multiprocessor systems programmed using the monitor paradigm. This characterization is useful in relating performance of such systems to algorithmic parameters in subproblems such as domain decomposition. We apply this approach to a parallel, adaptive grid code for solving the equations of one-dimensional gas dynamics implemented on shared memory multiprocessors such as the Encore Multimax.

1. Introduction. Often the only measure of the efficiency of a parallel program is the speed-up S of the program when run on N processors. Unfortunately, this figure gives little information about where the delays occur in the program or how the program may be changed to improve its efficiency. This paper presents a profiling technique that partially remedies these problems in systems that are programmed using the monitor programming paradigm [1] [3]. In §2, we describe a profiling technique, developed by researchers at Argonne National Laboratory, that measures and classifies into simple statistics the delays associated with acquiring the hardware locks through which the various processors are synchronized. In §3, we present, for systems programmed with monitors, a natural classification, corresponding directly to algorithmic issues, of these delays. A monitor macro package, also developed at Argonne and implemented on many machines including the Intel Hypercube, the Encore Multimax, and the Cray 2, was modified so that the various monitor calls would generate the timing data necessary to apply this technique. In §4, we apply these ideas to a parallel implementation on the Encore Multimax (a shared-memory, MIMD computer) of an adaptive numerical method to solve the Euler equations of gas dynamics. It is shown that the strategy presented here can account for and classify all but a few percent of the delays incurred in the program; this strategy also tells the programmer exactly where the delays occur and what parameters at the programmer's disposal (the size of computational subdomains, for example) could be modified to make the program more efficient. In §5, we present our conclusions.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specfic permission.

© 1988 ACM 0-89791-254-3/88/0005/0022 $1.50

2. The Dritz-Boyle Performance Evaluation Framework. In [2], Ken Dritz and Jim Boyle introduce the following framework for the performance evaluation of multiprocessors. Although their paper is somewhat restricted to considering shared memory multiprocessors, the ideas can be generalized to other architectures.

To begin, Dritz and Boyle make the following assumptions:

(1) There are N identical processors in the system.
(2) One can measure the time spent waiting to acquire hardware locks.
(3) The only delays incurred in the system are the waits for locks, and hence are measurable. (There are no "invisible" delays, such as waiting because of cache contention.)
(4) The same amount of work is done with one processor as with N processors.

These assumptions are never strictly true in any multiprocessor system, of course, and we will consider later their validity for our application. Next, let

T_k: Program execution time with k processors.
W_k: Total time spent waiting for locks when run with k processors.
S_k: Speed-up over one processor when run with k processors.

Obviously, $W_1 = 0$ because all locks are immediately available. Because of the above assumptions we can calculate T_1 based solely on the measured values of T_N and W_N:

$$T_1 = NT_N - W_N; \tag{2.1}$$

so

$$S_N = \frac{T_1}{T_N} = \frac{NT_N - W_N}{T_N} = N - \frac{W_N}{T_N}. \tag{2.2}$$

These formulas assume implicitly that the CPU time used to distribute problems among and synchronize the processes can be considered useful work. Dritz and Boyle make several remarks about these formulas.

First, one does not have to make a one-processor run to calculate the speed-up of the program. This has obvious advantages when large numbers of processors will be used in a production environment.

Second, the quantity W_N/T_N, although dimensionless, may be interpreted as the number of processors lost in the computation due to synchronization delays. Dritz and Boyle recognized that waits incurred in acquiring locks in different parts of the code could be attributed to different aspects of their scheduling algorithm, so when one uses their profiling strategy, statements such as "contention for the *get new task* lock cost 1.24 processors" are meaningful.

Third, if the value of T_1 calculated in (2.1) is close to the measured value of T_1, then one may conclude that the assumptions that were made are reasonably valid for the purposes of computation. Because countervailing architectural effects may cancel each other out, (2.1) may give a closer estimate of T_1 than if each effect were present alone. Thus careful examination of the data is still required.

Based on the above assumptions, one can collect a wealth of information about how long each processor waited to obtain each synchronization lock in a program. It would be very useful to classify this information into statistics that allow the synchronization delays to be correlated with aspects of the code's underlying algorithm. For example, at the hardware level on many shared memory MIMD machines, all acquisitions of locks use the same system call, but the interpretation of each call may differ depending on a certain call's use. In the next section we present a classification of synchronization delays that is useful when programming with monitors.

3. Classification of Synchronization Delays when using Monitors. In this section we apply the Dritz-Boyle performance evaluation framework to synchronization using monitors, which, having more structure than some other synchronization primitives, can yield to a more precise analysis.

Monitors, introduced by Hoare [3] and Brinch Hansen [1], are a way to control parallel access by several processes to shared resources. (This section will talk more generally about processes rather than processors.) A monitor consists of a shared data structure or physical resource, together with a set of procedures that manipulate this structure, and several delay queues. Only one process at a time may execute code in the monitor, so critical regions are enforced. Once inside the monitor, a process may choose to be delayed in a delay queue if it is not able to operate on the shared data as it likes. A process inside the monitor may release another process from a delay queue to start executing code in the monitor; if it does so, then it must immediately exit the monitor.

When using this paradigm for process synchronization, there are only two places where delays may occur—at the entrance to a monitor, when a process must wait to enter because another process is inside the monitor, and in the monitor delay queues, where processes wait because they have no useful work to do until the status of the shared resource changes. This classification is quite useful, because delays at monitor entry and in delay queues can generally be attributed to two different algorithmic issues: delays at monitor entry are due to *critical region contention*, while delays in the delay queues are due to *lack of parallelism*. Processes contend at a monitor entry because each process executes too much code inside the monitor, or because the processes as a group return too often to the monitor without doing enough work outside the monitor. Processes wait in delay queues, on the other hand, because other processes have not yet done the work that *they* should do to prepare the shared resource for processing by the processes in the delay queue.

Parts of the above paragraph are necessarily vague, because we claim a characterization of delays in *all* uses of monitors, which is clearly impossible. However, the basic classification of delays due to monitor entry and waiting in delay queues is valid.

For many algorithms, the monitors that are used to control the distribution of subtasks are more specialized and can sustain a more detailed analysis. We refer specifically to the case when a major problem is broken up into smaller minor problems through a process called *domain decomposition*. These minor problems are then distributed by a monitor to the processes that can do the work. Often, all the processes must synchronize at the end of of each major problem—no process may proceed across the synchronization point, called a *barrier*, until all processes have reached this point. Thus, processes waiting in a delay queue are released from the queue with one of two results: they either get a minor problem to work on, or they pass the synchronization point with the other processes. Using this classification, waits in delay queues may be characterized as being due either to *task distribution* or *barrier synchronization*.

In summary, synchronization delays in algorithms pro-

grammed using monitors are due either to:

- Critical region contention, manifested through delays at monitor entry points; or
- Lack of parallelism, manifested through delays in monitor delay queues. Waits in delay queues can often be subclassified as being due to:
  - Task distribution delays, if the process receives useful work to do when it is released from the queue; or
  - Barrier synchronization, if the process waited solely for other processes to pass a synchronization point.

This classification is of use because it corresponds directly to algorithmic parameters. Consider a monitor that controls the subdivision of a large problem into smaller problems and distributes these smaller problems to various processes. If the monitor entry delays are excessive, they can be decreased by increasing the subproblem size. This will decrease the number of subproblems (and the number of times processes require the monitor) and will increase the amount of work each process does outside the monitor. If barrier synchronization times are excessive, they can be made smaller by reducing the size of subproblems. This will decrease the inequity in total process execution times. Thus, differing reasons for small speed-ups can be identified, and a choice can be made between conflicting algorithmic corrections. This is one of the most important reasons for the existence of classification schemes.

We implemented these ideas by modifying a monitor macro package, written in the C programming language by Ross Overbeek and E. L. Lusk at Argonne National Laboratory, that runs on many parallel computer systems, including the Intel Hypercube, the Encore Multimax, the Sequent, and the Cray 2 [6]. Specifically, the macros for the Encore Multimax were modified to take two extra arguments, variables that were to be incremented for monitor entry delay times (in the *menter* macro) and queue wait delay times (in the *delay* macro). All process synchronization is achieved using these two macros, and the variables used to record various delays in the higher level macros (such as *receive* and *send*) were just passed on to *menter* and *delay*. If the timing arguments are not present, or if the non-profiling monitor macro package is used, then timing data is not generated.

4. Application: A Parallel Adaptive Numerical Code. In this section the ideas in §3 are used to analyze the performance of a parallel, adaptive, computer code to calculate the solution of the one-dimensional Euler equations of gas dynamics [5]. The code was written in the programming language C on the Encore Multimax, a shared-memory MIMD computer that runs a version of the Unix operating system. For process synchronization we used a C-language monitor macro package developed at Argonne National Laboratory, modified, as described in the previous section, to generate timing statistics for each monitor call. (See [6] for a description of the FORTRAN version of the macro package.)

```
struct node {
    struct node *parent; /* parent of p */
    struct node *lsib;   /* left sibling (cousin) of p */
    struct node *rsib;   /* right sibling (cousin) of p */
    struct node *lchild; /* left child of p */
    struct node *rchild; /* right child of p */
    struct node *lneigh; /* left neighbor of p */
    struct node *rneigh; /* right neighbor of p */
    struct node *lbound; /* left boundary of p */
    struct node *rbound; /* right boundary of p */
    struct node *next;   /* next new node at same depth */
    NLOCKDEC(lock)       /* synchronization lock for p */
/* in accumulating data from children into a node, count
 * records how many children have finished.
 */
    int count;
    int depth;           /* depth of p in the tree */
/* tree_size is the size of the subtree headed by p */
    int tree_size;
/* lnorm, rnorm, norm and uxx are variables used to
 * determine if p should have children.
 */
    double lnorm;         /* used for adaptive criteria */
    double rnorm;         /* used for adaptive criteria */
    double norm;          /* used for adaptive criteria */
    double uxx;           /* used for adaptive criteria */
    struct { double rho, m, e, u, p, c;}
               st, newst, stm2, stm1, stp1, stp2;
                         /* various state variables */
    double x;            /* x coordinate of p */
    BOOL isleaf;         /* TRUE if node is a leaf */
    BOOL isleft;         /* TRUE if node is a left child */
    BOOL isdone;         /* TRUE if node has been updated
                            by one of its neighbors */
};
```

FIG. 1. *Data structure for a node in the tree.*

The Euler equations consist of a system of nonlinear, time-dependent, hyperbolic partial differential equations. These equations do not have smooth, or differentiable, solutions; discontinuities, or shocks, develop in the solutions even if the initial data are smooth. The adaptive code therefore attempts to insert more mesh points (points at which the solution is approximated) near these shocks and other singularities in the solution. As the solution progresses in time, the places where the mesh-point density is high move along with the evolving singularities. The issues related to the parallel implementation of the algorithm are as follows.

The mesh points are created through a process of recursive bisection starting with an interval $[a, b]$, so the basic data structure for organizing the set of mesh points is a threaded binary tree. Each interval that appears during any period of this recursive bisection has positioned at its midpoint a mesh point, so mesh points, tree nodes, and in-

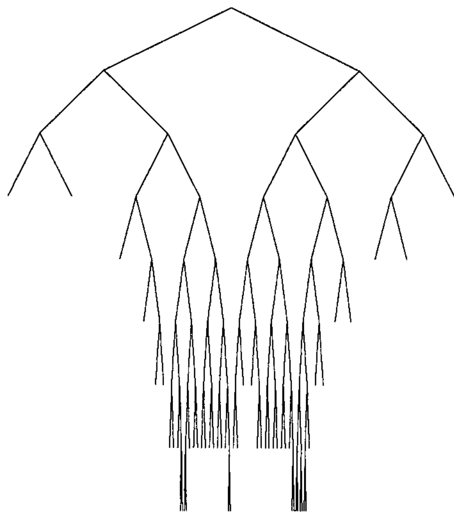

FIG. 2. *A sample tree as computational domain.*

tervals can be identified. The corresponding data structure is shown in Figure 1 (c.f., [4]), and a small sample tree is shown in Figure 2. The trees are very deep near shocks in the solution and get more unbalanced as the number of nodes increases.

Unusual terms used in Figure 1 are defined below. Assume that the node p corresponds to a point x_i at the center of an interval (x_l, x_r), and assume that the points x_i are ordered so that $x_{i-1} < x_i$ for all i. Then the terms are defined as follows.

- `lsib` The greatest point x_j with $x_j < x_i$ and the depth of x_j equal to the depth of x_i in the tree. Similarly for `rsib`.
- `lneigh` The node x_{i-1}. Similarly for `rneigh`.
- `lbound` The node x_l. Similarly for `rbound`.
- `next` When nodes are added to the tree, they are collected in an array of lists, one for each depth in the tree. `next` points to the next item in the list.
- `rho`, etc. These are physical parameters that describe the state of the gas. `rho` is the density, `m` is the momentum, `e` is the energy per unit volume, `u` is the velocity, `p` is the pressure, and `c` is the local sound speed.

The total time progression of the solution from time 0 to some time T is broken down into time steps of size Δt. Conceptually, at each time step the mesh is re-calculated based on the approximate solution at that time, and then a finite-difference approximation to the differential equations advances the approximate solution to the next time. In more detail, for each n between 0 and $T/\Delta t$, the work is divided into several passes.

- Pass 1. Approximate Riemann problems are solved for each interval (x_{i-1}, x_i) and (x_i, x_{i+1}), where x_i is a leaf in the tree. (For the present purposes, it is not necessary to know what a Riemann problem is. One must just know that for each leaf in the tree some small amount of work must be done.) Information used to decide the set of mesh points at the next time step is passed up the tree to all interior nodes.
- Pass 2a. The tree is examined in a top-down manner from the root to see which nodes (subtrees, in fact) will remain through the next time-step; unnecessary nodes are removed. For each leaf node that remains and which will not be modified with the addition of children during this time step, the values of the state variables at that leaf node and its left and right neighbors are updated using the approximate Riemann solver information calculated in Pass 1. Also during this pass, all new nodes that are immediate children of existing nodes in the tree are added to the tree.
- Pass 2b. This is a clean-up pass. Sibling links are added to the parents of nodes that were added to the tree in Pass 2a, approximate Riemann problems are solved for the new nodes, and the solution is advanced at the new nodes and their neighbors.

Passes 1 and 2a can be executed completely in parallel with minimal synchronization logic. The domain (the tree) is decomposed into small parts of varying sizes using an algorithm introduced in [5] and illustrated in Figure 3. Given a parameter *min_size*, this algorithm breaks up a binary tree (in which each node has either no or two children) into parts that contain between *min_size* + 1 and 3*min_size* nodes. It uses a counter in each node that records the size of the subtree headed by that node. Subtrees that are pruned off to be worked on by the process are stored on a local stack. This algorithm increases parallel processing efficiency because when subtasks picked up by processes are about the same size, barrier delays are decreased.

The algorithm in Figure 3 uses a global shared stack on which to put nodes that represent the entire subtrees that they head. Access to this global stack is controlled by an *askfor* monitor [6]. This monitor has two main procedures. The first can add any amount of work to the stack, and through the second a processor can request work from the stack. When a process requests work from the stack, then either work is available and is given to the process, or the process is delayed in the delay queue of the moni-

```
Tree Partition Algorithm {
    Let stack_size denote the number of nodes in the
        subtrees stored temporarily on the local stack
    pop interval (node) I from global stack
    stack_size := 0
    while (stack_size ≤ min_size and
            stack_size + I→tree size > 3 (min_size)) {
        process I as an interior node
        let min_tree head the smaller subtree of I
        let max_tree head the larger subtree of I
        if (min_tree→tree size + stack_size > 3 (min_size)) {
            push min_tree onto the global stack
        } else {
            push min_tree onto the local stack
            stack_size := stack_size + min_tree→tree size
        }
        I := max_tree
    }
    if (I→tree size + stack_size > 3 (min_size)) {
        push I onto the global stack
    } else {
        push I onto the local stack
    }
    Process all subtrees on the local stack
}
```

FIG. 3. *Tree partition algorithm.*

tor. Whenever a process, P_1 say, adds work to the stack or successfully gets work from the stack, then before leaving the monitor that process releases another process, P_2, from the delay queue, if any are waiting. (P_1 releases P_2 when it picks up work from the stack because it may not have taken *all* the work from the stack.) P_2 then checks whether more work is available. If all processes are either waiting in the delay queue or are in the monitor looking for work, then there is no more work to be done during this part of the computation and all processes are released from the monitor. (This is an implicit barrier synchronization.) One distinguished process then sets up the problem for the next pass by changing a few flags and pushing the root of the tree onto the global stack. The other processes continue to enter the monitor looking for work.

The clean-up pass of the computation is different. It is a fact that to decide whether an interval is to be subdivided according to the criteria presented in [5] then the left and right sibling links must be present for that interval's parent. Since sibling links cannot, in general, be added in parallel to parents of new nodes (the sibling nodes themselves may not have been added to the tree yet, if such was the responsibility of another process) the sibling links are added in the clean-up pass. To ensure the top-down property of sibling links alluded to above, it was decided to add sibling links to the parents of new nodes during the clean-up pass in a top-down, breadth-first ordering. As it turned out, this had a severe impact on the clean-up pass—all the code was executed sequentially by one process.

Various delays due to processor synchronization occur in each pass of the algorithm. The synchronization delays during Pass 1 of each time step occur when processors:

(1) Wait for node locks before updating information in a node's parent.

(2) Wait to enter the *askfor* monitor.

(3) Wait to leave the delay queue in the *askfor* monitor when the processor subsequently obtains work to do (task distribution delays).

(4) Wait to leave the delay queue in the *askfor* monitor when the processor obtains no work (barrier synchronization).

A graph showing the speed-up of this part of the algorithm, together with the attribution of the remaining CPU time to synchronization delays, is given in Figure 4. The program was run with from one to sixteen processors. There were 339 nodes in the tree at the final time step, and *min_size* in Figure 3 was set to two. One can see that the major cause of processor efficiency loss was the barrier synchronization delay, so that if this algorithm were applied to larger problems, with more subproblems, the speed-up would be even better. (The barrier synchronization time depends on the product of the number of time steps, the number of processors, and *min_size*, while the amount of useful work performed by all processes is proportional to the number of time steps times the tree size.)

Figure 5 presents the speed-up and synchronization delays for Passes 2a and 2b of the algorithm when applied to the same problem. The delays for Pass 2a are similar to those for Pass 1; they occur when the processors:

(1) Wait for access to a shared list of free nodes (memory management locks).

(2) Wait for node locks before updating information in a node's parent.

(3) Wait for access to a linked list of new nodes.

(4) Wait to enter the *askfor* monitor.

(5) Wait to leave the delay queue in the *askfor* monitor when the processor subsequently obtains work to do (task distribution delays).

(6) Wait to leave the delay queue in the *askfor* monitor when the processor obtains no work (barrier synchronization).

In the clean-up pass, delays occur when processors:

(1) Wait to enter the *askfor* monitor.

(2) Wait to leave the delay queue in the *askfor* monitor when the processor subsequently obtains work to do (task distribution delays). (This time is zero, because only one processor got work to do, and the work was immediately available.)

(3) Wait to leave the delay queue in the *askfor* monitor when the processor obtains no work (barrier synchronization).

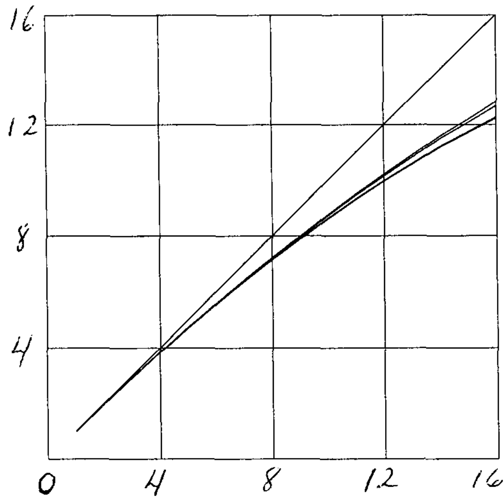

FIG. 4. *Attribution of processor use in Pass 1 of the algorithm. The horizontal axis represents the number of physical processors available, the vertical axis represents how these processors were used. The diagonal line $y = x$ represents the theoretical maximum speed-up. The lowest curve represents the actual speed-up (or effective processor use) versus the number of actual processors used. Each band between the line $y = x$ and the speed-up curve represents the number of processors lost to various synchronization delays. The reasons for the various losses, from the top band down, are: barrier synchronization delays in the askfor monitor, task distribution delays in the askfor monitor, monitor entry contention in the askfor monitor, and various node lock delays.*

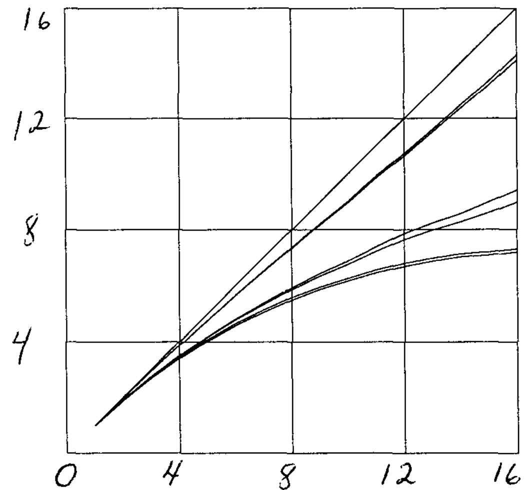

FIG. 5. *Attribution of processor use in Passes 2a and 2b of the algorithm. The horizontal axis represents the number of physical processors available, the vertical axis represents how these processors were used. The diagonal line $y = x$ represents the theoretical maximum speed-up. The lowest curve represents the actual speed-up (or effective processor use) versus the number of actual processors used. Each band between the line $y = x$ and the speed-up curve represents the number of processors lost to various synchronization delays. The reasons for the various losses, from the top band down, are: barrier synchronization delays in the askfor monitor during Pass 2b, monitor entry contention in the askfor monitor during Pass 2b, barrier synchronization delays in the askfor monitor during Pass 2a, task distribution delays in the askfor monitor during Pass 2a, monitor entry contention in the askfor monitor during Pass 2a, and various node lock delays during Pass 2a. The task distribution delays during Pass 2b, and the memory management lock contention times and the new node lock contention times during Pass 2a were insignificant.*

Here one can see that when run with sixteen processors, the serialization of the clean-up pass alone costs 1.7 processors. (The barrier synchronization time in the clean-up pass is generated when $N - 1$ processors are sitting there idle and eventually leave the monitor delay queue without picking up any work.)

The number of processors lost in Pass 2a due to the *askfor* monitor synchronization is larger than in Pass 1 because the amount of work done for each node in the tree is much less in the second pass than in the first. This implies that both the time spent in monitor entry contention and the ratio of time spent in barrier delays to clock time will be greater in the second pass than in the first.

The statistics compiled for the present analysis revealed that during the clean-up phase no process waiting in the *askfor* delay queue subsequently obtained work. Thus, this part of the code had been executed sequentially by one processor. Subsequent to this finding, a more careful algorithmic analysis showed that a breadth-first traversal of the new nodes is not necessary. It is only necessary to add nodes to the tree in "layers," with all new children of old nodes added before any grandchildren of old nodes are added, etc. Implementation of this changed algorithm might have speeded up the program by allowing more parallelism.

The effect of changing the parameter *min_size* in the tree partition algorithm of Figure 3 is exhibited in Figure 6. Here, one compares the number of processors lost in Pass 1 and Pass 2ab of the program for $N = 16$ and *min_size* equal to two and four. It is seen that reducing the size of subtrees in the tree partition from four to two makes the algorithm more efficient. One might expect this to be true because the barrier synchronization time in the *askfor* monitor is much greater than the monitor entry (critical region contention) time when *min_size* is four. A test run of a FORTRAN version of this program run on the HEP [5] showed exactly the opposite behavior. For the FORTRAN implementation of the monitors, the amount of code executed at the head of each procedure in the *askfor* monitor was so large that monitor entry contention times overwhelmed the barrier

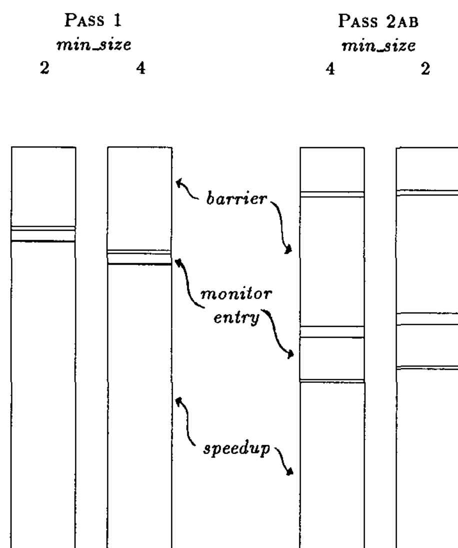

FIG. 6. *Effect of changing the size of computational subdomains on the delays in the program. Each bar shows, for a run with sixteen processors, the fraction of time spent in either useful work (speed-up), or time the processors spent waiting for busy locks. The three bars of interest are the ones that indicate the program speed-up, the time spent in monitor entry contention in the* askfor *monitor, and the time spent waiting for barrier synchronization in the* askfor *monitor. (The bands represent the same quantities as in Figures 4 and 5.) The graphs show that when the parameter* min_size *in the tree partition algorithm is four, barrier synchronization time is much larger than contention to enter the monitor, so that reducing* min_size *should allow for more speed-up. This indeed happens when* min_size *is reduced to two.*

synchronization times until *min_size* was larger.

The simple analysis in §2 was predicated on four assumptions, whose validity we will examine here for the Encore Multimax computer. First, the assumption of having N identical processors seems to have been valid for N no greater than sixteen on a twenty-processor system with no other activity on the system, as new processes were assigned to otherwise idle processors.

Second, we assumed that time spent waiting to acquire a lock could be measured accurately. Event times on the Encore Multimax can be measured accurately using its one-microsecond hardware clock, which is accessed by reading a hardware register. It took about fifteen microseconds for a processor to read the hardware clock, call the system routine to acquire a lock, and read the hardware clock again, even if the lock is immediately available. Accordingly, the time to acquire a hardware lock was consistently over-estimated by about fifteen microseconds. By counting the number of times that hardware locks were called in this application, one can conclude that this particular timing error comprised less than one percent of the total CPU time. (The actual overestimate was 38 seconds out of a total run time of 6050 seconds for one processor. This time is dwarfed by the amount of time processors actually waited to acquire busy locks in a sixteen processor run, which was over 2800 seconds, or about 175 seconds per processor.)

Third, we assumed that there were no other synchronization delays in the architecture other than waiting for locks, an assumption that is obviously false. For example, the Encore has a fast, shared, system bus. However, the amount of local computing required for each word of shared data taken off the bus was large enough that the shared bus seemed to slow down the computation by less than about 3 or 4 percent (see below). In addition to the shared bus, hardware caches are shared between pairs of processors, so that certain data brought into the cache for one processor may displace data needed by another, causing artificial cache invalidation. In this application, however, all processors are executing the same code at the same time, so cache interference seemed not to be a problem.

Finally, we assumed that the same amount of work was done for N processors as for one. The *askfor* monitor logic causes very little more code to be executed for N processors than for one, so this hypothesis was experimentally valid.

Any particular combination of real operating conditions can be evaluated by comparing the measured T_1 with the quantities $NT_N - W_N$, which would be precisely T_1 if the assumptions were valid. In our example, T_1 is 6050 seconds, while the estimate $NT_N - W_N$ monotonically increases, for N between one and sixteen, between 6012 and 6176 seconds, a difference of about two percent.[1] One can conclude therefore that the assumptions are experimentally valid for our program on the Encore Multimax.

5. Conclusions. We have introduced a classification of synchronization delays that occur in multiprocessor systems programmed using monitors. This classification is useful in that it differentiates classes of delays that are affected in known ways by changes in algorithmic parameters, such as the size of subdomains in domain decomposition. This analysis has been applied to a parallel, adaptive code on the Encore Multimax for solving the one-dimensional Euler equations of gas dynamics, where it accounted for all but a few percent of the delays that occur in the system.

Acknowledgments. This work was supported in part by NSF grant No. DMS–8403219, by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38, and by the Institute for Mathematics and its

[1] The fact that the measured value of W_1 is not identically zero decreases the estimate $NT_N - W_N$ when $N = 1$ from 6050 to 6012 seconds, while unmeasured system contention causes W_N to be more and more underestimated as N increases, causing the estimated value of T_1 to increase. It is from these figures that we estimate the shared bus contention to be less than about three or four percent with sixteen processors.

Applications with funds provided by the National Science Foundation.

REFERENCES

[1] P. Brinch Hansen, *The programming language Concurrent Pascal*, IEEE Trans. Software Eng., SE-1 (1975), pp. 199–207.

[2] K. Dritz and J. Boyle, *Beyond "speedup": Performance analysis of parallel programs*, Argonne National Laboratory Tech. Rep. ANL–87–7.

[3] C. A. R. Hoare, *Monitors: an operating system structuring concept*, Comm. of the ACM, 17 (1974), pp. 549–557.

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

[5] B. J. Lucier and R. A. Overbeek, *A parallel adaptive numerical scheme for hyperbolic systems of conservation laws*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. s203–s219.

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