2

ANL 87-7

ANL-87-7 9

Mathematics and Computer Science Division
Mathematics and Computer Science Division
Mathematics and Computer Science Division

Beyond "Speedup": Performance Analysis of Parallel Programs

by Kenneth W. Dritz and James M. Boyle

Argonne National Laboratory, Argonne, Illinois 60439
operated by The University of Chicago
for the United States Department of Energy under Contract W-31-109-Eng-38

Printed in the United States of America
Available from
National Technical Information Service
U. S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

Distribution Category:
Mathematics and Computers
(UC-32)

ANL-87-7

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

BEYOND "SPEEDUP": PERFORMANCE ANALYSIS OF PARALLEL PROGRAMS

Kenneth W. Dritz
James M. Boyle

Mathematics and Computer Science Division

February 1987

This work 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.

Contents

List of Figures

BEYOND "SPEEDUP": PERFORMANCE ANALYSIS OF PARALLEL PROGRAMS

Kenneth W. Dritz
James M. Boyle

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439-4844

Abstract. This paper addresses the problem of measuring and analyzing the performance of fine-grained parallel programs running on shared-memory multiprocessors. Such processors use locking (either directly in the application program, or indirectly in a subroutine library or the operating system) to serialize accesses to global variables. Given sufficiently high rates of locking, the chief factor preventing linear speedup (besides lack of adequate inherent parallelism in the application) is lock contention—the blocking of processes that are trying to acquire a lock currently held by another process. We show how a high-resolution, low-overhead clock may be used to measure both lock contention and lack of parallel work. Several ways of presenting the results are covered, culminating in a method for calculating, in a single multiprocessing run, both the speedup actually achieved and the speedup lost to contention for each lock and to lack of parallel work. The speedup losses are reported in the same units, "processor-equivalents," as the speedup achieved. Both are obtained without having to perform the usual one-process comparison run.

We chronicle also a variety of experiments motivated by actual results obtained with our measurement method. The insights into program performance that we gained from these experiments helped us to refine the parts of our programs concerned with communication and synchronization. Ultimately these improvements reduced lock contention to a negligible amount and yielded nearly linear speedup in applications not limited by lack of parallel work. We describe two generally applicable strategies ("code motion out of critical regions" and "critical-region fissioning") for reducing lock contention and one ("lock/variable fusion") applicable only on certain architectures.

1. Background and Motivation

How does one measure the "speed" or "efficiency" of a parallel program to determine how effectively parallelism is being employed? Usually, the technique is simply to measure the total running time of the program in a multiprocessing environment (say, with n processors) and compare that with the running time of the *same program* in a uniprocessing environment: the ratio of the two times yields a dimensionless quantity that can be interpreted as the "speedup."

The central motivation for the research we describe here was our belief that, by itself, the speedup figure for a parallel program provides little information. In particular, if the speedup is less than n, as is often the case for applications exhibiting fine-grained parallelism, one gains no insight into the reasons for the loss. Typically those reasons range from inherent lack of sufficient parallelism, about which one can do little without rethinking one's algorithms, to faults of implementation causing unnecessary contention for resources in the form of synchronization delays—contention that can often be reduced by simple refinements. Our work has yielded techniques for observing, characterizing, and measuring these various factors, and in the process we have also begun to catalog some standard techniques for reducing resource contention in fine-grained parallel programs.

We shall assume throughout this paper that a parallel program whose performance is to be measured is organized so as to divide its work in some way among the physical processors available to it. The program's work is performed by cooperating computational *processes*, which may be fewer than or more than the number of physical processors, and whose number may even vary in time. If the number of processes is less than the number of processors, the processes can be simultaneously active on different processors. Whether all of them are or not, at any given moment, depends in part on the other activity in the system. If the number of processes is more than the number of processors, there is in general no hope for them all to be active simultaneously[1], and adding more processes cannot increase the speedup. Thus, we confine our interest to the case of fewer processes than processors, or an equal number of each, and we also assume that the other activity in the system during the measurement runs is negligible (indeed, we require it). This permits us to ascribe reasons with greater confidence to observed losses of speedup.

An alternative way to compute a measure of performance akin to speedup is to calculate the average number of busy (not blocked) processes during the run. (A technique for doing so is described in Section 6.) Indeed, on the assumption that the total amount of work to be performed is fixed by the problem and not influenced by the number of processes sharing in that work, the two techniques yield comparable results. If that assumption does not hold for the parallel program in question, then the two techniques will in general yield quite different performance measures, and the result yielded by the alternative technique should probably be called "average concurrency" rather than speedup. Our applications have all been characterized by a fixed amount of work.

Our measurement techniques perturb the program somewhat, though we have been careful to keep the instrumentation overhead to a minimum. We are satisfied that it is small enough, in relation to the phenomena being measured as well as the real work performed by the program, that little would be gained by trying to distinguish the overhead from productive work. Nor do we measure and report the overhead associated with the extra logic required to parallelize a serial program; although it may be substantial, it can be considered productive work because it is necessary to the organization of the program in parallel form. These two kinds of overhead will still be present when the program is run using only one process, although there will of course be no synchronization delays produced by the parallelization logic or observed by the instrumentation logic. As long as one is interested in studying the performance of a parallel program as a function of the number of processes, the overhead for parallelization and instrumentation will be common to all runs and should not significantly affect the comparisons. On the other hand, if one is interested in ascertaining whether it is worth parallelizing a serial program in the first place, then the cost of the extra logic required to do so is clearly of interest; it may be that the running time of the parallel program using n processes exceeds the running time of the *serial* program, even though it does not exceed that of the parallel program using one process.

We assume that, on the shared-memory multiprocessors with which we are concerned, serialization of concurrent accesses to global variables by multiple processes is achieved, at least at some level, with locks and locking. The application program may use locks directly, or it may use higher level primitives provided by a subroutine library or the operating system. In the latter case, locks and locking operations on them will typically underlie the implementation of the higher level primitives. In any case, serialization is achieved by the temporary suspension of a process that is attempting to acquire a lock already held by another process. (We consider "busy waiting" to be equivalent to process suspension.) The duration of the suspension can be measured.

[1]However, the Denelcor HEP, which takes processes making memory references out of the pipeline shared by those not doing so and routes them to a memory switch, permits speedups in excess of the number of processors (pipeline slots). If there is sufficient parallelism in the problem, the slow memory references will be overlapped so well that their effects will not be felt, while at one process they will keep the pipeline from achieving even its full one-processor speed.

All of our measurement methods are based on the interpretation of accumulated times spent waiting to acquire individual locks[2]. The duration of the wait for acquisition of a lock may be arbitrarily small, so that it is important to use a high resolution clock to measure that duration. Often one has no choice but to call some run-time routine to get the time of day just before trying to acquire the lock, call it again just after acquiring the lock, and subtract the two times. One has to be concerned as well about the overhead of the calls, especially since the second one takes place *inside* the critical region. On the Denelcor HEP the clock routine we used had a resolution of 100 nanoseconds. On the Encore Multimax, we used a free-running counter with a period of one microsecond. The free-running counter is particularly attractive because reading the clock amounts only to reading a memory location. Moreover, on that system the timing facilities were integrated with the locking routines for this study, reducing the number of subroutine calls needed to perform a timed lock from three to one.

Our applications have all been obtained by using the TAMPR program transformation system to derive parallel Fortran programs from various abstract programs written in pure applicative Lisp, as described in [1]. Our ultimate motivation and goal was to create, use, and study the performance of a parallel version of the TAMPR program transformer itself. That goal has been achieved, subject to the qualification that incremental, parallel garbage collection was initially deferred and is at present being developed. Other programs, some with even more inherent parallelism than the TAMPR transformer, have been used for early experiments with our performance measurement techniques. One such program, a simple Lisp program to compute Fibonacci numbers recursively, is clearly a toy as far as practical matters are concerned; it has, however, been an extremely useful model of an inherently parallel program for our research purposes, and it has the advantage of not using CONS and thus not requiring a garbage collector.

2. Structure of the Applications

Since expression evaluations can have no side effects in pure applicative Lisp (PROG and SETQ are not used), those that are the arguments of a function invocation may be evaluated in any order, and therefore in parallel, prior to invoking the function. Thus, opportunities for parallel evaluation are explicit and easy to identify, needing only a strategy for exploitation.

The arguments of a function invocation may be elementary objects, primitive function invocations (CAR, CDR, etc.), or user-defined function invocations. Evaluation of arguments of the first two types, which we will call "trivial" arguments, involves an amount of computation that is known in advance to be very small; in fact it is too small, in relation to the communication and synchronization overhead involved, to be worth parallelizing. On the other hand, evaluation of "non-trivial" arguments—that is, user-defined function invocations—involves an amount of computation that is potentially much larger, though unknowable in advance. We *assume* that each user-defined function invocation represents an amount of work worth executing in parallel with other work, if such an opportunity exists.

But not every user-defined function invocation represents an opportunity for parallelism. Only when *two or more* of them occur as arguments in some argument list do they actually represent work that can be overlapped with other work. Thus, a direct strategy for exploiting the parallelizable work would be to look for argument lists containing two or more user-defined function invocations as arguments and arrange to create new processes to evaluate those function invocations in parallel. Of course, this strategy would still leave open the question of how to implement user-defined function invocations that do *not* create opportunities for parallelism.

The latter user-defined function invocations could be implemented by the usual methods for implementing recursion, that is, by using a stack. However, using two mechanisms raises the possibility that two implementations of each function would be required, one for recursive invocations and one for parallel ones. Moreover, this approach requires an unbounded stack for each parallel process, which would create major problems of memory management.

The fact that creation of a new parallel process requires creation of a new copy of local memory for the process, together with the fact that creation of such memory is one of the major steps in implementing

[2]Though we describe our methods in terms of timed lock attempts, it should be possible to adapt them to higher level synchronization primitives if low-level locking operations are not directly accessible. The important principle is the timing of the duration of the process suspension caused by the primitive operation. The timing of a high-level synchronization primitive may include a significant amount of time that is not actually process suspension, and it will be necessary to identify that "overhead" time and treat it properly. One method for distinguishing the overhead from the actual waiting time is discussed at the end of Section 3.

recursion, suggests that the implementation of user-defined function invocations could be simplified by creating a new process for *every* such invocation, whether it gives rise to parallelism or not.

Thus, a straightforward approach to extracting the parallelism potentially represented by user-defined function invocations can now be described. Whenever a user-defined function invocation is encountered, a process (which can now be called the "parent" process) would create a subprocess to evaluate it. Meanwhile, the parent process would continue with whatever it can do concurrently (such as evaluating neighboring trivial arguments or creating additional subprocesses to evaluate non-trivial ones). Eventually the parent process would arrive at a "synchronization point," where it must wait for the subprocesses it has created to complete and deliver their results to it before it can proceed.

The undeniable appeal of this straightforward, "unbounded number of processes" approach is the ease with which it can be proved correct. Nevertheless, two considerations make it infeasible. First of all, the number of simultaneously extant processes would be unbounded. Few systems have the capability to create and manage a large number of processes, let alone an unbounded number of them. Even when that is possible, having more of them than the number of physical processors is unlikely to be profitable, as we remarked in the previous section. Secondly, process creation is often a rather expensive operation, since it usually has to create a new swap image by copying the current one and writing out the copy. The overhead of frequent process creations is simply intolerable in fine-grained parallel programs, where the "grains" of computational work between the process creation points are very small.

We proceeded by adopting the straightforward approach as a desirable abstraction, which we then refined by using correctness-preserving transformations. The result was a much more efficient, and still correct, implementation. Even during the refinement process, however, we deferred low-level implementation decisions as long as possible. This approach permitted us to separate the work of producing a correct design from the work of producing an efficient one, with the result that neither activity was compromised for the sake of the other. Such simplification of programming is typically the result of employing abstract programming techniques.

The final design employs a fixed number of processes that remain continuously in existence for the duration of the run. We call these processes ***servers***. When a server arrives at a point where it has some work that can be done in parallel with other work, it creates a conceptual entity called a *chore*[3] to represent that parallel work and adds it to a queue of chores awaiting service by a server. The chore is physically represented by a structure called a *frame*[4]. A chore's frame contains storage for the chore's arguments and local variables as well as an assortment of control items, one of which is a number, called the ***resume point***, that identifies the computation to be performed by the chore. All the servers are identical and contain within them the instructions for performing the computations required by any chore.

The basic actions of a server, as actually implemented in our system, are described in the next few paragraphs. To avoid straying too far from our main theme of performance analysis, we have elected to omit a discussion of how we arrived at these actions by refining and optimizing the abstract model presented above. Further details may be found in [1].

A server with nothing to do tries to obtain a chore from the chore queue. (Servers are in this state initially and from time to time as they complete chores.) If the chore queue is empty, the server waits until a chore becomes available, i.e., until another server creates a new chore and queues it for service. Once a server obtains, or is bound to, a chore, it branches to the instructions for performing the chore's computations, as identified by the resume point.

While carrying out the instructions for its current chore, C, a server may be directed to create a subchore, SC, of C. To do that, it first creates and initializes a frame for SC. The server will shortly continue by putting C aside and adopting SC as the next chore to serve. Before it does so, however, it must arrange for the proper future handling of C. What is needed for C depends on whether SC was the *last* of a number of subchores that can be created before C needs their collective results to continue, or more remain to be created. If there ***are*** more subchores to create, the server updates the resume point in C's frame to identify the continuation of C and requeues that frame on the chore queue; if not, it updates the resume

[3]Some researchers use the term *task* to refer to what we are calling a chore; others use task as a synonym for process. To avoid confusion, we prefer to invent a new term.

[4]Whenever we talk about queuing a chore or obtaining a chore from the chore queue, we are of course referring to the queuing or dequeuing of frames representing chores.

point nevertheless but does not requeue C's frame. (Later, we will say what subsequently happens to C in this case.) It has now effectively unbound itself from C and rebound itself to SC, which it then begins to serve.

Assuming for the moment that there are more subchores to create, and C has been requeued, an unoccupied server, if one exists, will obtain C from the chore queue, and both SC and C will proceed in parallel. On the other hand, if all the servers are occupied, then C will reside on the chore queue for a time (during which maximal parallelism is achieved) before being acquired by a server. C will sooner or later be dequeued and resumed at its resume point, and thus its next subchore will be created. Eventually its last subchore (of the current group that can be performed in parallel) will have been created, and its next step must use their results. Finally, with its resume point identifying that next step, it is left not bound to any server and also not on the chore queue. In this state it languishes (while neither demanding service from servers nor keeping them from serving other chores) until all of its subchores have completed and delivered their results to local variables in C's frame.

Now, what changes the state of a such a chore, C, that is neither bound to a server nor on the chore queue? C's frame is pointed to by each of its subchores' frames. (That linkage was created during the initialization of those frames.) As some server arrives at the end of the instructions for one of those subchores, say SC, the server stores SC's result in C's frame, frees SC's frame, and decrements a count (kept in C's frame) of the number of subchores of C still remaining to be completed. What it does next depends on whether SC was the last of the subchores on which C was waiting, i.e., on whether the count has just been reduced to zero or not. If so, the server simply adopts C as the next chore to serve (effectively unbinding from SC and rebinding to C), thus changing C's state. If not, the server requests another chore from the chore queue, leaving the state of C to be changed in the future by the server that completes the last subchore of C.

The implementation we have just described is not the most intuitive one imaginable. For example, when a server creates a subchore, SC, of a chore, C, and SC is not the last of a group of subchores that can be performed in parallel, why does the server requeue C and take over SC, instead of queuing SC and merely continuing with C? Under conditions of full load, this alternative implementation would build and traverse the computation tree in breadth-first order. The chore queue would contain more work, at any given moment, than is required to keep the servers busy, and this would merely increase the storage requirements without achieving additional parallelism. On the other hand, our actual implementation, under the same full-load conditions, builds and traverses the computation tree in depth-first order; together with a LIFO discipline for the chore queue, it achieves a kind of *load balancing* by delaying the creation and queuing of additional chores when all the servers are occupied. Our chore queue contains fewer chores on average, with those that are present representing the *potential* for generating additional work rapidly whenever a server does become unoccupied. The depth-first behavior closely mimics recursion in one-process runs, with each link in the recursive call-stack represented either by an implicit link to a chore on the chore queue (i.e., one whose final subchore has not yet been created) or by an explicit link to a chore not on the chore queue (i.e., one whose final subchore has been created). In multiprocess runs, the behavior of *each server* also mimics recursion when all the servers are occupied—with the exception that, since parts of their respective call-stacks are interleaved on the chore queue, a server completing a non-final subchore of a chore C may resume an eligible chore other than C (and some other server may resume C).

Program initialization starts by reading n, the number of processes. Then $n-1$ server processes are started (forked) and told, by a parameter passed to them, that they are not the "main" server (which has yet to be started). On seeing that parameter, these servers all immediately request a chore from the chore queue. Since no chores have been queued yet, they all become blocked, waiting for work. The main program then executes the server subroutine itself (by calling it synchronously), passing a parameter that identifies this invocation of the server as the "main," or distinguished, one. On seeing that parameter, the server takes a different path from the others: it falls into the first chore, eventually generating subchores that will cause the other servers to receive work.

Near the end of the program, the computation tree collapses naturally until there is only one server left with work to do (the others will be once again blocked, waiting for work). When it finishes its last chore, that server (which may, but need not, be the distinguished server) creates and queues a special "suicide" chore, then terminates. One of the remaining servers gets the suicide chore from the chore queue; it performs that chore by queuing another suicide chore and then terminating itself. Thus, all the servers commit suicide in their turn. The non-distinguished servers terminate by exiting (the number of processes decreases by one as each exits), while the distinguished or "main" server terminates by returning to its caller, the main program.

The behavior of non-distinguished servers differs from that of the distinguished server in only one other respect. Should a non-distinguished server happen to get a chore requiring it to read some input, that server cannot perform that chore (each process may have its own copy of the file control block of an open file; thus one process may not know how another left the file positioned by previous reads). In this case the server requeues the chore to be picked up by another server; then it spins in a loop until, as must eventually happen, the distinguished server gets that chore and resets a shared (i.e., global) variable, breaking the loop. By then, many non-distinguished servers may have received the input chore, passed it on, and entered a loop; they are all released when the distinguished server gets the input chore. Aside from these differences, any server may perform any chore. There is no requirement, for example, that the final non-suicide chore be performed by the same server that performed the initial chore.

It is possible to recognize the instructions for all the chores, and to grasp their hierarchical relationships, just by looking at the program text. It is also possible to infer the path that will be taken through the program text when it is executed using just one process. On the other hand, it is not possible to predict, just by reading the program text, the path that any given server will take when the program is executed using multiple processes. Though the results of executing the program (i.e., the outputs it produces) are deterministic, its detailed behavior—even the global flow of control through the program—is decidedly non-deterministic and highly variable from one run to the next.

It should be apparent that the servers share certain global resources, and that to maintain those resources in a consistent state requires the servers to serialize their accesses to them. For example, the chore queue is a shared (global) resource: any server can put a frame on the chore queue, and any server can take a frame off the chore queue. Clearly, two different servers could attempt to access the chore queue simultaneously. To prevent two servers from dequeuing the same chore or otherwise destroying the integrity of the linked list structure and control information representing the chore queue, and to preserve the semantics of the original Lisp program, the second server must be prevented from accessing the chore queue until the first completes its transaction. Similarly, there is a reservoir of frames that may be allocated by any server in the act of creating a chore, or freed by any server in the act of completing a chore. (Freed frames are available for reallocation.) Thus the "frame space" is a shared (global) resource, and two servers needing to access it must serialize their accesses. By the same token, the "list space" consumed by the CONS function is a shared resource whose accesses by multiple servers must be serialized, as is another space in which atoms like print names and big numbers are allocated. Finally, the count field of each frame is a shared resource requiring serialized access, since two servers could simultaneously be completing subchores of the same parent chore, leading them both to attempt to decrement, and compare against zero, the same count field simultaneously.

The protocol that we use for serializing accesses to shared resources is that of monitors [3,5]. A *monitor* can be thought of as an abstract data type incorporating the shared resources as objects of the type and the operations defined on them as implicitly serialized operations of the type. In practice, a given monitor is composed of one *monitor procedure* for each operation, encapsulating all the necessary references to the relevant objects, with the property that no more than one process can be executing the code in any of the procedures of the monitor at any given moment. This property of mutual exclusion is typically achieved by associating a unique lock with each monitor and requiring each procedure of the monitor to successfully acquire the lock as soon as it is entered, and release it just before returning.

We have several monitors, of which three are worth considering in some detail. An example of a simple one is the monitor associated with the reservoir of frames. It has two monitor procedures, CRFRAM ("create, i.e. allocate, a frame") and FRFRAM ("free a frame"). A server calls CRFRAM to allocate a new frame (CRFRAM is a function that returns a pointer to the newly allocated frame), and FRFRAM to free a frame and make it available for reallocation. The mutual exclusion property of monitors is relied upon to guarantee that two servers will not be allocating or freeing frames simultaneously, or one allocating and another freeing. The monitor lock, which for this monitor is called FRMELK ("frame lock"), is held only for the brief duration of a CRFRAM or FRFRAM call; it is released by the same server that acquired it.

A more complicated example is the monitor associated with the chore queue. It, too, has two monitor procedures, QCHORE ("queue a chore") and GUBKCR ("get an unblocked chore"). Its monitor lock is called UBCQLK ("unblocked chore queue lock"). Like the frame monitor, the chore queue monitor provides two complementary functions. The QCHORE monitor procedure is entirely analogous to FRFRAM in that it adds an object to a global pool. However, GUBKCR differs in a significant way from CRFRAM. Whereas the exhaustion of the frame reservoir is a fatal condition if it should occur, inability of GUBKCR to deliver a chore to the requesting server is not at all fatal. Whenever it happens, it means only that there are fewer than *n* unblocked chores in existence at the moment—not enough to keep all the servers busy. That naturally happens at the beginning and end of the run. As we have already said, only one chore exists initially, and it will in general be a little while before a sufficient number of them are generated to keep all the servers busy. Similarly, near the end of the program less and less work exists to be done, until finally there is only one chore left. The amount of parallel work existing at any given time well away from the beginning or end of the program varies, of course, with the application, the number of servers, and other factors, and it may occasionally dip below *n* unblocked chores. In any case, when there is insufficient work to keep all the servers busy, the excess servers (those without work to do) must somehow be suspended until work becomes available. GUBKCR uses another feature of the monitor protocol, the so-called *delay queue*, to suspend servers that call it when, and as long as, the chore queue is empty.

The delay queue is implemented with another lock (called UBCQSW), which is manipulated in the following way. UBCQSW is initialized to the locked state (unlike the monitor locks, which are initialized to the unlocked state). GUBKCR locks the monitor lock, UBCQLK, as usual on entry. It then determines whether the chore queue is empty or not. If not, it dequeues a chore (that is, a frame representing a chore), unlocks the monitor lock, and returns a pointer to the dequeued chore. If, on the other hand, the chore queue is empty, it increases a count of the number of idle servers (UBCQCT, initially zero) by one, *releases the monitor lock*, and tries to acquire the delay queue lock. Since the delay queue lock is initially (and normally) locked, the server will become suspended. Note that no server "owns" the monitor lock at the moment, so any of the remaining servers can execute either GUBKCR or QCHORE. If another server executes GUBKCR, it will find the chore queue still empty and likewise increment UBCQCT and become suspended waiting on UBCQSW, the delay queue lock[5].

On the other hand, when a server executes QCHORE, that server will do the following. It first locks the monitor lock, of course. Then it performs its normal function of adding a chore to the chore queue. It then examines UBCQCT to determine whether any servers are waiting (in the delay queue) for work. If none are (i.e., UBCQCT is zero), it merely unlocks the monitor lock and returns to its caller. However, if the cumulative number of GUBKCR calls exceeds the cumulative number of QCHORE calls, UBCQCT will be greater than zero, indicating that one or more servers are indeed waiting, in which case QCHORE unlocks the delay queue lock and returns to its caller *without ever unlocking the monitor lock*. The effect of this is to release one of the servers that had previously become suspended in GUBKCR at the attempt to lock the delay queue lock. Thus one server will finally succeed in acquiring that lock (so that the lock almost immediately returns to its normal, or locked, state). That server proceeds by decrementing UBCQCT, the number of waiting servers, by one, dequeuing the chore recently added to the chore queue, unlocking the monitor lock (finally), and returning the dequeued chore to its caller.

It should be emphasized that the chore queue monitor lock, UBCQLK, may be unlocked by a different process from the one that locked it. Nevertheless, like all monitor locks it is held only for brief periods. That lock is not the one that provides for the indefinitely long suspension of a server while it is waiting for work; it is the delay queue lock associated with the chore queue monitor that does that.

The chore queue monitor is the only one of our monitors that uses a delay queue. Actually, it would be theoretically possible for the frame monitor to use one in a similar way, so that exhaustion of the frame reservoir need not be fatal. That is, if CRFRAM finds the frame reservoir empty, it could delay the server making the call until some other server releases a frame with FRFRAM. In marginal cases this might allow some runs to complete that would otherwise have failed. But there is no guarantee that *all* the servers will not become suspended waiting for another one (which does not exist) to free a frame. If that happens, it will be a fatal condition (and may even lead to deadlock unless sufficient care is used in the design of the

[5]While our *chore queue* is physically implemented as a queue of linked frames that we manage, the *delay queue* does not have such an embodiment as a physical queue and is not managed by us. Servers in the delay queue are those waiting to acquire UBCQSW. In our case, they are active processes spinning on a spin lock, and the absence of a physical queue and its associated structure means that there is nothing that specifies which process will get the lock when it is released. In other cases, the lock attempts might be mediated by the operating system with use of an actual queue—its, not ours. We use the term "delay queue" only because of historical precedent.

monitor). Because the advantages of a delay queue for the frame monitor are so marginal, we never gave serious thought to implementing it.

The remaining monitor that we will discuss in detail has the same simple behavior as the frame monitor ("acquire the monitor lock, perform the requested service, release the monitor lock, and get out"), but a more complicated structure. It is the monitor that serializes accesses to the count fields of frames. It could have been organized with a global monitor lock and a single monitor procedure to decrement the count[6], and while this would indeed prevent two servers from updating the *same* count field simultaneously, it would also prevent (unnecessarily) two servers from updating *different* count fields simultaneously. While a count field is a shared (global) resource, in the sense that any number of servers could be trying to access it simultaneously, each count field is in reality a *separate* shared resource.

It is appropriate to have a *separate* monitor, with a separate monitor lock, for each count field. We do, and the monitor lock protecting the count field in a particular frame resides in that frame as well. We could have placed the code for acquiring the lock, updating the count, and releasing the lock in a procedure, but that would have required the lock (as well as the count field) to be passed to the procedure as an argument. The closest we could have come to passing the lock as a parameter would have been to pass a pointer (array index) to it. Actually, certain unique properties of the Denelcor HEP, discussed in the next section, enabled us in our initial implementation to call a non-standard intrinsic function in HEP Fortran in place of calling a monitor procedure of our own design. Once we moved to other machines and had to abandon the use of the intrinsic function, instead of calling a monitor procedure to update the count (near the end of each chore) we simply executed, in-line, the few instructions (including those to acquire and release a lock) required to do so. Even though the count monitor is really a dynamically varying number of separate monitors, with the monitor procedure code statically replicated in a fixed number of places, it is useful for categorization purposes to think of it as *the* count monitor.

3. Measurement of Critical-Region Contention Times

Our programs are so structured that they can be run with any number of processes from one up to some reasonable maximum (usually the number of physical processors available). The real work to be done is essentially independent of the number of processes; all that changes as the number of processes, n, is varied is the distribution of the grains of work among the processes. In general, as n is increased we would expect to see higher levels of parallelism, with the result that the fixed amount of work would take less time to complete.

Our earliest investigation, using the abundantly parallel Fibonacci program on the Denelcor HEP, simply timed the recursive computation of nine Fibonacci numbers (the largest being *fib*(25), which is 121393) and displayed the time at the conclusion of the run[7]. The results are displayed in Figure 1. The speedup figures reported in Figure 1 were obtained in the usual way by dividing the one-process time by the shorter running time using n processes. For a sufficiently small number of processes, the running times exhibit nearly the expected inverse relationship to the number of processes, and the speedups exhibit nearly the expected linear relationship. The gain resulting from further increase in the number of processes slows and eventually halts.

Following [6], we intuitively believed that increasing contention for the locks used to serialize access to the critical regions (interior of monitor procedures) is what ultimately limited the speedup achievable in this program as more server processes were added. When contention is negligible, each server tackles chores in this simple program at a rate that is essentially independent of the number of servers; thus, as the number of servers is increased from one, the rate at which attempts are made to acquire a particular monitor lock, by one server or another, increases in proportion. As the average time between lock attempts decreases, the probability of finding a particular lock locked increases (once a server acquires a lock, the server holds it for a fixed time related to the function of the monitor procedure that was called). Thus, to

[6]One might imagine the need for a second monitor procedure to initialize the count field of a chore just before creating its first subchore. But that access of the count field can be made without taking steps to serialize it, since no other server can possibly access the same count field until at least one subchore of the chore is created and queued.

[7]The timing included the parallelizable problem-dependent computation only; i.e., it excluded the initialization of the Lisp environment, the reading of n, and the creation (forking) of the servers, all of which involve considerable I/O and proceed serially.

Running Time and Speedup vs. *n*		
n	Time (sec.)	Speedup
1	89.017	1.000
2	45.645	1.950
3	31.272	2.847
4	24.187	3.680
5	20.049	4.440
6	17.437	5.105
7	15.741	5.655
8	14.773	6.026
10	14.425	6.171
12	14.335	6.210
14	14.337	6.209

Program: Fibonacci
Special features: None
Machine: Denelcor HEP
Conditions: Path to local memory enabled

Fig. 1. Times and speedups for the Fibonacci program on the Denelcor HEP, original implementation of monitors

minimize lock contention one should strive to make critical regions as short as possible[8].

Our original monitor procedures were designed to be as simple as possible. The initial goal was to make them functionally correct; optimizations could come later. We felt ready, after our first few runs, to consider ways of shrinking our critical regions, and we were interested in seeing what effect that would have on achievable speedups.

Thus, we reimplemented our monitor procedures with efficiency in mind. One strategy we employed for shrinking them was to take code at the top or bottom of a monitor procedure that was not required to be in the critical region (because it did not reference shared data) out of the critical region. This was accomplished by moving the monitor lock operation down over any instructions that did not reference shared data, and by moving the monitor unlock operation up where possible. For example, in CRFRAM it was convenient to initialize several fields of a newly created frame before returning it to the caller, but those actions do not need to be serialized with the actions of any other server. Thus, it is sufficient to obtain a frame from the reservoir while inside the critical region, then end the critical region by releasing the monitor lock, then initialize the frame *outside* the critical region (but still inside the monitor procedure), then deliver the frame to the caller. We call this type of optimization *code motion out of critical regions.*

Another strategy for shrinking critical regions, applicable on the HEP because of its unique architecture, is what we call *lock/variable fusion.* On the HEP, any word of memory is capable of being used as a lock while simultaneously holding data. Depending on the pattern of usage of one's shared variables inside monitor procedures, it may be possible to combine in one word the function of a shared variable and the function of the monitor lock or the delay queue lock, thus allowing the use of hardware instructions that simultaneously acquire the lock and fetch the variable, or store the variable and release the lock. All of our locks were capable of being fused with appropriate shared variables, thereby eliminating the overhead of fetching and executing additional instructions for synchronization.

The possibility of lock/variable fusion was apparent to us very early, so early in fact that it influenced the original structure of the count monitor. At the end of the previous section we discussed the fact that the count field of each frame is a separate shared variable requiring its own monitor. In the "unoptimized" or naive implementation of monitors, this would necessitate a separate monitor lock for each count field. The obvious place to store the lock is in the same frame as the count field that it protects. By fusing the lock with the count field we were able to reduce the *size* of the control portion of each frame. Even better, once we performed the fusion we were able to dispense with the writing of the count monitor procedure and merely call the non-standard intrinsic function IAINC provided with HEP Fortran. The function reference

[8]This idea is not new. It permeates many early writings, at least implicitly (see, e.g., [7, 9]; also [8, pp. 262–266]).

IAINC (<*count field*>, −1) generates in-line code to lock the count field, simultaneously fetching the count; decrement it by one; store the updated count back into the count field, simultaneously unlocking it; and return the previous value of the count. These advantages prompted us to adopt what was effectively an optimized version of the count monitor from the very start. Once we progressed to other machines, where lock/variable fusion was not possible, we were forced to adopt a more conventional approach (and also accept a 20% increase in the size of the control portion of a frame), though we continued to use an in-line expansion of a small monitor procedure instead of actually calling one.

Before we can describe our third strategy for shrinking critical regions, we must fill in further details of the implementation of the reservoir of frames. Frames come in different sizes (the control portion is fixed, but the number of fields for parameters of the corresponding Lisp function invocation and the number of fields for chore-local variables depend on the chore); thus the size of the variable part of the required frame is passed as a parameter to CRFRAM. When a frame is freed by FRFRAM, it is put on a list of freed frames of its size (referred to as the *frame list* of that size). Thus, CRFRAM first tries to reuse a frame of size s by consulting the frame list of size s. If that list is non-empty, a frame is removed from the list and returned to the caller. If it is empty, on the other hand, then a new frame of size s must be allocated from a heap of space reserved for that use. The frame heap is just a contiguous area from one end of which space is consumed as needed for new frames. Nothing is ever returned to the frame heap, and its state is therefore identified by a single shared variable, which is the pointer to the next available position in the heap. Initially, of course, all the frame lists are empty. Consequently, the early calls to CRFRAM will result in frames being allocated from the frame heap. The rate of new allocations from the frame heap will trail off as a supply of freed and reusable frames builds up on the frame lists for each size.

The original, unoptimized version of the frame monitor used a single global monitor lock, FRMELK. In that version, if one server attempts (say) to get a frame of size 5 while another is already engaged in getting one (or even freeing one) of size 4, the second request will be delayed. If both requests involve only the frame lists, and not the frame heap, then, since they involve *different* frame lists, they can proceed in parallel. Only if both need to access the frame heap is there a need to serialize parts of the two requests. Thus, it is appropriate to consider each frame-list head (pointer to the first element on the associated frame list) to be a separate shared variable, the accesses to which are serialized by separate locks. Both the frame-list heads and their locks are arrays indexed by the size of the requested frame (the head for the frame list of size s is called FRMLST(s), while the lock for that list is called FRLLKS(s)). Also, the pointer to available space in the frame heap is a separate shared variable (AVAILP), which too can have its own lock (FRMELK is now used solely for that).

The optimized version of FRFRAM, which never needs to access the frame heap, simply locks the frame list corresponding to the size of frame to be freed (i.e., the appropriate element of FRLLKS), adds the frame to the list, and unlocks the list. The optimized version of CRFRAM first locks the frame list of the appropriate size, then checks to see if that list is non-empty. If it is non-empty, CRFRAM removes a frame from the list, unlocks the list, and returns the frame to its caller. If it is empty, on the other hand, it unlocks the list, then it locks the heap (i.e., locks FRMELK), allocates space from the heap (aborting if the heap is exhausted), unlocks the heap, initializes the new frame, and returns the new frame to its caller.

We call this third strategy *critical-region fissioning*. It is applicable whenever the shared resources protected by a critical region can be partitioned into a collection of subunits that can be accessed in parallel, effectively partitioning the critical region into either physically separate or logically separate subregions that can be executed in parallel.

Our second timing investigation assessed the effect of performing all three of these optimizations simultaneously. We again merely measured the total time of the run and computed the speedup as before. The results are displayed in Figure 2, which should be compared with Figure 1. It appears from the data that the optimizations were effective, presumably by reducing contention for critical regions (i.e., for their locks) in the regime where it had earlier been significant. Moreover, the saturation point (at which adding more processes produces no further speedup) was pushed farther away[9].

[9]The reader is reminded (cf. footnote 1) that speedups in excess of the number of physical processors, in this case eight, are possible with the HEP. At the same time, one should not assume that the leveling off of speedup at about 9, which is apparent in this figure and the next, signals exhaustion of the capacity of the eight processors. Some of the Encore Multimax experiments we describe in Section 4 were in fact first performed on the HEP, where they produced maximum speedups in excess of 13 (for 24 servers).

Running Time and Speedup vs. *n*		
n	Time (sec.)	Speedup
1	89.136	1.000
2	45.412	1.963
3	30.854	2.889
4	23.589	3.779
5	19.261	4.628
6	16.391	5.438
7	14.374	6.201
8	12.902	6.909
10	10.965	8.129
12	9.913	8.992
14	9.490	9.393
20	9.705	9.185

Program: Fibonacci
Special features: Optimized monitors
Machine: Denelcor HEP
Conditions: Path to local memory enabled

Fig. 2. Times and speedups for the Fibonacci program on the Denelcor HEP, after optimizing the monitors

It was at this point in our investigation of parallel programs that we first developed the desire to learn more about what they were doing than the gross speedup figure could tell us. We had just improved our software to reduce critical-region contention and observed the expected improvement in speedup. The data did not tell us by how much we reduced the contention for any particular lock. We were beginning to imagine scenarios in which improvements in one monitor procedure reduce contention for its lock so much that *other* locks become bottlenecks, by virtue of the increased rate of attempts to lock them. Under these conditions it would be hard to identify what really happens as improvements are made, and hard to understand the observed magnitude of their effects. Before conducting any further studies, we decided to explore ways to instrument the programs to gather direct information on the amount of contention for individual locks.

What we wanted to know was how much total time is spent unproductively waiting for any particular lock to be acquired. More precisely, we wanted to know much of its total execution time a typical server spends just waiting for any particular lock. This information can be gathered by timing each lock attempt and amassing those times in an appropriate way. Fortunately, a clock routine having enough resolution to be useful was available on the HEP.

Our first attempt to measure and display critical-region contention times was implemented on the HEP as follows. In each monitor procedure of interest, just before trying to acquire the monitor lock we call the 100-nanosecond clock routine (named CLOCK) and save the result in a process-local variable. As soon as possible after successfully acquiring a lock, we again call CLOCK and save the result in another process-local variable. Later, after the end of the critical region (i.e., after releasing the monitor lock, but before returning from the monitor procedure), we subtract the first time from the second and add the difference to the accumulated acquisition time for the lock. That accumulation is held in another process-local variable. Thus, each server accumulates its acquisition time for each lock of interest independently of the other servers. Each server initializes its acquisition time accumulators to zero as it starts up, and each prints the final values of its accumulators as it terminates[10].

It is vitally important to organize the instrumentation code so that it perturbs the program as little as possible. This means adding as little extra code as possible, especially inside critical regions. The second call on CLOCK, and the saving of the value returned by that call, are the only unavoidable additions to critical regions.

[10]The printing was added to the suicide chore described in Section 2 at a point before it queues another suicide chore. This was enough on the HEP to avoid output synchronization problems, though in environments where such output is sufficiently asynchronous other techniques will be necessary.

Using the technique just described, we instrumented the optimized version of the Fibonacci program to report the time each server spent trying to acquire the locks for the frame monitor and the chore queue monitor. For the former we had a single acquisition time accumulator, rather than one for each frame list, and we did not measure the relatively insignificant time spent trying to acquire the frame heap lock (recall that it is accessed far less frequently than the frame list locks once the frame lists are populated with freed frames). Thus, the contention that we would infer from the data would be the time that a typical server was delayed in getting or freeing a frame of some size because another server was busy getting or freeing a frame *of the same size* at the moment. For the chore queue monitor we merely needed a single accumulator to amass the time spent trying to acquire the chore queue monitor lock, UBCQLK.

At the end of the run we were presented with *n* lines, each displaying one server's total lock acquisition time for these two monitors. Though it was possible for the times to vary from one server to another, we observed very little variation (not more than about five percent), which is almost surely due to the randomizing effect of the division of work in our program. Figure 3 summarizes the data obtained from one

Total Per-Server Lock Acquisition Time vs. *n*

n	Time (sec.)	Speedup	Frame list locks acquisition time (sec.)	Chore queue lock acquisition time (sec.)
1	108.5	1.000	8.9	2.8
2	55.2	1.966	4.8	1.4
3	37.4	2.901	3.4	1.0
4	28.5	3.807	2.8	.78
5	23.3	4.657	2.4	.65
6	19.8	5.480	2.2	.57
7	17.4	6.236	2.1	.51
8	15.6	6.955	2.1	.47
9	14.3	7.587	2.1	.44
10	13.4	8.097	2.2	.43
11	12.8	8.477	2.4	.41
12	12.4	8.750	2.8	.39
13	12.3	8.821	3.3	.36
14	12.2	8.893	3.9	.34
15	12.2	8.893	4.4	.32
16	12.2	8.893	4.9	.30
17	12.2	8.893	5.3	.28
18	12.2	8.893	5.6	.27
19	12.2	8.893	6.0	.25
20	12.2	8.893	6.3	.24

Program: Fibonacci
Special features: Optimized monitors,
lock acquisition time instrumentation
Machine: Denelcor HEP
Conditions: Path to local memory enabled

Fig. 3. Total per-server lock acquisition times for the Fibonacci program on the Denelcor HEP, using optimized monitors

of these runs on the HEP. The lock acquisition times shown in the figure are "eyeball averages" taken over all the servers. By comparing the time and speedup columns in Figure 3 to those in Figure 2, one can see that, although the instrumentation code increased the running time, it had very little effect on the speedup[11].

The lock acquisition times we obtained were *not* the sought-after contention times. They include a certain amount of overhead associated with the attempt to acquire a lock and with the reading of the clock. That overhead results in the recording of a measurable (i.e., non-zero) interval between the pre-lock clock reading and the post-lock clock reading, even when the lock is immediately available and there is no con-

[11]The instrumentation code amounts to a CLOCK call and an assignment added both inside and outside critical regions, and an accumulation and further assignment added outside critical regions. The invariance of the speedup would be expected if these additions maintained the ratio of time spent outside of critical regions to time spent inside them.

tention. Clearly, the lock acquisition times reported for the one-process run are made up entirely of these overheads, since there is no other process to produce contention. As the number of processes is increased, but not to the point where contention becomes noticeable, the total lock acquisition time experienced by a typical server declines (in inverse proportion to the number of servers) because the same number of lock attempts, and therefore the same amount of measurable overhead, is spread over a larger number of servers. Our data show that the frame list lock acquisition times behave in this manner up through about five or six processes. As the number of processes is further increased, rising contention dominates further declines in the overhead, causing the lock acquisition times to turn back up. For the frame monitor, the data show this effect beginning at about ten processes; because there are fewer calls to the procedures of the chore queue monitor, contention for its lock never really becomes apparent in our data.

Further analysis of this data was performed after the departure of our HEP. Using the fact that 529559 CRFRAM/FRFRAM calls and 264767 QCHORE/GUBKCR calls are made during a run, one can compute the overhead to be about 16.8 microseconds per lock attempt for the former and 10.5 microseconds for the latter[12]. On the reasonable assumption that the work is indeed spread more or less evenly among the servers, one can then compute the overhead component of a typical server's total lock acquisition time (it is the assumed number of lock attempts per server times the overhead per lock attempt). Subtracting out that component leaves the total true contention time, and dividing by the assumed number of lock attempts per server produces the contention time per lock attempt. Figure 4 shows the results of

Calculated Critical-Region Contention Times vs. *n*

n	Frame list contention time		Chore queue contention time	
	Total per server (sec.)	Per lock attempt (μsec.)	Total per server (sec.)	Per lock attempt (μsec.)
1	.00	.0	.00	.0
2	.35	1.3	.01	.0
3	.43	2.4	.07	.8
4	.58	4.3	.08	1.2
5	.62	5.8	.10	1.7
6	.72	8.1	.11	2.4
7	.83	10.9	.11	2.9
8	.99	14.9	.12	3.7
9	1.11	18.8	.13	4.4
10	1.31	24.7	.15	5.7
11	1.59	33.0	.16	6.5
12	2.06	46.6	.16	7.1
13	2.62	64.2	.15	7.1
14	3.26	86.3	.14	7.4
15	3.81	107.8	.13	7.6
16	4.34	131.2	.13	7.6
17	4.78	153.3	.12	7.4
18	5.11	173.5	.12	7.8
19	5.53	198.4	.10	7.4
20	5.86	221.1	.10	7.6

Program: Fibonacci
Special features: Optimized monitors,
lock acquisition time instrumentation
Machine: Denelcor HEP
Conditions: Path to local memory enabled

Fig. 4. Calculated critical-region contention times based on the data in Figure 3

these calculations based on the data in Figure 3. While the time and speedup numbers in Figure 3 *suggest* some kind of saturation effect involving the frame list locks above about 12 processes, the evidence for that in Figure 4 is even more convincing. The total per-server chore queue contention time begins to *decline* at about that point, and the chore queue contention time per lock attempt levels off; in addition, the frame

[12]The overhead per lock attempt is not the same for the two monitors because the former involves a subscripting operation between the two CLOCK calls while the latter does not.

list[13] contention time per lock attempt, which had been rising at an accelerating rate through about 12 processes, continues to rise beyond that point at a *constant* rate of about 22 microseconds for each additional server. The observed behavior can be explained by saturation of the frame list locks above about 12 processes. That is, the contention for more than 12 processes is so great that one server or another holds the frame list locked at all times. Whenever the end of that critical region is reached, there is always another server already waiting to enter the same critical region, so the lock is relocked as soon as it is released. As even more servers are added, the rate of frame list lock attempts cannot increase further, and the number of servers waiting on the lock merely increases by one as each new server is added, keeping the speedup unchanged. As the number of servers is increased from n to $n+1$, the time to cycle through all the servers once, giving them each a pass through the critical region, increases by $(n+1)/n$. But since the number of such cycles required to complete the run decreases by the same factor (each server has that much less work to do), the total running time remains unchanged. Since only half of the frames that ever come into existence are actually queued for service and subsequently dequeued, as is apparent from the behavior described in Section 2[14], the rate of attempts to lock UBCQLK is limited to one half the rate for the frame list locks, which remains fixed above 12 processes. That explains why the chore queue contention per lock attempt remains fixed above that point. Consequently, as the work is spread over more and more servers, the total chore queue contention per server declines.

Finally, it is interesting to note that the preceding analysis supports the inference that the critical region inside CRFRAM or FRFRAM takes about 22 microseconds to execute, on the average. The analysis is illustrated in Figure 5, where it has been applied to a hypothetical situation involving a smaller number of processes.

The measurement of critical-region contention times was further developed and refined on the Encore Multimax. On that machine high-resolution timing is accomplished with a one-microsecond free-running counter that can be mapped to a memory location and then accessed for the cost of a memory reference. The SPINLOCK subroutine, used to acquire a lock[15], was enhanced by Encore with a pair of reads of the free-running counter's image in memory, one just before the attempt to acquire the lock and the other just after the attempt succeeds. The modified version, a function called TSPINLOCK ("timed spin lock"), returns to the caller the time spent waiting for the lock. It is hard to imagine how that can be done with less overhead.

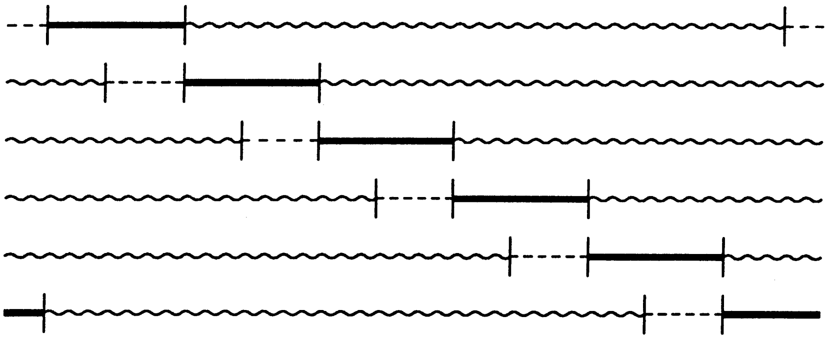
Our refinements of the contention time measurements were directed toward eliminating the measurement and locking overheads from the results before displaying them. First we had to modify Encore's TSPINLOCK so that these overheads are incurred uniformly and are identifiable[16]. Next, we added code to count the number of TSPINLOCKs for each lock, which was then used in the following way. In a one-process run, the total time reported for all the TSPINLOCK calls (which must be entirely overhead) is divided by the number of such calls to obtain the overhead per call; this is saved in a file for use in later multiprocess runs. In a multiprocess run, on the other hand, the previously saved figure is retrieved from the file. In both types of run, the measurement and locking overhead per lock attempt so determined, times the

[13]For grammatical smoothness, we refer to the multiple frame lists, and their locks, as if there were only one list and one lock.

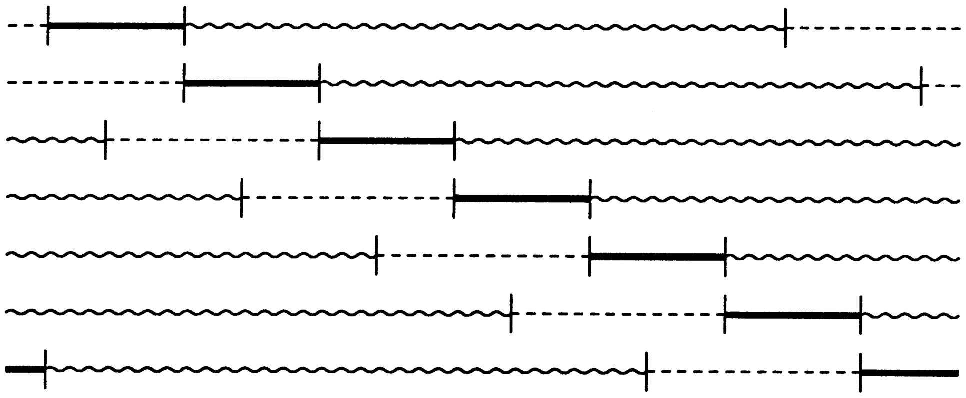
[14]After a server creates a subchore for the first recursive invocation of a *fib* chore, it requeues itself and then takes over the subchore. After a server creates a subchore for the second (i.e., last) recursive invocation of a *fib* chore, it takes over the subchore without queuing anything. Two new frames have come into existence, but only one frame has been queued.

[15]By busy waiting, as the name implies.

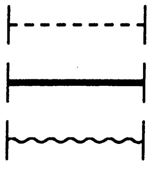
[16]Encore's original version of TSPINLOCK had the property that if the lock is found to be immediately available, the free-running counter's image is not accessed at all, and a constant value of zero (which is the true waiting time) is returned. On the other hand, if the lock is not immediately available, then the free-running counter's image is accessed once before spinning on the lock and again after the spin-wait loop is broken by successful lock acquisition, and the difference of the two readings is returned. Included in that difference is some measurement overhead, i.e., a portion of the instructions to read and save the free-running counter's image twice, and some locking overhead, i.e., the instructions that actually acquire (set) the lock—both of which are inescapable and neither of which is true waiting time. This asymmetry between the two cases tends to skew the results by increasingly overstating the true waiting time as the probability of finding the monitor lock initially locked increases (e.g., with increasing n). To eliminate the asymmetry, our modified version *always* accesses the free-running counter's image before and after the lock attempt. When the lock is immediately available, a non-zero time is therefore returned. In our version, all lock attempts incur the same measurement and locking overheads. Thus their sum can be determined: in a one-process run, the locks are always immediately available and the entire time reported by TSPINLOCK is made up of these two overheads. Once the total non-waiting overhead per lock attempt has been determined, it can effectively be subtracted from the time reported by each TSPINLOCK in a subsequent run at any number of processes to determine the true waiting time.



Above: Idealized depiction of one cycle of service for six servers assumed to be saturated on the frame monitor (see legend at bottom). The duration of one cycle is six times that of the critical region.



Above: Idealized depiction of one cycle of service after addition of a seventh server. The durations of the critical and non-critical regions remain the same. The duration of one cycle is now seven times that of the critical region. All of the additional time has gone into the wait, which has consequently lengthened by the duration of the critical region. Each cycle is now 7/6 as long as before. The addition of the seventh server allows 7/6 as much work as before to be performed in each cycle. Thus, only 6/7 as many cycles as before are now required to complete the run. The total time for the run thus remains unchanged, as does the speedup.



Server waiting to acquire frame monitor lock (held by another server)

Server holding frame monitor lock, executing critical region

Server executing code ("non-critical region") outside the frame monitor

Fig. 5. Effect of adding another server when a monitor lock is already saturated

number of lock attempts for a given lock, is then subtracted from the accumulation of TSPINLOCK times for that lock, yielding the total critical-region contention time associated with the lock. Division once again by the number of lock attempts for the lock yields the average critical-region contention time per call. Indeed, in a one-process run the critical-region contention times so calculated, both total and average-per-call, are zero but for insignificant residues.

The results obtained from these refinements on the Encore Multimax are shown in Figure 6[17]. At the time these runs were made, we found timings to be less repeatable on this machine than on the HEP, necessitating the reporting of our results by means of ranges (all of the results reported in the form of a range are for four or more runs). There were variations from one run to the next, even when no other users were on the machine. Virtual memory paging, cache sharing, and Ethernet service requirements probably all contributed to the variation in timings. Even one-process runs varied noticeably from one run to the next, both in terms of the total execution time and in terms of the calculated overhead per lock attempt; the latter, for example, varied from about 12.2 microseconds to about 13.8. While the data displayed in Figure 6 are not regular enough to permit an analysis comparable to that supported by Figure 4, they *are* useful for comparison with similar data in the next section.

True Critical-Region Contention Times vs. *n*

n	Speedup	Frame list contention time		Chore queue contention time	
		Total per server (sec.)	Per lock attempt (μsec.)	Total per server (sec.)	Per lock attempt (μsec.)
1	1.00	.0	0	.00	.0
2	1.90-1.95	1.1- 1.5	4- 6	.06- .37	.4- 2.8
3	2.80-2.83	1.8- 2.4	10- 14	.12- .62	1.3- 7.0
4	3.61-3.69	2.1- 2.7	16- 20	.17- .79	2.7-11.9
5	4.15-4.43	2.4- 3.1	23- 29	.28- .63	5.3-11.9
6	5.11-5.27	2.6- 3.3	30- 37	.28- .70	6.4-15.2
7	5.70-5.84	3.2- 3.8	42- 49	.34- .78	9.0-20.6
8	5.98-6.33	4.5- 5.1	68- 77	.36- .71	11.0-21.5
9	6.61-6.95	5.0- 5.8	85- 98	.37- .76	12.8-25.6
10	6.76-7.02	6.9- 7.8	131-148	.37-1.00	14.0-37.8
11	6.60-6.94	8.4-10.0	176-207	.34-1.18	14.1-19.0
12	6.69-7.08	10.4-11.0	236-250	.31- .86	14.0-38.8
13	6.48-6.98	11.8-13.6	289-335	.32- .58	15.7-28.3
14	6.36-7.02	13.0-15.0	344-396	.34- .92	18.0-48.6
15	6.18-6.85	13.9-16.4	394-466	.34-1.12	19.0-63.2
16	6.42-6.95	14.8-17.0	447-511	.25- .71	14.9-42.9
17	6.25-6.83	16.1-18.5	519-593	.45- .83	29.0-53.2
18	6.52-6.71	17.5-18.0	599-612	.25- .48	17.5-32.4

Program: Fibonacci
Special features: Optimized monitors,
critical-region contention time instrumentation
Machine: Encore Multimax
Conditions: Operating system X1.4
One-process time: 190.269 sec.
Overhead per lock attempt: 12.548 μsec.

Fig. 6. True critical-region contention times measured on the Encore Multimax

4. Experiments with Further Refinements of Critical Regions

Why do the frame list locks seem to be such a bottleneck, especially given the fact that there are separate monitor locks for the frame lists of each size? The answer has to do with the nature of the Fibonacci application: the bulk of the computation time is spent "recursively" computing Fibonacci numbers, wherein most chores create two subchores similar to themselves. Thus, essentially all of the frames are of one size, and nearly all of the frame creation and freeing activity is concentrated in only one list and therefore serialized with only one monitor lock. In this application, unlike others (such as the parallel version of the TAMPR transformer) in which a variety of functions having different frame sizes is invoked, the full benefits of critical-region fissioning had not been realized.

[17]The times reported for all of the runs made on the Multimax are for the computation of *fib* (25) only.

A simple experiment enabled us to demonstrate the adverse effect of uniform frame sizes on frame monitor lock contention in the Fibonacci program. We artificially increased the requested size for one of the two subchores' frames from 3 to 4, allowing the activity previously concentrated in one frame list and one lock to be distributed over two. Figure 7 shows data obtained from that experiment, performed other-

True Critical-Region Contention Times vs. *n*

n	Speedup	Frame list contention time		Chore queue contention time	
		Total per server (sec.)	Per lock attempt (μsec.)	Total per server (sec.)	Per lock attempt (μsec.)
1	1.00	.0	0	.00	.0
2	1.94- 1.95	.5- .8	2- 3	.11- .34	.8- 2.6
3	2.79- 2.89	.8-1.2	3- 7	.31- .41	3.6- 4.7
4	3.50- 3.68	.9-1.5	7- 11	.23- .44	3.5- 6.7
5	4.40- 4.70	.8-1.7	7- 16	.48- .62	9.0- 11.7
6	5.27- 5.56	.9-1.6	10- 18	.33- .54	7.5- 12.1
7	6.05- 6.36	1.2-1.8	16- 25	.39- .76	10.4- 20.0
8	6.46- 7.11	1.2-2.0	19- 30	.62-1.31	18.8- 39.4
9	7.60- 7.89	1.2-2.0	23- 34	.49- .84	16.8- 32.4
10	8.04- 8.38	1.5-2.2	29- 40	.56- .78	21.3- 29.5
11	8.27- 9.20	1.6-3.0	34- 61	.54-1.00	22.7- 41.5
12	9.21- 9.58	2.1-2.6	48- 58	.59- .92	27.0- 41.5
13	10.09-10.27	2.2-2.8	56- 69	.85- .90	41.9- 43.9
14	10.28-10.74	2.4-2.8	65- 73	.75-1.08	39.7- 57.1
15	10.51-11.06	2.7-3.4	78- 96	.76-1.33	43.3- 75.3
16	9.88-11.12	3.5-4.4	106-132	.82-2.60	49.6-157.0
17	10.14-11.82	2.5-4.0	83-128	1.03-3.49	65.9-224.0
18	10.01-11.95	3.7-5.2	128-177	.97-1.32	66.5- 89.1

Program: Fibonacci
Special features: Optimized monitors,
critical-region contention time instrumentation,
artificially varied frame sizes
Machine: Encore Multimax
Conditions: Operating system X1.4
One-process time: 191.146 sec.
Overhead per lock attempt: 12.493 μsec.

Fig. 7. True critical-region contention times measured on the Encore Multimax, with artificially varied frame sizes

wise under the same conditions as the runs whose results were presented in Figure 6 (to which it should be compared). The speedups are somewhat better, but the most dramatic improvement is seen in the frame list contention time, especially for the more highly parallel runs. Also the point at which contention begins to erase the gains made by splitting the work among more and more servers is clearly pushed farther away. Finally, it is also evident that the decreased contention for frame list locks has increased the rate at which chore queue lock attempts are made, resulting in perceptible contention for that lock in the more highly parallel runs. The speedup improvements would have been even better had they not been limited by increasing chore queue lock contention.

It is worth reviewing, for a moment, why we have a frame list monitor at all. Very simply, the frame space is considered to be a global resource, available for sharing by any process needing a frame, and the monitor provides the necessary serialization of what could otherwise be concurrent accesses to that global resource. But the frame reservoir could have been organized as a collection of local spaces, with each process accessing only its own local space. There would then be no serialization requirements at all for the management of the frame reservoir. Since space reserved for one process cannot be used by another, under this arrangement, the total space requirements could be expected to increase. In this classical space-time tradeoff, we chose originally to optimize the use of space at the possible expense of time (for serialization), not knowing what that expense might be in practice. Once we had seen that contention for the critical region controlling this resource can become a bottleneck, we developed an interest in assessing the time-optimized alternative. For our second experiment, we therefore developed a version of the Fibonacci program (called the "local frame lists" version) for comparison purposes. The contention timing results are presented in Figure 8. As one would expect, the speedup figures have again improved but the chore queue contention time has greatly increased. (The decrease in the one-process time is, of course, due to the elimi-

True Critical-Region Contention Times vs. *n*

n	Speedup	Frame list contention time		Chore queue contention time	
		Total per server (sec.)	Per lock attempt (μsec.)	Total per server (sec.)	Per lock attempt (μsec.)
1	1.00	0	0	.00	.0
2	1.94- 2.00	0	0	.32- .45	2.3- 2.5
3	2.86- 2.94	0	0	.33- .63	3.8- 6.0
4	3.80- 3.86	0	0	.49-1.04	7.4- 15.7
5	4.62- 4.80	0	0	.60- .93	11.3- 17.6
6	5.33- 5.68	0	0	.74-1.09	16.7- 24.6
7	6.33- 6.57	0	0	.73-1.13	19.5- 46.2
8	7.19- 7.43	0	0	.86-1.31	26.1- 39.5
9	7.71- 8.20	0	0	.96-1.25	32.7- 67.4
10	8.89- 9.20	0	0	1.08-1.72	40.3- 64.9
11	9.40- 9.83	0	0	1.27-1.80	53.0- 74.8
12	9.99-10.26	0	0	1.81-2.20	81.3- 99.7
13	10.27-10.91	0	0	1.82-2.45	89.4-120.3
14	10.56-11.29	0	0	2.34-2.92	124.0-154.3
15	10.15-11.68	0	0	2.56-3.15	145.1-178.0
16	11.39-11.79	0	0	3.03-3.48	183.4-210.0
17	11.43-11.94	0	0	3.59-4.36	230.6-279.0
18	11.45-12.02	0	0	4.05-4.28	275.8-290.9

Program: Fibonacci
Special features: Optimized monitors,
critical-region contention time instrumentation,
local frame lists
Machine: Encore Multimax
Conditions: Operating system X1.4
One-process time: 156.731 sec.
Overhead per lock attempt: 12.206 μsec.

Fig. 8. True critical-region contention times measured on the Encore Multimax, with local frame lists

nation of the TSPINLOCK and SPINUNLOCK subroutine calls on the frame list locks.)

Our third experiment with the refinement of critical regions represents an extension of the second one. With some care, it is possible to implement a "distributed chore queue," by which one might expect to extend the benefit of reduced contention for a global resource through critical-region fissioning to another such resource, the unblocked chore queue. The idea is to give each server its own chore queue so that calls to QCHORE and GUBKCR in different servers can proceed concurrently, without interference. But, in contrast to the local frame lists version, the distributed chore queue version cannot totally escape from the need for serialization. A server cannot be totally oblivious to the chore queues of other servers (otherwise only one server would ever get any work!). While it seems reasonable to make servers queue subchores only on their own chore queue, and to dequeue chores from their own chore queue when it is non-empty, a server finding its own chore queue empty must be allowed to look elsewhere for work; if it were not allowed to do so, an available processor would go unutilized. Since we would expect each chore queue to be non-empty most of the time, cross-server raiding of chore queues would occur infrequently, but its occasional occurrence necessitates the locking of chore queues, both local and remote, before referencing them.

Implementation decisions made during the development of the distributed chore queue version were reasoned as follows. The global chore queue head, UBCQPT, was replaced by an array, LCHQPT, indexed by a server number in the range 1 to *n*. The global chore queue monitor lock, UBCQLK, was similarly replaced by an array, LCHQLK, indexed by server number. QCHORE locks the monitor lock corresponding to the server that invokes it and appends the given chore to that server's chore queue, then unlocks the monitor lock. As stated above, the locking is necessary because of the possibility, however slight, of an attempt by another server to dequeue a chore from the same chore queue concurrently. Only infrequently would that locking be expected to produce any contention, so it is not a source of inefficiency[18]. Similarly, GUBKCR first tries to get a chore from the chore queue corresponding to the server that invokes it; it, too, locks that queue[19]. If that queue is non-empty, a chore is removed and the queue is unlocked. If it is empty, on the other hand, the queue is unlocked and the server is started on a loop in which it repeatedly cycles through all the other servers' chore queues until it finds one of them non-empty. It will definitely find one sooner or later. (The proof of this is left as an exercise for the reader. Hint: remember the "suicide" chores.) Since the amount of work involved in finding a non-empty queue is dependent on how tight this loop is, and since we desire to find a non-empty queue as quickly as possible, it is appropriate to test each queue for emptiness, as it is visited, without locking it. Thus, when a non-empty one is found, it is locked and then retested. If, upon this second examination, it is found to be still non-empty, a chore is removed from it and it is unlocked. With a small probability, however, it could be emptied (by another server) between the moment it is initially found to be non-empty and the moment it is locked just prior to the second test; in this case it is merely unlocked and abandoned in order to continue the search. The locking of non-local chore queues only when necessary to remove a chore recently known to be present and probably still present, rather than to determine whether one is present or not, is what enables us to anticipate only infrequent contention among servers for chore queue locks.

Note that this design allows us to dispense with the chore queue *delay queue lock*, UBCQSW, entirely. In the global chore queue version, UBCQSW serves to suspend a server when no work is available, until another server queues some work. In the distributed chore queue version, a server, S, is *effectively* suspended by being made to loop inside GUBKCR until work is made available by another server and S reaches the point in its loop where it sees that work on the other server's queue[20]. This loop effectively takes the place of the "spin loop" inside SPINLOCK or TSPINLOCK when that routine is called to lock UBCQSW (in the global chore queue version). This strategy might not be so effective on the HEP, where locking is performed without consuming CPU resources by delaying a process in the memory switch.

When a distributed chore queue was combined with local frame lists, the results were spectacular, as Figure 9 shows. The highest speedup we ever observed, 18.386, was attained in a run of this version at 19 processes. Critical-region contention has been reduced to a negligible amount.

Certainly some gain would have been achieved simply by fissioning the global chore queue into two chore queues, or some other small, fixed number of them. For queuing, a server would pick a queue either at random or by deterministically mapping the server number into a queue number. The starting point for the search for a non-empty queue from which to dequeue a chore would be determined in like manner. The structure and behavior of this general model reduces to our global chore queue version when the number of chore queues is fixed at one (after suitable optimizations). At the other extreme, where the number of chore queues is at least as large as the number of servers, but at no smaller number, one is guaranteed to eliminate contention entirely when two or more servers are attempting to queue chores simultaneously. Letting the number of chore queues increase with the number of servers, as we did in our distributed chore queue version, has two additional virtues: it permits a trivial mapping between servers and chore queues; and, analytically, it holds the dequeuing contention for any given chore queue lock *constant* as the number of servers increases. (When, in the limit, all the servers are looping through empty chore queues looking for work, the average *rate* of hits on any one chore queue remains unchanged as another server is added.)

Further slight improvement might be expected if a server forced to raid another server's chore queue were to take a chore from the bottom, rather than the top, of the queue. (The distributed chore queues, like

[18]If the locking could be eliminated altogether, the execution time would improve at all values of n, as it did with the local frame lists version. One might think that a strategy such as appending to one end of the queue and removing from the other end would allow the locking operation to be dispensed with when the two ends of the queue are sufficiently far apart from each other, but that is not a safe optimization; the possibility of a long service interruption on a processor that had already decided to remove a chore from its queue without locking, allowing other processors to deplete that queue before it can remove its chore, cannot be ignored.

[19]Since GUBKCR's behavior depends on whether its server's own chore queue is empty or not, a possible optimization here would be to determine whether the queue is empty before locking it. If the queue is empty, it cannot become non-empty by the action of another server, so performing that test without locking is safe in that case. If it is non-empty, it must nevertheless be locked before dequeuing an entry, since another server might empty it in the meantime. But our assumption is that it is non-empty most of the time, so this "optimization" might even produce a net loss.

[20]In the distributed chore queue version, there will sometimes be a slightly longer delay between the queuing of that work by the other server and its dequeuing by S.

True Critical-Region Contention Times vs. *n*

n	Speedup	Frame list contention time		Chore queue contention time	
		Total per server (sec.)	Per lock attempt (μsec.)	Total per server (sec.)	Per lock attempt (μsec.)
1	1.000	0	0	.000	.0
2	1.939- 1.998	0	0	.000-.087	.0- .7
3	2.900- 2.982	0	0	.007-.225	.1- 2.6
4	3.970- 3.997	0	0	.006-.065	.1- 1.0
5	4.956- 4.997	0	0	.007-.090	.1- 1.7
6	5.901- 5.966	0	0	.002-.090	.0- 2.0
7	6.930- 6.994	0	0	.004-.070	.1- 1.9
8	7.860- 7.967	0	0	.003-.054	.1- 1.2
9	8.785- 8.881	0	0	.003-.033	.1- 1.2
10	9.811- 9.922	0	0	.006-.031	.2- 1.2
11	10.911-10.973	0	0	.005-.052	.2- 2.1
12	11.666-11.964	0	0	.006-.138	.2- 6.1
13	12.731-12.859	0	0	.007-.066	.3- 3.2
14	13.647-13.798	0	0	.008-.073	.4- 3.8
15	14.420-14.646	0	0	.006-.048	.3- 2.7
16	15.427-15.675	0	0	.008-.022	.4- 1.3
17	16.434-16.724	0	0	.007-.011	.4- .7
18	17.319-17.598	0	0	.010-.203	.6-13.3*

*Anomalous value probably due to a swapout occurring while a lock is held.

Program: Fibonacci
Special features: Optimized monitors,
critical-region contention time instrumentation,
local frame lists,
distributed chore queue
Machine: Encore Multimax
Conditions: Operating system X1.4
One-process time: 159.668 sec.
Overhead per lock attempt: 12.357 μsec.

Fig. 9. True critical-region contention times measured on the Encore Multimax, with local frame lists and a distributed chore queue

the global chore queue discussed in Section 2, were implemented as LIFO lists.) Earlier frames on a chore queue, those nearer its bottom, represent larger future commitments of work, since they occurred earlier (higher) in the computation tree. These chores hold the greatest promise for generating further work (in the form of subchores that can generate further subchores, and so on). Thus, a server that has exhausted its own queue would be less likely to do so ***again*** in the near future if it were to take an "old" chore, rather than a "young" one, off some other server's queue; in other words, it could enjoy a longer period operating in the highly efficient local environment before it again had to look elsewhere for work. Contention between a server's dequeuing of a chore from the top of its own chore queue and another server's dequeuing of a chore from the bottom of that same server's queue could then be reduced to zero, except when the queue is very small, by using separate locks (or groups of locks) for dequeuing at opposite ends of the queue. However, the present statistics show no practical necessity for this improvement, so it remains only a matter of intellectual curiosity.

What are the space requirements of the local frame lists and distributed chore queue refinements? Because the former refinement makes frame space allocated by one server and subsequently freed unavailable to any other server, the total space requirement from the heap would be expected to increase, as we remarked earlier. In fact, measurements have shown that the total frame space used from the heap, in the Fibonacci program, increases approximately eightfold at 16 processes when the global frame list is replaced by local frame lists. Since the distributed chore queue does not have this property of exclusivity, i.e., chores on any chore queue remain available to all servers, one would not expect the storage requirements to be increased by implementing the latter refinement. However, at 16 processes the frame space used from the heap did double again. We believe this is due to a doubling of the maximum number of chores queued (in the aggregate) at any moment, made possible by the higher rate of doing work[21]. The hypothesis could be checked with appropriate instrumentation. In any case, reflection on the properties of the distributed chore queue refinement has suggested that a better strategy for refining the frame list might have been to implement a distributed, rather than a local, frame lists version. That is, instead of allocating from the heap when a server has exhausted its frame list of a given size, we first try to raid another server's frame list of that size. If one cycle through the other frame lists yields no usable frame, then a heap allocation would be indicated. This would be expected to increase the frame space requirements, as compared to the global frame list version, only modestly. Yet another suggestion has been to capitalize on the fact that a fixed-sized block of storage must be reserved for frames in Fortran, so the space might as well be used: one should allocate from the heap as soon as a server's frame list (of a given size) is exhausted, and if that leads to exhaustion of the heap, then—and only then—try to raid another server's frame list (of the same size).

We venture to say that the insights that led us to experiment with the refinements described in this section, and to propose others, would not have been gained without the analytical assistance provided by the critical-region contention time instrumentation.

5. Obtaining the Signature of Process-Busy Times

The techniques of the previous section can be applied to any monitor lock to determine whether it is a bottleneck, has an insignificant influence on the speedup, or lies somewhere between these extremes. If critical-region contention is indeed the cause of sublinear speedups in a particular application, these techniques will reveal it. But losses can also occur, in some applications, because of failure to generate enough subchores to keep all the servers busy. If there is insufficient parallelism in the problem, addition of one more server will not decrease the execution time (because there is little or no work for it to do) and therefore will not increase the speedup. The critical-region contention instrumentation does not provide any direct indication that servers are idle, but if the measured critical-region contention inadequately explains the observed losses in speedup, insufficient parallelism may be suspected.

The speedups achieved by the parallel version of the TAMPR transformer depended strongly on the program being transformed and the transformations being applied (both of which are input data to the transformer), but they usually fell short of those attained by the abundantly parallel Fibonacci program. When critical-region contention measurements applied to the frame monitor, the chore queue monitor, and the CONS monitor failed to account for the observed losses, we sought ways to gather direct data about the actual parallelism present throughout a run. This led to the development of another kind of instrumentation, which we describe in this section.

As discussed in Section 2, when the "main" server is started at the beginning of a run, the other $n-1$ servers, having been previously started, will be waiting for work. They will be suspended, trying to acquire the delay queue lock, UBCQSW, which is unlocked for the first time when the main server generates its first parallelizable subchore and requeues the parent. When one of the slave servers succeeds in locking UBCQSW, it also decrements the number of servers waiting in the queue for work. That count is held in the global variable UBCQCT, which decreases from $n-1$ to $n-2$ at that point. Thereafter, the value of UBCQCT fluctuates up and down as varying numbers of servers succeed in obtaining work, never going below zero (the state in which all servers are working) or above $n-1$ (the state in which only one server, the minimum if deadlock is to be avoided, is working). As the computation draws to a close, fewer and fewer chores remain for servers to perform, and the value of UBCQCT climbs for the last time to $n-1$ just prior to program termination (actually, just prior to the queuing of the first suicide chore, which initiates a progressive shutdown of servers). After the start-up transient, and before the shutdown transient, one would expect to see UBCQCT at or near zero most of the time in runs with abundant parallelism. To the extent that a significant fraction of the run is spent with UBCQCT well above zero, insufficient parallelism exists.

The most useful display, we decided, would be the signature of process-busy times presented cumulatively. The display should show the fraction F_k of the total time that k or more processes had work to do, for k ranging from 1 to n. F_1, of course, is 100%, and each succeeding value is less. If F_n is still relatively high, one may conclude that the run is highly parallel. Additional information can be gained from the

[21]By the same token, the increase in the rate at which work is done when the global frame list is replaced by local frame lists accounts for some of the increased storage requirement; not all of the octupling mentioned above is due to the inability to share storage.

detailed shape of the cumulative distribution. For example, if F_2 is much less than F_1, then the run has major parts that are essentially sequential. Similarly, if the distribution falls nearly to zero at some point, it is clear that the same performance could have been achieved with fewer processes. Finally, if the distribution is relatively flat for F_2 through F_n, and at a modest level there compared to F_1, then one may conclude that the program exhibits bimodal behavior, alternating one or more times between a highly parallel mode and an essentially sequential mode, with the fraction of time spent in the parallel mode given by the level of F_2 through F_n.

The gathering of times from which the cumulative percentages are computed is straightforward. The value of UBCQCT is used to index an array, BUSY, of time accumulators, defined so that BUSY(k) is the cumulative amount of time that k or more processes have work to do, for $k = 1$ to n. An additional array, STRTAT, is used in support of the record keeping in the following way. Whenever a server that is suspended in the delay queue (i.e., trying to lock UBCQSW in GUBKCR) is taken out of suspension by the unlocking of UBCQSW in QCHORE, that server, resuming its execution of GUBKCR, saves the current time in STRTAT(k), where $n-k+1$ is the value of UBCQCT; then it decrements UBCQCT. This records the time at which the number of busy processes last increased from $k-1$ to k. Similarly, whenever an active server runs out of work and again becomes suspended in the delay queue, by trying to lock UBCQSW in GUBKCR, it increments UBCQCT; then it takes the current time; and finally it adds to BUSY(k) the difference between that time and the time previously saved in STRTAT(k), where $n-k+1$ is the new value of UBCQCT. So when, in particular, the number of busy processes returns to $k-1$ from k, the duration of the interval during which k or more processes had work to do is added into BUSY(k).

Data gathering is initiated by a call to a subroutine called STATS0, which must be made when only one server is active. STATS0 zeroes all the elements of the BUSY array and saves the current time in STRTAT(1). Data gathering is terminated, and the results are reported, by a call to a subroutine called RPSTAT, which must likewise be called when only one server is active. RPSTAT takes the current time once again and stores in BUSY(1) the difference between that time and the time previously saved in STRTAT(1). (This is the duration of the interval during which one or more processes were busy.) Then, for $k = 1$ to n, RPSTAT computes and prints F_k, the fraction of the time that k or more servers had work to do; this is just BUSY(k)/BUSY(1).

Results were obtained on the HEP for Fibonacci and several runs of the TAMPR transformer (with a variety of transformations and programs to be transformed). With the calls to STATS0 and RPSTAT bracketing the computation of *fib*(25) in Fibonacci, the signature of process-busy times is nearly flat at 100%. For example, when run at 20 processes, the distribution declines from 100% for F_1 only to 99.89% for F_{20}, revealing that all 20 servers had chores to perform 99.89% of the time. By way of contrast, Figure 10 shows the signature of process-busy times obtained by the transformer in one of the phases of a tranformational application which constructs type specification statements for undeclared Fortran identifiers. It reveals a bimodality in which roughly one-fifth of the time is spent in highly parallel behavior and four-fifths of it in essentially sequential behavior. Other transformational applications, or phases of them, seem to be characterized by the same bimodality, with the fraction of time spent in the highly parallel mode varying anywhere from 7% to about 95%. That this behavior, and in general the overall speedup achieved with the transformer, is so strongly dependent on the input data (transformations and program to be transformed) is due to the fact that different parts of the transformational algorithm have varying degrees of inherent parallelism, and the fraction of time spent in each part is data dependent. For example, the post-order traversal of the program tree, looking for places where transformations could apply (based on the transformations submitted), is highly parallel; the pattern matching at each of these places, to determine whether a transformation does apply, is inherently sequential (a requirement of the semantics of certain aspects of the pattern matching language); and the construction of the replacement tree, following a successful pattern match, is again parallel.

We have found that the data presented by this technique can be overwhelming. While it can indeed reveal unexpected properties of one's parallel program, it presents more data than is required to answer the simple question, "How much speedup is lost because of inadequate parallelism in the application?" Except for a few runs made on the HEP, we have not pursued this measurement technique further. It has largely been supplanted by the technique described in the next section.

Signature of process-busy times
1 or more servers had work to do for 15.207 sec (100.00% of the total time)
2 or more servers had work to do for 3.415 sec (22.45% of the total time)
3 or more servers had work to do for 3.306 sec (21.74% of the total time)
4 or more servers had work to do for 3.218 sec (21.16% of the total time)
5 or more servers had work to do for 3.164 sec (20.81% of the total time)
6 or more servers had work to do for 3.131 sec (20.59% of the total time)
7 or more servers had work to do for 3.097 sec (20.37% of the total time)
8 or more servers had work to do for 3.065 sec (20.16% of the total time)
9 or more servers had work to do for 3.041 sec (20.00% of the total time)
10 or more servers had work to do for 3.026 sec (19.90% of the total time)
11 or more servers had work to do for 3.012 sec (19.81% of the total time)
12 or more servers had work to do for 2.998 sec (19.72% of the total time)
13 or more servers had work to do for 2.990 sec (19.66% of the total time)
14 or more servers had work to do for 2.982 sec (19.61% of the total time)
15 or more servers had work to do for 2.970 sec (19.53% of the total time)
16 or more servers had work to do for 2.957 sec (19.45% of the total time)
17 or more servers had work to do for 2.946 sec (19.37% of the total time)
18 or more servers had work to do for 2.936 sec (19.30% of the total time)
19 or more servers had work to do for 2.927 sec (19.25% of the total time)
20 or more servers had work to do for 2.918 sec (19.19% of the total time)

Program: TAMPR transformer
Special features: Optimized monitors,
process-busy time instrumentation
Machine: Denelcor HEP
Conditions: Path to local memory disabled

Fig. 10. Typical signature of process-busy times

6. A Novel Technique for Measuring Speedup and the Factors That Limit It

Up to this point, our speedups have been computed by dividing the one-process execution time by the multiprocess execution time, which requires that two runs be made. We gradually came to realize that the speedup achieved in a multiprocess run can be determined *without ever making a one-process run*; that is, the multiprocess run, suitably instrumented, can be used to *predict* the one-process execution time, regardless of the amount of degradation resulting from critical-region contention and lack of parallel work.

Two assumptions made in Section 1 allow us to predict the one-process execution time during a multiprocess run. One is that the computational work performed during a run is independent of the number of processes sharing in that work. Thus, if the sum of the execution times for each of n processes working together to solve a problem is W, then one process running by itself should be expected to require W units of time to solve the same problem. The second assumption is that the other activity in the system is negligible during a run in which timings are being taken, so that one may conclude that a process is performing useful computational work except when it is waiting for a lock.

If the second assumption does in fact hold, then the computational work performed in a multiprocess run can be determined by subtracting from the aggregate execution time of all the processes the durations of all the waits for locks. (Each wait causes a process to stop performing computational work, and by assumption nothing else does.) If the first assumption also holds, then the computational work determined as above in a multiprocess run can be taken as the computational work that would be performed by a hypothetical one-process run of the same program. Finally, using the obvious fact that there are no waits for locks in a one-process run, another application of the second assumption allows us to equate its computational work with its execution time.

Our method for measuring speedup during a multiprocess run based on the predicted one-process execution time is as follows. The portion of the run to be measured is bracketed by calls to two subroutines, UZEROB and UREPTB, analogous to STATS0 and RPSTAT of the previous section. Like those earlier routines, UZEROB and UREPTB may be called only when all the servers (except the one calling them) are waiting for work. UZEROB records the start time of the measurement interval. UREPTB takes the time again and determines the duration, T_n, of the interval; this is the multiprocess execution time.

If there were to be no waits for locks during the interval, then the computational work performed during it would be nT_n. Taking the waits into account, the computational work, W, is computed as nT_n minus the sum of all the waits. By the earlier assumptions, W is also the predicted one-process execution time, and the speedup based on it is W/T_n.

The sum of all the waits is determined as follows. Each wait is timed as it is performed, using the modified TSPINLOCK of Section 3. Each server accumulates its *own* waiting time for each lock, or class of locks[22], separately. Accumulating lock-related statistics separately in each server has the advantage of allowing the accumulation to be performed after the end of the associated critical region in the server, so as not to unduly lengthen the critical region and unwittingly cause more contention. (That advantage is discussed in [2].) To support the separate, unserialized, accumulations by each server while giving the server executing UREPTB access to all of them, a shared array of wait-time accumulators (indexed by server number) is associated with each lock or class of locks. A shared array of counters, used to record the number of lock attempts, is similarly associated with each lock or class of locks. The counters are used, as described in Section 3, to remove the measurement and locking overhead from the times reported by TSPINLOCK.

Unhampered by other servers, the server that executes UREPTB sums the elements in each array and performs the other steps required to obtain the total waiting time for each lock or class of locks. Each such total is then subtracted from nT_n, yielding W. UREPTB then displays the results, examples of which are shown in Figure 11. In addition to the predicted one-process execution time (W) and the speedup based on it (W/T_n), UREPTB displays the speedup lost to contention for each lock or class of locks. The speedup lost to chore queue lock contention, for example, is equal to the total waiting time for the chore queue monitor lock divided by T_n; the lost speedup associated with other locks is computed in like manner. The speedup loss associated with the delay queue lock, UBCQSW, is interpreted not as contention but as the loss attributable to lack of parallel work[23].

The units in which these speedup losses are reported are the same as the units in which the speedup itself is reported, e.g., *processor-equivalents*. (Note that the losses plus the speedup sum to n.) The impact of seeing the losses reported in the same units as the speedup cannot be overemphasized; the knowledge that lock contention (or lack of parallel work) caused a loss of speedup equivalent to what some particular number of processors would have bought leaves an indelible impression not fostered by merely knowing that a certain number of seconds was spent waiting for locks.

The implementation described above is affected by the fact that UBCQSW, unlike the other locks, is initially and at most later times locked rather than unlocked; in particular, it is locked (and $n-1$ servers are waiting for it to be unlocked) when UZEROB, and again when UREPTB, are called. This fact has two adverse consequences. One is that the $n-1$ TSPINLOCKs on UBCQSW that are already in progress when UZEROB is called will report, upon their subsequent termination, an unknown amount of waiting time from *before* the start of the measurement interval (i.e., from before the start time recorded by UZEROB). The second is that the $n-1$ TSPINLOCKs on UBCQSW that are already in progress when UREPTB is called will not yield their contribution to the legitimate waiting time, since they are not terminated within the measurement interval. Because the magnitude of the errors introduced by these problems can be arbitrarily large[24], they cannot be ignored.

[22]We could accumulate the waiting time for each individual lock, but that would provide more detail than we really need. Thus we group mutually exclusive locks serving a similar function into a class of locks and accumulate the waiting time for any lock in the class. The locks guarding the frame lists comprise one such class; the distributed "count monitor" locks comprise another.

[23]Because of the particular structure of our programs, lack of parallel work may always be equated simply to the time spent waiting for this one particular lock. In programs obtained by other methods, different indicators of the lack of parallel work must be sought. It is always possible to find them, however, since lack of parallel work *means* that one or more processes are suspended (i.e., not working), and the only way to suspend processes is to execute a synchronization primitive that *causes* process suspension under the appropriate conditions. For example, *barrier synchronization* may be used to suspend processes at given places (such as the ends of certain loops) until other processes "catch up." These process suspensions represent a lack of parallel work in the strict sense. In general, one has the freedom to report separately the time spent waiting at different barriers, or even to interpret these wait times in unique ways.

[24]The final wait from before the start of the measurement interval, which is included but should not be, could be much longer than the measurement interval itself. This occurs, for instance, when a brief measurement interval is preceded by a long period during which the parallelism is so low that one or more servers end that period with a long wait for work; it will have the effect of rendering the speedup computation meaningless, since the sum of the apparent waits during the measurement interval could appear to exceed the length of the interval. Similarly, each server's final wait at the end of the measurement interval, which is not included but should be, could be nearly as long as the measurement interval itself. In that case most of the actual waiting time in the interval is not seen; since

```
Examples of outputs from UREPTB
Actual time for fib(7) =    0.012 sec.
  3.034 = actual speedup (compare to potential of 12.000)
  0.379 = speedup lost to frame list lock contention
  0.001 = speedup lost to frame heap lock contention
  0.096 = speedup lost to chore queue lock contention
  0.000 = speedup lost to count lock contention
  8.490 = speedup lost to lack of parallel work
Predicted one-process time for fib(7) =    0.035 sec.
Actual time for fib(20) =    1.997 sec.
  7.544 = actual speedup (compare to potential of 12.000)
  4.266 = speedup lost to frame list lock contention
  0.004 = speedup lost to frame heap lock contention
  0.119 = speedup lost to chore queue lock contention
  0.002 = speedup lost to count lock contention
  0.065 = speedup lost to lack of parallel work
Predicted one-process time for fib(20) =  15.064 sec.
Actual time for fib(25) =  23.374 sec.
  7.298 = actual speedup (compare to potential of 12.000)
  4.528 = speedup lost to frame list lock contention
  0.000 = speedup lost to frame heap lock contention
  0.117 = speedup lost to chore queue lock contention
  0.019 = speedup lost to count lock contention
  0.038 = speedup lost to lack of parallel work
Predicted one-process time for fib(25) = 170.596 sec.
```

Program: Fibonacci
Special features: Optimized monitors,
predictive speedup and losses instrumentation
Machine: Encore Multimax with recent hardware upgrades
Conditions: Operating system X1.4
Overhead per lock attempt: 10.467 μsec.

Fig. 11. Example of outputs produced by the predictive speedup and losses instrumentation during a run using 12 processes

To solve both of these problems, we prepared a special version of TSPINLOCK, called TSPINLOCKQ, which is used only to perform (and time) the lock attempt on UBCQSW. Unlike TSPINLOCK, which saves its initial reading of the free-running counter in a register for the duration of the wait, TSPINLOCKQ saves that time in an element of a shared array, indexed by the server number. (A reference to that array element is passed as an additional argument when TSPINLOCKQ is called.) Storing the start time of the wait in that way permits the server executing UZEROB to update the start times of the other $n-1$ servers' preexisting waits for work, setting them equal to the time it records as the start of the measurement interval. Similarly, the server executing UREPTB is able to compute the duration of the other $n-1$ servers' final waits for work without actually terminating them[25].

The computational work performed in a multiprocess run will be recognized as the integral, over time, of the number of active (not waiting) processes, and the speedup is that integral divided by the length of the interval over which it is computed. It is important to note that we do not compute that integral directly, e.g., by updating a weighted sum each time the number of active processes changes. We arrive at

it is effectively mistaken for work, the speedup will be drastically overestimated.

[25]The data presented in this paper were obtained using a different solution to the problem of waits for work that straddle the start and end of the measurement interval—a solution that was much more elaborate and, as we tardily discovered, slightly faulty. It involved the queuing of some "artificial" work at the start and end of the measurement interval to terminate the waits extant at those points, plus some contortions intended to correct for the second-order effect of the staggered receipt of the artificial work by the other servers. Before deciding to use our original data, we verified that both methods provide the same results to within the level of noise (despite the error in the original method). The present method is easier to describe, easier to implement, and—we can say from experience—more likely to be implemented correctly. Finally, it is worth noting that the additional memory references made by TSPINLOCKQ do not represent significant additional overhead, because TSPINLOCKQ, when limited to the locking of UBCQSW only, is called very few times indeed: for the computation of *fib* (25), only about 40 of the 141403 requests for chores result in a wait for work (at 12 processes), and 22 of those represent the waits in effect at the start and end of the measurement interval.

an analytically and numerically equivalent value by distributing the accumulation of the components of an appropriate sum over the processes in such a way that no additional serialization is required.

How tenable are the assumptions on which our method of measuring the speedup rests? The first, that the work performed is independent of the number of processes, is by design as nearly true as we can make it. The same functions are computed, and the same list structures traversed, as the number of processes is varied; only the distribution of that work among the processes varies. Nevertheless, under very close scrutiny one can find small differences in the code that is executed for two different values of n. One obvious example is that at one process there is never a wait for work: whenever the one process asks for work, there is some on the chore queue. Thus, the code in the procedures of the chore queue monitor to suspend and resume processes in the delay queue, and to increment and decrement the number of processes waiting there, is never executed in a one-process run. But this difference is clearly insignificant because it amounts to a few instructions executed (in a multiprocess run) a few dozen times. Another example is that the number of frames allocated from the heap increases with increasing parallelism. This, too, amounts to only two or three hundred extra executions of a few instructions in a multiprocess run (a total of perhaps five milliseconds).

Unfortunately, we do not have such good control over system and hardware issues. For example, the larger working set in the more highly parallel runs might decrease the cache hit ratio and increase page faults, both leading to slower memory references which would be interpreted as "more work." Indeed, we do observe some such effect, since the predicted one-process times (computed in multiprocess runs) are consistently too high by an amount that increases with the parallelism, reaching three to five percent at 12 processes.

One can imagine other ways of structuring and organizing fine-grained parallel programs which would not result in an invariant amount of work. For example, by searching a series of spaces in parallel instead of sequentially for an item of interest, and aborting the fruitless searches when the item is found, one may do either more or less work than in the sequential run. The measurement technology described here is still useful; however, the result should be interpreted as "average concurrency" rather than speedup, since the one-process and multiprocess runs are no longer strictly comparable.

The second assumption, that the other activitiy in the system is negligible during an instrumented run, is certainly not guaranteed to hold, but to varying degrees it may be controllable. Any activity that steals cycles from a processor can increase either the apparent amount of work (if a lock is not being held by that processor at the time) or both the apparent amount of work and the apparent lock contention time (if the processor is holding a lock). Our runs were made with the cooperation of other users, who graciously permitted us sole access to the machine. Even so, occasional runs produced times that were way out of line with the modal time, and those were discarded. (Periodically executed preemptive system processes could account for this interference.)

We present below a selection of results produced by the predictive speedup and losses instrumentation. Figure 12[26] shows the speedup and losses as a function of n for runs of Fibonacci incorporating optimized monitors, but none of our later refinements. Except for the instrumentation code, this version of Fibonacci is identical to that used to produce the data for Figure 6[27]. One may see the saturation of the frame monitor lock in several ways: the leveling off of the speedup above about nine processes; the increase of approximately one in the speedup lost to frame list lock contention as each new process is added, above nine; and the leveling off of the speedup lost to chore queue lock contention, above nine processes.

Figure 13 is for the version incorporating artificially varied frame sizes (the same version, except for the instrumentation code, used to produce Figure 7). As we saw in Section 4, the (artificial) use of different sizes for the frames allocated for the two recursive subchores of a *fib* chore reaps the benefits of critical-region fissioning as applied to the frame monitor. In contrast to the previous figure, the speedups shown in Figure 13 continue to rise beyond nine processes, with the first hint of leveling off coming at the end of the table. For the more highly parallel runs, the speedup lost to frame list lock contention is down dramatical-

[26]In this figure and the next three, we have omitted the column showing the loss of speedup attributable to frame heap lock contention, since it is nearly always zero.

[27]The higher speedups achieved with the present version are due to the differences in the instrumentation code. Wait times and counts were not obtained and accumulated for the frame heap lock, the delay queue lock, or the count locks in the version used for Figure 6. They *are* obtained in the present version, where their accumulation outside of critical regions represents additional parallelizable work.

Predictive Speedup and Losses vs. *n*					
		Speedup lost to...			
n	Speedup	Frame list lock contention	Chore queue lock contention	Count lock contention	Lack of parallel work
1	1.000	.000	.000	.000	.000
2	1.970-1.976	.017- .022	.002-.007	.000-.004	.001
3	2.900-2.935	.054- .079	.006-.013	.000-.006	.002
4	3.811-3.863	.117- .166	.013-.019	.001-.004	.004-.014
5	4.638-4.745	.213- .278	.033-.076	.001-.015	.007
6	5.380-5.515	.333- .502	.044-.134	.001-.007	.008-.011
7	6.085-6.235	.688- .771	.063-.180	.002-.022	.010-.047
8	6.822-6.879	.908-1.016	.132-.190	.002-.011	.015-.020
9	7.289-7.462	1.411-1.577	.099-.111	.003-.012	.017-.024
10	7.440-7.699	2.084-2.219	.115-.332	.003-.019	.017-.038
11	7.440-7.871	2.958-3.238	.134-.299	.004-.021	.026-.033
12	7.668-7.808	3.904-4.008	.143-.398	.004-.022	.024-.038
13	7.491-7.766	4.917-5.342	.129-.260	.005-.022	.025-.035
14	7.557-7.670	5.929-6.188	.144-.364	.005-.014	.032-.106
15	7.789-7.929	6.847-7.007	.147-.259	.007-.015	.036-.098

Program: Fibonacci
Special features: Optimized monitors,
predictive speedup and losses instrumentation
Machine: Encore Multimax
Conditions: Operating system X1.4
One-process time: 204.705 sec.
Overhead per lock attempt: 12.428 μsec.

Fig. 12. Predictive speedup and losses using optimized monitors

ly, while the speedup lost to chore queue lock contention is higher and still rising at the end of the table. The 15th process provided almost no additional speedup and, in fact, was lost to a combination of frame list and chore queue lock contention.

Figure 14 is for the local frame lists version (the same version, except for the instrumentation code, used to produce Figure 8). Speedups are, of course, higher still, and with no frame list lock contention to pace the progress of the program the chore queue lock becomes a moderate bottleneck and causes a much more significant loss of speedup than before. Approximately half of the 15th process was lost to chore queue lock contention.

Figure 15 is for the version incorporating local frame lists and a distributed chore queue (the same version, except for the instrumentation code, used to produce Figure 9). Since this version replaces the delay queue lock as the means of suspending a server when no work is available by a loop which cycles through other servers' chore queues until work is found on one of them, the use of TSPINLOCKQ to time a simple lock attempt as the basis for measuring the speedup lost to lack of parallel work had to be replaced by another technique. The times at the beginning and end of the wait for work were obtained, on the Encore Multimax, by reading the free-running counter directly (using GETTMR). But how shall we define what constitutes a wait for work? On some occasions, a server's own chore queue may well be empty, but that server will find work on another server's chore queue on its first cycle through the other queues. While that work may or may not have been present at the instant the server found its own queue empty, it is rather likely that it *was* present. Thus, it seems clear that the brief period from the discovery that its own queue is empty to the finding of work on its first examination of some other server's queue should not be considered a wait for work. Work *existed*; it just took a little while to retrieve it. On the other hand, if one cycle through the other servers' chore queues turns up no work, we might reasonably say that no work is available. In our instrumentation of GUBKCR for the distributed chore queue version, we define the start of the wait for work to be the end of the first cycle through the other servers' chore queues, if that point is reached without finding work. It ends, of course, when work is found on some queue. As before, the first fruitless examination of each other server's queue is considered to be "useful work," i.e., part of the work

Predictive Speedup and Losses vs. *n*

n	Speedup	Speedup lost to...			
		Frame list lock contention	Chore queue lock contention	Count lock contention	Lack of parallel work
1	1.000	.000	.000	.000	.000
2	1.973- 1.982	.011- .020	.002- .009	.000-.002	.001-.002
3	2.927- 2.943	.034- .056	.008- .020	.003-.007	.002
4	3.853- 3.891	.062- .108	.030- .064	.000-.009	.003-.005
5	4.730- 4.818	.115- .162	.053- .099	.001-.011	.006-.007
6	5.613- 5.706	.173- .289	.052- .121	.001-.020	.010-.012
7	6.541- 6.646	.196- .323	.119- .173	.002-.016	.013-.019
8	7.253- 7.411	.260- .548	.166- .298	.002-.012	.014-.027
9	8.044- 8.357	.439- .702	.154- .246	.003-.029	.020-.025
10	8.927- 9.241	.491- .756	.221- .342	.004-.035	.030-.047
11	9.543- 9.785	.707- .947	.265- .675	.015-.035	.026-.110
12	10.276-10.376	1.085-1.197	.456- .473	.006-.019	.042-.071
13	10.671-11.370	1.169-1.641	.405- .669	.007-.020	.012-.056
14	11.240-11.962	1.443-2.027	.482- .990	.009-.034	.056-.065
15	11.490-12.186	1.900-2.371	.757-1.398	.010-.013	.058-.070

Program: Fibonacci
Special features: Optimized monitors,
predictive speedup and losses instrumentation,
artificially varied frame sizes
Machine: Encore Multimax
Conditions: Operating system X1.4
One-process time: 202.731 sec.
Overhead per lock attempt: 12.505 μsec.

Fig. 13. Predictive speedup and losses with artificially varied frame sizes

necessary to get the chore that is eventually found[28].

7. Conclusions

Many factors affect the performance of parallel programs. Software structure and programming methodology are now recognized as having potentially profound effects on program performance [4], certainly more than was true before the advent of multiprocessors. When synchronization primitives implement process suspension by "busy waiting," the resources of entire general-purpose processors—not just small, highly specialized functional units—can be wasted by bad design decisions. The incentive for measuring performance is strong under these conditions.

Speedup continues to be a useful measure of performance of parallel programs; it requires neither special hardware nor program instrumentation, and it can serve as a basis for comparing refinements of synchronization protocols. By itself, however, it fails to provide any clues as to why an unexpectedly slow parallel program does not achieve its full potential. And because the removal of one bottleneck may produce another, it may even mislead one into thinking that a particular refinement had no effect, when in fact it had precisely the desired effect of reducing contention for a particular lock.

We have shown how a high-resolution, low-overhead clock can form the basis of a performance measurement methodology that provides more information about the behavior of fine-grained parallel programs. In its most highly developed form, our measurement technique not only determines the speedup achieved by a portion of a parallel program during a single multiprocess run, but it simultaneously reveals how much potential speedup is *not* achieved because of either contention for the resources used to implement synchronization or a lack of parallelizable work. Contention statistics are obtained separately for

[28]Many additional, potentially interesting, statistics come to mind. We leave it to future researchers to answer the following questions. How often does a server find its own chore queue empty? How often does it find its own queue empty, but another nonempty the first time it looks there? In this case, through how many queues does it have to look before finding the work? (Obtain the distribution.) How often does it truly enter a "wait for work"? How much speedup is lost to the retrieval of a chore from some other server's chore queue (i.e., to the examination of up to $n-1$ other queues before either finding work or entering a wait for work)?

Predictive Speedup and Losses vs. *n*					
		Speedup lost to...			
n	Speedup	Frame list lock contention	Chore queue lock contention	Count lock contention	Lack of parallel work
1	1.000	0	.000	.000	.000
2	1.987- 1.995	0	.004- .007	.000-.006	.001
3	2.953- 2.982	0	.015- .032	.001-.013	.002-.006
4	3.910- 3.953	0	.035- .083	.002-.008	.004-.006
5	4.817- 4.921	0	.067- .140	.003-.017	.009-.029
6	5.822- 5.868	0	.107- .146	.012-.020	.011-.014
7	6.730- 6.749	0	.227- .230	.004-.024	.015-.019
8	7.629- 7.731	0	.241- .330	.007-.017	.021-.026
9	8.353- 8.603	0	.345- .618	.006-.018	.022-.034
10	9.234- 9.365	0	.589- .719	.008-.034	.037-.039
11	9.877-10.296	0	.642-1.070	.010-.037	.025-.051
12	10.693-11.046	0	.933-1.227	.010-.026	.006-.070
13	11.311-11.700	0	1.216-1.646	.012-.028	.007-.179
14	12.090-12.296	0	1.662-1.845	.014-.058	.006-.018
15	12.530-12.790	0	2.160-2.398	.016-.052	.014-.022

Program: Fibonacci
Special features: Optimized monitors,
predictive speedup and losses instrumentation,
local frame lists
Machine: Encore Multimax
Conditions: Operating system X1.4
One-process time: 165.883 sec.
Overhead per lock attempt: 12.468 μsec.

Fig. 14. Predictive speedup and losses with local frame lists

each synchronization lock, allowing one to observe directly the success, as well as the sometimes unexpected side effects, of efforts to reduce contention for a particular lock.

Of even more importance than the statistics themselves is the insight that can be gained from their interpretation. One soon learns the importance of sparing no effort in the attempt to make critical regions as brief as possible and to decentralize the resources that need to be protected by them (subject to other con-

Predictive Speedup and Losses vs. *n*					
		Speedup lost to...			
n	Speedup	Frame list lock contention	Chore queue lock contention	Count lock contention	Lack of parallel work
1	1.000	0	.000	.000	.000
5	4.975- 4.997	0	.000-.007	.000-.005	.009-.014
10	9.831- 9.974	0	.000-.052	.000-.025	.005-.177
15	14.875-14.977	0	.000-.067	.000-.033	.025-.031

Program: Fibonacci
Special features: Optimized monitors,
predictive speedup and losses instrumentation,
local frame lists,
distributed chore queue
Machine: Encore Multimax
Conditions: Operating system X1.4
One-process time: 166.932 sec.
Overhead per lock attempt: 12.882 μsec.

Fig. 15. Predictive speedup and losses with local frame lists and distributed chore queue

straints, such as storage utilization). Our own experience with the measurement techniques enabled us to explore several methods of optimizing critical regions, which we cataloged in Section 3.

Frequent execution of critical regions, even non-trivial ones, *need not* be a source of contention, as we showed in our discussion of the distributed chore queue monitor in Section 4. Generalizing, we feel justified in saying that fine-grained parallelism need not be inefficient. After reducing contention losses to a negligible amount, there remains, of course, the overhead necessary for parallel execution (the calls to monitor procedures, the less efficient linkage of dynamic storage elements obtained from a heap instead of a stack, etc.). Is the cost of parallelization, especially when one organizes it to minimize contention, high? Our parallel (distributed chore queue) version of Fibonacci, when run at one process, takes 60% longer than the unparallelized version. Evaluated in those terms, the overhead for parallelization seems high. Looked at another way, though, it seems acceptable: the same parallel version, at 19 processes, runs 11.5 times faster than the *serial* version.

Our experiences with the signature of process-busy times reported in Section 5 fostered a better understanding of the algorithms embodied in the TAMPR transformer, and their implementation. They also suggested an experimental non-conservative implementation in which additional speedup was sought by putting otherwise idle servers to work on speculative pattern matching attempts out of their semantically ordained order.

Finally, we remark that a high-resolution, low-overhead clock is not commonplace in today's parallel machines. Accordingly, we hope that our studies will have the side effect of establishing its value as a research tool, and that it will be seen more often as a standard hardware feature of parallel computers.

References

1. J. M. Boyle, K. W. Dritz, M. N. Muralidharan, and R. J. Taylor, "Deriving Sequential and Parallel Programs from Pure LISP Specifications by Program Transformation," *Proc. IFIP WG2.1 Working Conference on Programme Specifications and Transformations*, North-Holland, Amsterdam (April 1986).
2. D. A. Fisher and R. M. Weatherly, "Issues in the Design of a Distributed Operating System for Ada," *Computer* 19(5), pp. 38–47 (May 1986).
3. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *CACM* **17**(10), pp. 549–557 (October 1974).
4. D. J. Kuck *et al.*, "The Effects of Program Restructuring, Algorithm Change, and Architectural Choice on Program Performance," *Proc. 1984 International Conference on Parallel Processing*, pp. 129–138, IEEE Computer Society Press, Silver Spring, Maryland (1984).
5. E. L. Lusk and R. A. Overbeek, "Use of Monitors in FORTRAN: A Tutorial on the Barrier, Self-Scheduling DO-loop, and Askfor Monitors," ANL-84-51, Argonne National Laboratory (July 1984).
6. P. Møller-Nielsen and J. Staunstrup, "Saturation in a Multiprocessor," *Proc. IFIP 83*, pp. 383–388, North-Holland, Amsterdam (1983).
7. S. E. Madnick, "Multi-processor Software Lockout," *Proc. 23rd ACM National Conference*, pp. 19–24, Brandon/Systems Press, Princeton (1968).
8. S. E. Madnick and J. J. Donovan, *Operating Systems*, McGraw-Hill, New York (1974).
9. R. F. Vaughan and M. S. Anastas, "Limiting Multiprocessor Performance Analysis," *Proc. 1979 International Conference on Parallel Processing*, pp. 55–64, IEEE Computer Society Press, New York (1979).

Distribution for ANL-87-7

Internal:

J. M. Beumer (2)
J. M. Boyle (35)
K. W. Dritz (49)
A. B. Krisciunas
P. C. Messina
G. W. Pieper
E. P. Steinberg

ANL Patent Department
ANL Contract File
ANL Libraries
TIS Files (5)

External:

DOE-TIC, for distribution per UC-32 (165)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
  J. L. Bona, Pennsylvania State University
  T. L. Brown, U. of Illinois, Urbana
  P. Concus, LBL
  S. Gerhart, MCC, Austin, Texas
  G. H. Golub, Stanford University
  W. C. Lynch, Xerox Corp., Palo Alto
  J. A. Nohel, U. of Wisconsin, Madison
D. Austin, ER-DOE
G. Michael, LLL