42

WorkCrews: An Abstraction for Controlling Parallelism

by Eric S. Roberts and Mark T. Vandevoorde

April 2, 1989

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984 — their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

WorkCrews: An Abstraction for Controlling Parallelism

Eric S. Roberts and Mark T. Vandevoorde

April 2, 1989

Mark T. Vandevoorde is a member of the Laboratory for Computer Science at the Massachusetts Institute of Technology, Cambridge, Massachusetts, 02139.

©Digital Equipment Corporation 1989

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Systems Research Center of Digital Equipment Corporation in Palo Alto, California; an acknowledgment of the authors and individual contributors to the work; and all applicable portions of the copyright notice. Copying, reproducing, or republishing for any other purpose shall require a license with payment of fee to the Systems Research Center. All rights reserved.

Authors' abstract

When implementing parallel programs, it is important to find strategies for controlling parallelism that make the most effective use of available resources. In this paper, we introduce a dynamic strategy called WorkCrews for controlling the use of parallelism on small-scale, tightly-coupled multiprocessors. In the WorkCrew model, tasks are assigned to a finite set of workers. As in other mechanisms for specifying parallelism, each worker can enqueue subtasks for concurrent evaluation by other workers as they become idle. The WorkCrew paradigm has two advantages. First, much of the work associated with task division can be deferred until a new worker actually undertakes the subtask, and avoided altogether if the original worker ends up executing the subtask serially. Second, the ordering of queue requests under WorkCrews favors coarse-grained subtasks, which reduces further the overhead of task decomposition.

Eric S. Roberts and Mark T. Vandevoorde

Contents

1. Overview

In implementing parallel programs, identifying the opportunities for concurrency is only part of the problem. In most cases, it is equally important to recognize that unrestricted parallelism can lead to inefficiency. When this occurs, it is important to find strategies for controlling parallelism in order to make the most effective use of available resources. In this paper, we introduce a dynamic strategy for controlling the use of parallelism on small-scale, tightly-coupled multiprocessors. That strategy is based on WorkCrews—a scheduling abstraction for parallel programs originally developed for a parallel C compiler in Mark Vandevoode's Master's thesis [Vandevoorde88].

In the WorkCrew model, the decision to subdivide a task does not rely solely on instantaneous information about processor availability. Instead, potential task subdivisions are queued for execution by any processor which becomes available while the requesting processor is busy. Using a queue extends the window during which the subdivision can be made and increases the available parallelism.

The idea of using a pool of cooperating workers reading tasks from a queue is not new. This idea was used in the C.mmp and Cm* projects at Carnegie-Mellon [Oleinick78, Gehringer87, Hibbard78] and forms the basis of the problem heap paradigm used at Aarhus University in Denmark [Møller-Nielsen85]. It is also related to the implementation strategy used for "strips" in the BBN Pluribus [Ornstein75] and for "futures" in Multilisp [Halstead85]. Although based on this old idea, the WorkCrew model offers two important advantages. First, the WorkCrew strategy often makes it possible to avoid some of the overhead associated with task subdivision. This is accomplished by permitting "lazy evaluation" of any work that is required only when concurrent execution actually occurs. When tasks are processed serially, these costs can be avoided, resulting in significant efficiency increases for some applications. Second, the ordering of task queue entries under the WorkCrew strategy follows the recursive decomposition of the problem. This gives preference to coarse-grained decompositions that usually offer the best opportunities for speedup.

In section 2, we establish the context for this work by illustrating the problems associated with unbridled concurrency. Section 3 offers a simple solution to this problem and discusses some of the opportunities for improvement that remain. Section 4 presents the basic WorkCrew mechanism, and section 5 outlines an adaptation of the basic model necessary to retain intuitive procedural semantics. Section 6 demonstrates the use of lazy evaluation to reduce the overhead cost in decomposition. Section 7 and 8 discuss implementation and performance, respectively, and we offer some general conclusions in Section 9.

2. The need to limit concurrency

On a typical multiprocessor, there is no performance advantage in having more runnable threads than available processors. Instead, this situation represents a performance liability, since the additional threads imply increased scheduler overhead. To illustrate the importance of controlling parallelism, this section develops a simple parallel implementation of the standard Quicksort algorithm [Bentley84, Hoare62, Sedgewick78]. Figure 1 illustrates a Modula-2+ implementation of Quicksort that does not involve concurrency. Modula-2+ [Rovner85] is an extension of Wirth's standard Modula-2 [Wirth85] that includes new primitives in support of concurrency. These extensions are supplied via the **Thread** interface [Birrell87, Birrell89], which makes it possible to create new lightweight processes, that is, processes that share the same address space.

For small numbers of items, Quicksort is less efficient than other sorting algorithms. Therefore, the **Quicksort** procedure is optimized to call **SelectionSort** when there are fewer than **MinQuick** elements

```
PROCEDURE Quicksort(array: RefIntArray; low, high: INTEGER);
VAR
  boundary: INTEGER;
BEGIN
  IF high - low < MinQuick THEN
    SelectionSort(array, low, high);
  ELSE
    boundary := Partition(array, low, high);
    Quicksort(array, low, boundary-1);
    Quicksort(array, boundary+1, high);
  END;
END Quicksort;
```

Figure 1
SerialQuicksort

---

to be sorted. In the general case, the **Quicksort** procedure calls **Partition** to divide the array into partitions which satisfy the properties

array^[i] ≤ pivot	**low ≤ i < boundary**
array^[i] = pivot	**i = boundary**
array^[i] > pivot	**boundary < i ≤ high**

where **pivot** is an element chosen by **Partition**, and **boundary** is the index of that element.

In this implementation, the two recursive calls to **QuickSort** are entirely independent and could easily be performed in parallel. The most obvious strategy is simply to fork a new thread at each recursive subdivision. Coding this in Modula-2+ gives rise to the "fork always" implementation shown in Figure 2.

Before discussing this example in detail, a few notes on the presentation are required. This example introduces the primitives **Thread.Fork** and **Thread.Join** which provide the basic mechanism for concurrency in Modula-2+. **Thread.Fork**(*proc, arg*) creates a new thread of control executing *proc*(*arg*) and returns a handle of type **Thread.T** which can later be passed to **Thread.Join** to wait for completion of that thread. Since the Modula-2+ implementation of **Thread.Fork** allows only a single argument (and we will make use of the limitation in section 6), the individual arguments passed to **Quicksort** must be assembled into an argument block which can then be passed as a unit. Thus, in **ForkAlwaysQuicksort** and the examples which follow, we will make use of the type definition

```
TYPE
  ArgBlock = REF RECORD
    array: RefIntArray;
    low, high: INTEGER;
  END;
```

and assume the existence of a procedure **CreateArgBlock** which assembles a new block from its arguments. This change in the argument structure also explains the need for two procedures in this example. **Quicksort** itself is used only at the outer level and implements precisely the same abstraction as the **Quicksort** procedure in the serial example. **QSort** is a private procedure, suitable for concurrent invocation.

```
PROCEDURE Quicksort(array: RefIntArray; low, high: INTEGER);
BEGIN
   QSort(CreateArgBlock(array, low, high));
END Quicksort;

PROCEDURE QSort(args: ArgBlock);
VAR
   boundary: INTEGER;
   part1, part2: ArgBlock;
   child: Thread.T;
BEGIN
   WITH args^ DO
      IF high - low < MinQuick THEN
         SelectionSort(array, low, high);
      ELSE
         boundary := Partition(array, low, high);
         part1 := CreateArgBlock(array, low, boundary-1);
         part2 := CreateArgBlock(array, boundary+1, high);
         child := Thread.Fork(QSort, part2);
         QSort(part1);
         Thread.Join(child);
      END;
   END;
END QSort;
```

Figure 2
ForkAlwaysQuicksort

In **ForkAlwaysQuicksort**, **Thread.Fork** is used to create a new thread to execute one of the recursive calls to **QSort** while the original thread performs the other. **Thread.Join** is used to ensure that both operations are complete before the call on **QSort** returns. When this occurs, both threads have completed their work, and the array is sorted.

ForkAlwaysQuicksort is easy to code, but not particularly practical. Even though Modula-2+ threads are reused whenever possible during the execution of a program, **ForkAlwaysQuicksort** creates an excessive number of threads — far more than the number of processors. On a 10,000-element array, the resulting overhead is so severe that the program runs 5.9 times *more slowly* than **SerialQuicksort** on the Firefly [Thacker87], which served as the base for our experimentation. Most of the additional time is spent creating, forking, and joining threads.

3. Avoiding excess concurrency — the fork-when-idle strategy

One simple and effective way to avoid unproductive parallelism is to use a fork-when-idle strategy: at each division point, check to see if there are any idle processors; if so, perform the task in parallel, otherwise do it serially. Figure 3 illustrates a procedure **QSort** that implements this strategy.

In **ForkWhenIdleQuicksort**, the global variable **nIdle** is used to maintain a count of the number of idle processors. Since this variable may be referenced simultaneously by several independent threads, the lock **nIdleMutex** is required to control access to **nIdle**. After calling **Partition**, **QSort** checks if **nIdle** is greater than zero. If so, it performs the recursive calls to **QSort** in parallel as in **ForkAlwaysQuicksort**. Otherwise, it performs the calls serially, avoiding the overhead of calling **Fork** and **Join**.

```
PROCEDURE QSort(args: ArgBlock);
VAR
   boundary: INTEGER;
   part1, part2: ArgBlock;
   child: Thread.T;
   shouldfork: BOOLEAN;
BEGIN
   WITH args^ DO
      IF high - low < MinQuick THEN
         SelectionSort(array, low, high);
      ELSE
         LOCK nIdleMutex DO
            shouldfork := (nIdle > 0);
            IF shouldfork THEN nIdle := nIdle - 1; END;
         END;
         boundary := Partition(array, low, high);
         part1 := CreateArgBlock(array, low, boundary-1);
         part2 := CreateArgBlock(array, boundary+1, high);
         IF shouldfork THEN
            child := Thread.Fork(QSort, part2);
            QSort(part1);
            LOCK nIdleMutex DO nIdle := nIdle + 1; END;
            Thread.Join(child);
         ELSE
            QSort(part1);
            QSort(part2);
         END;
      END;
   END;
END QSort;
```

Figure 3
ForkWhenIdleQuicksort

The fork-when-idle strategy succeeds in avoiding almost all of the overhead of dividing a task for parallel execution. At each division point, the question "is a processor free now?" is posed to choose between the serial and parallel options. This test is too strict, however, since the decision is made on the basis of an instantaneous snapshot of the processor utilization. A better criterion is "will a processor become free while there is still a possibility of sharing responsibility for this task?". This is the test used in WorkCrews.

4. The basic WorkCrew strategy

In the WorkCrew paradigm, a set of worker threads cooperate to perform divisible tasks. When a worker has a task that can be divided into two possibly concurrent subtasks, it begins work on one of the subtasks and queues a help request for the other. When it finishes the first subtask, it checks to see if its help request was answered by another worker. If so, the original worker assumes that the operation is in good hands and returns. If not, the pending request is canceled, and the original worker completes the remainder of the task itself.

WorkCrews are created by calling **Create**(*n*) where *n* is the number of worker threads (usually equal to the number of processors). The **Create** procedure returns a handle for the WorkCrew which is then passed to the other WorkCrew primitives. New top-level tasks are added by calling **AddTask**(*crew, proc, data*) where *crew* is the WorkCrew handle returned by **Create**, *proc* is a procedure which should be run on the client's behalf, and *data* is a bundled data value to be passed to the procedure. **AddTask** eventually causes one of *crew*'s workers to execute *proc*(*data*). Calling **Join**(*crew*) suspends the caller until all of *crew*'s tasks have been completed.

A procedure performing a task may subdivide its work by calling **RequestHelp**(*proc, data*). **RequestHelp** is similar to **Thread.Fork**, except that the call to *proc*(*data*) is not performed unless and until a worker becomes idle. After queuing this subtask, the worker that called **RequestHelp** must eventually issue a corresponding call to **GotHelp**. If another worker has answered the help request in the interim, **GotHelp** returns **TRUE** and the original worker can move on to other tasks. If not, the call to **GotHelp** cancels the pending request, and the original worker must complete the remainder of the task itself. Calls to **RequestHelp** and **GotHelp** may be nested, but it is the responsibility of the client to ensure that each call to **RequestHelp** is properly paired with a corresponding **GotHelp**.

Figure 4 presents the WorkCrew version of **Quicksort**. After the array has been partitioned, **WorkCrewQuicksort** requests that the second recursive call to **QSort** be performed in parallel. It then calls **QSort** on the first half of the subdivided array, and, on returning, uses **GotHelp** to determine whether its earlier request for help was answered. If not, it calls **QSort** on the second half of the array.

A worker's requests are answered in the order received. Thus, as long as the subdivision follows a traditional divide-and-conquer strategy, WorkCrews will favor coarser grains of parallelism over finer ones. This improves performance by reducing the time spent dividing tasks. By contrast, the fork-when-idle strategy does not distinguish between coarse and fine parallelism in its steady state. As noted above, the fork-when-idle strategy may also miss opportunities for parallel execution when WorkCrews would not, since the former makes no allowance for the possibility that a processor might soon become free.

5. WorkCrews *vs.* procedural semantics

The client must exercise some degree of caution in using WorkCrews. In the absence of a call to **WorkCrew.Join**, procedures that are written to use the WorkCrew paradigm do not necessarily adhere to procedural semantics; when a recursive call on **QSort** returns in the **WorkCrewQuicksort** example, the caller cannot assume that all of the internal work is complete. The procedure guarantees only that the necessary operations to complete the call have been initiated, but this work may still be in progress when the original worker returns.

In many practical applications, this fits well with the structure of the problem. In Quicksort, for example, it is sufficient to fork worker tasks for the independent subtasks and then use **WorkCrew.Join** to ensure that all tasks are complete. In other cases, however, it is necessary to provide a more fine-grained mechanism for synchronizing the activity of the individual workers. In the WorkCrew interface, this is accomplished by using the procedure pair **EnterSubtaskGroup** and **JoinSubtaskGroup**. Like **RequestHelp** and **GotHelp**, these are paired operations, and clients must ensure proper bracketing.

The basic structure of these procedures is illustrated by the program fragment

```
WorkCrew.EnterSubtaskGroup();
P1();
WorkCrew.JoinSubtaskGroup();
P2();
```

In this example, suppose that procedure **P1** and **P2** must both be executed but that **P2** can only be started when **P1** is finished. If **P1** calls **RequestHelp**, it is not legitimate to assume that **P1** is complete

```
PROCEDURE Quicksort(array: RefIntArray; low, high: INTEGER);
VAR
   crew: WorkCrew.T;
BEGIN
   crew := WorkCrew.Create(nWorkers);
   WorkCrew.AddTask(crew, QSort, CreateArgBlock(array, low, high));
   WorkCrew.Join(crew);
END Quicksort;

PROCEDURE QSort(args: ArgBlock);
VAR
   boundary: INTEGER;
   part1, part2: ArgBlock;
BEGIN
   WITH args^ DO
      IF high - low < MinQuick THEN
         SelectionSort(array, low, high);
      ELSE
         boundary := Partition(array, low, high);
         part1 := CreateArgBlock(array, low, boundary-1);
         part2 := CreateArgBlock(array, boundary+1, high);
         WorkCrew.RequestHelp(QSort, part2);
         QSort(part1);
         IF NOT WorkCrew.GotHelp() THEN
            QSort(part2);
         END;
      END;
   END;
END QSort;
```

Figure 4
WorkCrewQuicksort

---

when the worker who started **P1** returns—other workers who came to help with task **P1** may not yet be complete.

When a worker calls **JoinSubtaskGroup**, it blocks until all **RequestHelp** operations issued since the corresponding **EnterSubtaskGroup** are complete. To maintain a constant number of active workers, the WorkCrew implementation automatically activates a new worker while the caller of **JoinSubtaskGroup** is blocked.

To demonstrate where the use of these primitives is required, consider once again the Quicksort example. The implementation from the last section misses an important opportunity for parallelism. In that implementation, the **Partition** function makes a sequential pass over the entire array before any opportunity for concurrency arises. If parallelism could also be used here, the total running time of the algorithm could be reduced further.

One strategy is to implement a new routine **PartitionSubset** which can be used to partition only the even or odd-numbered elements. If the same pivot value is chosen (55 in the example below), the result after both calls on **PartitionSubset** will be an array divided into three parts, as shown in Figure

5. To the left of position **A**, all elements are less than or equal to the pivot; to the right of **B**, all elements are larger.

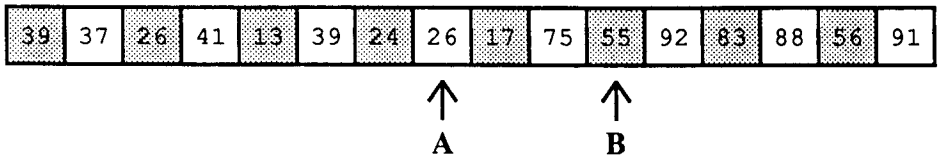

Figure 5
Odd/Even Partitioning

Between **A** and **B**, the elements may be jumbled, but this can be repaired by an additional step which partitions all elements in that range using the same pivot.

The advantage here is that the first two calls to **PartitionSubset** may be made in parallel, since the elements they touch are disjoint. If this parallelism is achieved using WorkCrews, however, the final call on **Partition** cannot be started until the first two are complete. This means that an **EnterSubtaskGroup/JoinSubtaskGroup** pair are required, as shown in the **ParallelPartitionQuicksort** example in Figure 6.

6. Deferring the cost of task decomposition

The Quicksort example illustrates the structure of the WorkCrew mechanism, but does not demonstrate one of the most important advantages of WorkCrews: the ability to defer the cost of splitting a task into component parts until that division is actually performed. The structure of the parallel Quicksort program is an example of "embarrassing parallelism." The only significant cost in the decomposition is the cost of the fork and join.

In most practical applications, splitting a task into concurrent subtasks requires some additional computation. Many applications that perform intermediate computation steps in parallel will need to serialize the output of those computations. This is an example of overhead which is present in the parallel decomposition but unnecessary in the traditional sequential coding.

When this sort of overhead exists, it is even more important to avoid splitting a task when there are not enough processors to carry out the computations concurrently. In such cases, dividing the task incurs not only the overhead of the fork and join, but also the cost of any operations required to manage the decomposition. The WorkCrew mechanism makes it rather easy to avoid these costs when there are not enough processors to warrant task division.

For example, suppose that **Subtask1** and **Subtask2** are two subtasks that could conceivably be executed in parallel. When the subtasks are executed serially, no additional work is required; if the task is divided, some additional overhead is incurred. We assume that the overhead cost of splitting the task can be encapsulated in a function **PrepareTheData** which updates the argument block to reflect the required additional processing. Using the traditional fork/join mechanism, this would be represented as:

```
child := Thread.Fork(Subtask2, PrepareTheData(args));
Subtask1(args);
Thread.Join(child);
```

In the WorkCrew model, the key observation is that **PrepareTheData** need only be called if the task is actually divided and not when the request for help is posted. To make this possible, **WorkCrew.RequestHelp** accepts an optional third argument which is a "data preparer." The data preparer is stored with the task request and called whenever a worker in the crew actually takes on the

```
TYPE
   SubsetType = (Odd, Even, All);

   PartitionBlock = REF RECORD
      array: RefIntArray;
      low, high: INTEGER;
      pivot: INTEGER;
      type: SubsetType;
      result: INTEGER;
   END;

PROCEDURE Partition(array: RefIntArray; low, high: INTEGER) : INTEGER;
VAR
   pivot, lowp, highp: INTEGER;
   odds, evens, combined: PartitionBlock;
BEGIN
   pivot := ChoosePivot(array, low, high);
   odds := CreatePartitionBlock(array, low, high, pivot, Odd);
   evens := CreatePartitionBlock(array, low, high, pivot, Even);
   WorkCrew.EnterSubtaskGroup();
      WorkCrew.RequestHelp(PartitionSubset, evens);
      PartitionSubset(odds);
      IF NOT WorkCrew.GotHelp() THEN
         PartitionSubset(evens);
      END;
   WorkCrew.JoinSubtaskGroup();
   lowp := MIN(odds^.result, evens^.result);
   highp := MAX(odds^.result, evens^.result);
   combined := CreatePartitionBlock(array, lowp, highp, pivot, All);
   PartitionSubset(combined);
   RETURN combined^.result;
END Partition;
```

Figure 6
ParallelPartitionQuicksort

---

subtask. Thus, in the WorkCrew case, the example above would be coded as

```
WorkCrew.RequestHelp(Subtask2, args, PrepareTheData);
Subtask1(args);
IF NOT WorkCrew.GotHelp() THEN
   Subtask2(args);
END;
```

Note that **PrepareTheData** is called only if another worker actually takes over the subtask operation. If the original worker manages to complete **Subtask1** before any helper arrives to handle the request for help with **Subtask2**, this degenerates into the purely sequential case and the additional processing is not required. In effect, the data preparer is applied "lazily" so that it is called only when needed.

To make this aspect of WorkCrews more concrete, consider the problem of implementing a file searching program (like Unix **grep**) that can process several independent files in parallel. As a primitive, assume that you have a procedure **Grep** which takes three arguments: the pattern string, the name of a single file on which to operate, and a stream on which to write the output (such streams are called "writers" in the Modula-2+ environment and have type **Wr.T**). Thus, to find and print on standard output all lines containing "xyzzy" in the file **adventure.txt**, we could call

```
Grep("xyzzy", "adventure.txt", stdout);
```

Unfortunately, concurrent execution of this operation on two different files at the same time would be inappropriate, since the output would be hopelessly interleaved. Instead, we need some mechanism to serialize the output stream.

Fortunately, such a mechanism (called "splitwriters") exists in the Modula-2+ library. Given a splitwriter **w1**, the statement

```
w2 := SplitWriter.Split(w1);
```

yields a second splitwriter **w2** so that everything written to **w1** will precede in the eventual output stream everything written to **w2**. Internally, any writes to **w2** are buffered until **w1** is closed and then dumped on the output stream. Either descendent of the **SplitWriter.Split** operation can be split arbitrarily often.

This makes it possible to code a procedure **MultiGrep** which calls the **Grep** procedure on a list of files. Splitting a writer has an associated overhead cost, however, and we would like to avoid this whenever possible. The WorkCrew structure makes this quite convenient since the **SplitWriter** operation can be included in the data preparer function and thus be invoked only when the task is actually split.

For concreteness, assume that **MultiGrep** takes an argument block of the following form:

```
TYPE
  TaskBlock = REF RECORD
    files: Text.RefArray;
    low, high: INTEGER;
    pattern: Text.T;
    outfile: Wr.T;
  END;
```

The **MultiGrep** program itself is shown in Figure 7.

7. Implementation

Our implementation of WorkCrews is based on the **Thread** module of Modula-2+. In section 2, we explained the **Thread** abstraction and made use of the operations **Fork** and **Join**. The **Thread** module also provides abstractions for locks and conditions. Locks have the operations **Acquire** and **Release**, which are automatically generated by the compiler when the **LOCK** statement is used. The principal operations of conditions are **Signal** and **Wait**. A thread calls **Wait** to block until a condition occurs. **Signal** is called to indicate that a condition has occurred. In designing our implementation of WorkCrews, we also exploited several properties of the **Thread** module. In the absence of contention, acquiring and releasing a lock costs a total of only five instructions. Signaling a condition on which no thread is waiting costs only two instructions. In other cases, operations on locks and conditions require system calls.

The description of our WorkCrew implementation is divided into two sections. First, we describe a basic implementation without support for subtask groups. Later, we describe the extensions required for managing subtask groups.

```
PROCEDURE MultiGrep(args: TaskBlock);
VAR
   part1, part2: TaskBlock;
   midpoint: INTEGER;
BEGIN
   WITH args^ DO
      IF low > high THEN RETURN END;
      IF low = high THEN
         Grep(pattern, filelist^[low], outfile);
      ELSE
         midpoint := (low + high) DIV 2;
         part1 := args;
         part2 := CopyTaskBlock(part1);
         part1^.high := midpoint;
         part2^.low := midpoint + 1;
         WorkCrew.RequestHelp(MultiGrep, part2, SplitTheWriter);
         MultiGrep(part1);
         IF NOT WorkCrew.GotHelp() THEN
            MultiGrep(part2);
         END;
      END;
   END;
END MultiGrep;

PROCEDURE SplitTheWriter(args: TaskBlock) : TaskBlock;
BEGIN
   args^.outfile := SplitWriter.Split(args^.outfile);
   RETURN args;
END SplitTheWriter;
```

Figure 7
MultiGrep

7.1 Basic implementation

WorkCrews are implemented using two principal data types: **Crew** and **Worker**. A **Crew** is a tuple including:

workers	the set of all Workers in this Crew
taskQueue	a queue of unstarted tasks created by **AddTask**
noMoreTasks	a flag indicating whether **Join** has been called
nForked	the number of workers created
nRetired	the number of workers that have exited
nIdle	the number of workers waiting for tasks
allDone	the condition to denote completion of all tasks
wakeWorker	the condition to denote that a new task exists or that all tasks have been completed
crewMutex	a lock to synchronize access to the above

A **Worker** is a tuple including:

requests	a stack of help requests
sp	the top of stack pointer for **requests**
helpedPtr	a pointer into **requests**
workerMutex	a lock to synchronize access to the above
crew	a backpointer to the shared **Crew** object

An important invariant in the implementation is that **helpedPtr** is between **sp** and the base of **requests**, inclusive. All help requests between the base and **helpedPtr** have been answered, and all others have not.

Most of the WorkCrew operations have straightforward implementations. **Create(*n*)** initializes a new **Crew** and *n* new **Workers**, forking a new thread for each worker. **AddTask** simply enqueues a task in **taskQueue** while holding the **crewMutex** lock and wakes up an idle worker, if any exist. **Join** sets the flag **noMoreTasks** to be **TRUE** and waits for the condition **allDone**.

Each worker thread executes the internal procedure **WorkerRoot** shown in Figure 8. **WorkerRoot** is the key to understanding how a worker finds a task and, when none exist, how a worker determines whether to exit or wait for new tasks to appear.

```
PROCEDURE WorkerRoot(me: Worker)
VAR
   task: Task;
   crew: Crew;
BEGIN
   crew := me^.crew;
   LOOP
      IF FindTask(me, task) THEN
         DoTask(me, task);
         IF CompletedSubtaskGroup(task) THEN
            WakeJoinSG(task, crew);
            EXIT;
         END;
      ELSE
         LOCK crew^.crewMutex DO
            IF AllTasksDone(crew) THEN
               Terminate(crew);
               EXIT;
            ELSE Block(crew)
            END;
         END;
      END;
   END;
END WorkerRoot;
```

Figure 8
WorkerRoot

FindTask first checks the **taskQueue** for unstarted tasks. If the queue is empty, then it searches for a **Worker** with unanswered help requests. Testing for unanswered requests is fast since it requires only a pointer comparison of **sp** and **helpedPtr**. **FindTask** extracts the first unanswered request it finds, adjusts **helpedPtr** to mark the request as answered, and returns **TRUE**. If no unanswered request is found, it returns **FALSE**.

DoTask performs the task just found by **FindTask**. Note that this may involve calling a data preparer if the task is from a help request. The call to **CompletedSubtaskGroup** is related to subtask management, which will be described shortly. In the absence of subtask groups, the call always returns **FALSE** and the worker simply goes to the beginning of the loop.

When **FindTask** returns **FALSE**, **WorkerRoot** checks for termination by calling **AllTasksDone**. The criteria for termination are that **Join** has been called (**noMoreTasks** is **TRUE**), and that all workers other than the one executing **AllTasksDone** are blocked (**nRetired + nIdle = nForked** – 1). By calling **Terminate**, the first worker to detect termination wakes all other blocked workers, which then exit in succession. **Terminate** also signals **allDone** to wake the thread that called **Join**.

If **AllTasksDone** returns **FALSE**, there is a possibility that new tasks will be created. Therefore, **WorkerRoot** calls **Block** to mark the worker as idle and suspend execution until either a new task is created or termination is detected (condition **wakeWorker**).

We now turn to the implementations of **RequestHelp** and **GotHelp**. These operations must be fast since they are called frequently. **RequestHelp** simply pushes its arguments onto the worker's **requests** stack, while holding **workerMutex**, and then calls **Signal** to wake an idle worker, if any exist. Typically, the calls to **Acquire**, **Release**, and **Signal** will be the efficient ones described at the start of this section: there is little contention for **workerMutex** since each worker has its own, and a worker only briefly acquires another worker's **workerMutex** when searching for a task to perform. Also, if we assume that there is an excess of parallelism, workers will seldom block waiting for tasks, so the call to **Signal** in **RequestHelp** is efficient.

The **GotHelp** operation is similarly fast. It acquires **workerMutex** and pops the **requests** stack. If **helpedPtr** ≥ **sp**, then the request was answered, and **GotHelp** sets **helpedPtr** to **sp** to maintain the invariant on the stack of requests. Finally, it releases **workerMutex**. As in **RequestHelp**, the lock operations are typically fast since contention for each **workerMutex** is low.

7.2 Managing subtask groups

Given the implementation of WorkCrews described thus far, only a few extensions are required to support subtask groups. A subtask group is simply a set of tasks created by **RequestHelp** bracketed by calls to **EnterSubtaskGroup** and **JoinSubtaskGroup**. When a call to **RequestHelp** is bracketed by more than one enter/join pair, the task it creates belongs to the innermost enter/join pair.

The purpose of **JoinSubtaskGroup** is to suspend its caller until *all* tasks created since the corresponding call to **EnterSubtaskGroup** are completed. Because the subtask group operations must follow stack discipline, we observe that tasks in nested subtask groups will be completed when **JoinSubtaskGroup** is called.

Our strategy for managing subtask groups is to maintain a count of the number of workers performing tasks in each subtask group. A worker increments the count when it begins a task in the group, and decrements the count when it finishes the task. **JoinSubtaskGroup** blocks until the count is zero, and the last worker to finish a task in the group is responsible for waking the join, if necessary.

We introduce a new type, **SubtaskGroup**, which is a tuple:

count	the number of workers cooperating on this subtask group
groupMutex	a lock to synchronize access to count
subtaskDone	a condition used to detect completion of a subtask group
prev	a pointer to the smallest enclosing **SubtaskGroup** (**NIL** if none)

In addition, we augment the **Worker** type to include the component:

groupStack	the most deeply nested **SubtaskGroup** for which worker is executing (**NIL** if there is none).

Semantically, **groupStack** behaves like a stack to the particular worker (entries are chained using the **prev** component). When viewed together, the worker **groupStacks** form an inverted tree.

The operation **EnterSubtaskGroup** pushes a new **SubtaskGroup** onto **groupStack**. The new **SubtaskGroup** has a count of one to indicate that a single worker (the one executing **EnterSubtaskGroup**) is computing the subtask.

JoinSubtaskGroup decrements the count in the top of the **groupStack**. If the result is zero, then all workers that ever cooperated in performing the subtask have finished, so **JoinSubtaskGroup** pops the **groupStack** and returns. Otherwise, it blocks until the count is zero by waiting for the condition **subtaskDone**. Since the number of active workers should remain constant, a new worker thread is created before blocking. Once awakened, **JoinSubtaskGroup** pops **groupStack**.

A minor extension to **RequestHelp** is necessary to maintain the association of each task with its correct **SubtaskGroup**. Each entry in the **requests** stack is augmented to include a pointer to its associated **SubtaskGroup**. **RequestHelp** initializes this pointer to be the worker's **groupStack** at the time of the call. Note that this is consistent with our earlier definition that a help request's subtask group is the one created by the last **EnterSubtaskGroup** call prior to that request.

The final extensions involve the internal procedures for selecting and performing tasks. When **FindTask** chooses to answer a help request, it initializes **task**'s **SubtaskGroup** to be the one noted in the help request. **DoTask** sets the worker's **groupStack** to refer to the same **SubtaskGroup** and increments the count therein to reflect the activity of the worker answering the request.

The corresponding decrement of the count occurs when **WorkerRoot** calls **CompletedSubtaskGroup** (see Figure 8). If the **SubtaskGroup** is **NIL**, then **CompletedSubtaskGroup** immediately returns; otherwise, it decrements the count. If the result is zero, all tasks in the subtask group are completed, so it returns **TRUE**; otherwise it returns **FALSE**.

If the subtask group was completed, the **WorkerRoot** calls **WakeJoinSG** to wake the worker blocked in **JoinSubtaskGroup** and exits. The effect is to keep the number of active workers constant, since **JoinSubtaskGroup** had created a new worker before blocking.

As a final note, the implementation avoids deadlock because no worker ever acquires more than one of each kind of lock, and multiple locks are always acquired in the order **crewMutex**, **workerMutex**, **groupMutex**.

8. Performance

To assess the success of the WorkCrew mechanism, we measured the performance of the examples presented above on the Firefly [Thacker87]. The Firefly is a shared-memory multiprocessor workstation, designed and built at the Digital Equipment Corporation's Systems Research Center in Palo Alto.

The timings for the five versions of Quicksort are presented in Table I below. In each case, the experiment was run repeatedly on an otherwise idle Firefly configured with five MicroVAX-II processors. The average for ten trials is reported, along with the standard deviation. The speedup is calculated using **SerialQuicksort** as a baseline. The **ForkAlways** strategy occurs in the table only for 10K elements, since anything larger turned out to be infeasible with this strategy.

As Table I shows, the **WorkCrew** implementation of Quicksort has a considerable performance advantage over **ForkWhenIdle** at each of the problem sizes tested. The fact that the speedup is not closer to

strategy	N	mean time (sec)	standard deviation	speedup serial = 1
Serial	10K	3.7	0.036	1.00
ForkAlways	10K	21.8	0.536	0.17
ForkWhenIdle	10K	1.6	0.084	2.31
WorkCrew	10K	1.3	0.017	2.85
ParallelPartition	10K	1.2	0.056	3.08
Serial	100K	46.4	0.129	1.00
ForkWhenIdle	100K	16.3	0.273	2.85
WorkCrew	100K	14.4	0.143	3.22
ParallelPartition	100K	12.9	0.311	3.60
Serial	1M	560.3	2.444	1.00
ForkWhenIdle	1M	183.1	4.427	3.06
WorkCrew	1M	157.7	0.773	3.55
ParallelPartition	1M	147.6	3.312	3.80

Table I
Quicksort Performance

---

the number of processors is due in part to the fact that the algorithm includes a long serial partition step during which no parallelism is available. In addition, the basic WorkCrew mechanism and the scheduler both introduce some overhead which reduces the speedup below the theoretical limit.

To evaluate the performance advantage that we can achieve through lazy evaluation of the task decomposition overhead, we built two WorkCrew-based implementations of **MultiGrep**. One implementation was the one given in section 6, in which the call to **SplitWriter** occurs in a separate data preparer routine so that this cost is paid only when concurrent evaluation actually occurs. The other was an identical implementation in which the **SplitWriter** call appears directly within the body of the **MultiGrep** procedure. We used these implementations to search for the common string "**INTEGER**" in a library directory containing 229 files.

In our first experiment, the implementation, which used lazy evaluation to avoid unnecessary **SplitWriter** calls, provided a 2% performance advantage. This seemed disappointingly low until we recognized that **MultiGrep** was disk-limited to such an extent that the entire process was essentially serialized, reducing any advantage due to parallelism. When we repeated the experiment starting with the files in memory, the increase jumped to 13%, illustrating that splitting the writer represented a significant fraction of the total cost.

Note that the strategy used in the subdivision has considerable influence on the savings that can be achieved. As written, the **MultiGrep** example from section 6 divides the list of files in half and issues a help request for the second portion. By keeping both subtasks relatively large, this strategy increases the chance that each subtask can process several files sequentially without incurring a split. If, on the other hand, **MultiGrep** had been coded to split the task at the next file and request help for the entire remainder of the list, workers would tend to leapfrog forward through that list, incurring the overhead of the splitwriter for almost every file. When we ran this experiment, the divide-in-half approach ended up requiring only 12 split operations when scanning the 229 files, while the split-at-next-file approach required 222 splits.

In the **MultiGrep** example, of course, we expected that the mechanics of serializing the output would represent a significant fraction of the overhead, since relatively little work is done in the subtasks themselves: definitions files are individually short and the search algorithm is reasonably efficient. In other applications, the balance between the actual work and the overhead due to task decomposition will be different, and it is hard to predict to what extent this technique will reduce the total time. Based on our experience, however, we believe that the use of lazy evaluation to reduce decompositional overhead will provide significant efficiency improvements in a variety of practical situations.

9. Conclusions

In developing the parallel C compiler [Vandevoorde88] and other applications here at SRC, we have found the WorkCrew concept to be a useful mechanism for the efficient subdivision of tasks. Unlike traditional mechanisms for expressing parallelism, the WorkCrew strategy makes a dynamic decision about the availability of processing resources so that fewer opportunities for parallel decomposition are lost. Moreover, the WorkCrew mechanism makes it possible to reduce the overhead associated with task decomposition by deferring this cost until the decomposition actually occurs.

10. References

1. Jon Bentley. "Programming Pearls: How to Sort," *Communications of the ACM*. Vol. 27, no. 4, April 1984.
2. Andrew D. Birrell, John V. Guttag, James J. Horning and Roy Levin. "Synchronization Primitives for a Multiprocessor," *Proceedings of the the Symposium on Operating Systems Principles, RAustin, Texas*. November 1987.
3. Andrew Birrell. "An Introduction to Programming with Threads," Research Report #35, Palo Alto, California: Digital Equipment Corporation, Systems Research Center, January 6, 1989.
4. Edward F. Gehringer, Daniel P. Siewiorek and Zary Segall. *Parallel Processing: The Cm* Experience*. Bedford, MA: Digital Press, 1987.
5. Robert Halstead. "Multilisp: a language for concurrent symbolic computation," *ACM Transcations on Programming Languages and Systems*. Vol. 7, no. 4, October 1985.
6. Peter Hibbard, Andy Hisgen and Tom Rodeheffer. "A Language Implementation Design for a Multiprocessor Computer System," *Proceedings of the Fifth Annual Symposium on Computer Architecture*. April 1978.
7. C. A. R. Hoare. "Quicksort," *Computer Journal*. Vol. 5, 1962.
8. Peter Møller-Nielsen and Jørgen Staunstrup. "Problem-heap: a paradigm for multiprocessor algorithms," DAIMI PB-200, Aarhus University, October 1985.
9. Peter N. Oleinick and Samuel H. Fuller. "The Implementation and Evaluation of a Parallel Algorithm on C.mmp," Computer Science Department Report CMU-CS-78-125, Carnegie-Mellon University, June 1978.
10. Severo Ornstein, William R. Crowther, Michael F. Kraley, Robert D. Bressler, Anthony Michel and Frank Heart. "Pluribus: A Reliable Multiprocessor," *AFIPS Conference Proceedings*. Vol. 44, May 1975.
11. Paul Rovner, Roy Levin and John Wick. "On Extending Modula-2 for Building Large, Integrated Systems," Research Report #3, Palo Alto, California: Digital Equipment Corporation, Systems Research Center, January 1985.
12. Robert Sedgewick. "Implementing Quicksort Programs," *Communications of the ACM*. Vol. 21, no. 10, October 1978.
13. Charles Thacker and Larry Stewart. "Firefly: A Multiprocessor Workstation," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS II), Palo Alto, CA. October 1987.
14. Mark Vandevoorde. "Parallel Compilation on a Tightly-Coupled Multiprocessor," Research Report #26, Palo Alto, California: Digital Equipment Corporation, Systems Research Center, March 1, 1988.
15. Niklaus Wirth. *Programming in Modula-2*. New York: Springer-Verlag, 1985.

SRC Reports

"A Kernel Language for Modules and Abstract Data Types."
R. Burstall and B. Lampson.
Research Report 1, September 1, 1984.

"Optimal Point Location in a Monotone Subdivision."
Herbert Edelsbrunner, Leo J. Guibas, and Jorge Stolfi.
Research Report 2, October 25, 1984.

"On Extending Modula-2 for Building Large, Integrated Systems."
Paul Rovner, Roy Levin, John Wick.
Research Report 3, January 11, 1985.

"Eliminating **go to**'s while Preserving Program Structure."
Lyle Ramshaw.
Research Report 4, July 15, 1985.

"Larch in Five Easy Pieces."
J. V. Guttag, J. J. Horning, and J. M. Wing.
Research Report 5, July 24, 1985.

"A Caching File System for a Programmer's Workstation."
Michael D. Schroeder, David K. Gifford, and Roger M. Needham.
Research Report 6, October 19, 1985.

"A Fast Mutual Exclusion Algorithm."
Leslie Lamport.
Research Report 7, November 14, 1985.

"On Interprocess Communication."
Leslie Lamport.
Research Report 8, December 25, 1985.

"Topologically Sweeping an Arrangement."
Herbert Edelsbrunner and Leonidas J. Guibas.
Research Report 9, April 1, 1986.

"A Polymorphic λ-calculus with Type:Type."
Luca Cardelli.
Research Report 10, May 1, 1986.

"Control Predicates Are Better Than Dummy Variables For Reasoning About Program Control."
Leslie Lamport.
Research Report 11, May 5, 1986.

"Fractional Cascading."
Bernard Chazelle and Leonidas J. Guibas.
Research Report 12, June 23, 1986.

"Retiming Synchronous Circuitry."
Charles E. Leiserson and James B. Saxe.
Research Report 13, August 20, 1986.

"An $O(n^2)$ Shortest Path Algorithm for a Non-Rotating Convex Body."
John Hershberger and Leonidas J. Guibas.
Research Report 14, November 27, 1986.

"A Simple Approach to Specifying Concurrent Systems."
Leslie Lamport.
Research Report 15, December 25, 1986. Revised January 26, 1988

"A Generalization of Dijkstra's Calculus."
Greg Nelson.
Research Report 16, April 2, 1987.

"*win* and *sin*: Predicate Transformers for Concurrency."
Leslie Lamport.
Research Report 17, May 1, 1987. Revised September 16, 1988.

"Synchronizing Time Servers."
Leslie Lamport.
Research Report 18, June 1, 1987.

"Blossoming: A Connect-the-Dots Approach to Splines."
Lyle Ramshaw.
Research Report 19, June 21, 1987.

"Synchronization Primitives for a Multiprocessor: A Formal Specification."
A. D. Birrell, J. V. Guttag, J. J. Horning, R. Levin.
Research Report 20, August 20, 1987.

"Evolving the UNIX System Interface to Support Multithreaded Programs."
Paul R. McJones and Garret F. Swart.
Research Report 21, September 28, 1987.

"Building User Interfaces by Direct Manipulation."
Luca Cardelli.
Research Report 22, October 2, 1987.

"Firefly: A Multiprocessor Workstation."
C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr.
Research Report 23, December 30, 1987.

"A Simple and Efficient Implementation for Small Databases."
Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber.
Research Report 24, January 30, 1988.

"Real-time Concurrent Collection on Stock Multiprocessors."
John R. Ellis, Kai Li, and Andrew W. Appel.
Research Report 25, February 14, 1988.

"Parallel Compilation on a Tightly Coupled Multiprocessor."
Mark Thierry Vandevoorde.
Research Report 26, March 1, 1988.

"Concurrent Reading and Writing of Clocks."
Leslie Lamport.
Research Report 27, April 1, 1988.

"A Theorem on Atomicity in Distributed Algorithms."
Leslie Lamport.
Research Report 28, May 1, 1988.

"The Existence of Refinement Mappings."
Martín Abadi and Leslie Lamport.
Research Report 29, August 14, 1988.

"The Power of Temporal Proofs."
Martín Abadi.
Research Report 30, August 15, 1988.

"Modula-3 Report."
Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson.
Research Report 31, August 25, 1988.

"Bounds on the Cover Time."
Andrei Broder and Anna Karlin.
Research Report 32, October 15, 1988.

"A Two-view Document Editor with User-definable Document Structure."
Kenneth Brooks.
Research Report 33, November 1, 1988.

"Blossoms are Polar Forms."
Lyle Ramshaw.
Research Report 34, January 2, 1989.

"An Introduction to Programming with Threads."
Andrew Birrell.
Research Report 35, January 6, 1989.

"Primitives for Computational Geometry."
Jorge Stolfi.
Research Report 36, January 27, 1989.

"Ruler, Compass, and Computer: The Design and Analysis of Geometric Algorithms."
Leonidas J. Guibas and Jorge Stolfi.
Research Report 37, February 14, 1989.

"Can fair choice be added to Dijkstra's calculus?"
Manfred Broy and Greg Nelson.
Research Report 38, February 16, 1989.

"A Logic of Authentication."
Michael Burrows, Martín Abadi, and Roger Needham.
Research Report 39, February 28, 1989.

"Implementing Exceptions in C."
Eric S. Roberts.
Research Report 40, March 21, 1989.

"Evaluating the Performance of Software Cache Coherence."
Susan Owicki and Anant Agarwal.
Research Report 41, March 31, 1989.

Controlling Parallelism

by Eric S. Roberts and Mark T. Vandevoorde

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301