01 Jan 1985

A Parallel Array Scanning Algorithm

Ralph M. Butler

Ralph W. Wilkerson
Missouri University of Science and Technology, ralphw@mst.edu

A Parallel Array Scanning Algorithm

Ralph M. Butler
and
Ralph W. Wilkerson

CSc-85-5

Department of Computer Science
University of North Florida
Jacksonville, Florida

Department of Computer Science
University of Missouri-Rolla
Rolla, Missouri

A Parallel Array Scanning Algorithm

Ralph M. Butler

Department of Computer Science
University of North Florida
Jacksonville, Florida

Ralph W. Wilkerson

Department of Computer Science
University of Missouri - Rolla
Rolla, Missouri

1. Introduction

Suppose we are given a vector X of n real numbers and we want to find the maximum sum found in any contiguous subvector of X. In Jon Bentley's article [1] on algorithm design and technique, a simple vector scanning problem and a series of progressively more efficient algorithms to solve this problem were discussed in some detail. Clearly, any algorithm must visit each location of X at least once and consequently a lower bound on the running time for problem is O(n), which is in fact attainable as Bentley's paper illustrates. However, the original motivation for this problem was the analagous two dimensional problem for an n x n array. That is, find the maximum sum contained in any contiguous rectangular subarray. Currently, the fastest algorithm obtained for this problem is $O(n^3)$[2] ; the theoretical lower bound would be at least $O(n^2)$. In this note, we will present a parallel processing approach to this problem which results in excess of one order of magnitude speed up for large problems in the $O(n^3)$ algorithm.

2. An $O(n^3)$ sequential algorithm for the two dimensional problem

Let us begin by briefly recounting the procedure used in the one dimensional case since it is the basis for the $O(n^3)$ algorithm used in the two dimensional case. Let X[1...n] be a vector of n real values and start scanning X at X[1] and scan to the right keeping the maximum sum encountered in a contiguous subvector in the variable *maxsofar*. Let us look at the situation inductively by supposing that if the maximum sum has been found in X[1...i-1] for i > 1, then we can extend the solution to X[1...i] by making the following observation. The maximum sum found in X[1...i] is either the maximum found in the first i-1 positions or it is the maximum found in the subvector that ends in position i. The variable *maxtohere* will contain the sum of the values in the subvector that ends in position i-1. Thus we increase *maxtohere* by X[i] as long as this sum remains positive else we reset the value of *maxtohere* to zero. Finally, *maxtohere* is compared to *maxsofar* in order to find a possibly larger maximum sum. The O(n) algorithm becomes:

```
maxsofar = 0;
for i=1 to n
{
    maxtohere = max(0,maxtohere + X[i]);
    maxsofar = max(maxsofar,maxtohere);
}
```

For the two dimensional problem, suppose we are given an array A[1...n,1...n] of n^2 real values. We construct an array B[1...n,0...n] such that B[i,0] = 0 and B[i,j] = B[i,j-1] + A[i,j] for j > 0. Observe that B[i,j] is nothing more that the sum of the first j entries of the i-th row of A for j > 0, or more precisely $B[i,j] = \sum_{k=1}^{j} A[i,k]$ for j > 0. Now, for $1 \leq t \leq s \leq n$ consider the difference $D_i(t,s) = B[i,s] - B[i,t-1] = \sum_{k=t}^{s} A[i,k]$ = sum of the t^{th} through the s^{th} entries of the i^{th} row of A.

Recall that any rectangular subarray of A must span adjacent columns of A and hence consecutive values of $D_i(t,s)$ for some fixed values of s and t. The application of the linear time maximum sum scanning algorithm to the two dimensional problem is now straightforward. For a fixed choice of the values s and t, $1 \leq t \leq s \leq n$, consider the values $D_1(t,s)$, ... , $D_n(t,s)$, in that order. By applying the linear time algorithm to these values, we can determine the maximum sum rectangular subarray between the t and s columns of A. Hence, by repeating this process for all choices of s and t with $1 \leq t \leq s \leq n$, one can determine the rectangular subarray of A with maximum sum. The O(n^3) algorithm is given below.

```
for i = 1 to n
{
    B[i,0] = 0;
    for j = 1 to n
    {
        B[j,i] = A[j,i] + B[j,i-1];
    }
}
maxsum = 0;
for i = 1 to n
{
    for j = 1 to n
    {
        maxsofar = 0;
        maxtohere = 0;
        for k = 1 to n
        {
            maxtohere = max(0,maxtohere + B[k,j] - B[k,i-1]);
            maxsofar = max(maxsofar,maxtohere);
        }
        maxsum = max(maxsum,maxsofar);
    }
}
```

3. Parallel Environments

During the development of a parallel algorithm an attempt should be made to remain independent of:

(1) any particular machine's architecture, and

(2) the number of processes devoted to the problem solution.

In particular, the parallel algorithm should execute correctly on a machine that supports only one process, as well as on a machine that supports several processes. This approach has a distinct advantage in that it supports porting the code to a wide variety of machines. A nice side-effect is that because of the independence of the number of processes devoted to the problem the algorithm can be tested on a sequential machine before uploading it to a multiprocessor.

For example, initial testing of the code for this algorithm was done on a Vax 11/780. For parallel testing, the code was ported to a Denelcor HEP at first, and then later to a Sequent Balance 8000 and an Encore Multimax.

Lusk and Overbeek [4] have discussed the portability issues at some length. They have developed a set of macros that support portability by hiding machine-dependent details from the programmer. The macros provide monitors as the synchronization mechanism. They allow the programmer to think in terms of high-level monitor operations and to ignore the low-level details of a particular machine. Because monitors play such an important role in the design of this algorithm, a discussion of them follows.

4. Monitors

A monitor is an abstract concept consisting of three parts:

(1) a shared resource, or a data structure representing the resource,

(2) the code to initialize the shared stuctures, and

(3) the code which performs the critical section operations on the resource.

The operations of a monitor may be called by any process at any time. It is necessary, however that only one process be permitted to enter the monitor at one time. In other words, from a process's point of view, the monitor is a serially reusable resource. This does not imply that the invoking processes are completely serialized; they are merely serialized through their critical sections in which they access a shared resource through the monitor. Permission to enter the monitor is typically gained through the use of some locking mechanism, e.g. a test-and-set primitive. This is the portion of code that is usually machine dependent and is best hidden in macros.

It is convenient to think of a monitor as an enclosure protecting some item or group of items. The items must be protected because they are shared among processes, and only one process at a time should be allowed to use them. In applications such as the array scan, the items being protected are a group of subproblems that must be solved. One reason that only one process at a time may access them, is that more than one process might access the same subproblem and attempt to solve it.

5. The Parallel Algorithm

5.1. Overview

At the start of the program, an integer (n) is read indicating how many total processes are to be used to solve the problem. Then the array defining the problem is read in. Next, the problem is broken into a set of subproblems that may be solved individually by parallel processes. The subproblems are placed in a monitor for protection to ensure that the processes will each obtain a unique subproblem to solve. Then, n-1 parallel processes are spawned. There are n-1 of them because the n^{th} process is the mainline which will also work on a subproblem.

Upon being spawned, each process goes to the monitor and attempts to get a problem to solve. At first, the pool of problems is empty, and the processes are forced to wait. When the mainline has placed all subproblems in the pool, and completed other initialization functions, it marks the pool as available. This releases the processes allowing them to enter the monitor, obtain a subproblem, and leave to solve it. After marking the pool as full, thus starting the parallel processes, the mainline attempts to enter the monitor and get a subproblem to solve.

Each process solves the subproblem it retrieved and then returns to the monitor for more work to do. Eventually, the entire problem is solved. The mainline can detect this by going to the monitor for work to do and being notified that there is no more. It then notifies the other processes and they all terminate.

At this point, the program is coded such that it solves one array problem and then ends. If it were to read in multiple arrays for solution however, it would *not* restart the parallel processes every time. Instead, it would merely let the processes wait at the monitor while the mainline read in a new problem, broke it into subproblems, and added them to the pool. The mainline would then mark the pool as full again so that the processes could all begin executing again. This approach is preferable to re-spawning the processes after each problem, because spawning a process is an expensive operation on some machines.

5.2. Partitioning into Subproblems

Partitioning a problem into subproblems is an interesting issue. On the one hand, if the subproblems are too small, the expense of creating and managing separate processes to handle them becomes too high. On the other hand, if some of the problems are too large, one process may finish its work and be forced to wait while another continues on a very large problem that should have been broken down. The addition of two integers is usually too small to deserve a totally separate process. For this algorithm, the solution of the entire array would typically be too large a problem.

The partitioning scheme which we have chosen is to allow each process to apply the linear time algorithm to each D_i(s,t) described above in section 2 on the sequential algorithm. As described in that section, each subproblem may be represented by an integer pair (s,t), where s and t are used as subscripts into the array under examination. It is these integer pairs that are manipulated by the monitor, not the rows and columns of the array. Indeed, the monitor actually protects a pair of variables named s and t, incrementing them each time it needs a new integer pair. It also checks for incrementing beyond the array's boundaries, indicating end of problem. Protecting the variables in the monitor ensures that a process gets a unique pair of values (and thus a unique subproblem) each time it enters the monitor.

The pair of integers is initialized to s = 1, t = 0 before a problem is begun. Then, the critical section code of the monitor alters the value(s) each time a process requests the next pair of values. The critical section is invoked by a macro invocation such as: GETPROB(i,j,n) where i is a variable local to the invoking process which will take on the next value of s, j takes on the next value of t, and n is the dimension of the array being processed.

The GETPROB macro representing the critical section is coded as:

```
if (s > 0)
{
    t = t + 1;
    if (t > $3)
    {
        s = s + 1;
        t = s;
    }
    if (s <= $3)
    {
        $1 = s;
        $2 = t;
    }
}
```

where the $n variables are macro variables, e.g. $1 in the macro corresponds to i in the GETPROB(i,j,n) invocation.

5.3. The Parallel Processes

As mentioned above, it is desirable to maintain independence of the number of processes devoted to the problem. Thus, if there is only one process available, it must be capable of solving every subproblem by itself. This means that the process should obtain a subproblem, solve it, and return to the monitor for more work to do until there is no more. Pseudo-code for such a process is as follows:

```
while (more subproblems to solve)
{
    GETPROB(i,j,n);
    process subproblem (i,j);
    if (this subproblem gives best result so far)
    {
        LOCK;   /* lock out other processes */
        best_so_far = this calculated value;
        UNLOCK;
    }
}
```

Coding the process in this manner permits a single copy of it to handle every subproblem if necessary. Thus, using a single copy of the process, the entire program can be tested on a sequential machine. On a multiprocessor, multiple copies of the process can be used to solve the problem. Their activities are synchronized by the monitor which ensures that no two processes work on the same problem, and that no subproblem is skipped.

6. Results

Table 1 summarizes test results that were run on the Denelcor HEP at Argonne National Laboratory. For the test cases, we considered n x n arrays of random integers where n took on the values 5, 10, 20, 40, 50, and 100. In each case the problem was solved using 1, 2, 4, 8, 12, and 16 processes. All times given are in milliseconds. The times are for solving problems and do not include initialization, etc.

Execution Time in Milliseconds						
n	*number of processes*					
	1	2	4	8	12	16
5	3.8	2.1	1.2	1.0	1.2	1.2
10	22.3	11.5	6.0	3.5	3.1	3.2
20	147.8	74.6	38.0	20.3	16.0	17.0
40	1049.0	528.7	268.9	143.3	121.6	121.0
50	3579.2	1801.3	913.8	472.5	328.3	261.4
100	27087.6	13614.8	6901.4	3553.7	2467.2	1956.9

Note that on the smaller problems, only modest speed ups were realized at first, and decays even began to creep in as more processes were added. This is due to the fact that, under the chosen partitioning scheme, the smaller problems did not partition into as many subproblems as there were processes. Thus, the extra processes entered the monitor only to discover there was nothing for them to do.

The algorithm showed substantial speed ups (over 13) for the larger problems, where there were more subproblems than processes. Indeed, these speed ups are about the best that can be obtained [3] on a single-PEM HEP such as we were using. In the larger problems of course, there was almost always something in the pool when a process would enter the monitor asking for work.

We were able to port the code to a Sequent Balance 8000 machine and obtain a few preliminary timings. The obtained timings were different from those on the HEP, but the relative speed ups were approximately the same.

References

1. J. Bentley, "Algorithm design techniques," *Communications of the ACM*, vol. 27, no. 9, pp. 865-871, September, 1984.
2. J. Bentley, "Perspective on Performance ," *Communications of the ACM*, vol. 27, no. 11, pp. 1087-1092 , November, 1984.
3. Harry F. Jordan, *HEP Architecture, Programming and Performance*, pp. 1-40, MIT Press, Parallel MIMD Computation: HEP Supercomputer and its Applications, 1985.
4. Ewing L. Lusk and Ross A. Overbeek, "Implementation of Monitors with Macros: A Programming Aid for the HEP and Other Parallel Processors," Technical Report ANL-83-97, Argonne National Laboratory, Argonne, Illinois, December 1983.