IMPLEMENTING A PARALLEL CONNECTED COMPONENT ALGORITHM ON MIMD ARCHITECTURES

Robert Hummel
Alan Rojer

Reprinted from IEEE COMPUTER SOCIETY WORKSHOP ON COMPUTER ARCHITECTURE FOR PATTERN ANALYSIS AND IMAGE DATABASE MANAGEMENT — CAPAIDM
Miami Beach, Florida, November 18-20, 1985

IMPLEMENTING A PARALLEL CONNECTED COMPONENT ALGORITHM ON MIMD ARCHITECTURES

Robert Hummel *and* *Alan Rojer*

Courant Institute of Mathematical Sciences
New York University
251 Mercer, New York, NY 10012

Abstract

The choice of an appropriate architecture for parallel image processing can have a large impact on the efficiency and the ease of implementation of vision algorithms. A shared memory MIMD architecture may be effective for higher level algorithms which require non-local communication between image elements. In such a machine, processor allocation and synchronization are key issues. We present techniques for allocation and synchronization of processors in the NYU Ultracomputer, using the fetch&add primitive. We apply this scheme to implement a parallel connected component algorithm. We conclude that the method allows a high degree of parallelism with relative ease of programming.

1. Introduction

Many higher level image processing algorithms suggest dynamic allocation of multiple processors to image processing subtasks. However, most parallel algorithms in image processing assume an SIMD architecture with processors statically assigned to one or a group of image pixels. For vision tasks such as component labeling, convex hulls of regions, feature extraction of regions, and object matching, fixed assignments of processors will inevitably lead to low parallelism, due to the high overhead of communication between distant parts of the image. It seems more natural to have an intermediate number of more powerful processors to conduct tasks which require global information. If memory is shared among MIMD processors, the communication overhead is removed (or hidden) and the processors become the limiting resource. Then we need to consider allocation and synchronization of the processors. If we can structure the problem as a (possibly dynamic) set of tasks which must be created and performed, an MIMD algorithm results by simply assigning any available processor to a queued task, which may result in creation of further tasks.

Our target machine is the NYU Ultracomputer, a shared memory MIMD machine, which uses the fetch and add primitive for communication and control of processes. The fetch and add instruction returns to the issuing processor the value of a specified shared variable, and has the side effect of incrementing the value stored in the variable by a specified increment. Concurrent fetch&add's are handled as though they had been issued in some arbitrary sequential order, but with no execution time penalty [3].

We study, as an example, the Shiloach/Vishkin connected component algorithm, which is an O(logn) SIMD algorithm which requires roughly 4n processors, where n is the number of pixels. We show how the algorithm may be converted to an MIMD algorithm, and acheive nearly optimal speedup from limited parallel resources. We also present enhancements in the implementation to reduce redundant parallel work.

2. Processor Allocation and Syncronization in the NYU Ultracomputer

We assume the algorithm we wish to implement is structured as a sequence of *steps*, each of which may be decomposed to a set of independent *tasks*. We assume that all tasks in a step must be completed before the next step is begun, but that within a step, tasks may be executed in parallel or asynchronously, in arbitrary order. Many image processing algorithms have this structure. Thus our allocation problem is simply to assign available processors to outstanding tasks. The synchronization problem is simply to prevent work beginning on a task which is not in the current step.

To allocate processors to tasks in a step, we use an array of pointers to the data to be processed. This array is a task list. At inception of a step, or on completion of a task, a processor executes a fetch&add with increment 1 to the (shared) list index of this array. The nature of the fetch&add ensures a unique index for each request. The processor can then perform the task associated with the returned index; no other processors will be assigned to the same task. Tasks can be created in a similar fashion.

A fetch&add to the list index of a new task list provides an index in which to write a pointer to the new task.

Synchronization of processors to ensure sequential execution of steps is accomplished in a complementary fashion. We store the total number of tasks (i.e. the length of the task queue) in a shared "coordination variable" at the beginning of the step. On completion of a subtask, a processor does a fetch&add to the coordination variable with increment -1. At any time, the coordination variable contains the number of uncompleted subtasks. In contrast, the list index contains at all times the number of subtasks which have been allocated (i.e initiated). When there are no tasks remaining (list index less than number of tasks), the processor waits for completion of the step (zero coordination variable) before beginning the next step. Note this strategy does NOT permit creation of tasks for the current step, since it relies on a static number of tasks. Tasks for future steps of course can be created. An illustration of the general control structure for a step follows.

```
/* General control structure for parallelization of a step */
/* N subtasks */

shared int J; /* task list index */
shared int T; /* coordination variable */
private j; /* local index to current task */

J ← 0;
T ← N;
/* (all processors) cobegin */
/* request subtask */
while((j ← fetch&add(J,1)) < N ) {
  /* perform allocated subtask on object indexed by j */
  /* report completion of subtask */
  fetch&add(T,-1);
  }
/* coend */
while (T != 0) /* wait */ ;
```

The overhead in this approach stems from extra time spent on task allocation (i.e. fetch&add to list indices), as well as time at the end of the cycle when there are no tasks left as as processors wait for other processors to finish before they begin the next step. Let t_p be the time to perform the processing of a subtask. In a fully parallel implementation, t_p is sufficient for the entire step. Let t_a be time spent on allocation, in particular the extra fetch&add required to coordinate the list index. Then the time t_s required for a subtask in the partially parallel implementation is given by $t_s=t_a+t_p$. If we have k processors, we can complete k subtasks in time t_s. Thus, the time $T(N)$ required for N subtasks is

$$T(N)=\left\lceil\frac{Nt_s}{k}\right\rceil=\left\lceil\frac{N(t_a+t_p)}{k}\right\rceil.$$

If t_a is small relative to t_p and k is small relative to N, we approach full parallel speedup of $T(N)=t_p/k$.

3. A Parallel Algorithm for Connected Components of a Graph

We consider an Ultracomputer implementation of a parallel algorithm for connected components. The algorithm was discovered by Shiloach and Vishkin [1]. Their computation model is similar to the Ultracomputer; however, they assume a processor for each vertex and two for each edge. For reasonable sized images and Ultracomputers, we are likely to have far fewer processors available.

Each vertex in the graph is assigned a *parent* pointer. The algorithm uses these pointers to construct a forest of root-directed trees, also called a *pointer graph*, each of which comprises vertices known to reside in the same connected component. Initially each vertex is its own parent, i.e., each vertex comprises a rooted tree with only one node. At termination, the parent pointers of all vertices in the same connected component point to the same root vertex.

Each vertex is assigned a processor. Each edge is assigned two processors; in effect, a processor in each direction of edge traversal. Two operations are performed in the algorithm. The *hook* operation directs the pointer of the root of a tree (self-directed before the operation) to point to some node in another tree. Hooking is mediated by the edge processors. The *shortcut* operation redirects a vertex's pointer to the parent of its parent, reducing the height of the trees. Shortcutting is mediated by the vertex processors.

At intermediate stages of the algorithm, the trees in the pointer graph can be classified as *live*, *stagnant*, and *dead*. A live tree has been shortcut or hooked in the current iteration. A stagnant tree has not been shortcut or hooked in the current interation. A dead tree has not been shortcut or hooked in the previous iteration. The algorithm follows.

```
        /* intitialize */
[0]     (∀v)[v ∈ V] cobegin
          parent[v] ← v;
          age[v] ← 0;
        coend
        I ← 0;
        While (∃v)[age[v] = I] begin
          I++;
          /* Shortcutting */
[1]       for all v ∈ V cobegin
            old-parent[v] ← parent[v];
            parent[v] ← parent[parent[v]];
            if (old-parent[v] ≠ parent[v]) age[parent[v]] ← I;
          coend
          /* Ordered Hooking */
[2]       for all (u,v) ∈ E cobegin
            If parent[u] = parent[parent[u]] /* u points to a root */
            and parent[u] < parent[v] then begin
              parent[parent[u]] ← parent[v];
              age[parent[v]] ← I;
            end
          coend
```

```
      /* Stagnant Hooking */
[3]   for all (u,v) ∈ E cobegin
        if parent[u] = parent[parent[u]] /* u points to a root */
        and age[u]<i /* stagnant */
        and parent[u] ≠ parent[v] then begin
          parent[parent[u]] ← parent[v];
          age[parent[v]] ← I;
        end
      coend
      /* Shortcutting */
[4]   for all v ∈ V cobegin
        old-parent[v] ← parent[v];
        parent[v] ← parent[parent[v]];
        if (old-parent[v] ≠ parent[v]) age[parent[v]] ← I;
      coend
    end
```

The proof of the complexity of the algorithm depends on the observation that the shortcut operation leads to a height reduction which is logarithmic in I, while hooking only increases the heights of trees additively by combining them. The algorithm is shown [1] to have a bound on the number of iterations of $O(\log_{3/2}|V|)$.

4. Implementation Concerns

The major concern in implementing the alogrithm is allocation of processors. In the existing Ultracomputer, there are far less than $2M+N$ processors for any reasonably sized image. We can save N processors by allowing the edge processors to handle the shortcutting in steps 1 and 4. We can take advantage of the bounded connectivity of pixels in the image to note that we require only $4N$ processors. We still are not likely to have nearly enough processors available.

In the extreme case of execution of this algorithm by a single processor, we can determine that the algorithm is $O(M\log N)$, since control of the main loop of $\log N$ iterations remains the same, and each step inside the main loop will require at most $O(M)$ operations. For a planar graph like an image, this becomes $O(N\log N)$. Our goal is to acheive parallel speedup to the full limit of our processor resources; for $k<N$ processors, the maximum speedup is $O(N\log N/k)$. In the case of $k \ll N$, we achieve nearly this optimal speedup.

We consider now modifications to the algorithm along the lines indicated in Section 2, equeuing tasks and eliminating subtasks which don't contribute to the solution. In particular, we are interested in recognizing edges which have nothing to add to the solution because the vertices they connect are known to reside is the same component, and in recognizing vertices which are components of dead trees in the pointer graph. These vertices can be shown to comprise a connected component, so they do not require further consideration.

Consider first vertices in a dead tree. In a dead tree, all vertices point to the root (otherwise it could be shortcut). Furthermore, the age of the root is less than the iteration counter, indicating that no changes have been made in the last iteration. We can introduce a "pruning" step, which examines every vertex, and enqueues only vertices which don't point to dead roots. This queue forms the list of subtasks to be performed in the next shortcut step.

We note that edges can be classified into three groups. We say an edge is *used* if it has succeeded in hooking. An edge is *redundant* if both its vertices point to the same parent. An edge is *indeterminate* if it is neither used nor redundant. We can tell if an attemped hook succeeds by limiting access to the root. In effect, we let edges compete for the right to hook by using the fetch&add to some variable which is initially 0. Only the edge which receives the zero result actually performs the hook, so it is assured of success. Obviously, we can readily identify a redundant edge. Thus after each subtask in a hook step, we enqueue edges which are indeterminate onto a new edge list, eliminating used and redundant edges. This queue is used as the task list for the next hook step.

We can keep the active lists of edges and vertices by allocating space for two vertex task lists, and two edge task lists. Initially, all pixels within "on" regions are placed on one of the vertex lists, and all edges between these vertices are enqueued on one of the edge lists. During the first steps, we read from the active list, and enqueue tasks to the other list. We then alternate between these lists, using one for current tasks and one for tasks for the next step. Indicies returned by the fetch&add are used to allocate tasks to available processors. Note that removing redundant tasks doesn't affect the asymptopic complexity of the algorithm, although in practice it should reduce the run-time.

5. References

[1] Y. Shiloach and U. Vishkin, "An O(log n) Parallel Connectivity Algorithm", *Journal of Algorithms*, **3**, pp. 57-67 (1982).

[2] R. A. Hummel, "Image Processing on the NYU Ultracomputer", Courant Institute NYU Ultracomputer Note No. 72 (1984).

[3] A. Gottlieb, R. Grishman, C.P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU-Ultracomputer-Designing an MIMD shared Memory Parallel Computer", *IEEE Transactions on Computers*, **C-32**, No. 2, pp. 175-189 (1983).