Dynamic Processor Allocation for Parallel Algorithms in Image Processing

by

Robert A. Hummel
Kiazhong Zhang

Ultracomputer Note #123
Robotics Report #94
Computer Science Technical Report #268
January, 1987

This research was supported by NSF grand DCR-8403300 and Office of Naval Research Grant N00014-85-K-0077, Work Unit NR 4007006. The code for the Ultracomputer, given in the appendix of this paper, from a psuedo-code version that appears in an earlier paper. [1]

Abstract

Because image processing is numerically intensive, there has been much interest in parallel processing for image analysis applications. While much of low-level vision can be attacked by SIMD mesh-connected architectures, intermediate and high-level vision applications might be able to make effective use of MIMD and distributed architectures. We have taken a standard parallel connected components algorithm, and applied it to image segmentation using an MIMD architecture. The resulting version of the Shiloach/Vishkin algorithm runs on the prototype NYU Ultracomputer. We will describe the implementation and the results of some experiments. We take note of the lesson learned from this implementation: that processor power should be focused dynamically to those portions of the image requiring greatest attention. We then consider the implications of this lesson to other image processing tasks.

1. Parallelism in Image Processing

Low-level vision, including image sensing, enhancement, deblurring, and simple feature extraction is, for the most part, mediated by extremely local, convolution-like operations. Mesh-connected computers such as the MPP[5] or pipeline processors are able to handle these operations very efficiently. Intermediate-level and high-level vision requires more global communication, and greater flexibility in processor programming. To effectively devote multiple processors to high-level vision tasks, we expect to be led to shared-memory architectures, with each of a number of powerful processors capable of accessing all data representing the image. The question arises, then, as to how to coordinate the multiple processors to complete the vision tasks efficiently and without redundancy.

Certain image processing tasks are trivial to parallelize. With the appropriate architecture, convolution, feature extraction, histograming, and even Hough transforms can be coded easily. (We need, for many of the tasks, a fast way of summing all values in an image.) However, many standard algorithmic tasks require development of specialized parallel algorithms. An example, to be considered in some detail later in this paper, is the connected components labeling problem. There are several dozen other examples of parallel algorithms that have been developed and described in the last few years. Each is typically the serendipitous discovery of some very clever and experienced algorithm designer. (The name Uzi Vishkin comes up remarkably often when discussing "pram" — parallel random access machine — algorithms.) Many of these algorithms have applications to image processing. Apart from the connected components algorithm, there are now parallel algorithms for a number of computational geometry results[1], including convex hulls, Voronoi diagrams, and minimum spanning trees. There are also substring matching algorithms[8], and inexact string matching[4], which certainly have applications to model-based vision.

2. Shiloach/Vishkin Connected Components Algorithm

The Shiloach/Vishkin connected components algorithm[7] is an $O(\log N)$ parallel algorithm for an SIMD shared-memory architecture. The algorithm requires a processor for each node and each edge in the graph. In a recent paper[3], we have analyzed this algorithm in detail, and have shown how it can be mapped to an MIMD architecture with fewer processors for image processing applications. The time complexity is then $O((N/P)\log N)$, where P is the number of processors. However, we believe that the algorithm can be of practical use for image processing applications with even modest numbers of processors. A typical number of processors anticipated in an NYU Ultracomputer, for example, will be 512.

For the purposes of this paper, we will describe briefly the original Shiloach/Vishkin algorithm, and describe the general ideas involved in converting the algorithm to an MIMD architecture. We have implemented this algorithm on a prototype eight-processor NYU Ultracomputer. The code is listed in the appendix, and results will be described in the next section. In the final section, we consider some implications for dynamic processor allocation in parallel algorithms for a couple other image processing applications.

In the Shiloach/Vishkin algorithm, every pixel with a "one" value in the binary image has a "parent pointer" which can point to any other pixel. The pixels are numbered, and at the outset, each pixel sets its parent pointer to point to itself. The algorithm proceeds by iteratively applying the following four steps. Each block of four steps constitutes one iteration.

Step 1: Short-cutting. Each pointer looks at the pixel to whom it points. Suppose pixel u points to v. At v, we check the pointer, and determine that the pointer there points to w. The pointer at u is changed to point to w instead of v. This is done concurrently at all pixels.

Step 2: Ordered hooking. Each ordered edge (u,v) looks to see if it is in a configuration where u points to a pixel x that is a root (i.e., x's pointer points to x), and v points to a pixel y that has a larger value than x (i.e., $y>x$). In that case, the processor doing this check for the edge (u,v) tells x to change its pointer to point to y. Note that x may receive several simultaneous instructions. In the case that these instructions conflict, exactly one such instruction succeeds, and all others are thrown away. The processor that successfully writes x's new parent pointer is arbitrary.

Step 3: Stagnant node hooking. Each edge processor (u,v) checks to see if it is in a configuration where u points to a stagnant root x. A stagnant node is a node that has no new pointers pointing into it as a result of the immediately preceding Step 1 and 2. As before, x is a root if x's pointer points to x. In the case that u is a stagnant root, and v points to a node y that is different than x, then the processor (u,v) tells x to change its pointer to point to y. Concurrent writes are handled as in step 2. Unlike step 2, y will not have a larger numerical value than x.

Step 4: Short-cutting. Step 1 is repeated. If there are no changes, the algorithm is done. If there are some changes, then after completing step 4, the next iteration begins with another short-cutting operation, step 1. Actually, step 4 can be omitted, but its inclusion can halve the number of iterations that are required for completion.

Note that global communication is needed, since each processor must be able to access any pixel. As defined, the algorithm requires one processor for each node, and one processor for each directed edge. The algorithm is SIMD, in the sense that every node processor and every edge processor can execute, synchronously, the same code. However, communication paths may vary among the processors, and the time bounds require that there be no time penalties for contention on the communication paths.

The fact that the algorithm works, and the $O(\log N)$ time bound, are not immediate. Details are given elsewhere[3].

Next, we consider how to convert this algorithm for implementation on an MIMD machine. The basic idea is simple: form lists of the nodes and edges that must be processed in steps 1, 2, 3, and 4. We then allow processors to dequeue items from the lists, and process each item appropriately. There are three critical points:

- We should keep the lists as short as possible. If it is known that an edge or node does not need to be processed, then it should not show up on the lists. This implies that the lists must be created dynamically: that the lists for one iteration should be formed during the previous iteration. In this way, processor power is concentrated on locations where processing is needed. This is the main point of this paper.
- There needs to be a way for a processor to grab an item from the list, reserve that item for itself, and ensure that no other processor simultaneously or subsequently grabs the same item. On the NYU Ultracomputer, this allocation of items from a queue is accomplished by means of the "fetch-and-add" instruction. This not very well-known concept is undoubtedly of fundamental importance to parallel processing. An

alternative, which on the face of it is not very satisfactory, is to have a master processor in charge of all allocation of tasks.

- We must identify when the subtasks can be performed asynchronously, and then there must be a synchronization. For the Shiloach/Vishkin algorithm, it turns out that nodes or edges may be processed asynchronously within each step without disrupting the ability of the algorithm to correctly label the components. In fact, asynchronous performance of the steps can result in fewer iterations needed for convergence. However, there must be synchronization between steps. That is, all nodes must be processed in step 1 before any edges can be considered in step 2, for example.

In the Shiloach/Vishkin algorithm, it is possible to determine when a node or edge need not be considered further. Nodes can be deleted from consideration when the component to which they belong has converged. A component is done when all pointers in that component point to a single pixel.

Similarly, an edge need no longer be considered after it has caused a hook to take place, or if both vertices of the edge have pointers pointing to the same pixel.

Accordingly, the node and edge lists can be handled as follows. Initially, all pixels within the "white regions " of the image belong to the node list. All edges joining pixels within the "white regions" are placed on the edge list. In the first iteration, step 1 may be skipped. Thus we begin by processing step 2 using the initial edge list. During this processing, a new edge list is formed. The new edge list is used for the list of edges to be processed in step 3. Then during step 3, a new edge list is again formed. This new edge list is used as the list of edges for step 2 of the next iteration. Step 4 used the node list, and forms two new lists of nodes. One list is used to process nodes in step 1 of the next iteration, and the other is used for step 4 of the subsequent iteration. The former list will always be a subset of the latter. We refer to these lists as Queue 1 for the node list used by step 1, Queue 2 and Queue 3 for the edge lists used by steps 2 and 3 respectively, and Queue 4 as the node list used by step 4. Initially, we are given Queues 2 and 4, which will be universal lists containing all edges and all nodes. Queue 2 is used to form Queue 3. Queue 3 forms the next Queue 2. Queue 4 is used to form Queue 1 for the next iteration, and also the subsequent Queue 4.

3. Results

The parallel MIMD approach to the Shiloach/Vishkin connected components algorithm described in the previous section has been implemented on a parallel machine. We present in the Appendix the actual parallel code, and hope that some readers will find it interesting to inspect the way the parallel algorithm maps into the programming. The code was then run on a prototype NYU Ultracomputer. The current prototype has eight processors, although the identical code could be used to run on a 512 processor (or, for that matter, an N-processor) ultracomputer. (IBM is building a parallel computer, the "RP-3," which encompasses a 512 processor ultracomputer). Figure 1 shows the results of running the algorithm on a small test image. Of course, the connected components are correctly labeled.

Of greater interest is the size of the queues. One hopes that the queue lengths will drop quickly, so that later iterations require much less work than earlier iterations. In Figure 2, we have plotted the queue lengths as a function of iteration for a 512 by 512 image run using the same algorithm. It should be noted, incidentally, that the queue lengths, and indeed the number of iterations, is not a deterministic function of the image. Because the ultracomputer does not specify which processor wins in a concurrent write, and because the algorithm requires concurrent writes in steps 2 and 3, separate runs of the algorithm on a single image can produce different results. Thus the image in Figure 1 generally required five iterations, but sometimes used six iterations. Likewise, the plots shown in Figure 2 could vary from run to run. However, the variations are slight. Note that the 512 by 512 example for Figure 2 required only eight iterations on the run used for the plots. This run was typical.

In fact, the queue lengths do drop, but not exponentially. The edge lists become small quickly, whereas the node lists stay large and drop off precipitously in the last few iterations. The length of the queue measures the amount of work required in that step of the corresponding iteration. The short-cutting steps, of course, are simpler than the edge processing steps, but still require three accesses to shared memory. The processing of each item on the queue can take a variable amount of time. For example, if a hook is required, processing an edge in step 2 takes much longer than if a hook is not required. Also, access to shared memory is not guaranteed to take a fixed amount of time. Thus items in steps 1 and 4 can take variable amounts of time, depending on memory access times. Processors simply process items in turn, and when completed with the current item, grab the next one from the queue. In this way, processors are kept busy processing the algorithm where work is needed. Nonetheless, on the average, we expect that if there are L items on a list, and P

processors, and a "fetch-and-add" computation is used to do processor allocation, then $O(\lceil L/P \rceil)$ time will be needed to process the step.

It turns out that step 3 of the Shiloach/Vishkin algorithm may be omitted, and convergence to correctly labeled connected components is still guaranteed. However, the $O(\log N)$ time bound is valid only with the inclusion of step 3. Nonetheless, we removed step 3, and ran some experiments on example images. We conjecture that the average case performance will remain $O(\log N)$, although the worst case may become $O(\sqrt{N})$. In our experiments, the number of iterations actually required became less without the inclusion of step 3, after the initialization was modified (see below). However, the length of the queues are larger. In particular, the node list was kept constant, since it is no longer easy to determine when a component is complete, before the entire algorithm terminates. This saves the bother of having to make a new list during each step, but keeps the list quite long. The edge lists, on the other hand, are allowed to decrease as before. The advantage of omitting step 3 is that when done, the pointers within a component will necessarily point to the largest value pixel within that component. With step 3, the stagnant node hooking in step 3 can result in the root node of a labeled component being less than the maximum node within that component. A simple $O(\log N)$ algorithm can then be used to find the actual maximum, but it is interesting that without step 3, the maximum node is found as part of the labeling process. Although the complexity analysis yields no advantage with the omission of step 3, and indicates that there might be asymptotically more iterations required, it is our guess that omitting step 3 will be useful. We also suspect that it is advantageous, in this case, to initialize the pointer graph slightly differently. Here, we should have each node point to its maximum nearest neighbor. The algorithm can then begin with step 1. More empirical analysis, and timing of results, will be needed.

4. Other Image Processing Applications

The lesson of our analysis and implementation of the Shiloach/Vishkin connected components algorithm is that higher-level vision processing can make use of dynamic allocation of processors to concentrated processor power where processing is needed. We now briefly consider a applications other than connected components labeling, and consider how dynamic processor allocation can be applied to these problems.

Convex hulls, Voronoi diagrams, and other computational geometry results are of primary importance in shape analysis and matching. For example, the convex hull of N points can be found in time $O(\log N)$ by a rather unobvious new parallel algorithm[1]. Another common vision task in object recognition involves branch-and-bound search. For example, in model-based vision, we wish to match extracted edges to edges of the model. A error can be accumulated, and if the error in a proposed match becomes larger than an already-considered match for the same set of edges, the proposed match can be dropped. Branch-and-bound and related parallel searches are easy to parallelize, and make use of global shared memory constructs and dynamic processor allocation.

Many recognition tasks in computer vision make use of a border trace of a region. The conversion of a raster binary image into a set of chain codes for the borders of the objects is a standard problem, and the serial border-following algorithm is a classical topic of elementary courses in computer vision[6]. Border-following can be made parallel, by allowing processors to execute their own border-following code on unmarked edge pixels, and mark a pixel as completed whenever the edge is place on a list. When a processor meets an edge pixel that has already been visited, it has completed its construction of an edge segment, and can proceed to construct a new segment. The result is that the edges of the regions are converted into a linked list structure of edge segments. Each edge segment, formed by a single processor performing border following, can be represented by a list (a queue) of one or more edge pixels.

However, linked lists are not favorable data structures in a parallel processing environment. Linked lists inherently require serial processing. We would prefer to restructure the border list as a queue. Then features such as Fourier descriptors and curvature measures can be computed by multiple processors performing portions of the computation on sections of the border. We saw in the previous section that allocation of tasks from queues is the appropriate way to coordinate multiple processors. Thus the border lists should be organized as queues. Unfortunately, the independent processors performing border following produce linked lists of segments, where the links are formed by having a processor set a pointer when it finds a marked edge pixel. (This means that the markings on visited edge pixels has to include the information as to which list that pixel belongs.)

Uzi Vishkin and others have considered parallel algorithms for the conversion of linked lists to queues. The appropriate method, resulting in an $O(\log N)$ algorithm, is similar to the short-cutting step of the connected components algorithm. The method can be modified to apply to the conversion of a linked list of queues into a single queue, as long as each queue has known length. We give a very brief description here, where we assume that each initial queue is of length one, i.e., the linked list is a simple linked list. Initially, a pointer graph is assigned, as in the previous algorithm, except that in this case the initial pointer graph is the linked list. Each pointer is associated with a distance to the parent node. Initially, all distances are either zero or one. Short-cutting is performed. If node u points to v, and v points to w, and the pointer at u has length d_1, and the pointer at v has length d_2, then after short-cutting, u points to w and has length d_1+d_2. After $\log(N)$ iterations, short-cutting will cause no further changes, and each node will point to the tail of the list. The distances will give the relative position of each node, so that each node can then simply write itself to the appropriate position in a queue. Some recent results have shown that this this same $O(\log N)$ performance can be achieved with only $N/\log N$ processors, but we omit the details.

References

1. Aggarwal, A., B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap, "Parallel computationl geometry," *Proceedings of the 26th Fundamentals of Computer Science (FOCS) Conference*, pp. 468-477 (1986).

2. Hummel, Robert A. and Alan S. Rojer, "Implementing a parallel connected component algorithm for the NYU Ultracomputer," *Proceedings of the IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, pp. 88-90 (November, 1985).

3. Hummel, Robert A., "Connected component labeling in image processing with MIMD architectures," pp. 101-127 in *Intermediate-level Image Processing*, ed. M. J. B. Duff,Academic Press (1986).

4. Landau, G. and U. Vishkin, "Introducing efficient parallelism into approximate string matching and a new serial algorithm," *Proceedings of the 18th ACM symposium on Theory of Computing*, pp. 220-230 (1986).

5. Potter, J. L., *The massively-parallel processor,* MIT Press, Cambridge, MA (1985).

6. Rosenfeld, Azriel and A. C. Kak, *Digital Picture Processing,* Academic Press, New York (1976).

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

8. Vishkin, Uzi, "Optimal Parallel Pattern Matching in Strings," *Proceedings of the 12th ICALP*, pp. 497-508 Springer-Verlag, (1985).

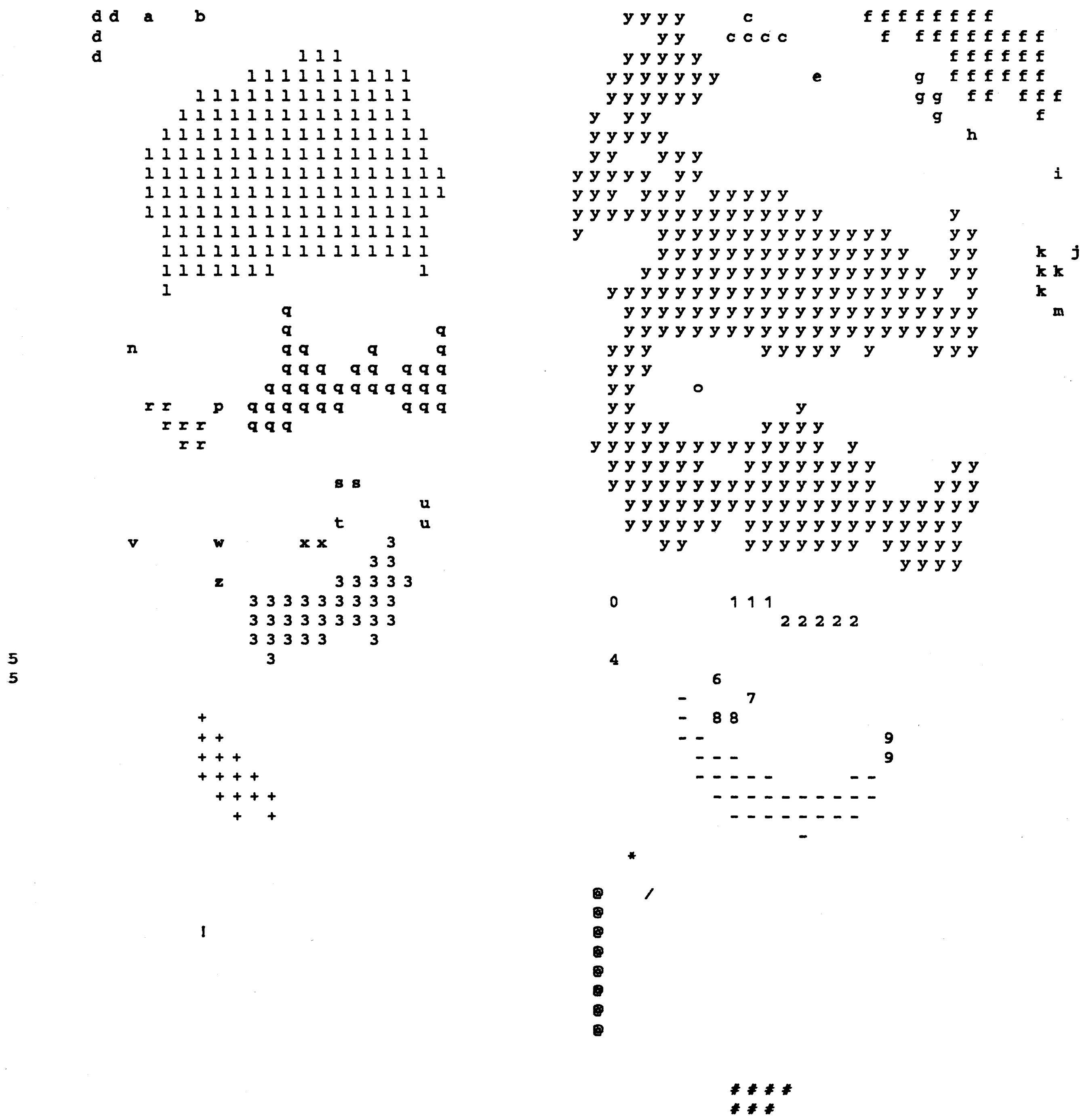
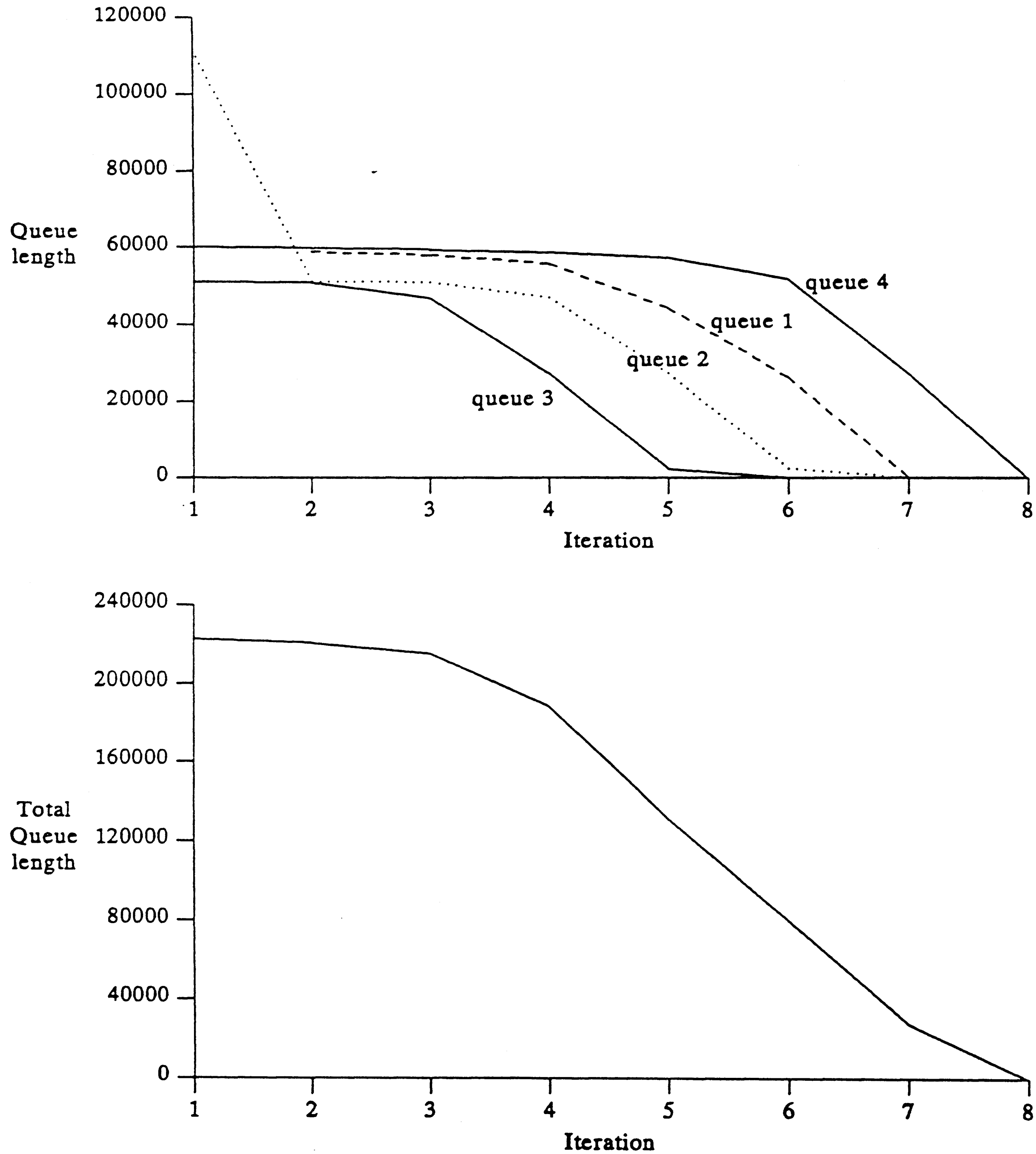


Figure 1. A binary image, and connected components labeled by the parallel Shiloach/Vishkin algorithm. (From the figure, it is impossible to tell that the processing was done on a truly parallel machine). Five iterations of the algorithm were needed for this 64 by 64 example.

Figure 2. Plots of the Queue lengths at each iteration for a 512 by 512 binary image (similar to the image shown in Figure 1). Note that queue 1 begins only at the second iteration. The lower plot shows the sum of all queue lengths in a single iteration as a function of the iteration. The lengths of the queues form a measure of the amount of work needed.



Appendix

Parallel code for connected components algorithm

```
/* Shiloach/Vishkin algorithm, coded in parallel C
in NYU Ultracomputer  */

#include <stdio.h>
#include <par.h>
#include <busy.h>
#include <errno.h>
extern int errno;

#define NP 8
#define N 64  /* Image is N by N */
typedef int pnode; /* pnode should be in range 0..N*N-1 */
typedef struct e {pnode x,y;} edge;

shared char  Img[N*N];/Image is N by N, raster scan order */
shared pnode Parent[N*N];
shared int   Age[N*N+1],Hooked[N*N+1];

main()
{
  getImg();
  SVCC();
  getoutput();
}

getImg() /* Procedure to get Image */
{
  int i,j;
  char ch;

  for (i=0; i<N; i++)
  {
    for (j=0; j<N; j++)
    {
      scanf("%c",&ch);
      if (ch=='*')
        Img[i*N+j]=31;
      else
        Img[i*N+j]=0;
    }
    scanf("0);
  }
}

getoutput() /* Procedure to get output */
{
  static char cst[47]={' ','a','b','c','d','e','f','g','h','i','j',
       'k','l','m','n','o','p','q','r','s','t','u','v','w','x','y',
       'z','0','1','2','3','4','5','6','7','8','9','+','-','*','/',
       '!','@','#','$','%','^'};
```

```
 int i,j,neg;

 neg=0;
 for (i=0;i<N*N;i++)
   if (Img[i]!=0 && Parent[i]==i) {
     neg=neg-1;
     Parent[i]=neg;
   }
 for (i=0; i<N; i++) {
   for (j=0; j<N; j++) {
     if (Img[i*N+j]!=0) {
       neg=Parent[i*N+j];
       if (neg>=0)
         neg=Parent[neg];
       neg=0-neg;
       if (neg<=0)
         printf("%c",'*');
       else
         printf("%c",cst[neg]);
     }
     else
       printf("%c",' ');
   }
   printf("0);
 }
}

shared static int  I;
shared static int  Index;

SVCC()
{
/* Vertex Lists */
shared static pnode Vlist1[N*N],Vlist4a[N*N],Vlist4b[N*N];
/* Edge Lists */
shared static edge  Elist2[2*N*N+1],Elist3[2*N*N+1];
shared static int  NV1,NV4,NE2,NE3;
shared static bw_barrier_t barr;

shared static int nv4;
int taskid,i;

  bw_barrierinit(&barr,NP);
  taskid=spawn(NP-1,0,(int*)NULL,0,(int(*))NULL,(void*)0);
  if (taskid<0) {
    printf("Spawn failed0);
    exit(4);
  }
  if (taskid==0) {
    NV4=0;
    NE2=0;
    Index=0;
  }
  bw_barrier(&barr);
/* Create vertex list for step 4 and edge list for step 2 */
  step0(Vlist4a,&NV4,Elist2,&NE2);
  if (taskid==0)
    I=0;
  bw_barrier(&barr);
```

```
 while (NV4>0) { /* While there are non-dead trees */
  if (taskid==0) {
   I=I+1;
   printf("0);
   printf("Iteration %d 0,I);
  }
  bw_barrier(&barr);
  if (I>1) {
   if (taskid==0) {
    printf("    Step1:  Queue Length=  %d0,NV1);
    Index=0;
   }
   bw_barrier(&barr);
   step1(Vlist1,NV1); /* Uses vertex list Vlist1 */
   bw_barrier(&barr);
  }
  if (taskid==0) {
   printf("    Step2:  Queue Length=  %d0,NE2);
   NE3=0;
   Index=0;
  }
  bw_barrier(&barr);
  step2(Elist2,NE2,Elist3,&NE3); /*Using Elist2, create Elist3 */
  bw_barrier(&barr);
  if (taskid==0) {
   printf("    Step3:  Queue Length=  %d0,NE3);
   NE2=0;
   Index=0;
  }
  bw_barrier(&barr);
  step3(Elist3,NE3,Elist2,&NE2); /*Using Elist3, create Elist2 */
  bw_barrier(&barr);
  if (taskid==0) {
   printf("    Step4:  Queue Length=  %d0,NV4);
   NV1=0;
   nv4=NV4;
   NV4=0;
   Index=0;
  }
  bw_barrier(&barr);
/* Using Vlist4, Create Vlist1 and Vlist4 */
  if (isodd(I))
   step4(Vlist4a,nv4,Vlist1,&NV1,Vlist4b,&NV4);
  else
   step4(Vlist4b,nv4,Vlist1,&NV1,Vlist4a,&NV4);
  bw_barrier(&barr);
 }
 if (taskid>0)
  exit(0);
 else
  while((i=mwait(0)) >0 || (i <0 && errno != ECHILD)) {
   if (i<0) continue;
   printf("%d children terminated abnormally.0,i);
  }
}

/* Initializes pointer graph and make vertex list and edge list */
step0(Vlist,pNV,Elist,pNE)
 pnode Vlist[];
```

```
 edge  Elist[];
 int   *pNV,*pNE;
{

   pnode i;
   int k;

   while ((i=faa(&Index,1))<N*N-N) {
     if (Img[i]!=0) {
       Parent[i]=i; /* Self pointing root */
       Age[i]=0;
       Hooked[i]=0; /* Counter for hook requests */
       k=faa(pNV,1);
       Vlist[k]=i; /* Enqueue node i */
       if (Img[i+1]!=0) {
         k=faa(pNE,1);
         Elist[k].x=i; /* Enqueue east edge */
         Elist[k].y=i+1;
       }
       if (Img[i+N]!=0) {
         k=faa(pNE,1);
         Elist[k].x=i; /* Enqueue south edge */
         Elist[k].y=i+N;
       }
     }
   }
}

/* Shortcutting */
step1(Vlist,NV)
pnode Vlist[];
int NV;
{

   pnode old_parent,new_parent;
   int k;

   while ((k=faa(&Index,1))<NV) {
     k=Vlist[k];
     old_parent=Parent[k];
     new_parent=Parent[old_parent];
     Parent[k]=new_parent;
     if (old_parent!=new_parent)
       Age[new_parent]=I;
   }
}

/* Ordered Hooking */
step2(Elist,NE,Elistout,pNEout)
edge Elist[],Elistout[];
int NE,*pNEout;
{

   edge e;
   pnode u,v;
   int k,h;

   while ((k=faa(&Index,1))<NE)
   {
```

```
        h=1;
        e=Elist[k]; /* Consider edge (e.x,e.y) */
        u=Parent[e.x];
        v=Parent[e.y];
        if (u<v && Parent[u]==u)
        {
         h=faa(&Hooked[u],1);
         if (h==0) /* Allow hook only if not hooked yet */
         {
           Parent[u]=v;
           Age[v]=I;
         }
        }
        else if (u>v && Parent[v]==v)
        {
         h=faa(&Hooked[v],1);
         if (h==0) /* Allow hook only if not hooked yet */
         {
           Parent[v]=u;
           Age[u]=I;
         }
        }
        else
        {}
        if (u!=v && h!=0)
        {
         k=faa(pNEout,1);
         Elistout[k]=e;
        }
       }
}

/* Stagnant Hooking */
step3(Elist,NE,Elistout,pNEout)
edge Elist[],Elistout[];
int NE,*pNEout;
{

    edge e;
    pnode u,v;
    int k,h;

    while ((k=faa(&Index,1))<NE)
    {
     h=1; /* Hook only if h=0 */
     e=Elist[k];
     u=Parent[e.x];
     v=Parent[e.y];
     if (u!=v)
     {
       if (Age[u]<I && Parent[u]==u)
       {
         h=faa(&Hooked[u],1);
         if (h==0) /* Allow hook only if not hooked yet */
         {
           Parent[u]=v;
           Age[v]=I;
         }
       }
```

```
        else if (Age[v]<I && Parent[v]==v)
        {
          h=faa(&Hooked[v],1);
          if (h==0) /* Allow hook only if not hooked yet */
          {
            Parent[v]=u;
            Age[u]=I;
          }
        }
        if (h!=0)
        {
          k=faa(pNEout,1);
          Elistout[k]=e;
        }
      }
    }
}

/* Shortcutting */
step4(Vlist,NV,Vlist1,pNV1,Vlist4,pNV4)
pnode Vlist[],  Vlist1[],  Vlist4[];
int NV,*pNV1,*pNV4;
{

    pnode old_parent,new_parent;
    int k,j;

    while ((k=faa(&Index,1))<NV)
    {
      k=Vlist[k];
      old_parent=Parent[k];
      new_parent=Parent[old_parent];
      Parent[k]=new_parent;
      if (old_parent!=new_parent)
      {
        j=faa(pNV1,1);
        Vlist1[j]=k;
      }
      if (!(old_parent==new_parent && Age[new_parent]<I))
      {
        j=faa(pNV4,1);
        Vlist4[j]=k;
      }
    }
}

isodd(num)
int num;
{
  if (num%2==1)
    return(1);
  else
    return(0);
}
```