Directed Acyclic Graph
Using Explicit Instruction Parallelism

Code and Analysis Documentation

Efraim Berkovich
advisor: Prof. Uzi Vishkin

January 22, 1998

This document contains the C (extended C) and assembly code for each of the three programming models we investigated: serial superscalar, Spawn-MT, and EMT. Following the code, the document contains a step-wise analysis of the execution and tables with the overall execution work and time counts for each of the algorithms.

This document contains the appendices of a more detailed technical report which is under preparation. In addition to a high-level algorithm description, the report contains analysis of specific decisions made and the programming techniques used as well as results for various input graphs.

Note that the speedup results quoted in the main paper refer to ranges of parameters for classes of input graphs. The results can be derived through analysis which appears in the technical report.

Table of Contents

Appendix A: C and Assembly Code

This appendix is a listing of the header C file shared by all three algorithms and the listing of the three algorithms' C code. The C code is followed by the machine assembly code.

C Header file: DAG.H

```
/*         97/07/31    dag.h                         */

/*    Data Structures: Directed Acyclic Graph         */
/*                     Serial and Parallel Algorithms */


/*    NODE structure                                  */
struct {
  int in_edges_start;     /* ptr to edges_in array   */
  int in_edges_num;       /* num of incoming edges   */
  int out_edges_start;    /* ptr to edges_out array  */
  int out_edges_num;      /* num of outgoing edges   */
  int in_edge_count;      /* counter -- init to 0    */
  int max_length_path;    /* longest path to node    */
                          /* must be init =-infinity */
} NODE;



/*    EDGE structure                                  */
struct {
  int start_node;         /* start node              */
  int end_node;           /* end node                */
  int edge_len;           /* length of edge          */
} EDGE;




/*    INPUTS:                                         */
/*         nodes -- array containing the nodes in    */
/*                  the graph                         */
/*      edges_in -- array containing the edges in    */
/*                  the graph which go to a           */
/*                  particular node, partitioned by  */
/*                  node                              */
/*     edges_out -- array containing the edges in    */
/*                  the graph which go from a         */
/*                  particular node, partitioned by  */
/*                  node                              */


NODE      nodes[MAX_NODES];
EDGE      edges_in[MAX_EDGES];
EDGE      edges_out[MAX_EDGES];

int       num_nodes;
int       num_edges;
```

```
/*     AUXILIARY:                                        */
/*        nodesptr -- array to hold ptrs to enabled  */
/*                  nodes list                           */
/*        curr_enabled_nodes, next_enabled_nodes     */
/*                  ptrs to the current and next     */
/*                  list of node ptrs (the enabled   */
/*                  nodes) in the nodesptr array     */
/*        num_enabled, next_num_enabled              */
/*                  size of current and next enabled */
/*                  nodes list                           */

NODE       *nodesptr[MAX_NODES];

NODE       **curr_enabled_nodes;
NODE       **next_enabled_nodes;

int        num_enabled;
int        next_num_enabled;
```

C Serial Code: SER_DAG.C

```
/*   97/12/12   ser_dag.c                      */

/* SERIAL CODE:                                        */
/* Directed Acyclic Graph (DAG) longest Path         */
/*****************************************************/


void  Serial_DAG_Path()
{

  NODE *my_node;

  /*  enabled array creation by checking which nodes have */
  /*  ZERO in_edges_num                                   */
  /******************************************************/

  num_enabled = 0;
  next_num_enabled = 0;

  curr_enabled_nodes = &(nodesptr[0]);

  for(my_node = &(nodes[0]);
      my_node != &(nodes[num_nodes]);
      my_node += sizeof(NODE))
    {
      if (my_node->num_in_edges == 0)
       {
         curr_enabled_nodes[num_enabled] = my_node;
         num_enabled = num_enabled + 1;
       }
      my_node->max_length_path = MINUS_INFINITY;
    }


  /* loop for first layer -- max_length_path is MINUS_INFINITY */
  /***********************************************************/
  for(my_node = curr_enabled_nodes[0];
      my_node != curr_enabled_nodes[num_enabled];
      my_node += sizeof(NODE_PTR))
    {

      EDGE *this_edge;
      NODE *nextnode;
      int r;

      /* "activate" the outgoing edges                       */
      /******************************************************/
      for(this_edge = edges_out[my_node->out_edges_start];
         this_edge != edges_out[my_node->out_edges_start +
                          my_node->out_edges_num];
         this_edge += sizeof(EDGE))
       {

         nextnode = &(nodes[this_edge->end_node]);
         r = this_edge->edge_len;

         /* check for max in nextnode                        */
         /************************************************/
         if (r > nextnode->max_length_path)
           nextnode->max_length_path = r;
```

```
      nextnode->in_edge_count = nextnode->in_edge_count + 1;

      /* check if last incoming edge -> enable the node */
      /*************************************************/
      if (nextnode->in_edge_count == nextnode->in_edges_num)
        {
          /* last to arrive, so enable the node, by    */
          /* copying pointer into enabled nodes array  */
          /*********************************************/
          next_enabled_nodes[next_num_enabled] = nextnode;
          next_num_enabled = next_num_enabled + 1;
        }
    } /* end activate edges */
  } /* end loop for enabled nodes */


/* update the pointers & counters to the enabled array            */
/******************************************************************/
curr_enables_nodes = next_enabled_nodes;
next_enabled_nodes = &(next_enabled_nodes[next_num_enabled]);

num_enabled = next_num_enabled;
next_num_enabled = 0;


/* while there are still enabled nodes -> work */
/***********************************************/
while (num_enabled != 0)
  {
    for(my_node = curr_enabled_nodes[0];
      my_node != curr_enabled_nodes[num_enabled];
      my_node += sizeof(NODE_PTR))
     {

      EDGE *this_edge;
      NODE *nextnode;
      int r;

      /* "activate" the outgoing edges                        */
      /********************************************************/
      for(this_edge = edges_out[my_node->out_edges_start];
          this_edge != edges_out[my_node->out_edges_start +
                            my_node->out_edges_num];
          this_edge += sizeof(EDGE))
        {

          nextnode = &(nodes[this_edge->end_node]);
          r = my_node->max_length_path + this_edge->edge_len;

          /* check for max in nextnode                        */
          /****************************************************/
          if (r > nextnode->max_length_path)
           nextnode->max_length_path = r;

          nextnode->in_edge_count = nextnode->in_edge_count + 1;

          /* check if last incoming edge -> enable the node */
          /*************************************************/
          if (nextnode->in_edge_count == nextnode->in_edges_num)
           {
             /* last to arrive, so enable the node, by    */
             /* copying pointer into enabled nodes array  */
             /*********************************************/
```

```
                  next_enabled_nodes[next_num_enabled] = nextnode;
                  next_num_enabled = next_num_enabled + 1;
                }
             } /* end activate edges */
         } /* end loop for enabled nodes */


        /* update the pointers & counters to the enabled array      */
        /**********************************************************/
        curr_enables_nodes = next_enabled_nodes;
        next_enabled_nodes = &(next_enabled_nodes[next_num_enabled]);

        num_enabled = next_num_enabled;
        next_num_enabled = 0;

     } /* while end */

 } /* end FUNCTION */
```

C EMT Code: PAR_DAG.C

```
/*   97/12/12   par_dag.c                      */

/* PARALLEL CODE:                                     */

/* Directed Acyclic Graph (DAG) longest Path         */
/*****************************************************/




void  Parallel_DAG_Path()
{

  /* We use the in_paths array to hold the values of   */
  /* the longest paths provided by the arriving edges  */
  /* and to hold the binary tree used in calculating   */
  /* the maximum of these values.                      */
  /* Each node’s in_paths tree starts at the index     */
  /*     node.in_edges_start * 3                       */
  int     in_paths[NUM_NODES * 3];

  /* START FUNCTION                                   */

  num_enabled = 0;
  next_num_enabled = 0;

  curr_enabled_nodes = &(nodesptr[0]);

  /*  enabled array creation by checking which nodes */
  /*  have ZERO in_edges_num                         */
  /*************************************************/
  spawn (0, num_nodes)
    {
      int  n;

      if (nodes[$].in_edges_num == 0)
       {
         n = 1;
         ps (num_enabled, n);
         curr_enabled_nodes[n] = &(nodes[$});
       }
    }

  next_enabled_nodes = &(curr_enabled_nodes[num_enabled]);


  /* first iteration -- no need to find            */
  /*    max_length_path, just activate out edges   */
  /***********************************************/
  spawn (0, num_enabled)
    {
      NODE *my_node;

      my_node = curr_enabled_nodes[$];
      my_node->max_length_path = MINUS_INFINITY;

      /* "activate" the outgoing edges                          */
      /*******************************************************/
      spawn (0, my_node->out_edges_num)
       {
         EDGE *this_edge;
```

```
        NODE *nextnode;
        int *arr;
        int n = 1;

        /* find index into the next node's in_paths      */
        /* and write "your value" there                   */
        /*************************************************/
        this_edge = &(edges_out[my_node->out_edges_start + $]);
        nextnode = &(nodes[this_edge->end_node]);

        ps (nextnode->in_edge_count, n);
        arr = &(in_paths[3*nextnode->in_edges_start]);
        arr[n] = this_edge->edge_len;


        /* check if last incoming edge -> enable the node */
        /**************************************************/
        if (n == nextnode->in_edges_num)
          {
            /* last to arrive, so enable the node, by    */
            /* copying pointer into enabled nodes array  */
            /*********************************************/
            n = 1;
            ps (next_num_enabled, n);
            next_enabled_nodes[n] = nextnode;
          }
      } /* end activate edges spawn */
    } /* end first nodes layer */


/* update the pointers & counters to the enabled array            */
/******************************************************************/

curr_enabled_nodes = &(curr_enabled_nodes[num_enabled]);
next_enabled_nodes = &(next_enabled_nodes[next_num_enabled]);

num_enabled = next_num_enabled;
next_num_enabled = 0;



/* while there are still enabled nodes -> work   */
/************************************************/
while (num_enabled != 0)
  {
    spawn (0, num_enabled)
     {
        Node *my_node;
        int *arr;
        int *next;
        int size_arr;
        int n;

        my_node = curr_enabled_nodes[$];

        /* find the max length path for CURRENT (my_node) NODE  */
        /*******************************************************/
        size_arr = my_node->in_edges_num;
        arr = &(in_paths[3*nextnode->in_edges_start]);


        /* PARALLEL algorithm                             */
        /* using a binary tree to find the max            */
```

```
/**************************************************/
while (size_arr > 3)
  {
    size_arr = (size_arr + 1)/2;
    next = arr + (size_arr + 1)*sizeof(int);
    arr[size_arr] = MINUS_INFINITY;

    spawn(0, size_arr)
     {
       if (arr[2*$] > arr[2*$ + 1])
         next[$] = arr[2*$];
       else
         next[$] = arr[2*$ + 1];
     }

    arr = next;
  }

/* finish up serially                            */
/************************************************/
if (size_arr == 3)
  {
    if (arr[0] > arr[1])
     {
       if (arr[0] > arr[2])
            my_node->max_length_path = arr[0];
       else
            my_node->max_length_path = arr[2];
     }
    else
     {
       if (arr[1] > arr[2])
         my_node->max_length_path = arr[1];
       else
         my_node->max_length_path = arr[2];
     }
  }
else if (size_arr==2)
  {
    if (arr[0] > arr[1])
     my_node->max_length_path = arr[0];
    else
     my_node->max_length_path = arr[1];
  }
else
  {
    my_node->max_length_path = arr[0];
  }

/* end of maximum calculation */


/* "activate" the outgoing edges                          */
/*****************************************************/
spawn (0, my_node->out_edges_num)
  {
    EDGE *this_edge;
    NODE *nextnode;
    int *arr;
    int n = 1;


    /* find index into the next node’s in_paths      */
```

```
            /* and write "your value" there                   */
            /*************************************************/
            this_edge = &(edges_out[my_node->out_edges_start + $]);
            nextnode = &(nodes[this_edge->end_node]);

            ps (nextnode->in_edge_count, n);
            arr = &(in_paths[3*nextnode->in_edges_start]);
            arr[n] = my_node->max_length_path + this_edge->edge_len;


            /* check if last incoming edge -> enable the node */
            /**************************************************/
            if (n == nextnode->in_edges_num)
             {
               /* last to arrive, so enable the node, by    */
               /* copying pointer into enabled nodes array  */
               /*********************************************/
               n = 1;
               ps (next_num_enabled, n);
               next_enabled_nodes[n] = nextnode;
             }
          } /* end activate edges spawn */
      } /* end enabled nodes spawn */

      /* update the pointers & counters to the enabled array            */
      /*****************************************************************/

      curr_enabled_nodes = &(curr_enabled_nodes[num_enabled]);
      next_enabled_nodes = &(next_enabled_nodes[next_num_enabled]);

      num_enabled = next_num_enabled;
      next_num_enabled = 0;

    } /* end while */

} /* end FUNCTION*/
```

C Spawn MT Code: PAR_DAG_SPAWN-MT.C

```
/*   97/12/1   par_dag_spawn-mt.c                    */
/*                                                   */
/* PARALLEL CODE: Spawn-MT model                     */
/* Directed Acyclic Graph (DAG) longest Path         */
/*                                                   */
/*****************************************************/



void  Parallel_DAG_Path_SpawnMT()
{

  /* We use the in_paths array to hold the values of   */
  /* the longest paths provided by the arriving edges  */
  /* and to hold the binary tree used in calculating   */
  /* the maximum of these values.                      */
  /* Each node’s in_paths tree starts at the index     */
  /*     node.in_edges_start * 3                       */
  int     in_paths[NUM_NODES * 3];


  /* We need the following extra arrays to accomodate  */
  /* the edge information which is derived from the    */
  /* nodes since we must deal with the edges directly  */
  /* without reference to the nodes.                   */
  /*                                                   */
  /*  edge_space maps an edge to the nodes location in */
  /*             the enabled_nodes array               */
  /*  node_space_len holds the segment lengths of the  */
  /*             incoming edges in edge_space          */
  /*  arr_lock   holds the gatekeeper values for the   */
  /*             binary tree used in the async         */
  /*             max computation                       */
  /*****************************************************/

  int     edge_space[NUM_EDGES + 1];

  int     node_space_len[MAX_NUMBER_NODES_IN_A_LAYER*4];
  /* the extra space in node_space_len is for the      */
  /* binary tree used in the prefix-sum calculation    */

  int     arr_lock[NUM_NODES * 2];

  int     *curr_edges;
  int     edge_space_size;

  int n;

  /* These auxiliary pointers are used for general ps */
  /*                                                  */
  /* ps_arr - current read values in the tree         */
  /* ps_next - current write values in the tree       */
  /* ps_old - on the fan-out, points to matching layer*/
  /*       on the fan-in tree                         */
  /* ps_layer_start - ptrs for the layers of the      */
  /*       fan-in tree                                */
  /* ps_sz - sizes of the layers of the fan-in tree   */

  int *ps_arr, *ps_next, *ps_old;
  int ps_layer_start[LOG_NUM_NODES], ps_sz[LOG_NUM_NODES];
```

```
/* START FUNCTION                                    */

num_enabled = 0;
next_num_enabled = 0;

curr_enabled_nodes = &(nodesptr[0]);

curr_edges = &(edge_space[1]);

/* initialize edge_space                    */
/* and arr_lock                             */
spawn (1, num_edges)
  {
    curr_edges[$] = 0;
  }

spawn (0, num_nodes * 2)
  {
    arr_lock[$] = 0;
  }



/*  enabled array creation by checking which nodes */
/*  have ZERO in_edges_num                         */
/*************************************************/
spawn (0, num_nodes)
  {
    NODE *tmpnode;
    int n, m;

    tmpnode = &(nodes[$]);
    if (tmpnode->in_edges_num == 0)
     {
       tmpnode->max_length_path = MINUS_INFINITY;

       n = 1;
       ps(num_enabled, n);
       curr_enabled_nodes[n] = tmpnode;

       m = tmpnode->out_edges_num;
       node_space_len[n] = m;
     }
  }

next_enabled_nodes = &(curr_enabled_nodes[num_enabled]);



/* FIRST LAYER: no need to add max length path value  */
/* to the edge length value when exploring the        */
/* outgoing edges                                     */
/****************************************************/
/* write parent node's index into edge_space  */
/* -- flattening of nesting                   */


/* GENERAL PS: create node_space_start[] from */
/*          node_space_len[]                  */
```

```
ps_arr = &(node_space_len[0]);
ps_arr[num_enabled] = 0;
ps_next = &(node_space_len[num_enabled + 1]);

n = 0;
ps_sz[0] = num_enabled + 1;

while (ps_sz[n] != 1)
  {
    n = n + 1;

    ps_sz[n] = (ps_sz[n-1] + 1)/2;
    ps_next = ps_arr + (ps_sz[n] + 1)*sizeof(int);
    ps_arr[ps_sz[n]] = 0;
    ps_layer_start[n] = ps_next;

    spawn(0, ps_sz[n])
     {
       ps_next[$] = ps_arr[2*$] + ps_arr[2*$ + 1];
     }
    ps_arr = ps_next;
  }

ps_next = ps_arr + sizeof(int);

/* go back down the tree         */
/* stop before last level        */
/* and write to node_space_start */

while (n != 1)
  {
    n = n - 1;

    ps_old = ps_layer_start[n];
    ps_next[0] = ps_old[0];

    spawn(1, ps_sz[n]-1)
     {
       if (($ & 1) == 0)
         ps_next[$] = ps_arr[$/2 - 1] + ps_old[$];
       else
         ps_next[$] = ps_arr[$/2];
     }
    ps_arr = ps_next;
    ps_next = ps_next + sizeof(int) * ps_sz[n];
  }


/* last layer -- put ones in the locations */
/* specified in curr_edges                 */
/* skip first ps val (it is zero) and      */
/* leave room for edge_space_size on end   */

spawn (1, num_enabled)
  {
    int m;

    if (($ & 1) == 0)
     m = ps_arr[$/2 - 1] + node_space_len[$];
    else
     m = ps_arr[$/2];

    curr_edges[m - 1] = 1;
```

```
  }

/* edge_space_size is the last location */
edge_space_size = node_space_len[num_enabled];

n = 0;

spawnsync(0, edge_space_size)
  {
    ps(n, curr_edges[$]);
  }

/* done flattening                           */
/* EXPLORE OUT from all the edges            */
/*********************************************/
spawn (0, edge_space_size)
  {
    NODE *my_node;
    EDGE *this_edge;
    NODE *nextnode;
    int m, n, i;
    int *arr, *lock_arr;
    int lock_val;
    int max_val, comp_val, arr_size;


    my_node = curr_enabled_nodes[curr_edges[$]];

    this_edge = &(edges_out[my_node->out_edges_start +
                       $ - node_space_start[$]]);

    nextnode = &(nodes[this_edge->end_node]);

    max_val  = this_edge->edge_len;

    i = 1;
    ps (nextnode->in_edge_count, i);
    m = i;


    /* maximum calculation -- async                    */
    /***************************************************/

    arr = &(in_paths[3*nextnode->in_edges_start]);
    arr_size = nextnode->in_edges_num;

    lockarr = &(arr_locks[2*nextnode->in_edges_start]);

    while (arr_size!=1)
     {
      /* if not last element in odd size array */
      if (((arr_size&1)==0) || (m!=arr_size-1))
        {
          arr[m] = max_val;

          lock_val = 1;
          ps (lockarr[m>>1], lock_val);

          if (lock_val==0) break;

          n = m - ((m & 1) << 1) - 1;
          comp_val = arr[n];
          if (max_val < comp_val)
```

```
            max_val = comp_val;
          }

        arr = arr + sizeof(int)*arr_size;
        arr_size = arr_size>>1;

        m = m >> 1;

        lockarr = lockarr + sizeof(int)*arr_size;
      }/* end while */

     /* if last edge to arrive, then add node to the  */
     /* enabled array, for the next stage             */
     /* being last also means that the max has been   */
     /* found                                         */
     if (arr_size==1)
      {
        nextnode->max_length_path = max_val;

        /* add to next enabled nodes */
        n = 1;
        ps (next_num_enabled, n);
        next_enabled_nodes[n] = nextnode;

        /* add into node_space    */
        node_space_len[n] = nextnode->out_edges_num;
      }
   } /* end  edges spawn */


/* update the pointers & counters to the enabled array            */
/*****************************************************************/

curr_edges = &(curr_edges[edge_space_size]);

curr_enables_nodes = &(curr_enabled_nodes[num_enabled]);
next_enabled_nodes = &(next_enabled_nodes[next_num_enabled]);

num_enabled = next_num_enabled;
next_num_enabled = 0;






/* while there are still enabled nodes -> work   */
/************************************************/
while (num_enabled != 0)
  {

    /* write parent node's index into edge_space  */
    /* -- flattening of nesting                   */


    /* GENERAL PS: create node_space_start[] from */
    /*          node_space_len[]                  */

    ps_arr = &(node_space_len[0]);
    ps_arr[num_enabled] = 0;
    ps_next = &(node_space_len[num_enabled + 1]);
```

```
n = 0;
ps_sz[0] = num_enabled + 1;

while (ps_sz[n] != 1)
 {
   n = n + 1;

   ps_sz[n] = (ps_sz[n-1] + 1)/2;
   ps_next = ps_arr + (ps_sz[n] + 1)*sizeof(int);
   ps_arr[ps_sz[n]] = 0;
   ps_layer_start[n] = ps_next;

   spawn(0, ps_sz[n])
     {
       ps_next[$] = ps_arr[2*$] + ps_arr[2*$ + 1];
     }
   ps_arr = ps_next;
 }

ps_next = ps_arr + sizeof(int);

/* go back down the tree         */
/* stop before last level        */
/* and write to node_space_start */

while (n != 1)
 {
   n = n - 1;

   ps_old = ps_layer_start[n];
   ps_next[0] = ps_old[0];

   spawn(1, ps_sz[n]-1)
     {
       if (($ & 1) == 0)
        ps_next[$] = ps_arr[$/2 - 1] + ps_old[$];
       else
        ps_next[$] = ps_arr[$/2];
     }
   ps_arr = ps_next;
   ps_next = ps_next + sizeof(int) * ps_sz[n];
 }


/* last layer -- put ones in the locations */
/* specified in curr_edges                 */
/* skip first ps val (it is zero) and      */
/* leave room for edge_space_size on end   */

spawn (1, num_enabled)
 {
   int m;

   if (($ & 1) == 0)
     m = ps_arr[$/2 - 1] + node_space_len[$];
   else
     m = ps_arr[$/2];

   curr_edges[m - 1] = 1;
 }

/* edge_space_size is the last location */
edge_space_size = node_space_len[num_enabled];
```

```
n = 0;

spawnsync(0, edge_space_size)
 {
   ps(n, curr_edges[$]);
 }

/* done flattening                           */
/* EXPLORE OUT from all the edges            */
/*********************************************/
spawn (0, edge_space_size)
 {
   NODE *my_node;
   EDGE *this_edge;
   NODE *nextnode;
   int m, n, i;
   int *arr, *lock_arr;
   int lock_val;
   int max_val, comp_val, arr_size;


   my_node = curr_enabled_nodes[curr_edges[$]];

   this_edge = &(edges_out[my_node->out_edges_start +
                      $ - node_space_start[$]]);

   nextnode = &(nodes[this_edge->end_node]);

   max_val  = my_node->max_length_path + this_edge->edge_len;

   i = 1;
   ps (nextnode->in_edge_count, i);
   m = i;


   /* maximum calculation -- async                   */
   /*************************************************/

   arr = &(in_paths[3*nextnode->in_edges_start]);
   arr_size = nextnode->in_edges_num;

   lockarr = &(arr_locks[2*nextnode->in_edges_start]);

   while (arr_size!=1)
     {
       /* if not last element in odd size array */
       if (((arr_size&1)==0) || (m!=arr_size-1))
        {
          arr[m] = max_val;

          lock_val = 1;
          ps (lockarr[m>>1], lock_val);

          if (lock_val==0) break;

          n = m - ((m & 1) << 1) - 1;
          comp_val = arr[n];
          if (max_val < comp_val)
            max_val = comp_val;
        }

       arr = arr + sizeof(int)*arr_size;
```

```
            arr_size = arr_size>>1;

            m = m >> 1;

            lockarr = lockarr + sizeof(int)*arr_size;
          }/* end while */

        /* if last edge to arrive, then add node to the  */
        /* enabled array, for the next stage             */
        /* being last also means that the max has been   */
        /* found                                         */
        if (arr_size==1)
          {
            nextnode->max_length_path = max_val;

            /* add to next enabled nodes */
            n = 1;
            ps (next_num_enabled, n);
            next_enabled_nodes[n] = nextnode;

            /* add into node_space    */
            node_space_len[n] = nextnode->out_edges_num;
          }
      } /* end  edges spawn */


      /* update the pointers & counters to the enabled array            */
      /*****************************************************************/

      curr_edges = &(curr_edges[edge_space_size]);

      curr_enables_nodes = &(curr_enabled_nodes[num_enabled]);
      next_enabled_nodes = &(next_enabled_nodes[next_num_enabled]);

      num_enabled = next_num_enabled;
      next_num_enabled = 0;

    } /* end while */

} /* end FUNCTION*/
```

ASM Serial Code: SER_DAG.ASM

```
#       97/12/12       ser_dag.asm
#       ASM for Serial DAG algorithm
#       R1      address nodes[0]
#       R2      address edges_in[0]
#       R3      address edges_out[0]
#       R4      num_nodes
#       R5      num_enabled
#       R6      next_num_enabled
#       R7      curr_enabled_nodes
#       R8      next_enabled_nodes
#       R9      nodesptr

#       R28     <scratch>
#       R29     MINUS_INFINITY

#       START FUNCTION
########################################################

        li      R5,     0
        li      R6,     0
        move    R7,     R9

        li      R29,    MINUS_INFINITY

#       create enabled array by checking nodes
#       R11     step thru memory == my_node
#       R12     finish
#       R13     <scratch>
#########################################################
        move    R11,    R1
        multi   R12,    R4,     SZ_NODE
        add     R12,    R12,    R1
START_LOOP1:
        beq     R11,    R12,    END_LOOP1
        lw      R13,    in_edges_num_OFF(R11)
        sw      R29,    max_length_path_OFF(R11)

        beq     R13,    R0,     NOT_STARTING_NODE

        multi   R13,    R5,     SZ_NODE_PTR
        add     R13,    R13,    R1
        sw      R11,    0(R13)
        addi    R5,     R5,     1

NOT_STARTING_NODE:
        addi    R11,    SZ_NODE
        j       START_LOOP1
END_LOOP1:




#       first loop
#       R12     my_node->out_edges_num
#       R13     my_node->out_edges_start
#       R20     step thru memory == my_node
#       R21     finish
##########################################################
        move    R20,    R7
        multi   R21,    R5,     SZ_NODE_PTR
```

```
        add     R21,    R21,    R7

START_LOOP2:
        beq     R20,    R21,    END_LOOP2

        lw      R12,    out_edges_num_OFF(R20)
        lw      R13,    out_edges_start_OFF(R20)

#       ACTIVATE the outgoing edges
#       R14     nextnode
#       R15     nextnode->max_length_path
#       R16     nextnode->in_edge_count
#       R17     nextnode->in_edges_num
#       R18     r, this_edge->edge_len
#       R19     <scratch>
#       R22     step thru memory == this_edge
#       R23     finish
#############################################################
        multi   R22,    R13,    SZ_EDGE
        add     R22,    R22,    R3
        multi   R23,    R12,    SZ_INT
        add     R23,    R23,    R22

START_LOOP3:
        beq     R22,    R23,    END_LOOP3

        lw      R19,    end_node_OFF(R22)
        lw      R18,    edge_len_OFF(R22)
        multi   R19,    R19,    SZ_NODE
        add     R14,    R1,     R19

        lw      R15,    max_length_path_OFF(R14)
        lw      R16,    in_edge_count(R14)
        lw      R17,    in_edges_num_OFF(R14)

        sgt     R19,    R18,    R15
        bne     R19,    R0,     NOT_NEW_MAX1
        sw      R18,    max_length_path_OFF(R14)
NOT_NEW_MAX1:

        add     R16,    R16,    1
        sw      R16,    in_edge_count(R14)

#       check if last incoming edge
##############################################################
        bne     R16,    R17,    NOT_LAST_INCOMING_EDGE1
        multi   R19,    R6,     SZ_NODE_PTR
        add     R19,    R19,    R8
        sw      R14,    0(R19)
        addi    R6,     R6,     1

NOT_LAST_INCOMING_EDGE1:
        addi    R22,    R22,    SZ_EDGE
        j       START_LOOP3
END_LOOP3:

#       END enabled loop
#############################################################

        addi    R20,    R20,    SZ_NODE_PTR
        j       START_LOOP2
END_LOOP2:
```

```
#      move curr and next ptrs
##############################################################
       move   R7,    R8
       multi  R28,   R6,    SZ_NODE_PTR
       add    R8,    R8,    R28

       move   R5,    R6
       li     R6,    0


START_WHILE:
       beq    R5,    R0,    END_WHILE

#      ENABLED nodes loop
#      R11    my_node->max_length_path
#      R12    my_node->out_edges_num
#      R13    my_node->out_edges_start
#      R20    step thru memory == my_node
#      R21    finish
###########################################################
       move   R20,   R7
       multi  R21,   R5,    SZ_NODE_PTR
       add    R21,   R21,   R20

START_LOOP4:
       beq    R20,   R21,   END_LOOP4

       lw     R11,   max_length_path_OFF(R20)
       lw     R12,   out_edges_num_OFF(R20)
       lw     R13,   out_edges_start_OFF(R20)

#      ACTIVATE the outgoing edges
#      R14    nextnode
#      R15    nextnode->max_length_path
#      R16    nextnode->in_edge_count
#      R17    nextnode->in_edges_num
#      R18    r, this_edge->edge_len
#      R19    <scratch>
#      R22    step thru memory == this_edge
#      R23    finish
##############################################################
       multi  R22,   R13,   SZ_EDGE
       add    R22,   R22,   R3
       multi  R23,   R12,   SZ_INT
       add    R23,   R23,   R22

START_LOOP5:
       beq    R22,   R23,   END_LOOP5

       lw     R19,   end_node_OFF(R22)
       lw     R18,   edge_len_OFF(R22)
       multi  R19,   R19,   SZ_NODE_PTR
       add    R14,   R1,    R19

       lw     R15,   max_length_path_OFF(R14)
       lw     R16,   in_edge_count(R14)
       lw     R17,   in_edges_num_OFF(R14)

       add    R18,   R18,   R11

       sgt    R19,   R18,   R15
       bne    R19,   R0,    NOT_NEW_MAX2
```

```
        sw      R18,    max_length_path_OFF(R14)
NOT_NEW_MAX2:

        add     R16,    R16,    1
        sw      R16,    in_edge_count(R14)

#       check if last incoming edge
###############################################################
        bne     R16,    R17,    NOT_LAST_INCOMING_EDGE2
        multi   R19,    R6,     SZ_NODE_PTR
        add     R19,    R19,    R8
        sw      R14,    0(R19)
        addi    R6,     R6,     1

NOT_LAST_INCOMING_EDGE2:
        addi    R22,    R22,    SZ_EDGE
        j       START_LOOP5
END_LOOP5:

#       END enabled loop
##############################################################

        addi    R20,    R20,    SZ_NODE_PTR
        j       START_LOOP4
END_LOOP4:

#       move curr and next ptrs
##############################################################
        move    R7,     R8
        multi   R28,    R6,     SZ_NODE_PTR
        add     R8,     R8,     R28

        move    R5,     R6
        li      R6,     0

        j       START_WHILE
END_WHILE:



#       END FUNCTION
##########################################
```

ASM EMT Code: PAR_DAG.ASM

```
#       97/08/07        par_dag.asm
#       ASM for Parallel DAG algorithm
#       R1      address nodes[0]
#       R2      address edges_out[0]
#       R3      address in_paths[0]
#       R4      num_nodes
#       R5      num_enabled
#       R6      next_num_enabled
#       R7      curr_enabled_nodes
#       R8      next_enabled_nodes
#       R9      nodesptr

#       R21     1
#       R22     2
#       R23     3
#       R29     MINUS_INFINITY
#       R30     spawn count reg

#       START FUNCTION
########################################################

        li      R5,     0
        li      R6,     0
        move    R7,     R9
        li      R29,    MINUS_INFINITY
        li      R21,    1
        li      R22,    2
        li      R23,    3

#       create enabled array by checking nodes
#########################################################
        spawn   R30,    0,      R4,     3
        laa     R1$,    0(R1), SZ_NODE[R0$]
        lw      R2$,    in_edges_num_OFF(R1$)
        beq     R2$,    R0,     END_JOIN1
        li      R3$,    1
        ps      R5,     R3$
        swa     R1$,    0(R7), SZ_NODE_PTR[R3$]

ENDJOIN1:
        join    R30,    R4

        laa     R8,     0(R7), SZ_NODE_PTR[R5]

#       first loop -- no need for max check
#       R1$     my_node
#       R2$     my_node->out_edges_num
#       R3$     my_node->out_edges_start
#       R4$     <scratch>
#       R5$     <scratch>
#       R6$     spawn count register (nested spawns)
############################################################
        spawn   R30,    0,      R5,     6

        lwa     R1$,    0(R7), SZ_NODE_PTR[R0$]
        sw      R29,    max_length_path_OFF(R1$)

        lw      R2$,    out_edges_num(R1$)
        lw      R3$,    out_edges_start(R1$)
```

```
#       spawn
#       R1$     nextnode
#       R2$     this_edge
#       R3$     n
#       R4$     nextnode->in_edges_start
#       R5$     nextnode->in_edge_count
#       R6$     nextnode->in_edges_num
#       R7$     this_edge->edge_len
#       R8$     arr
##############################################################
        spawn   R6$,    0,      R2$,    8

        li      R3$,    1
        add     R8$,    R3$P,   R0$
        laa     R2$,    0(R2), SZ_EDGE[R8$]
        lw      R8$,    end_node_OFF(R2$)

        lw      R7$,    edge_len_OFF(R2$)
        laa     R1$,    0(R1), SZ_NODE[R8$]

        lw      R4$,    in_edges_start_OFF(R1$)
        lw      R6$,    in_edges_num_OFF(R1$)

        psm     in_edge_count_OFF(R1$),     R3$
        laa     R8$,    0(R3), 3*SZ_INT[R4$]
        swa     R7$,    0(R8$),         SZ_INT[R3$]

#       check if last incoming edge
###############################################################
        bne     R3$,    R6$,    NOT_LAST_INCOMING_EDGE1
        li      R3$,    1
        ps      R6,     R3$
        swa     R1$,    0(R8), SZ_NODE_PTR[R3$]


NOT_LAST_INCOMING_EDGE1:
        join    R6$,    R2$

#       END enabled spawn
##############################################################
        join    R30,    R5


#       move curr and next ptrs
##############################################################
        laa     R7,     0(R7), SZ_NODE_PTR[R5]
        laa     R8,     0(R8), SZ_NODE_PTR[R6]

        move    R5,     R6
        li      R6,     0




START_WHILE:
        beq     R5,     R0,     END_WHILE

#       do spawn for enabled nodes
#       R1$     my_node
#       R2$     arr
#       R3$     next
#       R4$     size_arr
#       R5$     n
```

```
#      R6$    spawn count register (nested spawns)
#      R7$    my_node->max_length_path
#      R8$    my_node->in_edge_start, <scratch>
#      R9$    <scratch>
#      R10$   <scratch>
################################################################
       spawn  R30,   0,     R5,    10

       lwa    R1$,   0(R7), SZ_NODE_PTR[R0$]
       lw     R4$,   in_edges_num_OFF(R1$)

       lw     R8$,   in_edge_start_OFF(R1$)
       laa    R2$,   0(R3), 3*SZ_INT[R8$]


#      while loop for binary tree
##################################################################
START_BINARY_WHILE:
       sgti   R8$,   R4$,   3
       bne    R8$,   R0,    DONE_BINARY_WHILE
       addi   R4$,   R4$,   1
       sri    R4$,   R4$,   1
       laa    R3$,   SZ_INT(R2$),  SZ_INT[R4$]
       swa    R29,   0(R2$),       SZ_INT[R4$]

#      nested spawn registers:
#      R1$    arr[2*$]
#      R2$    arr[2*$ + 1]
#      R3$    comparison reg
#################################################################
       spawn  R4$,   0,     R6$,   3
       lwa    R1$,   0(R2P$),      2*SZ_INT[R0$]
       lwa    R2$,   SZ_INT(R2P$), 2*SZ_INT[R0$]
       sgt    R3$,   R1$,   R2$
       beq    R3$,   R0,    SECOND_IS_GREATER2

       swa    R1$,   0(R3P$),      SZ_INT[R0$]
       j      ENDJOIN3

SECOND_IS_GREATER2:
       swa    R2$,   0(R3P$),      SZ_INT[R0$]

ENDJOIN3:
       join   R4P$,  R6P$

#      arr = next
##################################################################
       move   R2$,   R3$
       j      START_BINARY_WHILE
DONE_BINARY_WHILE:


#      finish up using serial method
#################################################################
       bne    R4$,   R23,   TRY_TWO
DO_THREE:
       lw     R7$,   0(R2$)
       lw     R8$,   SZ_INT(R2$)
       lw     R9$    2*SZ_INT(R2$)
       sgti   R10$,  R7$,   R8$
       beq    R10$,  R21,   ARR_0_IS_GREATER

#      arr[1] is greater -- keep checking
```

```
###########################################
       sgti   R10$,  R8$,   R9$
       beq    R10$,  R21,   ARR_1_IS_GREATEST
       move   R7$,   R9$
       j      DONE_WHOLE_SEARCH

ARR_1_IS_GREATEST:
       move   R7$,   R8$
       j      DONE_WHOLE_SEARCH

ARR_0_IS_GREATER:
       sgti   R10$,  R7$,   R9$
       beq    R10$,  R21,   DONE_WHOLE_SEARCH
       move   R7$,   R9$
       j      DONE_WHOLE_SEARCH


TRY_TWO:
       bne    R4$,   R22,   DO_ONE
       lw     R7$,   0(R2$)
       lw     R8$,   SZ_INT(R2$)
       sgti   R9$,   R7$,   R8$
       beq    R9$,   R21,   DONE_WHOLE_SEARCH
       move   R7$,   R8$
       j      DONE_WHOLE_SEARCH



DO_ONE:
       lw     R7$,   0(R2$)


DONE_WHOLE_SEARCH:
#      store my_node->max_length_path
###############################################################
       sw     R7$,   max_length_path_OFF(R1$)



#      ACTIVATE the outgoing edges
#      R1$    my_node
#      R2$    my_node->out_edges_num
#      R3$    my_node->out_edges_start
#      R4$    <scratch>
#      R5$    <scratch>
#      R6$    spawn count register (nested spawns)
#      R7$    my_node->max_length_path
##############################################################
       lw     R2$,   out_edges_num(R1$)
       lw     R3$,   out_edges_start(R1$)

#      spawn
#      R1$    nextnode
#      R2$    this_edge
#      R3$    n
#      R4$    nextnode->in_edges_start
#      R5$    nextnode->in_edge_count
#      R6$    nextnode->in_edges_num
#      R7$    this_edge->edge_len
#      R8$    arr
###############################################################
       spawn  R6$,   0,     R2$,   8
```

```
	li	R3$,	1
	add	R8$,	R3$P,	R0$
	laa	R2$,	0(R2), SZ_EDGE[R8$]
	lw	R8$,	end_node_OFF(R2$)

	lw	R7$,	edge_len_OFF(R2$)
	laa	R1$,	0(R1),		SZ_NODE[R8$]

	lw	R4$,	in_edges_start_OFF(R1$)
	lw	R6$,	in_edges_num_OFF(R1$)

	add	R7$,	R7$,	R7$P

	psm	in_edge_count_OFF(R1$),	R3$
	laa	R8$,	0(R3), 3*SZ_INT[R4$]
	swa	R7$,	0(R8$),		SZ_INT[R3$]

#	check if last incoming edge
##############################################################
	bne	R3$,	R6$,	NOT_LAST_INCOMING_EDGE2
	li	R3$,	1
	ps	R6,	R3$
	swa	R1$,	0(R8), SZ_NODE_PTR[R3$]

NOT_LAST_INCOMING_EDGE2:
	join	R6$,	R2$

#	END enabled spawn
#############################################################
	join	R30,	R5


#	move curr and next ptrs
#############################################################
	laa	R7,	0(R7), SZ_NODE_PTR[R5]
	laa	R8,	0(R8), SZ_NODE_PTR[R6]

	move	R5,	R6
	li	R6,	0

	j	START_WHILE
END_WHILE:


#	END FUNCTION
##########################################
```

ASM Spawn MT Code: PAR_DAG_SPAWN-MT.ASM

```
#      97/12/2             par_dag_spawn-mt.asm
#      ASM for Parallel DAG algorithm
#          using SPAWN-MT model


#      R1     address nodes[0]
#      R2     address edges_out[0]
#      R3     address in_paths[0]
#      R4     num_nodes
#      R5     num_enabled
#      R6     next_num_enabled
#      R7     curr_enabled_nodes
#      R8     num_edges, next_enabled_nodes
#      R9     nodesptr
#      R10    curr_edges
#      R11    edge_space_size
#      R12    arr_locks
#      R13    node_space_len
#      R14    node_space_start
#      R15    <scratch>

#      R16    ps_arr
#      R17    ps_next
#      R18    ps_old
#      R19    ps_layer_start
#      R20    ps_sz

#      R27    n
#      R28    1
#      R29    MINUS_INFINITY
#      R30    spawn count reg



#      START FUNCTION
########################################################

       li     R5,    0
       li     R6,    0
       move   R7,    R9

       li     R10,   edge_space_BASE + SZ_INT
       li     R28,   1
       li     R29,   MINUS_INFINITY


#      initialize edge_space & arr_lock
#############################################
       spawn  R30,   0,     R8,    0
       swa    R0,    0(R10),       SZ_INT[R0$]
       join   R30,   R8

       sli    R15,   R4,    1
       spawn  R30,   0,     R15,   0
       swa    R0,    0(R12),       SZ_INT[R0$]
       join   R30,   R15

#      create enabled array by checking nodes
#      R1$    tmpnode
#      R2$    tmpnode->in_edges_num
```

```
#      R3$    n
#      R4$    m
##########################################################
       spawn  R30,   0,     R4,    4
       laa    R1$,   0(R1), SZ_NODE[R0$]
       lw     R2$,   in_edges_num_OFF(R1$)

       beq    R2$,   R0,    END_JOIN1

       sw     R29,   max_length_path_OFF(R1$)

       li     R3$,   1
       ps     R5,    R3$
       swa    R1$,   0(R7), SZ_NODE_PTR[R3$]

       lw     R4$,   out_edges_num_OFF(R1$)
       swa    R4$,   0(R13),      SZ_INT[R3$]

ENDJOIN1:
       join   R30,   R4


       laa    R8,    0(R7), SZ_NODE_PTR[R5]



# IMPORTANT NOTE:
#
# In order to improve readability of this already long
# assembly code program, we skip writing a portion of
# the program.  The portion skipped is an exact copy
# of the main loop with the modification that the line
# of code is skipped where the maximum path value is added
# to the edge length.  This value is entered into
# the binary tree of the destination node in order to find
# the maximum path length for that node.  In the case of
# the first layer, there is no maximum path value so
# only the edge length is used. Therefore, the first layer
# is done as a separate (modified) version of the main
# loop.  (See the C code for the C version and the EMT
# code for a similar example.)
#
#########################################################


START_WHILE:
       beq    R5,    R0,    END_WHILE


#      flatten nesting
##########################################################


#      GENERAL PS: node_space_len to node_space_start
#
#      R16    ps_arr
#      R17    ps_next
#      R18    ps_old
#      R19    ps_layer_start
#      R20    ps_sz
#      R21    ps_sz[n]
#      R27    n
############################################################
```

```
        move   R16,   R18
        sw     R0,    0(R18),       SZ_INT[R5]
        laa    R17,   SZ_INT(R18),  SZ_INT[R5]

        li     R27,   0
        addi   R21,   R5,    1
#       no need to store ps_sz[0] to mem -- never used

#       start  going up the tree
############################################################
START_PS_UP:
        beq    R21,   R28,   END_PS_UP
        addi   R27,   R27,   1

        addi   R21,   R21,   1
        sri    R21,   R21,   1
        swa    R21,   0(R20),       SZ_INT[R27]

        laa    R17,   SZ_INT(R16),  SZ_INT[R21]
        swa    R0,    0(R16),       SZ_INT[R21]
        swa    R17,   0(R19),       SZ_INT[R27]

#       spawn to add the layer
#       R1$    ps_arr[2$], ps_next[$]
#       R2$    ps_arr[2$ + 1]
##########################################################
        spawn  R30,   0,     R21,   2
        lwa    R1$,   0(R16),       2*SZ_INT[R0$]
        lwa    R2$,   SZ_INT(R16),  2*SZ_INT[R0$]
        add    R1$,   R1$,   R2$
        swa    R1$,   0(R17),       SZ_INT[R0$]
        join   R30,   R21

        move   R16,   R17
        j      START_PS_UP
END_PS_UP:

        addi   R17,   R16,   SZ_INT

#       go back down the tree
############################################################
START_PS_DOWN:
        beq    R27,   R28,   END_PS_DOWN
        subi   R27,   R27,   1

        lwa    R18,   0(R19),       SZ_INT[R27]
        lw     R15,   0(R18)
        sw     R15,   0(R17)

        lwa    R21,   0(R20),       SZ_INT[R27]
        subi   R21,   R21,   1

#       spawn for the layer
#       R1$    $&1, ps_old[$]
#       R2$    $/2, ps_arr[$/2] or ps_arr[$/2 - 1], ps_next[$]
##########################################################
        spawn  R30,   1,     R21,   2
        andi   R1$,   R0$,   1
        sri    R2$,   R0$,   1

        bne    R1$,   R0,    IF_ODD_IN_PS_DOWN
```

```
        lwa    R1$,   0(R18),        SZ_INT[R0$]
        lwa    R2$,   -SZ_INT(R16),  SZ_INT[R2$]
        add    R2$,   R2$,   R1$
        j      IF_END_IN_PS_DOWN

IF_ODD_IN_PS_DOWN:
        lwa    R2$,   0(R16),        SZ_INT[R2$]

IF_END_IN_PS_DOWN:

        swa    R2$,   0(R17),        SZ_INT[R0$]
        join   R30,   R21


        move   R16,   R17
        laa    R17,   SZ_INT(R17),   SZ_INT[R21]

        j      START_PS_DOWN

END_PS_DOWN:



#       last layer --
#       put 1's to delimit segments
#
#       R1$    $&1, node_space_len[$]
#       R2$    $/2, ps_arr[$/2] or ps_arr[$/2 - 1], m
#
##########################################################
        spawn  R30,   1,     R5,    2
        andi   R1$,   R0$,   1
        sri    R2$,   R0$,   1

        bne    R1$,   R0,    IF_ODD_IN_PS_DOWN2

        lwa    R1$,   0(R13),        SZ_INT[R0$]
        lwa    R2$,   -SZ_INT(R16),  SZ_INT[R2$]
        add    R2$,   R2$,   R1$
        j      IF_END_IN_PS_DOWN

IF_ODD_IN_PS_DOWN2:
        lwa    R2$,   0(R16),        SZ_INT[R2$]

IF_END_IN_PS_DOWN2:

        swa    R28,   -SZ_INT(R10),  SZ_INT[R2$]
        join   R30,   R5



#       edge_space_size is the last location
        lwa    R11,   0(R13),        SZ_INT[R5]

        li     R27,   0


#       now do ps to get parent node's index
#       R1$    curr_edges[$]
############################################################
        spawnsync     R30,   0,     R11,   1
        lwa    R1$,   0(R14),        SZ_INT[R0$]
        ps     R27,   R1$
```

```
        swa     R1$,    0(R14) SZ_INT[R0$]
        join    R30,    R11


#       done flattening
#
#       EXPLORE OUT from the edges
############################################################
#       R1$     curr_edges[$], my_node->max_length_path
#       R2$     my_node, arr
#       R3$     this_edge, n
#       R4$     node_space_start[$], nextnode
#       R5$     i = nextnode->in_edge_count
#       R6$     m
#       R7$     nextnode->in_edge_start, lock_arr
#       R8$     nextnode->in_edge_num,     arr_size
#       R9$     my_node->out_edges_start, max_val
#       R10$    this_edge->edge_len, <scratch>
#       R11$    lock_val and comp_val and <scratch>
############################################################
        spawn   R30,    0,      R11,    11

        lwa     R1$,    0(R10),         SZ_INT[R0$]
        lwa     R2$,    0(R7), SZ_NODE_PTR[R1$]
        lw      R9$,    out_edges_start_OFF(R2$)

        lwa     R4$,    0(R14),         SZ_INT[R0$]

        add     R3$,    R9$,    R0$
        sub     R3$,    R3$,    R4$
        laa     R3$,    0(R2), SZ_EDGE[R3$]

        lw      R4$,    end_node_OFF(R3$)
        laa     R4$,    0(R1$),         SZ_NODE[R4$]

#       NOTE: As explained above, in the first layer
#       of the loop, this code would exist in a modified
#       version. The next line would read:
#       lw      R9$,    edge_len_OFF(R3$)
#       and the line add R9$, R1$, R10$ would be
#       eliminated

        lw      R10$,   edge_len_OFF(R3$)
        lw      R7$,    in_edge_start_OFF(R4$)
        lw      R8$,    in_edge_num_OFF(R4$)
        lw      R1$,    max_length_path_OFF(R2$)

        add     R9$,    R1$,    R10$

        li      R5$,    1
        psm     in_edge_count_OFF(R4$),     R5$

        move    R6$,    R5$


#       maximum calculation -- async
############################################################
        laa     R2$,    0(R3), 3*SZ_INT[R7$]
        laa     R7$,    0(R12),         2*SZ_INT[R7$]


#       while loop to go up the binary tree
############################################################
```

```
START_BINARY_TREE_WHILE:
       bne    R8$,   R28,   END_BINARY_TREE_WHILE

       andi   R11$,  R8$,   1
       beq    R11$,  R0,    TRY_LOCK

       subi   R11$,  R8$,   1
       beq    R11$,  R6$,   DONE_TRY_LOCK

TRY_LOCK:
       swa    R9$,   0(R2$),       SZ_INT[R6$]

       li     R11$,  1
       sri    R10$,  R6$,   1
       laa    R10$,  0(R7$),       SZ_INT[R10$]
       psm    0(R10$),R11$

       beq    R11$,  R28,   END_BINARY_TREE_WHILE

       andi   R3$,   R6$,   1
       sli    R3$,   R3$,   1
       laa    R3$,   -1(R0),       -1[R3$]

       lwa    R11$,  0(R2$),       SZ_INT[R3$]
       slt    R10$,  R9$,   R11$
       beq    R10$,  R0,    DONE_TRY_LOCK
       move   R9$,   R11$
DONE_TRY_LOCK:

       laa    R2$,   0(R2$),       SZ_INT[R8$]
       sri    R8$,   R8$,   1
       sri    R6$,   R6$,   1
       laa    R7$,   0(R7$),       SZ_INT[R8$]
       j      START_BINARY_TREE_WHILE
END_BINARY_TREE_WHILE:


#      check if last incoming edge
##############################################################
       bne    R18$,  R28,   NOT_LAST_INCOMING_EDGE

       sw     R9$,   max_length_path(R4$)

       li     R5$,   1
       ps     R6,    R5$
       swa    R4$,   0(R8), SZ_NODE_PTR[R5$]

       lw     R6$,   out_edges_num_OFF(R4$)
       swa    R6$,   0(R13),       SZ_INT[R5$]

NOT_LAST_INCOMING_EDGE:

#      END edges spawn
#############################################################
       join   R30,   R11



#      move curr and next ptrs
#############################################################
       laa    R10,   0(R10),       SZ_INT[R11]

       laa    R7,    0(R7), SZ_NODE_PTR[R5]
```

```
        laa     R8,     0(R8), SZ_NODE_PTR[R6]

        move    R5,     R6
        li      R6,     0

        j       START_WHILE
END_WHILE:


#       END FUNCTION
##########################################
```

Appendix B: Time and Work Analysis

This section contains analysis of the three algorithms on a step by step instruction level. For a detailed explanation of the specification of the machine assembly instructions, the specifications of the extended C, and the assumptions used in formulating this analysis, see the Specifications and Assumptions document at the the following website: `http://www.umiacs.umd.edu/~vishkin/EMT/`

Time and Work Analysis of Serial Algorithm

N = Number of nodes
E = Number of edges
L = Number of layers – 1
N_0 = Number of nodes with 0 in edges (start nodes)
E_0 = Number of outgoing edges in N_0
L_2 = Number of nodes in the second layer (cancels out in the final analysis)
N′ = N - N_0
E′ = Number of outgoing edges in N′, i.e. E - E_0

Execution		**Measurement**	
Instruction	**Cycle executed**	**Time**	**Work**
# ASM for Serial DAG algorithm # R1 address nodes[0] # R2 address edges_in[0] # R3 address in_paths[0] # R4 num_nodes # R5 num_enabled # R6 next_num_enabled # R7 curr_enabled_nodes # R8 next_enabled_nodes # R9 nodesptr # R29 MINUS_INFINITY # START FUNCTION ##			
li R5, 0	1		
li R6, 0	1		
move R7, R9	1		
li R30, MINUS_INFINITY	1		
# create enabled array by checking nodes # R11 step thru memory # R12 finish # R13 <scratch> ##			
move R11, R1	1		
multi R12, R4, SZ_NODE_PTR	1		
add R12, R12, R11	2		
BLOCK TOTAL: **Do once**		**2**	**7**

START_LOOP1: beq R11, R12, END_LOOP1	1		
lw R13, in_edges_num_OFF(R11)	1-5	Branch Predict	
sw R29, max_length_path_OFF(R11)	2		
beq R13, R0, NOT_STARTING_NODE	6		
multi R13, R5, SZ_NODE_PTR	7	N_0	N_0
add R13, R13, R1	8	N_0	N_0
sw R11, 0(R13)	9		N_0
addi R5, R5, 1	9	N_0	N_0
NOT_STARTING_NODE: addi R11, SZ_NODE	10		
j START_LOOP1 END_LOOP1:	10		
BLOCK TOTAL: **Do for each node in the graph, i.e. N times** **Do branch for each starting node, i.e. N_0 times**		**7N + $3N_0$**	**6N + $4N_0$**
# first loop -- no need for max check # R12 my_node->out_edges_num # R13 my_node->out_edges_start # R20 step thru memory == my_node # R21 finish ##			
move R20, R7	1		
multi R21, R5, SZ_NODE_PTR	1		
add R21, R21, R20	2		
BLOCK TOTAL: **Do once**		**2**	**3**
START_LOOP2: beq R20, R21, END_LOOP2	1		
lw R12, out_edges_num(R20)	1-5	Branch predict	
lw R13, out_edges_start(R20)	1-5	Branch predict	
# ACTIVATE the outgoing edges # R14 nextnode # R15 nextnode->in_edges_start # R16 nextnode->in_edge_count # R17 nextnode->in_edges_num # R18 this_edge->edge_len # R19 this_edge->end_node, <scratch> # R22 step thru memory == this_edge # R23 finish ##			
multi R22, R13, SZ_EDGE	1		
add R22, R22, R3	2		

multi R23, R12, SZ_INT	1		
add R23, R23, R22	3		
BLOCK TOTAL: **Do for each node in N_0, i.e. N_0 times**		**$3N_0$**	**$7N_0$**
START_LOOP3: beq R22, R23, END_LOOP3	1		E_0
lw R19, end_node_OFF(R22)	1-5	Branch predict	E_0
lw R18, edge_len_OFF(R22)	1-5		E_0
multi R19, R19, SZ_NODE_PTR	6		E_0
add R14, R1, R19	7		E_0
lw R15, in_edges_start_OFF(R14)	8-12		E_0
lw R16, in_edges_count(R14)	8-12		E_0
lw R17, in_edges_num_OFF(R14)	8-12		E_0
sgt R19, R18, R15	13		E_0
bne R19, R0, NOT_NEW_MAX1	14		E_0
sw R18, max_length_path_OFF(R14)	15	Swap max value (assume this will happen half the time)	1/2 E_0
NOT_NEW_MAX1: add R16, R16, 1	15		E_0
sw R16, in_edge_count(R14)	16		E_0
# check if last incoming edge ##			
bne R16, R17, NOT_LAST_INCOMING_EDGE1	16		E_0
multi R19, R6, SZ_NODE_PTR	17	L_2	L_2
add R19, R19, R8	18	L_2	L_2
sw R14, 0(R19)	19	L_2	L_2
addi R6, R6, 1	18		L_2
NOT_LAST_INCOMING_EDGE1: addi R22, R22, SZ_EDGE	17		E_0
j START_LOOP3 END_LOOP3:	17		E_0
BLOCK TOTAL: **Do for each outgoing edge in N_0, i.e. E_0 times**		**$16E_0$ + $3L_2$**	**15.5 E_0 + $4L_2$**

# END enabled loop ##			
addi R20, R20, SZ_NODE_PTR	1		
j START_LOOP2 END_LOOP2:	1		
BLOCK TOTAL: **Do for each node in N_0, i.e. N_0 times**		**N_0**	**2 N_0**
# swap curr and next arrays ##			
move R9, R7	1		
move R7, R8	2		
move R6, R9	3		
# num_enabled = next_num_enabled ##			
move R5, R6	1		
li R6, 0	2		
BLOCK TOTAL: **Do once. This block overlaps the following one by once cycle.**		**2**	**5**
START_WHILE: beq R5, R0, END_WHILE	1		
# R10 arr # R11 my_node->max_length_path # R12 my_node->out_edges_num # R13 my_node->out_edges_start # R14 my_node->in_edges_start # R15 my_node->in_edges_num # R16 <scratch> # R17 <scratch> # R20 step thru memory == my_node # R21 finish ##			
move R20, R7	1		
multi R21, R5, SZ_NODE_PTR	1		
add R21, R21, R20	2		
BLOCK TOTAL: **Do for each layer in the graph (except the first),** **i.e. L times**		**2L**	**4L**
START_LOOP4: beq R20, R21, END_LOOP4	1	Branch predict	
lw R11, max_length_path_OFF(R20)	1-5	Overlap next block	
lw R12, out_edges_num(R20)	1-5	Overlap next block	

lw R13, out_edges_start(R20)	1-5	Overlap next block	
# ACTIVATE the outgoing edges # R14 nextnode # R15 nextnode->in_edges_start # R16 nextnode->in_edge_count # R17 nextnode->in_edges_num # R18 this_edge->edge_len # R19 this_edge->end_node, <scratch> # R22 step thru memory == this_edge # R23 finish ###			
multi R22, R13, SZ_EDGE	1		
add R22, R22, R3	2		
multi R23, R12, SZ_INT	1		
add R23, R23, R22	3		
BLOCK TOTAL: **Do for each node in N', i.e. N' times**		**3N'**	**8N'**
START_LOOP5: beq R22, R23, END_LOOP5	1		E'
lw R19, end_node_OFF(R22)	1-5	Branch predict	E'
lw R18, edge_len_OFF(R22)	1-5		E'
multi R19, R19, SZ_NODE_PTR	6		E'
add R14, R1, R19	7		E'
lw R15, in_edges_start_OFF(R14)	8-12		E'
lw R16, in_edges_count(R14)	8-12		E'
lw R17, in_edges_num_OFF(R14)	8-12		E'
add R18, R18, R11	6		E'
sgt R19, R18, R15	13		E'
bne R19, R0, NOT_NEW_MAX2	14		E'
sw R18, max_length_path_OFF(R14)	15		1/2 E'
NOT_NEW_MAX2: add R16, R16, 1	15		E'
sw R16, in_edge_count(R14)	16		E'
# check if last incoming edge ###			
bne R16, R17, NOT_LAST_INCOMING_EDGE2	16		E'
multi R19, R6, SZ_NODE_PTR	17	$N' - L_2$	$N' – L_2$

add R19, R19, R8	18	N' - L_2	N′ – L_2
sw R14, 0(R19)	19	N' - L_2	N′ – L_2
addi R6, R6, 1	18		N′ – L_2
NOT_LAST_INCOMING_EDGE2: addi R22, R22, SZ_EDGE	17		E′
j START_LOOP5 END_LOOP5:	17		E′
BLOCK TOTAL: **Do for each outgoing edge in N′, i.e. E times**		**16E′ + 3(N′ – L_2)**	**16.5E′ + 4(N′ – L_2)**
# END enabled loop ##			
addi R20, R20, SZ_NODE_PTR	1		N'
j START_LOOP4 END_LOOP4:	1		N'
BLOCK TOTAL: **Do for each node (except start nodes), i.e. N′ times**		**N′**	**2N′**
# swap curr and next arrays ##			
move R9, R7	1		L
move R7, R8	2		L
move R6, R9	3		L
# num_enabled = next_num_enabled ##			
move R5, R6	1		L
li R6, 0	2		L
j START_WHILE END_WHILE:	2		L
# END FUNCTION ##			
BLOCK TOTAL: **Do for each layer (except first), i.e. L times. This block overlaps the following one by one cycle.**		**2L**	**6L**

<u>Total Time:</u>

$2 + (7N + 3\,N_0) + 2 + 3N_0 + (16E_0 + 3L_2) + N_0 + 2 + 2L + 3N' + (16E' + 3(N' - L_2))$
$+ N' + 2L$

	const	N	N0	E0	L2	L	N'	E'
	2	7	3	16	3	2	3	16
	2		3		-3	2	3	
	2		1				1	
Total	6	7	7	16	0	4	7	16

$\mathbf{6 + 7N + 7N_0 + 16E_0 + 4L + 7N' + 16E'}$

<u>Total Work:</u>

$7 + (6N + 4N_0) + 3 + 7N_0 + (15.5\,E_0 + 4L_2) + 2N_0 + 5 + 4L + 8N'$
$+ (16.5E' + 4(N' - L_2)) + 2N' + 6L$

	const	N	N0	E0	L2	L	N'	E'	
	7	6	4	15.5	4	4	8	16.5	
	3		7		-4	6	4		
	5		2				2		
Total	15	6	13	15.5	0	10	14	16.5	

$\mathbf{15 + 6N + 13N_0 + 15.5E_0 + 10L + 14N' + 16.5E'}$

Time and Work Analysis of Parallel EMT Algorithm

N	= Number of nodes
E	= Number of edges
L	= Number of layers - 1
N_0	= Number of nodes with 0 in edges (start nodes)
E_0	= Number of edges outgoing from the start nodes
L_2	= Number of nodes in the second layer
N′	= Number of nodes – start nodes, i.e. $N - N_0$
E′	= Number of outgoing edges in N′, i.e. $E - E_0$
N_1	= Number of nodes with one incoming edge
N_2	= Number of nodes with two incoming edges
N_3	= Number of nodes with three incoming edges
N″	= $N - N_1 - N_2 - N_3 - N_0$
E″	= Number of incoming edges in N″, i.e. $E - E_0 - N_1 - 2N_2 - 3N_3$
D	= Max degree of nodes in N″
N_D	= Sum(log(degreeNode)) over all the nodes in N"

Execution		Measurement	
Instruction	**Cycle executed**	**Time**	**Work**
# ASM for Parallel DAG algorithm # R1 address nodes[0] # R2 address edges_out[0] # R3 address in_paths[0] # R4 num_nodes # R5 num_enabled # R6 next_num_enabled # R7 curr_enabled_nodes # R8 next_enabled_nodes # R9 nodesptr # R21 1 # R22 2 # R23 3 # R29 MINUS_INFINITY # R30 spawn count reg # START FUNCTION ##			
li R5, 0	1		
li R6, 0	1		
move R7, R9	1		
li R29, MINUS_INFINITY	1		
li R21, 1	1		
li R22, 2	1		
li R23, 3	1		
BLOCK TOTAL **Do once. This block can be overlapped with the first instruction of the following block.**		**0**	**7**

# create enabled array by checking nodes ###			
spawn R30, 0, R4, 3	1		1
laa R1$, 0(R1), SZ_NODE[R0$]	2		N
lw R2$, in_edges_num_OFF(R1$)	3-7		N
beq R2$, R0, END_JOIN1	8		N
li R3$, 1	9		N_0
ps R5, R3$	10		N_0
swa R1$, 0(R7), SZ_NODE_PTR[R3$]	11		N_0
ENDJOIN1: join R30, R9	12		N
BLOCK TOTAL **Each node is visited in one spawn; do once (time), N (work)**		**12**	**1 + 4N + 3N_0**
laa R8, 0(R7), SZ_NODE_PTR[R5]	1		1
# first loop – no need for max check # R1$ my_node # R2$ my_node->out_edges_num # R3$ my_node->out_edges_start # R4$ <scratch> # R5$ <scratch> # R6$ spawn count register (nested spawns) ###			
spawn R30, 0, R5, 6	1		1
lwa R1$, 0(R7), SZ_NODE_PTR[R0$]	2-6		N_0
sw R29, max_length_path_OFF(R1$)	7		N_0
lw R2$, out_edges_num(R1$)	7-11		N_0
lw R3$, out_edges_start(R1$)	7-11		N_0
BLOCK TOTAL **Each start node is processed; do once (time), N_0 (work). This block can overlap the next block by one cycle.**		**10**	**2 + 4N_0**
# spawn # R1$ nextnode # R2$ this_edge # R3$ n # R4$ nextnode->in_edges_start # R5$ nextnode->in_edge_count # R6$ nextnode->in_edges_num # R7$ this_edge->edge_len # R8$ <scratch> ###			
spawn R6$, 0, R2$, 8	1		N_0
li R3$, 1	2		E_0
add R8$, R3$P, R0$	2		E_0

laa R2$, 0(R2), SZ_EDGE[R8$]	3		E_0
lw R8$, end_node_OFF(R2$)	4-8		E_0
lw R7$, edge_len_OFF(R2$)	4-8		E_0
laa R1$, 0(R1), SZ_NODE[R8$]	9		E_0
lw R4$, in_edges_start_OFF(R1$)	10-14		E_0
lw R6$, in_edges_num_OFF(R1$)	10-14		E_0
psm in_edge_count_OFF(R1$), R3$	10-14		E_0
add R8$, R3$, R4$	15		E_0
swa R7$, 0(R3), SZ_INT[R8$]	16		E_0
# check if last incoming edge ###			
bne R3$, R6$, NOT_LAST_INCOMING_EDGE1	16		E_0
li R3$, 1	17		L_2
ps R6, R3$	18		L_2
swa R1$, 0(R8), SZ_NODE_PTR[R3$]	19		L_2
NOT_LAST_INCOMING_EDGE1: join R6$, R2$	20		E_0
BLOCK TOTAL **Each outgoing edge in N_0 is processed; do once (time), do all outgoing edges in N_0. The instructions in the branching are executed the number of nodes in the second layer (work)**		**20**	**N_0 + $13E_0$ + $3L_2$**
# END enabled spawn ###			
join R30, R5	1		N_0
BLOCK TOTAL **Each starting node is processed; do once (time), N_0 (work)**		**1**	**N_0**
# move curr and next ptrs ###			
laa R7, 0(R7), SZ_NODE_PTR[R5]	1		1
laa R8, 0(R8), SZ_NODE_PTR[R6]	1		1
move R5, R6	2		1
li R6, 0	3		1
BLOCK TOTAL **Do once**		**3**	**4**
START_WHILE: beq R5, R0, END_WHILE	1		L
BLOCK TOTAL **Do for each layer in the graph (except the first one), i.e. L times**		**L**	**L**

# do spawn for enabled nodes # R1$ my_node # R2$ arr # R3$ next # R4$ size_arr # R5$ n # R6$ spawn count register (nested spawns) # R7$ my_node->max_length_path # R8$ my_node->in_edge_start, <scratch> # R9$ <scratch> # R10$ <scratch> ###			
spawn R30, 0, R5, 10	1		L
lwa R1$, 0(R7), SZ_NODE_PTR[R0$]	2-6		N’
lw R4$, in_edges_num_OFF(R1$)	7-11		N’
lw R8$, in_edge_start_OFF(R1$)	7-11		N’
laa R2$, 0(R3), 3*SZ_INT[R8$]	12		N’
BLOCK TOTAL **Each node (except starting nodes) is processed; do L times (time), do N’ times (work)**		**12L**	**L+ 4N’**
# while loop for binary search ###			
START_BINARY_WHILE: sgti R8$, R4$, 3	1		N_D - N" + N_1 +N_2 +N_3
bne R8$, R0, DONE_BINARY_WHILE	2	Branch predict	N_D - N" + N_1 +N_2 +N_3
addi R4$, R4$, 1	2		N_D - N"
sri R4$, R4$, 1	3		N_D - N"
laa R3$, SZ_INT(R2$), SZ_INT[R4$]	4		N_D - N"
swa R29, 0(R2$), SZ_INT[R4$]	4		N_D - N"
BLOCK TOTAL **For each node in N’, so long as size_arr > 4, we do the loop.** **So the loop is done log(degree of node) - 1 times (since we do not do the last layer of the binary tree) i.e. log(degree/2). Taking the upper bound for the time, this takes L*log(D/2) [maximum degree] (time).** **For the work, Sum(log(degreeNode) - 1) times, summed over all the nodes in N" or N_D - N" for the block. The first two lines are done for all nodes N_1, N_2, and N_3 also.** **This block can overlap the following block by once cycle.**		**3L log(D/2)**	**6(N_D - N") + 2N_1 + 2N_2 + 2N_3**

# nested spawn registers: # R1$ arr[2*$] # R2$ arr[2*$ + 1] # R3$ comparison reg ##			
spawn R4$, 0, R6$, 3	1		N_D - N''
lwa R1$, 0(R2P$), 2*SZ_INT[R0$]	2-6		E" - 2N'' + N_D
lwa R2$, SZ_INT(R2P$), 2*SZ_INT[R0$]	2-6		E" - 2N'' + N_D
sgt R3$, R1$, R2$	7		E" - 2N'' + N_D
beq R3$, R0, SECOND_IS_GREATER2	8		E" - 2N'' + N_D
swa R1$, 0(R3P$), SZ_INT[R0$]	9	Assume half are greater	½ (E" - 2N'' + N_D)
j ENDJOIN3	9		½ (E" - 2N'' + N_D)"
SECOND_IS_GREATER2: swa R2$, 0(R3P$), SZ_INT[R0$]			½ (E" - 2N'' + N_D)
ENDJOIN3: join R4P$, R6P$	10		E" - 2N'' + N_D
BLOCK TOTAL **For work, we process the incoming edges in each node in N'', as we go up a binary tree, doing ceil(½d) + ceil(¼d) + ceil(1/8d) + … + 4 work. The ceil function comes from the fact that we round up to an even number of threads in this spawn—therefore the work time here is an upper bound; thus, the total work per node is d - 2 + log(d).** **So the total work is E'' (for the summation of the geometric series of d) - 2N' ' (for stopping the series before the last two rounds) + N_D (for the ceil function)** **The time is the same as the number of rounds so do L*log(D/2) [maximum degree] (time).**		**10L log(D/2)**	**13/2 (E" - 2N'' + N_D) + N_D -N''**
# arr = next ##			
move R2$, R3$	1		N_D - N''
j START_BINARY_WHILE DONE_BINARY_WHILE:	1		N_D - N''
BLOCK TOTAL **For each node in N"; do L*log(D/2) times (time), do N_D - N'' times (work)**		**L log(D/2)**	**2N_D - 2N"**

# finish up using serial method ###			
bne R4$, R23, TRY_TWO	1		N′
DO_THREE: lw R7$, 0(R2$)	2-6		½ N″ + N_3
lw R8$, SZ_INT(R2$)	2-6		½ N″ + N_3
lw R9$ 2*SZ_INT(R2$)	2-6		½ N″ + N_3
sgti R10$, R7$, R8$	7		½ N″ + N_3
beq R10$, R21, ARR_0_IS_GREATER2	8		½ N″ + N_3
# arr[1] is greater -- keep checking ###			
sgti R10$, R8$, R9$	9		1/2 (½ N″ + N_3)
beq R10$, R21, ARR_1_IS_GREATEST2	10		1/2 (½ N″ + N_3)
move R7$, R9$	11		1/4 (½ N″ + N_3)
j DONE_WHOLE_SEARCH	11		1/4 (½ N″ + N_3)
ARR_1_IS_GREATEST2: move R7$, R8$			1/4 (½ N″ + N_3)
J DONE_WHOLE_SEARCH			1/4 (½ N″ + N_3)
ARR_0_IS_GREATER2: sgti R10$, R7$, R9$			1/2 (½ N″ + N_3)
beq R10$, R21, DONE_WHOLE_SEARCH			1/2 (½ N″ + N_3)
move R7$, R9$			1/4 (½ N″ + N_3)
J DONE_WHOLE_SEARCH			1/4 (½ N″ + N_3)
TRY_TWO:			
bne R4$, R22, DO_ONE			N' - 1/2N" - N_3
lw R7$, 0(R2$)			(½ N″ + N_2)
lw R8$, SZ_INT(R2$)			(½ N″ + N_2)
sgti R9$, R7$, R8$			(½ N″ + N_2)
beq R9$, R21, DONE_WHOLE_SEARCH			(½ N″ + N_2)

move R7$, R8$			1/2(½ N″ + N_2)
j DONE_WHOLE_SEARCH			1/2(½N″ + N_2)
DO_ONE: lw R7$, 0(R2$)			N_1
DONE_WHOLE_SEARCH:			
BLOCK TOTAL **This block is done for each node in N'. The various work counts reflect the number of nodes which have N_1, N_2, N_3 nodes and assuming that half the N'' nodes finish with 3 values left to find after the previous block and the other half have 2 values left.** **assumption: equal probability of following the branches in the various maximum calculations**		**11 L**	**2N' + 6.25N'' + 7.5N_3 + 5N_2 + N_1**
# store my_node->max_length_path ###			
sw R7$, max_length_path_OFF(R1$)	1		N′
# ACTIVATE the outgoing edges # R1$ my_node # R2$ my_node->out_edges_num # R3$ my_node->out_edges_start # R4$ <scratch> # R5$ <scratch> # R6$ spawn count register (nested spawns) # R7$ my_node->max_length_path ###			
lw R2$, out_edges_num(R1$)	1-5		N′
lw R3$, out_edges_start(R1$)	1-5		N′
BLOCK TOTAL **For each node in N′; do L times (time), do N′ times (work)**		**5L**	**3 N′**
# spawn # R1$ nextnode # R2$ this_edge # R3$ n # R4$ nextnode->in_edges_start # R5$ nextnode->in_edge_count # R6$ nextnode->in_edges_num # R7$ this_edge->edge_len # R8$ <scratch> ###			
spawn R6$, 0, R2$, 8	1		N'
li R3$, 1	2		E′
add R8$, R3$P, R0$	2		E′
laa R2$, 0(R2), SZ_EDGE[R8$]	3		E′
lw R8$, end_node_OFF(R2$)	4-8		E′
lw R7$, edge_len_OFF(R2$)	4-8		E′

laa R1$, 0(R1), SZ_NODE[R8$]	9		E′
lw R4$, in_edges_start_OFF(R1$)	10-14		E′
lw R6$, in_edges_num_OFF(R1$)	10-14		E′
psm in_edge_count_OFF(R1$), R3$	10-14		E′
add R8$, R3$, R4$	15		E′
add R7$, R7$, R7$P	15		E′
swa R7$, 0(R3), SZ_INT[R8$]	16		E′
# check if last incoming edge ###			
bne R3$, R6$, NOT_LAST_INCOMING_EDGE2	16		E′
li R3$, 1	17		$N' - L_2$
ps R6, R3$	18		$N' - L_2$
swa R1$, 0(R8), SZ_NODE_PTR[R3$]	19		$N' - L_2$
NOT_LAST_INCOMING_EDGE2: join R6$, R2$	20		E′
BLOCK TOTAL **For each outgoing edge in N'; do L times (time), do E' times (work)**		**20 L**	**$14E' + 4N' - 3L_2$**
# END enabled spawn ###			
join R30, R5	1		
BLOCK TOTAL **For each node (except starting nodes); do L times (time), do N' times (work)**		**L**	**N′**
# move curr and next ptrs ###			
laa R7, 0(R7), SZ_NODE_PTR[R5]	1		L
laa R8, 0(R8), SZ_NODE_PTR[R6]	1		L
move R5, R6	2		L
li R6, 0	3		L
j START_WHILE END_WHILE:	3		L
# END FUNCTION ##			
BLOCK TOTAL **Do L times**		**3L**	**5L**

Total Time:

0 + 12 + 10 + 20 + 1 + 3 + L +12L + 3L*log(D/2) + 10L*log(D/2) + L*log(D/2)
+ 11L + 5L + 20L + L + 3L

	const	L	L*log(D/2)			
	0	1	3			
	12	12	10			
	10	11	1			
	20	5				
	1	20				
	3	1				
		3				
Total:	46	53	14			

46 + 53L + 14L*log(D/2) = 46 + 53L + 14L(logD - 1)
=
46 + 39L + 14L*log(D)

Total Work:

$7 + (1 + 4N + 3N_0) + (2 + 4N_0) + (13E_0 + 3L_2 + N_0) + N_0 + 4 + L + (4N' + L)$
$(6(N_D - N'') + 2N_1 + 2N_2 + 2N_3) +$
$(13/2\ (E'' - 2N'' + N_D) + N_D - N'') +$
$(2N_D - 2N'') +$
$(2N' + 6.25N'' + 7.5N_3 + 5N_2 + N_1) +$
$3N' + (14E' + 4N' - 3L_2) + N' + 5L$

	const	N	N0	E0	L2	L	N’	ND	N”	N1	N2	N3	E”	E’
	7	4	3	13	3	1	4	6	-6	2	2	2	6.5	14
	1		4		-3	1	2	6.5	-13	1	5	7.5		
	2		1			5	3	1	-1					
	4		1				4	2	-2					
							1		6.25					
total	14	4	9	13	0	7	14	15.5	-15.75	3	7	9.5	6.5	14

$\mathbf{14 + 4N + 9N_0 + 13E_0 + 7L + 14N' + 15.5N_D - 15.75N''}$
$\mathbf{+ 2N_1 + 7N_2 + 9.5N_3 + 6.5E'' + 14E'}$

Time and Work Analysis of Parallel Spawn MT Algorithm

N	= Number of nodes
E	= Number of edges
N_0	= Number of nodes with 0 incoming edges, the start nodes
E_0	= Number of edges outgoing from the start nodes
L	= Number of layers - 1
D	= Max degree of nodes
L_x	= log(num nodes in layer 2) + log(num nodes in layer 3) + .. + log(num nodes in layer L)

Execution		Measurement	
Instruction	**Cycle executed**	**Time**	**Work**
# ASM for Parallel DAG algorithm # using SPAWN-MT model # R1 address nodes[0] # R2 address edges_out[0] # R3 address in_paths[0] # R4 num_nodes # R5 num_enabled # R6 next_num_enabled # R7 curr_enabled_nodes # R8 num_edges, next_enabled_nodes # R9 nodesptr # R10 curr_edges # R11 edge_space_size # R12 arr_locks # R13 node_space_len # R14 node_space_start # R15 <scratch> # R16 ps_arr # R17 ps_next # R18 ps_old # R19 ps_layer_start # R20 ps_sz # R27 n # R28 1 # R29 MINUS_INFINITY # R30 spawn count reg # START FUNCTION ##			
li R5, 0	1		1
li R6, 0	1		1
move R7, R9	1		1
li R10, edge_space_BASE + SZ_INT	1		1
li R28, 1	1		1
li R29, MINUS_INFINITY	1		1
BLOCK TOTAL **Do once. This block overlaps the next block by once cycle.**		**0**	**6**

# initialize edge_space & arr_lock ###			
spawn R30, 0, R8, 0	1		1
swa R0, 0(R10), SZ_INT[R0$]	2		E
join R30, R8	2		E
BLOCK TOTAL **Do once. E work.**		**2**	**1 + 2E**
sli R15, R4, 1	1		1
spawn R30, 0, R15, 0	2		1
swa R0, 0(R12), SZ_INT[R0$]	3		2N
join R30, R15	3		2N
BLOCK TOTAL **Do once. 2N work.**		**3**	**2 + 4N**
# create enabled array by checking nodes # R1$ tmpnode # R2$ tmpnode->in_edges_num # R3$ n # R4$ m ##			
spawn R30, 0, R4, 4	1		1
laa R1$, 0(R1), SZ_NODE[R0$]	2		N
lw R2$, in_edges_num_OFF(R1$)	3-7		N
beq R2$, R0, END_JOIN1	8		N
sw R29, max_length_path_OFF(R1$)	9		N_0
li R3$, 1	9		N_0
ps R5, R3$	10		N_0
swa R1$, 0(R7), SZ_NODE_PTR[R3$]	11		N_0
lw R4$, out_edges_num_OFF(R1$)	9	Assume it is in the cache	N_0
swa R4$, 0(R13), SZ_INT[R3$]	11		N_0
ENDJOIN1: join R30, R4	11		N
BLOCK TOTAL **Do once. Spawn over all the nodes (N work), branch block done on starting nodes (N_0 nodes)**		**11**	**1 + 4N + $6N_0$**
laa R8, 0(R7), SZ_NODE_PTR[R5]	1		1
BLOCK TOTAL **Do once. This block can overlap the next one by one cycle.**		**0**	**1**

# IMPORTANT NOTE: # # In order to improve readability of this already long # assembly code program, we skip writing a portion of # the program. The portion skipped is an exact copy # of the main loop with the modification that the line # of code is skipped where the maximum path value is added # to the edge length. This value is entered into # the binary tree of the destination node in order to find # the maximum path length for that node. In the case of # the first layer, there is no maximum path value so # only the edge length is used. Therefore, the first layer # is done as a separate (modified) version of the main # loop. (See the C code for the C version and see the EMT # code for a similar example.) ##			
START_WHILE: beq R5, R0, END_WHILE	1		L
# flatten nesting ##			
# GENERAL PS: node_space_len to node_space_start # # R16 ps_arr # R17 ps_next # R18 ps_old # R19 ps_layer_start # R20 ps_sz # R21 ps_sz[n] # R27 n ##			
move R16, R18	2		L
sw R0, 0(R18), SZ_INT[R5]	2		L
laa R17, SZ_INT(R18), SZ_INT[R5]	2		L
li R27, 0	2		L
addi R21, R5, 1	2		L
# no need to store ps_sz[0] to mem -- never used			
BLOCK TOTAL **Do for each layer (L time, L work)**		**2L**	**6L**
# start going up the tree ##			
START_PS_UP: beq R21, R28, END_PS_UP	1		L_x
addi R27, R27, 1	2		L_x
addi R21, R21, 1	2		L_x
sri R21, R21, 1	3		L_x
swa R21, 0(R20), SZ_INT[R27]	4		L_x

laa R17, SZ_INT(R16), SZ_INT[R21]	4		L_x
swa R0, 0(R16), SZ_INT[R21]	4		L_x
swa R17, 0(R19), SZ_INT[R27]	5		L_x
BLOCK TOTAL **Do this while loop log(Li) times, where Li is the number of nodes in layer i. And since we do this for each layer, this code is done Sum(log(Li)) times i=2..L i.e. L_X**		**$5L_X$**	**$8L_x$**
# spawn to add the layer # R1$ ps_arr[2$], ps_next[$] # R2$ ps_arr[2$ + 1] ###			
spawn R30, 0, R21, 2	1		L_x
lwa R1$, 0(R16), 2*SZ_INT[R0$]	2-6		N
lwa R2$, SZ_INT(R16), 2*SZ_INT[R0$]	2-6		N
add R1$, R1$, R2$	7		N
swa R1$, 0(R17), SZ_INT[R0$]	8		N
join R30, R21	8		N
BLOCK TOTAL **This spawn adds the values in the layer of the PS binary tree. So the work is 1/2 Li + 1/4 Li + 1/8 Li + ... = Li where Li is the number of nodes in layer i; xming over all the layers, we get N work.**		**$8L_X$**	**L_x + 5N**
move R16, R17	1		L_x
j START_PS_UP END_PS_UP:	1		L_x
BLOCK TOTAL **This block is in the PS while loop as above.**		**L_X**	**$2L_x$**
addi R17, R16, SZ_INT	1		L
BLOCK TOTAL **Do once per layer of the graph. L times. This block overlaps the following block by one cycle.**		**0**	**L**
# go back down the tree ###			
START_PS_DOWN: beq R27, R28, END_PS_DOWN	1		L_x - L
subi R27, R27, 1	2		L_x - L
lwa R18, 0(R19), SZ_INT[R27]	3-7		L_x - L
lw R15, 0(R18)	8-12		L_x - L
sw R15, 0(R17)	13		L_x - L
lwa R21, 0(R20), SZ_INT[R27]	3-7		L_X - L

subi R21, R21, 1	8		L_X - L
BLOCK TOTAL **Do this while loop log(Li) - 1 times, where Li is the number of nodes in layer i. We stop before the last round, hence the minus one. And since we do this for each layer, this code is done Sum(log(Li)-1) = L_X - L.**		**13 (L_X-L)**	**7(L_X - L)**
# spawn for the layer # R1$ $&1, ps_old[$] # R2$ $/2, ps_arr[$/2] or ps_arr[$/2 - 1], ps_next[$] ##			
spawn R30, 1, R21, 2	1		L_X - L
andi R1$, R0$, 1	2		N
sri R2$, R0$, 1	2		N
bne R1$, R0, IF_ODD_IN_PS_DOWN	3		N
lwa R1$, 0(R18), SZ_INT[R0$]	4-8		1/2 N
lwa R2$, -SZ_INT(R16), SZ_INT[R2$]	4-8		1/2 N
add R2$, R2$, R1$	9		1/2 N
j IF_END_IN_PS_DOWN	9		1/2 N
IF_ODD_IN_PS_DOWN: lwa R2$, 0(R16), SZ_INT[R2$]			1/2 N
IF_END_IN_PS_DOWN: swa R2$, 0(R17), SZ_INT[R0$]	10		N
join R30, R21	10		N
BLOCK TOTAL **This spawn processes the values in the layer of the PS binary tree, going down. We stop one layer before the last so the work is 2 + .. + 1/8 Li + 1/4 Li + 1/2 Li = Li** **where Li is the number of nodes in layer i; summing over all the layers, we get N work. Since some code is done for odd values and some for even, the branching occurs 1/2 the time.**		**10(L_X - L)**	**15/2 N + L_X - L**
move R16, R17	1		L_X - L
laa R17, SZ_INT(R17), SZ_INT[R21]	2		L_X - L
j START_PS_DOWN END_PS_DOWN:	2		L_X - L
BLOCK TOTAL **This block is in the PS while loop as above.**		**2(L_X - L)**	**3(L_X - L)**
# last layer -- # put 1's to delimit segments # # R1$ $&1, node_space_len[$] # R2$ $/2, ps_arr[$/2] or ps_arr[$/2 - 1], m # ##			

spawn R30, 1, R5, 2	1		L
andi R1$, R0$, 1	2		N
sri R2$, R0$, 1	2		N
bne R1$, R0, IF_ODD_IN_PS_DOWN2	3		N
lwa R1$, 0(R13), SZ_INT[R0$]	4-8		1/2 N
lwa R2$, -SZ_INT(R16), SZ_INT[R2$]	4-8		1/2 N
add R2$, R2$, R1$	9		1/2 N
j IF_END_IN_PS_DOWN	9		1/2 N
IF_ODD_IN_PS_DOWN2: lwa R2$, 0(R16), SZ_INT[R2$]			1/2 N
IF_END_IN_PS_DOWN2: swa R28, -SZ_INT(R10), SZ_INT[R2$]	10		N
join R30, R5	10		N
BLOCK TOTAL **This spawn is the last layer of the PS tree, going down. So the work here is Li; summing over layers, we get N work.**		**10 L**	**15/2 N + L**
# edge_space_size is the last location			
lwa R11, 0(R13), SZ_INT[R5]	1-5		L
li R27, 0	1		L
BLOCK TOTAL **Do once in each layer of the graph**		**5L**	**2L**
# now do ps to get parent node's index # R1$ curr_edges[$] ###			
spawnsync R30, 0, R11, 1	1		L
lwa R1$, 0(R14), SZ_INT[R0$]	2-6		E
ps R27, R1$	7		E
swa R1$, 0(R14) SZ_INT[R0$]	8		E
join R30, R11	9		E
BLOCK TOTAL **This synchronous spawn does not allow scheduling more than one instruction for a time slot. We spawn over the size of all the edges in the layer, and do this for every layer. So the total work is E.**		**9L**	**L + 4E**
# done flattening #			

# EXPLORE OUT from the edges ## # R1$ curr_edges[$], my_node->max_length_path # R2$ my_node, arr # R3$ this_edge, n # R4$ node_space_start[$], nextnode # R5$ i = nextnode->in_edge_count # R6$ m # R7$ nextnode->in_edge_start, lock_arr # R8$ nextnode->in_edge_num, arr_size # R9$ my_node->out_edges_start, max_val # R10$ this_edge->edge_len, <scratch> # R11$ lock_val and comp_val and <scratch> ##			
spawn R30, 0, R11, 11	1		L
lwa R1$, 0(R10), SZ_INT[R0$]	2-6		E
lwa R2$, 0(R7), SZ_NODE_PTR[R1$]	7-11		E
lw R9$, out_edges_start_OFF(R2$)	12-16		E
lwa R4$, 0(R14), SZ_INT[R0$]	2-6		E
add R3$, R9$, R0$	17		E
sub R3$, R3$, R4$	18		E
laa R3$, 0(R2), SZ_EDGE[R3$]	19		E
lw R4$, end_node_OFF(R3$)	20-24		E
laa R4$, 0(R1$), SZ_NODE[R4$]	25		E
# NOTE: As explained above, in the first layer # of the loop, this code would exist in a modified # version. The next line would read: # lw R9$, edge_len_OFF(R3$) # and the line add R9$, R1$, R10$ would be # eliminated			
lw R10$, edge_len_OFF(R3$)	20-24		E
lw R7$, in_edge_start_OFF(R4$)	26-30		E
lw R8$, in_edge_num_OFF(R4$)	26-30		E
lw R1$, max_length_path_OFF(R2$)	12-16		E
add R9$, R1$, R10$	25	This line is skipped in the first layer. (See notes above)	E - E_0
li R5$, 1	2		E
psm in_edge_count_OFF(R4$), R5$	26		E

move R6$, R5$	27		E
# maximum calculation -- async ###			
laa R2$, 0(R3), 3*SZ_INT[R7$]	31		E
laa R7$, 0(R12), 2*SZ_INT[R7$]	32		E
BLOCK TOTAL **For each layer, we explore out on all the edges that go out from the layer, i.e. E work. This block can overlap the following block by one cycle.**		**31L**	**L + 19E - E_0**
# while loop to go up the binary tree ###			
START_BINARY_TREE_WHILE: bne R8$, R28, END_BINARY_TREE_WHILE	1		2E
andi R11$, R8$, 1	2		2E
beq R11$, R0, TRY_LOCK	3	1/2 of all layers will take this branch	2E
subi R11$, R8$, 1			E
beq R11$, R6$, DONE_TRY_LOCK		This branch is taken when a value is the last in the tree layer	E
BLOCK TOTAL **This is the while loop to go up the binary tree for the max calculation. All the edges enter (E) + 1/2 go to the next level (1/2 E) + 1/2 of those go to the next level (1/4E) + …** **for a total of 2E work. Assuming half the layer sizes will be odd, the first branch will be taken half the time.**		**3L* log(D)**	**8E**
TRY_LOCK: swa R9$, 0(R2$), SZ_INT[R6$]	1		2E
li R11$, 1	1		2E
sri R10$, R6$, 1	1		2E
laa R10$, 0(R7$), SZ_INT[R10$]	2		2E
psm 0(R10$),R11$	3-7		2E
beq R11$, R28, END_BINARY_TREE_WHILE	8	Half will proceed	2E
andi R3$, R6$, 1	9		E
sli R3$, R3$, 1	10		E
laa R3$, -1(R0), -1[R3$]	11		E

lwa R11$, 0(R2$), SZ_INT[R3$]	12-16		E
slt R10$, R9$, R11$	17		E
beq R10$, R0, DONE_TRY_LOCK	18	Assume half the time we will find a new max and swap	E
move R9$, R11$	19		1/2 E
DONE_TRY_LOCK: laa R2$, 0(R2$), SZ_INT[R8$]	18		E
sri R8$, R8$, 1	19		E
sri R6$, R6$, 1	10		E
laa R7$, 0(R7$), SZ_INT[R8$]	20		E
j START_BINARY_TREE_WHILE END_BINARY_TREE_WHILE:	20		E
BLOCK TOTAL **This block is done as the threads go up a binary tree to find the max. At each layer half the threads drop out, so the work is E + 1/2E + 1/4E for the code which is done by all the threads. The last value in the tree layer (if the layer is odd-sized) is automatically moved up to the next layer, so the work measure for this block is an upper bound.**		**20L* log(D)**	**23.5 E**
# check if last incoming edge ##			
bne R18$, R28, NOT_LAST_INCOMING_EDGE	1		E
sw R9$, max_length_path(R4$)	2		$N - N_0$
li R5$, 1	2		$N - N_0$
ps R6, R5$	3		$N - N_0$
swa R4$, 0(R8), SZ_NODE_PTR[R5$]	4		$N - N_0$
lw R6$, out_edges_num_OFF(R4$)	2-6		$N - N_0$
swa R6$, 0(R13), SZ_INT[R5$] NOT_LAST_INCOMING_EDGE:	7		$N - N_0$
# END edges spawn ##			
join R30, R11	7		E
BLOCK TOTAL **Here we check to see if this thread is the last thread to arrive into the node. Hence only $N - N_0$ meet this criteria and the work is $N-N_0$.**		**7L**	**$2E + 7(N-N_0)$**

# move curr and next ptrs ###			
laa R10, 0(R10), SZ_INT[R11]	1		L
laa R7, 0(R7), SZ_NODE_PTR[R5]	1		L
laa R8, 0(R8), SZ_NODE_PTR[R6]	1		L
move R5, R6	2		L
li R6, 0	3		L
j START_WHILE END_WHILE:	3		L
# END FUNCTION #######################################			
BLOCK TOTAL **Do once per layer, so L work and L time.**		**3L**	**6L**

Total Time:

$0 + 2 + 3 + 11 + 0 + 2L + 5L_X + 8L_X + L_X + 0 + 13(L_X - L) + 10(L_X - L) + 2(L_X - L)$
$+ 10L + 5L + 9L + 31L + 3L*\log D + 20L*\log D + 7L + 3L$

	const	L	LX	L*logD
	0	2	5	3
	2	-13	8	20
	3	-10	1	
	11	-2	13	
	0	10	10	
	0	5	2	
		9		
		31		
		7		
		3		
Total:	**16**	**42**	**39**	**23**

$16 + 42L + 39L_X + 23L*\log D$

Total Work:

$6 + (1 + 2E) + (2 + 4N) + (1 + 4N + 6N_0) + 1 + 6L + 8L_X + (L_X + 5N) + 2L_X + L$
$+ 7(L_X - L) + (15/2\ N + L_X - L) + 3(L_X - L) + (15/2\ N + L)$
$+ 2L + (L + 4E) + (L + 19E - E_0) + 8E$
$+ 23.5E + (2E + 7(N - N_0)) + 6L$

	const	N0	N	E	L	LX	E0	
	6	6	4	4	6	8	-1	
	1	5	4	19	1	1		
	2	-7	5	8	-7	2		
	1		7.5	23.5	-1	7		
	1		7.5	2	-3	1		
			7		1	3		
					2			
					1			
					1			
					6			
Total:	**11**	**4**	**35**	**56.5**	**7**	**22**	**-1**	

$11 + 4N_0 + 35N + 56.5E + 7L + 22L_X - E_0$