Experimenting with List Ranking Algorithms for Explicit Instruction Parallelism

Shlomit Dascal
Advisor: Prof. Uzi Vishkin

November 30, 1999

Following are the C (extended-C) and assembly code for each of the five list ranking algorithms we experimented with. Follows it a short explanation on why this compilation is feasible, and a brief explanation of the simulations we performed to support the execution analysis of the algorithms. After that we included tables with the step-wise analysis of the execution and the overall work and time counts for each of the algorithms.

This file includes supporting documents for a journal submission of the paper with the same title, which was presented at the 3rd Worskhop on Algorithms Engineering (WAE'99), London, U.K. July 1999. In addition to a detailed high level description of each of the algorithms the journal submission contains a summary of specific optimizations and decisions made (including a comparison to additional algorithms attempted), further analysis of the algorithms behavior, programming techniques used, some explanations on why the compilation is feasible, comparison to other work on list ranking, as well as results for various inputs.

Note that in order to derive the final work and time results from the step-wise counts in the current document, some additional analysis that appears in the journal submission is necessary, such as: a bound on the longest chain in the "no-cut" randomized coin-tossing algorithm; the analysis of the convergence of the series of lengths of the arrays iteration-wise and the length of the series in both of the coin-tossing algorithms; further considerations for the loop unrolling in the final ranking of the serial algorithm; and some explanations on how we calculated the value of $log_2 N$ efficiently in Wyllie's algorithm.

In appendix D we describe and explain briefly the simulations we performed to support the execution analysis of the more complex algorithms. The source code for the supporting simulations described here can be reached through the "Supporting Simulations" link on our web page.

The journal submission makes some general observations regarding the compilation feasibility from the high level code (C and extended-C) to the optimized assembly code, using known compiler techniques. The compilation feasibility section in the current document provides more concrete reference. The line numbers given as examples are all in the assembly code table of the “no-cut” algorithm, unless stated otherwise.

Contents

A High Level Code for the Algorithms

This section presents the high-level source code for four of the algorithms implemented. It starts with the header file *list_rank.h* used by all the c files, then presents the C/extended-C code for the serial algorithm and then for the three parallel algorithms.

A.1 Header File

```
/*  97/08/27    @(#)list_rank.h 1.7   */

#define  TRUE   1
#define  FALSE  0

#define  MAX_ELEMENTS   1024

/* a definition of a BYTE - unsigned-int-8-bits (char) */
typedef char  UINT_8;


/*   Element:   An element contains the following fields: */
typedef struct List_Array {
  void               *data;     /* pointer to a structure with the element data */
  int                 weight;   /* weight value of exiting edge */
  int                 rank;     /* rank value of element, initialized to be WEIGHT
                                   except for LAST element */
  struct List_Array  *back;     /* pointer to prev element, NULL if head (first) */
  struct List_Array  *next;     /* pointer to next element, NULL if last */
  UINT_8              select;   /* TRUE if element selected, FALSE if not */
  UINT_8              done;     /* TRUE if finished working on this element
                                   (e.g., "skipped"), FALSE if not */
  struct List_Array  *prnt;     /* pointer to the ranking-parent element in the original
                                   list. The final rank of a skipped element is
                                   determined relative to the rank of its
                                   ranking-parent:
                                       rank = rank(ranking-parent) - ranking-difference
                                 */
  struct Actv_Struct *actv_ptr; /* pointer to the copy of the element in the CURRENT
                                   active list */
#ifdef DBG
  char                str[256]; /* element identification string (if any) */
#endif
} List_Array;


/* copy an element's contents */
#define COPY_ELEMENT(to, from)\
       { (to)->data   = (from)->data; \
```

```
        (to)->weight = (from)->weight; \
        (to)->rank   = (from)->rank; \
        (to)->back   = (from)->back; \
        (to)->next   = (from)->next; \
        (to)->select = (from)->select; \
        (to)->done   = (from)->done; \
        (to)->prnt   = (from)->prnt; \
        (to)->actv_ptr = (from)->actv_ptr; \
      }


typedef struct Actv_Struct{
  List_Array   *list_ptr;      /* ptr to element's place in original list */
  List_Array    element;
} Actv_Struct;


typedef struct{
  List_Array   *list_ptr;  /* ptr to element's place in original list */
  int           rank;      /* a copy of "rank", which is actually the "ranking-diff"
                              from the ranking-prnt */
  List_Array   *prnt_addr; /* ptr to ranking-parent of element */
} Nactv_Struct;


typedef struct { /* information for backwards iterations */
  int            bck_num;        /* number of elements in nactv_array */
  Nactv_Struct  *bck_start_addr; /* where in the nactv_array the iteration started */
} Iter_Struct;
```

A.2 Serial Algorithm

```
/*  97/12/17     @(#)ser_list_rank.c    1.7   */

/* List Ranking - serial alg. */

#include <stdio.h>
#include "list_rank.h"

/* Partially parallel implementation of the serial alg. (ranking at the
   end can be calculated to all nodes at once.
   Goal - to extract as much ILP as possible from the serial algorithm.
 */

void serial_parallel_ranking(List_Array *list, int list_size)
```

```
{
  int          max_rank=0, i=0;
  List_Array  *next=NULL, *curr=NULL, *list_head = NULL;

  if (list_size <= 0) /* empty list */
    return;

  /* if the list has only 1 element -> nothing to do */
  if (list_size == 1)
    {
      list->rank = 0;
      return;
    }

  /* run through the list: and find the HEAD */
  for (i=0; i<list_size; i++)
    if (list[i].back == NULL) /* found the HEAD */
      {
        list_head = &(list[i]);
        break;
      }

  next = list_head->next;
  curr = list_head;

  /* Forward Loop - run through the list: getting to the end to know what is maximal
     RANK & "mark" rank differences (from the MAX) along the way */
  while (next != NULL) /* the list is not ended yet */
    {
      curr->rank = -1 * max_rank; /* rank-difference from final MAX_RANK
                                     (BEFORE updating it with current weight) */
      max_rank += curr->weight;   /* add weighted distance to next element! */
      curr = next;
      next = next->next;
    }
  /* special treatment for last element */
  curr->rank = -1 * max_rank;

  /* Final Ranking - rank each element */
  for (i=0; i<list_size; i++)
    {
      list[i].rank += max_rank;
    }

}
```

A.3 Parallel Randomized Coin-Tossing Algorithm (no-cut)

```
/*   97/12/17   @(#)par_list_rank.c     1.13   */

/*** List Ranking - Parallel Random Alg. (coing tossing) - Time Optimized (no-cut) ***/

#include <stdio.h>
#include "list_rank.h"
#include "par_list_rank.h"

/********  General Variables  ********/

/** the orignial list **/

static List_Array       local_list[MAX_ELEMENTS];
static int              local_next_avail=0;


/** parallel iterations **/

/* Array of active elements and the index up to where is it filled */
static Actv_Struct  actv_array[MAX_ELEMENTS*MAX_ROUNDS];
static int  actv_indx_used = 0;

/* Array of non-active elements and the index up to where is it filled */
static Nactv_Struct  nactv_array[MAX_ELEMENTS*MAX_ROUNDS];
static int  nactv_indx_used = 0;

/* Information held for each of the iterations to "back-track", e.g., propagate
   the final ranking to all elements
 */
static Iter_Struct   iterations_info[MAX_ROUNDS];
static int  iter = 0;

static UINT_8    coin_tossing_array[MAX_ELEMENTS]; /* BYTE array for coin tossing data */
static UINT_8   *curr_coin_array=NULL;


/***************  Utility  Macros  ***************/

#define  IS_SELECTED(index)\
       ((curr_coin_array[index]) & 0x01) /* masking on lowest bit in this address */


/****************  Principal Functions ***************/

/*** The Control (flow) of the algorithm (calling the various parts) ***/
```

```
void parallel_rand_ranking(List_Array *list, int list_size)
{
  int            curr_list_size, next_list_size;
  Actv_Struct   *curr_actv;
  Nactv_Struct  *curr_nactv;

  if (list_size <= 1) /* no list OR ranking is OBVIOUS */
    return;

  /** preparations for "forward iterations" **/
  /*    random selection process based on coin-tossing and copying the elements to the
   *    active array
   */
  par_first_fwditr(list, list_size, actv_array);


  /** initializations **/
  iter = 0;
  curr_list_size = list_size;
  curr_actv  = actv_array; /* for READ - use beginning of actv_array to start with */
  curr_nactv = nactv_array;


  /** "forward iterations" - in parallel **/
  /* In the first stage "forward iterations" are performed, where a selection process
     is applied between the iterations, and a pointer-jumping process is applied starting
     at the selected elements, until another selected element is hit. At the end of
     this stage the rank of the head of the list will be known.
   */

  while (curr_list_size > 1) /* condition to finish */
    {
       /* works on actv_array */
      next_list_size = par_fwditr(curr_actv, curr_list_size, curr_nactv);

       /* update this iteration's info */
      iterations_info[iter].bck_start_addr = curr_nactv;
      iterations_info[iter].bck_num   = curr_list_size - next_list_size;
      iter += 1;

       /* update starting addresses to actv_array & nactv_array */
        /* before updating with CURRENT active - for read */
      curr_actv  = &(actv_array[actv_indx_used]);
      actv_indx_used += next_list_size;

      nactv_indx_used += (curr_list_size - next_list_size);
        /* after updating with CURRENT non-active - for write at the end */
```

```
      curr_nactv = &(nactv_array[nactv_indx_used]);

      curr_list_size = next_list_size;
    }


  /** "backward iterations" - in parallel **/
  /* In the second stage "backward iterations" are performed, where the ranking
     information is spread to the rest of the elements in the list.
   */

  while (nactv_indx_used > 0) /* condition to finish - since all will be RANKED */
    {
       /* works on the nactv_array */
      iter -= 1;
      par_bckitr(iterations_info[iter].bck_start_addr, iterations_info[iter].bck_num);
      nactv_indx_used -= iterations_info[iter].bck_num;
    }

}


/* first forward iteration works DIRECTLY on ALL elements  (as opposed to active array)
    DOES:  random selection process based on coin-tossing and copying the elements to the
           active array

   orig_list   - (IN)  ptr to array of original linked list,
   list_size   - (IN)  number of elements in the list,
   actv        - (OUT) ptr to start of ACTIVE ARRAY where elements will be copied to
 */

void  par_first_fwditr(List_Array *orig_list /*IN*/, int list_size /*IN*/,
                       Actv_Struct *actv /*OUT*/)
{
  curr_coin_array = coin_tossing(); /* where to start_looking */

  SPAWN (0, list_size)
    {
      /* variables local to each thread */
      UINT_8          selected;

      /** random selection process based on coin-tossing **/
      if (orig_list[$].back == NULL) /* HEAD of the list */
        selected = TRUE;
      else if (orig_list[$].next == NULL) /* LAST of the list - not selected */
        selected = FALSE;
      else /* all the "middle" elements */
```

```
        {
          /* TRUE only if I'm selected and my-next is NOT */
          selected = IS_SELECTED($) && !(is_selected_addr(orig_list[$].next));
        }

      /** copying the elements to the active array **/
      actv[$].list_ptr = &(orig_list[$]);
      orig_list[$].select = selected;
      COPY_ELEMENT(&(actv[$].element), &(orig_list[$]));

    }

  /* when done should set actv space used */
  actv_indx_used = list_size;

}


/* forward iterations - works on active array

   curr_actv      - (IN)  ptr to start of relevant part of active array to read
                    (OUT) ptr to the active array relevant for next iteration
   curr list_size - (IN)  number of active elements in the array
   curr_nactv     - (OUT) ptr to start of nactv_array resulting of this iter.

   RETURN VALUE:  number of active elements left (in shrinked list).
*/

int  par_fwditr(Actv_Struct  *curr_actv /*IN/OUT*/, int  curr_list_size /*IN*/,
                Nactv_Struct  *curr_nactv /* OUT*/)
{
  int    actv_counter, nactv_counter;

  /** initialize counters **/
  actv_counter  = 0;
  nactv_counter = 0;


  /*  for each selected element -> skip the following elements (incrementing
          by their weighted-distances, and storing for them the
          current element as the ranking-parent) until the next
          selected element is encountered (NOT restricting the number of
          skipped elements, e.g., no-cut)
   */
  /*  for each element if still active (was selected) ->
          copy-shrink into next active array and do the selection
          process for the next iteration
```

```
    for each element if was skipped (not selected) -> copy-shrink into non-active array
                                                     during the short-cutting process
 */

curr_coin_array = coin_tossing(); /* where to start_looking */

SPAWN (0, curr_list_size)
  {
    /* variables local to each thread */
    int           rank, tmp_add_rank;
    Actv_Struct  *me;
    List_Array   *me_orig, *next_orig;
    UINT_8        selected;
    int           new_indx;

    me = &(curr_actv[$]);
    me_orig = curr_actv[$].list_ptr;

    /* each of the "selected" continues advancing the ptr & updated ptr value */
    if (me->element.select)
      {
        next_orig = curr_actv[$].element.next;
        rank = me->element.rank;

        /* run until reach next selected element */
        while (!(next_orig->select))
          {
            /* update rank-difference, ranking-parent, etc. for "skipped" element
               in actv & ORIGINAL list */
              /* rank-diff: for quick calculation of final rank, from ranking_parent */
            tmp_add_rank = next_orig->rank;
            next_orig->rank = rank;
              /* rank-prnt */
            next_orig->prnt = me_orig;

            /***** since SKIPPED -> shrink into curr_nactv (not-actv array) *****/
            /* prefix-sums on NOT-active */
            new_indx = PS(nactv_counter, 1);

            /* address of original list into nactv-array */
            curr_nactv[new_indx].list_ptr = next_orig;

            /* update copy of "ranking parent" and "ranking difference" for "skipped"
               element in nactv-array */
            curr_nactv[new_indx].rank = next_orig->rank;
            curr_nactv[new_indx].prnt_addr = next_orig->prnt;
```

```
            /* add current weight to rank */
            rank += tmp_add_rank;

            /* advance to next element (if there is one) */
            next_orig = next_orig->next;
            if (next_orig == NULL)
              break;
          }

        /* update final rank & next in original-list */
        me_orig->rank = rank;
        me_orig->next = next_orig;

        /***** since SELECTED shrink into actv array *****/
        /** if NOT SKIPPED (still active) -> copy-shrink into next active array and
                                   do the selection process for the next iteration**/

        /** random selection process based on coin-tossing **/
        if (curr_actv[$].element.back == NULL) /* Check if I'm the HEAD of the list */
          selected = TRUE;
        else if (next_orig == NULL) /* check if I'm the LAST of the list */
          selected = FALSE; /* not selected */
        else /* all the "middle" elements */
          {
            /* TRUE only if I'm selected and next ("new" one after skipping) is NOT */
            selected = is_selected_addr(me_orig) && !(is_selected_addr(next_orig));
          }

        me_orig->select = selected;

        /** shrink into active array **/
        new_indx = PS(actv_counter, 1); /* prefix-sums on ACTIVE */
        curr_actv[curr_list_size + new_indx].list_ptr = me_orig;
        me_orig->actv_ptr = &(curr_actv[curr_list_size + new_indx]);

        /** copying the active elements to the active array **/
        COPY_ELEMENT(&(curr_actv[curr_list_size + new_indx].element), me_orig);

      } /* finished the skipping, shrinking & selection for next iteration */

    }  /* end of SPAWN */ /*  synchronize  */

  return(actv_counter);
}


void  par_bckitr(Nactv_Struct  *curr_nactv /*IN*/, int  size /*IN*/)
```

```
{
  /* for each element in the non-active array -> calculate final ranking based
         on the ranking-parent's final rank
   */

  SPAWN (0, size)
    {
      Nactv_Struct   *me=NULL;
      List_Array     *me_orig=NULL, *ranking_prnt=NULL;

      me = &(curr_nactv[$]);
      me_orig = curr_nactv[$].list_ptr;
      ranking_prnt = curr_nactv[$].prnt_addr;

      me_orig->rank = ranking_prnt->rank - me->rank;
    }
}


/*************** Utility Functions ***************/

UINT_8 *coin_tossing()
{
  /* coin tossing (used to determine which of the list_size elements in the array
   * will be "selected" for the next stage):
   * mark a bit: 1 OR 0
   */

  static   int   first_time = TRUE;
  static   int   curr_coin_indx=0;


  if (first_time)
    { /* only initialize once */
      get_random_coin_tossing_values(&(coin_tossing_array));
      first_time = FALSE;
    }
  else
    /* SHIFTs the place we're looking at in the coin_tossing_array */
    curr_coin_indx += 1;

  return(&(coin_tossing_array[curr_coin_indx]));
}


/* Selection Process:
   A subset of the elements of the list are selected, such that approximately
```

```
   one quarter (1/4) of the elements are selected and no two consecutive elements are
   selected. The selection of elements which are more than two apart is
   probabilistically independent.
 */
UINT_8  is_selected_addr(List_Array*  addr)
{
  int   my_index;

  my_index = (int) addr - (int) &(local_list); /* my distance from begining of list */
  my_index = my_index / sizeof(List_Array);  /* divided by element-size */

  return(IS_SELECTED(my_index));
}
```

A.4 Time-Optimized Parallel Coin-Tossing Algorithm (cut-6)

```
/*   97/12/17   @(#)par_list_rank.c     1.10.1.2   */

/*** List Ranking - Parallel Random Alg. (coing tossing) - Work Optimized (cut-K) ***/

#include <stdio.h>
#include "list_rank.h"
#include "par_list_rank.h"


/********  General Definitions  ********/

#define K_CUT  6   /* the optimal K to cut at is 6 */


/********  General Variables  ********/

/** the orignial list **/

static List_Array       local_list[MAX_ELEMENTS];
static int              local_next_avail=0;


/** parallel iterations **/

/* Array of active elements and the index up to where is it filled */
static Actv_Struct  actv_array[MAX_ELEMENTS*MAX_ROUNDS];
static int  actv_indx_used = 0;

/* Array of non-active elements and the index up to where is it filled */
static Nactv_Struct  nactv_array[MAX_ELEMENTS*MAX_ROUNDS];
static int  nactv_indx_used = 0;
```

```
/* Information held for each of the iterations to "back-track", e.g., propagate
   the final ranking to all elements
 */
static Iter_Struct   iterations_info[MAX_ROUNDS];
static int  iter = 0;

static UINT_8    coin_tossing_array[MAX_ELEMENTS]; /* BYTE array for coin tossing data */
static UINT_8   *curr_coin_array=NULL;


/***************  Utility  Macros  ***************/

#define  IS_SELECTED(index)\
       ((curr_coin_array[index]) & 0x01) /* masking on lowest bit in this address */


/****************  Principal Functions ***************/

/*** The Control (flow) of the algorithm (calling the various parts) ***/

void parallel_rand_ranking(List_Array *list, int list_size)
{
  int             curr_list_size, next_list_size;
  Actv_Struct   *curr_actv;
  Nactv_Struct  *curr_nactv;

  if (list_size <= 1) /* no list OR ranking is OBVIOUS */
    return;

  /** preparations for "forward iterations" **/
  /*    random selection process based on coin-tossing and copying the elements to the
   *    active array
   */
  par_first_fwditr(list, list_size, actv_array);


  /** initializations **/
  iter = 0;
  curr_list_size = list_size;
  curr_actv  = actv_array; /* for READ - use beginning of actv_array to start with */
  curr_nactv = nactv_array;


  /** "forward iterations" - in parallel **/
  /* In the first stage "forward iterations" are performed, where a selection process
     is applied between the iterations, and a pointer-jumping process is applied starting
```

```
     at the selected elements, up to (at most) "K_CUT" elements ahead. At the end of
     this stage the rank of the head of the list will be known.
   */

  while (curr_list_size > 1) /* condition to finish */
    {
       /* works on actv_array */
      next_list_size = par_fwditr(curr_actv, curr_list_size, curr_nactv);

       /* update this iteration's info */
      iterations_info[iter].bck_start_addr = curr_nactv;
      iterations_info[iter].bck_num   = curr_list_size - next_list_size;
      iter += 1;

       /* update starting addresses to actv_array & nactv_array */
        /* before updating with CURRENT active - for read */
      curr_actv  = &(actv_array[actv_indx_used]);
      actv_indx_used += next_list_size;

      nactv_indx_used += (curr_list_size - next_list_size);
        /* after updating with CURRENT non-active - for write at the end */
      curr_nactv = &(nactv_array[nactv_indx_used]);

      curr_list_size = next_list_size;
    }


  /** "backward iterations" - in parallel **/
  /* In the second stage "backward iterations" are performed, where the ranking
     information is spread to the rest of the elements in the list.
   */

  while (nactv_indx_used > 0) /* condition to finish - since all will be RANKED */
    {
       /* works on the nactv_array */
      iter -= 1;
      par_bckitr(iterations_info[iter].bck_start_addr, iterations_info[iter].bck_num);
      nactv_indx_used -= iterations_info[iter].bck_num;
    }

}


/* first forward iteration works DIRECTLY on ALL elements  (as opposed to active array)
    DOES:  random selection process based on coin-tossing and copying the elements to the
           active array
```

```
   orig_list   - (IN)  ptr to array of original linked list,
   list_size   - (IN)  number of elements in the list,
   actv        - (OUT) ptr to start of ACTIVE ARRAY where elements will be copied to
 */

void  par_first_fwditr(List_Array *orig_list /*IN*/, int list_size /*IN*/,
                       Actv_Struct *actv /*OUT*/)
{
  curr_coin_array = coin_tossing(); /* where to start_looking */

  SPAWN (0, list_size)
    {
      /* variables local to each thread */
      UINT_8          selected;

      /** random selection process based on coin-tossing **/
      if (orig_list[$].back == NULL) /* HEAD of the list */
        selected = TRUE;
      else if (orig_list[$].next == NULL) /* LAST of the list - not selected */
        selected = FALSE;
      else /* all the "middle" elements */
        {
          /* TRUE only if I'm selected and my-next is NOT */
          selected = IS_SELECTED($) && !(is_selected_addr(orig_list[$].next));
        }

      /** copying the elements to the active array **/
      actv[$].list_ptr = &(orig_list[$]);
      orig_list[$].select = selected;
      COPY_ELEMENT(&(actv[$].element), &(orig_list[$]));

    }

  /* when done should set actv space used */
  actv_indx_used = list_size;

}


/* forward iterations - works on active array

   curr_actv      - (IN)  ptr to start of relevant part of active array to read
                    (OUT) ptr to the active array relevant for next iteration
   curr list_size - (IN)  number of active elements in the array
   curr_nactv     - (OUT) ptr to start of nactv_array resulting of this iter.

   RETURN VALUE:  number of active elements left (in shrinked list).
```

```
*/

int  par_fwditr(Actv_Struct  *curr_actv /*IN/OUT*/, int  curr_list_size /*IN*/,
                Nactv_Struct  *curr_nactv /* OUT*/)
{
  int    actv_counter, nactv_counter;

  /** initialize counters **/
  actv_counter  = 0;
  nactv_counter = 0;


  /*  for each selected element -> skip the following elements (incrementing
          by their weighted-distances, and storing for them the
          current element as the ranking-parent) until the next
          selected element is encountered OR at most until
          skipped "K_CUT" elements, e.g., cut-k (cut-6 in our case)
   */

  SPAWN (0, curr_list_size)
    {
      /* variables local to each thread */
      int           rank, tmp_add_rank, skipped;
      Actv_Struct  *me;
      List_Array   *me_orig, *next_orig;

      me = &(curr_actv[$]);

      /* each of the "selected" continues advancing the ptr & updated ptr value */
      if (me->element.select)
        {
          me_orig = curr_actv[$].list_ptr;
          next_orig = curr_actv[$].element.next;
          rank = me->element.rank;
          skipped = 0;

          /* run until reach next selected element OR at most skipped "K_CUT" elements */
          while ( (skipped < K_CUT) && (next_orig != NULL) && (!(next_orig->select)) )
            {
              /* update rank-difference, ranking-parent, etc. for "skipped" element
                 in actv & ORIGINAL list */
              /* rank-diff: for quick calculation of final rank, from ranking_parent */
              tmp_add_rank = next_orig->rank;
              next_orig->rank = -1 * rank;
              next_orig->actv_ptr->element.rank = -1 * rank;
                /* rank-prnt */
              next_orig->prnt = me_orig;
```

```
            next_orig->actv_ptr->element.prnt = me_orig;
              /* done FLAG - since finished FORWARD work on this element */
            next_orig->done = TRUE;
            next_orig->actv_ptr->element.done = TRUE;

            /* add current weight to rank and advance to next element */
            rank += tmp_add_rank;
            next_orig = next_orig->next;
            skipped += 1;
          }

        /* update final rank & next both in actv-array and original-list */
        me->element.rank = rank;
        me->element.next = next_orig;
        me_orig->rank = rank;
        me_orig->next = next_orig;

      } /* finished the skipping */

  }  /* end of SPAWN */ /*  synchronize  */


/*  for each element if still active (was active OR was not skipped) ->
        copy-shrink into next active array and do the selection
        process for the next iteration
    for each element if was skipped -> copy-shrink into non-active array
 */

curr_coin_array = coin_tossing(); /* where to start_looking */

SPAWN (0, curr_list_size)
  {
    /* variables local to each thread */
    UINT_8        selected;
    int           new_indx;
    Actv_Struct  *me;
    List_Array   *me_orig;

    me = &(curr_actv[$]);
    me_orig = curr_actv[$].list_ptr;

    /** if DONE -> copy-shrink into curr_nactv **/
    if (me->element.done)
      {
        /* prefix-sums on NOT-active */
        new_indx = PS(nactv_counter, 1);
```

```
          /* update address in original list to nactv-array */
          curr_nactv[new_indx].list_ptr = me->list_ptr;

          /* update copy of "ranking parent" and "ranking difference" for "skipped"
             element in nactv-array */
          curr_nactv[new_indx].rank = me->element.rank;
          curr_nactv[new_indx].prnt_addr = me->element.prnt;
        }
      /** if NOT DONE (still active) -> copy-shrink into next active array and
                                        do the selection process for the next iteration**/
      else
        {
          /** random selection process based on coin-tossing **/
          if (curr_actv[$].element.back == NULL) /* HEAD of the list */
            selected = TRUE;
          else if (curr_actv[$].element.next == NULL) /* LAST of the list - not selected*/
            selected = FALSE;
          else /* all the "middle" elements */
            {
              /* TRUE only if I'm selected and my-next is NOT */
              selected = is_selected_addr(curr_actv[$].list_ptr) &&
                !(is_selected_addr(curr_actv[$].element.next));
            }

          me_orig->select = selected;
          me->element.select = selected;

          /** shrink into active array **/
          new_indx = PS(actv_counter, 1); /* prefix-sums on ACTIVE */
          curr_actv[curr_list_size + new_indx].list_ptr = me->list_ptr;
          me_orig->actv_ptr = &(curr_actv[curr_list_size + new_indx]);

          /** copying the active elements to the active array **/
          COPY_ELEMENT(&(curr_actv[curr_list_size + new_indx].element), &(me->element) );
        }
    } /* end of SPAWN */


  return(actv_counter);
}


void  par_bckitr(Nactv_Struct  *curr_nactv /*IN*/, int  size /*IN*/)
{
  /* for each element in the non-active array -> calculate final ranking based
         on the ranking-parent's final rank
   */
```

```
  SPAWN (0, size)
    {
      Nactv_Struct   *me=NULL;
      List_Array     *me_orig=NULL, *ranking_prnt=NULL;

      me = &(curr_nactv[$]);
      me_orig = curr_nactv[$].list_ptr;
      ranking_prnt = curr_nactv[$].prnt_addr;

      me_orig->rank = me->rank + ranking_prnt->rank;
    }
}


/***************  Utility  Functions  ***************/

UINT_8 *coin_tossing()
{
  /* coin tossing (used to determine which of the list_size elements in the array
   * will be "selected" for the next stage):
   * mark a bit: 1 OR 0
   */

  static   int   first_time = TRUE;
  static   int   curr_coin_indx=0;

  if (first_time)
    { /* only initialize once */
      get_random_coin_tossing_values(&(coin_tossing_array));
      first_time = FALSE;
    }
  else
    /* SHIFTs the place we're looking at in the coin_tossing_array */
    curr_coin_indx += 1;

  return(&(coin_tossing_array[curr_coin_indx]));
}


/* Selection Process:
   A subset of the elements of the list are selected, such that approximately
   one quarter (1/4) of the elements are selected and no two consecutive elements are
   selected. The selection of elements which are more than two apart is
   probabilistically independent.
 */
UINT_8  is_selected_addr(List_Array*  addr)
```

```
{
  int   my_index;

  my_index = (int) addr - (int) &(local_list); /* my distance from begining of list */
  my_index = my_index / sizeof(List_Array);  /* divided by element-size */

  return(IS_SELECTED(my_index));
}
```

A.5 Wyllie's Parallel Algorithm

```
/* 97/12/17     @(#)par_wyllie.c       1.4 */

/* for alternating READ & WRITE lists */
static List_Array  list1[MAX_ELEMENTS], list2[MAX_ELEMENTS];


void parallel_wyllie_ranking(List_Array *actv_list, int list_size)
{
  List_Array     *read_list, *write_list;
  int             number_iters;

  if (list_size <= 1) /* no list OR ranking is OBVIOUS */
    return;

  /** NOTE: we assume that the ranks are already initialized **/

  /* we want a list with INDEXES (as opposed to pointers/addresses),
     so that we can use alternating READ and WRITE lists */
  read_list  = list1;
  write_list = list2;

  /* prepare the first READ-list */
  SPAWN (0, list_size)
    {
      me = &(actv_list[$]);      /* my_index = $; */

      /* give each it's index in the current active list */
      if ((me->next) != NULL) /* not LAST */
        {
          next = me->next->actv_ptr; /* this is an address in the actv list */
          next_index = (next - actv_list) / sizeof(Actv_Struct);
        }
      COPY_ELEMENT(&(read_list[$]),&(me->element));
```

```
      /* NOTE: in the assembly we only copy essential info, as part of optimizing */

      /* overwrite with index instead of address! */
      read_list[$].next = next_index;

    } /* end of SPAWN */


  /** pointer-jumping **/

  number_iters = ceil(log2(list_size)); /* In the assembly we use an efficient
                                            implementation of log2() with a spawn */

  /* The forward iterations */
  for (k=1; k<= number_iters; k++)
    {
      SPAWN (0, list_size)
        {
          me = &(read_list[$]);

          if ( ((read_list[me->next]).next) != NULL) /* next is not LAST */
            {
              write_list[$].rank = me->rank + read_list[me->next].rank;
              write_list[$].next = read_list[me->next].next;
            }
        } /* end of SPAWN */

      /* need to swap the READ & WRITE lists for next iteration */
      read_list  ^= write_list;
      write_list ^= read_list;
      read_list  ^= write_list;

    } /* end of "for" */

  /* AT the end - update in original-list-array (to do the rest of the backtracking) */
  SPAWN (0, list_size)
    {
      me = actv_list[$].list_ptr;
      me->rank = write_list[$].rank;
    } /* end of SPAWN */

}
```

B Assembly Code for the Algorithms

This section presents the commented assembly (instruction set) code for the algorithms implemented. It starts with the *MIPS* assembly code for the serial algorithm followed by the *spawn-MT* assembly code for the four parallel algorithms, all of which use the asynchronous-mode.

B.1 Serial Algorithm

```
#       97/07/25        @(#)ser_list_rank.asm   1.9

# List Ranking - Serial-Parallel Ranking Algorithm

### Some Constant OFFsets and SIZEs meaning:
#    ----------------------------------------
#     rank_OFF        - offset of the FIELD of rank, in the structure of an element
#     weight_OFF      - offset of the FIELD of weight, in the element structure
#     back_OFF        - offset of the FIELD of back, indication if the element is
#                       the HEAD of the list
#     next_OFF        - offset of the FIELD of next address, in the structure of an
#                       element
#     element_SIZE    - size of an element in the list array


# main part of the ranking
# C-function:   void serial_parallel_ranking(List_Array *list, int list_size)

    # registers
    #     R1    pointer to the array of the list
    #     R2    list size
    #     R3    pointer to HEAD of the list
    #     R4    max_rank
    #     R5    current element (on which we are working)
    #           [e.g., list[i] / i-th after HEAD]
    #     R6    back value / next element / address of last element
    #           in array
    #     R7    new rank (diff) / total used array size
    #     R8    current weight / rank-diff & final-rank / misc
    #     R9 ...  used for loop unrolling


  lw    R1, list_array_OFF(SP)          # load list-array pointer from stack
  lw    R2, list_size_OFF(SP)           # load list_size value from stack

  ble   R2, R0, DONE                    # if (list_size <= 0) /* empty */ -> exit
  li    R3, 0                           # list_head = NULL
  li    R4, 0                           # max_rank = 0
  li    R7, 0                           # i = 0

  li    R8, 1
  ble   R2, R8, HEAD                    # if (list_size != 1) -> find the HEAD
  # list only has ONE element
  sw    R0, rank_OFF(R1)                # list->rank = 0
  j     DONE
```

```
HEAD:
# run through the list and find the HEAD
#       for (i=0; i<list_size; i++)
#         if (list[i].back == NULL) -> found HEAD

  move  R5, R1                          # start at beginning of list array (R1)
  # calculate end of array address
  multi R7, R2, element_SIZE            #  N * element_SIZE
  add   R7, R7, R1

HEAD_LOOP:
  bge   R5, R7, CONT                    # check if reached the end of the array
  lw    R6, back_OFF(R5)                # load list[i].back
  bne   R6, R0, CONT_LOOP               # if (list[i].back == NULL) -> found HEAD
  move  R3, R5                          # list_head (R3) = &(list[i]);
  j     CONT                            # break from the loop
CONT_LOOP:
  addi  R5, R5, element_SIZE            # advance to list[i+1]
  j     HEAD_LOOP

CONT:
  lw    R6, next_OFF(R3)                # load: next(R6) = list_head->next
  move  R5, R3                          # curr(R5) = list_head(R3)

# Forward loop:
#       run through the list: getting to the end to know what is maximal RANK & store
#       the rank-differences (from the MAX) along the way

FWD:
  sub   R7, R0, R4                      # rank_diff = 0 - max_rank
  sw    R7, rank_OFF(R5)                # store the rank-difference
  lw    R8, weight_OFF(R5)              # load curr->weight
  move  R5, R6                          # curr = next
  lw    R6, next_OFF(R5)                # load next's address
  add   R4, R4, R8                      # max_rank += curr->rank
  bne   R6, R0, FWD

  # special treatment for the last element
  sub   R7, R0, R4                      # curr->rank = 0 - max_rank
  sw    R7, rank_OFF(R5)

# Final ranking (backward) loop:
#       for (i=0; i<list_size; i++)
#       { local_list[i].rank += max_rank;
#       }

# First we have the unrolled loop (as many copies as possible) and later the "left-overs"
# Note: COPIES - is the number of copies of the unrolled loop, and will be known
#       by the compiler, according to the machine specifications, as follows:
#       COPIES = floor(R/Ri)
#               where   R  - is the # of available registers to be used for unrolling
#                       Ri - is the # of additional registers for each copy of the loop
```

```
  # setting initializations for the unrolled loop
  move  R5, R1                         # set to starting address of list array
  move  R6, R2                         # set loop index to list_size

FINAL1:
  blt   R6, COPIES, FINAL2             # check if still larger than number of
                                         # COPIES unrolled
  # 1st copy of unrolled loop
  lw    R8, rank_OFF(R5)               # load rank-difference
  add   R8, R8, R4                     # local_list[i] += max_rank
  sw    R8, rank_OFF(R5)               # store final rank
  # 2nd copy of unrolled loop
  lw    R9, element_SIZE+rank_OFF(R5)  # load rank-difference
  add   R9, R9, R4                     # local_list[i] += max_rank
  sw    R9, element_SIZE+rank_OFF(R5)  # store final rank
  # 3rd copy of unrolled loop
  lw    R10, 2*element_SIZE+rank_OFF(R5)# load rank-difference
  add   R10, R10, R4                   # local_list[i] += max_rank
  sw    R10, 2*element_SIZE+rank_OFF(R5)# store final rank

  .                                      # here will be the rest of the
  .                                      # COPIES of the unrolled loop
  .

  # unrolled loop control
  addi  R5, R5, COPIES*element_SIZE    # advance base_address by number of COPIES
  subi  R6, R6, COPIES                 # loop index -= COPIES done
  j     FINAL1

FINAL2:
  blez  R6, DONE                       # check if there are any elements left
  # calculate the address of the last used element in the array:
  # local_list[list_size]
  multi R7, R6, element_SIZE           # total_size = elements left*element_SIZE
  add   R7, R5, R7                     # start_addr + total_size

  lw    R8, rank_OFF(R5)               # load first element's rank-difference

  # finally must rank what is left (and NOT a multiple of the unrolled copies)
FINAL:
  add   R8, R8, R4                     # local_list[i] += max_rank
  sw    R8, rank_OFF(R5)
  addi  R5, R5, element_SIZE           # curr = local_list[i+1]
  lw    R8, rank_OFF(R5)
  blt   R5, R7, FINAL                  # if not reached last address -> loop

DONE:
  jr
```

B.2 Parallel Randomized Coin-Tossing Algorithm (no-cut)

```
#       97/12/19        @(#)par_list_rank.asm   1.23

# List Ranking - Parallel Random Alg. (coin tossing) - Work Optimized (no-cut)
```

```
### Some NOTATIONS:
### ===============
#       1) Load from address SP + list_OFF the word into register R2
#          where SP - Stack Pointer and
#                list_OFF - a constant OFFset where list is on the stack
#
#               "lw  R2,  list_OFF(SP)"
#
#       2) Load from address IN thread-register R2$ + rank_OFF the word into
#          thread-register R3$, where
#               rank_OFF -  a constant OFFset where RANK is in the structure-data-type
#
#               "lb     R3$, rank_OFF(R2$)"

### Some Constant OFFsets and SIZEs meaning:
#   ---------------------------------------
#    rank_OFF        - offset of the FIELD of rank, in the structure of an element
#    select_OFF
#    next_OFF        - offset of the FIELD of next address, in the structure of an element
#    prnt_OFF
#    back_OFF        - offset of "back" - indication if the element is the HEAD of the list
#    list_ptr_OFF    - address of the correponding element in the ORIGINAL list_array
#
#    ptr_SIZE        - size of a pointer (i.e., an address)
#    iter_SIZE
#    list_element_SIZE - size of an element in the original List_Array
#    actv_elmnt_SIZE - size of an element in the active array
#    skpd_SIZE
#    addr_SIZE

##### main loop and book-keeping #####
# C-function:   void parallel_rand_ranking(List_Array *list, int list_size)

      # registers' mapping:
      #       SPWNR   spawn-join counter
      #       R1      current list size (both for fwd & bckwd iterations)
      #       R2      actv_array starting addr to READ elements / list-array start (for 1st)
      #       R3      active-counter
      #       R4      non-active-counter
      #       R5      nactv_array statring addr (for this ROUND)
      #       R6      accumulative-non-active-index (into nactv[]) - till where already used
      #       R7      accumulative-active-index (into actv[]) - till where already used
      #       R8      iteration number
      #       R9      actv_array start addr to WRITE (when copying/shrinking)
      #       R10     1 (for branching comparisons)

  lw    R1, list_size_OFF(SP)                  # check starting size > 1 (i.e.,  <= 1 -> exit)
  ble   R1, R10,  EXIT1

  lw    R2, list_OFF(SP)                       # load pointer to list from STACK
  la    R9, actv(R0)                           # start address of active array (for first iteration)
  li    R7, 0                                  # initialize index into actv array
  j     FIRST_FWDITR           # special preparation for first iteration DIRECTLY on list-array
```

```
END_FIRST_FWDITR:

  # initialization for iteration process (to work on actv[] and nactv[] arrays)
  move  R7, R1                              # actv_indx = list_size (after 1st copy)
  li    R8, 0                               # iter = 0
  la    R5, nactv(R0)                       # init start address of non-active array (curr_nactv)
  li    R6, 0                               # nactv_indx = 0

JFWD:
  # loop on forward parallel iterations (pointer jumping)
  move  R2, R9                              # start address of active array to READ (curr_actv)
  laa   R9, actv(R0), actv_element_SIZE[R7]     # start address of active array to WRITE
  j     FWDITR                              # do one forward parallel iteration
END_FWDITR:                                   # in R1 we have the new current list size

  # iteration oriented information (each round "looks" at different part of the array)
  # calculate and store relevant info for back iterations:
  #               skipped start addr in array, num skipped in this round

  # store nactv_start_addr(R5) & indx (R4) for this iteration (R8)
  swa   R4, bck_num_OFF+iter_info(R0), iter_SIZE[R8]
  swa   R5, bck_start_addr_OFF+iter_info(R0), iter_SIZE[R8]
  addi  R8, R8, 1                    # iter = iter + 1

  add   R7, R7, R3                   # Add the num of actives into accumulative-active-index

  add   R6, R6, R4        # ADD the num of non-actives into the accumulative-non-active-index
  laa   R5, nactv(R0), skpd_SIZE[R6]  # for next round start_address (of non active array)

  bne   R1, R10, JFWD                # if curr != 1 (i.e. larger than) do "fwditr"

  # backward parallel final ranking
JBCK:
  # update iteration oriented information (each round "looks" at different part of the array)
  subi  R8, R8, 1                           # iter = iter - 1
  # load the starting address in nactv_array & number of elements for this iteration
  lwa   R5, bck_start_addr_OFF+iter_info(R0), iter_SIZE[R8]
  lwa   R1, bck_num_OFF+iter_info(R0), iter_SIZE[R8]

  j     BCKITR                              # do ONE backward parallel ranking (all at once)

END_BCKITR:

  subi  R6, R6, R1              # accumulative_non_selected -= num_bck_in_this_iteration (R1)
  bgt   R6, R0, JBCK            # if (accumulative-non-selected > 0) do "bckitr"

EXIT1:
  jr

##### END of main loop and book-keeping #####


######## First ITERATION  (special treatment) #######
# special treatment for FIRST iteration: selection and active-array preparation
# C-function:   void  par_first_fwditr(List_Array *orig_list /*IN*/, int list_size /*IN*/,
```

```
#                                       Actv_Struct *actv /*OUT*/)

FIRST_FWDITR:
  # initialize counters
  li    R3,  0                       # actv_counter = 0

  ### update bits from coin tossing to "select", for first iteration
  ### and prepare active array (copying each element according to its thread-ID)

  # asynch pardo - fork_into_threads  $ = [1..list_size] (R1)
  spawn SPWNR, 0, R1, 8         # where SPWNR is the spawning reg., 0 is the offset & R1 is the size
  #########  HERE   START   THREADS #########
      # threads registers' mapping:
      #       R1$     address of $'s element
      #       R2$     misc
      #       R3$     address in active array
      #       R4$     initial rank (weight) value
      #       R5$     next value
      #       R6$     back value
      #       R7$     element's coin tossing and coin BYTE content & if selected or NOT
      #       R8$     next-index into coin-tossing and the coin BYTE contents


  laa   R1$, 0(R2), list_element_SIZE[R0$]      # load $'s addr (R2 + $*list_element_SIZE)
  # load curr's back value (ASAP for selection since also indicates if HEAD of list)
  lwa   R6$, back_OFF(R2), list_element_SIZE[R0$]
  # load next's address asap
  lwa   R5$, next_OFF(R2), list_element_SIZE[R0$]

  li    R2$, 1                              # init for selection

  # load rest of the available info in an element (from list_array itself!)
  lw    R4$, rank_OFF(R1$)

  # need to "copy" relevant elements into active array
  laa   R3$, 0(R9), actv_elmnt_SIZE[R0$] # load $'s addr in active array (R9 + $*actv_elmnt_SIZE)
  swa   R1$, 0(R9), actv_elmnt_SIZE[R0$]        # store into actv[$] = MY abs addr. in list_array
  # store all the available info in an element
  sw    R4$, rank_OFF(R3$)                      # the initial rank is also the weight
  sw    R5$, next_OFF(R3$)
  sw    R6$, back_OFF(R3$)

  ## do "SELECTION", except for LAST element in the list
  beq   R5$, R0, SELECT1
  # have to check also (COIN[ME->NEXT]==0)
  subi  R8$, R5$, LOCAL_LIST # next_addr - &(local_list); dist. from list-array start (bytes)
  divi  R8$, R8$, list_element_SIZE # next_index / sizeof(List_Array); divided by element-size
  lb    R8$, COIN(R8$)                 # load NEXT's coin tossing BYTE to memory

SELECT1:                                # jumped here if KNOW that am LAST in the list (no NEXT)
  lb    R7$, COIN(R0$)                 # load coin tossing BYTE to memory:  COIN[$]

  # check if I'm the HEAD of the list -> always selected
  beq   R6$, R0, END_SLCTD1
```

```
  # check if I'm the LAST of the list -> never selected
  beq   R5$, R0, END_SLCTD2

  # finish checking for ME & my NEXT
  andi  R7$, R7$, 0x01                      # get relevant BIT - by masking rest:    0x01
  bne   R7$, R10, END_SLCTD2                # if (my_coin!=1)  NOT selected
  andi  R8$, R8$, 0x01                      # get relevant BIT - by masking rest:    0x01
  beq   R8$, R0, END_SLCTD1                 # if (next_coin==0)  than "I'm" YES selected (R2$ has 1 in it)
  # since (COIN[ME]==1) but (COIN[ME->NEXT]!=0) -> NOT selected
END_SLCTD2:
  li    R2$, 0                          # NOT selected
END_SLCTD1:
  sb    R2$, select_OFF(R1$)                # store VALUE of selected[i] BYTE to memory
  sb    R2$, select_OFF(R3$)                # store VALUE of selected to actv

  #########  HERE   END   THREADS #########       # this thread done
  join  SPWNR, R1                           #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $

  # only after the last thread arrived -> done
  j     END_FIRST_FWDITR

######## END of First ITERATION #######


######## FORWARD ITERATION #######
# returns NEW_LIST_SIZE of shrinked list in R1
# The input arguments are in registers:
#       beginning of the list for READ (R2), the list size (R1) & beginning of nactv (R5)
# C-function:   int  par_fwditr(Actv_Struct  *curr_actv /*IN/OUT*/, int  curr_list_size /*IN*/,
#                               Nactv_Struct  *curr_nactv /* OUT*/)

FWDITR:
  # NOTE:  bits in "select" were already updated at end of previous iteration

  # initialize counters
  li    R3,  0                      # actv_counter = 0
  li    R4,  0                      # nactv_counter = 0

  #### includes: skipping, shrinking & update bits from coin tossing to "select", for next iteration

  # asynch pardo - fork_into_threads  $ = [1..list_size] (R1)
  spawn SPWNR, 0, R1, 10        # where SPWNR is the spawning reg., 0 is the offset & list_size (R1) is the size
  #########  HERE   START   THREADS #########
      # threads registers' mapping:
      #       R1$     address of $'s element (which is prnt for "skipped")
      #       R2$     element's selected BYTE content & if selected or NOT
      #       R3$     element's rank
      #       R4$     next-element's LIST address
      #       R5$     next-element's LIST address (to avoid unecessary dependences)
      #       R6$     next-element's selected / element's index into coin-tossing & value of
      #    coin tossing and NEW selected BYTE
      #       R7$     next-element's rank / next-index into coin-tossing
      #       R8$     next-element's LIST address (to aviod unecessary dependences) / misc
      #       R9$     back value
      #       R10$    index to active OR non-active array
```

```
  lwa   R1$, 0(R2), actv_elmnt_SIZE[R0$]        # calculate $'s addr in actv array (R2 + $*addr_size) & load the
                                                  # element's addr

  # checking if an element was selected
  lba   R2$, select_OFF(R2), actv_elmnt_SIZE[R0$] # load each selected BYTE to memory
  bne   R2$, R10, SKPD                          # if (actv[i].select != 1, not selected) jump to end

  # continue since i is selected
  lwa   R3$, rank_OFF(R2), actv_elmnt_SIZE[R0$] # load curr's rank - rank(list($))
  lwa   R9$, back_OFF(R2), actv_elmnt_SIZE[R0$] # load curr's back value from actv-copy
                                                  # (ASAP for selection since also indicates if HEAD of list)

  # advance to next element (till run into a selected element)
  lw    R4$, next_OFF(R2), actv_element_SIZE[R0$] # pre-load 1st's address
ADVNC:
  # skipping the following non-selected nodes (loop unrolled)

  #### 1st (unrolled) node to skip, address loaded at END of loop
  # check if element selected (if so -> done)
  lba   R6$, select_OFF(R4$)                    # load next's selected BYTE to memory
  beq   R6$, R10, CUT                           # if (next(list[i]).selected)  done /* NEXT is selected */

  # continue since next is NOT selected
  lw    R7$, rank_OFF(R4$)                      # load next's rank

  # store "ranking diff" from "ranking parent"
  sw    R3$, rank_OFF(R4$)                      # store rank(i)
  sw    R1$, prnt_OFF(R4$)                      # store "ranking parent" addr

  ### since SKIPPED -> shrink into curr_nactv (not-actv array)
  # prefix-sums (unordered) on NOT "selected" flags to keep array of "skipped"
  li    R10$,   1                               # init for PS
  ps    R4, R10$                                # each thread does this for each skipped to find new indx for it
  # now need to "copy" relevant NON selected (i.e., skipped) elements into NEW array
  swa   R4$, 0(R5), skpd_SIZE[R10$]             # store into nactv[nactv_indx] place in array "SKIPPED abs addr."

  # store to nactv[nactv_indx] the available info in an element
  swa   R3$, rank_OFF(R5), skpd_SIZE[R10$]
  swa   R1$, prnt_addr_OFF(R5), skpd_SIZE[R10$] # store into nactv also prnt_addr - for shorter load

  # calculate new $'s rank & next
  add   R3$, R3$, R7$                           # add ranks:    rank(list(i)) + rank(next(list(i)))

  #### 2nd (unrolled) node to skip                # using a different register to aviod unecessary dependences
  lw    R5$, next_OFF(R4$)                      # load next address - next(curr)
  move  R4$, R5$                                # for correctness of the "CUT" final storing
  beq   R5$, R0, CUT                            # if (next == NULL) -> break

  # check if element selected (if so -> done)
  lb    R6$, select_OFF(R5$)                    # load next's selected BYTE to memory
  beq   R6$, R10, CUT                           # if (next(list[i]).selected)  done /* NEXT is selected */

  # continue since next is NOT selected
  lw    R7$, rank_OFF(R5$)                   # load next's rank - rank(next(list(i)))
```

```
# store "ranking diff" from "ranking parent"
sw    R3$, rank_OFF(R5$)                     # store rank(i)
sw    R1$, prnt_OFF(R5$)                     # store "ranking parent" addr

### since SKIPPED -> shrink into curr_nactv (not-actv array)
# prefix-sums (unordered) on NOT "selected" flags to keep array of "skipped"
li    R10$,   1                              # init for PS
ps    R4, R10$                               # each thread does this for each skipped to find new indx for it
# now need to "copy" relevant NON selected (i.e., skipped) elements into NEW array
swa   R5$, 0(R5), skpd_SIZE[R10$]            # store into nactv[nactv_indx] place in array "SKIPPED abs addr."

# store to nactv[nactv_indx] the available info in an element
swa   R3$, rank_OFF(R5), skpd_SIZE[R10$]
swa   R1$, prnt_addr_OFF(R5), skpd_SIZE[R10$] # store into nactv also prnt_addr - for shorter load

# calculate new $'s rank & next
add   R3$, R3$, R7$                    # add ranks:    rank(list(i)) + rank(next(list(i)))

#### 3rd (unrolled) node to skip               # using a different register to aviod unecessary dependences
lw    R8$, next_OFF(R5$)                     # load next address - next(curr)
move  R4$, R8$                               # for correctness of the "CUT" final storing
beq   R8$, R0, CUT                           # if (next == NULL) -> break

# check if element selected (if so -> done)
lb    R6$, select_OFF(R8$)                   # load next's selected BYTE to memory
beq   R6$, R10, CUT                          # if (next(list[i]).selected)  done /* NEXT is selected */

# continue since next is NOT selected
lw    R7$, rank_OFF(R8$)                     # load next's rank - rank(next(list(i)))

# store "ranking diff" from "ranking parent"
sw    R3$, rank_OFF(R8$)                     # store rank(i)
sw    R1$, prnt_OFF(R8$)                     # store "ranking parent" addr

### since SKIPPED -> shrink into curr_nactv (not-actv array)
# prefix-sums (unordered) on NOT "selected" flags to keep array of "skipped"
li    R10$,   1                              # init for PS
ps    R4, R10$                               # each thread does this for each skipped to find new indx for it
# now need to "copy" relevant NON selected (i.e., skipped) elements into NEW array
swa   R8$, 0(R5), skpd_SIZE[R10$]            # store into nactv[nactv_indx] place in array "SKIPPED abs addr."

# store to nactv[nactv_indx] the available info in an element
swa   R3$, rank_OFF(R5), skpd_SIZE[R10$]
swa   R1$, prnt_addr_OFF(R5), skpd_SIZE[R10$] # store into nactv also prnt_addr - for shorter load

# calculate new $'s rank & next
add   R3$, R3$, R7$                          # add ranks:    rank(list(i)) + rank(next(list(i)))

# Note:  here further copies of the loop can be unrolled if desired

lw    R4$, next_OFF(R8$)                     # load next address - next(curr)
bne   R4$, R0, ADVNC                         # if (next == NULL) -> break (otherwise continue while-loop)
# end of unrolled loop for skipping
```

```
CUT:
  # store updated info in original-list
  sw    R3$, rank_OFF(R1$)                  # store new rank: rank(list(i)) = rank(list(i)) + rank(next(list(i)))
  sw    R4$, next_OFF(R1$)                  # store new next: next(list(i)) = next(next(list(i)))...

  ### since SELECTED copy-shrink into next active array and do the selection process for the next iteration
  # prefix-sums (unordered) on "active" (e.g., previously selected) to shrink into actv array
  li    R10$,   1                           # init for PS
  ps    R3,  R10$                        # each thread does this for each active find new indx

  # now need to "copy"/"shrink" relevant elements into NEW array
  swa   R1$, 0(R9), actv_elmnt_SIZE[R10$]       # store into actv[R9+R10$*SIZE] place in array MY abs addr.
  # store all the available/updated info in an element to NEW actv array
  swa   R3$, rank_OFF(R9), actv_elmnt_SIZE[R10$]
  swa   R4$, next_OFF(R9), actv_elmnt_SIZE[R10$]
  swa   R9$, back_OFF(R9), actv_elmnt_SIZE[R10$]

  # random selection process based on coin-tossing (on the active ones -> do "SELECTION")
  # except for LAST element in the list
  beq   R4$, R0, SELECT
  li    R8$, 1                           # init for selection

  # load  (COIN[ME->NEXT]==0)
  subi  R7$, R4$, LOCAL_LIST # next_addr - &(local_list); distance from list-array start (bytes)
  divi  R7$, R7$, list_element_SIZE # next_index / sizeof(List_Array); divided by element-size
  lb    R7$, COIN(R7$)                # load next coin tossing BYTE to memory

SELECT:                              # jumped here if KNOW that am LAST in the list (no NEXT)
  # load COIN[ME]
  subi  R6$, R1$, LOCAL_LIST # addr - &(local_list); distance from list-array start (bytes)
  divi  R6$, R6$, list_element_SIZE # my_index / sizeof(List_Array); divided by element-size
  lb    R6$, COIN(R6$)                # load coin tossing BYTE to memory

  # check if I'm the HEAD of the list -> always selected
  beq   R9$, R0, END_SLCTD

  # check if I'm the LAST of the list -> never selected
  beq   R4$, R0, END_NOT_SLCTD

  #finish checking for ME & my NEXT
  andi  R6$, R6$, 0x01                      # get relevant BIT - by masking rest:    0x01
  bne   R6$, R10, END_NOT_SLCTD             # if (my_coin!=1)  NOT selected
  andi  R7$, R7$, 0x01                      # get relevant BIT - by masking rest:    0x01
  beq   R7$, R0,  END_SLCTD                 # if (next_coin==0)  than "I'm" YES selected (R8$ has 1 in it)
  # here since (COIN[ME]==1) but (COIN[ME->NEXT]!=0) -> NOT selected
END_NOT_SLCTD:
  li    R8$, 0                           # NOT selected
END_SLCTD:
  sb    R8$, select_OFF(R1$)                # store VALUE of selected[i] BYTE to memory
  sba   R8$, select_OFF(R9), actv_elmnt_SIZE[R10$] # store VALUE of selected to NEW actv

  # finished the skipping, shrinking & selection for next iteration
SKPD:                                       # this thread done
  #########  HERE   END   THREADS #########
  # synchronize all threads
```

```
  join  SPWNR, R1                              #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $

  # only now (the last thread to arrive is the only one alive) will we end "fwditr"
  move  R1,  R3                     # get final actv_counter (new list size) and put it in R1
  j     END_FWDITR

######## END of FORWARD ITERATION #######



######## BACKWARD ITERATION (final ranking) #######
# The input arguments are in registers:
#       starting address in skipped array (R5) & the number of skipped
#       elements in this iteration (R1)
# C-function:  void  par_bckitr(Nactv_Struct  *curr_nactv /*IN*/, int  size /*IN*/)

BCKITR:

  # asynch pardo - fork_into_threads  $ = [1..list_size] (R1)
  spawn SPWNR, 0, R1, 4         # where SPWNR is the spawning reg., 0 is the offset & list_size (R1) is the size
  #########  HERE   START   THREADS #########
      # threads registers' mapping:
      #       R1$     address of $'s element
      #       R2$     misc, temporary storage, etc.
      #       R3$     element's rank
      #       R4$     element's ranking_prnt addr.

  lwa   R1$, 0(R5), skpd_SIZE[R0$]             # calculate $'s addr in actv array (R5 + $*array_entry_size) &
                                                 # load the element's addr.
  lwa   R3$, rank_OFF(R5), skpd_SIZE[R0$]      # load curr's rank - rank(list($))
  lwa   R4$, prnt_addr_OFF(R5), skpd_SIZE[R0$]  # load curr's rank_prnt addr. (from the skipped array)

  lw    R2$, rank_OFF(R4$)                  # load rank_parent's rank - rank(rank_prnt)
  # calculate & store:     "ranking parent" - "ranking diff"
  sub   R3$, R2$, R3$
  sw    R3$, rank_OFF(R1$)

DONE2:                                           # this thread done
  #########  HERE   END   THREADS #########
  # synchronize all threads before ending (and starting next round)
  join  SPWNR, R1                              #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $

  # only now (the last thread to arrive is the only one alive) will we end "bckitr"
  j     END_BCKITR

######## END of BACKWARD ITERATION #######
```

B.3 Time-Optimized Parallel Coin-Tossing Algorithm (cut-6)

```
#       97/12/23        @(#)par_list_rank.asm   1.19.1.3

# List Ranking - Parallel Random Alg. (coin tossing) - Time-Optimized algorithm (cut-6)
# (NO duplication of "next" in the active array)
```

```
##### main loop and book-keeping #####
# C-function:   void parallel_rand_ranking(List_Array *list, int list_size)

      # registers' mapping:
      #        SPWNR   spawn-join counter
      #        R1      current list size (both for fwd & bckwd iterations)
      #        R2      actv_array starting addr to READ elements / list-array start (for 1st)
      #        R3      active-counter
      #        R4      non-active-counter
      #        R5      nactv_array statring addr (for this ROUND)
      #        R6      accumulative-non-active-index (into nactv[]) - till where already used
      #        R7      accumulative-active-index (into actv[]) - till where already used
      #        R8      iteration number
      #        R9      actv_array start addr to WRITE (when copying/shrinking)
      #        R10     always 1 (for branching comparisons, already loaded)

  lw    R1, list_size_OFF(SP)                 # check starting size > 1 (i.e.,  <= 1 -> exit)
  ble   R1, R10, EXIT1

  lw    R2, list_OFF(SP)                      # load pointer to list from STACK
  la    R9, actv(R0)                          # start address of active array (for first iteration)
  li    R7, 0                                 # initialize index into actv array
  j     FIRST_FWDITR          # special preparation for first iteration DIRECTLY on list-array

END_FIRST_FWDITR:

  # initialization for iteration process (to work on actv[] and nactv[] arrays)
  move  R7, R1                                # actv_indx = list_size (after 1st copy)
  li    R8, 0                                 # iter = 0
  la    R5, nactv(R0)                         # init start address of non-active array (curr_nactv)
  li    R6, 0                                 # nactv_indx = 0

JFWD:
  # loop on forward parallel iterations (pointer jumping)
  move  R2, R9                                # start address of active array to READ (curr_actv)
  laa   R9, actv(R0), actv_element_SIZE[R7]   # start address of active array to WRITE
  j     FWDITR                                # do one forward parallel iteration
END_FWDITR:                                     # in R1 we have the new current list size

  # iteration oriented information (each round "looks" at different part of the array)
  # calculate and store relevant info for back iterations:
  #               skipped start addr in array, num skipped in this round

  # store nactv_start_addr(R5) & indx (R4) for this iteration (R8)
  swa   R4, bck_num_OFF+iter_info(R0), iter_SIZE[R8]
  swa   R5, bck_start_addr_OFF+iter_info(R0), iter_SIZE[R8]
  addi  R8, R8, 1                     # iter = iter + 1

  add   R7, R7, R3                    # Add the num of actives into accumulative-active-index

  add   R6, R6, R4        # ADD the num of non-actives into the accumulative-non-active-index
  laa   R5, nactv(R0), skpd_SIZE[R6]  # for next round start_address (of non active array)

  bne   R1, R10, JFWD                 # if curr != 1 (i.e. larger than) do "fwditr"
```

```
  # backward parallel final ranking
JBCK:
  # update iteration oriented information (each round "looks" at different part of the array)
  subi  R8, R8, 1                             # iter = iter - 1
  # load the starting address in nactv_array & number of elements for this iteration
  lwa   R5, bck_start_addr_OFF+iter_info(R0), iter_SIZE[R8]
  lwa   R1, bck_num_OFF+iter_info(R0), iter_SIZE[R8]

  j     BCKITR                                # do ONE backward parallel ranking (all at once)

END_BCKITR:

  subi  R6, R6, R1              # accumulative_non_selected -= num_bck_in_this_iteration (R1)
  bgt   R6, R0, JBCK            # if (accumulative-non-selected > 0) do "bckitr"

EXIT1:
  jr

##### END of main loop and book-keeping #####


######## First ITERATION  (special treatment) #######
# special treatment for FIRST iteration: selection and active-array preparation
# C-function:   void  par_first_fwditr(List_Array *orig_list /*IN*/, int list_size /*IN*/,
#                                      Actv_Struct *actv /*OUT*/)

FIRST_FWDITR:
  # initialize counters
  li    R3,  0                       # actv_counter = 0

  ### update bits from coin tossing to "select", for first iteration
  ### and prepare active array (copying each element according to its thread-ID)

  # asynch pardo - fork_into_threads  $ = [1..list_size] (R1)
  spawn SPWNR, 0, R1, 8   # where SPWNR is the spawning reg., 0 is the offset & R1 is the size
  #########  HERE   START   THREADS #########
      # threads registers' mapping:
      #       R1$     address of $'s element
      #       R2$     misc
      #       R3$     address in active array
      #       R4$     initial rank (weight) value
      #       R5$     next value
      #       R6$     back value
      #       R7$     element's coin tossing and coin BYTE content & if selected or NOT
      #       R8$     next-index into coin-tossing and the coin BYTE contents


  laa   R1$, 0(R2), list_element_SIZE[R0$]      # load $'s addr (R2 + $*list_element_SIZE)
  # load curr's back value (ASAP for selection since also indicates if HEAD of list)
  lwa   R6$, back_OFF(R2), list_element_SIZE[R0$]
  # load next's address asap
  lwa   R5$, next_OFF(R2), list_element_SIZE[R0$]

  li    R2$, 1                               # init for selection
```

```
  # load rest of the available info in an element (from list_array itself!)
  lw    R4$, rank_OFF(R1$)

  # need to "copy" relevant elements into active array
  laa   R3$, 0(R9), actv_elmnt_SIZE[R0$] # load $'s addr in active array (R9 + $*actv_elmnt_SIZE)
  swa   R1$, 0(R9), actv_elmnt_SIZE[R0$]       # store into actv[$] = MY abs addr. in list_array
  sw    R3$, actv_ptr_OFF(R1$)                 # store into list_array MY place in current-actv (abs addr)
  # store all the available info in an element
  sw    R4$, rank_OFF(R3$)                     # the initial rank is also the weight
  sw    R5$, next_OFF(R3$)
  sw    R6$, back_OFF(R3$)

  ## do "SELECTION", except for LAST element in the list
  beq   R5$, R0, SELECT1
  # have to check also (COIN[ME->NEXT]==0)
  subi  R8$, R5$, LOCAL_LIST # next_addr - &(local_list); distance from list-array start (bytes)
  divi  R8$, R8$, list_element_SIZE # next_index / sizeof(List_Array); divided by element-size
  lb    R8$, COIN(R8$)             # load NEXT's coin tossing BYTE to memory

SELECT1:                           # jumped here if KNOW that am LAST in the list (no NEXT)
  lb    R7$, COIN(R0$)             # load coin tossing BYTE to memory:  COIN[$]

  # check if I'm the HEAD of the list -> always selected
  beq   R6$, R0, END_SLCTD1

  # check if I'm the LAST of the list -> never selected
  beq   R5$, R0, END_SLCTD2

  # finish checking for ME & my NEXT
  andi  R7$, R7$, 0x01                         # get relevant BIT - by masking rest:    0x01
  bne   R7$, R10, END_SLCTD2                   # if (my_coin!=1)  NOT selected
  andi  R8$, R8$, 0x01                         # get relevant BIT - by masking rest:    0x01
  beq   R8$, R0,  END_SLCTD1                   # if (next_coin==0)  than "I'm" YES selected (R2$ has 1 in it)
  # since (COIN[ME]==1) but (COIN[ME->NEXT]!=0) -> NOT selected
END_SLCTD2:
  li    R2$, 0                                 # NOT selected
END_SLCTD1:
  sb    R2$, select_OFF(R1$)                   # store VALUE of selected[i] BYTE to memory
  sb    R2$, select_OFF(R3$)                   # store VALUE of selected to actv

DONE1:  # store both to the actv AND original list
  sb    R0, done_OFF(R1$)                      # initialize curr->done = 0
  sb    R0, done_OFF(R3$)
                                                 # this thread done
  #########  HERE   END   THREADS #########
  join  SPWNR, R1                              #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $

  # only after the last thread arrived -> done
  j     END_FIRST_FWDITR

######## END of First ITERATION #######


######## FORWARD ITERATION #######
```

```
# returns NEW_LIST_SIZE of shrinked list in R1
# The input arguments are in registers:
#       beginning of the list for READ (R2), the list size (R1) & beginning of nactv (R5)
# C-function:   int  par_fwditr(Actv_Struct  *curr_actv /*IN/OUT*/, int  curr_list_size /*IN*/,
#                               Nactv_Struct  *curr_nactv /* OUT*/)

FWDITR:
  # NOTE:  bits in "select" were already updated at end of previous iteration

  # initialize counters
  li    R3,  0                       # actv_counter = 0
  li    R4,  0                       # nactv_counter = 0

  # asynch pardo - fork_into_threads  $ = [1..list_size] (R1)
  spawn SPWNR, 0, R1, 9         # where SPWNR is the spawning reg., 0 is the offset & list_size (R1) is the size
  #########  HERE   START   THREADS #########
      # threads registers' mapping:
      #       R1$     address of $'s element
      #       R2$     element's selected BYTE content & if selected or NOT & if done or NOT
      #       R3$     element's rank
      #       R4$     next-element's LIST address
      #       R5$     next-element's LIST address (to avoid unecessary dependences)
      #       R6$     next-element's selected
      #       R7$     next-element's rank
      #       R8$     next-element's LIST address (to aviod unecessary dependences)
      #       R9$     next-element's ACTV address

  lwa   R1$, 0(R2), actv_elmnt_SIZE[R0$]        # calculate $'s addr in actv array (R2 + $*addr_size) & load the
                                                  # element's addr

  # checking if an element was selected
  lba   R2$, select_OFF(R2), actv_elmnt_SIZE[R0$] # load each selected[i] BYTE to memory
  bne   R2$, R10, SKPD                          # if (selected[i] is not selected) jump

  # continue since i is selected
  lwa   R3$, rank_OFF(R2), actv_elmnt_SIZE[R0$] # load curr's rank - rank(list($))

# start of time-optimization (6 elements are skipped at most)
  # advance to next element (till run into a selected element OR till skipped k=6)
ADVNC:
  #### 1st node to skip
  lw    R4$, next_OFF(R2), actv_element_SIZE[R0$] # load next's address

  # check if element selected (if so -> done)
  lba   R6$, select_OFF(R4$)                    # load next's selected BYTE to memory
  beq   R6$, R10, CUT                           # if (next(list[i]).selected)  done /* NEXT is selected */

  # continue since next is NOT selected
  lw    R7$, rank_OFF(R4$)                      # load next's rank - rank(next(list(i)))
  lw    R9$, actv_ptr_OFF(R4$)                  # load the address in the actv (from the LIST array itself)

  # store "ranking diff" from "ranking parent"
  sw    R3$, rank_OFF(R4$)                      # store rank(i)
  sw    R1$, prnt_OFF(R4$)                      # store "ranking parent" addr
```

```
# calculate new $'s rank & next
add   R3$, R3$, R7$                           # add ranks:    rank(list(i)) + rank(next(list(i)))

sb    R2$, done_OFF(R4$)                      # set done(next(list(i)...) = 1 (since skipped it)
sb    R2$, done_OFF(R9$)                      # save through PTR from list_array to active

#### 2nd node to skip                           # using a different register to aviod unecessary dependences
lw    R5$, next_OFF(R4$)                      # load next address - next(curr)
move  R4$, R5$                                # for correctness of the "CUT" final storing
beq   R5$, R0, CUT                            # if (next == NULL) -> break

# check if element selected (if so -> done)
lb    R6$, select_OFF(R5$)                    # load next's selected BYTE to memory
beq   R6$, R10, CUT                           # if (next(list[i]).selected)  done /* NEXT is selected */

# continue since next is NOT selected
lw    R7$, rank_OFF(R5$)                      # load next's rank - rank(next(list(i)))
lw    R9$, actv_ptr_OFF(R5$)                  # load the address in the actv (from the LIST array itself)

# store "ranking diff" from "ranking parent"
sw    R3$, rank_OFF(R5$)                      # store rank(i)
sw    R1$, prnt_OFF(R5$)                      # store "ranking parent" addr

# calculate new $'s rank & next
add   R3$, R3$, R7$                           # add ranks:    rank(list(i)) + rank(next(list(i)))

sb    R2$, done_OFF(R5$)                      # set done(next(list(i)...) = 1 (since skipped it)
sb    R2$, done_OFF(R9$)                      # save through PTR from list_array to active

#### 3rd node to skip                           # using a different register to aviod unecessary dependences
lw    R8$, next_OFF(R5$)                      # load next address - next(curr)
move  R4$, R8$                                # for correctness of the "CUT" final storing
beq   R8$, R0, CUT                            # if (next == NULL) -> break

# check if element selected (if so -> done)
lb    R6$, select_OFF(R8$)                    # load next's selected BYTE to memory
beq   R6$, R10, CUT                           # if NEXT is selected -> done

# continue since next is NOT selected
lw    R7$, rank_OFF(R8$)                      # load next's rank - rank(next(list(i)))
lw    R9$, actv_ptr_OFF(R8$)                  # load the address in the actv (from the LIST array itself)

# store "ranking diff" from "ranking parent"
sw    R3$, rank_OFF(R8$)                      # store rank(i)
sw    R1$, prnt_OFF(R8$)                      # store "ranking parent" addr

# calculate new $'s rank & next
add   R3$, R3$, R7$                           # add ranks:    rank(list(i)) + rank(next(list(i)))

sb    R2$, done_OFF(R8$)                      # set done(next(list(i)...) = 1 (since skipped it)
sb    R2$, done_OFF(R9$)                      # save through PTR from list_array to active
                                                # other info NOT necessary to update in active!
#### 4th node to skip
lw    R4$, next_OFF(R8$)                      # load next address - next(curr)
beq   R4$, R0, CUT                            # if (next == NULL) -> break
```

```
# check if element selected (if so -> done)
lb    R6$, select_OFF(R4$)                 # load next's selected BYTE to memory
beq   R6$, R10, CUT                        # if NEXT is selected -> done

# continue since next is NOT selected
lw    R7$, rank_OFF(R4$)                   # load next's rank
lw    R9$, actv_ptr_OFF(R4$)               # load the address in the actv (from the LIST array itself)

# store "ranking diff" from "ranking parent"
sw    R3$, rank_OFF(R4$)                   # store rank(i)
sw    R1$, prnt_OFF(R4$)                   # store "ranking parent" addr

# calculate new $'s rank & next
add   R3$, R3$, R7$                        # add ranks:    rank(list(i)) + rank(next(list(i)))

sb    R2$, done_OFF(R4$)                   # set DONE = 1 (since skipped it)
sb    R2$, done_OFF(R9$)                   # save DONE through PTR from list_array to active

#### 5th node to skip                        # using a different register to aviod unecessary dependences
lw    R5$, next_OFF(R4$)                   # load next address - next(curr)
move  R4$, R5$                             # for correctness of the "CUT" final storing
beq   R5$, R0, CUT                         # if (next == NULL) -> break

# check if element selected (if so -> done)
lb    R6$, select_OFF(R5$)                 # load next's selected BYTE to memory
beq   R6$, R10, CUT                        # if NEXT is selected -> done

# continue since next is NOT selected
lw    R7$, rank_OFF(R5$)                   # load next's rank
lw    R9$, actv_ptr_OFF(R5$)               # load the address in the actv (from the LIST array itself)

# store "ranking diff" from "ranking parent"
sw    R3$, rank_OFF(R5$)                   # store rank(i)
sw    R1$, prnt_OFF(R5$)                   # store "ranking parent" addr

# calculate new $'s rank & next
add   R3$, R3$, R7$                        # add ranks:    rank(list(i)) + rank(next(list(i)))

sb    R2$, done_OFF(R5$)                   # set DONE = 1 (since skipped it)
sb    R2$, done_OFF(R9$)                   # save DONE through PTR from list_array to active

#### 6th node to skip                        # using a different register to aviod unecessary dependences
lw    R8$, next_OFF(R5$)                   # load next address - next(curr)
move  R4$, R8$                             # for correctness of the "CUT" final storing
beq   R8$, R0, CUT                         # if (next == NULL) -> break

# check if element selected (if so -> done)
lb    R6$, select_OFF(R8$)                 # load next's selected BYTE to memory
beq   R6$, R10, CUT                        # if NEXT is selected -> done

# continue since next is NOT selected
lw    R7$, rank_OFF(R8$)                   # load next's rank
lw    R9$, actv_ptr_OFF(R8$)               # load the address in the actv (from the LIST array itself)
```

```
  # store "ranking diff" from "ranking parent"
  sw    R3$, rank_OFF(R8$)                      # store rank(i)
  sw    R1$, prnt_OFF(R8$)                      # store "ranking parent" addr

  # calculate new $'s rank & next
  add   R3$, R3$, R7$                           # add ranks:    rank(list(i)) + rank(next(list(i)))

  sb    R2$, done_OFF(R8$)                      # set DONE = 1 (since skipped it)
  sb    R2$, done_OFF(R9$)                      # save DONE through PTR from list_array to active
                                                  # other info NOT necessary to update in active!
  # need to have NEXT updated at the end
  lw    R4$, next_OFF(R8$)                      # load next address - next(curr)

CUT:
  # store updated info both in original-elements and in current active-array
  sw    R3$, rank_OFF(R1$)                      # store new rank
  sw    R4$, next_OFF(R1$)                      # store new next
  swa   R3$, rank_OFF(R2), actv_elmnt_SIZE[R0$] # store new rank in actv-array
  swa   R4$, next_OFF(R2), actv_elmnt_SIZE[R0$] # store new next in actv-array

SKPD:
                                                  # this thread done
  #########  HERE   END   THREADS #########
  # synchronize all threads before starting next round
  join  SPWNR, R1                               #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $


  #### shrinking must be in SEPARATE spawn due to the optimization!!!
  #### update bits from coin tossing to "select", for next iteration

  # asynch pardo - fork_into_threads  $ = [1..list_size] (R1)
  spawn SPWNR, 0, R1, 11  # where SPWNR is the spawning reg., 0 is the offset & R1 is the size
  #########  HERE   START   THREADS #########
      # threads registers' mapping:
      #       R1$     address of $'s element
      #       R2$     done value
      #       R3$     index to active OR non-active array
      #       R4$     rank value
      #       R5$     next value
      #       R6$     back value
      #       R7$     element's coin tossing and selected BYTE content & if selected or NOT
      #       R8$     index into coin-tossing
      #       R9$     next-index into coin-tossing
      #       R10$    misc
      #       R11$    prnt value

  lwa   R1$, 0(R2), actv_elmnt_SIZE[R0$]        # calculate $'s addr in actv array (R2 + $*addr_size) & load the
                                                # element's addr
  lba   R2$, done_OFF(R2), actv_elmnt_SIZE[R0$] # load curr's done value
  lwa   R6$, back_OFF(R2), actv_elmnt_SIZE[R0$] # load curr's back value from actv-copy
                                                # (ASAP for selection since also indicates if HEAD of list)
  lwa   R5$, next_OFF(R2), actv_elmnt_SIZE[R0$] # load next's address

  li    R3$, 1                                  # init for PS
```

```
  # load rest of the available info in an element (from list_array itself!)
  lw    R4$, rank_OFF(R1$)
  lw    R11$, prnt_OFF(R1$)                 # prnt-addr for non-active array

  bne   R2$, R0, NON_ACTV                   # if (done != 0) goto "non_actv" -> shrink to non_actv array

  # prefix-sums (unorderd) on "not-done" to shrink into actv array
  ps    R3, R3$                             # each thread does this for each not-done find new indx

  # now need to "copy"/"shrink" relevant elements into NEW array
  laa   R10$, 0(R9), actv_elmnt_SIZE[R3$]   # get NEW addres in active array
  swa   R1$, 0(R9), actv_elmnt_SIZE[R3$]    # store into actv[R9+R3$*SIZE] place in array MY abs addr.
  sw    R10$, actv_ptr_OFF(R1$)             # store into list_array MY place in current-actv (abs addr)
  # store all the available info in an element
  swa   R4$, rank_OFF(R9), actv_elmnt_SIZE[R3$]
  swa   R5$, next_OFF(R9), actv_elmnt_SIZE[R3$]
  swa   R6$, back_OFF(R9), actv_elmnt_SIZE[R3$]

  # except for LAST element in the list
  beq   R5$, R0, SELECT
  li    R10$,   1                           # init for selection

  # on the active ones -> do "SELECTION"
  # load  (COIN[ME->NEXT]==0)
  subi  R9$, R5$, LOCAL_LIST # next_addr - &(local_list); distance from list-array start (bytes)
  divi  R9$, R9$, list_element_SIZE # next_index / sizeof(List_Array); divided by element-size
  lb    R9$, COIN(R9$)             # load next coin tossing BYTE to memory

SELECT:                            # jumped here if KNOW that am LAST in the list (no NEXT)
  # load COIN[ME]
  subi  R8$, R1$, LOCAL_LIST # addr - &(local_list); distance from list-array start (bytes)
  divi  R8$, R8$, list_element_SIZE # my_index / sizeof(List_Array); divided by element-size
  lb    R7$, COIN(R8$)             # load coin tossing BYTE to memory

  # check if I'm the HEAD of the list -> always selected
  beq   R6$, R0, END_SLCTD

  # check if I'm the LAST of the list -> never selected
  beq   R5$, R0, END_NOT_SLCTD

  #finish checking for ME & my NEXT
  andi  R7$, R7$, 0x01                      # get relevant BIT - by masking rest:    0x01
  bne   R7$, R10, END_NOT_SLCTD             # if (my_coin!=1)  NOT selected
  andi  R9$, R9$, 0x01                      # get relevant BIT - by masking rest:    0x01
  beq   R9$, R0,  END_SLCTD                 # if (next_coin==0)  than "I'm" YES selected (R10$ has 1 in it)
  # here since (COIN[ME]==1) but (COIN[ME->NEXT]!=0) -> NOT selected
END_NOT_SLCTD:
  li    R10$,   0                           # NOT selected
END_SLCTD:
  sb    R10$, select_OFF(R1$)               # store VALUE of selected[i] BYTE to memory
  sba   R10$, select_OFF(R9), actv_elmnt_SIZE[R3$] # store VALUE of selected to actv
  # initialize DONE=0 for next round - both to the actv AND original list
  sb    R0, done_OFF(R1$)                   # initialize curr->done = 0
  sba   R0, done_OFF(R9), actv_elmnt_SIZE[R3$]
  j     DONE
```

```
NON_ACTV:       # prefix-sums (unordered) on NOT "selected" flags to keep array of "skipped"
  ps    R4, R3$                            # each thread does this for each done to find new indx for it
  # now need to "copy" relevant NON selected (i.e., done) elements into NEW array
  swa   R1$, 0(R5), skpd_SIZE[R3$]         # store into nactv[nactv_indx] place in array "SKIPPED abs addr."

  # store to nactv[nactv_indx] the available info in an element
  swa   R4$, rank_OFF(R5), skpd_SIZE[R3$]
  swa   R11$, prnt_addr_OFF(R5), skpd_SIZE[R3$] # store into nactv also prnt_addr - for shorter load

DONE:                                        # this thread done
  #########  HERE   END   THREADS #########
  join  SPWNR, R1                          #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $

  # only now (the last thread to arrive is the only one alive) will we end "fwditr"
  move  R1, R3                    # get final actv_counter (new list size) and put it in R1
  j     END_FWDITR

######## END of FORWARD ITERATION #######


######## BACKWARD ITERATION (final ranking) #######
# The input arguments are in registers:
#       starting address in skipped array (R5) & the number of skipped
#       elements in this iteration (R1)
# C-function:  void  par_bckitr(Nactv_Struct  *curr_nactv /*IN*/, int  size /*IN*/)

BCKITR:

  # asynch pardo - fork_into_threads  $ = [1..list_size] (R1)
  spawn SPWNR, 0, R1, 4   # where SPWNR is the spawning reg., 0 is the offset & list_size (R1) is the size
  #########  HERE   START   THREADS #########
      # threads registers' mapping:
      #       R1$     address of $'s element
      #       R2$     misc, temporary storage, etc.
      #       R3$     element's rank
      #       R4$     element's ranking_prnt addr.

  lwa   R1$, 0(R5), skpd_SIZE[R0$]             # calculate $'s addr in actv array (R5 + $*array_entry_size) &
                                               # load the element's addr.
  lwa   R3$, rank_OFF(R5), skpd_SIZE[R0$]      # load curr's rank - rank(list($))
  lwa   R4$, prnt_addr_OFF(R5), skpd_SIZE[R0$]  # load curr's rank_prnt addr. (from the skipped array)

  lw    R2$, rank_OFF(R4$)                     # load rank_parent's rank - rank(rank_prnt)
  # calculate & store:     "ranking parent" + "ranking diff"
  sub   R3$, R2$, R3$
  sw    R3$, rank_OFF(R1$)

DONE2:                                       # this thread done
  #########  HERE   END   THREADS #########
  # synchronize all threads before ending (and starting next round)
  join  SPWNR, R1                          #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $

  # only now (the last thread to arrive is the only one alive) will we end "bckitr"
  j     END_BCKITR
```

```
######## END of BACKWARD ITERATION #######
```

B.4 Wyllie's Parallel Algorithm

```
#       97/12/23        @(#)par_wyllie.asm      1.5

##### Wyllie ranking #####
# C-function:   void parallel_wyllie_ranking(List_Array *list, int list_size)

      # registers' mapping:
      #       SPWNR   spawn-join counter
      #       R1      current list size (both for fwd & bckwd iterations)
      #       R2      actv_array
      #       R3      array to READ elements
      #       R4      pointer jumping iteration number
      #       R5      number of bits in a integer (for calculating log2(n))
      #       R6      1 - for comparisons / array to WRITE elements

  # assuming inlineing - so R1 (list size) & R2 (actv array) are already updated by caller
  li    R6, 1
  ble   R1, R6, EXIT_WYLLIE            # check starting size > 1 (i.e.,  <= 1 -> exit)

  ### we want a list with INDEXES (as opposed to pointers/addresses),
  #   so that we can use alternating READ and WRITE lists

  li    R3, list1_ADDR                 # set R3 to the "spare space"
  # prepare the first READ-list
  spawn SPWNR, 0, R1, 2        # where SPWNR is the spawning reg., 0 is the offset & R1 is the size
  #########  HERE   START   THREADS #########
      # threads registers' mapping:
      #       R1$     next's address & index in actv_array
      #       R2$     rank

  lwa   R1$, next_OFF(R2), actv_element_SIZE[R0$]       # load next's address asap
  lwa   R2$, rank_OFF(R2), actv_element_SIZE[R0$]       # load rank

  beq   R1$, R0, END_PREPARE                            # check if not LAST
  lw    R1$, actv_ptr_OFF(R1$)                          # load next's actv_array address
  # calculate next's index in actv_array:  next_index = (next - actv_list) / sizeof(Actv_Struct);
  sub   R1$, R1$, R2
  divi  R1$, R1$, actv_element_SIZE

END_PREPARE:
  swa   R2$, rank_OFF(R3), element_SIZE[R0$]            # store rank in new READ-list
  swa   R1$, next_OFF(R3), element_SIZE[R0$]            # store next as INDEX

  #########  HERE   END   THREADS #########       # this thread done
  join  SPWNR, R1                              #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $

  li    R6, list2_ADDR                         # set R6 to the additional "spare space" (WRITE)

  li    R5, bits_in_word_SIZE
```

```
  # calculate ceil(log2(R1)) and put it in R4 - number of iterations necessary
  spawn SPWNR, 0, R5, 2   # where SPWNR is the spawning reg., 0 is the offset & R5 is the size
  #########  HERE   START   THREADS #########
      # threads registers' mapping:
      #       R1$     the result of the shifting
      #       R2$     the value of 1

  li    R2$, 1
  srlv  R1$, R1, R0$            # shift value in R1 right by the thread ID (which is a BIT number)
  bne   R1$, R2$, DONE_LOG2     # if ((R1>>$) != 1) -> only the MSG bit thread WRITES to R4
  sllv  R1$, R1$, R0$           # shift back to check "ceil"
  beq   R1$, R1, EXACT_POWER2   # check if it's an exact power of 2
  addi  R4, R0$, 1              # ceil(log2(R1))
  j     DONE_LOG2
EXACT_POWER2:
  move  R4, R0$                 # log2(R1)
DONE_LOG2:
  #########  HERE   END   THREADS #########       # this thread done
  join  SPWNR, R5             #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $


JFWD_WYLLIE:
  # loop on forward parallel iterations (pointer jumping)

  ## figure out new rank & next values
  # asynch pardo - fork_into_threads  $ = [1..list_size] (R1)
  spawn SPWNR, 0, R1, 3   # where SPWNR is the spawning reg., 0 is the offset & R1 is the size
  #########  HERE   START   THREADS #########
      # threads registers' mapping:
      #       R1$     index of $'s next element/ index of next's next
      #       R2$     rank of $'s element
      #       R3$     rank of next's

  lwa   R1$, next_OFF(R3), element_SIZE[R0$]            # load next's index
  lwa   R2$, rank_OFF(R3), element_SIZE[R0$]            # load rank

  beq   R1$, R0, END_SKP                                # check if not LAST
  # load next's next index & next's rank
  lwa   R3$, rank_OFF(R3), element_SIZE[R1$]
  lw    R1$, next_OFF(R3), element_SIZE[R1$]

  beq   R1$, R0, END_SKP                        # check if not already pointing to LAST
  # update new_rank & new_next
  add   R2$, R2$, R3$                           # rank + next's rank
END_SKP:
  swa   R2$, rank_OFF(R6), element_SIZE[R0$]
  swa   R1$, next_OFF(R6), element_SIZE[R0$]

  #########  HERE   END   THREADS #########       # this thread done
  join  SPWNR, R1                         #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $

  ## we will be switching between two copies of the "list" (READ & WRITE)
  xor   R6, R6, R3
  xor   R3, R6, R3                              # R6 will always be written into
```

```
  xor   R6, R6, R3                              # R3 will always be read from

  ## checking if continue looping
  subi  R4, R4, 1                               # iter = iter - 1
  bgt   R4, R0, JFWD_WYLLIE                     # if iter > 0 loop again


  ### After ranking is done have to update info in the original-array
  # asynch pardo - fork_into_threads  $ = [1..list_size] (R1)
  spawn SPWNR, 0, R1, 2   # where SPWNR is the spawning reg., 0 is the offset & R1 is the size
  ######### HERE   START   THREADS #########
      # threads registers' mapping:
      #       R1$     address of $'s element in original list array
      #       R2$     new rank of $'s element

  # load $'s address in original list
  lwa   R1$, 0(R2), actv_element_SIZE[R0$]
  # load updated rank
  lwa   R2$, rank_OFF(R3), element_SIZE[R0$]

  # store new_rank in original-list-array
  sw    R2$, rank_OFF(R1$)

  ######### HERE   END   THREADS #########       # this thread done
  join  SPWNR, R1                               #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $


EXIT_WYLLIE:
  jr

##### END of Wyllie ranking #####
```

B.5 Reid-Miller's Parallel Algorithm

```
# List Ranking - Reid-Miller Parallel Random Alg.

### Some Constant OFFsets and SIZEs meaning:
#   ----------------------------------------
#       rank_OFF        - offset of the FIELD of rank, in the structure of an element
#       select_OFF
#       next_OFF        - offset of the FIELD of next address, in the structure of an element
#       prnt_OFF        - ranking-parent for the backward iterations
#       back_OFF        - offset of "back" - indication if the element is the HEAD of the list
#       list_ptr_OFF    - address of the correponding element in the ORIGINAL list_array
#
#       ptr_SIZE        - size of a pointer (i.e., an address)
#       iter_SIZE
#       list_element_SIZE - size of an element in the original List_Array
#       actv_elmnt_SIZE - size of an element in the active array (address size in this alg.)
#       skpd_SIZE
#       addr_SIZE
#
# NOTE: In this alg. we use actv_array only for indexing during selection,
```

```
#       and later work ONLY on the list_array.


##### main loop and book-keeping #####
# C-function:   void parallel_rand_ranking(List_Array *list, int list_size)

                        # registers' mapping:
                        #       SPWNR   spawn-join counter
                        #       R1      current list size (both for fwd & bckwd iterations)
                        #       R2      list-array start (for 1st)
                        #       R3      active-counter
                        #       R4      non-active-counter
                        #       R5      nactv_array statring addr (for this ROUND)
                        #       R6      accumulative-non-active-index (into nactv[]) - till where already used
                        #       R7      address of HEAD in the list_arry (used to force HEAD as selected)
                        #       R8      iteration number
                        #       R9      actv_array start addr (we overwrite the array every iteration)
                        #       R10     1 (for branching comparisons)
                        #       R11     k = list_size/4
                        #       R12     for PS to get index to selected_list - k_unique (# of unique
                        #               elements out of k randomly picked)
                        #       R13     start of selected_list (we overwrite the list every iteration)

        lw      R1, list_size_OFF(SP)                   # check starting size > 1 (i.e.,  <= 1 -> exit)
        ble     R1, R10,        EXIT1

        lw      R2, list_OFF(SP)                        # load pointer to list from STACK
        la      R9, actv(R0)                            # start address of active array
        # initialization for iteration process (to work on nactv[] array)
        li      R8, 0                                   # iter = 0
        la      R5, nactv(R0)                           # init start address of non-active array (curr_nactv)
        li      R6, 0                                   # nactv_indx = 0

        j       FIRST_FWDITR                            # special preparation for first iteration DIRECTLY on list-array

JFWD:
        # loop on forward parallel iterations (pointer jumping)
        j       FWDITR                                  # do one forward parallel iteration
END_FWDITR:                                             # in R1 we have the new current list size

        # iteration oriented information (each round "looks" at different part of the array)
        # calculate and store relevant info for back iterations:
        #               skipped start addr in array, num skipped in this round

        # store nactv_start_addr(R5) & indx (R4) for this iteration (R8)
        swa     R4, bck_num_OFF+iter_info(R0), iter_SIZE[R8]
        swa     R5, bck_start_addr_OFF+iter_info(R0), iter_SIZE[R8]
        addi    R8, R8, 1                               # iter = iter + 1

        add     R6, R6, R4                              # ADD the num of non-actives into the accumulative-non-active-index
        laa     R5, nactv(R0), skpd_SIZE[R6]            # for next round start_address (of non active array)

        bne     R1, R10, JFWD                           # if curr != 1 (i.e. larger than) do "fwditr"

        # backward parallel final ranking
```

```
JBCK:
        # update iteration oriented information (each round "looks" at different part of the array)
        subi    R8, R8, 1                               # iter = iter - 1
        # load the starting address in nactv_array & number of elements for this iteration
        lwa     R5, bck_start_addr_OFF+iter_info(R0), iter_SIZE[R8]
        lwa     R1, bck_num_OFF+iter_info(R0), iter_SIZE[R8]

        j       BCKITR                                  # do ONE backward parallel ranking (all at once)
END_BCKITR:

        subi    R6, R6, R1                              # accumulative_non_selected -= num_bck_in_this_iteration (R1)
        bgt     R6, R0, JBCK                            # if (accumulative-non-selected > 0) do "bckitr"

EXIT1:
        jr

##### END of main loop and book-keeping #####

######## First ITERATION  (special treatment) #######
# special treatment for FIRST iteration: selection and active-array preparation
# C-function:   void  par_first_fwditr(List_Array *orig_list /*IN*/, int list_size /*IN*/,
#                                      Actv_Struct *actv /*OUT*/)

FIRST_FWDITR:
        ### select k elements for first iteration (directly on list_array)
        ### and find address of HEAD

### Prepare for selection of k elements (k=list_size/4)
        divi    R11, R1, 4              # set k = list_size/4 (could use shift right by 2 bits instead)
        li      R12,    0               # counter for position in selected_list

### find HEAD & set all "selected" to ZERO (for marking)
        spawn   SPWNR, 0, R1, 1
        # NOTE: the "selected" field is used also as gate-keepers for the markM - for avoiding duplicities
        # in the k-selection. The alternative is to do a serial sort and duplicity check and "save" this spawn
        #########  HERE   START   THREADS #########
                        # threads registers' mapping:
                        #       R1$     back value
        swa     R0, select_OFF(R2), list_element_SIZE[R0$]    # set selected to ZERO
        lwa     R1$, back_OFF(R2), list_element_SIZE[R0$]     # load back value - indicates if HEAD of list
        bne     R1$, R0, NOT_HEAD                             # check if I'm the HEAD of the list (back == 0)
        laa     R7, 0(R2), list_element_SIZE[R0$]             # R7 = Head's address
NOT_HEAD:
        #########  HERE   END   THREADS #########
        # synchronize all threads
        join    SPWNR, R1                               #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $

### Randomly pick k selected elements out of list_size
        spawn   SPWNR, 0, R11, 3
        #########  HERE   START   THREADS #########
                        # threads registers' mapping:
                        #       R1$     random value /address of element in list_array
                        #       R2$     result from markM
                        #       R3$     index into selected_list
```

```
        li      R3$, 1          # prepare for PS
        # use THREAD 0 to force the HEAD of the list -> always selected
        bne     R0$, R0, OTHER_THREADS1                 # check if thread 0
        move    R1$, R7                                 # address of HEAD
        j       CONT_HEAD1

OTHER_THREADS1:
        R1$ = rand(1,R1-1)                          # randomly number from [1, R1-1]
        # NOTE: HEAD, which is element 0 is ALWAYS selected
        # We check that an element wasn't seleced twice by using "markM" - ONLY one succeeds in marking
        laa     R1$, 0(R2), list_element_SIZE[R1$]      # calculate the element's address
CONT_HEAD1:
        markM   select_OFF(R1$), R2$
        bne     R2$, R0, NOT_SELECTED1                  # if (R2$==0) mark succeeded
                                                        # and selected is SET to 1 in list_array
        ps      R12, R3$                                # find index in selected_list
        swa     R1$, 0(R13), ptr_SIZE[R3$]              # store this element's address in selected_list
        # NOTE: we do NOT store 1 to selected in actv_array, since we only use actv_array for indexing
        #       during selection, and later work ONLY on the list_array

NOT_SELECTED1:
        ######### HERE  END  THREADS #########
        # synchronize all threads
        join    SPWNR, R11                               #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $

### Short-cutting and shrinking sub-lists
        j       SHORT_CUTTING

######## END of First ITERATION #######


######## FORWARD ITERATION #######
# returns NEW_LIST_SIZE of shrinked list in R1
# The input arguments are in registers:
#       beginning of the actv_list for READ (R9), the list size (R1) & beginning of nactv (R5)
# C-function:   int  par_fwditr(Actv_Struct  *curr_actv /*IN/OUT*/, int  curr_list_size /*IN*/,
#                               Nactv_Struct  *curr_nactv /* OUT*/)

FWDITR:
### Prepare for selection of k elements (k=list_size/4)
        divi    R11, R1, 4              # set k = list_size/4 (could use shift right by 2 bits instead)
        li      R12,    0               # counter for position in selected_list

        spawn   SPWNR, 0, R1, 1         # set all "selected" to ZERO (for marking)
        # NOTE: the "selected" field is used also as gate-keepers for the markM - for avoiding duplicities
        # in the k-selection. The alternative is to do a serial sort and duplicity check and "save" this spawn
        ######### HERE  START  THREADS #########
                # threads registers' mapping:
                #       R1$     address of element in list_array
        lwa     R1$, list_ptr_OFF(R9), actv_element_SIZE[R0$] # load address of element from actv_array
        sw      R0, select_OFF(R1$)                           # set selected to ZERO
        ######### HERE  END  THREADS #########
        # synchronize all threads
        join    SPWNR, R1                               #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $
```

```
### Randomly pick k selected elements out of list_size
        spawn   SPWNR, 0, R11, 3
        #########  HERE   START   THREADS #########
                        # threads registers' mapping:
                        #       R1$     random value /address of element in list_array
                        #       R2$     result from markM
                        #       R3$     index into selected_list

        li      R3$, 1          # prepare for PS
        # use THREAD 0 to force the HEAD of the list -> always selected
        bne     R0$, R0, OTHER_THREADS                  # check if thread 0
        move    R1$, R7                                 # address of HEAD
        j       CONT_HEAD

OTHER_THREADS:
        R1$ = rand(1,R1-1)                          # randomly number from [1, R1-1]
        # NOTE: HEAD, which is element 0 is ALWAYS selected
        # We check that an element wasn't seleced twice by using "markM" - ONLY one succeeds in marking
        lwa     R1$, list_ptr_OFF(R9), actv_element_SIZE[R1$] # load the address in list_array of the element picked
CONT_HEAD:
        markM   select_OFF(R1$), R2$
        bne     R2$, R0, NOT_SELECTED                   # if (R2$==0) mark succeeded
                                                        # and selected is SET to 1 in list_array
        ps      R12, R3$                                # find index in selected_list
        swa     R1$, 0(R13), ptr_SIZE[R3$]              # store this element's address in selected_list
        # NOTE: we do NOT store 1 to selected in actv_array, since we only use actv_array for indexing
        #       during selection, and later work ONLY on the list_array

NOT_SELECTED:
        #########  HERE   END   THREADS #########
        # synchronize all threads
        join    SPWNR, R11                               #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $

SHORT_CUTTING:
### Short-cutting and shrinking sub-lists
        # initialize counters for short-cutting
        li      R3,     0                       # actv_counter = 0
        li      R4,     0                       # nactv_counter = 0

        #### includes: skipping, shrinking

        # asynch pardo - fork_into_threads  $ = [0..k_unique] (R12)
        spawn   SPWNR, 0, R12, 8        # where SPWNR is the spawning reg., 0 is the offset & k_unique (R12) is the size
        #########  HERE   START   THREADS #########
                        # threads registers' mapping:
                        #       R1$     address of $'s element (which is prnt for "skipped")
                        #       R2$     index to active OR non-active array
                        #       R3$     element's rank
                        #       R4$     next-element's LIST address
                        #       R5$     next-element's LIST address (to avoid unecessary dependences)
                        #       R6$     next-element's selected / element's index into coin-tossing & value of
                        #                coin tossing and NEW selected BYTE
                        #       R7$     next-element's rank / next-index into coin-tossing
                        #       R8$     next-element's LIST address (to aviod unecessary dependences) / misc
```

```
        # load selected elements to start short-cutting from them
        lwa     R1$, 0(R13), ptr_SIZE[R0$]            # load $'s addr in list_array (from selected_list)
        # must wait until have R1$ (address) loaded
        lw      R3$, rank_OFF(R1$)                    # load curr's rank - rank(list($))

        # advance to next element (till run into a selected element)
        lw      R4$, next_OFF(R1$)                    # pre-load 1st's address
        # check that it's not the end of the list
        beq     R4$, R0, CUT                          # if (next == NULL) -> break
ADVNC:
        # skipping the following non-selected nodes (loop unrolled)

        #### 1st (unrolled) node to skip, address loaded at END of loop
        # check if element selected (if so -> done)
        lba     R6$, select_OFF(R4$)                  # load next's selected BYTE to memory
        beq     R6$, R10, CUT                         # if (next(list[i]).selected)  done /* NEXT is selected */

        # continue since next is NOT selected
        lw      R7$, rank_OFF(R4$)                    # load next's rank

        # store "ranking diff" from "ranking parent"
        sw      R3$, rank_OFF(R4$)                    # store rank(i)
        sw      R1$, prnt_OFF(R4$)                    # store "ranking parent" addr

        ### since SKIPPED -> shrink into curr_nactv (not-actv array)
        # prefix-sums (unordered) on NOT "selected" flags to keep array of "skipped"
        li      R2$,  1                              # init for PS
        ps      R4, R2$                              # each thread does this for each skipped to find new indx for it
        # now need to "copy" relevant NON selected (i.e., skipped) elements into NEW array
        swa     R4$, 0(R5), skpd_SIZE[R2$]           # store into nactv[nactv_indx] place in array "SKIPPED abs addr."

        # store to nactv[nactv_indx] the available info in an element
        swa     R3$, rank_OFF(R5), skpd_SIZE[R2$]
        swa     R1$, prnt_addr_OFF(R5), skpd_SIZE[R2$] # store into nactv also prnt_addr - for shorter load

        # calculate new $'s rank & next
        add     R3$, R3$, R7$                         # add ranks:    rank(list(i)) + rank(next(list(i)))

        #### 2nd (unrolled) node to skip               # using a different register to aviod unecessary dependences
        lw      R5$, next_OFF(R4$)                    # load next address - next(curr)
        move    R4$, R5$                              # for correctness of the "CUT" final storing
        beq     R5$, R0, CUT                          # if (next == NULL) -> break

        # check if element selected (if so -> done)
        lb      R6$, select_OFF(R5$)                  # load next's selected BYTE to memory
        beq     R6$, R10, CUT                         # if (next(list[i]).selected)  done /* NEXT is selected */

        # continue since next is NOT selected
        lw      R7$,    rank_OFF(R5$)                 # load next's rank - rank(next(list(i)))

        # store "ranking diff" from "ranking parent"
        sw      R3$, rank_OFF(R5$)                    # store rank(i)
        sw      R1$, prnt_OFF(R5$)                    # store "ranking parent" addr

        ### since SKIPPED -> shrink into curr_nactv (not-actv array)
```

```
        # prefix-sums (unordered) on NOT "selected" flags to keep array of "skipped"
        li      R2$,   1                              # init for PS
        ps      R4, R2$                               # each thread does this for each skipped to find new indx for it
        # now need to "copy" relevant NON selected (i.e., skipped) elements into NEW array
        swa     R5$, 0(R5), skpd_SIZE[R2$]            # store into nactv[nactv_indx] place in array "SKIPPED abs addr."

        # store to nactv[nactv_indx] the available info in an element
        swa     R3$, rank_OFF(R5), skpd_SIZE[R2$]
        swa     R1$, prnt_addr_OFF(R5), skpd_SIZE[R2$] # store into nactv also prnt_addr - for shorter load

        # calculate new $'s rank & next
        add     R3$,    R3$,    R7$                    # add ranks:    rank(list(i)) + rank(next(list(i)))

        #### 3rd (unrolled) node to skip               # using a different register to aviod unecessary dependences
        lw      R8$, next_OFF(R5$)                     # load next address - next(curr)
        move    R4$, R8$                               # for correctness of the "CUT" final storing
        beq     R8$, R0, CUT                           # if (next == NULL) -> break

        # check if element selected (if so -> done)
        lb      R6$, select_OFF(R8$)                   # load next's selected BYTE to memory
        beq     R6$, R10, CUT                          # if (next(list[i]).selected)  done /* NEXT is selected */

        # continue since next is NOT selected
        lw      R7$, rank_OFF(R8$)                     # load next's rank - rank(next(list(i)))

        # store "ranking diff" from "ranking parent"
        sw      R3$, rank_OFF(R8$)                     # store rank(i)
        sw      R1$, prnt_OFF(R8$)                     # store "ranking parent" addr

        ### since SKIPPED -> shrink into curr_nactv (not-actv array)
        # prefix-sums (unordered) on NOT "selected" flags to keep array of "skipped"
        li      R2$,   1                              # init for PS
        ps      R4, R2$                               # each thread does this for each skipped to find new indx for it
        # now need to "copy" relevant NON selected (i.e., skipped) elements into NEW array
        swa     R8$, 0(R5), skpd_SIZE[R2$]            # store into nactv[nactv_indx] place in array "SKIPPED abs addr."

        # store to nactv[nactv_indx] the available info in an element
        swa     R3$, rank_OFF(R5), skpd_SIZE[R2$]
        swa     R1$, prnt_addr_OFF(R5), skpd_SIZE[R2$] # store into nactv also prnt_addr - for shorter load

        # calculate new $'s rank & next
        add     R3$, R3$, R7$                          # add ranks:    rank(list(i)) + rank(next(list(i)))

        # Note:  here further copies of the loop can be unrolled if desired

        lw      R4$, next_OFF(R8$)                     # load next address - next(curr)
        bne     R4$, R0, ADVNC                         # if (next == NULL) -> break (otherwise continue while-loop)
        # end of unrolled loop for skipping

CUT:
        # store updated info in original-list
        sw      R3$, rank_OFF(R1$)                     # store new rank: rank(list(i)) = rank(list(i)) + rank(next(list(i)))
        sw      R4$, next_OFF(R1$)                     # store new next: next(list(i)) = next(next(list(i)))...

        ### since SELECTED copy-shrink into next active array
```

```
        # prefix-sums (unordered) on "active" (e.g., previously selected) to shrink into actv array
        li      R2$,   1                              # init for PS
        ps      R3,     R2$                           # each thread does this for each active find new indx

        # now need to "copy"/"shrink" relevant elements into NEW array
        swa     R1$, list_ptr_OFF(R9), actv_elmnt_SIZE[R2$] # store into actv[R9+R2$*SIZE] my addr. in list_array
        # NOTE: no need to store all element info in actv array since it is used only for
        #       figuring the addresses during selection

        # finished the skipping & shrinking
        #########  HERE   END   THREADS #########
        # synchronize all threads
        join    SPWNR, R12                             #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $

        # only now (the last thread to arrive is the only one alive) will we end "fwditr"
        move    R1,     R3                     # get final actv_counter (new list size) and put it in R1
        j       END_FWDITR

######## END of FORWARD ITERATION #######



######## BACKWARD ITERATION (final ranking) #######
# The input arguments are in registers:
#       starting address in skipped array (R5) & the number of skipped
#       elements in this iteration (R1)
# C-function:   void  par_bckitr(Nactv_Struct  *curr_nactv /*IN*/, int  size /*IN*/)

BCKITR:

        # asynch pardo - fork_into_threads  $ = [1..list_size] (R1)
        spawn   SPWNR, 0, R1, 4         # where SPWNR is the spawning reg., 0 is the offset & list_size (R1) is the size
        #########  HERE   START   THREADS #########
                        # threads registers' mapping:
                        #       R1$     address of $'s element
                        #       R2$     misc, temporary storage, etc.
                        #       R3$     element's rank
                        #       R4$     element's ranking_prnt addr.

        lwa     R1$, 0(R5), skpd_SIZE[R0$]             # load the element's addr. from nactv array
        lwa     R3$, rank_OFF(R5), skpd_SIZE[R0$]      # load curr's rank - rank(list($))
        lwa     R4$, prnt_addr_OFF(R5), skpd_SIZE[R0$]  # load curr's rank_prnt addr. (from the skipped array)

        lw      R2$, rank_OFF(R4$)                     # load rank_parent's rank - rank(rank_prnt)
        # calculate & store:     "ranking parent" - "ranking diff"
        sub     R3$,    R2$,    R3$
        sw      R3$, rank_OFF(R1$)

DONE2:                                                 # this thread done
        #########  HERE   END   THREADS #########
        # synchronize all threads before ending (and starting next round)
        join    SPWNR, R1                              #if ( sumi(SPWNR, 1) != num_thrds-1) KILL $

        # only now (the last thread to arrive is the only one alive) will we end "bckitr"
        j       END_BCKITR
```

```
######## END of BACKWARD ITERATION #######
```

C Compilation Feasibility

This section summarizes why the translation from the high level code (C and extended-C) to the optimized assembly code is feasible with known compiler techniques. The line numbers given as examples are all in the assembly code table of the "no-cut" algorithm, unless stated otherwise.

C.1 Compilation Techniques

1. Functions in the C code are in-lined in the assembly code. Note that the absolute jumps in assembly code are present only for assembly code readability, and are not counted in either the work or time analysis.

   For example, see line 15 for such a jump, and lines 129-146 for an in-lined function.

2. Register usage is optimized for global, local and re-used variables. Special attention was given to minimizing loads and stores by not overwriting registers which contain variables that are frequently used.

   For example, global register $R1$ holds the current list size throughout the code. It is once loaded with the input value in line 1, then updated at the end of every forward iteration in line 149. The same is done with local thread registers. For example, thread register $R3\$$ holds the rank of the active element throughout the forward iteration. It is read once in line 73, it is updated and stored as the ranking-difference in lines 80, 85, 87, ..., 115 and then the resulting rank at the end of the short-cutting is stored in lines 119 and 124.

3. Loop unrolling is used in the serial algorithm, when the various iterations of the for loop are independent (in the final ranking part). It is also used in the parallel coin-tossing algorithm (no-cut), where an unknown number of elements are pointer-jumped.

   For example, there are instances of the unrolled short-cutting loop in lines 77-87, 88-101 and 102-115. The decision to unroll this loop 3 times was arbitrary, but according to our analysis the additional copies would not improve execution due to the control dependencies, but will increase the code size.

   In the assembly code table of the serial algorithm the instances of the unrolled loop are in lines 41-43, 44-46, etc.

4. Code independence can be increased by the compiler by using separate registers for every instance of the unrolled loops.

   For example, in line 88 local thread register $R5\$$ is used for loading the next element's address, instead of waiting for register $R3\$$. The value is then updated also in register $R4\$$ just in case the conditional branch in line 90 is taken.

5. In the serial portions of the code several consecutive writes to the same address (without a read in between them) can be removed by the compiler, leaving only the last write.

6. Instructions which are independent can be rearranged by the compiler to allow optimal execution.

7. A compiler could insert prefetch instructions to minimize stall penalties due to load-misses. A prefetch instruction would initiate the load ahead of time, wherever the address can be resolved beforehand. Prefetch instructions were not explicitly included in the assembly code (both serial and parallel), however their existence is assumed for the performance analysis.

   For example, in line 88 the load of address of the next element to be short-cut is prefetched, and started at time 9 in the block, although it can commit only at time 13 (dependent on the result of the conditional branch in line 78).

8. However, using our conservative comparison approach for further details) the serial assembly code was generated in two steps. First the *gcc* C compiler (on a *SPARC* workstation) with the optimization "-O2" produced optimized code. Then further optimization by hand was added, while converting the assembly code to the *MIPS* assembly language.

   With '-O', the *gcc* compiler tries to reduce code size and execution time. '-O2' optimizes even more. Nearly all supported optimizations are performed. As compared to '-O', this option increases compilation time and improves the performance of the generated code (from the "GCC manual pages", last changed: 1993/10/13).

   Note that since we advocate the parallel code it was not necessary to argue that a compiler could derive this optimized serial code (since we are looking to establish a lower bound on the performance of the serial code).

D Supporting Simulations

This section describes and explains briefly the simulations we perfformed to support the execution analysis of the more complex parallel algorithms. The details can be found in the submitted paper.

The source code for the supporting simulations described here can be reached through the "Supporting Simulations" link on our web page.

D.1 Parallel Randomized Coin-Tossing Algorithm (no-cut)

Two issues arise when analyzing the randomized coin-tossing algorithm: what is the behaviour of the forward iterations (such as their convergence) and finding a bound for the length of the longest chain of short-cut elements (before encountering another selected element).

Convergence of Iterations

Let N be the initial input size, and let M_i be the expected number of active elements in the i-th iteration (i.e., the input size for the i-th iteration).

The sum of this series, necessary for the summing of the work in the algorithm will be:

$$\Sigma M_i \leq N + \frac{1}{4}N + \cdots + M_i \leq (\frac{1}{4})^i N + \cdots \leq \frac{1}{(1-\frac{1}{4})}N = \frac{4}{3}N \leq 1.34N$$

The number of iterations necessary for the algorithm to finish, e.g., that the list size be 1 in the last iteration, is:

$$(\frac{1}{4})^{\#iter} N = 1$$

$$\Longrightarrow \#iter = log_4 N = 0.5 log_2 N$$

To support the full analysis presented in the paper we simulated the iterations of the algorithm for various sizes of inputs (N values). We repeated the simulation for each input size (N) 20 times, and looked at $E(\Sigma M_i)$ and $E(\#iter)$ that resulted. According to the simulations:

$$\Sigma M_i \leq 1.33N$$

and

$$\#iter \leq 0.46 log_2 N$$

both fully supporting our analysis.

Simulations to Support the Bound for the Length of the Longest Chain

In order to analyze the execution time for the *no-cut* algorithm we need to find a realistic bound on the length of the longest consecutive number of elements that will be short-cut due to our coin-tossing selection mechanism (also denoted as "longest chain" for short).

We would like to show that the longest chain is on the order of magnitude of $log_2(N)$ with very high probability.

We ran (randomized) simulations for over 200 different input-size (N) values, each of which was repeated for 20 times. The input-size (N) values ranged from 10^2 to 10^7.

The coin-tossing was done using the *addrans* function from the *UNIX* mathematical library. The function provides uniform random additive numbers and is based on Knuth's discussion in Knuth D.E. *The Art of Computer Programming, Vol. 2 - Seminumerical Algorithms*, 2nd Edition. Addison-Wesley. 1981.

A brief description of the simulations:

```
Repeat for Various N-values
{ Some N-value is randomly chosen
  Repeat 20 times:
  { Toss a coin N times
    Find the longest chain that was formed [denoted Maxlen]
  }
  Calculate E(Maxlen)
  Calculate E(Maxlen)/log2(N) [denoted R]
}
Find MAX(R) over all the different N-values
```

The simulations resulted in the following:

- We ran over 200 different N-values in the range: $10^2 \leq N \leq 10^7$
- $Max(R) < 1.23$
- Only 6 simulations resulted in $R \geq 1.20$
- Practical conclusion:
  $E(Maxlen) = 1.2 * log_2 N$

D.2 Reid-Miller's Parallel Algorithm

Two issues arise when analyzing Reid-Miller's randomized algorithm: what is the behaviour of the forward iterations (such as their convergence) and finding a bound for the length of the longest chain (i.e., sub-list) of short-cut elements (before encountering another selected element).

Convergence of Iterations

Let N be the initial input size, and let M_i be the expected number of active elements in the i-th iteration (i.e., the input size for the i-th iteration).

The sum of this series, necessary for the summing of the work in the algorithm will be:

$$\Sigma M_i \leq N + 0.221N + \cdots + M_i \leq (0.221)^i N + \cdots \leq \frac{1}{(1-0.221}N = \frac{1}{0.779}N \leq 1.29N$$

The number of iterations necessary for the algorithm to finish, e.g., that the list size be 1 in the last iteration (of course cascading is in order, but it is separated for comparison purposes), is:

$$(0.221)^{\#iter} N = 1$$

$$\Longrightarrow \#iter = log_{\frac{1}{0.221}} N \simeq 0.46 log_2 N$$

To support the full analysis presented in the paper we simulated the iterations of the algorithm for various sizes of inputs (N values). We repeated the simulation for each input size (N) 20 times, and looked at $E(\Sigma M_i)$ and $E(\#iter)$ that resulted. According to the simulations:

$$\Sigma M_i \leq 1.28N$$

and

$$\#iter \leq 0.43 log_2 N$$

both fully supporting our analysis.

Simulations to Support the Bound for the Length of the Longest Chain

In order to analyze the execution time for the algorithm we need to find a realistic bound on the length of the longest consecutive number of elements that will be short-cut due to the selection mechanism used (also denoted as "longest chain" for short).

We would like to show that the longest chain is on the order of magnitude of $log_2(N)$ with very high probability.

We ran (randomized) simulations for over 200 different input-size (N) values, each of which was repeated for 20 times. The input-size (N) values ranged from 10^2 to 10^7.

The random picking of numbers between 1 and N was done using the *addrans* function from the *UNIX* mathematical library. The function provides uniform random additive numbers and is based on Knuth's discussion in Knuth D.E. *The Art of Computer Programming, Vol. 2 - Seminumerical Algorithms*, 2nd Edition. Addison-Wesley. 1981.

A brief description of the simulations:

```
Repeat for Various N-values
{ Some N-value is randomly chosen
```

```
  Repeat 20 times:
  { Pick a value between 1 and N with uniform probability N/4 times
    Remove the values that appear more than once (uniqueness)
    Find the longest chain that was formed [denoted Maxlen]
  }
  Calculate E(Maxlen)
  Calculate E(Maxlen)/log2(N) [denoted R]
}
Find MAX(R) over all the different N-values
```

The simulations resulted in the following:

- We ran over 200 different N-values in the range: $10^2 \leq N \leq 10^7$
- $Max(R) \leq 2.75$
- Less than 5% of the simulations resulted in $R \geq 2.70$
- Practical conclusion:
  $E(Maxlen) \simeq 2.75 * log_2 N$

E Time and Work Analysis for the Algorithms

Following are tables for each of the five algorithms presenting the detailed counts.

The assembly code in the tables is the same as the one in the assembly files (in appendix B), expect it is stripped from the comments.

The tables for the algorithms are presented on separate pages due to their size. The order in which they are presented is:

1. Serial Algorithm
2. Parallel Randomized Coin-Tossing Algorithm (no-cut)
3. Time-Optimized Parallel Coin-Tossing Algorithm (cut-6)
4. Wyllie's Parallel Algorithm
5. Reid-Miller's Parallel Algorithm

Serial List Ranking - Time and Work Analysis

Legend for work and time analysis:

N	- is the input size (size of the list)
COPIES	- is the number of copies of the loop in lines 39-55 that can be unrolled

A full understanding of the work and time counting requires reading the detailed counting assumptions and techniques documentation. Below (in the footnotes) there is a partial explanation.

Line #	Instruction Code	Total Work per block	Time Cycle inside block		Total Time
			1st Iter.	2nd Iter.	Per block
1:	lw R1, list_array_OFF(SP)	11[1]	1[2]		5
2:	lw R2, list_size_OFF(SP)		1[3]		
3:	ble R2, R0, DONE		2		
4:	li R3, 0		1		
5:	li R4, 0		1		
6:	li R7, 0		1		
7:	li R8, 1		1		
8:	ble R2, R8, HEAD		3		
9:	sw R0, rank_OFF(R1)		4		
10:	j DONE		4		
11:	HEAD:				
12:	move R5, R1		4		
13:	multi R7, R2, element_SIZE		4		
14:	add R7, R7, R1		5		

[1] The work for lines 1-14 is 11 since we don't count line 11 (a label), and instructions 9 & 10 are only executed in the less interesting case when the list only has one element and the program terminates.

[2] Each instruction is put in the earliest time cycle (relative to the block) in which it can be executed to completion.

[3] Load from the stack (e.g., relative to the Stack Pointer (SP)) is counted as a single cycle since it is assumed the information was stored immediately before the call, and therefore available in the cache.

15:	HEAD_LOOP:	3*N			(2+4*N) since[4] 6 for 1st iter. & 4 more per iter. (N-1)
16:	bge R5, R7, CONT		1	7	
17:	lw R6, back_OFF(R5)		0-4[5]	4-8	
18:	bne R6, R0, CONT_LOOP		5	9	
19:	move R3, R5	2	6	10	
20:	j CONT		6		
21:	CONT_LOOP:	2*(N-1)			
22:	addi R5, R5, element_SIZE		6	10	
23:	j HEAD_LOOP		6		
24:	CONT:	2			1
25:	lw R6, next_OFF(R3)		1		
26:	move R5, R3		1		
27:	FWD:	7*N			(1+5*N) since 6 for 1st iter. & 5 more per iter. (N-1)
28:	sub R7, R0, R4		0	7	
29:	sw R7, rank_OFF(R5)		1	8	
30:	lw R8, weight_OFF(R5)		1	7	
31:	move R5, R6		2	9	
32:	lw R6, next_OFF(R5)		1-5	6-10	
33:	add R4, R4, R8		2	8	
34:	bne R6, R0, FWD		6	11	

[4]Scheduling of first and second iterations suffices for evaluating total time per block.

[5]We charge a total of 5 time cycles for each indirect memory access (we prefetch to start the 5-cycle count early, for earliest completion).
The load in line 17 starts at time cycle 0 relative to the block, e.g., the last time cycle within the previous block. Overlap of blocks is allowed, while taking dependencies into consideration.

35:	sub R7, R0, R4	4	1		3
36:	sw R7, rank_OFF(R5)		2		
37:	move R5, R1		3		
38:	move R6, R2		1		
39:	FINAL1:	$1*(\lfloor \frac{N}{COPIES} \rfloor)$			$4+4*(\lfloor \frac{N}{COPIES} \rfloor)$ since 8 for 1st iter. & 4 per rest of iters.
40:	blt R6, COPIES, FINAL2		1	3	
41:	lw R8, rank_OFF(R5)	$3*COPIES*(\lfloor \frac{N}{COPIES} \rfloor)$	1-5	5-9	
42:	add R8, R8, R4		6	10	
43:	sw R8, rank_OFF(R5)		7	11	
44:	lw R9, element_SIZE+rank_OFF(R5)		1-5	5-9	
45:	add R9, R9, R4		6	10	
46:	sw R9, element_SIZE+rank_OFF(R5)		7	11	
47:	lw R10, 2*element_SIZE+rank_OFF(R5)		1-5	5-9	
48:	add R10, R10, R4		6	10	
49:	sw R10, 2*element_SIZE+rank_OFF(R5)		7	11	
50:	.		.	.	
51:	.		.	.	
52:	.		.	.	
53:	addi R5, R5, COPIES*element_SIZE	$3*(\lfloor \frac{N}{COPIES} \rfloor)$	8	12	
54:	subi R6, R6, COPIES		2	4	
55:	j FINAL1		2	4	

56:	FINAL2:	4			5
57:	blez R6, DONE		0		
58:	multi R7, R6, element_SIZE		1		
59:	add R7, R5, R7		2		
60:	lw R8, rank_OFF(R5)		1-5		
61:	FINAL:	*5*(Nmod COPIES)*			*4*(Nmod COPIES)* since 4 for 1st iter. & 4 more per iter.
62:	add R8, R8, R4		1	5	
63:	sw R8, rank_OFF(R5)		2	6	
64:	addi R5, R5, element_SIZE		3	7	
65:	lw R8, rank_OFF(R5)		0-4	4-8	
66:	blt R5, R7, FINAL		4	8	
67:	DONE:	1			1
68:	jr		1		

Note that the number of copies of the loop that can be unrolled is:

$$COPIES = \left\lfloor \frac{R}{R_i} \right\rfloor = \frac{R}{1} = R$$

Where: R - is the total number of registers available
R_i - is the number of additional registers necessary for each copy of the unrolled loop (in this case 1)

Total Work:

$$11+3*N+2+2*(N-1)+2+7*N+4+(1+3*COPIES+3)*\left\lfloor \tfrac{N}{COPIES}\right\rfloor+4+5*(N \bmod COPIES)+1$$
$$=12*N+22+(4+3*COPIES)*\left\lfloor \tfrac{N}{COPIES}\right\rfloor+5*(N \bmod COPIES)$$
$$=12N+22+(4+3R)*\left\lfloor \tfrac{N}{R}\right\rfloor+5(N \bmod R)\leq 12N+22+4\tfrac{N}{R}+3N+5(N \bmod R)$$
$$=15N+4\tfrac{N}{R}+5(N \bmod R)+22$$

Total Time:

$$5+2+4*N+1+1+5*N+3+4+4*\left\lfloor \tfrac{N}{COPIES}\right\rfloor+5+4*(N \bmod COPIES)+1$$
$$=9N+4*\left\lfloor \tfrac{N}{R}\right\rfloor+4*(N \bmod R)+22$$

Upper Bound on Total Execution Time resulting from the formula (W/P + T):

$$\frac{(15N+4\frac{N}{R}+5(NmodR)+22)}{P}+(9N+4*\left\lfloor \tfrac{N}{R}\right\rfloor+4*(N \bmod R)+22)$$

Parallel Randomized Coin-Tossing List Ranking Time and Work Analysis

The time and work analysis of the wok optimized algorithm, also referred to as "no-cut", is presented.

Legend for work and time analysis:

N - is the input size (size of the list)
Iter - is the number of iterations necessary, e.g. $\log_{(4)}(N) = 0.5*\log_2 N$
M_i - is the number of active elements in the i-th iteration where: $M_0 = N$, $M_1 <= (1/4)*N$, ...

$$\sum M_i \leq N + (\tfrac{1}{4})*N + \ldots + (\tfrac{1}{4})^i * N + \ldots \leq \frac{1}{(1-\frac{1}{4})} * N = (\tfrac{4}{3})*N \leq 1.34*N$$

the sum of the series is:
M_i' - is the number of active elements after the i-th iteration, e.g., M_{i+1}, e.g., $1/4*M_i$

A full understanding of the work and time counting requires reading the detailed counting assumptions and techniques documentation. Below (in the footnotes) there is a partial explanation.

Line #	Instruction Code	Total Work per block	Time Cycle inside block		Total Time per block
			1st Iter.	2nd Iter.	
1:	lw R1, list_size_OFF(SP)	5[1]	1		2
2:	ble R1, R10, EXIT1		2		
3:	lw R2, list_OFF(SP)		2[2]		
4:	la R9, actv(R0)		2		
5:	li R7, 0		2		
6:	j FIRST_FWDITR		x		
7:	END_FIRST_FWDITR:	4			1
8:	move R7, R1		1		
9:	li R8, 0		1		
10:	la R5, nactv(R0)		1		
11:	li R6, 0		1		
12:	JFWD:	2*Iter			1*Iter
13:	move R2, R9		0		
14:	laa R9, actv(R0), actv_element_SIZE[R7]		1		
15:	j FWDITR		x		
16:	END_FWDITR:	7*Iter			1*Iter
17:	swa R4, bck_num_OFF+iter_info(R0), iter_SIZE[R8]		0		

[1]The jump instructions in lines # 6, 15, 28, 65, 150 and 161 are not counted, as work or time, since they can be inlined, and are in the code only for readability purposes.

[2]Speculation (of the branch in line #2) with very high probability, since in most cases the list-size will be larger than 1.

18:	swa R5, bck_start_addr_OFF+iter_info(R0), iter_SIZE[R8]		0		
19:	addi R8, R8, 1		1		
20:	add R7, R7, R3		1		
21:	add R6, R6, R4		0		
22:	laa R5, nactv(R0), skpd_SIZE[R6]		1		
23:	bne R1, R10, JFWD		1		
24:	JBCK:	3*Iter			6+3*Iter since 7 for 1st iter. & 3 more per iter. & 2 more for last iter
25:	subi R8, R8, 1		1	t+3	
26:	lwa R5, bck_start_addr_OFF+iter_info(R0), iter_SIZE[R8]		1-5	(t-1) - (t+3)	
27:	lwa R1, bck_num_OFF+iter_info(R0), iter_SIZE[R8]		1-5	(t-1) - (t+3)	
28:	j BCKITR		x	x	
29:	END_BCKITR:	2*Iter			
30:	subi R6, R6, R1		t+1[3]		
31:	bgt R6, R0, JBCK		t+2		
32:	EXIT1:	1			1
33:	jr		1		
34:	FIRST_FWDITR:	2			17
35:	li R3, 0		1		
36:	spawn SPWNR, 0, R1, 8		1		

[3] "t" is the time-cycle in which the first backward iteration (BCKITR, line #151) is finished, and the program flow continues at line #30. The timing of the following backward iterations can be somewhat overlapped with the current iteration.

37:	laa R1$, 0(R2), list_element_SIZE[R0$]	11*N	2	
38:	lwa R6$, back_OFF(R2), list_element_SIZE[R0$]		1-5	
39:	lwa R5$, next_OFF(R2), list_element_SIZE[R0$]		1-5	
40:	li R2$, 1		2	
41:	lw R4$, rank_OFF(R1$)		5[4]	
42:	laa R3$, 0(R9), actv_elmnt_SIZE[R0$]		2	
43:	swa R1$, 0(R9), actv_elmnt_SIZE[R0$]		3	
44:	sw R4$, rank_OFF(R3$)		6	
45:	sw R5$, next_OFF(R3$)		6	
46:	sw R6$, back_OFF(R3$)		6	
47:	beq R5$, R0, SELECT1		6	
48:	subi R8$, R5$, LOCAL_LIST	3*(N-1)	6[5]	
49:	divi R8$, R8$, list_element_SIZE		7	
50:	lb R8$, COIN(R8$)		8-12	
51:	SELECT1:	2*N		
52:	lb R7$, COIN(R0$)		1-5	
53:	beq R6$, R0, END_SLCTD1		7	
54:	beq R5$, R0, END_SLCTD2	1*(N-1)	8	
55:	andi R7$, R7$, 0x01	2*(N-2)	9	
56:	bne R7$, R10, END_SLCTD2		10	

[4] Same memory accessed as in line #38 (spatial locality).

[5] Speculation (of the branch in line #47) with very high probability, since only one element is the last element in the list.

57:	andi R8$, R8$, 0x01	$2*\frac{(N-2)}{2}$	13		
58:	beq R8$, R0, END_SLCTD1		14		
59:	END_SLCTD2:	$1*\frac{3}{4}N$			
60:	li R2$, 0		15		
61:	END_SLCTD1:	3*N			
62:	sb R2$, select_OFF(R1$)		16		
63:	sb R2$, select_OFF(R3$)		16		
64:	join SPWNR, R1		17		
65:	j END_FIRST_FWDITR	0	x		
66:	FWDITR:	3*Iter			6*Iter (before working on skipped elements)
67:	li R3, 0		0		
68:	li R4, 0		0		
69:	spawn SPWNR, 0, R1, 10		1		
70:	lwa R1$, 0(R2), actv_elmnt_SIZE[R0$]	$3*M_i$ per Iter	0-4		
71:	lba R2$, select_OFF(R2), actv_elmnt_SIZE[R0$]		0-4		
72:	bne R2$, R10, SKPD		5		
73:	lwa R3$, rank_OFF(R2), actv_elmnt_SIZE[R0$]	$3*\frac{M_i}{4}$ per Iter	6		
74:	lwa R9$, back_OFF(R2), actv_elmnt_SIZE[R0$]		6		
75:	lw R4$, next_OFF(R2), actv_element_SIZE[R0$]		6		

76:	ADVNC:	$14*(\frac{3M_i}{4})$ per Iter since 14 per skipped element			3+7*skpd per Iter since 10 for 1st skipped element per iter. & 7 more per additional skipped element per iter.
77:	lba R6$, select_OFF(R4$)		7-11		
78:	beq R6$, R10, CUT		12		
79:	lw R7$, rank_OFF(R4$)		13		
80:	sw R3$, rank_OFF(R4$)		14		
81:	sw R1$, prnt_OFF(R4$)		13		
82:	li R10$, 1		13		
83:	ps R4, R10$		14		
84:	swa R4$, 0(R5), skpd_SIZE[R10$]		15		
85:	swa R3$, rank_OFF(R5), skpd_SIZE[R10$]		15		
86:	swa R1$, prnt_addr_OFF(R5), skpd_SIZE[R10$]		15		
87:	add R3$, R3$, R7$		16		
88:	lw R5$, next_OFF(R4$)		9-13		
89:	move R4$, R5$		16		
90:	beq R5$, R0, CUT		14		
91:	lb R6$, select_OFF(R5$)		14-18		
92:	beq R6$, R10, CUT		19		
93:	lw R7$, rank_OFF(R5$)		20		
94:	sw R3$, rank_OFF(R5$)		21		
95:	sw R1$, prnt_OFF(R5$)		20		
96:	li R10$, 1		20		
97:	ps R4, R10$		21		
98:	swa R5$, 0(R5), skpd_SIZE[R10$]		22		

99:	swa R3$, rank_OFF(R5), skpd_SIZE[R10$]		22		
100:	swa R1$, prnt_addr_OFF(R5), skpd_SIZE[R10$]		22		
101:	add R3$, R3$, R7$		23		
102:	lw R8$, next_OFF(R5$)		16-20		
103:	move R4$, R8$		21		
104:	beq R8$, R0, CUT		21		
105:	lb R6$, select_OFF(R8$)		21-25		
106:	beq R6$, R10, CUT		26		
107:	lw R7$, rank_OFF(R8$)		27		
108:	sw R3$, rank_OFF(R8$)		28		
109:	sw R1$, prnt_OFF(R8$)		27		
110:	li R10$, 1		27		
111:	ps R4, R10$		28		
112:	swa R8$, 0(R5), skpd_SIZE[R10$]		29		
113:	swa R3$, rank_OFF(R5), skpd_SIZE[R10$]		29		
114:	swa R1$, prnt_addr_OFF(R5), skpd_SIZE[R10$]		29		
115:	add R3$, R3$, R7$		30		
116:	lw R4$, next_OFF(R8$)		27		
117:	bne R4$, R0, ADVNC		28		

118:	CUT:	$9*\frac{M_i}{4}$			10*Iter
119:	sw R3$, rank_OFF(R1$)	per Iter	t+1[6]		(after
120:	sw R4$, next_OFF(R1$)		t+1		working
121:	li R10$, 1		t		on
122:	ps R3, R10$		t+1		skipped
123:	swa R1$, 0(R9), actv_elmnt_SIZE[R10$]		t+2		elements)
124:	swa R3$, rank_OFF(R9), actv_elmnt_SIZE[R10$]		t+2		
125:	swa R4$, next_OFF(R9), actv_elmnt_SIZE[R10$]		t+2		
126:	swa R9$, back_OFF(R9), actv_elmnt_SIZE[R10$]		t+2		
127:	beq R4$, R0, SELECT		t+1		
128:	li R8$, 1	$4*(\frac{M_i}{4}-1)$	t+1[7]		
129:	subi R7$, R4$, LOCAL_LIST	per Iter	t+1		
130:	divi R7$, R7$, list_element_SIZE		t+2		
131:	lb R7$, COIN(R7$)		t+3		
132:	SELECT:	$4*\frac{M_i}{4}$			
133:	subi R6$, R1$, LOCAL_LIST	per Iter	t+1		
134:	divi R6$, R6$, list_element_SIZE		t+2		

[6] "t" is determined according to the number of nodes that are skipped.

[7] Speculation (of the branch in line #127) with very high probability, since only one element is the last element in the list.

135:	lb R6$, COIN(R6$)		t+3		
136:	beq R9$, R0, END_SLCTD		t+2		
137:	beq R4$, R0, END_NOT_SLCTD	$1*(\frac{M_i}{4}-1)$ per Iter	t+3		
138:	andi R6$, R6$, 0x01	$2*(\frac{M_i}{4}-2)$ per Iter	t+4		
139:	bne R6$, R10, END_NOT_SLCTD		t+5		
140:	andi R7$, R7$, 0x01	$2*\frac{1}{2}(\frac{M_i}{4}-2)$ per Iter	t+5		
141:	beq R7$, R0, END_SLCTD		t+6		
142:	END_NOT_SLCTD:	$1*\frac{3}{4}*(\frac{M_i}{4})$ per Iter			
143:	li R8$, 0		t+7		
144:	END_SLCTD:	$2*\frac{M_i}{4}$ per Iter			
145:	sb R8$, select_OFF(R1$)		t+8		
146:	sba R8$, select_OFF(R9), actv_elmnt_SIZE[R10$]		t+8		
147:	SKPD:	$1*M_i$ per Iter			
148:	join SPWNR, R1		t+9		
149:	move R1, R3	1*Iter	t+10		
150:	j END_FWDITR		x		
151:	BCKITR:	1*Iter			13*Iter
152:	spawn SPWNR, 0, R1, 4		1		
153:	lwa R1$, 0(R5), skpd_SIZE[R0$]	7*N	1-5		
154:	lwa R3$, rank_OFF(R5), skpd_SIZE[R0$]		1-5		
155:	lwa R4$, prnt_addr_OFF(R5), skpd_SIZE[R0$]		1-5		

156:	lw R2$, rank_OFF(R4$)		6-10		
157:	sub R3$, R2$, R3$		11		
158:	sw R3$, rank_OFF(R1$)		12		
159:	DONE2:				
160:	join SPWNR, R1		13		
161:	j END_BCKITR	0	x		

Total Work:

$$
\begin{aligned}
&(5+4+1+2-3-1-4-\tfrac{4}{2})+(2+7+3+2+3-4-1-4-2+1+1)*Iter+\\
&(11+3+2+1+2+\tfrac{2}{2}+\tfrac{3}{4}+3+7)*N+[3+(3+9+4+4+1+2+\tfrac{2}{2}+\tfrac{3}{4}+2+1)*\tfrac{1}{4}+(14)*\tfrac{3}{4}]*\sum M_i\\
&=30\tfrac{3}{4}*N+(20\tfrac{7}{16})*\sum M_i+8*Iter+2=(30\tfrac{3}{4}+\tfrac{4}{3}*20\tfrac{7}{16})*N+8*\log_4 N+2\\
&=58*N+8*\tfrac{1}{2}*\log_2 N+2\\
&=58*N+4*\log_2 N+2
\end{aligned}
$$

Total Time:

$$
\begin{aligned}
&(2+1+6+1+17)+(1+1+3+6+3+10+13)*Iter+7*Iter*(\#\max\_elements\_skipped)\\
&=7*(\max\_chain)*\log_4 N+37*\log_4 N+27=7*(\max\_chain)*\tfrac{1}{2}*\log_2 N+37*\tfrac{1}{2}*\log_2 N+27\\
&=3.5*\log_2 N*(\max\_chain)+18.5*\log_2 N+27\\
&\le 3.5*1.2(\log_2 N)^2+18.5*\log_2 N+27=4.2(\log_2 N)^2+18.5\log_2 N+27
\end{aligned}
$$

Upper Bound on Total Execution Time resulting from the formula (W/P + T):

$$
\frac{(58*N+4*\log_2 N+2)}{P}+(4.2*(\log_2 N)^2+18.5*\log_2 N+27)
$$

Time Optimized Parallel Randomized Coin-Tossing List Ranking[1] Time and Work Analysis

The time and work analysis of the time optimized algorithm, also referred to as "cut-6", is presented.

Legend for work and time analysis:

N - is the input size (size of the list)
Iter - is the number of iterations necessary,
e.g. $\log_{(256/73)}(N) = 0.552*\log_2 N$
M_i - is the number of active elements in the i-th iteration
where: $M_0 = N$, $M_1 <= (73/256)*N$, ...

$$\sum M_i \leq N + \left(\tfrac{73}{256}\right) * N + \ldots + \left(\tfrac{73}{256}\right)^i * N + \ldots \leq \frac{1}{\left(1 - \frac{73}{256}\right)} * N = \left(\tfrac{256}{183}\right) * N \leq 1.4 * N$$

the sum of the series is:
M_i' - is the number of active elements after the i-th iteration,
e.g., M_{i+1}, e.g., $73/256*M_i$

A full understanding of the work and time counting requires reading the detailed counting assumptions and techniques documentation. Below (in the footnotes) there is a partial explanation.

[1] We also refer to this algorithm as "6-cut", since we stop short-cutting after the 6th non-selected element.

Line #	Instruction Code	Total Work per block	Time Cycle inside block		Total Time
			1st Iter.	2nd Iter.	per block
1:	lw R1, list_size_OFF(SP)	5[2]	1		2
2:	ble R1, R10, EXIT1		2		
3:	lw R2, list_OFF(SP)		2[3]		
4:	la R9, actv(R0)		2		
5:	li R7, 0		2		
6:	j FIRST_FWDITR		x		
7:	END_FIRST_FWDITR:	4			1
8:	move R7, R1		1		
9:	li R8, 0		1		
10:	la R5, nactv(R0)		1		
11:	li R6, 0		1		
12:	JFWD:	2*Iter			1*Iter
13:	move R2, R9		0		
14:	laa R9, actv(R0), actv_element_SIZE[R7]		1		
15:	j FWDITR		x		

[2]The jump instructions in lines # 6, 15, 28, 69, 203 and 214 are not counted, as work or time, since they can be inlined, and are in the code only for readability purposes.

[3]Speculation (of the branch in line #2) with very high probability, since in most cases the list-size will be larger than 1.

16:	END_FWDITR:	7*Iter			1*Iter
17:	swa R4, bck_num_OFF+iter_info(R0), iter_SIZE[R8]		0		
18:	swa R5, bck_start_addr_OFF+iter_info(R0), iter_SIZE[R8]		0		
19:	addi R8, R8, 1		1		
20:	add R7, R7, R3		1		
21:	add R6, R6, R4		0		
22:	laa R5, nactv(R0), skpd_SIZE[R6]		1		
23:	bne R1, R10, JFWD		1		
24:	JBCK:	3*Iter			(6+3*Iter) since 7 for 1st iter. & 3 more per iter. & 2 more for last iter.
25:	subi R8, R8, 1		1	(t+3)	
26:	lwa R5, bck_start_addr_OFF+iter_info(R0), iter_SIZE[R8]		1-5	(t-1) - (t+3)	
27:	lwa R1, bck_num_OFF+iter_info(R0), iter_SIZE[R8]		1-5	(t-1) - (t+3)	
28:	j BCKITR		x	x	
29:	END_BCKITR:	2*Iter			
30:	subi R6, R6, R1		t+1[4]		
31:	bgt R6, R0, JBCK		t+2		
32:	EXIT1:	1			1
33:	jr		1		

[4] "t" is the time-cycle in which the first backward iteration (BCKITR, line #204) is finished, and the program flow continues at line #30. The timing of the following backward iterations can be somewhat overlapped with the current iteration.

34:	FIRST_FWDITR:	2			17
35:	li R3, 0		1		
36:	spawn SPWNR, 0, R1, 8		1		
37:	laa R1$, 0(R2), list_element_SIZE[R0$]	12*N	2		
38:	lwa R6$, back_OFF(R2), list_element_SIZE[R0$]		1-5		
39:	lwa R5$, next_OFF(R2), list_element_SIZE[R0$]		1-5		
40:	li R2$, 1		2		
41:	lw R4$, rank_OFF(R1$)		5[5]		
42:	laa R3$, 0(R9), actv_elmnt_SIZE[R0$]		2		
43:	swa R1$, 0(R9), actv_elmnt_SIZE[R0$]		3		
44:	sw R3$, actv_ptr_OFF(R1$)		3		
45:	sw R4$, rank_OFF(R3$)		6		
46:	sw R5$, next_OFF(R3$)		6		
47:	sw R6$, back_OFF(R3$)		6		
48:	beq R5$, R0, SELECT1		6		
49:	subi R8$, R5$, LOCAL_LIST	3*(N-1)	6[6]		
50:	divi R8$, R8$, list_element_SIZE		7		
51:	lb R8$, COIN(R8$)		8-12		
52:	SELECT1:	2*N			
53:	lb R7$, COIN(R0$)		1-5		

[5] Same memory accessed as in line #38 (spatial locality).

[6] Speculation (of the branch in line #48) with very high probability, since only one element is the last element in the list.

54:	beq R6$, R0, END_SLCTD1		7		
55:	beq R5$, R0, END_SLCTD2	1*(N-1)	8		
56:	andi R7$, R7$, 0x01	2*(N-2)	9		
57:	bne R7$, R10, END_SLCTD2		10		
58:	andi R8$, R8$, 0x01	$2*\frac{(N-2)}{2}$	13		
59:	beq R8$, R0, END_SLCTD1		14		
60:	END_SLCTD2:	$1*\frac{3}{4}N$			
61:	li R2$, 0		15		
62:	END_SLCTD1:	5*N			
63:	sb R2$, select_OFF(R1$)		16		
64:	sb R2$, select_OFF(R3$)		16		
65:	DONE1:				
66:	sb R0, done_OFF(R1$)		16		
67:	sb R0, done_OFF(R3$)		16		
68:	join SPWNR, R1		17		
69:	j END_FIRST_FWDITR	0	x		
70:	FWDITR:	3*Iter			44*Iter
71:	li R3, 0		0		
72:	li R4, 0		0		
73:	spawn SPWNR, 0, R1, 9		1		
74:	lwa R1$, 0(R2), actv_elmnt_SIZE[R0$]	$3*M_i$ per Iter	0-4		
75:	lba R2$, select_OFF(R2), actv_elmnt_SIZE[R0$]		0-4		
76:	bne R2$, R10, SKPD		5		

77:	lwa R3$, rank_OFF(R2), actv_elmnt_SIZE[R0$]	$4*\frac{M_i}{4}$ per Iter	6[7]	
78:	ADVNC:			
79:	lw R4$, next_OFF(R2), actv_element_SIZE[R0$]		6	
80:	lba R6$, select_OFF(R4$)		7-11	
81:	beq R6$, R10, CUT		12	
82:	lw R7$, rank_OFF(R4$)	$10*(\frac{M_i}{4}-\frac{1}{2})$ per Iter	13	
83:	lw R9$, actv_ptr_OFF(R4$)		13	
84:	sw R3$, rank_OFF(R4$)		14	
85:	sw R1$, prnt_OFF(R4$)		13	
86:	add R3$, R3$, R7$		15	
87:	sb R2$, done_OFF(R4$)		13	
88:	sb R2$, done_OFF(R9$)		14	
89:	lw R5$, next_OFF(R4$)		11[8]	
90:	move R4$, R5$		15	
91:	beq R5$, R0, CUT		13	
92:	lb R6$, select_OFF(R5$)	$2*(\frac{M_i}{4}-1)$ per Iter	12-16	
93:	beq R6$, R10, CUT		17	
94:	lw R7$, rank_OFF(R5$)	$10*(\frac{3M_i}{16})$ per Iter	18	
95:	lw R9$, actv_ptr_OFF(R5$)		18	
96:	sw R3$, rank_OFF(R5$)		19	
97:	sw R1$, prnt_OFF(R5$)		18	
98:	add R3$, R3$, R7$		20	

[7] Same memory accessed as in line #74 (spatial locality).

[8] Same memory accessed as in line #80 (spatial locality).

99:	sb R2$, done_OFF(R5$)		18	
100:	sb R2$, done_OFF(R9$)		19	
101:	lw R8$, next_OFF(R5$)		16[9]	
102:	move R4$, R8$		18	
103:	beq R8$, R0, CUT		18	
104:	lb R6$, select_OFF(R8$)	$2*(\frac{3M_i}{16})$ per Iter	17-21	
105:	beq R6$, R10, CUT		22	
106:	lw R7$, rank_OFF(R8$)	$11*(\frac{4M_i}{32})$ per Iter	23	
107:	lw R9$, actv_ptr_OFF(R8$)		23	
108:	sw R3$, rank_OFF(R8$)		24	
109:	sw R1$, prnt_OFF(R8$)		23	
110:	add R3$, R3$, R7$		25	
111:	sb R2$, done_OFF(R8$)		23	
112:	sb R2$, done_OFF(R9$)		24	
113:	lw R4$, next_OFF(R8$)		23[10]	
114:	beq R4$, R0, CUT		24	
115:	lb R6$, select_OFF(R4$)		24-28	
116:	beq R6$, R10, CUT		29	
117:	lw R7$, rank_OFF(R4$)	$12*(\frac{5M_i}{64})$ per Iter	30	
118:	lw R9$, actv_ptr_OFF(R4$)		30	
119:	sw R3$, rank_OFF(R4$)		31	
120:	sw R1$, prnt_OFF(R4$)		30	
121:	add R3$, R3$, R7$		32	

[9]Same memory accessed as in line #92 (spatial locality).

[10]Same memory accessed as in line #104 (spatial locality).

122:	sb R2$, done_OFF(R4$)		30	
123:	sb R2$, done_OFF(R9$)		31	
124:	lw R5$, next_OFF(R4$)		28[11]	
125:	move R4$, R5$		30	
126:	beq R5$, R0, CUT		30	
127:	lb R6$, select_OFF(R5$)		29-33	
128:	beq R6$, R10, CUT		34	
129:	lw R7$, rank_OFF(R5$)	$12*(\frac{6M_i}{128})$	35	
130:	lw R9$, actv_ptr_OFF(R5$)	per Iter	35	
131:	sw R3$, rank_OFF(R5$)		36	
132:	sw R1$, prnt_OFF(R5$)		35	
133:	add R3$, R3$, R7$		37	
134:	sb R2$, done_OFF(R5$)		35	
135:	sb R2$, done_OFF(R9$)		36	
136:	lw R8$, next_OFF(R5$)		33[12]	
137:	move R4$, R8$		35	
138:	beq R8$, R0, CUT		35	
139:	lb R6$, select_OFF(R8$)		34-38	
140:	beq R6$, R10, CUT		39	
141:	lw R7$, rank_OFF(R8$)	$8*(\frac{7M_i}{256})$	40	
142:	lw R9$, actv_ptr_OFF(R8$)	per Iter	40	
143:	sw R3$, rank_OFF(R8$)		41	
144:	sw R1$, prnt_OFF(R8$)		40	

[11] Same memory accessed as in line #115 (spatial locality).

[12] Same memory accessed as in line #127 (spatial locality).

145:	add R3$, R3$, R7$		42		
146:	sb R2$, done_OFF(R8$)		40		
147:	sb R2$, done_OFF(R9$)		41		
148:	lw R4$, next_OFF(R8$)		40[13]		
149:	CUT:	$4*\frac{M_i}{4}$ per Iter			
150:	sw R3$, rank_OFF(R1$)		43		
151:	sw R4$, next_OFF(R1$)		41		
152:	swa R3$, rank_OFF(R2), actv_elmnt_SIZE[R0$]		43		
153:	swa R4$, next_OFF(R2), actv_elmnt_SIZE[R0$]		41		
154:	SKPD:	$1*M_i$ per Iter			
155:	join SPWNR, R1		44		
156:	spawn SPWNR, 0, R1, 11	1*Iter	1		19*Iter
157:	lwa R1$, 0(R2), actv_elmnt_SIZE[R0$]	$8*M_i$ per Iter	1-5		
158:	lba R2$, done_OFF(R2), actv_elmnt_SIZE[R0$]		1-5		
159:	lwa R6$, back_OFF(R2), actv_elmnt_SIZE[R0$]		1-5		
160:	lwa R5$, next_OFF(R2), actv_elmnt_SIZE[R0$]		1-5		
161:	li R3$, 1		2		
162:	lw R4$, rank_OFF(R1$)		6-10		
163:	lw R11$, prnt_OFF(R1$)		6-10		
164:	bne R2$, R0, NON_ACTV		6		
165:	ps R3, R3$	$8*M'_i$	7		

[13] Same memory accessed as in line #139 (spatial locality).

166:	laa R10$, 0(R9), actv_elmnt_SIZE[R3$]	per Iter	8	
167:	swa R1$, 0(R9), actv_elmnt_SIZE[R3$]		8	
168:	sw R10$, actv_ptr_OFF(R1$)		9	
169:	swa R4$, rank_OFF(R9), actv_elmnt_SIZE[R3$]		11	
170:	swa R5$, next_OFF(R9), actv_elmnt_SIZE[R3$]		8	
171:	swa R6$, back_OFF(R9), actv_elmnt_SIZE[R3$]		8	
172:	beq R5$, R0, SELECT		7	
173:	li R10$, 1	$4*(M'_i-1)$ per Iter	10	
174:	subi R9$, R5$, LOCAL_LIST		6[14]	
175:	divi R9$, R9$, list_element_SIZE		7	
176:	lb R9$, COIN(R9$)		8-12	
177:	SELECT:	$4*M'_i$ per Iter		
178:	subi R8$, R1$, LOCAL_LIST		6	
179:	divi R8$, R8$, list_element_SIZE		7	
180:	lb R7$, COIN(R8$)		8-12	
181:	beq R6$, R0, END_SLCTD		8	
182:	beq R5$, R0, END_NOT_SLCTD	$1*(M'_i-1)$ per Iter	9	
183:	andi R7$, R7$, 0x01	$2*(M'_i-2)$ per Iter	13	
184:	bne R7$, R10, END_NOT_SLCTD		14	

[14]Speculation (of the branch in line #172) with very high probability, since only one element is the last element in the list.

185:	andi R9$, R9$, 0x01	$2*(\frac{M_{i'}}{2})$	13		
186:	beq R9$, R0, END_SLCTD	per Iter	15		
187:	END_NOT_SLCTD:	$1*(\frac{M_{i'}}{4})$			
188:	li R10$, 0	per Iter	16		
189:	END_SLCTD:	$5*M'_i$			
190:	sb R10$, select_OFF(R1$)	per Iter	17		
191:	sba R10$, select_OFF(R9), actv_elmnt_SIZE[R3$]		17		
192:	sb R0, done_OFF(R1$)		17		
193:	sba R0, done_OFF(R9), actv_elmnt_SIZE[R3$]		17		
194:	j DONE		17		
195:	NON_ACTV:	$4*(M_i-M'_i)$			
196:	ps R4, R3$	per Iter	7		
197:	swa R1$, 0(R5), skpd_SIZE[R3$]		8		
198:	swa R4$, rank_OFF(R5), skpd_SIZE[R3$]		11		
199:	swa R11$, prnt_addr_OFF(R5), skpd_SIZE[R3$]		11		
200:	DONE:	$1*M_i$			
201:	join SPWNR, R1	per Iter	18		
202:	move R1, R3	1*Iter	19		
203:	j END_FWDITR		x		
204:	BCKITR:	1*Iter			13*Iter
205:	spawn SPWNR, 0, R1, 4		1		
206:	lwa R1$, 0(R5), skpd_SIZE[R0$]	7*N	1-5		
207:	lwa R3$, rank_OFF(R5), skpd_SIZE[R0$]		1-5		

208:	lwa R4$, prnt_addr_OFF(R5), skpd_SIZE[R0$]		1-5		
209:	lw R2$, rank_OFF(R4$)		6-10		
210:	sub R3$, R2$, R3$		11		
211:	sw R3$, rank_OFF(R1$)		12		
212:	DONE2:				
213:	join SPWNR, R1		13		
214:	j END_BCKITR	0	x		

Total Work:

$$\begin{aligned}
&(5+4+1+2-3-1-4-2)+(2+7+3+2+3-5-2+1-4-1-4+1+1)*Iter+\\
&(12+3+2+1+2+\tfrac{2}{2}+\tfrac{3}{4}+5+7)*N+\\
&(3+12*\tfrac{3}{16}+11*\tfrac{4}{32}+12*\tfrac{5}{64}+12*\tfrac{6}{128}+8*\tfrac{7}{256}+1+8+4+1)*\sum M_i+\\
&(4+10+2+4)*\tfrac{1}{4}*\sum M_i+(8+4+4+1+2+\tfrac{2}{2}+\tfrac{1}{4}+5-4)*\sum M_i'\\
&=2+4*Iter+33\tfrac{3}{4}*N+(27\tfrac{11}{32})*\sum M_i+(21\tfrac{1}{4})*\sum M_i'\\
&=[33\tfrac{3}{4}+\tfrac{256}{183}*(27\tfrac{11}{32}+\tfrac{73}{256}*21\tfrac{1}{4})]*N+4*\log_{\frac{256}{73}}N+2\\
&=80.5*N+4*0.552*\log_2 N+2\\
&=80.5*N+2.21*\log_2 N+2
\end{aligned}$$

Total Time:

$$\begin{aligned}
&(2+1+6+1+17)+(1+1+3+44+19+13)*Iter\\
&=81*\log_{\frac{256}{73}}N+27=81*0.552*\log_2 N+27\\
&\le 44.71*\log_2 N+27
\end{aligned}$$

Upper Bound on Total Execution Time resulting from the formula (W/P + T):

$$\frac{(80.5*N+2.21*\log_2 N+2)}{P}+(44.71*\log_2 N+27)$$

Wyllie's Parallel List Ranking
Time and Work Analysis

Legend for work and time analysis:

N	- is the input size (size of the list)
#bits	- is the number of bits in an integer

A full understanding of the work and time counting requires reading the detailed counting assumptions and techniques documentation. Below (in the footnotes) there is a partial explanation.

Line #	Instruction Code	Total Work per block	Time Cycle inside block		Total Time
			1st Iter.	2nd Iter.	per block
1:	li R6, 1	3	1		14
2:	ble R1, R6, EXIT_WYLLIE		2		
3:	li R3, list1_ADDR		2[1]		
4:	spawn SPWNR, 0, R1, 2	1	3		
5:	lwa R1$, next_OFF(R2), actv_element_SIZE[R0$]	3*N	1-5		
6:	lwa R2$, rank_OFF(R2), actv_element_SIZE[R0$]		1-5		
7:	beq R1$, R0, END_PREPARE		6		
8:	lw R1$, actv_ptr_OFF(R1$)	3*(N-1)	6-10		
9:	sub R1$, R1$, R2		11		
10:	divi R1$, R1$, actv_element_SIZE		12		

[1]Speculation with very high probability, since in most cases the list-size will be larger than 1.

11:	END_PREPARE:	3*N			
12:	swa R2$, rank_OFF(R3), element_SIZE[R0$]		7		
13:	swa R1$, next_OFF(R3), element_SIZE[R0$]		13		
14:	join SPWNR, R1		14		
15:	li R6, list2_ADDR	2	2		
16:	li R5, bits_in_word_SIZE		2		
17:	spawn SPWNR, 0, R5, 2	1	1		7
18:	li R2$, 1	3*(#bits)	2		
19:	srlv R1$, R1, R0$		2		
20:	bne R1$, R2$, DONE_LOG2		3		
21:	sllv R1$, R1$, R0$	2	4		
22:	beq R1$, R1, EXACT_POWER2		5		
23:	addi R4, R0$, 1	2[2]	6		
24:	j DONE_LOG2		6		
25:	EXACT_POWER2:				
26:	move R4, R0$		6		
27:	DONE_LOG2:	1*(#bits)			
28:	join SPWNR, R5		7		

[2]Executes either instructions 23-24 (most of the time) OR instruction 26 (once), so we can take it as work of 2 instructions.

29:	JFWD_WYLLIE:	1			(1+15* $\log_2 N$) since 16 for 1st iter. & 15 more per iter. ($\log_2 N$-1)
30:	spawn SPWNR, 0, R1, 3		1	15	
31:	lwa R1$, next_OFF(R3), element_SIZE[R0$]	3*N*$\log_2 N$[3]	1-5	16-20	
32:	lwa R2$, rank_OFF(R3), element_SIZE[R0$]		1-5	16-20	
33:	beq R1$, R0, END_SKP		6	21	
34:	lwa R3$, rank_OFF(R3), element_SIZE[R1$]	3*(N-1)* $\log_2 N$	6-10	21-25	
35:	lw R1$, next_OFF(R3), element_SIZE[R1$]		6-10	21-25	
36:	beq R1$, R0, END_SKP		11	26	
37:	add R2$, R2$, R3$	1*[N*($\log_2 N$-1)+1]	12	27	
38:	END_SKP:	3*N*$\log_2 N$			
39:	swa R2$, rank_OFF(R6), element_SIZE[R0$]		13	28	
40:	swa R1$, next_OFF(R6), element_SIZE[R0$]		11	26	
41:	join SPWNR, R1		14	29	
42:	xor R6, R6, R3	5*$\log_2 N$	14	29	
43:	xor R3, R6, R3		15	30	
44:	xor R6, R6, R3		16	31	
45:	subi R4, R4, 1		2	16	
46:	bgt R4, R0, JFWD_WYLLIE		3	17	

[3] These forward iterations are run $\log_2 N$ times, spawning for every element in the list.

47:	spawn SPWNR, 0, R1, 2	1	1		7
48:	lwa R1$, 0(R2), actv_element_SIZE[R0$]	4*N	1-5		
49:	lwa R2$, rank_OFF(R3), element_SIZE[R0$]		1-5		
50:	sw R2$, rank_OFF(R1$)		6		
51:	join SPWNR, R1		7		
52:	EXIT_WYLLIE:	1			1
53:	jr		1		

Total Work:

$$
\begin{aligned}
&3+1+3*N+3*(N-1)+3*N+2+1+3*(\#bits)+2+2+1*(\#bits)+1+3*N\log_2 N \\
&+3*(N-1)*\log_2 N+1*[N*(\log_2 N-1)+1]+3*N\log_2 N+5*\log_2 N+1+4*N+1 \\
&= 10*N\log_2 N+12*N+2*\log_2 N+4*(\#bits)+12 \\
&= 10N\log_2 N+12N+2\log_2 N+4*64+12 \\
&= 10N\log_2 N+12N+2\log_2 N+268
\end{aligned}
$$

Total Time:

$$
\begin{aligned}
&14+7+(1+15*\log_2 N)+7+1 \\
&= 15\log_2 N+30
\end{aligned}
$$

Upper Bound on Total Execution Time resulting from the formula (W/P + T):

$$
\frac{(10\,N\log_2 N+12N+2\log_2 N+268)}{P}+(15\log_2 N+30)
$$

Parallel Reid-Miller
List Ranking
Time and Work Analysis

The time and work analysis of the Reid-Miller list ranking (also referred to as list scan) algorithm is presented.

Legend and assumptions for work and time analysis:

N - is the input size (size of the list)

Iter - is the number of iterations necessary,
e.g. $\log_{(1/0.221)}(N)$ $0.46*\log_2 N$ for $K_i = ¼*M_i$
(supported by simulations as $0.43*\log_2 N$)

M_i - is the number of active elements in the i-th iteration
where: $M_0 = N$, $M_1 <= (0.221)*N$, ...

the sum of the series is:

$$\sum M_i \le N + (0.221) * N + \ldots + (0.221)^i * N + \ldots \le \frac{1}{(1 - 0.221)} * N = (\tfrac{1}{0.779}) * N \le 1.29 * N$$

(supported by simulations as 1.28*N)

K_i - is the number of elements randomly selected (non unique) at the i-th iteration. In our case $K_i = 1/4*M_i$

M_i' - is the number of unique elements selected (from the K_i elements), and the actual number of elements performing short-cutting in the i-the iteration. It is thus the active elements starting the (i+1)-th iteration, e.g., $M_{i+1} <= (0.221)* M_i$

max_chain / max_elements skipped
- for $K_i = (1/4)* M_i$ the probability of having a chain longer than j is

$$P(length \ \ge \ j) = (0.221) \cdot (0.779)^j$$

(see the analysis presented earlier).
We take max_chain as the j for which that probability is <= 1/N.
We get $j > 2.77*\log_2 N$. According to the simulations we can use max_chain $2.75*\log_2 N$ (which we do).

A full understanding of the work and time counting requires reading the detailed counting assumptions and techniques documentation. Below (in the footnotes) there is a partial explanation.

Line #	Instruction Code	Total Work per block	Time Cycle inside block		Total Time
			1st Iter.	2nd Iter.	per block
1:	lw R1, list_size_OFF(SP)	7[1]	1		2
2:	ble R1, R10, EXIT1		2		
3:	lw R2, list_OFF(SP)		2[2]		
4:	la R9, actv(R0)		2		
5:	li R8, 0		2		
6:	la R5, nactv(R0)		2		
7:	li R6, 0		2		
8:	j FIRST_FWDITR		X		
9:	JFWD:	0			0
10:	j FWDITR		X		
11:	END_FWDITR:	6*Iter			1*Iter
12:	swa R4, bck_num_OFF+iter_info(R0), iter_SIZE[R8]		0		
13:	swa R5, bck_start_addr_OFF+iter_info(R0), iter_SIZE[R8]		0		
14:	addi R8, R8, 1		1		
15:	add R6, R6, R4		0		
16:	laa R5, nactv(R0), skpd_SIZE[R6]		1		
17:	bne R1, R10, JFWD		1		

[1]The jump instructions that can be inlined (such as the ones in lines # 8 and 10) are not counted, as work or as time, and are in the code for readability purposes only.

[2]Speculation (of branch in line #2) with very high probability, since in most cases the list-size will be larger than 1.

18:	JBCK:				
19:	subi R8, R8, 1	3*Iter	1	t+3	6+3*Iter since 7 for 1st iter. & 3 more per iter. & 2 more for last iter.
20:	lwa R5, bck_start_addr_OFF+iter_info(R0), iter_SIZE[R8]		1-5	(t-1) – (t+3)	
21:	lwa R1, bck_num_OFF+iter_info(R0), iter_SIZE[R8]		1-5		
22:	j BCKITR		X	X	
23:	END_BCKITR:				
24:	subi R6, R6, R1	2*Iter	t+1[3]		
25:	bgt R6, R0, JBCK		t+2		
26:	EXIT1:	1			1
27:	jr		1		
28:	FIRST_FWDITR:				
29:	divi R11, R1, 4	3	1		21
30:	li R12, 0		1		
31:	spawn SPWNR, 0, R1, 1		1		
32:	swa R0, select_OFF(R2), list_element_SIZE[R0$]	$3*M_0 =$ $2*M_0 +$ $1*N$	2		
33:	lwa R1$, back_OFF(R2), list_element_SIZE[R0$]		1-5		
34:	bne R1$, R0, NOT_HEAD		6		
35:	laa R7, 0(R2), list_element_SIZE[R0$]	1	7		
36:	NOT_HEAD:	$1* M_0$			
37:	join SPWNR, R1		8		

[3] “t” is the time-cycle in which the first backward iteration (BCKITR, line #135) is finished, and the program flow continues at line #24. The timing of the following backward iterations can be somewhat overlapped with the current iteration.

spawn SPWNR, 0, R11, 3	1	9		
li R3$, 1	2*K_0	10		
bne R0$, R0, OTHER_THREADS1		10		
move R1$, R7	2	11		
j CONT_HEAD1		11		
OTHER_THREADS1:	2*(K_0 -1)			
R1$ = rand(1,R1-1)[4]		11		
laa R1$, 0(R2), list_element_SIZE[R1$]		12		
CONT_HEAD1:	5*K_0			
markM select_OFF(R1$), R2$[5]		13-17		
bne R2$, R0, NOT_SELECTED1		18		
ps R12, R3$		19		
swa R1$, 0(R13), ptr_SIZE[R3$]		20		
NOT_SELECTED1:				
join SPWNR, R11		21		
j SHORT_CUTTING	0	X		
FWDITR:	3*(Iter-1)			24*(Iter-1)
divi R11, R1, 4		1		
li R12, 0		1		
spawn SPWNR, 0, R1, 1		1		
lwa R1$, list_ptr_OFF(R9), actv_element_SIZE[R0$]	3*M_i	1-5		
sw R0, select_OFF(R1$)		6		
join SPWNR, R1		7		

[4]The random selection is presented and counted as only one operation per iteration. This implies that the random numbers should be prepared ahead of time and at a minimum scaled (which should require at least one additional operation) to the relevant list-size at each iteration (to achieve uniform distribution for each iteration).

[5]We used the **markM** atomic instruction to achieve uniqueness of the *k* selected elements. This instruction is counted as 1 work unit and 5 time units (since it has to read a memory location). In the case where an explicit uniqueness examination is wanted the time and work necessary increase. These assumptions were used in order to favor the Reid-Miller algorithm in our comparison.

spawn SPWNR, 0, R11, 3	1*(Iter-1)	8		
li R3$, 1		9		
bne R0$, R0, OTHER_THREADS	9*K_i	9		
move R1$, R7		10		
j CONT_HEAD		10		
OTHER_THREADS:				
R1$ = rand(1,R1-1)[4]		10		
lwa R1$, list_ptr_OFF(R9), actv_element_SIZE[R1$]		11-15		
CONT_HEAD:				
markM select_OFF(R1$), R2$[5]		16-20		
bne R2$, R0, NOT_SELECTED		21		
ps R12, R3$		22		
swa R1$, 0(R13), ptr_SIZE[R3$]		23		
NOT_SELECTED:				
join SPWNR, R11		24		
SHORT_CUTTING:				
li R3, 0	3*Iter	0		
li R4, 0		0		
spawn SPWNR, 0, R12, 8		1		
lwa R1$, 0(R13), ptr_SIZE[R0$]		0-4		10*Iter (before skipping elements)
lw R3$, rank_OFF(R1$)	4*M_{i+1}	5-9		
lw R4$, next_OFF(R1$)		5-9		
beq R4$, R0, CUT		10		

ADVNC:				
lba R6$, select_OFF(R4$)	14* $(M_i - M_{i+1})$ per Iter start at i=0 since 14 per skipped element	10-14		3+7*skpd per Iter since 10 for 1st skipped element per iter. & 7 more per additional skipped element per iter.
beq R6$, R10, CUT		15		
lw R7$, rank_OFF(R4$)		16		
sw R3$, rank_OFF(R4$)		17		
sw R1$, prnt_OFF(R4$)		16		
li R2$, 1		16		
ps R4, R2$		17		
swa R4$, 0(R5), skpd_SIZE[R2$]		18		
swa R3$, rank_OFF(R5), skpd_SIZE[R2$]		18		
swa R1$, prnt_addr_OFF(R5), skpd_SIZE[R2$]		18		
add R3$, R3$, R7$		19		
lw R5$, next_OFF(R4$)		16		
move R4$, R5$		19		
beq R5$, R0, CUT		17		
lb R6$, select_OFF(R5$)		17-21		
beq R6$, R10, CUT		22		
lw R7$, rank_OFF(R5$)		23		
sw R3$, rank_OFF(R5$)		24		
sw R1$, prnt_OFF(R5$)		23		
li R2$, 1		23		
ps R4, R2$		24		
swa R5$, 0(R5), skpd_SIZE[R2$]		25		
swa R3$, rank_OFF(R5), skpd_SIZE[R2$]		25		
swa R1$, prnt_addr_OFF(R5), skpd_SIZE[R2$]		25		
add R3$, R3$, R7$		26		

110:	lw R8$, next_OFF(R5$)		19-23		
111:	move R4$, R8$		24		
112:	beq R8$, R0, CUT		24		
113:	lb R6$, select_OFF(R8$)		24-28		
114:	beq R6$, R10, CUT		29		
115:	lw R7$, rank_OFF(R8$)		30		
116:	sw R3$, rank_OFF(R8$)		31		
117:	sw R1$, prnt_OFF(R8$)		30		
118:	li R2$, 1		30		
119:	ps R4, R2$		31		
120:	swa R8$, 0(R5), skpd_SIZE[R2$]		32		
121:	swa R3$, rank_OFF(R5), skpd_SIZE[R2$]		32		
122:	swa R1$, prnt_addr_OFF(R5), skpd_SIZE[R2$]		32		
123:	add R3$, R3$, R7$		33		
124:	lw R4$, next_OFF(R8$)		30		
125:	bne R4$, R0, ADVNC		31		
126:	CUT:				
127:	sw R3$, rank_OFF(R1$)	$6*M_{i+1}$ per Iter start at i=0	t+1[4]		4*Iter (after skipping elements)
128:	sw R4$, next_OFF(R1$)		t+1		
129:	li R2$, 1		t		
130:	ps R3, R2$		t+1		
131:	swa R1$, list_ptr_OFF(R9), actv_elmnt_SIZE[R2$]		t+2		
132:	join SPWNR, R12		t+3		
133:	move R1, R3	1*Iter	t+4		
134:	j END_FWDITR		X		

[4] "t" is determined acording to the number of nodes that are skipped.

135:	BCKITR:	1*Iter			13*Iter
136:	spawn SPWNR, 0, R1, 4		1		
137:	lwa R1$, 0(R5), skpd_SIZE[R0$]	7*N	1-5		
138:	lwa R3$, rank_OFF(R5), skpd_SIZE[R0$]		1-5		
139:	lwa R4$, prnt_addr_OFF(R5), skpd_SIZE[R0$]		1-5		
140:	lw R2$, rank_OFF(R4$)		6-10		
141:	sub R3$, R2$, R3$		11		
142:	sw R3$, rank_OFF(R1$)		12		
143:	DONE2:				
144:	join SPWNR, R1		13		
145:	j END_BCKITR	0	X		

Total Work:

$$(7+0+1+1)+(6+3+2+3+1+3+1+1)*\mathit{Iter}+$$
$$(1+7)*N+9*\sum K_i+(3+14)*\sum M_i+(4-14+6)*\sum M_{i+1}$$
$$=9+20*\mathit{Iter}+8*N+17*\sum M_i+9*\sum K_i-4*\sum M_{i+1}$$
$$=(8+4)*N+(17+\tfrac{9}{4}-4)*\sum M_i+20*\mathit{Iter}+9\le(12+15\tfrac{1}{4}*1.29)*N+20*0.46*\log_2 N+9$$
$$\le 32*N+10*\log_2 N+9$$

Total Time:

$$(2+6+1-3)+(1+3+24+10+3+4+13)*\mathit{Iter}+7*\mathit{Iter}*(\#\mathrm{max}\_\mathit{elementsskipped}$$
$$=7*(\mathrm{max}\_\mathit{chain})*0.46*\log_2 N+58*0.46*\log_2 N+6$$
$$\le 7*0.46*2.75*(\log_2 N)^2+58*0.46*\log_2 N+6$$
$$\le 9*(\log_2 N)^2+27*\log_2 N+6$$

Upper Bound on Total Execution Time resulting from the formula (W/P + T):

$$\frac{(32*N+10*\log_2 N+9)}{P}+(9*(\log_2 N)^2+27*\log_2 N+6)$$