Integer Sort
Using Explicit Instruction Parallelism

Code and Analysis Documentation

Joseph Nuzman
advisor: Prof. Uzi Vishkin

January 20, 1998

This document contains the C (extended C) and assembly code for quicksort implementations in both the serial and Spawn MT programming models. Following the code, the document contains tables with a step-wise analysis of execution and the work and time counts for each of the algorithms.

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

Table of Contents

Appendix A: C and Assembly Code

This appendix is a listing of both algorithms C code, followed by the machine assembly codes.

C Serial Code: ser_radixsort.c

```
/* Serial radix sort */
/* Joe Nuzman */
/* 6/30/97 */

/* function to get started */
void integersort(unsigned int data[], int n) {

  unsigned int scratch[n], *temp;
  int range;
  int radixBits = RAD_BITS;
  range = 1 << radixBits;

  //iterate over all bitfields
  for (int i=0; i<NUMDIGS; i+=radixBits) {
    serStable(data, scratch, n, i, range);
    temp = scratch;
    scratch = data;
    data = temp;
  }
}

void serStable(unsigned int indata[], unsigned int outdata[], int n,
             int bitshift, int range) {

  int binCount[range], masked[n], temp;
  int sum = 0;

  //initialize bin counters to 0
  for (int i=0; i<range; i++) {
    binCount[i] = 0;
  }

  //increment the bin counter of the corresponding value for each element
  for (int ele=0; ele<n; ele++) {
    masked[ele] = (indata[ele] >> bitshift) && MASK;
    binCount[masked[ele]]++;
  }

  //perform prefix sum of bins
  for (int i=0; i<range; i++) {
    temp = sum;
    sum += binCount[i];
    binCount[i] = temp;
  }

  //copy each element to new location
  for (int ele=n-1; i>=0; i--) {
    outdata[binCount[masked[ele]]--] = indata[ele];
  }
}
```

C Spawn MT Code: par_radixsort.c

```
/* Parallel radix sort */
/* Joe Nuzman */
/* 7/17/97 */

/* function to get started */
void integersort(unsigned int data[], int n) {

  unsigned int scratch[n], *temp;
  int boundary[NUM_THREADS+1], range;
  int radixBits = RAD_BITS;
  int range = 1 << radixBits;
  int virtualThreads = NUM_THREADS;

  //set up boundary array
  spawn (virtualThreads+1;0) {
    boundary[$] = (n * $) / virtualThreads;
  }

  for (int i=0; i<NUMDIGS; i+=radixBits) {
    parStable(data, scratch, n, i, range, virtualThreads);
    temp = scratch;
    scratch = data;
    data = temp;
  }
}

void parStable(unsigned int indata[], unsigned int outdata[], int n,
             int bitshift, range, virtualThreads) {

  structure index {
    unsigned int masked; //holds masked bitfield, then bin index
    int count; //holds relative position for segment & value
  };

  int work[2*virtualThreads*range]; //leave room for binary tree ps
  int temp;
  int *binCount;
  int sum = 0;
  index iarray[n];
  int *binCount = work + virtualThreads*range;

  //compute masked bit-field for all elements
  spawn (n; 0) {
    iarray[$].masked = ((indata[$] >> bitshift) && MASK) * virtualThreads;
  }

  //spawn virtual threads to perform bin counting
  spawn (virtualThreads; 0) {
    for (int i=boundary[$]; i<boundary[$+1]; i++) {
      iarray[i].masked += $;
      iarray[i].count = binCount[iarray[i].masked]++;
    }
  }
```

```
  btreePrefixSum(work, binCount, virtualThreads*range);

  spawn (n; 0) {
    outdata[binCount[iarray[$].masked]-iarray[$].count];
  }
}
```

ASM Serial Code: ser_radixsort.asm

```
# Serial Radix Sort (mem)
# Bins in memory
# Joe Nuzman
# 7/28/97

# input
#   array of n integers
#   number of elements in the array (n)
# output:
#   sorted array of n integers

#*********
# void serialRS(unsigned int data[], int n) {
#*********

      # R1: Data array address
      # R2: Number of elements n
      # R3: Scratch array address
      # R4: Bit field indicator
      # R5: Bin counting array address
      # R6: Loop var
      # R7: Temporary result, R5 + RANGE*INT_SZ
      # R8: Temp val
      # R9: Temp val
      # R10:      Temporary result, R1 + R2*INT_SZ
      # R11:      Temp val

      lw    R1,   DATA_OFF(SP)
      lw    R2,   ARRAY_SIZE_OFF(SP)
      li    R3,   SCRATCH_ARRAY

      li    R5,   BINCOUNT_ARRAY
      li    R4,   0

      addi  R7,   R5,   RANGE*INT_SZ
      mult  R10,  R2,   INT_SZ
      add   R10,  R10,  R1

      # begin sorting each round
begin_round:

      #initialize bin registers
      sw    R0,   0*INT_SZ(R5)
      sw    R0,   1*INT_SZ(R5)
      sw    R0,   2*INT_SZ(R5)
      ...
      sw    R0,   (RANGE-1)*INT_SZ(R5)

      # count bin frequency
      move  R6,   R1
      move  R12,  R0
top_count_loop:
      lw    R8,   0(R6)
      srlv  R8,   R8,   R4
```

```
        andi  R11,  R8,   MASK
        multi R11,  R11,  INT_SZ
        add   R13,  R11,  R5
        sw    R13,  INDEX_VAL(R12)
        move  R14,  R13
        move  R15,  R13
        lw    R9,   0(R14)
        addi  R9,   R9,   INT_SZ
        sw    R9,   0(R15)
        addi  R6,   INT_SZ
        addi  R12,  INT_SZ
        blt   R6,   R10,  top_count_loop

        # perform prefix sum of bins
        move  R5,   R3
        lw    R11,  0*INT_SZ(R5)
        add   R8,   R8,   R11
        sw    R5,   0*INT_SZ(R5)

        move  R5,   R8
        lw    R11,  1*INT_SZ(R5)
        add   R8,   R8,   R11
        sw    R5,   1*INT_SZ(R5)

        move  R5,   R8
        lw    R11,  2*INT_SZ(R5)
        add   R8,   R8,   R11
        sw    R5,   2*INT_SZ(R5)

        . . .

        move  R5,   R8
        lw    R11,  (RANGE-1)*INT_SZ(R5)
        add   R8,   R8,   R11
        sw    R5,   (RANGE-1)*INT_SZ(R5)

        # prepare for copy loop
        move  R6,   R1
        move  R12,  R0

        # prepare for next iteration
        move  R13,  R1
        move  R1,   R3
        move  R3,   R13
        addi  R4,   R4,   RAD_BITS

        # copy elements to new position
top_copy_loop:
        lw    R8,   0(R6)
        lw    R11,  INDEX_VAL(R12)
        lw    R9,   0(R11)
        sw    R8,   0(R9)
        subi  R9,   R9,   INT_SZ
        sw    R9,   0(R11)
        addi  R6,   INT_SZ
        addi  R12,  INT_SZ
        blt   R6,   R10, top_copy_loop
```

```
blt   R4,   INT_BITS

jr 31
# end execution
```

ASM Spawn MT Code: par_radixsort.asm

```
# Parallel Radix Sort, PRAM algorithm, full spawning
#
# Joe Nuzman
# 1/14/97

# input
#   array of n integers
#   number of elements in the array (n)
# output:
#   sorted array of n integers

#*********
# void radixsort(unsigned int data[], int n) {
#*********

      # R1: Data array base address
      # R2: Number of elements n
      # R3: Scratch array input is copied to
      # R4: Bit field indicator
      # R5: Bin counting array base address
      # R6: Number of threads ("processors") to operate in parallel
      # R7: Partition boundary index array base address
      # R8: Masked bitfield value array base address
      # R9: Prefix sum binary tree storage array base address
      # R10:      Spawn register
      # R11:      Temporary value
      # R12:      Temporary value
      # R13:      Temporary value


      # load input array base and input size from stack
      lw    R1,   DATA_OFF(SP)
      lw    R2,   ARRAY_SIZE_OFF(SP)

      # determine number of parallel threads, choose a power of two
      li    R6,   NUM_THREADS

      # allocate array to copy input to,
      #   size R2
      li    R3,   SCRATCH_ARRAY
      # allocate array to store bin counts for each thread,
      # including space to store binary tree for prefix sum,
      # R9 points to root of tree
      # R5 points to base of bin array (leftmost leaf)
      #   size 2*RANGE*R6
      li    R9,   BINCOUNT_ARRAY
      laa   R5,   0(R9),      INT_SZ*RANGE[R6]
      # allocate array to store partition boundaries,
      #   size R6+1
      li    R7,   BOUNDARY_ARRAY
      #allocate array to store the masked bitfield values for each element,
      #   size R2
      li    R8,   INDEX_ARRAY
```

```
        # initialize bit field indicator to rightmost bitfield
        li    R4,   0

        # evaluate partition boundaries, store to array
        addi  R11,  R6,   1
        spawn R10, 0, R11, 1
        # R1$:      Partition boundary index
        mult  R1$,  R2,   R0$
        div   R1$,  R1$,  R6
        swa   R1$,  0(R7),      INT_SZ[R0$]
        join  R10, R11

begin_round:

        # load, mask, and store for all elements
        spawn R10, 0, R2, 1
        # R1$:      Working variable
        # load array element
        lwa   R1$,  0(R1),      INT_SZ[R0$]
        # mask bitfield
        srlv  R1$,  R1$,  R4
        andi  R1$,  R1$,  MASK
        # store masked value to index array
        mult  R1$,  R1$,  R6
        swa   R1$,  MASKED(R8), INDEX_SZ[R0$]
        join  R10, R2


        # spawn virtual processors to perform bin counting
        spawn R10, 0, R6, 7

        # R1$:      Thread offset into bin counter array
        # R2$:      Current element from input array
        # R3$:      Boundary element of input array for partition
        # R4$:      Temporary variable
        # R5$:      Temporary variable
        # R6$:      Holds bin counter incremented
        # R7$:      Temporary variable, used to allow overlapping loops

        # load in partition bounds
        lwa   R2$,  0(R7),      INT_SZ[R0$]
        lwa   R3$,  INT_SZ(R7), INT_SZ[R0$]

        laa   R1$,  0(R5),      INT_SZ[R0$]

        #initialize bin counters
        swa   R0,   0(R1$),     0*INT_SZ[R6]
        swa   R0,   0(R1$),     1*INT_SZ[R6]
        swa   R0,   0(R1$),     2*INT_SZ[R6]
        ...
        swa   R0,   0(R1$),     (RANGE-1)*INT_SZ[R6]

        # loop to count bin frequency
top_count_loop:
        #load in masked bitfield
        lwa   R5$,  MASKED(R8), INDEX_SZ[R2$]
        # calculate and remember bin address
```

```
        laa   R4$,  0(R1$),     INT_SZ[R5$]
        swa   R4$,  MASKED-INDEX_SZ(R8),    INDEX_SZ[R2$]     #now ptr to bin
        move  R4$,  R5$
        addi  R2$,  R2$,  1
        # load bin counter
        lwa   R6$,  0(R1$),     INT_SZ[R4$]
        # save place for later
        swa   R6$,  COUNT-INDEX_SZ(R8),     INDEX_SZ[R2$]
        # increment bin counter
        addi  R7$,  R6$,  1
        swa   R7$,  0(R1$),     INT_SZ[R4$]

        # loop to top
        blt   R2$,  R3$,  top_count_loop

        join  R10, R6


        # synchronous, up-tree prefix-sum

        multi R12,  R6,   RANGE
        li    R11,  1
top:
        srl   R12,  1

        spawn R10, 0, R12, 3    #spawn for each node in level L, 0-(2^L-1)
        # R1$:      Left child value
        # R2$:      Right child value
        # R3$:      Index value
        add   R3$,  R12$, R0$
        lwa   R1$,  0(R9),      2*INT_SZ[R3$]     #left child
        lwa   R2$,  1*INT_SZ(R9),     2*INT_SZ[R3$]     #right child
        add   R1$,  R1$,  R2$
        swa   R1$,  0(R9),      INT_SZ[R3$] #current
        join  R10, R12

        bne   R12,  R11,  top


        # synchronous, down-tree prefix-sum
        multi R13,  R6,   RANGE
top:
        sll   R12,  R12,  1
        subi  R11,  R12,  1

        spawn R10, 0, R12, 3    #spawn for each node in level L, 1-2^L
        # R1$:      Up, or up-and-left index
        # R2$:      Even/odd bit
        # R3$:      Current index
        add   R3$,  R11,  R0$
        srl   R1$,  R3$,  1
        andi  R2$,  R3$,  1     #mask last bit
        beqz  R2$,  even  #even (right-side) nodes
        beq   R0$,  R0,   end   #leftmost node, 0
odd:
        lwa   R2$,  0(R9),      INT_SZ[R3$] #current
        lwa   R1$,  0(R9),      INT_SZ[R1$] #up and left
```

```
      add   R1$,  R2$,  R1$
      swa   R1$,  0(R9),      INT_SZ[R3$] #current
      j     end
even:
      lwa   R1$,  0(R9),      INT_SZ[R1$] #up
      swa   R1$,  0(R9),      INT_SZ[R3$] #current
end:
      join  R10, R12

      bne   R12,  13,   top


      # copy elements to new position
      spawn R10, 0, R2, 3
      # R1$:      Holds array element
      # R2$:      Holds bin counter address / bin counter value
      # R3$:      Holds intra-thread place / place adjusted array address
      # load array element
      lwa   R1$,  0(R1),      INT_SZ[R0$]
      # load bin counter address
      lwa   R2$,  MASKED(R8), INDEX_SZ[R0$]
      # load place within thread/bin
      lwa   R3$,  COUNT(R8),  INDEX_SZ[R0$]
      # load bin counter
      lw    R2$,  0(R2$)
      sub   R3$,  R3,   R3$
      # store element to indicated position in scratch array
      swa   R1$,  0(R3$),     INT_SZ[R2$]
      join  R10, R2


      # prepare for next iteration

      move  R13,  R1
      move  R1,   R3
      move  R3,   R13
      addi  R4,   R4,   RAD_BITS

      blt   R4,   INT_BITS,   BEGIN_ROUND

      jr 31
```

Appendix B: Time and Work Analysis

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

Note that in the following tables, the counts for each block attempt to account for the total time and work cost for that particular block of code during a run of the entire algorithm. So, for example, a code block that contains 7 instructions, but is executed exactly once for each element of the input during the course of the algorithm, would be assigned a work count of 7*N, where N is the input size (total number of elements). For this algorithm, the main loop is performed a certain number of iterations. In the counting tables, I place the count for one iteration within square brackets and multiply by i, the number of iterations.

The parameters R, the size of the radix, and v, the number of virtual threads, are free to be chosen by the program. These choices are discussed in the technical report.

Serial Integer Sort - Time & Work Analysis

Legend for work and time analysis:

- N - is the input size (size of the array to be sorted)
- D - is the number of bits in an integer
- R - is the number of bits per bit-field. The bit field is the portion of an integer that is sorted on at each iteration of the main sorting algorithm. The range of values each bit-field may take on is 2^R.
- i - is the number of iterations of the main sorting algorithm. A sort must be performed on each bit-field in an integer, equals the ceiling of D/R.

Line #	Instruction Code	Total Work per block	Time Cycle inside block		Total Time per block
			First Iter.	General Iter.	
1:	lw R1, DATA_OFF(SP)	8	1		3
2:	lw R2, ARRAY_SIZE_OFF(SP)		1		
3:	li R3, SCRATCH_ARRAY		1		
4:	li R5, BINCOUNT_ARRAY		1		
5:	li R4, 0		1		
6:	addi R7, R5, RANGE*INT_SZ		2		
7:	mult R10, R2, INT_SZ		2		
8:	add R10, R10, R1		3		
9:	begin_round:				
10:	sw R0, 0*INT_SZ(R5)	$[2^R + 2] * i$	-1	1	0 + [1] * (i - 1)
11:	sw R0, 1*INT_SZ(R5)		-1	1	
12:	sw R0, 2*INT_SZ(R5)		-1	1	
13:	...				

14:	sw R0, (RANGE-1)*INT_SZ(R5)		-1	1	
15:	move R6, R1		-1	-4	
16:	move R12, R0		-2	-4	
17:	top_count_loop:	[14*N] * i			[3*N + 4] * i
18:	lw R8, 0(R6)		-8 - -4	-5 - -1	
19:	srlv R8, R8, R4		-3	0	
20:	andi R11, R8, MASK		-2	1	
21:	multi R11, R11, INT_SZ		-1	2	
22:	add R13, R11, R5		0	3	
23:	sw R13, INDEX_VAL(R12)		2	5	
24:	move R14, R13		4	7	
25:	move R15, R14		5	8	
26:	lw R9, 0(R14)		1 - 5	4 - 8	
27:	addi R9, R9, INT_SZ		6	9	
28:	sw R9, 0(R15)		7	10	
29:	addi R6, R6, INT_SZ		-3	0	
30:	addi R12, R12, INT_SZ		2	5	
31:	blt R6, R10, top_count_loop		-2	1	
32:	move R5, R3	[4*2^R + 6] * i	-8		[2*2^R] * i
33:	lw R11, 0*INT_SZ(R5)		1		
34:	add R8, R8, R11		2		
35:	sw R5, 0*INT_SZ(R5)		2		
36:	move R5, R8		3		
37:	lw R11, 1*INT_SZ(R5)		3		
38:	add R8, R8, R11		4		
39:	sw R5, 1*INT_SZ(R5)		4		

40:	move R5, R8		5		
41:	lw R11, 2*INT_SZ(R5)		5		
42:	add R8, R8, R11		6		
43:	sw R5, 2*INT_SZ(R5)		6		
44:	. . .				
45:	move R5, R8		$2*2^R$ - 1		
46:	lw R11, (RANGE-1)*INT_SZ(R5)		$2*2^R$ - 1		
47:	add R8, R8, R11		$2*2^R$		
48:	sw R5, (RANGE-1)*INT_SZ(R5)		$2*2^R$		
49:	move R6, R1		-8		
50:	move R12, R0		-5		
51:	move R13, R1		-8		
52:	move R1, R3		-7		
53:	move R3, R13		-6		
54:	addi R4, R4, RAD_BITS		-8		
55:	top_copy_loop:	[9*N] * i			[4*N] * i
56:	lw R8, 0(R6)		-8 - 4	-4 - 0	
57:	lw R11, INDEX_VAL(R12)		-8 - 4	-4 - 0	
58:	lw R9, 0(R11)		-3 - 1	1 - 5	
59:	sw R8, 0(R9)		2	6	
60:	subi R9, R9, INT_SZ		3	7	
61:	sw R9, 0(R11)		4	8	
62:	addi R6, R6, INT_SZ		-3	1	
63:	addi R12, R12, INT_SZ		-3	1	
64:	blt R6, R10, top_copy_loop		-2	2	

65:	blt R4, INT_BITS, begin_round	[1] * i	-5		[0] * i
66:	jr 31	1	1		1

Total Work:

$W = [23*N + 5*2^R + 9] * i + 9$

Total Time:

$T = [7*N + 2*2^R + 5] * i + 10$

Parallel Integer Sort - Time & Work Analysis

Legend for work and time analysis:

- N - is the input size (size of the array to be sorted)
- D - is the number of bits in an integer
- R - is the number of bits per bit-field. The bit field is the portion of an integer that is sorted on at each iteration of the main sorting algorithm. The range of values each bit-field may take on is 2^R.
- i - is the number of iterations of the main sorting algorithm. A sort must be performed on each bit-field in an integer, equals the ceiling of D/R.
- v - is the number of virtual processors that perform the bin counting

Line #	Instruction Code	Total Work per block	Time Cycle inside block		Total Time per block
			First Iter.	General Iter.	
1:	lw R1, DATA_OFF(SP)	10	1		2
2:	lw R2, ARRAY_SIZE_OFF(SP)		1		
3:	li R6, NUM_THREADS		1		
4:	li R3, SCRATCH_ARRAY		1		
5:	li R9, BINCOUNT_ARRAY		1		
6:	laa R5, 0(R9), INT_SZ*RANGE[R6]		2		
7:	li R7, BOUNDARY_ARRAY		1		
8:	li R8, INDEX_ARRAY		1		
9:	li R4, 0		1		
10:	addi R11, R6, 1		2		
11:	spawn R10, 0, R11, 1	1 + 4 * (v + 1)	1		5
12:	mult R1$, R2, R0$		2		
13:	div R1$, R1$, R6		3		

14:	swa R1$, 0(R7), INT_SZ[R0$]		4		
15:	join R10, R11		5		
16:	begin_round:				
17:	spawn R10, 0, R2, 1	[1 + 6*N] * i	1		[7] * i
18:	lwa R1$, 0(R1), INT_SZ[R0$]		-2 - 2		
19:	srlv R1$, R1$, R4		3		
20:	andi R1$, R1$, MASK		4		
21:	mult R1$, R1$, R6		5		
22:	swa R1$, MASKED(R8), INDEX_SZ[R0$]		6		
23:	join R10, R2		7		
24:	spawn R10, 0, R6, 7	[1 + (3 + 2^R) * v] * i	1		[3] * i
25:	lwa R2$, 0(R7), INT_SZ[R0$]		-2 - 2		
26:	lwa R3$, INT_SZ(R7), INT_SZ[R0$]		-2 - 2		
27:	laa R1$, 0(R5), INT_SZ[R0$]		2		
28:	swa R0, 0(R1$), 0*INT_SZ[R6]		3		
29:	swa R0, 0(R1$), 1*INT_SZ[R6]		3		
30:	swa R0, 0(R1$), 2*INT_SZ[R6]		3		
31:	...				
32:	swa R0, 0(R1$), (RANGE-1)*INT_SZ[R6]		3		
33:	top_count_loop:	[10*N] * i			[5*N/v + 2] * i
34:	lwa R5$, MASKED(R8), INDEX_SZ[R2$]		-4 - 0	1 - 5	
35:	laa R4$, 0(R1$), INT_SZ[R5$]		1	6	

36:	swa R4$, MASKED-INDEX_SZ(R8), INDEX_SZ[R2$]		2	7	
37:	move R4$, R5$		6	11	
38:	addi R2$, R2$, 1		3	8	
39:	lwa R6$, 0(R1$), INT_SZ[R4$]		1 - 5	6 - 10	
40:	swa R6$, COUNT-INDEX_SZ(R8), INDEX_SZ[R2$]		6	11	
41:	addi R7$, R6$, 1		6	11	
42:	swa R7$, 0(R1$), INT_SZ[R4$]		7	12	
43:	blt R2$, R3$, top_count_loop		4	9	
44:	join R10, R6	[v] * i	1		[1] * i
45:	multi R12, R6, RANGE	[2] * i	1		[1] * i
46:	li R11, 1		1		
47:	top:	[3*log$_2$(2^R*v) + 6*(2^R*v - 1)] * i			[8*log$_2$(2^R*v)]*i
48:	srl R12, 1		1	9	
49:	spawn R10, 0, R12, 3		2	10	
50:	add R3$, R12$, R0$		3	11	
51:	lwa R1$, 0(R9), 2*INT_SZ[R3$]		0 - 4	8 - 12	
52:	lwa R2$, 1*INT_SZ(R9), 2*INT_SZ[R3$]		0 - 4	8 - 12	
53:	add R1$, R1$, R2$		5	13	
54:	swa R1$, 0(R9), INT_SZ[R3$]		6	14	
55:	join R10, R12		7	15	
56:	bne R12, R11, top		8	16	
57:	multi R13, R6, RANGE	[1] * i	1		[1] * i

58:	top:	$[4*\log_2(2^R*v) + 11*(2*2^R*v - 2) - 4*(2^R*v - 1) - 5*\log_2(2^R*v)] * i$			$[14*\log_2(2^R*v) - 1]*i$
59:	sll R12, R12, 1		0	14	
60:	subi R11, R12, 1		1	15	
61:	spawn R10, 0, R12, 3		2	16	
62:	add R3$, R11, R0$		3	17	
63:	srl R1$, R3$, 1		4	18	
64:	andi R2$, R3$, 1		4	18	
65:	beqz R2$, even		5	19	
66:	beq R0$, R0, end		6	20	
67:	odd:				
68:	lwa R2$, 0(R9), INT_SZ[R3$]		3 - 7	18 - 21	
69:	lwa R1$, 0(R9), INT_SZ[R1$]		5 - 9	19 - 23	
70:	add R1$, R2$, R1$		10	24	
71:	swa R1$, 0(R9), INT_SZ[R3$]		11	25	
72:	j end		7	21	
73:	even:				
74:	lwa R1$, 0(R9), INT_SZ[R1$]				
75:	swa R1$, 0(R9), INT_SZ[R3$]				
76:	end:				
77:	join R10, R12		12	26	
78:	bne R12, 13, top		13	27	
79:	spawn R10, 0, R2, 3	$[1 + 7*N] * i$	1		$[9] * i$
80:	lwa R1$, 0(R1), INT_SZ[R0$]		-2 - 2		
81:	lwa R2$, MASKED(R8), INDEX_SZ[R0$]		-2 - 2		
82:	lwa R3$, COUNT(R8), INDEX_SZ[R0$]		-2 - 2		

83:	lw R2$, 0(R2$)		3 - 7		
84:	sub R3$, R3, R3$		3		
85:	swa R1$, 0(R3$), INT_SZ[R2$]		8		
86:	join R10, R2		9		
87:	move R13, R1	[5] * i	1		[3] * i
88:	move R1, R3		2		
89:	move R3, R13		3		
90:	addi R4, R4, RAD_BITS		1		
91:	blt R4, INT_BITS, BEGIN_ROUND		2		
92:	jr 31	1	1		1

Total Work:

$$W = [23*N + (25*2^R + 4)*v + 2*R + 2*\log_2(v) - 13] * i + 4*v + 16$$

Total Time:

$$T = [5*N/v + 22*R + 22*\log_2(v) + 25] * i + 8$$