Quicksort
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 EMT 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.

Note that the total time and work results which appear here cannot be derived directly from the step-wise counts. The additional analysis required to reach these results will appear in the technical report.

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_quicksort.c

```
/* Serial quicksort */
/* Joe Nuzman */
/* 6/02/97 */


/* function to get started */
void quicksort(int data[], int n) {
  qsrecur(data,0,n-1);
}


/* recursive function */
void qsrecur(int data[], int first, int last) {
  int pivot, pivotval, lowindex, highindex, temp;

  if ((first - last) < THRESHOLD) {

    pivot = rand(first, last);

    /* swap first and pivot */
    pivotval = data[pivot];
    data[pivot] = data[first];
    data[first] = pivotval;

    /* partition */
    lowindex = first - 1;
    highindex = last + 1;
    while (1) {
      do {
      highindex--;
      } while (data[highindex] >= pivotval);
      do {
      lowindex++;
      } while (data[lowindex] <= pivotval);
      if (lowindex < highindex) {
      temp = data[lowindex];
      data[highindex] = data[lowindex];
      data[lowindex] = temp;
      } else break;
    }

    /* recursive calls */
    qsrecur(data,first,highindex);
    qsrecur(data,lowindex,last);

  }
  else
    smallSort(data, first, last);
}
```

C EMT Code: par_quicksort.c

```
/* Parallel quicksort */
/* Joe Nuzman */
/* 6/05/97 */


/* function to get started */
void quicksort(int data[], int n) {
  int scratch[n];
  qsrecur(data,scratch,n,1);
}


/* recursive function */
void qsrecur(int data[], int scratch[], int count, int useScratch) {
  int pivot, pivotval, lowindex, highindex;

  if (count < THRESHOLD) {
    smallSort(data, scratch, count, useScratch);
  }
  else {
    pivot = rand(last);
    pivotval = data[pivot];

    lowindex = 0;
    highindex = 1;
    spawn (count; 0) {
      int dest = 1;
      int curval;

      curval = data[$];
      if (curval < pivotval) {
      ps(lowindex, dest);
      scratch[dest] = curval;
      }
      else {
      ps(highindex, dest);
        scratch[count - dest] = curval;
      }
    }

    spawn (2; 0) {
      if ($) {
        qsrecur(scratch[lowindex],data[lowindex],highindex-1,!useScratch);
      }
      else {
      qsrecur(scratch,data,lowindex,!useScratch);
      }
    }
  }
}
```

ASM Serial Code: ser_quicksort.asm

```
# Serial Quicksort
# Joe Nuzman
# 6/12/97

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

#*********
# void quicksort(int data[], int n)
#*********

      # R1 :  low address of segment
      # R2 :  high address of segment
      # R3 :  low moving address
      # R4 :  high moving address
      # R5 :  load/store register
      # R6 :  load/store register
      # R7 :  pivot value

      # set things up
      lw    R1,   DATA_OFF(SP)
      lw    R2,   ARRAY_SIZE_OFF(SP)
      multi R2,   R2,   DATA_SZ
      subi  R2,   R2,   DATA_SZ
      add   R2,   R2,   R1
      jal   quicksort


      # **** end execution ****



      #begin recursive procedure quicksort
quicksort:
      sub   R5,   R2,   R1
      blt   R5,   THRESHOLD,  continue
      j     smallsort
endsmallsort:
      jr    R31

continue:
      # rand(low, high) not implemented
      # R3 = rand(R1, R2)

      # swap first and pivot elements
      lw    R5,   DATA(R1)
      lw    R7,   DATA(R3)
      sw    R5,   DATA(R3)
      sw    R7,   DATA(R1)

      # partition elements
```

```
      subi  R3,   R1,   1
      addi  R4,   R2,   1
startloop:

repeat1:
      subi  R4,   R4,   DATA_SZ
      lw    R6,   DATA(R4)
      ble   R6,   R7

repeat2:
      addi  R3,   R3,   DATA_SZ
      lw    R5,   DATA(R3)
      bge   R5,   R7,   repeat2

      blt   R3,   R4,   endloop
      sw    R5,   DATA(R4)
      sw    R6,   DATA(R3)
      j     startloop
endloop:

      # recursive call for lower partition
      sw    R2,   0(SP)
      sw    R31,  PTR_SZ(SP)
      addi  SP,   SP,   (2*PTR_SZ)
      move  R2,   R4
      jal   quicksort
      addi  R1,   R2,   1
      subi  SP,   SP,   (2*PTR_SZ)
      lw    R2,   0(SP)
      lw    R31,  PTR_SZ(SP)

      # loop for higher partition
      j     quicksort

      # end program
```

ASM EMT Code: par_quicksort.asm

```
# Parallel quicksort
# Joe Nuzman
# 6/12/97

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

#*********
# void quicksort(int data[], int n)
#*********

      # R1 :  number initially spawned = 1
      # R2 :  usual number spawned = 2
      # R3 :  spawning register

      li    R1,   1
      li    R2,   2


      # ** initial spawn, necessary to began recursive structure **
      spawn R3, 0, R1, 9

      # R1$ :  count for this thread
      # R2$ :  pivot value / thread control
      # R3$ :  low split index
      # R4$ :  high split index
      # R5$ :  thread control / jump register
      # R6$ :  if 0 write to data, if 1 write to scratch
      # R7$ :  data address
      # R8$ :  scratch address
      # R9$ :      link register

      #initialize regs
      lw    R1$,  ARRAY_SIZE_OFF(SP)
      lw    R7$,  DATA_OFF(SP)
      li    R8$,  SCRATCH_ARRAY
      li    R6$,  1
      li    R9$,  endquicksort

      # ***** begin recursive section *****
recur:

      # THRESHOLD to be determined
      blt   R1$,  THRESHOLD,  smallsort

      # rand(count) not implemented, returns index
      # R2$ = rand(R1$)
      lwa   R2$,  0(R7$),     DATA_SZ[R2$]

      # set up indexes for splitting high and low
      li    R3$,  0
```

```
      li    R4$,  1


      # *** spawn thread for each element to partition ***
      spawn R5$, 0, R1$, 3
      # R1$ :  data value
      # R2$ :  working var

      li    R2$,  1
      lwa   R1$,  0(R7P$),    DATA_SZ[R0$]

      bge   R1$,  R2P$, great_eqinit
      ps    R3P$, R2$
      j     endinit
great_eqinit:
      ps    R4P$, R2$
      sub   R2$,  R1P$, R2$
endinit:
      swa   R1$,  0(R7P$),    DATA_SZ[R2$]

      join  R5$, R1$

      # ****** spawn child threads ******
      spawn R2$, 0, R2, 9
      # R1$ :  count for this thread
      # R2$ :  pivot value / thread control
      # R3$ :  low split index
      # R4$ :  high split index
      # R5$ :  thread control / jump register
      # R6$ :  if 0 write to data, if 1 write to scratch
      # R7$ :  data address
      # R8$ :  scratch address
      # R9$ :      link register

      #initialize regs
      beqz  R0$,  left
      laa   R7$,  0(R8P$),    DATA_SZ[R3P$]
      laa   R8$,  0(R7P$),    DATA_SZ[R3P$]
      j     endlr
left:
      move  R7$,  R8P$
      move  R8$,  R7P$
      move  R1$,  R3P$
endlr:
      slti  R6$,  R6P$, 1

      jal   recur,      R9$

      join  R2$, R2

endsmallsort:
      jr    R9$

endquicksort:
      join  R3, R1

######### end execution #########
```

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). To facilitate this goal, the following notation is used:

N	- is the input size (size of the array to be sorted)
N_i	- is the size of a given partition i
$\sum^{AP}$	- is the sum over all partitions created during a run of the algorithm
$\sum^{CP}$	- is the sum over partitions on the critical path (the longest sequence)

A work count of $\sum^{AP}\{\text{term1}\}$ would indicate that term1 is to be counted for each partition the algorithm operates on. Note that term1 might involve N_i, the number of elements in a particular partition, and therefore would depend on the sizes of each of the partitions.

As an example, we are given an algorithm that operates on a partition of size 10, splits it into two partitions of size 3 and 7, operates on each, and terminates. The term $\sum^{AP}\{N_i + 2\}$ would then evaluate as: (10 + 2) + (3 + 2) + (7 + 2) = 26.

Similarly, a work count of $\sum^{CP}\{\text{term1}\}$ would indicate that term1 is to be counted for each partition that had to be evaluated along the critical path to the last partition evaluated.

For both algorithms, the total counts in this summation form (i.e., still as a function of $\sum^{AP}$ and $\sum^{CP}$) are easily derived from the step-wise counts. The derivation of direct bounds requires some analysis and appears in the full technical report.

Serial Quicksort - Time & Work Analysis

Legend for work and time analysis:

- N - is the input size (size of the array to be sorted)
- N_i - is the size of a given partition i
- $\sum^{AP}$ - is the sum over all partitions created during a run of the algorithm
- $\sum^{CP}$ - is the sum over partitions on the critical path (the longest sequence)

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)	6	1		3
2:	lw R2, ARRAY_SIZE_OFF(SP)		1		
3:	multi R2, R2, DATA_SZ		2		
4:	subi R1, R1, DATA_SZ		2		
5:	add R2, R2, R1		3		
6:	jal quicksort		1		
7:	quicksort:	$3*N + \sum^{AP}\{9\}$			$3*N + \sum^{AP}\{9\}$
8:	sub R5, R2, R1		1		
9:	blt R5, THRESHOLD, continue		2		
10:	j smallsort				
11:	endsmallsort:				
12:	jr R31		3		
13:	continue: { R6 = rand(R1,R2) }		3		
14:	lw R5, DATA(R1)		2 - 6		
15:	lw R7, DATA(R6)		4 - 8		
16:	sw R5, DATA(R6)		9		

17:	sw R7, DATA(R1)		9		
18:	subi R3, R1, 1		3		
19:	addi R4, R2, 1		3		
20:	startloop:				
21:	repeat1:	$\sum^{AP}\{(N_i + 2) * 3\}$			$\sum^{AP}\{(N_i + 2) * 3\}$
22:	subi R4, R4, DATA_SZ		1	4	
23:	lw R6, DATA(R4)		2	5	
24:	bge R6, R7		3	6	
25:	repeat2:				
26:	addi R3, R3, DATA_SZ		1	4	
27:	lw R5, DATA(R3)		2	5	
28:	blt R5, R7, repeat2		3	6	
29:	bge R3, R4, endloop	$\sum^{AP}\{{}^{3}/_{16} * N_i * 4\}$	1		$\sum^{AP}\{{}^{3}/_{16} * N_i * 2\}$
30:	sw R5, DATA(R4)		2		
31:	sw R6, DATA(R3)		2		
32:	j startloop		2		
33:	endloop:				
34:	sw R2, 0(SP)	$\sum^{AP}\{5\}$	1		$\sum^{AP}\{2\}$
35:	sw R31, PTR_SZ(SP)		1		
36:	addi SP, SP, (2*PTR_SZ)		2		
37:	move R2, R4		2		
38:	jal quicksort		1		
39:	addi R1, R2, 1	$\sum^{AP}\{5\}$			$\sum^{AP}\{3\}$
40:	subi SP, SP, (2*PTR_SZ)				
41:	lw R2, 0(SP)				
42:	lw R31, PTR_SZ(SP)				

43:	j quicksort				

Total Work:

$W = \sum^{AP}\{3.75*N_i + 25\} + 3N + 6$

Making certain assumptions, and solving a recurrence equation (analysis omitted here), the following bound can be derived:

$W = 4.62*N*\log_2(N) + 50.0*N - 19$

Total Time:

$T = \sum^{AP}\{3.375*N_i + 20\} + 3N +3$

Making certain assumptions, and solving a recurrence equation (analysis omitted here), the following bound can be derived:

$T = 4.16*N*\log_2(N) + 43.0*N - 17$

Parallel Quicksort - Time & Work Analysis

Legend for work and time analysis:

- N - is the input size (size of the array to be sorted)
- N_i - is the size of a given partition i
- $\sum^{AP}$ - is the sum over all partitions created during a run of the algorithm
- $\sum^{CP}$ - is the sum over partitions on the critical path (the longest sequence)

Line #	Instruction Code	Total Work per block	Time Cycle inside block		Total Time per block
			First Iter.	General Iter.	
1:	li R1, 1	8	1		3
2:	li R2, 2		1		
3:	spawn R3, 0, R1, 9		2		
4:	lw R1$, ARRAY_SIZE_OFF(SP)		3		
5:	lw R7$, DATA_OFF(SP)		3		
6:	li R8$, SCRATCH_ARRAY		3		
7:	li R6$, 1		3		
8:	li R9$, endquicksort		3		
9:	recur:	4N			5
10:	blt R1$, THRESHOLD, smallsort {R2 = rand(R1$)}	$\sum^{AP}\{5\}$	1 2		$\sum^{CP}\{7\}$
11:	lwa R2$, 0(R7$), DATA_SZ[R2$]		3 - 7		
12:	li R3$, 0		2		
13:	li R4$, 1		2		
14:	spawn R5$, 0, R1$, 3	$\sum^{AP}\{1 + 7*N_i\}$	1		$\sum^{CP}\{7\}$
15:	li R2$, 1		2		

16:	lwa R1$, 0(R7P$), DATA_SZ[R0$]		-2 - 2		
17:	bge R1$, R2P$, great_eqinit		3		
18:	ps R3P$, R2$		4		
19:	j endinit		5		
20:	great_eqinit:				
21:	ps R4P$, R2$		4		
22:	sub R2$, R1P$, R2$		5		
23:	endinit:				
24:	swa R1$, 0(R7P$), DATA_SZ[R2$]		6		
25:	join R5$, R1$		7		
26:	spawn R2$, 0, R2, 9	$\sum^{AP}\{2 + 7*2\}$	1		$\sum^{CP}\{5\}$
27:	beqz R0$, left		2		
28:	laa R7$, 0(R8P$), DATA_SZ[R3P$]		3		
29:	laa R8$, 0(R7P$), DATA_SZ[R3P$]		3		
30:	j endlr		3		
31:	left:				
32:	move R7$, R8P$		3		
33:	move R8$, R7P$		3		
34:	move R1$, R3P$		3		
35:	endlr:				
36:	slti R6$, R6P$, 1		3		
37:	jal recur, R9$		3		
38:	join R2$, R2		4		
39:	endsmallsort:				

40:	jr R9$		5		
41:	endquicksort:				
42:	join R3, R1	1	1		1

Total Work:

$W = \sum^{AP}\{7*N_i + 22\} + 4N + 9$

Making certain assumptions, and solving a recurrence equation (analysis omitted here), the following bound can be derived:

$W = 8.63*N*\log_2(N) + 22.6*N - 9$

Total Time:

$T = \sum^{CP}\{19\} + 9$

Making certain assumptions, and solving a recurrence equation (analysis omitted here), the following bound can be derived:

$T = 7.89*\log_2(N) + 9$