Performance Engineering of Software Systems 6.172
Massachusetts Institute of Technology Practice Quiz 1
Prof. Charles E. Leiserson and Prof. Julian Shun

Practice Quiz 1

Name: ______________________________

Instructions

- DO NOT open this quiz booklet until you are instructed to do so.
- This quiz booklet contains 14 pages, including this one. You have 80 minutes to earn 80 points.
- This quiz is closed book, but you may use one handwritten, double-sided 8 1/2″ × 11″ crib sheet and the Master Method card handed out in lecture.
- When the quiz begins, please write your name on this coversheet, and write your name on the top of each page, since the pages may be separated for grading.
- Some of the questions are true/false, and some are multiple choice. You need not explain these answers unless you wish to receive partial credit if your answer is wrong. For these kinds of questions, **incorrect answers will be penalized, so do not guess unless you are reasonably sure**.
- Good luck!

Number	Question	Parts	Points	Score	Grader
0	Name on Every Page	14	2		
1	True or False	8	16		
2	Bit Tricks for the Queens Problem	5	13		
3	Parallel PageRank	3	9		
4	Bitonic Sort	6	18		
5	Heap Allocation	3	13		
6	Compilers & Assembly	2	9		
	Total		80		

1 True or False (8 parts, 16 points)

Incorrect answers will be penalized, so do not guess unless you are reasonably sure. You need not justify your answer unless you want to leave open the possibility of receiving partial credit if your answer is wrong.

1.1

Packing is an optimization that reduces data movement but may increase computation.

True **False**

1.2

There can never be a true-, anti- or output-data dependence between the following two lines of code:

```
movl %eax, (%esi)
movl (%edi), %ecx
```

True **False**

1.3

The `time` command can more readily diagnose kernel-mode performance variations (e.g., page zeroing) than `clock_gettime()`.

True **False**

1.4

Using `taskset` for a serial program diminishes performance variations caused by NUMA (nonuniform memory access).

True **False**

1.5

When using reference counting for garbage collection, cyclic data structures are never garbage collected.

True **False**

1.6

In the free-list heap-allocation algorithm, allocating to the least-full page maximizes the probability that two random accesses hit the same page.

True **False**

1.7

Memory-allocator performance is more important when requesting a large amount of memory than when requesting a small amount.

True **False**

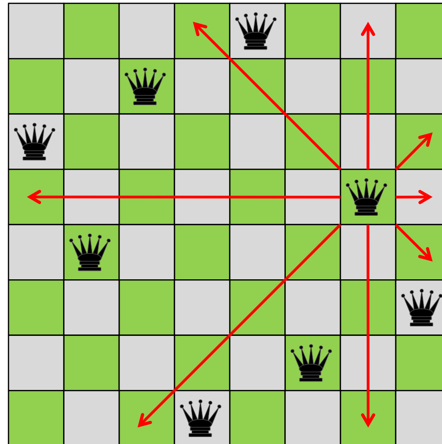
1.8

Eliminating common subexpressions generally improves the performance of code unless it causes too much register pressure.

True **False**

2 Bit Tricks for the Queens Problem (5 parts, 13 points)

Recall the ***Queens Problem*** from lecture: Place n queens on an $n \times n$ chessboard such that no two queens can attack each other, i.e., only one queen in placed each row, column, or diagonal:



We saw in lecture that this problem could be solved by a backtracking search that marches up and down the rows of the chessboard using bit tricks to represent columns and diagonals. In fact, the problem can be solved by an algorithm even more simple than the one Professor Leiserson presented in class.

The function `queens()` performs the backtrack search, returning the number of solutions to the queens problem. The four arguments to the function are (1) a bit mask `mask` representing the width of the board as n columns, (2) a bit mask `down` representing which columns contain queens, (3) a bit mask `left` representing which left-going diagonals ending on the current row contain queens, and (4) a bit mask `right` representing which right-going diagnoals ending on the current row contain queens.

The `queens()` function is called from `main()` as follows:

```
printf("The %d-queens problem has %d solutions.\n",
       n,
       queens((1 << n) - 1, 0, 0, 0));
```

The next page shows code for `queens()` with 5 blanks labeled with letters, as well as a collection of 20 expressions beneath.

```
int32_t queens( uint32_t mask,
                uint32_t down,
                uint32_t left,
                uint32_t right ) {
   int32_t count = 0;
   uint32_t possible, place;
   if (down == ______(A)______) return 1;
   for (possible = ~(down|left|right) & mask;
        possible != ______(B)______;
        possible &= ~place) {
       place = ______(C)______;
       count += queens( mask,
                        down|place,
                        ______(D)______,
                        ______(E)______ );
   }
   return count;
}
```

For each blank in the code, write its label next to the expression that best fits. (*Hint:* Some blanks can take more than one expression, but only one is "best.")

____	`0`	____	`place`
____	`-1`	____	`-place`
____	`down`	____	`~place`
____	`left`	____	`possible`
____	`left << 1`	____	`possible & (-possible)`
____	`(left|place) << 1`	____	`-possible`
____	`((left|place) << 1) & mask`	____	`right`
____	`mask`	____	`right >> 1`
____	`mask + 1`	____	`(right|place) >> 1`
____	`~mask`	____	`((right|place) >> 1) & mask`

3 Parallel PageRank (3 parts, 9 points)

In the following code, the variables `contribution` and `rank` are arrays of doubles, and `in_degree` and `out_degree` are arrays of integers. `neighbor` is a two dimensional array that stores the edges. neighbor[i][j] is the jth neighbor of the ith node.

```
1  void pagerank(double * rank, double * contribution,
2                int ** neighbor, int * in_degree,
3                int * out_degree, int num_vertices) {
4      for (size_t iter = 0; iter < 10; iter++) {
5          cilk_for (size_t i = 0; i < num_vertices; i++){

6              for (int j = 0; j < in_degree[i]; j++){
7                  rank[i] += contribution[neighbor[i][j]];
8              }
9          }
10         for (size_t i = 0; i < num_vertices; i++){
11             contribution[i] = rank[i]/out_degree[i];
12             rank[i] = 0.0;
13         }
14     }
15 }
```

For each of the following code modifications designed to improve performance, circle the appropriate option to specify whether it is safe to make the indicated change, whether it is safe if a reducer is used, or whether it is unsafe. *(Note: "safe" means that the output must be exactly the same as for the original code.)*

3.1

Replace the `for` in line 4 with `cilk_for`.

Safe **Safe with reducer** **Unsafe**

3.2

Replace the `for` in line 6 with `cilk_for`.

Safe **Safe with reducer** **Unsafe**

3.3

Replace the `for` in line 10 with `cilk_for`.

Safe **Safe with reducer** **Unsafe**

4 Bitonic Sort (6 parts, 18 points)

Consider the following multithreaded implementation of bitonic sort, a sorting algorithm based on ***bitonic*** sequences, which are sequences that can be cyclically shifted to be nonincreasing and then nondecreasing. Fortunately, for this problem you must understand neither bitonic sequences nor how the algorithm works. You must only understand its parallelism structure.

```
1  // Swap a[i] and a[j] if they are out of order, assuming that
2  // i < j and the boolean ascending indicates whether the
3  // sequence should be ascending (true) or descending (false)
4  void bitonic_swap(int *array, size_t i, size_t j, bool ascending) {
5    if ((array[i] > array[j]) == ascending) {
6      int temp = array[i];
7      array[i] = array[j];
8      array[j] = temp;
9    }
10 }
11
12 // Sort the elements in the subarray a[lo, .., hi-1] into
13 // ascending/descending order, assuming that the subarray forms
14 // a bitonic sequence of power-of-2 length.
15 void bitonic_merge(int *a, size_t lo, size_t hi, bool ascending) {
16   if (lo >= hi - 1) return;
17
18   size_t len = (hi - lo) / 2;
19   size_t mid = lo + len;
20   cilk_for (size_t i = lo; i < mid; i++) {
21     bitonic_swap(a, i, i + len, ascending);
22   }
23
24   cilk_spawn bitonic_merge(a, lo, mid, ascending);
25   bitonic_merge(a, mid, hi, ascending);
26   cilk_sync;
27 }
28
29 // Sort the elements in a[lo, .., hi-1] in ascending/descending order
30 // assuming that the length of the subarray is a power of 2.
31 void bitonic_sort(int *a, size_t lo, size_t hi, bool ascending) {
32   if (lo >= hi - 1) return;
33
34   size_t mid = (hi + lo) / 2;
35   cilk_spawn bitonic_sort(a, lo, mid, true);
36   bitonic_sort(a, mid, hi, false);
37   cilk_sync;
38
39   bitonic_merge(a, lo, hi, ascending);
40 }
```

For the following questions, let $n = \texttt{hi} - \texttt{lo}$, and assume that `n` is a power of 2.

4.1

Give a recurrence for the work $M_1(n)$ of `bitonic_merge()`, solve the recurrence, and express $M_1(n)$ in simple terms.

4.2

Give a recurrence for the span $M_\infty(n)$ of `bitonic_merge()`, solve the recurrence, and express $M_\infty(n)$ in simple terms.

4.3

Give the parallelism of `bitonic_merge()` in simple terms.

4.4

Give a recurrence for the work $S_1(n)$ of `bitonic_sort()` in terms of $M_1(n)$ and $M_\infty(n)$, solve the recurrence, and express $S_1(n)$ in simple terms.

4.5

Give a recurrence for the span $S_\infty(n)$ of `bitonic_sort()` in terms of $M_1(n)$ and $M_\infty(n)$, solve the recurrence, and express $S_\infty(n)$ in simple terms.

4.6

Give the parallelism of `bitonic_sort()` in simple terms.

5 Heap Allocation (3 parts, 13 points)

A certain program needs to allocate memory chunks that have alternating sizes of 60 bytes and 120 bytes. In other words, the program will allocate a chunk of 60 bytes, followed by 120 bytes, followed by 60 bytes, and so on. We shall consider two schemes for heap allocation.

Scheme F is a fixed-size allocator that uses a free-list of 120-byte blocks. Scheme V is a variable-sized allocator that uses binned free lists with blocks that are exact powers of 2.

5.1

What are likely the advantages of Scheme V over Scheme F? (Circle all that apply.)

A Faster allocation.

B Less internal fragmentation.

C Less external fragmentation.

D Fewer TLB (translation lookaside buffer) misses.

E Less false sharing when parallelized.

5.2

Ben Bitdiddle pushes an update to the program. Now, after 100,000 allocations of blocks with alternating sizes of 60 and 120 bytes (as described above), the program frees all blocks of size 60 bytes and proceeds to allocate 100,000 more blocks of size 120 bytes. After the update, which scheme would you prefer to use and why? (Circle all that apply.)

A Scheme V, because freeing blocks is faster with a variable-sized allocator.

B Scheme F, because there is less external fragmentation.

C Scheme F, because there is less internal fragmentation.

D Scheme F, because there is better space utilization and, therefore, fewer TLB (translation lookaside buffer) misses.

E Scheme V, because there is better space utilization and, therefore, fewer TLB (translation lookaside buffer) misses.

5.3

Lem E. Tweakit sends you a mystery Cilk program, in which each allocated object is labeled with the worker thread that created it (its owner), and freed objects are returned to the owner's heap. What is the bound on blowup B for this mystery Cilk program? (*Hint:* Blowup is the maximum of allocated storage across all workers divided by the maximum of allocated storage in the serial execution.)

A $B = 1$.

B $B = K$ for some constant $K > 1$.

C $B \leq P$, where $P =$ # workers.

D $P < B \leq KP$ for some constant $K > 1$.

E $B > KP$ for any constant $K > 1$.

6 Compilers & Assembly (2 parts, 9 points)

Consider the four code snippets below. Assume all four examples are compiled with Tapir/Clang on a 64-bit AVX2-enabled Linux machine, using the flags `-Rpass=loop-vectorize -mavx2` (the same conditions you used to complete Homework 3).

A

```
void func_A(int32_t * restrict X, int32_t * restrict Y) {
  for (int i = 0; i < 1000*1000; i++) {
    X[i] = X[i] + Y[i];
  }
}
```

B

```
void func_B(int32_t * restrict X, int32_t * restrict Y) {
  for (int i = 0; i < 1000*1000; i+=4) {
    X[i] = X[i] + Y[i];
  }
}
```

C

```
void func_C(int32_t * restrict X, int32_t * restrict Y) {
  for (int i = 0; i < 1000*1000; i++) {
    X[i] = X[i] / Y[i];
  }
}
```

D

```
void func_D(int32_t * restrict X, int32_t * restrict Y) {
  for (int i = 0; i < 1000*1000 - 1; i++) {
    X[i] = X[i+1] + Y[i];
  }
}
```

6.1

For which of the functions is the loop likely to be vectorized without further compiler directives? (Circle all that apply.)

A `func_A`

B `func_B`

C `func_C`

D `func_D`

6.2

The following text is the assembly code for one of the loops:

```
  movq   %rdi, -8(%rbp)   # X
  movq   %rsi, -16(%rbp)  # Y
  movl   $0, -20(%rbp)
.LBB0_1:
  cmpl   $1000000, -20(%rbp)
  jge    .LBB0_4
  movslq -20(%rbp), %rax
  movq   -8(%rbp), %rcx
  movl   (%rcx,%rax,4), %edx
  movslq -20(%rbp), %rax
  movq   -16(%rbp), %rcx
  addl   (%rcx,%rax,4), %edx
  movslq -20(%rbp), %rax
  movq   -8(%rbp), %rcx
  movl   %edx, (%rcx,%rax,4)
  movl   -20(%rbp), %eax
  addl   $4, %eax
  movl   %eax, -20(%rbp)
  jmp    .LBB0_1
.LBB0_4:
```

Circle the letter for the function containing the loop corresponding to the assembly code.

A `func_A`

B `func_B`

C `func_C`

D `func_D`

Intel x86 Assembly Language Cheat Sheet

Instruction	Effect	Example
Data movement		
mov src, dest	Copy src to dest	mov $10,%eax
Arithmetic		
add src, dest	Dest = dest + src	add $10, %esi
mul reg	edx:eax = eax * reg (colon means the result spans across two registers)	mul %esi
div reg idiv reg	edx = edx:eax mod reg eax = edx:eax / reg	div %edi
inc dest	Increment destination	Inc %eax
dec dest	Decrement destination	dec (%esi)
sbb arg1, arg2	If CF = 1, (this is set by cmp instruction; refer cmp) arg2 = arg2 – (arg1 + 1) else arg2 = arg2 – arg1	sbb %eax, %ebx
Function Calls		
call label	Push eip, transfer control	call _fib
ret	Pop eip and return	ret
push item	Push item (constant or register) to stack	pushl $32 pushl %eax
pop [reg]	Pop item from stack; optionally store to register	pop %eax popl
Bitwise Operations		
and src,dest	Dest = src & dest	and %ebx, %eax
or src, dest	Dest = src | dest	orl (0x2000), %eax
xor src, dest	Dest = src ^ dest	xor $0xffffff, %eax
shl count, dest	Dest = dest << count	shl $2, %eax
shr count, dest	Dest = dest >> count	shr $4, (%eax)
sal count, dest	Same as shl, shifted bits will be the sign bit	
Conditionals and jumps		
cmp arg1, arg2	If arg1 > arg2 sets CF=1 (carry flag =1) This compares arg1 and arg2; you can use any conditionals jumps below to act upon the result of this comparison	cmp $0, %eax
test reg,imm/reg	Bitwise and of register and constant/register; the next jump command uses the result of this; consider this essentially as same as compare	test %rax, %rcx
je label	Jump to label if arg2 = arg1	je endloop
jne label	Jump to label if arg2 != arg1	jne loopstart
jg label / ja label	Jump to label if arg2 > arg1	jg exit / ja exit
jge label	Jump to label if arg2 >= arg1	jge format_disk
jl label	Jump to label if arg2 < arg1	jl error
jle label	Jump to label if arg2 <= arg1	jle finish
jz label	Jump to label if bits were not set	jz looparound
jnz label	Jump to label if bits were set	jnz error
jump label	Unconditional jump	jmp exit
Miscellaneous		
nop	No-op	nop
lea addr, dest	Move the address calculated to the dest	lea 23(%eax, %ecx,8),%eax
cqto	%rdx:%rax← sign-extend of %rax.	cqto

suffixes b=byte(8), w=word(16), l=long(32), q=quad(64)
base indexed scale displacement 172(%rdi, %rdx,8) = %rdi + 8 * %rdx + 172
Note that not both src and dest can be memory operands at the same time.
register - %eax   fixed address – (0x1000)
constant - $10   dynamic address – (%rsi)

MIT OpenCourseWare
https://ocw.mit.edu

6.172 Performance Engineering of Software Systems
Fall 2018

For information about citing these materials or our Terms of Use, visit: https://ocw.mit.edu/terms