ECE 454 – Computer Systems Programming

The Edward S. Rogers Sr. Department of Electrical and Computer Engineering

Final Examination Fall 2011

Name	
Student #	

Professor Greg Steffan

Answer all questions. Write your answers on the exam paper. Show your work.
Each question has a different assigned value, as indicated.

The exam is open book open notes (only simple calculators allowed, no cell phones or PDAs)

Total time available: **150 minutes**

Total marks available: **145** (roughly one mark per minute, with 5 extra minutes)

Verify that your exam has all the pages.

Part	Points	Mark
1	30	
2	20	
3	10	
4	10	
5	10	
6	20	
7	10	
8	15	
9	20	
Total	145	

PART 1) [30] Short Answer

a) Modern multicores often have on-chip L2 caches as well as L1 caches. It is common to implement something called the "inclusion property", which means that if a certain cache block is evicted from the L2 cache (eg., due to a conflict) then it also must be evicted from the L1 cache. In one sentence, why would this property be useful?

b) Suppose that you are planning to parallelize the following loop, where N is very large. However, you are considering first unrolling the loop four times (each new iteration would contain the work of four original iterations).

```
for (i=0;i<N;i++){
    x = work(i);
}
```

i) what are the properties of an instance of work() for which unrolling would likely improve parallel performance (compared to not unrolling), and why?

ii) what are the properties of an instance work() for which unrolling would likely reduce parallel performance (compared to not unrolling), and why?

c) You have created an implementation of an MCS lock: a linked-list-based queueing lock, where the lock() operation causes a thread to enqueue a lock_entry structure to the end of the lock-queue, and then spin on that lock_entry. The unlock() operation dequeues & frees its lock_entry and notifies the next lock_entry in the queue.

i) Your lock_entry struct contains a lock variable, pointers, and a few fields for tracking statistics. By reordering/optimizing the struct declaration you are able to reduce the size of each lock_entry structure from 64B to 32B. However, you find that performance of the reduced-size version is worse! How can this be the case?

ii) Assuming you keep the 64B version of lock_entry, should you consider adding "back-off" to the spin-wait implementation for the lock() routine? Why or why not?

d) Thread1 executes code that stores 5 to location X then loads X. In parallel thread2 executes code that stores 7 to X. There is no synchronization in either code. When Thread1 loads X it might get the value 7 or might get the value 5. This is because of the processor's implementation of coherence or consistency?

e) It is possible to implement MPI on top of a shared memory multiprocessor or multicore. Suppose you were to implement MPI using pthreads, i.e., using the following pthreads functions as described in class: `pthread_create()`, `pthread_join()`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `pthread_cond_signal()`, `pthread_cond_wait()`, `semaphore_signal()`, `semaphore_wait()`.

i) which of the above pthreads functions would you use in your code to implement a non-blocking MPI_Send() and briefly why?

ii) which of the above pthreads functions would you use in your code to implement a blocking MPI_Recv() and briefly why?

f) Comparing transactional memory and locks

i) In one sentence explain how Transactional Memory is similar to coarse-grain locking.

ii) In one sentence explain how Transactional Memory is similar to fine-grain locking.

iii) In one sentence explain the behavior of a critical section (executed by many parallel threads) within a loop where it would be better for performance to protect it with a regular lock than with a Transactional Memory transaction.

g) Altera visit question:

Some algorithms can produce different solutions to the same problem depending on how long they are allowed to execute. When parallelizing such an algorithm, what do you have to be careful of when deciding whether the parallel version is an improvement?

PART 2) [20] Cache Optimization

Optimize the cache performance of the following code. Do not parallelize it. Do not consider prefetching nor TLB behavior. You can assume that none of the operations overflow. Assume that N and M are extremely large powers of 2. Assume that A[], B[], andC[] are integer arrays, and that an int is 4B.

i)

```
for (i=0;i<N;i++)
  for (j=0;j<M;j++)
    A[i] += B[j][j] * C[i];
```

ii)

```
idx = 0;
for (i=0;i<N;i++)
  for (j=0;j<M;j++)
    if (A[i] == B[j])
      C[idx++] = hash(i,j);
```

PART 3) [10] Performance Measurement

Your code goes 1.4x faster after applying 4-thread perfect parallelization to loop1 and 8-wide perfect "SIMDization" (eg., exploiting SSE instructions) to loop2. Knowing that loop1 is 20% of the original (pre-optimization) execution time, what percentage of the original (pre-optimization) program execution time is loop2? You can assume that loop1 and loop2 are not nested.

PART 4) [10] Prefetching

Rewrite the following code with prefetching inserted for the array `D`. Use the function `pf(int *addr)` to make a prefetch (eg., `pf(&(x))` would prefetch x). Assume that a prefetch takes as long as 32 cache hits (ie., 32 memory accesses that hit in the cache can "hide" the latency of a prefetch). Assume that `D[]` is an array of pointers to arrays, that the cache block size is 32bytes, and that an int is 4bytes. Assume that only accesses to the array generate cache accesses (i.e., ignore accesses to i, j, sum). You should only issue one prefetch for a given cache block.

```
for (i=0;i<N;i++){ // loop1
  for (j=0;j<N;j++){ // loop2
    sum += D[i][j];
  }
}
```

PART 5) [10] TLBs

Suppose your C-code reads and updates a single 64x64 array of 128B structs repeatedly, visiting all of the elements of all of the rows in order. Assuming a 128-entry TLB and 4KB pages, would tiling this code into 4 tiles reduce TLB misses? If so by what fraction? If not, show your calculations as to why not.

PART 6) [20] Dynamic Memory Allocation

Consider an allocator with the following:

- Uses a single implicit free list.
- All memory blocks have a size that is a multiple of 4 bytes and is at least 8 bytes.
- All headers, footers, and pointers are 4 bytes in size
- Allocated blocks consist of a header and a payload (no footer)
- Free blocks consist of a header and a footer at the end of the block.
- All freed blocks are immediately coalesced if possible.
- The heap starts with 0 bytes, never shrinks, and only grows large enough to satisfy memory requests.
- The heap contains only allocated blocks and free blocks. There is no space used for other data or special blocks to mark the beginning and end of the heap.
- When a block is split, the lower part of the block becomes the allocated part and the upper part becomes the new free block (and is coalesced if possible).
- The **best-fit** free block is used when searching the free list.
- If no free block is large enough then the heap is extended only enough to fulfill the request, including using any remaining free block at the end of the free list.

Below you are given a series of memory requests. You are asked to show what the heap looks like after each request is completed using the best-fit placement policy. The heap is represented as a series of boxes, where each box is a single block on the heap, and the bottom of the heap is the left most box. In each block, you should write the total size (including headers and footers) of the block in bytes and either 'f' or 'a' to mark it as free or allocated, respectively. For example, the following heap contains an allocated block of size 16, followed by a free block of size 32.

16a	32f

Assume that the heap is empty before each of the sequences is run, with a single 400B free block (shown for you). You do not necessarily have to use all of the columns provided for the heap. There are three copies here in case you make a mess. Clearly cross out the ones you don't want graded.

COPY #1: (they are identical, cross out the ones you don't want graded)

	400f						
ptr1 = malloc(8)							
ptr2 = malloc(31)							
free(ptr1)							
ptr3 = malloc(29)							
ptr4 = malloc(4)							
ptr5 = malloc(312)							
free(ptr4)							
ptr6 = malloc(9)							
ptr7 = malloc(4)							
free(ptr3)							
free(ptr2)							
free(ptr6)							

COPY #2: (they are identical, cross out the ones you don't want graded)

	400f						
ptr1 = malloc(8)							
ptr2 = malloc(31)							
free(ptr1)							
ptr3 = malloc(29)							
ptr4 = malloc(4)							
ptr5 = malloc(312)							
free(ptr4)							
ptr6 = malloc(9)							
ptr7 = malloc(4)							
free(ptr3)							
free(ptr2)							
free(ptr6)							

COPY #3: (they are identical, cross out the ones you don't want graded)

	400f						
ptr1 = malloc(8)							
ptr2 = malloc(31)							
free(ptr1)							
ptr3 = malloc(29)							
ptr4 = malloc(4)							
ptr5 = malloc(312)							
free(ptr4)							
ptr6 = malloc(9)							
ptr7 = malloc(4)							
free(ptr3)							
free(ptr2)							
free(ptr6)							

PART 7) [10] Cache Coherence

Assuming a 4-CPU multicore with MESI invalidation-based cache coherence with write-allocate write-back caches, and locations X and Y that are in the same cache block, in the table below are shown the order in time of loads and stores to X and Y. Put a '1' or a checkmark in the column on the left next to each row for which a coherence message occurs that requests a copy of the cache block contents for the corresponding CPU's cache. HINT: a load-miss results in a copy request, while a write-hit does not.

Copy of cache block requested?	CPU 0	CPU 1	CPU 2	CPU 3
		Store X		
		Load X		
			Store Y	
			Load X	
				Load X
	Load X			
		Load X		
			Store Y	
		Store X		
			Load Y	
		Load X		
		Load Y		
				Load X
				Store Y

PART 8) [15] Dependence Analysis

For each loop in the following code, state whether or not it is parallel, and if not describe the dependence(s) that prevent it from being parallel (give the type of dependence and which terms c1,c2,c3,c4 are involved).

```
for (i=2;i<N-1;i++){ // loop1
  for (j=3;j<N;j++){ // loop2
    for (k=4;k<N-3;k++){ // loop3
      A[i][j][k] =                      // c1
                   A[i-2][j-3][k-4] + // c2
                   A[i+1][j][k+2] +   // c3
                   A[i][j][k-1];      // c4
    }
  }
}
```

PART 9) [20] OpenMP Parallelization

Parallelize the following code using openMP pragmas. Assume that the target machine has a cache block size of 128B, and that the size of an int is 4B. Be sure to explicitly specify the "schedule" options that should be used, even if you want to use the default options. For each please rewrite the code. If necessary you can assume that the variable P represents the number of processors to be used. Assume that N is large (in the tens of thousands or more), and that code that is not shown has initialized all arrays/variables to useful values. You must explicitly list all variables within the range of a parallel pragma that are private using the private() directive. Do not do tiling. Please rewrite the new code.

a)

```
for (i=0;i<N;i++){
    for (j=0;j<N;j++){
        A[i,j] = max(A[i,j],B[i,j]);
    }
}
```

b)

```
C[0] = 1;
for (i=1;i<N;i++){
    C[i] = C[i-1];
    for (j=0;j<N;j++){
        C[i] *= A[i,j] + B[i,j];
    }
}
```

c) (rewrite only the for loop)

```
typedef struct element
{
  int value;
  struct element *next;
} Element;

Element *D[N]; // D[] is an array of pointers to linked lists
of varying length
int C[N];
...
for (i=0;i<N;i++){
    C[i] = computeAverageValueOfAllListElements(D[i]);
}
```

d)

```
// Assume the same D[] array from part (c)
Element *pointer_to_max = NULL;
for (i=0;i<N;i++){
    Element *tmpPtr = findListElementWithMaxValue(D[i]);
    if (!pointer_to_max ||
        (tmpPtr && tmpPtr->value > pointer_to_max->value))
       pointer_to_max = tmpPtr;
}
```