ECE 459 W18 Final Exam Solutions

(1)

(a) You need to know (1) how big an improvement we could get by speculating, (2) how likely we are to be right, (3) how expensive is the penalty if we are wrong.

(b) Let's do the math then. Formula from Amdahl's Law: $1/(S + P/N)$

More CPUs: $1/(.25 + .75/8) = 2.909; 1/(.25 + .75/10) = 3.0769.$

Dev: $1/(0.23 + 0.77/8) = 3.065$

Conclusion: more CPUs!

However, you could have also interpreted the question like this: Dev improvement = $0.98\times$ original. More CPUs: multiplying out the parallel 0.75 of the original time per-CPU by 8 CPUs we have $0.75 \times 8 = 600$ units of work. Doing those 600 units of work on 10 CPUs would instead take time $0.6\times$ original, plus the original $0.25\times$ for the sequential part. So that yields $0.85\times$ original. Like the other math, more CPUs is still faster.

(c) No, it is not wait-free. An unlucky thread may have to execute the compare and swap operation many many times before it succeeds, and in the worst case scenario, it waits indefinitely.

(It is, however, lock-free, which is not the same thing.)

(d) Inlining can improve performance by reducing function call overhead and allowing the compiler to make decisions about variable storage and aliasing that might otherwise be undecidable. However, the cost is larger executable size, more cache misses, reduced ability to debug. Cache is a mixed bag though. Inlining can both help and harm caches: if a function is called in two contexts and inlining disambiguates them, inlining helps.

(Must get at least 1 specific advantage, 1 disadvantage, and a 3rd distinct point that can be either advantage or disadvantage).

(e) Use system profiling tools to determine where your bottlenecks are. These may include things like looking at CPU usage, disk usage, etc. You then choose the kind of instance based on whatever your limiting factor is.

(f) It is difficult and requires many messages to coordinate the order of transactions in a distributed database. It is therefore faster to send all writes to the Write Master and replicate its state on the replicas. (Even with reader/writer locks you still need to do writes when acquiring the RW lock in read mode, so that others know that you have the lock. Read replicas are truly write-free to read.)

(g) The compiler has to examine the content of the loop and estimate how long an iteration will run and how many iterations exist; this has to be then compared against an estimate of the size of the overhead (initialization and recombination) of the parallel region is. If the body of the loop time is less than the overhead, declare it not profitable.

(h) In the code on the right, the whole for loop is executed ordered and parallelization basically does nothing. In the code on the left, the function `do_something_useful` can be executed in parallel during different iterations of the loop; only the print statement has to be done ordered.

(i)

```
void * foo_thr( void * arg ) {
  struct bar* b = malloc(sizeof( struct bar ));
  pthread_cleanup_push( foo_handler, b );
  /* Initialize b and do something with it */
  pthread_cleanup_pop( 0 );
  pthread_exit( b );
}
```

```
void foo_handler( void * ptr ) {
        free( ptr );
}
```

(j) Yes, rematerialization makes sense sometimes! A read from memory is expensive (200-300 cycles) and an arithmetic CPU operation is very quick (single digit number of cycles, most likely). Thus, rather than using up a register for this value, we might just calculate it more than once.

(For the reader's context and not for marks: this is sometimes done to reduce the number of registers that are needed by the program. There are relatively few, even in modern CPUs. If we had to load and store more things we would lose time...)

(2.1) The simplest t2 implementation is:

```
store 2 to y
load x into register r2
```

Under a weak memory model, we can reorder the statements in each thread since they're independent.

1. t1 loads 0 into r1
2. t2 loads 0 into r2
3. t1 stores 1 to x
4. t2 stores 2 to y

final state: r1 = 0, r2 = 0, x = 1, y = 2

Now we need to establish that this is not an allowed state under sequential consistency. Write down the allowed orders and just enough of the result to see that the reordering gives something different.

t1 store, t2 store, t1 load, t2 load $\rightarrow$ r1 = 2
t1 store, t2 store, t2 load, 1 load $\rightarrow$ r1 = 2
t1 store, t1 load, t2 store, t2 load $\rightarrow$ r1 = 0, r2 = 1
t2 store, t1 store, t2 load, t1 load $\rightarrow$ r1 = 2
t2 store, t1 store, t1 load, t2 load $\rightarrow$ r1 = 2
t2 store, t2 load, t1 store, t1 load $\rightarrow$ r1 = 2, r2 = 0

We can see that this reordering is prohibited under sequential consistency.

(2.2) We're looking for a little drawing with steps that shows the progression of the problem. It should have an initial state for `a` or `b` that is changed by the assignment statement because array `c` overlaps one of those.

A super simple example involves all arrays being of length 2. `a`'s element 1 is `c`'s element 0. Let's imagine that `a` contains `42, 5`. Array `b` is `2, 2`. Then the assignment statement `c[0] = a[0] + b[0]` overwrites `a[1]` meaning the final state of `c` is not `44, 7`, but instead `44, 46`.

(2.3)

	CPU 0		CPU 1	
Step	**x**	**y**	**x**	**y**
0.	I	I	I	I
1.	I	E	I	I
2.	I	E	M	I
3.	I	M	M	I
4.	I	S	M	S
5.	I	S	M	S

(2.4) Each answer should be a super common scenario, that is, a thing that happens very frequently in web browsers. The answer should also have an appropriate justification. Some simple possible answers:

- Simply visiting generally popular websites (google, facebook, amazon, etc) as these are things people do a lot.
- Playing a video (HTML5) on Netflix or YouTube or similar.
- Running the Javascript execution engine (e.g., visiting webpages where there is a lot of Javascript content like Google Docs)

Bad (wrong) examples involve doing unusual things (looking in the dev console, changing settings, printing...)

(3)

```
CURLM* cm;
pthread_mutex_t lock;
double total = 0.0;

size_t callback(char *d, size_t n, size_t l, void *p) {
   double parsed = parse_response( d, n*l );
   pthread_mutex_lock( &lock );
   total += parsed;
   pthread_mutex_unlock( &lock );
   return n*l;
}

int main( int argc, char** argv ) {
    curl_global_init( CURL_GLOBAL_ALL );
    cm = curl_multi_init( );
    pthread_mutex_init( &lock, NULL );

    course_term* next = get_next();
    while( next != NULL ) {
        CURL* eh = convert( next );
        curl_multi_add_handle( cm, eh );
        free( next );
        next = get_next( );
    }

    int still_running = 0;
    curl_multi_perform( cm, &still_running );
    do {
       wait_for_curl( cm );
       curl_multi_perform( cm, &still_running );
    } while( still_running );

    int msgs_left = 0;
    CURLMsg *msg = NULL;
    while ( ( msg = curl_multi_info_read( cm, &msgs_left ) ) ) {
        if ( msg->msg == CURLMSG_DONE ) {
            CURL* eh = msg->easy_handle;

            CURLcode return_code = msg->data.result;
            if ( return_code != CURLE_OK ) {
                fprintf( stderr, "CURL␣error␣code:␣%d\n", msg->data.result );
                continue;
            }
            curl_multi_remove_handle( cm, eh );
            curl_easy_cleanup( eh );
        } else {
            fprintf( stderr, "error:␣after␣curl_multi_info_read(),␣CURLMsg=%d\n", msg->msg );
        }
    }

    curl_multi_cleanup( cm );
    curl_global_cleanup();
    pthread_mutex_destroy( &lock );
    printf( "Total␣AUs:␣%g\n.", total ); /* Can be anywhere after all curl reqs done */

    return 0;
}
```

(4) No changes to the struct are required, but you could implement this differently.

```
void itemcache_init( itemcache *c, int cache_size ) {
  c->replace_index = 0;
  c->max_size = cache_size;
  c->array = malloc( cache_size * sizeof( item ) );
  pthread_rwlock_init( &c->lock, NULL );
}

void itemcache_destroy( itemcache *c ) {
  free( c->array );
  pthread_rwlock_destroy( &c->lock );
}

item* itemcache_search( itemcache *c, unsigned int id ) {
  item* result = NULL;
  pthread_rwlock_rdlock( &( c->lock ) );
  for ( int i = 0; i < c->max_size; ++i ) {
    if ( c->array[i].id == id ) {
        result = copy( &c->array[i] );
    }
  }
  pthread_rwlock_unlock( &( c->lock ) );
  return result;
}
/* In put you _could_ search for empty spaces and use those, but you don't have to */
void itemcache_put( itemcache *c, item *i ) {
  int r = -1;
  pthread_rwlock_wrlock( &( c->lock ) );
  for ( int i = 0; i < c->max_size; ++i ) {
    if ( c->array[i].id == id ) {
      r = i;
      break;
    }
  }
  if ( r == -1 ) {
    replace = c->replace_index;
    (c->replace_index)++;
  }
  c->array[r] = *i;
  pthread_rwlock_unlock( &( c->lock ) );
}

void itemcache_invalidate( itemcache *c, unsigned int id ) {
  pthread_rwlock_wrlock( &( c->lock ) );
  for ( int i = 0; i < c->max_size; ++i ) {
    if ( c->array[i].id == id ) {
      memset( &c->array[i], 0, sizeof( item ) );
    }
  }
  pthread_rwlock_unlock( &( c->lock ) );
}

void itemcache_clear( itemcache *c ) {
  pthread_rwlock_wrlock( &( c->lock ) );
  memset( c->array, 0, max_size * sizeof( item ) );
  pthread_rwlock_unlock( &( c->lock ) );
}
```

(5.1) (a) returns stack-allocated `newStorage`.

(b)

```
fun deepCopy(other: &mut Box);
```

This parameter is owned by someone else (and is mutably borrowed by this function, but I didn't ask that).

(c) Freed when new storage is put into the field. That field owns the storage, and gets dropped when the object containing the field gets dropped.

(5.2)

Based on the given information, on average 50 jobs arrive per second and service time is 10 ms. So $\lambda = 50$ and $s = 0.01$. $\rho = \lambda s/N$ and for the last two, we consider what happens when $\rho = 1$.

Property	**Value**
Kendall Notation Description	M/M/2
Utilization (ρ)	0.25
Intermediate Value (K)	(1.5/1.625) = 0.923 [2 marks]
Completion time average (T_q)	0.101 [2 marks]
Average queue length (W)	0.033 [2 marks]
Maximum arrivals/hour the system can handle (full utilization)	720 000 (200 per second)
Maximum average service time the system can tolerate	40 ms

(6)

```
cl::Kernel kernelAdd(program, "add");
cl::Kernel kernelMax(program, "max_index");
cl::Kernel kernelReduce(program, "reduce");

// Create memory buffers
cl::Buffer bufferA = cl::Buffer(
   context,
    CL_MEM_READ_ONLY,
    SIZE * SIZE * 3 * 3 * sizeof(cl_float)
);
cl::Buffer bufferP = cl::Buffer(
    context,
    CL_MEM_READ_ONLY,
    SIZE * SIZE * sizeof(cl_float)
);
cl::Buffer bufferSums = cl::Buffer(
    context,
    CL_MEM_READ_WRITE,
    SIZE * SIZE * sizeof(cl_float)
);
cl::Buffer bufferGmax = cl::Buffer(
    context,
    CL_MEM_READ_WRITE,
    SIZE * SIZE * sizeof(cl_float)
);
cl::Buffer bufferLmax = cl::Buffer(
    context,
    CL_MEM_READ_WRITE,
    SIZE * SIZE * sizeof(cl_float)
);

// Copy lists A and B to the memory buffers
queue.enqueueWriteBuffer(
    bufferA,
    CL_TRUE,
    0,
    SIZE * SIZE * 3 * 3 * sizeof(cl_float),
    A
);
queue.enqueueWriteBuffer(
    bufferP,
    CL_TRUE,
    0,
    SIZE * SIZE * sizeof(cl_float),
    P
);

// Set arguments to kernel
kernelAdd.setArg(0, bufferP);
kernelAdd.setArg(1, bufferA);
kernelAdd.setArg(2, bufferSums);

kernelMax.setArg(0, bufferSums);
kernelMax.setArg(1, bufferGmax);
kernelMax.setArg(2, bufferLmax);

kernelReduce.setArg(0, bufferGmax);
```

```
      // Run the kernel on specific ND range
      cl::NDRange global((SIZE+2)*(SIZE+2));
      cl::NDRange local(1);
      queue.enqueueNDRangeKernel(
          kernelAdd,
          cl::NullRange,
          global,
          local
      );

      queue.enqueueNDRangeKernel(
          kernelMax,
          cl::NullRange,
          global,
          local
      );

      queue.enqueueNDRangeKernel(
          kernelReduce,
          cl::NullRange,
          global,
          local
      );

      // Read buffer C into a local list
      queue.enqueueReadBuffer(
           bufferGmax,
           CL_TRUE,
           0,
           (SIZE+2)*(SIZE+2) * sizeof(cl_float4),
           gmax
      );

      // gmax[0].w contains max index

#define twoD(x,y) (x*(SIZE+2)+y)
#define fourD(x1,y1,x2,y2) (((x*(SIZE+2)+y)*2+x2)*2+y2)

__kernel void add(__global float *P, __global float *A, __global float *sums) {
  uint x = get_global_id(0);
  uint y = get_global_id(1);
  sums[twoD(x,y)] = P[twoD(x,y)] + A[fourD(x,y,1,0)*P[twoD(x,y-1)]
                                 + A[fourD(x,y,0,1)*P[twoD(x-1,y)]
                                 + A[fourD(x,y,1,2)*P[twoD(x,y+1)]
                                 + A[fourD(x,y,2,1)*P[twoD(x+1,y)];
}

// I guess I should have said that you can assume everything is divisible.
__kernel void max_index(__global float *input, __global float4 *gmax, __local float4 *lmax) {
    uint count = (SIZE+2)*(SIZE+2) / get_global_size(0);
    uint idx = get_global_id(0);
    uint stride = get_global_size(0);
    cl_float4 pmax; pmax.x = input[idx]; pmax.w = idx;
    for (int i = 0; n < count; n++, idx += stride) {
      if (pmax.x < input[idx]) {
        pmax.x = index[idx]; pmax.w = idx;
      }
    }
```

```
    if (get_local_id(0) == 0)
      lmax = (uint) UINT_MAX;
    barrier(CLK_LOCAL_MEM_FENCE);
    if (pmax.x > lmax.x) {
      pmax.x = lmax.x; pmax.w = lmax.w;
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    if (get_local_id(0) == 0)
      gmax[get_group_id(0)] = lmax[0];
}

__kernel void reduce(__global float4 *gmax) {
    if (gmax[get_group_id(0)].x > gmax[0].x) {
      gmax[0].x = gmax[get_group_id(0)].x;
      gmax[0].w = gmax[get_group_id(0)].w;
    }
}
```