Stanford CS149: Parallel Programming
Written Assignment 1

Miscellaneous Short Answer Questions

Problem 1. (24 points):

A. (6 pts) Imagine you are asked to implement ISPC, and your system must run a program that launches 1000 ISPC tasks. Give one reason why it is very likely more efficient to use a fixed-size pool of worker threads rather than create a pthread per task. Also specify how many pthreads you'd use in your worker pool when running on a quad-core, hyper-threaded Intel processor. Why?

B. (6 pts) Assume you want to efficiently run a program with very high temporal locality (lots of reuse to the *same* pieces of data). If you could only choose one, would you add a data cache to your processor or add a significant amount of hardware multi-threading? Why?

C. (6 pts) Complete the ISPC code below to write an if-than-else statement that causes an 8-wide SIMD processor to run at nearly 1/8th its peak rate. (Assume the ISPC gang size is 8. Pseudocode for an answer is fine.)

```
void my_ispc_func() {

   int i = programIndex;

}
```

D. (6 pts) OpenMP is a nifty set of C/C++ programming extensions that makes it easy to define parallel execution on C loops. For example in the code below `#pragma omp parallel for` tells the C compiler to treat this loop like my fictitious `for_each` or `for_all` loops I often referred to in class—in other words it tells the OpenMP capable C compiler that the loop iterations are independent and can be parallelized. The *implementation* of OpenMP will distribute iterations of this loop onto *different software threads running on the different cores of a multi-core machine*. Note this is different from the implementation of ISPC `for_each` which distributes loop iterations onto program instances mapped to SIMD vector lanes.

Consider the following OpenMP program running on a 4-core processor with infinite bandwidth and 0 memory latency. (Assume memory load and store operations are "free".)

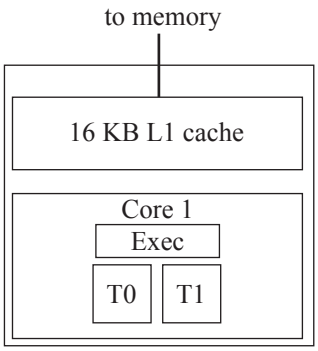
```
float total = 0.0;

#pragma omp parallel for
for (int i=0; i<N; i++) {  // assume N is very, very large
  if (i % 16 < 8)          // assume 0 ops
    out[i] = 1 * in[i];    // 1 op
  else
    out[i] = 2 + in[i];    // 1 op
}

for (int i=0; i<N; i++)
  total += out[i];         // 1 op
```

What speedup will this program realize on a 4 core machine? Careful, make sure you are thinking about differences between multi-core parallelism and SIMD parallelism. Also think about which parts of this program are parallelized and which are not.

A Task Queue on a Multi-Core, Multi-Threaded CPU

Problem 2. (30 points):

The figure below shows a simple single-core CPU with an 16 KB L1 cache and execution contexts for up to two threads of control. Core 1 executes threads assigned to contexts T0-T1 in an interleaved fashion by switching the active thread only on a memory stall); **Memory bandwidth is infinitely high in this system, but memory latency is 125 clocks. To keep things simple, assume a cache hit is only 1 cycle, a cache line is 4 bytes, and the cache implements a least-recently used (LRU) replacement policy—meaning that when a cache line is evicted, the line that was last accessed the furthest in the past is evicted. It may be helpful to think about how this cache behaves when a program reads 17 KB contiguous bytes of memory over and over. Hint: every load is a miss.**



You are implementing a task queue for a system with this CPU. The task queue is responsible for executing large batches of independent tasks that are created as a part of a bulk launch (much like how an ISPC task `launch` creates many independent tasks). You implement your task system using a pool of worker threads, all of which are spawned at program launch. When tasks are added to the task queue, the worker threads grab the next task in the queue by atomically incrementing a shared counter `next_task_id`. Pseudocode for the execution of a worker thread is shown below.

```
mutex queue_lock;
int   next_task_id;              // set to zero at time of bulk task launch
int   total_tasks;               // set to total number of tasks at time of bulk task launch
int*  task_args[MAX_NUM_TASKS];  // initialized elsewhere

while (1) {

   int my_task_id;

   LOCK(queue_lock);
   my_task_id = next_task_id++;
   UNLOCK(queue_lock);

   if (my_task_id < total_tasks)
       TASK_A(my_task_id, task_args[my_task_id]);
   else
       break;
}
```

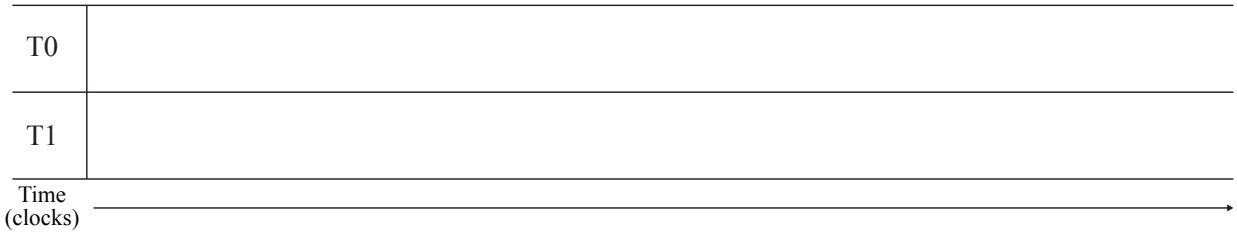
A. (6 pts) Consider one possible implementation of TASK_A from the code on the previous page:

```
function TASK_A(int task_id, int* X) {
   for (int i=0; i<1000; i++) {
      for (int j=0; j<8192; j++) {
         load X[j]   // assume this is a cold miss when i=0
         // ... 25 non-memory instructions using X
      }
   }
}
```

The inner loop of TASK_A scans over 32 KB of elements of array X, performing 25 arithmetic instructions after each load. This process is repeated over the same data 1000 times. **Assume there are no other significant memory instructions in the program and that each task works on a completely different input array X (there is no sharing of data across tasks). Remember the cache is 16 KB, a cache line is 4 bytes, and the cache implements a LRU replacement policy. Assume the CPU performs no prefetching.**

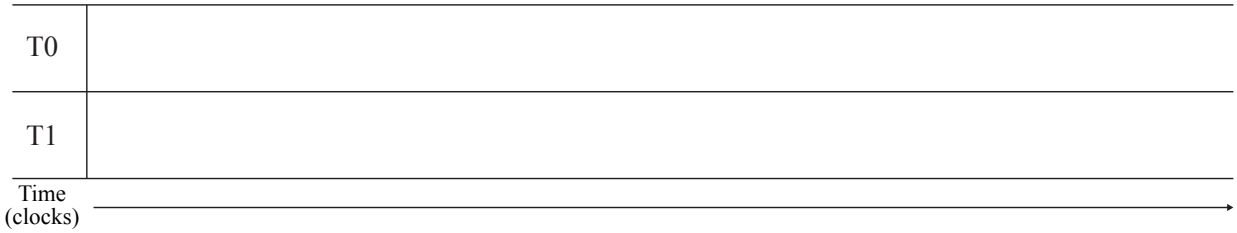
In order to process a bulk launch of TASK_A, you create two worker threads, WT0 and WT1, and assign them to CPU execution contexts T0 and T1. Do you expect the program to execute *substantially faster* using the two-thread worker pool than if only one worker thread was used? If so, please calculate how much faster. (Your answer need not be exact, a back-of-the envelop calculation is fine.) If not, explain why.

(Careful: please consider the program's execution behavior on average over the entire program's execution ("steady state" behavior). Past students have been tricked by only thinking about the behavior of the first loop iteration of the first task.) It may be helpful to draw when threads are running and stalled waiting for a load on the diagram below.



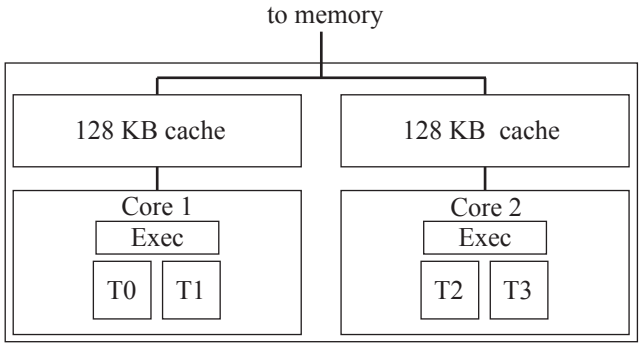
B. (6 pts) **Now consider the case where the program is modified to contain 10,000 instructions in the innermost loop.** Do you expect your two-thread worker pool to execute the program *substantially faster* than a one thread pool? If so, please calculate how much faster (your answer need not be exact, a back-of-the envelop calculation is fine). If not, explain why.

C. (6 pts) **Now consider the case where the cache size is changed to 128 KB and you are running the original program from Part A (25 math instructions in the inner loop).** When running the program from part A on this new machine, do you expect your two-thread worker pool to execute the program *substantially faster* than a one thread pool? If so, please calculate how much faster (your answer need not be exact, a back-of-the envelop calculation is fine). If not, explain why.



D. (6 pts) **Now consider the case where the L1 cache size is changed to 48 KB.** Assuming you cannot change the implementation of `TASK_A` from Part A, would you choose to use a worker thread pool of one or two threads? Why does this improve performance and how much higher throughput does your solution achieve?

E. (6 pts) Now consider the case where the task system is running programs on a dual-core processor. Each core is two-way multi-threaded, so there are a total of four execution contexts (T0-T3). Each core has a 128 KB cache.



If you maintain your two-worker thread implementation of the task system as discussed in prior questions, to which execution contexts do you assign the two worker threads WT0 and WT1? Why? Given your assignment, how much better performance do you expect than if your worker pool contained only one thread?

Because The Professor with the Most ALUs (Sometimes) Wins

Problem 3. (32 points):

Consider the following ISPC code that computes $ax^2 + bx + c$ for elements x of an entire input array.

```
void polynomial(float a, float b, float c,
                uniform float x[], uniform float output[], int elementsPerTask) {
  uniform int start = taskIndex * elementsPerTask;
  uniform int end = start + elementsPerTask;

  foreach (i = start ... end) {
    output[i] = (a * x[i] * x[i]) + (b * x[i]) + c;   // 5 arithmetic ops
  }
}

// assume N is very, very large, and is a multiple of 1024
void run(int N, float a, float b, float c, float* input, float* output) {
  uniform int elementsPerTask = 1024;
  launch[N/elementsPerTask] polynomial(a, b, c, input, output, elementsPerTask);
}
```

Professor Kayvon, seeking to capture the highly lucrative polynomial evaluation market, builds a multi-core CPU packed with ALUs. "The professor with the most ALUs wins, he yells!" The processor has:

- 4 cores clocked at 1 GHz, capable of one 32-wide SIMD floating-point instruction per clock (1 addition, 1 multiply, etc.)
- Two hardware execution contexts per core
- A 1 MB cache per core with 128-byte cache lines (In this problem assume allocations are cache-line aligned so that each SIMD vector load or store instruction will load one cache line). Assume cache hits are 0 cycles.
- The processor is connected to a memory system providing a whopping 512 GB/sec of BW
- The latency of memory loads is 95 cycles. (There is no prefetching.) For simplicity, assume the latency of stores is 0.

A. (2 pts) What is the peak arithmetic throughput of Prof. Kayvon's processor?

B. (4 pts) What should Prof. Kayvon set the ISPC gang size to when running this ISPC program on this processor?

C. (6 pts) Prof. Kayvon runs the ISPC code on his new processor, the performance of the code is not good. What fraction of peak performance is observed when running this code? Why is peak performance not obtained?

D. (6 pts) Prof. Olukuton sees Kayvon's struggles, and sees an opportunity to start his own polynomial computation processor company that achieves double the performance of Prof. Kayvon's chip. "Oh shucks, now I'll have to double the number of cores in my chip, that will cost a fortune." Kayvon says.

TA Mario writes Kayvon an email that reads "There's another way to achieve peak performance with your original design, and it doesn't require adding cores." Describe a change to Prof. Kayvon's processor that causes it to obtain peak performance on the original workload. Be specific about how you'd realize peak performance (give numbers).

The following year Prof. Kayvon makes a new version of his processor. The new version is the **exact same quad-core processor** as the one described at the beginning of this question, except now the chip **supports 64 hardware execution contexts per core**. Also, the ISPC code is changed to compute a more complex polynomial. In the code below assume that `coeffs` is an array of a few hundred polynomial coefficients and that `expensive_polynomial` involves 100's of arithmetic operations.

```
void polynomial(uniform float coeffs[], uniform float input[],
                uniform float output[], int elementsPerTask) {
  uniform int start = taskIndex * elementsPerTask;
  uniform int end = start + elementsPerTask;
  foreach (i = start ... end) {
    output[i] = expensive_poly(coeffs, input[i]);   // 100's of arithmetic ops
  }
}

void run(int N, float* coeffs, float* input, float* output) {
  uniform int elementsPerTask = 1024;
  launch[N/elementsPerTask] polynomial(coeffs, input, output, elementsPerTask);
}
```

E. (2 pts) What is the peak arithmetic throughput of Prof. Kayvon's new processor?

F. (6 pts) Imagine running the program with N=8×1024 and N = 64×1024. Assuming that the system schedules worker threads onto available execution contents in an efficient manner, do either of the two values of N result in the program achieving near peak utilization of the machine? Why or why not? (For simplicity, assume task launch overhead is negligible.)

G. (6 pts) Now consider the case where N=9×1024. Now what is the performance problem? Describe is simple code change that results in the program obtaining close to peak utilization of the machine. (Assume task launch overhead is negligible.)

Angry Students

Problem 4. (14 points):

Your friend is developing an mobile game that features a horde of angry students chasing after professors for making long problem sets. Simulating students is expensive, so your friend decides to parallelize the computation using ISPC, so the code can take advantage of the SIMD instructions on the phone's ARM processor. The state of the game's N students is stored in the global array `students` in the code below).

The code is written like this:

```
struct Student {
  float position;
  float angriness;
};

Student students[N];

// ispc function
void updateStudents(int N, Student* students) {
  foreach (i = 0 ... N) {
    students[i].position = compute_new_position(i);
    students[i].angriness = compute_new_angriness(i);
  }
}
```

Performance is lower than expected, so your friend changes the code to this:

```
float positions[N];
float angriness[N];

// ispc function
void updateStudents(int N, float* positions, float* angriness) {
  foreach (i = 0 ... N) {
    position[i] = compute_new_position(i);
    angriness[i] = compute_new_angriness(i);
  }
}
```

The resulting code runs significantly faster. Why?

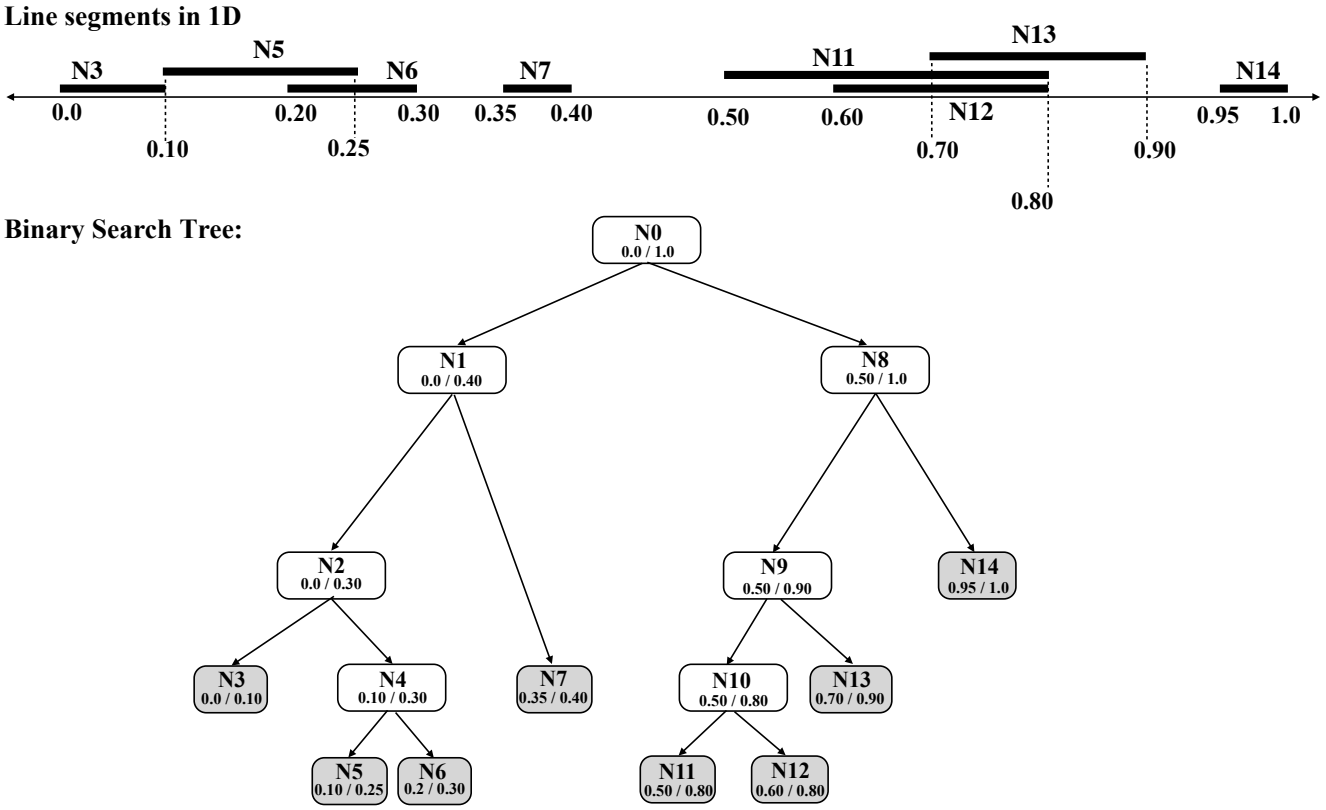
EXTRA CREDIT: SPMD Tree Search

Problem 5. (5 points):

NOTE: This question is tricky. We recommend you attempt it last since there is a LOT OF READING TO DO. If you can answer this question you really understand SIMD execution!

The figure below shows a collection of line segments in 1D. It also shows a binary tree data structure organizing the segments into a hierarchy. Leaves of the tree correspond to the line segments. Each interior tree node represents a spatial extent that bounds all its child segments. Notice that sibling leaves can (and do) overlap. Using this data structure, it is possible to answer the question "what is the largest segment that contains a specified point" without testing the point against all segments in the scene.

For example, the answer for point $p = 0.15$ is segment 5 (in node N5). The answer for the point $p = 0.75$ is segment 11 in node N11.




```
struct Node {
   float min, max;     // if leaf: start/end of segment, else: bounds on all child segments.
   bool leaf;          // true if nodes is a leaf node
   int segment_id;     // segment id if this is a leaf
   Node* left, *right; // child tree nodes
};
```

On the following two pages, we provide you two ISPC functions, `find_segment_1` and `find_segment_2` that both compute the same thing: they use the tree structure above to find the id of the largest line segment that contains a given query point.

```
struct Node {
   float min, max;     // if leaf: start/end of segment, else: bounds on all child segments.
   bool leaf;          // true if nodes is a leaf node
   int segment_id;     // segment id if this is a leaf
   Node* left, *right; // child tree nodes
};

// -- computes segment id of the largest segment containing points[programIndex]
// -- root_node is the root of the search tree
// -- each program instance processes one query point
export void find_segment_1(uniform float* points, uniform int* results, uniform Node* root_node) {

  Stack<Node*> stack;
  Node* node;
  float max_extent = 0.0;

  // p is point this program instance is searching for
  float p = points[programIndex];
  results[programIndex] = NO_SEGMENT;

  stack.push(root_node);

  while(!stack.size() == 0) {
    node = stack.pop();

    while (!node->leaf) {
      // [I-test]: test to see if point is contained within this interior node
      if (p >= node->min && p <= node->max) {
        // [I-hit]: p is within interior node... continue to child nodes
        push(node->right);
        node = node->left;
      } else {
        // [I-miss]: point not contained within node, pop the stack
        if (stack.size() == 0)
          return;
        else
          node = stack.pop();
      }
    }

    // [S-test]: test if point is within segment, and segment is largest seen so far
    if (p >= node->min && p <= node->max && (node->max - node->min) > max_extent) {
       // [S-inside]: mark this segment as ‘‘best-so-far’’
       results[programIndex] = node->segment_id;
       max_extent = node->max - node->min;
    }
  }
}
```

```
export void find_segment_2(uniform float* points, uniform int* results, uniform Node* root_node) {

  Stack<Node*> stack;
  Node* node;
  float max_extent = 0.0;

  // p is point this program instance is sarch for
  float p = points[programIndex];

  results[programIndex] = NO_SEGMENT;

  stack.push(root_node);

  while(!stack.size() == 0) {
    node = stack.pop();

    if (!node->leaf) {
       // [I-test]: test to see if point is contained within interior node
       if (p >= node->min && p <= node->max) {
          // [I-inside]: p is within interior node... continue to child nodes
          push(node->right);
          push(node->left);
       }
    } else {
       // [S-test]: test if point is within segment, and segment is largest seen so far
       if (p >= node->min && p <= node->max && (node->max - node->min) > max_extent) {
         // [S-inside]: mark this segment as ''best-so-far''
         results[programIndex] = node->segment_id;
         max_extent = node->max - node->min;
       }
    }
  }
}
```

Begin by studying `find_segment_1`.

Given the input $p = 0.1$, the a single program instance will execute the following sequence of steps: (I-test,N0), (I-hit,N0), (I-test, N1), (I-hit, N1), (I-test, N2), (I-hit, N2) (S-test,N3), (S-hit, N3), (I-test, N4), (I-hit, N4), (S-test, N5), (S-hit, N5), (S-test, N6), (S-test,N7), (I-test, N8), (I-miss, N8). Where each of the above "steps" represents reaching a basic block in the code (see comments):

- (I-test, Nx) represents a point-interior node test against node `x`.
- (I-hit, Nx) represents logic of traversing to the child nodes of node `x` when p is determined to be contained in `x`.
- (I-miss, Nx) represents logic of traversing to sibling/ancestor nodes when the point is not contained within node `x`.
- (S-test, Nx) represents a point-segment (left node) test against the segment represented by node `x`.
- (S-hit, Nx) represents the basic block where a new largest node is found `x`.

The question is on the next page...

A. (5 pts) Confirm you understand the above, then consider the behavior of a **gang of 4 program instances** executing the above two ISPC functions `find_segment_1` and `find_segment_2`. For example, you may wish to consider execution on the following array:

```
points = {0.15, 0.35, 0.75, 0.95}
```

Describe the difference between the traversal approach used in `find_segment_1` and `find_segment_2` in the context of SIMD execution. Your description might want to specifically point out conditions when `find_segment_1` suffers from divergence. (Hint 1: you may want to make a table of four columns, each row is a step by the warp and each column shows each program instance's execution. Hint 2: It may help to consider which solution is better in the case of large, heavily unbalanced trees.)

B. (5 pts) Consider a slight change to the code where as soon as a best-so-far line segment is found (inside [S-hit]) the code makes a call to a **very, very expensive function**. Which solution might be preferred in this case? Why?

This page is left as scratch...