Stanford CS149: Parallel Computing
Written Assignment 1

Problem 1: ISPC Image Processing (25 pts)

Consider the following program, where a C++ program allocates a $N \times N$ buffer and uses the ISPC function `do_work` to process its contents.

```
// ISPC function ///////////////////////////////////
void do_work(uniform int N, uniform float in[], uniform float out[]) {
  foreach (i=0 ... N*N) {
    if (in[i] == 0.0) {
       out[i] = 3.0 * in[i];  // one load, one mul, one store  (3 instructions)
    } else {
      float tmp = in[i];      // one load
      for (int j=0; j<500; j++) {
         tmp *= 2.0;          // 500 arithmetic instructions (in total)
      }
      out[i] = tmp;           // one store
    }
  }
}

// main.cpp ///////////////////////////////////
float* in = new float[N*N];
float* out = new float[N*N];

// initialize array
for (int i=0; i<N; i++)
   for (int j=0; j<N; j++)
     if (i == j)
       in[i*N+j] = 1.0;
     else
       in[i*N+j] = 0.0;

do_work(N, in, out);  // call ISPC function
```

Consider running this ISPC program on a machine with 8-wide SIMD instructions, and an ISPC gang size set to 8. Please assume that the only operations we are considering are loads, stores, and arithmetic operations in the main ISPC `foreach` loop that is marked in the figure, and that all operations take 1 clock on the processor.

A. (10 pts) How many clocks does it take to execute the call to `do_work` if $N = 16$? (An expression is fine, you do not need to reduce it to a number.)

B. (5 pts) Does your answer to part A change if the buffer initialization function was changed to the following? Why or why not?

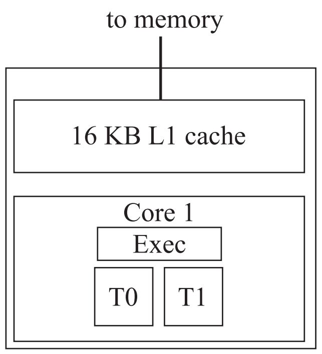
```
// initialize array
for (int i=0; i<N; i++)
   for (int j=0; j<N; j++)
     if (j == N/4 || j == 3*N/4)
       in[i*N+j] = 1.0;
     else
       in[i*N+j] = 0.0;
```

C. (10 pts) Now imagine you have a twelve-core version of the processor described in part 1 (assume single-threaded cores). Imagine that the program is changed so that `main.cpp` creates 16 ISPC tasks, where each task processes one row of the buffer, performing the same operation on each element as prescribed by `do_work` in part A. What speedup do you expect compared to running the ISPC code on the one-core processor, why? (Note: we asserted that all load/store/arithmetic instructions take one cycle, we do not want you thinking about memory access in this problem. You can also assume that there is no overhead to launching an ISPC task is 0.)

Problem 2: A Task Queue on a Multi-Core, Multi-Threaded CPU (40 pts)

The figure below shows a single-core CPU with an 16 KB L1 cache and execution contexts for up to two threads of control. The core 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 on a cache miss is 175 clocks.**

FAQ about the cache: To keep things simple, assume a cache hit takes only a one cycle. Assume cache lines are 4 bytes (a single floating point value), and the cache implements a least-recently used (LRU) replacement policy—meaning that when a cache line needs to be 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: confirm to yourself that in this situation every load will be a cache miss. (Assume the CPU does not pre-fetch loads.)



You are implementing a task queue for a system with this CPU. The task queue is responsible for executing 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
float* 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;
}
```

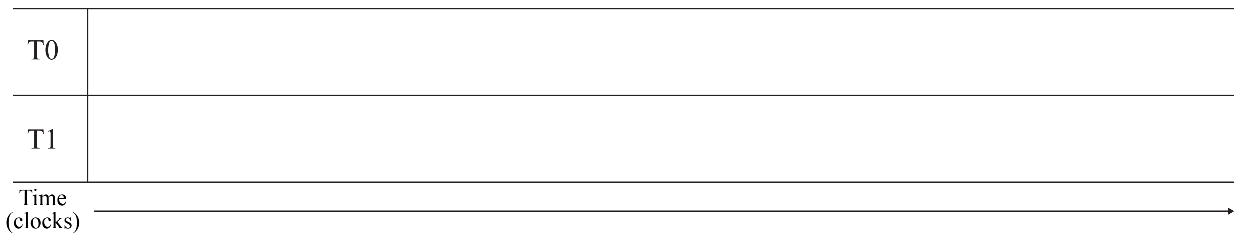
A. (8 pts) Consider one possible implementation of `TASK_A` from the code on the previous page:

```
function TASK_A(int task_id, float* X) {
   for (int i=0; i<1000; i++) {
      for (int j=0; j<1024*32; j++) {
         load X[j]   // assume this is a cache miss when i=0
         // ... 25 non-memory instructions using X
      }
   }
}
```

The inner loop of `TASK_A` scans over 32K elements = 128 KB of elements of array `X`, performing 25 arithmetic instructions after each load. This process of a load, followed by 25 instructions, 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 only 16 KB, small smaller than the amount of data read each iteration of the outer loop.**

In order to process a launch of many `TASK_A` tasks, 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.

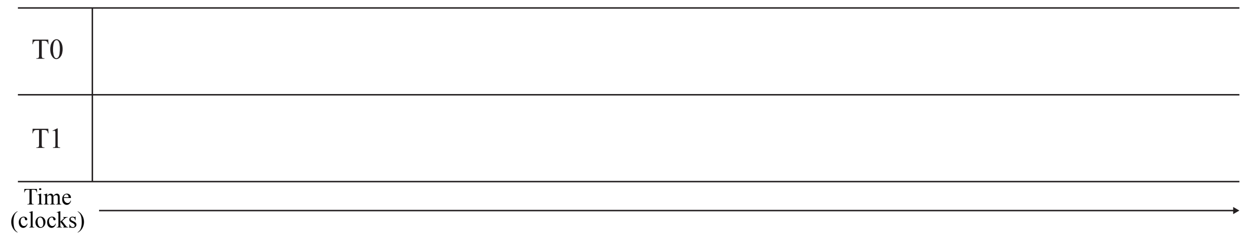
(Careful: please consider the program's execution behavior on average over the entire program's execution ("steady state" behavior). Past students have been tricked by only thinking about the behavior of the first loop iteration of the first task.) It may be helpful to draw when threads are running and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



B. (8 pts) Consider the same setup as the previous problem. How many hardware threads would the CPU core need in order for the machine to maintain peak throughput (100% utilization) on this workload?

C. (8 pts) **Now consider the case where the program is modified to so that it has very, very high arithmetic intensity (e.g, 1 million instructions per memory load 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.

D. (8 pts) **Now consider the case where the cache size is enlarged to 8 MB 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.



E. (8 pts) **Now consider the case where the L1 cache size is changed to 128 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? (Remember, each task scans over 128 KB of data, different tasks are scanning over different data, and the cache has a least-recently used eviction policy.)

Problem 3: Mixing Superscalar, Threads, and Cores (35 points)

Consider the following sequence of 10 instructions. (2 memory operations, 8 arithmetic ops)

```
1.  LD    R0 <- [R5]      // load from address given by R5
2.  MUL   R0 <- R0, R0
3.  MUL   R1 <- R0, R0
4.  ADD   R2 <- R1, R1
5.  MUL   R3 <- R1, R0
6.  ADD   R4 <- R0, R1
7.  ADD   R1 <- R2, R3
8.  ADD   R0 <- R0, R4
9.  ADD   R0 <- R0, R1
10. ST   [R5] <- R0       // store to address given by R5
```

A. (10 pts) Consider running this instruction sequence on a **single-core, but superscalar processor that can execute up to two independent instructions per clock.** For now, consider all ten instructions, both arithmetic and memory, as completing in a single cycle (latency = 1 clock). Draw the DAG of instruction dependencies given by the instruction stream above. Using this diagram compute the **speedup** observed by running this instruction stream on the 2-way super-scalar processor compared to a single-core processor that can only issue one instruction per clock (no superscalar)?

B. (6 pts) What is the speedup of a super-scalar processor that is able to issue **three instructions per clock compared to a non-superscalar processor that completes one instruction per clock?**

C. (6 pts) Now consider running this instruction sequence as the body of a loop, as shown below. The loop is written in OpenMP, a commonly used C++ extension that enable parallel execution. You will use OpenMP in Assignment 4 in this class, but for now, assume that the `#omp parallel for` syntax below means "C compiler, it is safe to parallelize iterations of this for loop by assigning them to *threads*. (And as you know each thread is mapped to one execution context on the CPU.)"

```
#pragma omp parallel for
for (int i=0; i<VERY_LARGE; i++) {
   // the sequence of ten instructions listed earlier in the problem
   // (loading from A[i] and storing to B[i])
}
```

The program is now run on a **single-core processor with support for two hardware threads (execution contexts)**. Assume the processor works as follows: each clock it ALWAYS switches which thread it can draw instructions from. It runs up to two independent instructions (if they exist) from ONLY THAT ONE THREAD, then switches to the next thread in the next clock. When running the parallel for loop above, what is the speedup of this processor compared to the **2-way superscalar, single-threaded processor from part A?** Why?

D. (7 pts) Now consider a slight modification to the single core, multi-threaded processor from the previous problem: **each clock the single-core processor can choose to execute any mixture of up to two independent instructions from (if needed) both hardware threads** (not just one thread like in part C). Given this new design, when running the code from part C, what is the expected speedup compared to **a single-core non-superscalar processor that completes one instruction per clock?**

E. (6 pts) What is the speedup obtained if we modify the baseline processor from part A to have two cores?

The following problems are PRACTICE PROBLEMS and will not be graded.

Practice 1: Buying a New Computer, Again

You plan to port the following sequential C++ code to ISPC so you can leverage the performance benefits of modern parallel processors.

```
float input[LARGE_NUMBER];
float output[LARGE_NUMBER];
// initialize input and output here ...

for (int i=0; i<LARGE_NUMBER; i++) {
   int iters;
   if (i % 16 == 0)
      iters = 256;
   else
      iters = 8;
   for (int j=0; j<iters; j++)
      output[i] += input[i];
}
```

Before sitting down to hack, you go the store, and see the following CPUs all for the same price:

- 4 GHz single core CPU capable of performing one floating point addition per clock (no parallelism)
- 1 GHz quad-core CPU capable of performing one 4-wide SIMD floating point addition per clock
- 1 Ghz dual-core CPU capable of performing one 16-wide SIMD floating point addition per clock

If your only use of the CPU will be to run your future ISPC port of the above code as fast as possible, which machine will provide the best performance for your money? Which machine will provide the least? Please explain why by comparing expected execution times on the various processors. When considering execution time, you may assume that (1) the only operations you need to account for are the floating-point additions in the innermost loop. (2) the ISPC gang size will be set to the SIMD width of the CPU.

(Hint: consider the execution time of groups of 16 elements of the input and output arrays).

Practive 2: Where Did the Speedup Go

You want to determine all prime numbers up to 10,000,000, using OpenMP to exploit multicore parallelism. The following function tests whether a number x is prime. It exploits the property that any composite number x must be divisible by some number y, such that $1 < y \leq \sqrt{x}$:

```
bool test_prime(int x)
{
    if (x == 0)
        return false;
    int lim = (int) sqrt((double) x);
    for (int i = 2; i <= lim; i++) {
        if (x % i == 0)
            return false;
    }
    return true;
}
```

Your overall scheme is to use OpenMP to launch T threads:

```
// this code parallelized nthreads iterations of the loop onto
#pragma omp parallel for schedule(static)
for (int t = 0; t < nthreads; t++) {
  tf(t, nthreads, xmax, isprime);
}
```

where each instance of thread function `tf` will do the primality testing for some subset of the possible values, setting the elements of a global array `isprime` to either `true` or `false`.

Here are two possible thread functions:

```
void thread_run_interleave(int t, int nthreads, int xmax, bool *isprime) {
  for (int x = t; x <= xmax; x += nthreads) {
        isprime[x] = test_prime(x);
  }
}

void thread_run_chunk(int t, int nthreads, int xmax, bool *isprime) {
  int npt = (xmax + nthreads - 1)/nthreads;
  int xstart = npt*t;
  int xlast = t == nthreads-1 ? xmax : xstart + npt - 1;
  for (int x = xstart; x <= xlast; x++) {
        isprime[x] = test_prime(x);
  }
}
```

Using these different functions, the following speedups are achieved:

Threads	2	3	4	5	6
interleave	1.04	1.93	1.92	3.53	1.93
chunk	1.62	2.17	2.74	3.08	3.86

A. Explain the speedup results for `thread_run_interleave`: **(Note: answering this question requires thinking about the behavior of `test_prime`. e.g., when must it do very little work?)**

(a) Why is their no speedup for 2 threads?

(b) Why isn't the speedup monotonic with respect to the number of threads? (Hint: note behavior at 4 and 6 threads.)

B. Explain why `thread_run_chunk` gets some speedup, but not very much?

Practice 3: Be a Parallel Processor Architect

You are hired to start the parallel processor design team at Lagunita Processors, Inc. Your boss tells you that you are responsible for designing the company's first shared address space multi-core processor, which will be constructed by cramming multiple copies of the company's best selling uniprocessor core on a single chip. Your boss expects the project to yield at least a 5× speedup on the performance of the program given below. You are not allowed to change the program, and assume that:

- Each Lagunita core can complete one floating point operation per clock
- Cores are clocked at 1 GHz, and each have a 1 MB cache using LRU replacement.
- All Lagunita processors (both single and multi-core) are attached to a 100 GB/s memory bus
- Memory latency is perfectly hidden (Lagunita processors have excellent pre-fetchers)

```
float A[N]; // let N = 100 million elements
float total = 0;

// ASSUME TIMER STARTS HERE ///////////////////////////////////

for (int i=0; i<N; i++)
   total += A[i];

for (int i=0; i<9; i++) {

   // made up syntax for brevity: 'parallel_for'
   // Assume iterations of this loop are perfectly partitioned
   // using blocked assignment among X pthreads each running on
   // one of the processor's X cores.
   parallel_for(int j=0; j<N; j++) {
      A[j] = A[j] / total;
   }
}

// ASSUME TIMER STOPS HERE ///////////////////////////////////
```

A. How do you respond to your boss' request? Do you believe you can meet the performance goal? If yes, how many cores should be included in the new multi-core processor? If no, explain why.

B. You tell your boss that if you were allowed to make a few changes to the code, you could deliver a much better speedup with your parallel processor design. How would you change the code to improve its performance by improving *speedup*? (A simple description of the new code is fine). If your answer was NO in part one, how many processors are required to achieve 5× speedup now? If your answer was YES, approximately what speedup do you expect from your previously proposed machine on the new code? (*Note: we are NOT looking for answers that optimize the program by rolling multiple divisions into one.*)

C. Assume that the following year, Lagunita Processors, Inc. decides to produce a 32-core version of your parallel CPU design. In addition to adding cores, your boss gives you the opportunity to further improve the processor through one of the following three options.

   - You may double each processor's cache to 2 MB.
   - You may increase memory bandwidth by 50%
   - You may add a 4-wide SIMD unit to the core so that each core can perform 4 floating point operations per clock.

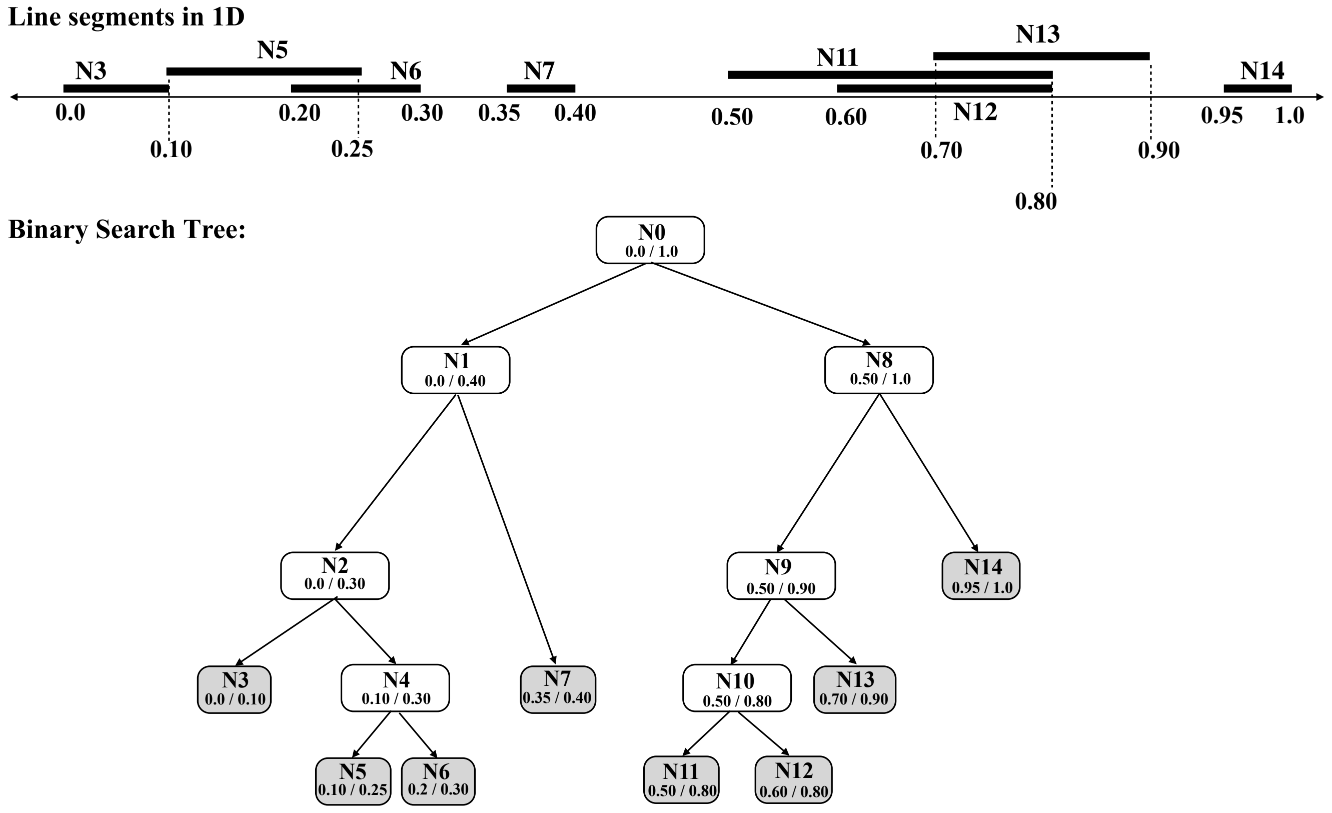
   If each of these options has the same cost, given the code you produced in part B (and what you learned from assignment 1), which option do you recommend to your boss? Why?

Practice 4: SPMD Tree Search

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. 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. 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?