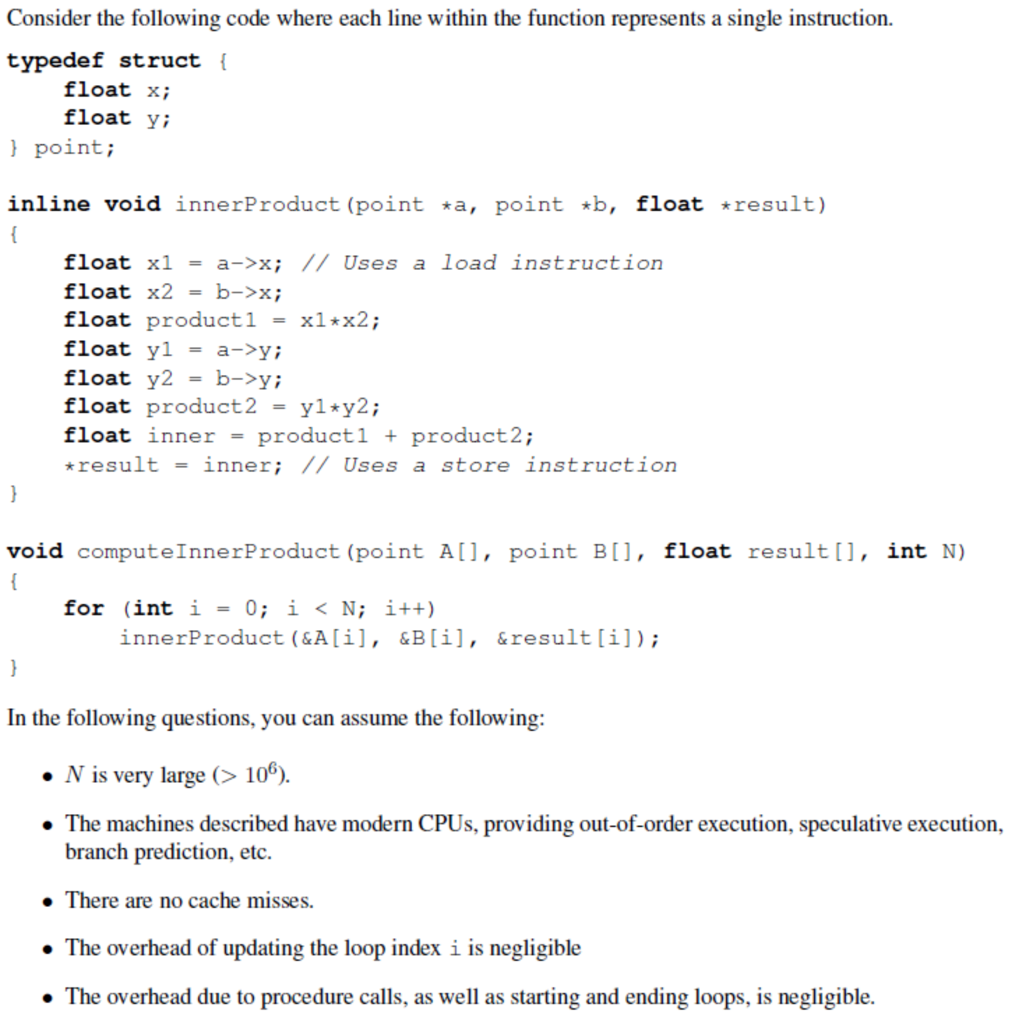
1. Why is it difficult to construct a true shared-memory computer? What is the minimum number of switches for connecting p processors to a shared memory with b words (where each word can be accessed independently)?

2. Consider a memory system with a level 1 cache of 32 KB and DRAM of 512 MB with the processor operating at 1 GHz. The latency to L1 cache is one cycle and the latency to DRAM is 100 cycles. In each memory cycle, the processor fetches four words (cache line size is four words). What is the peak achievable performance of a dot product of two vectors? Note: Where necessary, assume an optimal cache placement policy.

```
/* dot product loop */
  for (i = 0; i < dim; i++)
      dot_prod += a[i] * b[i];
```

Homework-asst-1



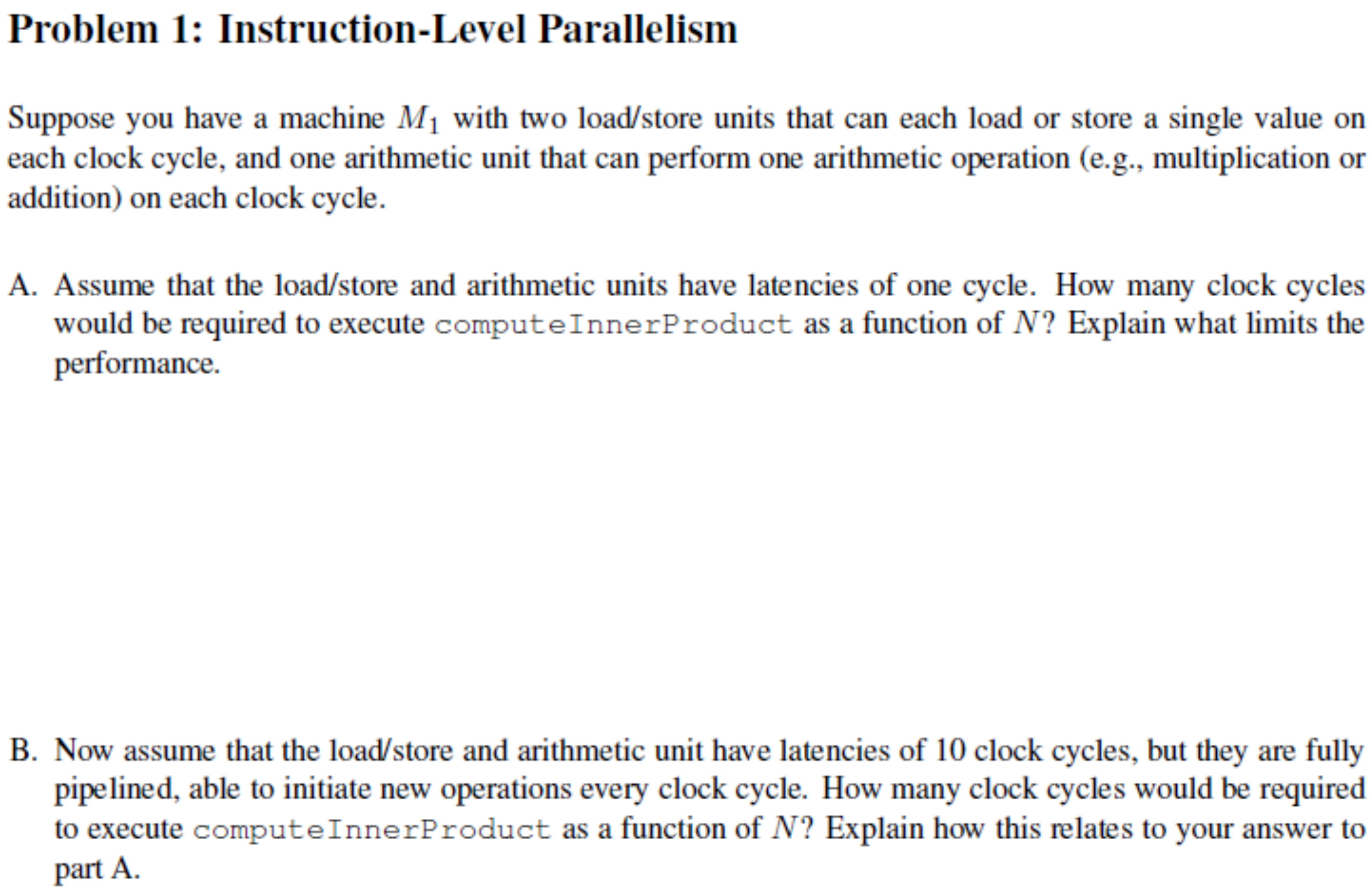
Consider the following code where each line within the function represents a single instruction.

```
typedef struct {
    float x;
    float y;
} point;

inline void innerProduct(point *a, point *b, float *result)
{
    float x1 = a->x; // Uses a load instruction
    float x2 = b->x;
    float product1 = x1*x2;
    float y1 = a->y;
    float y2 = b->y;
    float product2 = y1*y2;
    float inner = product1 + product2;
    *result = inner; // Uses a store instruction
}

void computeInnerProduct(point A[], point B[], float result[], int N)
{
    for (int i = 0; i < N; i++)
        innerProduct(&A[i], &B[i], &result[i]);
}
```

In the following questions, you can assume the following:

- N is very large ($> 10^6$).
- The machines described have modern CPUs, providing out-of-order execution, speculative execution, branch prediction, etc.
- There are no cache misses.
- The overhead of updating the loop index `i` is negligible
- The overhead due to procedure calls, as well as starting and ending loops, is negligible.

Homework-asst-1



Problem 1: Instruction-Level Parallelism

Suppose you have a machine M_1 with two load/store units that can each load or store a single value on each clock cycle, and one arithmetic unit that can perform one arithmetic operation (e.g., multiplication or addition) on each clock cycle.

A. Assume that the load/store and arithmetic units have latencies of one cycle. How many clock cycles would be required to execute `computeInnerProduct` as a function of N? Explain what limits the performance.

B. Now assume that the load/store and arithmetic unit have latencies of 10 clock cycles, but they are fully pipelined, able to initiate new operations every clock cycle. How many clock cycles would be required to execute `computeInnerProduct` as a function of N? Explain how this relates to your answer to part A.

Homework-asst-1

Problem 2: SIMD with ISPC

Consider running the following ISPC code.

```
export void computeInnerProductISPC(uniform point[] A,
                                    uniform point[] B,
                                    uniform float[] result,
                                    uniform int N)
{
    foreach(i = 0 ... N)
    {
        result[i] = A[i].x * B[i].x + A[i].y * B[i].y;
    }
}
```

Suppose machine M_2 has one 8-wide SIMD load/store unit, and one 8-wide SIMD arithmetic unit. Both have latencies of one clock cycle.

A. How many clock cycles would be required to execute `computeInnerProductISPC` as a function of N? Explain what limits the performance.

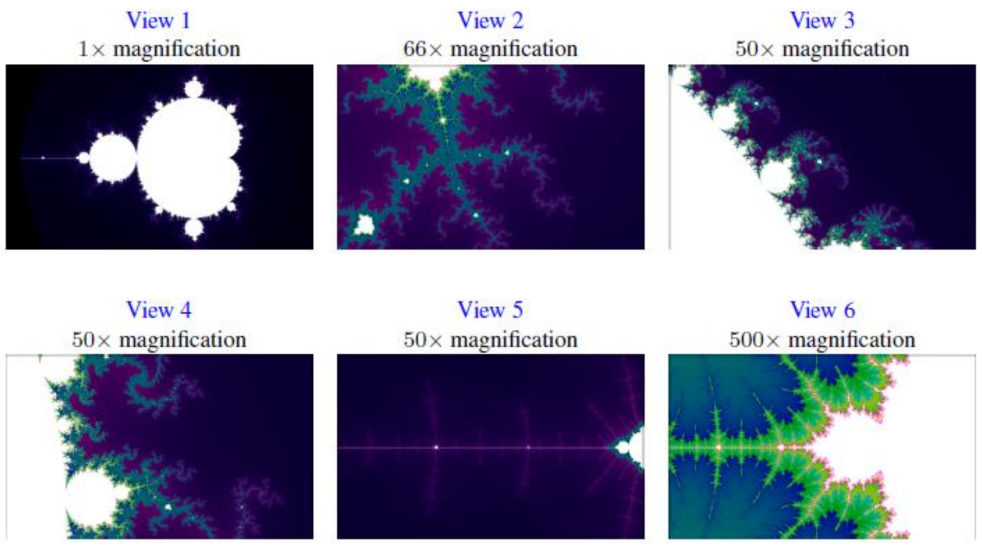
B. If we were to run the `computeInnerProductISPC` on a five-core machine M_3, where each core has the same SIMD capabilities as M_2, what would be the best speedup it could achieve over the single-core performance of part A? Explain.

Homework-asst-2

Parallel Fractal Generation Using Pthreads

Leverage the sample code provided in the course web site:
http://172.18.166.180/course/1/info

Build and run the code in the prob1_mandelbrot_threads directory of the Assignment 1 code base.This program produces the image file mandelbrot-vV -serial.ppm, where V is the view index. This image is a visualization of a famous set of complex numbers called the Mandelbrot set. As you can see in the images below, the result is a familiar and beautiful fractal. Each pixel in the image corresponds to a value in the complex plane, and the brightness of each pixel is proportional to the computational cost of determining whether the value is contained in the Mandelbrot set—white pixels required the maximum (256) number of iterations, dark ones only a few iterations, and colored pixels were somewhere in between. (See function mandel() defined in mandelbrot.cpp.) You can learn more about the definition of the Mandelbrot set at en.wikipedia.org/wiki/Mandelbrot set. Use the command option “--view V ” for V between 0 and 6 to get the different images. You can click the links below to see the different images on a browser. Take the time to do this—the images are quite striking. (View 0 is not shown—it is all white.)



Your job is to parallelize the computation of the images using Pthreads. The command-line option “--threads T” specifies that the computation is to be partitioned over T threads. In function mandelbrotThread(), located in mandelbrot.cpp, the main application thread creates T-1 additional thread using pthread_create(). It waits for these threads to complete using pthread_join(). Currently, neither the launched threads nor the main thread do any computation, and so the program generates an error message. You should add code to the workerThreadStart() function to accomplish this task. You will not need to use of any other Pthread API calls in this assignment. What you need to do:

1. Modify the code in mandelbrot.cpp to parallelize the Mandelbrot generation using two

cores.
Specifically, compute the top half of the image in thread 0, and the bottom half of the image in thread 1. This type of problem decomposition is referred to as spatial decomposition since different spatial regions of the image are computed by different processors.

2. Extend your code to utilize T threads for {2, 4, 8,16} , partitioning the image generation work into the appropriate number of horizontal blocks. You will need to modify the code in function workerThreadStart, to partition the work over the threads.

3. To confirm (or disprove) your hypothesis, measure the amount of time each thread requires to complete its work by inserting timing code at the beginning and end of workerThreadStart(). How do your measurements explain the speedup graph you previously created?