CMU 15-418/618: Parallel Computer Architecture and Programming
Practice Exercise 3 SOLUTIONS

Problem 1: Optimizing a Multi-Threaded Program (10 pts)

Your friend writes the following multi-threaded C++ program that combines two images `image1` and `image2`. The implementation uses three threads, and each thread is responsible for processing a single channel (red, green, or blue) of the image. Notice that this processing requires the thread to loop over the image data `MAX_ITERS` times.

```
struct Pixel {
   float r, g, b;
};

#define MAX_ITERS   1000
#define IMAGE_SIZE  64 * 64
float my_func(float, float);
Pixel *image1, *image2;

void workerR() {
   for (int iters=0; iters<MAX_ITERS; iters++)
      for (int i=0; i<IMAGE_SIZE; i++)
         result[i].r += my_func(image1[i].r, image2[i].r);
}

void workerG() {
   for (int iters=0; iters<MAX_ITERS; iters++)
      for (int i=0; i<IMAGE_SIZE; i++)
         result[i].g += my_func(image1[i].g, image2[i].g);
}

void workerB() {
   for (int iters=0; iters<MAX_ITERS; iters++)
      for (int i=0; i<IMAGE_SIZE; i++)
         result[i].b += my_func(image1[i].b, image2[i].b);
}

int main() {
  image1 = new Pixel[IMAGE_SIZE];
  image2 = new Pixel[IMAGE_SIZE];
  result = new Pixel[IMAGE_SIZE];

  // ... initialize result, image1, image2 here ...

  pthread_t t0, t1;
  pthread_create(&t0, NULL, workerR, NULL);
  pthread_create(&t1, NULL, workerG, NULL);
  workerB();
  pthread_join(t0, NULL);
  pthread_join(t1, NULL);

  // ... use 'result' image here ...
}
```

A. (5 pts) Your friend runs this program on the cache coherent quad-core Intel processor. Given that the problem is embarrassingly parallel and assuming the images are small enough that all three images can fit in the private L2 cache of each core, your friend expects near perfect ($3\times$) speedup. They are shocked when they don't obtain a good speedup. What is the cause of this suboptimal behavior?

Solution: The way the data is organized in memory will lead to false sharing. Specifically, as each processor writes to different color channels of the same pixel of the `result` array, all these writes map to the same cache line. The line will ping-pong between the caches of the three cores. These false sharing misses reduce program performance.

B. (5 pts) Modify the program to correct the performance problem you identified in part A. You are allowed to modify the data structures used in the code but you are not allowed to change what computations are performed by each thread. That is, `workerR` must still process the red channel of the image, `workerG` must still process the green channel, etc. You only need to describe your solution in text or pseudocode (compilable C++ is not required). (Hint: there is a very simple change.)

Solution: Modify the data layout to avoid false sharing. A simple solution is to lay out each color channel of an image in contiguous memory. That represent the image using four arrays, rather than an array of structs, as shown below.

```
struct Image {
   float r[IMAGE_SIZE];
   float g[IMAGE_SIZE];
   float b[IMAGE_SIZE];
};
```

One common incorrect answer was to pad the `Image` struct out to cache line size. This answer ensures that different pixels reside on different cache lines, but does nothing to eliminate the false sharing problem. In fact it only decreases locality of access in the input `image1` and `image2` arrays, potentially make performance even worse.

One common partial credit answer was to insert padding in the struct between the r,g,b components (effectively making the struct three cache lines in size, and placing each value on a different cache line). While this eliminates false sharing, it results in very poor spatial locality. As a result of this layout, each data access will result in a cache miss! However, since this answer eliminated false sharing, we gave partial credit.

Problem 2: Particle Simulation (10 pts)

Consider the following code that uses a simple $O(N^2)$ algorithm to compute forces due to gravitational interactions between all N particles in a particle simulation. One important detail of this algorithm is that force computation is symmetric (`gravity(i,j) = gravity(j,i)`). Therefore, iteration `i` only needs to compute interactions with particles with index `j`, where `i<j`. As a result, the work done by the algorithm is $N^2/2$ rather than N^2.

In this problem, **assume the code is run on a dual-core processor, with infinite memory bandwidth. The processor implements invalidation-based cache coherence across the cores. The cache line size is 64 bytes.**

```
struct Particle {
   float force;  // for simplicity, assume force is represented as a single float
};

Particle particles[N];

void compute_forces(int threadId) {

  // thread 0 takes first half, thread 1 takes second half
  int start = threadId * N/2;
  int end = start + N/2;

  for (int i=start; i<end; i++) {

    // only compute forces for each pair (i,j) once, then accumulate force
    // into *both* particle i and j

    for (int j=i+1; j<N; j++) {
       float force = gravity(i, j);
       particles[i] += force;
       particles[j] += force;
    }
  }
}
```

The question is on the next page.

A. (4 pts) The function `compute_forces` above is run by two threads on a dual-core processor. There is a correctness problem with the code. Using only the synchronization primitive:

`atomicAdd(float* addr, float val)`

Fix the correctness bug in the code. **However, to get full credit your solution should be efficient—it should do better than making N^2 calls to `atomic_add`** (at least by an integer constant factor). Solutions that incur significant storage overhead or increase the amount of work done by the algorithm are not allowed.

```
void compute_forces(int threadId) {

  // thread 0 takes first half, thread 1 takes second half
  int start = threadId * N/2;
  int end = start + N/2;

  for (int i=start; i<end; i++) {

    // only compute forces for each pair (i,j) once, then accumulate force
    // into *both* particle i and j

    for (int j=i+1; j<N; j++) {
        float force = gravity(i, j);
        if (i < N/2)
          particles[i] += force;
        else
          atomicAdd(&(particles[i]), force);
        if (j < N/2)
          particles[j] += force;
        else
          atomicAdd(&(particles[j]), force);
    }
  }
}
```

B. (2 pts) There is also a significant **performance problem** in the implementation that results in a speedup that is significantly lower than $2\times$ on the two-core processor. What is the problem?

Solution: The problem is workload imbalance. Thread 0 performs significantly more work than thread 1.

C. (4 pts) Give an implementation of `compute_forces` that (1) achieves good workload balance between the two threads (2) does not significantly increase the amount of work performed (work should be no more than $N^2/2 + O(N)$) and (3) does not use fine-grained `atomicAdd` synchronization. However, you are allowed to allocate O(N) storage and use a barrier. Pseudocode is fine.

```
void compute_forces(int threadId) {

  // per thread partial sums
  float particles[2][N];  // initialize to zero

  // interleave iterations (dynamic work queue would have been
  // a reasonable alternative as well)
  for (int i=threadId; i<N; i+=2) {
    for (int j=i+1; j<N; j++) {
        float force = gravity(i, j);
        particles[threadId][i] += force;
        particles[threadId][j] += force;
    }
  }

  barrier();

  // NOTE: In the answer below the work is parallelized, but we also
  // gave credit to answers that did not parallelize the combine step and
  // did all this work on one thread

  // combine partial sums into final result
  int start = threadId * N/2;
  int end = start + N/2;
  for (int i=start; i<end; i++) {
    particles[i] = particles[0][i] + particles[1][i];
  }
}
```