Stanford CS149: Parallel Computing
Written Assignment 2 SOLUTIONS

A Grading Pipeline

Problem 1. (25 points):

Some of the CS149 CAs get organized to grade the midterm, which you can assume has five questions. To ensure fairness, they decide that each question should be graded by one CA, and that to grade each exam they will organize themselves in a pipeline. James takes question 1, Arden takes question 2, Raj takes question 3, Keshav question 4, and Edmund question 5. The CAs sit in a line at the same table, and for each exam James grades question 1, then passes the exam to Arden who grades Q2, who passes to Raj to grade Q3, etc. Pranil and Drew throw up their hands and say "there seems to only be five questions, well we guess we can sit this one out!

Assume that it takes James 5 minutes to grade each exam's Q1, Arden needs 6 minutes to grade Q2, Raj needs 15 minutes to grade Q3, Keshav needs 5 minutes to grade Q4 and Edmund needs 5 minutes to grade Q5. The CAs grade exams in a pipelined fashion to maximize their throughput.

A. (8 pts) Given this configuration, what is the **latency** of completing the grading of any one exam?

Solution: 5 + 6 + 15 + 5 + 5 = 36 minutes

B. (8 pts) What is the **STEADY-STATE THROUGHPUT** of the CAs, in terms of exams per hour? Keep in mind that the CAs are pipelining grading of exams, so James grabs the next exam to grade as soon as he is done with Q1 from his current exam. (While it doesn't matter in the answer to this problem, since we are asking for steady-state throughput, it might be helpful to assume that the pile of ungraded exams between any two CAs is limited to a small fixed size.)

Solution: In steady state, the CAs will complete one exam every 15 minutes (bottlenecked by the rate at which Raj can complete Q3's), so the throughput is 4 exams per hour.

C. (9 pts) The professors start getting anxious because the CAs haven't completed grading, so they send an angry email to the staff. "Let's speed it up already!" they write. The CA's look at Pranil and Drew and, say "Quit surfing the internet reading articles about ML accelerators and please come help!" Assuming that Pranil and Drew grade questions at exactly the same speed as the other CAs, which question should they help with grading? (please choose one) Describe why? What is the new **steady state throughput** of the staff in terms of exams per hour? (Regardless of the question chosen, Assume that Pranil/Drew's help is going to come in the form of grading a different exam in parallel with the other CAs working on the same question. Their help doesn't reduce the amount of time it takes to grade one question on one exam.)

Solution: They should help Raj, since Raj is the bottleneck in the pipeline. As a result of them chipping in , the rate of grading Q3 will improve by 3×. Previously the rate of processing Q3 was 4 exams per hour, but now it's 12. As a result, the pipeline is now bottlenecked by Arden's performance on Q2. Since Arden can process Q2's at a rate of 10 exams per hour, that's now the overall rate of the pipeline. It was not required in your answer, but it is important to note that the extra CA's chipping in doesn't improve the latency of completing grading for any one exam. It remains the same 36 minutes.

Misc Problems

Problem 2. (25 points):

A. (12 pts) A key idea in this course is the difference between *abstraction* and *implementation*. Consider two abstractions we've studied: ISPC's `foreach` and Cilk's spawn construct. **Briefly describe how these two abstractions have similar semantics.** (Hint: what do the constructs declare about the associated loop iterations? Be precise!). **Then briefly describe how their implementations are quite different** (Hint: consider their mapping to modern CPUs). As a reminder, we give you two syntax examples below:

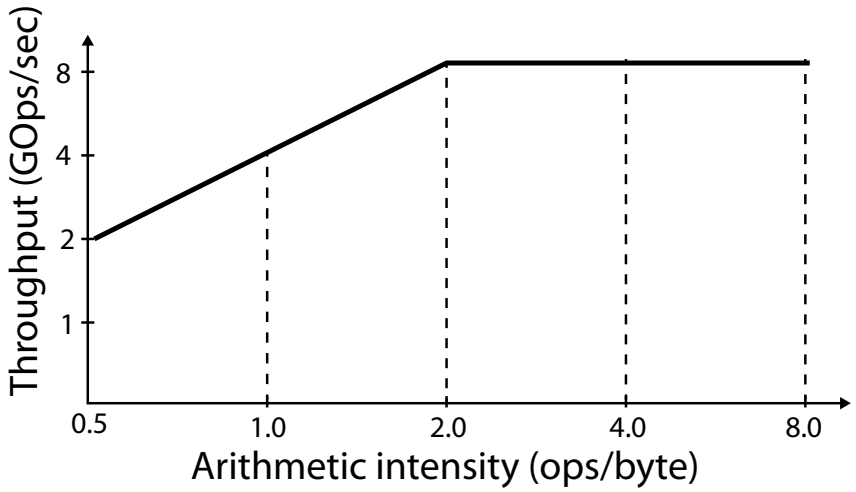
```
ISPC foreach:                          Cilk:
============================           ============================
                                       void f(int i, float* x, float* y) {
                                         x[i] = y[i]
                                       }

foreach (i = 0 ... 100) {              for (int i=0; i<100; i++) {
  x[i] = y[i];                           cilk_spawn f(i, x, y);
}                                      }
```

Solution: In both cases, the constructs declare that the loop iterations are independent and can be executed in any order without violating program correctness. One of those valid orderings is a parallel ordering, and both ISPC and Cilk use different mechanisms to implement parallel execution of the loops. ISPC employs SIMD vector instructions and most Cilk implementations will use a pool of worker threads. To make sure students understand the difference between a programming construct declaring work to be independent (allowing any possible order), vs explicitly parallel, we did not give full credit to solutions that stated the semantics of the constructs dictated "parallel execution".

B. (13 pts) In class we described the usefulness of making roofline graphs, which plots the instruction throughput of a machine (gigaops/sec) as a function of a program's arithmetic intensity (ops performed per byte transferred from memory). Note moving along the X axis is changing the properties of the code being run. The Y axis plots the performane of the machine when running a specified program. Consider the roofline plot below. Please plot the roofline curve for a machine featuring a **1 GHz dual-core processor. Each core can execute one 4-wide SIMD instruction per clock**. This processor is connected to a memory system providing 4 GB/sec of bandwidth. *Hint: what is the peak throughput of this processor? What are its bandwidth requirements when running a piece of code with a specified arithmetic intensity? Recall ops/second $\times$ bytes/op is bytes/sec. Arithemetic intensity is 1/(bytes/op).*

Plot the expected throughput of the processor when running code at each arithmetic intensity on the X axis, and draw a line between the points.

Solution: The maximum compute throughput is 8 Gigaops/second. Programs with an arithmetic intensity of 2 ops/byte require 4 GB/sec of bandwidth run at peak rate. At a arithmetic intensity of 1, the program would require double the bandwidth available on the machine. Therefore, it runs at 50% of the peak rate. At a arithmetic intensity of 1/2, the program would run at 25% of peak rate. Therefore, the slope of the line before hitting peak compute throughput is the memory bandwidth (4 GB/sec).



A Barrier is Worth a 1000 Locks

Problem 3. (25 points):

Consider the following code written in an SPMD style. Note this is not ISPC code. It's C-like code, but assume that N threads are running the code. The threads cooperate to compute a histogram for the values in an input array `data`. You can assume that data contains random numbers between 0 and 100, the histogram has 10 bins, and that `bins[i]` is supposed to contain the count of the number of elements in `data` that fall between $10 \times i$ an $10 \times (i + 1)$

```
// These variables are global variables accessible to all threads.

const int N = VERY_LARGE_NUMBER;  // assume N is a very large number
const int NUM_THREADS = 4;
int data[N];
int bins[10];  // assume initialized to 0
Lock myLock;

// This function is run in SPMD fashion by all threads

void run(int threadId) {
  int elsPerThread = N / NUM_THREADS;
  int start = threadId * elsPerThread;
  int end = start + elsPerThread;

  for (int i=start; i<end; i++) {
    int binId = data[i] / 10;
    myLock.lock();
    bins[binId]++;
    myLock.unlock();
  }
}
```

A. (10 pts) You run the program on a four core processor, and observe that it gets the correct answer, and that work is well distributed among the threads. However you don't observe a great speedup compared to a single threaded version of the code. What is a potential significant performance problem?

Solution: There is frequent locking that may significantly hurt performance since the cost of acquiring the lock is large compared to the cost of the work (a single division) done per iteration.

B. (15 pts) Imagine that instead of locks, you are allowed to use a single `barrier()` in the code. Please give a solution that yields good work distribution onto all 4 threads, uses no locks, and uses only a single call to `barrier()`. Your solution is allowed to allocate new global or per-thread variables. **Hint: Keep in mind that `N` is assumed to be much, much larger than the number of bins in the histogram.**

```
const int N = VERY_LARGE_NUMBER;  // assume N is a very large number
const int NUM_THREADS = 4;
int data[N];
int bins[10];  // assume initialized to 0
int localBins[NUM_THREADS][10];

void run(int threadId) {
  int elsPerThread = N / NUM_THREADS;
  int start = threadId * elsPerThread;
  int end = start + elsPerThread;

  for (int i=start; i<end; i++) {
    int binId = data[i] / 10;
    bins[threadId][binId]++;
  }

  // wait for all threads to complete updated to their local bins
  barrier();

  // now have thread 0 sum all the partial histograms
  if (threadId == 0) {
    for (int i=0; i<10; i++) {
      for (int j=0; j<4; j++) {
        bins[i] += localBins[j][i];
      }
    }
  }
```

Practice with Data-Parallel Thinking

Problem 4. (25 points):

Assume you are given a library that can execute a bulk launch of N independent invocations of an application-provided function using the following CUDA-like syntax:

```
my_function<<<N>>>(arg1, arg2, arg3...);
```

For example the following code would output: (`id` is a built-in id for the current function invocation)

```
void foo(int* x) {
   printf("Instance %d : %d\n", id, x[id]);
}
int A[] = {10,20,30}
foo<<<3>>>(A);

"Instance 0 : 10"
"Instance 1 : 20"
"Instance 2 : 30"
```

The library also provides the data-parallel function `exclusive_scan` (using the + operator) that works as discussed in class.

```
exclusive_scan(N, in, out);

Example usage:
N    = 6
in   = {1, 2, 3, 4, 5, 6}
====================================
out  = {0, 1, 3, 6, 10, 15}
```

In this problem, we'd like you to design a data-parallel implementation of `largest_segment_size()`, which, given an array of flags that denotes a partitioning of an array into segments, computes the size of the longest segment in the array.

```
int largest_segment_size(int N, int* flags);
```

The function takes as input an array of N flags (`flags`) (with 1's denoting the start of segments), and returns the size of the largest segment. The first element of flags will always be 1. For example, the following flags array describes five segments of lengths 4, 2, 2, 1, and 1.

```
N       = 10
flags   = {1,  0,  0,  0,  1,  0, 1,  0, 1,  1}
=============================================
result:  = 4
```

Questions on next page...

A. (12 pts) The first step in your implementation should be to compute the size of each segment. Please use the provided library functions (bulk launch of a function of your choice + `exclusive_scan` to implement the function `segment_sizes()` below. *Hint: We recommend that you get a basic solution done first, then consider the edge cases like how to compute the size of the last segment.*

```
// Example output of segment_sizes(N, flags, num_segs, sizes):
//   N        =  8
//   flags    =  {1, 0, 1, 0, 0, 0, 1, 0}
//   ==================================
//   num_segs = 3
//   sizes    = {2, 4, 2}

void write_starts(int N, int* flags, int* scanned, int* starts) {
  if (flags[id] == 1)
    starts[scanned[id]] = id;
}

void write_sizes(int N, int num_segs, int* starts, int* sizes) {
   if (id < N-1)
     sizes[id] = starts[id+1]-starts[id];
   else
     sizes[id] = N - starts[id];
}

// You can allocate any required intermediate arrays in this function
// You may assume that 'seg_sizes' is pre-allocated to hold N elements,
// which is enough storage for the worst case where the flags array
// is all 1's.
void segment_sizes(int N, int* flags, int* num_segs, int* seg_sizes) {

     int scanned[N];
     int seg_starts[N];

     exclusive_scan(N, flags, scanned);

     *num_segs = if (flags[N-1] == 0) scanned[N-1] ? scanned[N-1]+1;

     write_starts<<<N>>>(N, flags, scanned, seg_starts);
     write_sizes<<total_segs>>(N, *num_segs, seg_starts, seg_sizes)
}
```

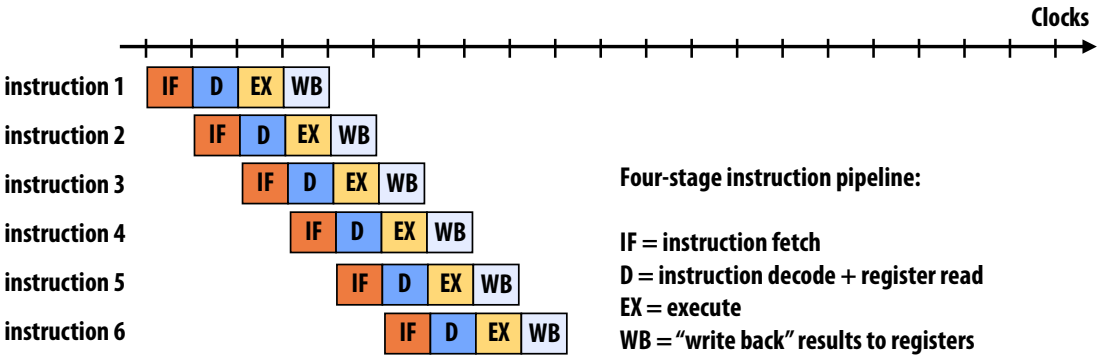
B. (13 pts) Now implement `largest_segment_size()` using `segment_sizes()` as a subroutine. **NOTE: this problem can be answered even without a valid answer to Part A.** Your implementation may assume that the number of segments described by `flags` is always a power of two. A full credit implementation will maximize parallelism and minimize work when computing the maximum segment size from an array of segment sizes. *Hint: we are looking for solutions with `lg2(num_segs)` span.*

```
void max_step(int num_segs, int stride, int* input, int* output) {
  int left_idx =  id * 2 * stride;
  int right_idx = left + stride;
  output[left_idx] = max(input[left_idx], input[right_idx]);
}

int largest_segment_size(int N, int* flags) {

  int num_segs;
  int seg_sizes[N];

  segment_sizes(N, flags, &num_segs, seg_sizes);

  int num_steps = lg_2(num_segs);
  int tmp[num_segs];

  // implement a max reduction tree using log_2(num_segs) steps
  for (int i=0; i<num_steps; i++) {
    int stride = pow(2, i);
    int num_workers = num_segs / (2 * stride);

    // note: safe to pass seg_sizes as input and output array here
    max_step<<<num_workers>>>(num_segs, stride, seg_sizes, seg_sizes);
  }

  // maximum segment size now sits in element 0
  return seg_sizes[0];
}
```

PRACTICE PROBLEM 1: A Cardinal Processor Pipeline

The fast-growing startup Cardinal Processors, Inc. builds a single core, single threaded processor that executes instructions using a simple four-stage pipeline. As shown in the figure below, each unit performs its work for an instruction **in one clock**. To keep things simple, assume this is the case for all instructions in the program, including loads and stores (memory is infinitely fast).

The figure shows the execution of a program with six **independent instructions** on this processor. *However, if instruction B depends on the results of instruction A, instruction B will not begin the IF phase of execution until the clock after WB completes for A.*



A. Assuming all instructions in a program are **independent** (yes, a bit unrealistic) what is the instruction throughput of the processor?

Solution: 1 instruction per clock. One instruction completes each cycle.

B. Assuming all instructions in a program are **dependent** on the previous instruction, what is the instruction throughput of the processor?

Solution: 1/4 instruction per clock. One instruction completes every four cycles.

C. What is the latency of completing an instruction?

Solution: 4 cycles

D. Imagine the IF stage is modified to improve its throughput to fetch TWO instructions per clock, but no other part of the processor is changed. What is the new overall maximum instruction throughput of the processor?

Solution: Throughput remains one instruction per clock. IF might be able to produce outputs at two instructions per clock, but the rest of the pipeline stages run at one element per clock, so the higher throughput of IF goes unused.

E. Consider the following C program:

```
float A[500000];
float B[500000];
// assume A is initialized here

for (int i=0; i<500000; i++) {
    float x1 = A[i];
    float x2 = 6 * x1;
    float x3 = 4 + x2;
    B[i] = x3;
}
```

Assuming that we consider only the four instructions in the loop body (for simplicity, disregard instructions for managing the loop or calculating load/store addresses), what is the average instruction throughput of this program? (Hint: You should probably consider instruction dependencies, and at least two loop iterations worth of work).

Solution: The throughput is 4/13 instructions per clock. All instructions within the loop body are dependent on each other, but the last instruction in the body is independent of the first instruction executed in the next iteration of the loop. Therefore there is a steady-state pattern of independent, dependent, dependent, dependent, ...

F. Modify the program to achieve peak instruction throughput on the processor. Please give your answer in C-pseudocode.

Solution: The main idea is to "unroll the loop four times" and reorder instructions so that each dependent instructions are separated by four other instructions. Therefore, the prior instruction will have completed before the later instruction dependent on it issues. Students might be interested to know the "loop unrolling" is a common compiler optimization. (Bonus fact: given the answer below, can you imagine a reason why setting the ISPC gang size to 16 on an AVX-capable machine might yield better performance than using a gang size of 8?)

```
for (int i=0; i<500000; i+=4) {
   float x1[4], x2[4], x3[4];

   x1[0] = A[i];    x1[1] = A[i+1];   x1[2] = A[i+2];   x1[3] = A[i+3];
   x2[0] = 6*x1[0]; x2[1] = 6*x1[1];  x2[2] = 6*x1[2];  x2[3] = 6*x1[3];
   x3[0] = 4+x2[0]; x3[1] = 4+x2[1];  x3[2] = 4+x2[2];  x3[3] = 4+x2[3];
   B[i] =  x3[0];   B[i+1] = x3[1];   B[i+2] = x3[2];   B[i+3] = x2[3];
}
```

G. Now assume the program is reverted to the original code from part E, but the for loop is parallelized using OpenMP. (Recall from written assignent 1 is that openMP is a set of C++ compiler extensions that enable thread-parallel execution. Iterations of the for loop will be carried out in parallel by a pool of worker threads.)

```
// assume iterations of this FOR LOOP are parallelized across multiple
// worker threads in a thread pool.
#pragma omp parallel for
for (int i=0; i<100000; i++) {
  float x1 = A[i];
  float x2 = 2*x1;
  float x3 = 3 + x2;
  B[i] = x3;
}
```

Given this program, imagine you wanted to add multi-threading to the **single-core processor** to obtain **peak instruction throughput** (100% utilization of execution resources). What is the smallest number of threads your processor could support and still achieve this goal? You may not change the program.

Solution: Four-threads per core. All instructions within a thread are still dependent (recall one thread would only obtain 4/13 efficiency, like in part B), but the four-cycle latency of instruction execution can be covered by executing instructions from other threads. This is because instructions from different threads will be independent.

PRACTICE PROBLEM 2: Particle Simulation

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, there are $N^2/2$ called to `gravity` 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
  Lock l;
};

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);

      lock(particles[i].l);
      particles[i] += force;
      unlock(particles[i].l);

      lock(particles[j].l);
      particles[j] += force;
      unlock(particles[j].l);
    }
  }
}
```

Question is on the next page...

A. Although the code makes $N^2/2$ calls to `gravity()` it takes N^2 locks. Modify the code so that the number of lock/unlock operations is reduced by $2\times$. You may not allocate additional variables or change how look iterations are mapped to the threads.

Solution: this problem was about identifying and understanding loop iterations that may write to the same outputs. Changing the loop body to the code below would allow 1/2 the lock/unlock operations to be eliminated::

```
  float force = gravity(i,j);

  if (i<N/2)
    particles[i] += force;
  else {
    lock(particles[i].l);
    particles[i] += force;
    unlock(particles[i].l);
  }

  if (j<N/2)
     particles[j] += force;
  else {
    lock(particles[j].l);
    particles[j] += force;
    unlock(particles[j].l);
  }
}
```

B. **(This questions can be answered independently from part A)** Looking at the original code, there another major performance problem that does not have to do with the number of lock/unlock operations. Please describe the problem and then describe a solution. Clearly describing an implementable solution strategy is fine, you do not need to write precise pseudocode.

Solution: The problem is work imbalance among the threads. Any solution that yields better workload balance is acceptable, such as setting up a work queue to dynamically assign rows to threads, or just statically interleaving rows to threads.

PRACTICE PROBLEM 3: Because The Professor with the Most ALUs (Sometimes) Wins

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. What is the peak arithmetic throughput of Prof. Kayvon's processor?

Solution: 4 cores $\times$ 32-wide SIMD ALUs $\times$1 GHz = 128 GFLOPS

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

Solution: It should be (at least) 32, since that is the SIMD width of the machine.

C. 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?

Solution: The code is memory latency bound. In steady state, a thread waits 95 cycles for a load, then performs 5 math ops, then waits another 95 cycles for a load, etc. Only 5 of these 95 stall cycles can be hidden by the other thread, so it runs at 10/100, or 1/10 of peak. It should be noted that this code is not bandwidth bound—even if the core was somehow running at peak rate, there would be sufficient bandwidth to feed the processor.

D. 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).

Solution: Add more hardware multi-threading. With 20 threads, memory latency will be fully hidden an the system will run at peak rate.

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. What is the peak arithmetic throughput of Prof. Kayvon's new processor?

Solution: Still 128 GFLOPS. Peak compute capability has not changed by adding more hardware execution contexts.

F. 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.)

Solution: In all cases, yes. Even though all thread slots are not utilized, there is no benefit to more than one thread of memory latency hiding in this scenario.

G. 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.)

Solution: Drop the elements per task so that there are at least as many tasks as execution contexts. A solution should also make sure the number of tasks is an exact multiple of the number of execution contexts, or at least several times greater.

PRACTICE PROBLEM 4: Sending Messages

A. Your friend suspects that their program is suffering from high communication overhead, so to overlap the sending of multiple messages, they try to change their code to use asynchronous, non-blocking sends instead of synchronous, blocking sends. The result is this code (assume it is run by thread 1 in two-thread program).

```
float mydata[ARRAY_SIZE];
int dst_thread = 2;

update_data_1(mydata); // updates contents of mydata
async_send(dst_thread, mydata, sizeof(float) * ARRAY_SIZE);

update_data_2(mydata); // updates contents of mydata
async_send(dst_thread, mydata, sizeof(float) * ARRAY_SIZE);
```

Your friend runs to you to say "my program no longer gives the correct results." What is their bug?

Solution: The problem is that even though the first `async_send` *call returns, there is no guarantee that the send operation has completed at this time. As a result, the contents of the buffer* `mydata` *may be overwritten before the send completes. The receiver may receive incorrect data for the first message.*

B. In class we talked about the `barrier()` synchronization primitive. No thread proceeds past a barrier until all threads in the system have reached the barrier. (In other words, the call to `barrier()` will not return to the caller until its known that all threads have called `barrier()`. Consider implementing a barrier in the context of a message passing program that is only allowed to communicate via **blocking sends and receives**. Using only the helper functions defined below, implement a barrier. Your solution should make no assumptions about the number of threads in the system. **Keep in mind that all threads in a message passing program execute in their own address space—there are no shared variables.**

```
// send msg with id msgId and contents msgValue to thread dstThread
void blockingSend(int dstThread, int msgId, int value);

// recv message from srcThread. Upon return, msgId and msgValue are populated
void blockingRecv(int srcThread, int* msgId, int* msgValue);

// returns the id of the calling thread
int getThreadId();

// returns the number of threads in the program
int getNumThreads();

#define TYPE_BARRIER_MESSAGE 0
#define KEWL_U_CAN_EXIT_NOW 1
#define ERMAHGERRD_IM_IN_HURR 2

void barrier() {

  int threadId = getThreadId();
  int msgId;
  int msgVal;

  if (thread_id == 0) {
    int msgId;
    int msgVal;
    int arrivals = 0;
    for (int i = 1; i < getNumThreads(); i++) {
      blockingRecv(i, &msgId, &msgVal);
      if (msgId == TYPE_BARRIER_MESSAGE && msgVal == ERMAHGERRD_IM_IN_HURR)
        arrivals++;
    }

    if (arrivals == getNumThreads()-1) {
      for (int i = 1; i < getNumThreads(); i++) {
        blockingSend(i, TYPE_BARRIER_MESSAGE, KEWL_U_CAN_EXIT_NOW);
      }
    } else {
      // do some error handing here
    }
  } else { // all other threads that aren't thread 0
    blockingSend(0, TYPE_BARRIER_MESSAGE, ERMAHGERRD_IM_IN_HURR);
    blockingRecv(0, TYPE_BARRIER_MESSAGE, KEWL_U_CAN_EXIT_NOW);
  }
}
```