Test 1: Lectures 1 to 13

EEE4084F 2017-04-25

ANSWERS

Question 1: Converting serial to parallel **[10 Total]**

You are tasked to implement a multi-threaded moving average filter. The filter operation works on separate input (X) and output (Y) buffers each of size N. The moving average has a window size of M (i.e. M is the number of elements that are averaged to get Y_i except at the start and end of the array). The OCTAVE code for this moving average filter is provided below. As you can see the behaviour is a bit different at the start and at the end of the array (i.e. where the indexing would have gone out of bounds).

```
# X : input array 1..N of floats Y : output array 1..N of floats
# M : int window size
function [Y] = maf (X,M)
  N = length(X);
  halfM = round(M/2);
  Y=zeros(1,N);
  for i=1:N
    if (i>halfM)
        if (i<N-halfM)
            Y(i)=mean(X(i-halfM:i+halfM));
        else
            Y(i)=mean(X(i:N));
        endif
    else
        Y(i)=mean(X(1:i));
    endif
   endfor
endfunction
```

You can assume that the size of N will be greater than M (generally M won't be more than about 10). A scenario of the results would be as follows if N was 10 and M was 2, Y(1) = X(1), Y(2) = mean(X(1),X(2),X(3)), Y(3) = mean(X(2),X(3),X(4)) … etc.

Answer the questions below:

#1 What level of granularity would you say this problem is? Is it course gained or fine grained? How much so? Motivate your answer. **[2]**

ANSWER: This is course granularity, the level is proportional to M. The greater M is the higher level of granularity. It is embarrassingly parallel as it is very easy to separate into distinct tasks and there is no data that needs to be shared between tasks (that is if the input is in shared memory as opposed to separate between tasks).

#2 This code needs to be parallelized using pthreads and should run on a multicore CPU (e.g. an Intel i7). Explain how you would design a solution for this, what threads would you have, what operations would they do. Use an illustration to help your explanation and to show how you would partition the data, and possibly having different tasks. **[5]**

ANSWER/MARKERS GUIDELINES: There are two fairly obvious approaches, translating the provided function into C and using shared memory. The data will be shared and an interleaved approach could be used, e.g. where thread i works on Y[i*nthreads]. There is no need for a mutex. It could be made slightly more efficient by the main thread working on the start and end sections while the other threads complete their tasks (i.e. to parallelize the special cases with the rest of the processing).

#3 With possible help from the cheatsheet on the last page, provide a C / C++ implementation of a thread routine for this program (you may or may not decide to have different thread routines, provide a comment as to which routine you are providing). You do not need to show main function code for creating threads. **[3]**

ANSWER/MARKERS GUIDELINES:

```
struct MAFData {
  float* X;
  float* Y;
  int N, M;
  int threadid;
}
void* maf_thread (void *ptr)
{
    int i;
    MAFData *p = ptr;  // assume the array is passed
    int halfM = round(p->M/2);
    for(i = 1; i <= p->N; i++) {
      if (i>halfM) {
        if (i<p->N-halfM)
            p->Y[i]=mean(p->X,i-halfM,i+halfM);
        else
            p->Y[i]=mean(p->X,i,p->N);
        endif
    else
        p->Y[i]=mean(X,1,i);
    }

}
```

Question 2: Landscape of Parallel Computing [10 Total]

This question relates to paper 1 titled "The Landscape of Parallel Computing Research: A View from Berkeley" by Asanovic et al.

#1 There is much talk in the paper about old conventional wisdoms and new conventional wisdoms. What is meant by old and new conventional wisdoms? (No need to provide an example for this question).

[2]

ANSWER/MARKERS GUIDELINES:

Old conventional wisdom = understanding and traditional beliefs that were established and assumed true in the past.

New conventional wisdom = new understanding and beliefs that have replaced (and sometimes extended) the old conventional wisdoms of the past.

#2 Name two old conventional wisdoms and how these have been replaced, in the new 'landscape of computing' by a corresponding new conventional wisdom.

[4]

ANSWER/MARKERS GUIDELINES:

Can be any of these as were listed in the paper:

1. Old CW: Power is free, but transistors are expensive.
• New CW is the "Power wall": Power is expensive, but transistors are "free". That is, we can put more transistors on a chip than we have the power to turn on.
2. Old CW: If you worry about power, the only concern is dynamic power.
• New CW: For desktops and servers, static power due to leakage can be 40% of total power. (See Section 4.1.)
3. Old CW: Monolithic uniprocessors in silicon are reliable internally, with errors occurring only at the pins.
• New CW: As chips drop below 65 nm feature sizes, they will have high soft and hard error rates. [Borkar 2005] [Mukherjee et al 2005]
4. Old CW: By building upon prior successes, we can continue to raise the level of abstraction and hence the size of hardware designs.
• New CW: Wire delay, noise, cross coupling (capacitive and inductive), manufacturing variability, reliability (see above), clock jitter, design validation, and so on conspire to stretch the development time and cost of large designs at 65 nm or smaller feature sizes. (See Section 4.1.)
5. Old CW: Researchers demonstrate new architecture ideas by building chips.
• New CW: The cost of masks at 65 nm feature size, the cost of Electronic Computer Aided Design software to design such chips, and the cost of design for GHz clock rates means researchers can no longer build believable prototypes. Thus, an alternative approach to evaluating architectures must be developed. (See Section 7.3.)
6. Old CW: Performance improvements yield both lower latency and higher bandwidth.
• New CW: Across many technologies, bandwidth improves by at least the square of the improvement in latency. [Patterson 2004]
7. Old CW: Multiply is slow, but load and store is fast.

• New CW is the "Memory wall" [Wulf and McKee 1995]: Load and store is slow, but multiply is fast. Modern microprocessors can take 200 clocks to access Dynamic Random Access Memory (DRAM), but even floating-point multiplies may take only four clock cycles.

8. Old CW: We can reveal more instruction-level parallelism (ILP) via compilers and architecture innovation. Examples from the past include branch prediction, out-of-order execution, speculation, and Very Long Instruction Word systems.

• New CW is the "ILP wall": There are diminishing returns on finding more ILP. [Hennessy and Patterson 2007]

9. Old CW: Uniprocessor performance doubles every 18 months.

• New CW is Power Wall + Memory Wall + ILP Wall = Brick Wall. Figure 2 plots processor performance for almost 30 years. In 2006, performance is a factor of three below the traditional doubling every 18 months that we enjoyed between 1986 and 2002. The doubling of uniprocessor performance may now take 5 years.

10. Old CW: Don't bother parallelizing your application, as you can just wait a little while and run it on a much faster sequential computer.

• New CW: It will be a very long wait for a faster sequential computer (see above).

11. Old CW: Increasing clock frequency is the primary method of improving processor performance.

• New CW: Increasing parallelism is the primary method of improving processor performance. (See Section 4.1.)

12. Old CW: Less than linear scaling for a multiprocessor application is failure.

• New CW: Given the switch to parallel computing, any speedup via parallelism is a success.

#3 There is a lot of mention about kernels in terms of parallel computing nowadays, but Asanovic et. al discuss DWARFS. Surely a DWARF is like any other kernel?? Or is it? Explain the difference between the Asanovic et. al's concept of a DWARF and how this may be different from the concept of a kernel in relation to OpenCL or CUDA.

[4]

ANSWER/MARKERS GUIDELINES:

DWARFS are essentially a higher level abstraction that captures a pattern of computation and communication common to a class of important applications. This is significantly different to the concept of an OpenCL or CUDA kernel. An OpenCL or CUDA kernel is a specific implementation of an operation that runs on a processing device such as a GPU or FPGA – these kernels are very much implementations as opposed to an abstraction of a DWARF that represents a pattern of computation and communication.

Question 3: Parallel Computing and Benchmarking **[10 Total]**

These questions are based on the lectures.

#1 What is meant by the term 'golden measure' in the context of parallel computing? (as in its used in our pracs and lectures.)

[1]

ANSWER/MARKERS GUIDELINES:

The golden measure is a trusted solution that provides an accurate result. It is not necessarily an optimized. It is typically a sequential implementation and is used as a 'yard stick' to compare the accuracy of a parallelized / optimized solution and speedup of the solution.

#2 Amdahl's law is sometimes used as a defacto excuse for a parallel solution not giving the awesome speedups as might have been anticipated at the start of a project. What is Amdahl's law? What aspect of it would account for a parallel solution not being so awesomely fast as hoped? Provide the Amdahl's Law maximum speedup calculation where *f* is the fraction of parallel code and *n* is the number of processors.

[6]

ANSWER/MARKERS GUIDELINES: Amdahl's law is a formula for predicting the theoretical maximum speed-up for parallel processing using multiple processors compared to a sequential version. It is the sequential portion of a parallel program that is largely accountable (as per Amdahl's law) for constraining the possible speedup. The maximum speedup is:

$$\text{Speedup}_{\text{parallel}} = \frac{1}{(1-f) + \frac{f}{n}}$$

#3 In the lectures the concept of a benchmark suite was mentioned a few times. What exactly is a benchmark suite as opposed to just a benchmark? What does a benchmark suite comprise and why is it better than just a single benchmark (e.g. doing a matrix multiply to decide which platform performs best).

[3]

ANSWER/MARKERS GUIDELINES: A benchmark suite is a collection of benchmark programs. There are suites that represent a collection of characteristic processing operations for a particular application domain. Such a suite would comprise a number of functions that are commonly used in the type of application concerned. This provides a more holistic view on a processors anticipated performance for a particular application domain, as opposed to a single benchmark (e.g. matrix multiple) that just performs one function which could be an overly limited representation of the type of processing that may be performed.

Question 4: Processing considerations **[20 Total]**

#1 FLOPS and GFLOPS are useful measures, but typically a machine is not doing only floating point operations. Have a look back at the moving average filter in Q1. You can see there aren't really many floating point operations except for ones that are presumably in the function 'mean'. If the function mean was implemented as below:

```
float mean (vector<float> X) {
  float sum=0.0F;
  for (int i=0; i<X.length(); i++) sum+=X[i];
  return sum/X.length();
}
```

Explain how you could get an approximate FLOPS measure for the moving average filter program (remember M can change). You can assume that you've already converted the OCTAVE code into efficient C code. (Bear in mind that even though the machine might be able to do a sustained 10 GFLOPS, e.g. one FP op after another, we would like to figure out how many FLOPS it is doing).

[5]

ANSWER/MARKERS GUIDELINES: This question is meant as a problem solving task. There are various approaches that the student could suggest. One approach is to disassemble the code and count the number of floating point operations that are performed in the program, based on the speed of executing the instructions it can be calculated the proportion of FP operations. An approximate estimate can then be determined by timing the program and calculating the fraction of time spent performing FP operations, this fraction can then be converted to a FP/s value. A simpler approach is to have a counter in the code to count the number of FP operations and then divide by the time it takes to complete the program (e.g., in mean() it could add X.length() +1 to the FP ops counter (since there are X.length() FP additions plus the divide at the end). This means an overhead of 1 integer add for each execution of mean(), the amount of time that the FP counter is divided by could be reduced slightly to compensate for this overhead.

The following scenario applies to #2 to #3 below…

A sequential process, which operates on 10M 32-bit samples takes 3ms to complete. This speed was too slow for the application, which required the process to complete within 1.5ms. Accordingly, the engineering team went about converting to a parallel pipelined version. They redesigned the system to have a 4-stage pipeline, and after thorough testing determined the average run time for each stage and communication transfer time between stages. Each stage works on a block of 1M samples at a time, so to process the whole 10M input 10 x 1M blocks are fed into the pipeline. Their analysis results are shown in the table below (for stages working at max data block inputs of 1M). The data transfer between stages is done by the source stage and can be done concurrently while the next stage is busy, i.e. if stage *i* needs to transfers to stage *i+1* then it doesn't need to wait for stage *i+1* to complete; however it is designed so that the processing of a new input block does not start until its next stage has completed.

Answer the questions that follow…

Stage → Timing ↓	**1: Remove outliers**	**2: Normalize**	**3: FFT**	**4: Find max**
Computation Time	30 us	50 us	100 us	10 us

Data transferred to next step / final output	1 us	2 us	20 us	1 us

Note this is a software system, not a hardware solution that is clocked. A stage blocks if the next stage is still busy. When the pipeline is filling up the first stage can complete in

#2 How long does it take for the first 1M block to get through the pipeline (i.e. for it to go from stage 1 and out stage 4, assuming that blocks keep getting fed into the system). Show your working. You can draw a table or pipeline sketch to work it out.

[5]

ANSWER/MARKERS GUIDELINES: For this answer the student could do a table to see how much time it takes for the first block of 1M samples to get through the pipeline. Note that each pipeline stage takes the computation time + the data transfer overhead to complete i.e. stage 1 takes 31us to complete because it needs to send data on to the next stage. Similarly stage 4 takes 11 us because it needs to send data out before it can start processing the incoming data from stage 3. The table could look like:

31
31 52
31 52 120
31 52 120 11 (at this stage the first block has got through the pipleline)
The total time for this is thus 31 + 52 + 120 + 11 = 124 us
Note that as soon as the first input is through the system the entire system start slowing down a bit to 120 us per input because the stages before stage3 all get blocked up waiting for stage 3 to complete.

#3 How long does it take to process the whole 10M input? Have the engineers achieved their objective of completing the processing within 1.5ms? What is the speedup of the processing of the new parallel version over the previous sequential version (based on processing 10M of samples, you also do not need to account for any start-up delays or time involved in doing the sampling done, again you can use and show your tables or sketches to work out the answer).

[8]

ANSWER/MARKERS GUIDELINES: The table start above could just be continue on:

Input block					Time to complete this step (in us)
1	31				31
2	31	52			52
3	31	52	120		120 (pipeline full, blocked by slowest stage!)
4	31	52	120	11	120
5	31	52	120	11	120
6	31	52	120	11	120

7	31	52	120	11	120
8	31	52	120	11	120
9	31	52	120	11	120
10	31	52	120	11	120
-	31	52	120	11	120
-	-	52	120	11	120
-	-	-	120	11	120
-	-	-	-	11	11

Total time: 31 + 52 + 120x10 + 11 = 1294us

So to complete 10M samples it takes 1.294 ms which is comfortably within the 1.5ms limit.

#4 Nice easy question to end off with. Assume that OCTAVE can run threads which can have parameter: data send to thread, threadid=number of this thread (from 1 to nthreads) and nthreads is the number of threads. What data domain decomposition is used in the code snippet below? You can choose between: a) continuous, b) blocked, c) interlaced, d) interleaved / cyclic.

[2]

```
function thread_ileve (A,threadid,nthreads)
  global global_sum;
  sum = 0;
  for i=threadid:nthreads:length(A)
     sum = sum + A(i);
  end
  global_sum(threadid) = sum;
endfunction
```

ANSWER: This is interleaved / cyclic

END OF TEST

CHEET SHEET

```
int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void
*(*start_routine)(void*), void *arg);   // attr can be set to NULL for default attributes

void pthread_join(pthread_t thread_id, void **value_ptr);

void pthread_exit(void *value);

int pthread_kill(pthread_t thread_id, int sig);

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```