Test 1: Lectures 2 to 8

EEE4084F 2018-03-15

SOLUTIONS

1. **[20 marks total]**

(a) The tension is between embedded systems with their specialized hardware and the complexities involved in developing software for these systems on the one side, and the other side is high performance computing, application development. The tension is not so much between the two but now what they are both putting strain on computing development solutions – as they say "We argue that these two ends of the computing spectrum have more in common looking forward than they did in the past". They go on to say that "perhaps the biggest difference between the two targets is the traditional emphasis on real-time computing in embedded" in such that embedded needs to be just fast enough to meet the deadlines, but running faster tends to be add valuable to server computing (e.g. handling more transactions at a time, which may mean making more profit). But server applications are also becoming more real-time, e.g. streaming videos. In all, the demands for embedded computers and high performance computing are becoming closer (the pendulum, using the CH24 analogy which we are still to get to, being in the centre of both embedded/custom and general purpose). **[2 marks]**

(b) It generally depends on the application. If one is speaking general purpose, running applications like Excel and Word, then there's going to be the probably of getting lots of different data to/from the processing nodes and the limit of available shared memory (i.e. the memory that can be kept in main memory on a single computer). The type of processing and the access to data is likely to be a significant bottleneck: lots of different processes running each processing different data differently → probably fewer cores per processor (I would think max 100s); but if it is lots of the same processing on fairly closely collocated data → probably more cores the better (1000s of cores). **[7 marks]**

(c) DWARFs are classes of types of computation where membership in that class is defined by similarity in computation and data movement; individual DWARFs are computational kernels that implement an instance of the relevant class for a parallel computing application. DWARFs are essentially specified at a high level of abstraction to allow reasoning about their behaviour across a broad range of applications. DWARF are intended to behave in a specific predictable way, provided they are implemented correctly. This makes it easier to reason about the behaviour of the processing systems that is composed of DWARFs, without necessarily delving into the nitty gritty of the code. This could massively improve productivity. Traditional benchmarks, like linpack, test only a specific set of (mainly math) operations. This can be rather limited and not give a sufficiently encompassing view on how well a platform will behave for a particular application – benchmarking with DWARFs however is largely moving away from testing specific calculations and move towards analysing more complex types of behaviour that a computer system will be expected to do – particular classes of computation that will be carried out by the system. It is somewhat similar to comparing the ability of someone to add numbers to someone who can do a range of accounting tasks – e.g. someone able to add numbers quickly probably suggest one can be a good accountant,

but accounting is not all about adding numbers quickly, these things like the speed of being able to debit and credit accounts which would also be a desirable ability. **[7 marks]**

(d) Old Conventional Wisdoms (CW) refers to understanding and the ways to do things or the opinion of things that were used in the past. New CW is the understanding of new technologies or considerations relating to new practices, how things are done or starting to be done currently – they essentially new understandings (some may be tentative hypothesises) that have become, or are soon likely to become, conventional practices. **[4 marks]**

2. **[24 marks]**

(a) speedup = Ts/Tp = 10.56

Tp = 158 ms ms

Therefore Ts = speedup * Tp = 1668.48 **[4 marks]**

(b)

(i)

```
x = [1:2:6] .* 8;
midpoint(x)
```

gives:

```
ans = 16  (or 32 if not /2)
```

[4 marks]

(ii)

First I would decide how to partition the data, what operations to perform and then design the thread function. For *Thread_Count* threads and an array of *n* elements, I would partition the data into separate blocks that each thread will process – each thread would process *Thread_Count/n* data items in its block. The main thread could process any remaining elements (i.e. the remainder of *Thread_Count/n*). In order to get data to the thread a struct could be implemented for each thread which would pass a pointer to the array and be used by the thread to save its min and max values for its block. The main thread would then go through all these structs at the end of the program to find the min and max of these submin and submax values. The code below provides examples of how this program could be implemented. (NOTE: students are not expected to provide the whole program obviously, they could perhaps provide some code snippets e.g. the thread function and what the struct might look like if they pass a struct to the thread functions).

```
/*  SAMPLE SOLUTION TO TEST1 question 2, EEE4084F 2018
 */

// Includes
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <iostream>

using namespace std;

// Number of threads to create
int Thread_Count = 4;
int Data_Size = 1024;
```

```
// Structure used for passing parameters to a thread
struct THREADDATA {
    float* x;
    int    id;
    int    n;
    float  min;
    float  max;
};

// The thread function
void* Thread_midpoint ( void* p)
{
    // set up variables (which will probably become regsiters)
    THREADDATA* d = (THREADDATA*)p;
    int blocksize = d->n/Thread_Count;
    int istart    = d->id * blocksize;
    int iend      = istart + blocksize;
    float* x      = d->x;
    float xmin    = x[istart];
    float xmax    = x[istart];
    cout << "  id=" << d->id << ": " << istart << "-" << iend-1 << endl;

    // go through the loop and find the min and max
    for (int i = istart+1; i<iend; i++) {
        if (x[i] < xmin) xmin = x[i];
        if (x[i] > xmax) xmax = x[i];
    }
  d->min = xmin;
  d->max = xmax;
  return NULL;
}


/****************** MAIN - Entry point to program ******************/

int main()
{
    int j; // counter variables
    cout << "Min Max Program!" << endl;

    // Structures to store data about threads
    int        Thread_ID[Thread_Count]; // Structure to keep the tread ID
    pthread_t  Thread   [Thread_Count]; // pThreads structure for thread admin
    THREADDATA Thread_D [Thread_Count]; // data to pass/get from the threads

    // Create the data vector to process
    float Data [Data_Size];
    for (j = 0; j < Data_Size; j++) Data[j] = 100*(j+1);

    // Set up data for threads and spawn threads...
    for(j = 0; j < Thread_Count; j++){
            Thread_ID[j]   = j;
            Thread_D[j].x  = Data;
            Thread_D[j].id = j;
            Thread_D[j].n  = Data_Size;
            pthread_create(Thread+j, 0, Thread_midpoint, &Thread_D[j]);
            }

    // wait for all threads to complete
    for(j = 0; j < Thread_Count; j++) {
         if (pthread_join(Thread[j], 0)){
             printf("Problem joining thread %d\n", j);
             }
    }

    // go though and check the values
    float min = Thread_D[0].min;
```

```
    float max = Thread_D[0].max;
    for (j = 1; j < Thread_Count; j++) {
        if (min > Thread_D[j].min) min = Thread_D[j].min;
        if (max < Thread_D[j].max) max = Thread_D[j].max;
    }

    // do the remainder
    int blocksize = Data_Size/Thread_Count;
    int istart    = Thread_Count * blocksize;
    int iend      = Data_Size;
    cout << "istart = " << istart << " iend = " << iend << endl;
    for (j = istart; j < iend; j++) {
        if (min > Data[j]) min = Data[j];
        if (max < Data[j]) max = Data[j];
    }

    // No more active threads, so no more critical sections required
    cout << "All threads have completed\n";
    cout << " min = : " << min << " max= " << max << endl;
    cout << "Answer = " << (max-min)/2.0 << endl;

    return 0;
}
```

[12 marks]

iii)
It would at best go 4x faster, splitting the work 4 ways. But there is a fair bit of overhead of setting up the structs and creating the threads and then finding the min and max of the submin and submax values found by the threads. So The portion of code is about ½ threads and ½ surrounding code, but obviously the longest loops are inside the thread function. So for really big arrays it might exhibit a speed up of say 3.5 if you are lucky.

With my solution for an array of 1024^2 elements I get:

execution time : 0.022 s : 1 thread (on average for 4 runs)

execution time : 0.014 s : 4 threads (on average for 4 runs)

So this is technically a disappointing speedup of 1.6 (i.e. not even double, and increasing to even bigger sizes seems to make little improvement).

[4 marks]

3.

(a) A BCE is an abstracted code, it models available computing resources on a particular core and is used to contrast processing behaviour between different classes of processor, of example a processor with many small but not powerful cores to few big but powerful cores. It models how alternative processor designs may operate if work together or to contrast the drawbacks and advantages of different potential processor structures. The approach is to support multicore designers to assess performance according to a decisions around choices of cores to have on a single chip. **[6 marks]**

(b) Granularity (in technical computing terms) refers to the ratio of the amount of computation (usually measure in instructions per datum) to the amount of communication (transfer cost per datum). Fine grained is defined as relatively little processing for the amount of communication needed (or how much data each calculated result depends on). So clearly if a process does a lot of communication to compute a result then it would surely be fine-grained. **[4 marks]**

Bonus:

The lawnmower scenario was aimed to (metaphorically) illustrate the overhead for parallelism, particularly the amount of time that may be spent on setting up a system to do parallel processing (for example preparing the processing cores, shipping instructions to them etc.) all of which might end up diminishing the speed improvement that are practically achievable. **[+1 mark]**