Coláiste na Tríonóide, Baile Átha Cliath
Trinity College Dublin
Ollscoil Átha Cliath | The University of Dublin

Faculty of Engineering, Mathematics and Science

School of Computer Science and Statistics

Integrated Computer Science **Semester 2019**
Year 3

Concurrent Systems

Thursday 25th April 2019 **RDS Simmonscourt** **09.30 – 11.30**

Dr David Gregg

Instructions to Candidates:

- Answer 2 out of the 3 questions
- All questions are marked out of 50
- All program code should be commented, indented and use good programming style

Materials permitted for this examination:

- Calculator

1.

a. For many years perhaps the most visible aspect of parallel computing was super-computing. Super-computers are large parallel computers designed to numerically simulate physical systems such as weather systems, wind-tunnels, or the core of a nuclear reactor. However, today parallel computers are widely found in mobile embedded systems. What are the main constraints on parallel embedded systems as compared to large scale parallel computers? Please describe the type of parallel computer architectures that you believe might be most suitable for low-power embedded systems.

[10 marks]

b. Examine each of the following pieces of code. State if each individual piece of code can be vectorized using SSE. If not, state clearly why. If the code can be vectorized, use SSE intrinsics to vectorize it. Write a short note explaining the parallelization strategy on any code you vectorize.

```
/* code segment 1 */
void scale(float * array, float factor, float offset) {
    for (int i = 0; i < 1024; i++ ) {
            array[i] = (array[i] * factor) + offset
    }
}
```

[10 marks]

```
/* code segment 2 */
float sum(float * a, int size) {
    float sum = 0.0;
    for ( int i = 0; i < size; i++ ) {
            sum = sum + a[i];
    }
    return sum;
}
```

[10 marks]

```
/* code segment 3 */
int mem_compare(char * first, char * second, int size) {
    for ( int i = 0; i < size; i++ ) {
        if ( first[i] != second[i] ) {
            return 0;
        }
    }
    return 1;
}
```

[10 marks]

```
/* code segment 4 */
void multiply(float ** matrix, float * vec, float * result)
{
    for ( int i = 0; i < 4096, i++ ) {
        float sum = 0.0;
        for ( int j = 0; j < 4096; j++ ) {
            sum += vec[j] * matrix[i][j];
        }
        result[i] = sum;
    }
}
```

[10 marks]

2.

The histogram of an image can be used to measure the distribution of colours in the image. For a greyscale image – that is an image where all colours are different shades of grey – the darkness (or lightness) of a pixel in the image can be represented with a single number. This number is often represented by a single byte value between 0 and 255 inclusive. The histogram is an array of 256 integer values, with one entry for each of the 256 possible shades of grey. If, for example, there are 25 pixels in the image with a greyscale shade of 110, then histogram[110] will have the value 25.

Write a parallel routine to compute the histogram of a grayscale image using C and OpenMP. Your routine should have the following prototype:

```
int * compute_histogram(unsigned char ** image, int height, int width)
```

Where *image* is a two-dimensional array of bytes representing the pixels of a greyscale image, *height* and *width* are the dimensions of the image. Your routine should return a new array of 256 integers, containing the histogram of the image.

[30 marks]

Write a short commentary on your function explaining how it works, and why you believe it to be efficient. You should comment on the time and space complexity of each step of your solution, and also explain why you think it is likely to run efficiently on a modern multi-core architecture. If you choose to compute a separate histogram for each part of the image you should comment in particular on the complexity of combining the histograms.

[20 marks]

3.

a. Modern architectures commonly provide one or more atomic machine instructions that can be used to implement locks. One of these is the atomic compare-and-swap instruction. The following pseudocode shows the high-level behaviour of the atomic compare-and-swap instruction:

```
int compare_and_swap(int * address, int testval, int newval) {
    int oldval = * address;
    if (oldval == testval) {
            *address = newval;
    }
    return old_val;
}
```

Explain how this instruction can be used to implement locks on shared-memory parallel computers. [20 marks]

b. The following C code multiplies square matrices A and B and produces an output result:

```
void matrix_mul(float ** A, float ** B, float ** result, int size) {
    for ( int i = 0; i < size; i++ ) {
            for ( int j = 0; j < size; j++ ) {
                    float sum = 0.0;
                    for ( int k = 0; k < size; k++ ) {
                            sum = sum + A[i][k] * B[k][j];
                    }
                    result[i][j] = sum;
            }
    }
}
```

Consider the case where you are designing a processor to execute the above code and similar types of code, assuming that the arrays are large. You are trying to select which sort of cores might be most suitable for executing this type of code. Make the case for and/or against each of the following type of cores to include in your processor:

I. very long instruction word (VLIW) cores

II. vector processor cores

III. multithreaded/simultaneous multithreaded cores

IV. multicore processor with many in-order cores

V. groups of cores of the sort found in graphics processing units (GPUs)

[6 marks per type of core]