CS3014 – Concurrent Systems
2018 Exam Solutions

Question 1

a) Compare-and-Swap (CAS) is an atomic instruction used in multithreading to achieve synchronization. It compares the contents of a memory location with a given value, and only if they are the same, modifies the contents of that memory location to a new given value. Explain how this can be used to create a lock between threads.

```
bool compare_and_swap(int *accum, int *dest, int newval) {
     if (*accum == *dest) {
          *dest = newval;
          return true;
     } else {
          return false;
     }
}
```

Compare and swap is done in a single atomic operation. The atomicity guarantees that the new value is calculated based on up to date information; if the value has been updated by another thread in the meantime, the write would fail. The result of the operation indicates whether the write was performed or not.

Algorithms based around CAS typically read some key from a memory location and remember the old value. Using this old value a new value is then computed. Then, using CAS, where the comparison checks the location still being equal to the old value. If this is true, no other thread has updated the value and as a result it is okay to perform the write.

b) Examine and state whether each of the following blocks of code can be parallelised effectively using SSE (x86). If not, clearly state why. If the code can be parallelised, use SSE intrinsics to vectorise it. Write a short note about the parallelisation strategy you used.

```
// Code Segment #1
void compute(int* array, int SIZE) {
     // array is 16-byte unaligned address
     int i = 1;
     while (i < SIZE) {
          array[i] = array[i] + array[i - 1];
          i = i + 1;
     }
}
```

The above code cannot be vectorised as the array addition step depends on the result of the previous addition. This dependency stems all the way from a[0] up to a[SIZE-1]. For each I we are replacing array[i] with a sum of an old value which would also have been calculated.

```
// Code Segment #2
void compute(float* array, int SIZE, float multiplier) {
      for(int i=SIZE-1; i>=0; i--) {
          array[i] = (array[i] * multiplier) / SIZE;
      }
}
```
[10 marks]

The above code can be vectorised as there is no underlying dependency within each operation within the for loop. We can reduce the number of loops by a factor of 4 by extracting 4 x 32-bit float values at once into an SSE vector. We can then perform the multiplication and division on 4 elements of the array simultaneously and store the result from the four operations back into the array. The for loop must also be decremented by 4 on each loop to account for this.

```
// Old code
void compute(float *array, int SIZE, float multiplier){
  for(int i = SIZE-1; i>=0; i--){
    array[i] = (array[i] * multiplier) / SIZE;
  }
}

// Vectorised code
#include <xmmintrin.h>

void compute(float *array, int SIZE, float multiplier){
  // Decrease i by 4 every time (performing 4 x 32-bit float operations per loop)
  for(int i = SIZE-1; i>=0; i-4){
    // Load 4 x 32-bit floats into vector
    __m128 vector = _mm_load_ps(&array[i]);

    // Load multiplier into vector 4 times
    __m128 multiplier = _mm_set_ps1(multiplier)

    // Perform vector multiplication
    __m128 mul_result = _mm_mul_ps(vector, multiplier);

    // Load SIZE into vector 4 times
    __m128 divisor = _mm_set_ps1(SIZE);

    // Perform vector division
    __m128 div_result = _mm_div_ps(mul_result, divisior);

    // Store results back into array
    _mm_store_ps(array[i], div_result);
  }
}
```

```
// Code Segment #3
int compute(int* array, int SIZE) {
      // array is 16-byte aligned address
      int max = 0;
      for(int i=0; i<SIZE; i++) {
          if (array[i] > max) { max = array[i]; }
      }
      return max;
}
```

[10 marks]

The above code can be vectorised as there is no underlying dependency within each operation within the for loop. We can perform 4 max operations at a time thus reducing the for loop by a factor of 4. Once we have finished, we are left with a vector containing 4 possible maximums. We then extract the max element from this vector to get a final result.

```
// Vectorised code
#include <xmmintrin.h>

int compute(int* array, int SIZE){
  // Array is 16-byte aligned address
  int max;

  // Set max to [0, 0, 0, 0]
  __m128i maxval = _mm_setzero_si128();
  __m128i elements

  // Reduce four loop by 8 (4 * 32-bit integers)
  for(int i=0; i<SIZE; i+=4){

    // Load 4 elements
    elements = _mm_load_si128(&array[i]);

    // Get vertical max result between vectors
    maxval = _mm_max_epi32(elements, maxval);
  }

  // Last step, find single maximum of resultant vector
  for (int i = 0; i < 3; i++) {
      maxval = _mm_max_epi32(maxval, _mm_shuffle_epi32(maxval, maxval, 0x93));
  }

  // Extract result from vector
  _mm_store_ss(&max, maxval);

  return max;
}
```

```
// Code Segment #4
bool strcmp(char* string1, char* string2, int strlength) {
    for(int i=0; i<strlength; i++) {
      if (string1[i] != string[2]) {
           return false;
      }
    }
    return true;
};
```
[10 marks]

The above code can be vectorised as there is no underlying dependencies between the loops. A char is represented as a single byte (8-bits) which can be loaded into an SSE intrinsic (vector) and compared between the two vectors which should be the exact same if the strings are equal.

```
// Old code
bool strcmp(char* string1, char* string2, int strlength){
  for(int i=0; i<strlength; i++{
    if(string1[i] != sting2[i]){
      return false
    }
  }
}

// Vectorised code
#include <xmmintrin.h>

bool strcmp(char* string1, char* string2, int strlength){

  // Using SSE we can load 16 * 8-bit char's at a time
  for(int i=0; i<strlength; i+=16){

    // Load 16 chars from string1 and string2 into vectors
    __m128i string1Vector = _mm_load_si128((__m128i *) string1[i]);
    __m128i string2Vector = _mm_load_si128((__m128i *) string2[i]);

    // Compare vecotrs
    __m128i vcmp = _mm_cmpeq_epi8(string1Vector, string2Vector);

    // Extract result
    int result;
    _mm_store_si128((__m128i *)result, vcmp);

    // If strings are not equal, return false
    if(!result) return false
  }
}
```

Question 2

A function called max_matrix finds the position of the maximum element from a three dimensional matrix of floats. It returns the position of the maximum element in the form of the three coordinates designated as x, y, z.

a) Write a C program using OpenMP to parallelise operations for faster execution. Your code is expected to store the position of the maximum in an integer array of size 3.

```c
// Open MP optimised code
void maximum2(int depth, int width, int height, int matrix[depth][width][height]){

  // Initialise result 0th index as maximum
  int result[3] = {0, 0, 0};
  int curMax = 0.0;
  int i,j,k;

  // Loop over array comparing max element
  #pragma omp parallel for private(i, j, k) shared(matrix, result, curMax) collapse(3) if(width * height * depth > 3000)
  for(int i=0; i<width; i++){
    for(int j=0; j<height; j++){
      for(int k=0; k<depth; k++){

        // Do max comparsion here
        int val = matrix[i][j][k];

        // If new max, update
        #pragma omp critical
        {
          if(val > curMax){
            curMax = val;
            result[0] = i;
            result[1] = j;
            result[2] = k;
          }
        }
      }
    }
  }

  printf("\n\nMax2: %d", curMax);
  printf("\nIndex2: [%d][%d][%d]", result[0], result[1], result[2]);
  return;
}
```

b) Describe your approach towards parallelisation and considerations regarding simultaneous execution of code. Write about underlying assumptions you have made, and how the number of CPI cores or threads will affect its execution. You should provide a reasonable estimate of the time-complexity of your solution.

The above solution uses OpenMP to divide the three for loops among multiple threads allowing them to be executed completely in parallel. Unfortunately, there is an underlying race condition where each of the threads needs to compare the value they are checking with the current maximum. The value of *curMax* could be replaced just after they have performed the comparison and the thread could falsely overwrite the value of *curMax*.

In order to prevent this race condition from occurring I made use of OpenMP's ***critical*** flag which puts a lock on all code within it only allowing one thread to execute this code at one given time. This will prevent threads from falsely replacing the current max value.