# Final exam - IN3200

Candidate nr: 15129
(Dated: June 11, 2020)

## PROBLEM 1A

The table below shows the distribution of work which yields the minimum amount of time. This turned out to be 11 hours.

| $W_1$ | $W_2$ | $W_3$ | $T$ |
|---|---|---|---|
| $T_{11}$ | | | 1 |
| $T_{21}$ | $T_{12}$ | | 2 |
| $T_{31}$ | $T_{22}$ | $T_{13}$ | 3 |
| $T_{41}$ | $T_{32}$ | $T_{23}$ | 4 |
| $T_{51}$ | $T_{42}$ | $T_{33}$ | 5 |
| $T_{14}$ | $T_{52}$ | $T_{43}$ | 6 |
| $T_{24}$ | $T_{15}$ | $T_{44}$ | 7 |
| $T_{34}$ | $T_{25}$ | $T_{54}$ | 8 |
| $T_{44}$ | $T_{35}$ | | 9 |
| $T_{54}$ | $T_{45}$ | | 10 |
| $T_{55}$ | | | 11 |

TABLE I.

## PROBLEM 1B

Once the first task in each column is opened, there are 5 concurrent collection of tasks that can be performed in parallel. Thus 5 workers will be the optimal choice. The following table shows the mimimum time this will take, which became 9 hours.

| $W_1$ | $W_2$ | $W_3$ | $W_4$ | $W_5$ | $T$ |
|---|---|---|---|---|---|
| $T_{11}$ | | | | | 1 |
| $T_{21}$ | $T_{12}$ | | | | 2 |
| $T_{31}$ | $T_{22}$ | $T_{13}$ | | | 3 |
| $T_{41}$ | $T_{32}$ | $T_{23}$ | $T_{14}$ | | 4 |
| $T_{51}$ | $T_{42}$ | $T_{33}$ | $T_{24}$ | $T_{15}$ | 5 |
| | $T_{52}$ | $T_{43}$ | $T_{34}$ | $T_{25}$ | 6 |
| | | $T_{53}$ | $T_{44}$ | $T_{35}$ | 7 |
| | | | $T_{54}$ | $T_{45}$ | 8 |
| | | | | $T_{55}$ | 9 |

TABLE II.

The maximum achievable speedup is thus

$$S = \frac{9}{25}. \tag{1}$$

## PROBLEM 2

It's sufficient to insert a few pragma directives. To avoid false sharing, we must make $i$ and $j$ private.

```
#pragma omp parallel for private(i)
  for (i=2; i<N; i++)
```

and

```
#pragma omp parallel for private(i,j)
  for (i=2; i<=sqrt_N; i++) {
    if (array[i]) {
      for (j=i*i; j<N; j+=i)
      array[j] = 0;
    }
  }
```

However, due to the fact that the array is shared, we experience false sharing as a result in such a way that the array must be sent back and forth among the threads private cache due to cache coherence policy. Also, threads that receive $i$'s that are close to $\sqrt{N}$ will end up not entering the if-test since the array element they access may already be set to 0, so they will simply remain idle. One could instead parallelize the inner-loop of course, and here the workers would not stay idle. However, false sharing will also be a problem here. Therefore I don't expect any significant speedup as a function of number of threads either way.

## PROBLEM 3A

Since the 2D-arrays will be access in row-major order, it is wisest to distributed rows among the threads. Furthermore, the outer-most loop is much longer than the inner-most ones, so the outer-most loop will bear more fruits if it is parallelized. This way, we will also avoid false sharing since the threads access separate rows of the 2D-arrays mean the they do not load the same cache lines into their private cache and thus no cache coherence policy will kick in.

```
void sweep(int N, double **table1, int n, double **mask, double **table2){
  int i, j, ii, jj;
  double temp;
  #pragma omp parallel for private(i, j, ii, jj, temp)
  for (i=0; i<=N-n; i++)
    for (j=0; j<=N-n; j++) {
      temp = 0.0;
      for (ii=0; ii<n; ii++)
        for (jj=0; jj<n; jj++)
          temp += table1[i+ii][j+jj]*mask[ii][jj];
      table2[i][j] = temp;
    }
}
```

## PROBLEM 3B

The code balance can be computed as follows. There are $2(N-n+1)^2n^2$ loads and Flops as well as $(N-n+1)^2$ stores. Let $L$ denote loads and stores and $F$ denote Flops. The code balance is then (divide by 8 to get bytes to words)

$$B_c = \frac{L}{F} = \frac{2(N-n+1)^2n^2 + (N-n+1)^2}{2(N-n+1)^2n^2} = \frac{2n^2+1}{2n^2} \tag{2}$$

which for relatively large values of $n$ approaches $B_c = 1$. To estimate the machine balance of the system, I'll use that the processor has 48 cores, can do 4 flops per cycle and has a base frequency of 2.7 GHz. Its peak memory bandwidth is roughly $B \approx 120$ GB/s $= 15$ Words/s . The peak performance $P$ is based on the information above given by

$$P = 2.7 \cdot 4 \cdot 48 \text{ GFlops/s} \approx 518 \text{ GFlops/s} \tag{3}$$

The machine balance is then

$$B_m = \frac{B}{P} = \frac{15}{518} \approx 0.03 \tag{4}$$

so in general $B_m \ll B_c$. This implies that the performance bottle-neck is bound by memory traffic. Assuming no latency, we can simply count how many loads and stores $L$ we need and divide by the bandwidth $B$ and number of cores $C = 48$ (which is a simplification that probably doesn't reflect reality particularly well).

$$T = \frac{L}{CB} = 8 \times \frac{2(N-n+1)^2 n^2 + (N-n+1)^2}{48 \times 120 \times 10^9} = \frac{(N-n+1)^2\left(2n^2+1\right)}{720} \times 10^{-9} \text{ sec} \tag{5}$$

## PROBLEM 4A

To be able to store information about the total population, a compressed row-storage format can be used to store information about which people meet. There are two good reasons for this: Firstly, the matrix will be a sparse one since most people won't meet at all. Furthermore, compressed row-storage greatly reduces the memory footprint of the simulation. This format is used to represent a matrix, which I'll call the interaction matrix denoted by $A_{ij}$. The interaction matrix will only contain 0's and 1's where $A_{ij} = 1$ if person $i$ and $j$ meet and otherwise is zero. To store this information it suffices to use a row pointer that keeps track of how many elements there are on each row of the matrix, and a column index pointer to store which position in a given row the value is 1. A separate pointer with length equal to the population size can be used to store information about a persons state, where one might use three different integers to represent a persons state, say 0 for healthy, 1 for sick and 2 for immune. These can be stored as chars to reduce memory footprint as well.

## PROBLEM 4B

The code below takes a row pointer of length $N+1$ and a column index pointer that points to which person a given person (that is, a row) meets. The function takes two pointers of length $N$ to store the state of yesterday and today where they initially contain all the same values. Furthermore the function takes the infection probability $f$ and the population size $N$. The comments in the code explains what and why a given part of the code is the way it is.

```
void advance_one_day(int *row_ptr, int *col_idx, int *state_old, int *state_new, double f, int N)
{
    int person1, person2, person1_state, person2_state;
    double u;
    int i,j;
    int max_int = 1000000;
    double max_inv = 1./max_int;
    //Iterate over every person i in the population
    for (i = 0; i < N; i++){
        person1 = i; //Extract ID of person 1
        person1_state = state_old[person1]; //Extract state of person 1

        //Only check the interactions if the person is not immune which is denoted with state = 2.
        if (person1_state != 2){
            //The matrix is symmetric, so we only want to check interactions below the diagonal
                which corresponds to col_idx[j] < i.
            for (j = row_ptr[i]; j < row_ptr[i+1] && col_idx[j] < i; j++){
                person2 = col_idx[j]; //Extract id of person 2
                person2_state = state_old[person2]; //Extract state of person 2

                //If person 1 is sick, check if person 2 gets infected.
                if (person1_state == 1 && person2_state == 0){
                    u = (double) (rand() % max_int)*max_inv;
                    if (u < f){
                        state_new[person2] = 1; //Person 2 gets infected.
                    }
                }
                //If person 2 is sick, check if person 1 gets infected.
                if (person1_state == 0 && person2_state == 1){
                    u = (double) (rand() % max_int)*max_inv;
                    if (u < f){
                        state_new[person1] = 1; //Person 1 gets infected.
                    }
                }
            }
        }
```

```
    }
    //Copy all the values from states_new to states_old.
    for (i = 0; i < N; i++) state_old[i] = state_new[i];
}
```

## PROBLEM 4C

For several values of $T$ and $f$, we could run different simulations in parallel where each core/thread receives different values of $T$ and $f$ and runs a simulation for a prescribed set of days. This way the workload will be fairly distributed. We could instead loop over several values of $T$ and $f$ and parallelize the actual workload, but since the actual number of people a given person meets varies, this may result in uneven workload distribution such that some threads sit idle while others do work. Therefore running several simulations in parallel for various values of $T$ and $f$ will likely yield a more efficient implementation. One could possibly use MPI to distribute various values of $T$ and $f$ to different cores, and then use OpenMP to parallelize the simulation each core has to compute.

## I. PROBLEM 5

With NUMA, it's simpler to use MPI if no or little communications among the processes are required since false sharing cannot occur with MPI. If one uses OpenMP and cache coherence occurs, cache lines may have to be sent back and forth among processors. But since communcations among processors in a NUMA system is generally much slower than in UMA, this would significantly slow down the process. For instance, running the parallel code suggested in problem 4c may work great on a NUMA system with MPI since each simulation for various values of $T$ and $f$ would be independent of each others and thus practically no communication between processes is necessary.