UNIVERSITY OF OSLO

Faculty of mathematics and natural sciences

Examination in IN3200/IN4200 — Exam questions and suggested solutions

Day of examination: June 11th-18th, 2020

This problem set consists of 9 pages.

Appendices: None

Permitted aids: Any

Please make sure that your copy of the problem set is complete before you attempt to answer anything.

This home exam accounts for 60% of the final grade

The first and second home exams account for 20% each. The final grade will be pass or no pass. **Each student should independently answer the following questions by her/himself.** All the answers should be contained in one PDF file. Please also read the additional info given at Inspera.

Problem 1 (weight 10%)

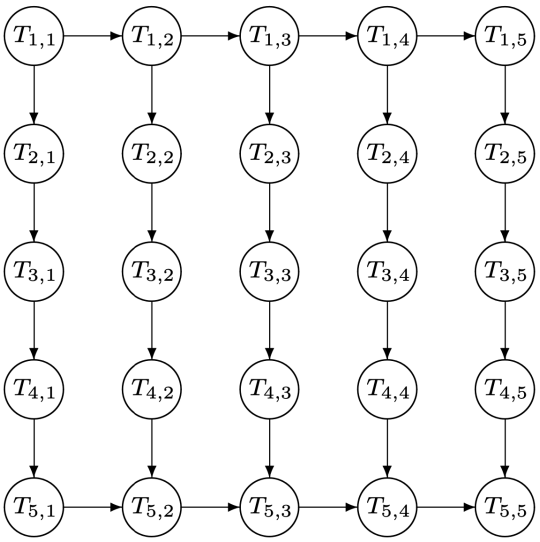

Figure 1: The dependency relationship between 25 tasks.

Figure 1 shows the dependency relationship between 25 tasks: $T_{1,1}, T_{1,2}, \ldots, T_{5,5}$. Each directed edge in the graph connects a pair of "source" and "destination" tasks.

(Continued on page 2.)

A destination task cannot be started until all its source tasks are carried out. All the 25 tasks are equally time-consuming, requiring one hour of a worker. (We also assume that it is not possible to let two or more workers collaborate on one task for a faster execution.)

1a (weight 5%)

Explain in detail how many hours minimum do three workers need to finish all the 25 tasks.

Suggested solution: The following table shows one possible fastest solution that involves three workers. (There are other solutions that will give the same shortest time usage.)

	Worker 1	Worker 2	Worker 3
hour 1	$T_{1,1}$		
hour 2	$T_{2,1}$	$T_{1,2}$	
hour 3	$T_{3,1}$	$T_{2,2}$	$T_{1,3}$
hour 4	$T_{4,1}$	$T_{3,2}$	$T_{2,3}$
hour 5	$T_{1,4}$	$T_{5,1}$	$T_{4,2}$
hour 6	$T_{3,3}$	$T_{2,4}$	$T_{1,5}$
hour 7	$T_{5,2}$	$T_{4,3}$	$T_{3,4}$
hour 8	$T_{2,5}$	$T_{5,3}$	$T_{4,4}$
hour 9	$T_{3,5}$	$T_{5,4}$	
hour 10	$T_{4,5}$		
hour 11	$T_{5,5}$		

1b **(Only relevant for IN3200)** (weight 5%)

What is the maximum speedup that can be achieved? How many workers are needed to achieve the maximum speedup? Please explain your answers.

Suggested solution: There are in total 9 "wavefronts", meaning that the fastest solution time (with a sufficient number of workers) is 9 hours. Thus, the maximum speedup is

$$\frac{25}{9}$$

To achieve this, five workers are needed (so that there is no delay in executing any of the wavefronts).

1c **(Only relevant for IN4200)** (weight 5%)

Now, let us extend the dependency relationship graph to have N rows and N columns. The dependency relationship between the N^2 tasks will have the same pattern as in Figure 1. That is, each vertical task column has the downward dependency, while the top and bottom horizontal task rows have the right-ward dependency.

(Continued on page 3.)

Please derive a formula for the minimum time usage needed by P workers. (You can assume that P is no larger than N.)

Suggested solution: It is important to notice that all the tasks on the same "wavefront" can be executed in parallel, provided that the previous wavefront is done. There are in total $2N-1$ wave fronts, each having at most N tasks. So, for the special case of $P=N$, a total number of $2N-1$ hours will be needed.

For all the other cases of $P < N$, the first and last P wave fronts ("wind-up" & "wind-down" phases) will need in total $2P$ hours to execute these $P(P+1)$ tasks. The remaining $N^2 - P(P+1)$ tasks thus require

$$\lceil \frac{N^2 - P(P+1)}{P} \rceil$$

hours. Therefore, the total number of hours needed by P workers is

$$2P + \lceil \frac{N^2 - P(P+1)}{P} \rceil = \lceil \frac{N^2}{P} \rceil + P - 1.$$

Problem 2 (weight 10%)

How would you parallelize the following code segment using OpenMP directives? Please provide sufficient explanations. Also, what are your opinions about the speedup that can be achieved as the number of threads is increased?

```
int i,j, sqrt_N;

char *array = malloc(N);  // N is a predefined very large integer
array[0] = array[1] = 0;
for (i=2; i<N; i++)
  array[i] = 1;

sqrt_N = (int)(sqrt(N));  // square root of N
for (i=2; i<=sqrt_N; i++) {
  if (array[i]) {
    for (j=i*i; j<N; j+=i)
      array[j] = 0;
  }
}

free (array);
```

Suggested solution: The above code implements the famous "Sieve of Eratosthenes" (`https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`), which can be used to identify all the prime numbers up to $N-1$.

(Continued on page 4.)

Obviously, OpenMP directive `#pragma omp parallel for` can be added before the first for-loop for the initialization of `array`. For the nested double for-loop, OpenMP directive `#pragma omp parallel for` can be added before the j-indexed inner for-loop, because these iterations are completely independent of each other. It is incorrect to parallelize the `i`-indexed outer for-loop, because of the risk of excessive unnecessary work and race conditions.

The obtainable speedup is mainly affected by two factors: (1) The realistically achievable memory bandwidth by multiple threads; (2) The OpenMP overhead related to distributing the `j`-indexed iterations among the threads. Note that the aggregately achievable memory bandwidth will *not* grow linearly with the number of threads, thus the obtained speedup will eventually saturate (and perhaps drop), whereas the NUMA architecture may spoil it further. To limit the OpenMP overhead, we can wrap the `i`-indexed loop inside a single OpenMP parallel region, while making `i` private per thread. (Still, only the `j`-indexed iterations are parallelized.)

Problem 3 (weight 15%)

Consider the following function

```
void sweep (int N, double **table1, int n, double **mask, double**table2)
{
  int i,j,ii,jj;
  double 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;
    }
}
```

All the three 2D arrays `table1`, `mask` and `table2` are allocated beforehand, where `table1` is of dimension $N \times N$, `mask` of dimension $n \times n$, and `table2` of dimension $(N-n+1) \times (N-n+1)$. It can be assumed that N is much larger than n. (For example, N is at least 10000 whereas n is at most 10.)

3a (weight 5%)

Show how OpenMP directives can be used to parallelize the function `sweep`. Please provide necessary explanations.

Suggested solution: OpenMP directive `#pragma omp parallel for private(j, ii, jj, temp)` (possibly also with a `collapse(2)` clause) can be added before the

(Continued on page 5.)

outmost for-loop, because all the `i`-indexed iterations are independent of each other. This will induce the least amuont of OpenMP overhead, is thus a good parallelization strategy provided that N is large enough with respect to the number of threads.

3b (weight 10%)

Given a computer with two Intel Xeon 24-core processors of model 8168 (`https://ark.intel.com/content/www/us/en/ark/products/120504/intel-xeon-platinum-8168-processor-33m-cache-2-70-ghz.html`). How would you estimate the theoretical minimum computing time needed by the function `sweep` when the values of N and n are known? Please elaborate your reasoning.

Only relevant for IN4200: In reality the actual computing time is longer than the theoretical minimum estimate. What can be the reasons for this performance discrepency?

Suggested solution: The main idea here is to derive two theoretical estimates $\mathrm{Time}_{\mathrm{fp}}$ and $\mathrm{Time}_{\mathrm{memory}}$, where the former looks at how fast the involved floating-point operations can be excuted, the latter considers the time neeed for the associated memory traffic (between main memory and last-level cache, as the usual suspect of bottleneck). Then, the theoretical minimum computing time is simply

$$\max\left(\mathrm{Time}_{\mathrm{fp}}, \mathrm{Time}_{\mathrm{memory}}\right)$$

(The above strategy is equivalent with the “balance analysis” and "lightspeed estimates" which are discussed in Chapter 3 of the textbook.)

The total number of floating-point operations executed inside the `sweep` function is $2(N-n+1)^2n^2$. So, the minimum time needed for executing the floating-point operations, using P cores per CPU, is

$$\mathrm{Time}_{\mathrm{fp}} = \frac{2(N-n+1)^2n^2}{2P \cdot 4 \cdot \text{CPU clock rate}} \quad \text{or} \quad \mathrm{Time}_{\mathrm{fp}}^{\mathrm{SIMD}} = \frac{2(N-n+1)^2n^2}{2P \cdot 32 \cdot \text{CPU clock rate}}$$

The factor of “4” in the denominator of $\mathrm{Time}_{\mathrm{fp}}$ is because each Intel Xeon Scalable core is theoretically capable of 4 floating-point operations per clock cycle. The factor of “32” in the denominator of $\mathrm{Time}_{\mathrm{fp}}^{\mathrm{SIMD}}$ is assuming full 512-bit SIMD vectorization in addition. (In reality, a compiler will achieve some level of vectorization, but not perfectly. Also, to achieve full SIMD vectorization, the CPU clock rate will be lowered.) So, for simplicity we don’t consider SIMD vectorization further here.

The amount of data traffic between the main memory and each of the two last-level (L3) caches is $(N-n+1)^2/2$ words stored (each word 8 bytes) for array `table2`, n^2 words loaded for array `mask`, assuming it resides in L3 throughout the computation. Moreover, minimum $N^2/2$ words are loaded for array `table1` per L3 cache. This assumes “perfect caching”, which means that if P OpenMP threads are concurrently in work per CPU, the L3 cache must at least have space to hold $P \cdot n \cdot N/2$ words of array `table1`, plus the space needed for array `mask`, as well as P cachelines in connection with storing the values of array `table2`.

(Continued on page 6.)

In the case of each L3 cache (of capacity C words) not being large enough, we can roughly estimate the number of times each value of `table1` will be reused in cache as

$$r = \lfloor \frac{C - n^2 - (P \cdot \text{words per cacheline})}{P \cdot N} \rfloor$$

Then, the actual amount of words loaded for array `table1` will be $(n - r + 1) \cdot N^2$ words.

Then, we can derive the following theoretical estimate of the minimum time needed for the memory traffic:

$$\text{Time}_{\text{memory}} = \frac{((N - n + 1)^2 + 2n^2 + (n - r + 1) \cdot N^2) \cdot 8\,\text{bytes}}{2\,\text{CPUs} \cdot 6 \cdot \text{memory frequency} \cdot 8\,\text{bytes}}$$

(The factor of "6" in the denominator is due to the six memory channels of a Xeon Scalable processor.)

The minimum time needed to execute the `sweep` function is thus the maximum value between Time_{fp} and $\text{Time}_{\text{memory}}$. It should be remarked that using fewer than 24 threads per CPU *can* be beneficial because this may increase the value of r, thus decrease memory traffic related to loading array `table1`.

(Only relevant for IN4200:) The reasons of having the actual computing time different from the above theoretical estimate can include (1) the realistic memory bandwidth achievable per CPU is considerably less than $6 \cdot$ memory frequency $\cdot\, 8$ (as can be found by the STREAM benchmark); (2) the L3 cache is not "perfectly" utilized (such as due to the m-way associativity); (3) the actual amount of memory traffic is larger than the theoretical estimate due to imperfect alignment (and/or imperfect use of the cache lines); (4) loop overhead associated with the `ii`- and `jj`-indexed loops; (5) imperfect load balancing between the threads.

.

Problem 4 (weight 20%)

We aim to design a *simplistic* simulator for the day-to-day spread of a virus among a large number of people. (**Note:** There is no need to write a complete program. Code segments and/or high-level pseudo code, with clear explanations, are sufficient.) The following are some assumptions to be used by this very simple simulator:

- The interactions among the people remain the *same* every day. (That is, if two people meet each other today, then they will also meet tomorrow, the day after tomorrow and so on; unless one of them is so ill to have to stay at home.)
- The *fixed* person-person daily interactions are described by a graph, where the people are represented by vertices, and an edge between a pair of vertices means that the corresponding two people have a long enough daily interaction that bears the risk of transmitting the virus in-between (if one of them is ill).
- No person is initially immune to the virus.
- Once infected, a person *cannot* immediately infect other people on the same day of the infection. Starting from the next day until day T (an input parameter to the simulator), the infected person becomes ill and has the potential of infecting other

(Continued on page 7.)

healthy, non-immune people with whom she/he interacts. After day T, the infected person is so ill that he/she will stay at home and thus without the possibility of infecting more people.

- An ill person will become healthy after staying at home for, say, X days. A recovered person is assumed to be immune to the virus, will thus not be infected again.
- If a healthy, non-immune person interacts with an ill person (who has been infected for at least one day), the probability of being infected is f every day until the ill person starts to stay at home. (The value of f is a prescribed percentage, as an input parameter to the simulator.)

4a (weight 5%)

Please describe in detail the data structure you will use to store the person-person interaction graph, and the healthy/ill/immune states of all the people. Please motivate your design by efficiency considerations if applicable.

Suggested solution: Let N denote the total number of people, and each person has a unique integer ID (between 0 and $N-1$). The most economical data structure for representing the person-person interactions is to use the `irow` and `jcol` arrays of the CSR format (please refer to the first home exam). It is important to represent the symmetry of person-person interactions in the array `jcol` (which can be 50% seemingly wasteful, but essential for the efficiency of the subsequent day-to-day evolvement simulation). Note that using a 2D table to store the person-person interactions is prohibitively expensive. Using a linked list to store the person-person interactions is memory-wise viable, but not efficient in operation.

Moreover, an integer array of length N can be used to store the current status of each person. For example, a status value of "-1" means a person is healthy and non-immune, a status value of "0" means a person is just infected. A positive status value will mean the number of days a person has been ill. Finally, a status value of "-100" can be used to indicate a person was ill but is now healthy and immune. The array of status values will be updated from day to day (by the function to be outline in Task 4b).

Last but not least, a very short array, which is also updated from day to day, will be used to monitor the current list of people who have a non-negative status value.

4b (weight 10%)

Please sketch a function that can be used to evolve the healthy/ill/immue states of all the people by one day. (For example, the inputs can include the person-person interaction graph and the healthy/ill/immune states of all the people from the previous day. The output can be the healthy/ill/immune states of all the people for the present day.) Please motivate your implementation sketch by efficiency considerations if applicable.

Hint: It is standard practice to use a random number generator (such as the standard function `rand`) to simulate whether a healthy, non-immune person is to be

(Continued on page 8.)

infected by an ill person, while statisfying the prescribed probability f.

Suggested solution: The function can be roughly sketched as follows

```
for (i=0; i<num_people_currently_being_monitered; i++) {
  j = monitered_array[i];  // integer ID of the person being monitored

  if (status[j]>0 && status[j]<T) {
    for (k=irow[j]; k<irow[j]; k++)
      if (status[k]==-1) {
        // draw a random number between 0 and 1
        // ...
        if (random_number<f) {
          status[k] = 0;
          add person k to the list of people currently being monitored
        }
      }
   }

   status[j]++;

   if (status[j]>=T+X) {
     status[j] = -100;
     remove person j from the list of people currently being monitored
   }
}
```

4c (weight 5%)

In order to study different scenarios and collect statistics, we will need to run a large number of simulations, which differ by the T value, and/or the f value, and/or who and how many the initially infected people are. Please present your high-level ideas about how parallel computing can be applied to efficiently run the large number of simulations. (There is no need to show the programming details.)

Suggested solution: The above sketch of the function is not suited for OpenMP parallelization due to race conditions (and also because the number of infected people is assumed to be small, thus not much work to be shared among threads).

For multiple simulations using the same person-person interactions (for studying different initial conditions, different values of f and/or T, etc.), OpenMP parallelization is convenient because the `irow` and `jcol` arrays can be shared. Each thread will otherwise be completely independent from the other threads, having its own status array and list of people currently being monitored.

For the purpose of simulating different person-person interaction graphs, MPI paralleliztion is most natural. Here, different MPI processes will independently work

(Continued on page 9.)

with different person-person interaction graphs. No inter-process communication is needed during the simulations.

Problem 5 (weight 5%)

Please present your thoughts (in your own words) about the advantages/disadvantages of MPI programming versus OpenMP programming for a computer with multiple sockets of multicore CPUs (that is, a shared-memory system with NUMA architecture).

Suggested solution: Some important points can be briefly mentioned as follows (the list is not exhaustive, but it is not necessary to mention all the points to get full score):

- MPI will lead to extra memory usage (due to layers of "ghost values", duplicated variables/arrays, MPI internal memory usage) ;
- MPI programming normally requires explicit work/data division (definitely extra programming effort required and possibly some overhead in this regard);
- Programming message exchanges requires coding effort (data exchange in OpenMP is implicitly enforced);
- MPI programming must handle the risk of deadlocks with respect to messaging;
- MPI programming gives the programmer better control (with regard to work division, for example);
- Data locality (no issues of NUMA, race condition, false sharing) is better in MPI programming than OpenMP programming;
- Shared cache may benefit OpenMP code, but not MPI code;
- OpenMP programming must consider NUMA issues (proper first touch needed);
- False sharing can happen with OpenMP code;
- Race conditions can happen with OpenMP code.