# Nytt dokument

# Exam in INF3380 spring semester 2018

**Assistance: One two-sided A4 page with hand-written notes, plus a calculator. No other assistance is allowed.**

**All the exam questions should be answered with keyboard and mouse. No need to use sketching paper.**

Weighting of questions:
Questions 2.1, 2.2 (Processing inhomogeneous, independent tasks): 10%
Questions 3.1, 3.2 (Processing homogeneous, dependent tasks): 10%
Questions 4.1, 4.2 (All-to-all broadcast): 15%
Questions 5.1, 5.2, 5.3 (OpenMP): 30%
Questions 6.1, 6.2, 6.3 (Sorting): 35%

## 2.1 Processing inhomogeneous, independent tasks; two workers

There are in total 20 independent tasks. These tasks however require different time usages. More specifically, task No. 1 requires one hour's of a worker, task No. 2 requires two hours, task No. 3 requires three hours, and so on. (We assume that it is not possible to let two or more workers collaborate on one task for a faster execution.) If there are two workers, how many hours at least do they need to finish all the 20 tasks? Please detail your answer.

**Fill in your answer here**

Format

Words: 0

Maximum marks: 5

## 2.2 Processing inhomogeneous, independent tasks; three workers

If there are now three workers, how many hours minimum do they need to finish the 20 tasks? Please detail your answer.

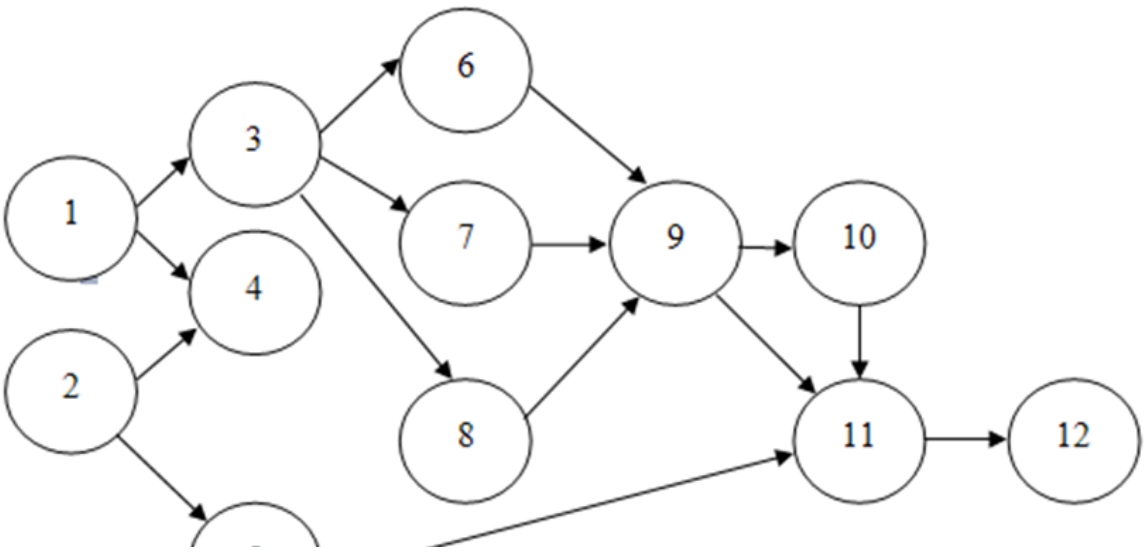
**Fill in your answer here**

Format

Words: 0

Maximum marks: 5

## 3.1 Processing homogeneous, dependent tasks; two workers




Above is a task-dependency graph showing how 12 tasks depend on each other. Each directed edge in the graph connects a pair of "source" and "destination" tasks. A detination task cannot be started until all its source tasks are carried out. All the 12 tasks are equally time-consuming, requiring one hour of a worker. (The digits in the circles show the numbering of the tasks. We also assume that it is not possible to let two or more workers collaborate on one task for a faster execution.)

If there are two workers, how many hours minimum do they need to carry out all the 12 tasks? Please detail your answer.

**Fill in your answer here**

Format | Σ

Words: 0

Maximum marks: 5

## 3.2 Processing homogeneous, dependent tasks; three workers

We continue with the same task-dependency graph. If there are now three workers, how many hours minimum do they need? Please detail your answer.

**Fill in your answer here**

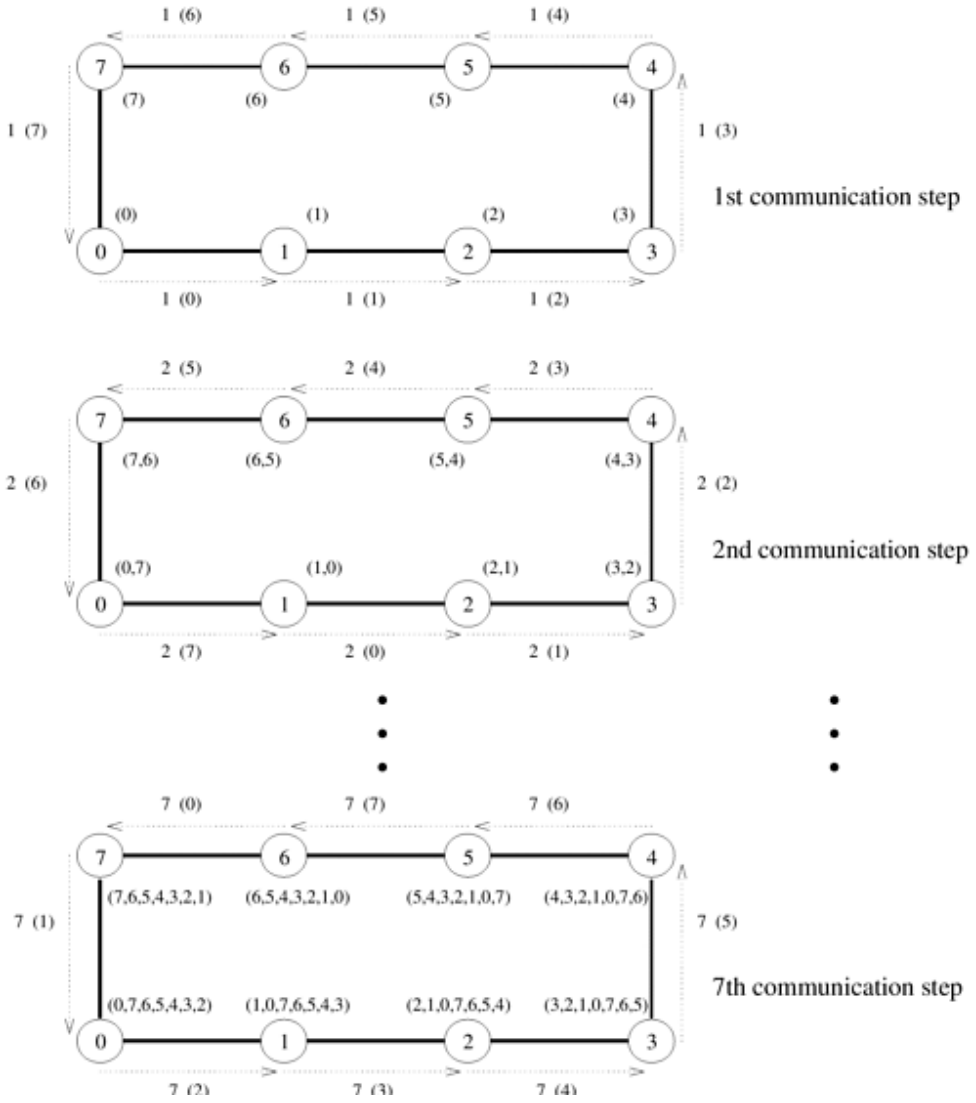
Format

Words: 0

Maximum marks: 5

## 4.1 All-to-all broadcast on a 2D torus

Here we have an exmaple of how all-to-all broadcast can be executed, step by step, on a 1D ring of 8 nodes.

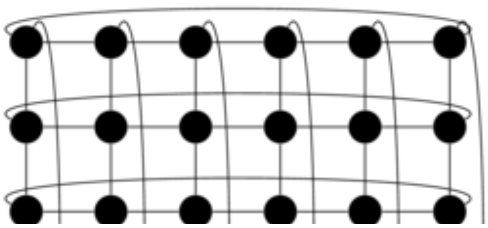


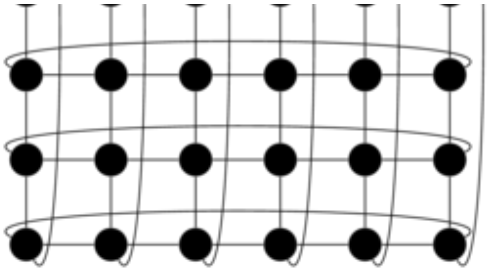
Please extend the idea to a 2D torus, which is a 2D mesh of nodes with "wraparound". (An example 2D torus can be seen at the bottom.) Explain how an all-to-all broadcast can be most efficiently executed on a 2D torus with $R$ rows and $C$ columns.

**Fill in your answer here**

Format | Σ

Words: 0





Maximum marks: 8

## 4.2 All-to-all broadcast on a 2D torus; performance model

We assume that following formula

$$t_s + t_w m$$

can be used to calculate the time usage of sending one message of $m$ bytes from one node to another. Here, $t_s$ og $t_w$ are two known constants. Derive a formula that quantifies the total time usage of an all-to-all broadcast that is exectuted on a 2D torus consisting of $R$ rows and $C$ columns. (Initially, each node has $m$ bytes as its own data.)

**Fill in your answer here**

Format

Words: 0

Maximum marks: 7

## 5.1 Private variable

Why are private variaibles needed in OpenMP programming? Please give one example of using a private variable.

**Fill in your answer here**

Format

Words: 0

Maximum marks: 10

## 5.2 Show the running result

If the following code snippet is executed by 4 OpenMP threads. What will be written as output?

```
int total_sum = 0;
int i;
#pragma omp parallel default(shared) reduction(+:total_sum)
{
  int my_id = omp_get_thread_num();
  int my_sum = 0;
  #pragma omp for schedule(static,10)
  for (i=1; i<=100; i++)
     my_sum += i;
  printf("From thread No.%d: my_sum=%d\n", my_id, my_sum);
  total_sum += my_sum;
}
printf("Total sum=%d\n",total_sum);
```

**Fill in your answer here**

Maximum marks: 10

## 5.3 Correcting error(s)

There is one (or several) error(s) in the following OpenMP code. Please correct the error(s).

. . .

```
int i;
double u[1000], v[1000];
for (i=0; i<1000; i++) {
   u[i] = 0.001*(i-500);
   v[i] = 0.0;
}

#pragma omp parallel default(shared)
{
  int time_step;
  double *tmp;

  for (time_step=0; time_step<100; time_step++)
  {
    #pragma omp for nowait
    for (i=1; i<999; i++)
       v[i] = u[i-1]-2*u[i]+u[i+1];

    tmp = v;
    v=u;
    u=tmp;
  }
}
```

**Fill in your answer here**

```
1
```

Maximum marks: 10

## 6.1 Merging two sorted sublists

The following function has the purpose of merging two input sub-lists and thereby producing a new list as the output. Both the input sub-lists are assumed to be already sorted and of length $m$. The output new list should be sorted and is of length $2m$.

The current implementation (see below) is missing something in the end, please complete the implementation.

```
void merge (int m, const int *sorted_sublist1, const int *sorted_sublist2, int *merged_list)
{
  int i,j,k;
  i = 0;
  j = 0;
  k = 0;


  while (i<m && j<m)
  {
    if (sorted_sublist1[i] <=sorted_sublist2[j])
    {
      merged_list[k] = sorted_sublist1[i];
      i++;
```

```
    }
    else
    {
      merged_list[k] = sorted_sublist2[j];
      j++;
    }

    k++;
  }

  /* The remaining part of this function is missing, please complete.
      .......
  */
}
```

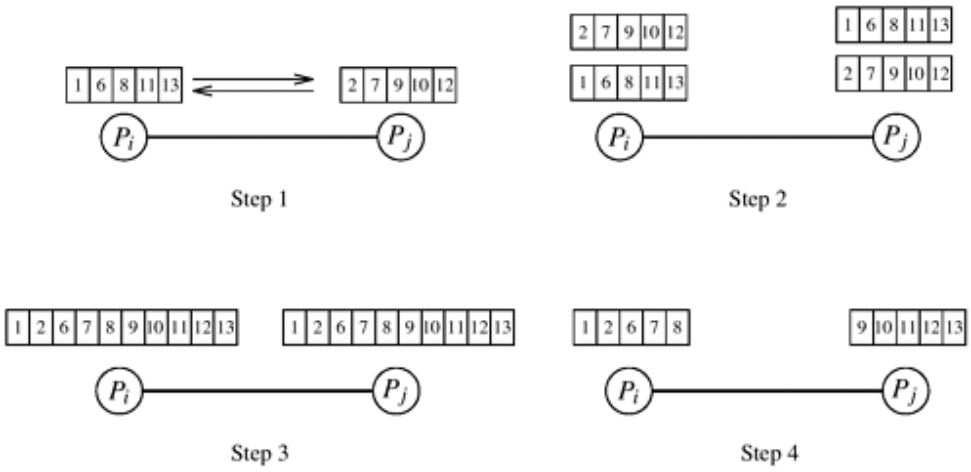
**Fill in your answer here**

```
1
```

Maximum marks: 10

## 6.2 Compare-split

The so-called compare-split operation involves two parallel processes, each having a sorted sub-list intitially. Such a parallel operation is an important building block in parallel sorting algorithms. The following figure shows an example of what happens during the compare-split operation, where each process intially has a sorted sub-list of length 5. (Rank $P_i$ is assumed to be lower than Rank $P_j$.)




Please implement the compare_split function with the follow syntax:

```
void compare_split (int m, int *my_sublist, int my_MPI_rank, int other_MPI_rank)
```

Note: You should use appropriate MPI command(s), as well as using the "merge" function implemented for the previous question.

Hint: The compare_split function is supposed to be called simultaneously by two MPI processes, each having as input its sorted sub-list of length $m$.

**Fill in your answer here**

```
1
```

Syntax for some of the most important MPI functions:

int MPI_Comm_size( MPI_Comm comm, int *size )

int MPI_Comm_rank( MPI_Comm comm, int *rank )

int MPI_Barrier( MPI_Comm comm )

int MPI_Send(const void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Status *status)

int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,
MPI_Comm comm )

int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,

```
            void *recvbuf, int recvcount, MPI_Datatype recvtype,
            MPI_Comm comm)

int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
            MPI_Datatype datatype,
            MPI_Op op, int root, MPI_Comm comm)

int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
            void *recvbuf, int recvcount, MPI_Datatype recvtype,
            int root, MPI_Comm comm)

int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
            void *recvbuf, int recvcount, MPI_Datatype recvtype,
            int root, MPI_Comm comm)
```

Maximum marks: 15

## 6.3 Block version of parallel odd-even transposition

**Parallel Odd-Even Transposition**

```
1.  procedure ODD-EVEN_PAR(n)
2.  begin
3.      id := process's label
4.      for i := 1 to n do
5.      begin
6.          if i is odd then
7.              if id is odd then
8.                  compare-exchange_min(id + 1);
9.              else
10.                 compare-exchange_max(id − 1);
11.         if i is even then
12.             if id is even then
13.                 compare-exchange_min(id + 1);
14.             else
15.                 compare-exchange_max(id − 1);
16.     end for
17. end ODD-EVEN_PAR
```

Parallel formulation of odd-even transposition.

The figure above sketches a psudo-code for executing the so-called parallel odd-even

transposition, which has the purpose of sorting a list of numbers. This psudo-code has assumed that the total length of the list is the same as the number of parallel processes.

You are now required to wirte a C function, based on the same idea of parallel odd-even transposition. The difference is that you should now allow the length of the list, $n$, to equal $P \cdot m$. That is, $n$ is a multiple of the number of MPI processes $P$. This means that each process will be intially assigned a sub-list of length $m$.

Implement the following C function:

```
void odd_even_block_paralell (int m, int *sublist)
```

Note: This function will be simultaneously called by all the MPI processes. You can assume that the input "sublist" (of length m) is already sorted on each MPI process before calling the odd_even_block_parallel function.

Hint: You should replace all appearances of "compare_exchange_min" and "comparre_exchange_max" by appropriately calling the compare_split function from the previous question.

**Fill in your answer here**

```
1
```

Maximum marks: 10