Suggested solutions for the INF3380 exam of spring 2016

Problem 1 (10%)

If the work assigned to p processors is denoted respectively by W_1, W_2, …, W_p, we can then define the *load imbalance ratio* as

$$\frac{\max_i(W_i) - \min_i(W_i)}{\min_i(W_i)}$$

Suppose a 100×100 matrix is partitioned into p rectangular blocks, as evenly as possible. What will the load imbalance ratio be, respectively, for the following three processor grids: 8×1, 4×2, and 3×3?

Suggested solution: For the case of a 8×1 processor grid, the processor with the most work will be responsible for 13 rows and 100 columns, wheres the processor with the least work will be responsible for 12 rows and 100 columns. Therefore the load imbalance raito can be calculated as

$$\frac{13 \times 100 - 12 \times 100}{12 \times 100} = \frac{1}{12}.$$

.

Similarly, for the case of a 4×2 processor grid, the load imbalance ratio is

$$\frac{25 \times 50 - 25 \times 50}{25 \times 50} = 0,$$

whereas for the 3×3 processor grid, the load imbalance ratio is

$$\frac{34 \times 34 - 33 \times 33}{33 \times 33} = \frac{67}{1089}.$$

Problem 2

Figure 1 (on page 2) shows a task dependency graph that involves six tasks: $t_1, t_2, \ldots, t_6$. For each task, the number inside the pair of parentheses shows its computational cost, as the number of time units needed. The number on each directed edge, between a pair of start and finish nodes, shows its communication cost as the number of time units needed.

We assume that each task can only be carried out by one worker. Moreover, communication on each edge will *execute by itself* (without manual initialization), as soon as the work on its start node is done. Note: If a task has one or several incoming communication edges, this task cannot start its work until all the incoming communications are done.

Problem 2a (10%)

If there is only one worker, what is the minimum number of time units needed before all the tasks are completed? Please explain your answer.

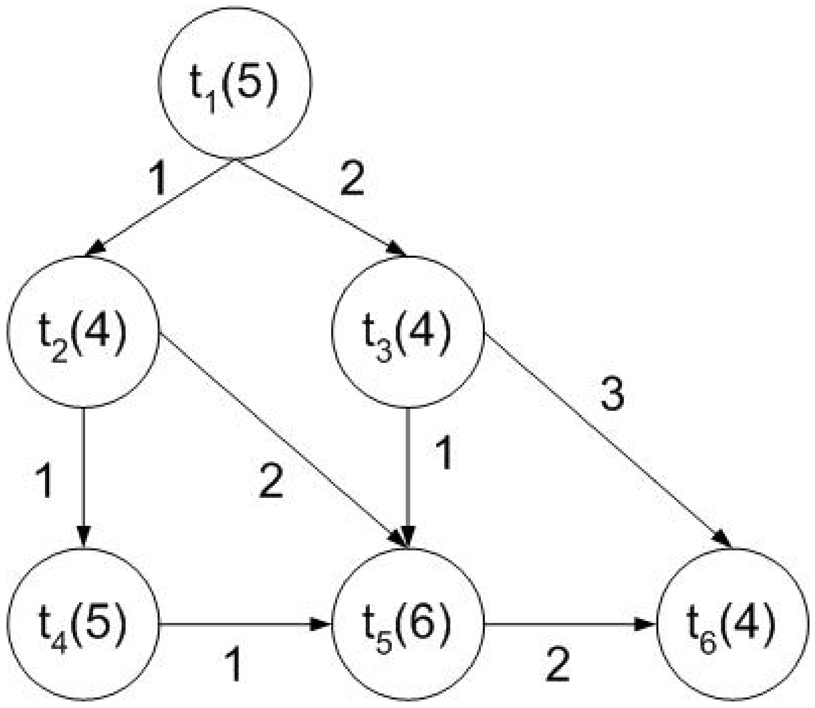

Figure 1: A task dependency graph

Suggested solution: The fastest-possible progress of a single worker is illustrated by the following table:

Time	Progress
End of time unit 5	task 1 is finished (and two messages are simultaneously sent out towards tasks 2 and 3)
End of time unit 6	task 2 is started (the message towards task 3 is still on its way)
End of time unit 10	task 2 is finished, and task 3 is started immediately
End of time unit 14	task 3 is finished, and task 4 is started immediately
End of time unit 19	task 4 is finished
End of time unit 20	task 5 is started (because the message from task 4 has just arrived)
End of time unit 26	task 5 is finished
End of time unit 28	task 6 is started (because the message from task 5 has just arrived)
End of time unit 32	task 6 is finished

Therefore, for the case of a single worker, the minimum number of time units needed is 32.

Problem 2b (5%)

What is the minimum number of time units needed, if there are two workers? Please explain your answer.

Suggested solution: Having a second worker can *only* allow task 3 to be carried out while the first worker is working on task 2. However, due to the fact that task 3 has to start one time unit later than task 2 (because the message from task 1 needs two time units to arrive at task 3), the number of time units saved is 3. Therefore, for the case of two workers, the minimum number of time units needed is 29.

Problem 2c (5%)

What is the minimum number of time units needed, if there are three workers? Please explain your answer.

Suggested solution: It doesn't help to have a third worker, because at most two tasks (task 2 and task 3) can be carried out concurrently. So the minimum number of time units needed is still 29.

Problem 3

Assume there are p processors in total, where each processor has a unique piece of data. The *gather* operation of collective communication has the purpose of letting one processor collect all the data from the other $p-1$ processors.

Problem 3a (10%)

If 16 processors are connected as a ring, please explain (step by step) how the gather operation can be most effectively achieved by making use of a series of one-to-one communications. (Assume processor 0 is to gather all the data pieces.)

Suggested solution: For the case of 16 processors, the simultaneous communication of messages (which get increasingly larger in size by the steps) is shown below:

Step 1	P1→P0, P3→P2, P5→P4, P7→P6, P9→P8, P11→P10, P13→P12, P15→P14
Step 2	P2→P0, P6→P4, P10→P8, P14→P12
Step 3	P4→P0, P12→P8
Step 4	P8→P0

Problem 3b (10%)

Assume the following cost model for sending a message of m words from one processor to another:

$$t_s + t_w m,$$

where t_s and t_w are two constants. Please derive the cost model of carrying out a gather operation on a ring of p processors, each initially having m words as its unique data. Here, you can assume that p is a power of 2.

Suggested solution: The total number of steps needed will be $\log_2 p$. During the first step, $p/2$ messages of size m are communicated at the same time (without disturbing each other). Then, during the second step, $p/4$ messages of size $2m$ are communicated simultaneously, and so on. Therefore, the total cost model is as follows:

$$\sum_{j=1}^{\log_2 p} t_s + t_w \left(m \cdot 2^{j-1}\right) = \log_2 p \cdot t_s + (p-1)m \cdot t_w.$$

Problem 3c (5%)

How will the cost model of the gather operation differ from above if p is not a power of 2? Please discuss the two specific cases of $p = 7$ and $p = 12$.

Suggested solution: For the case of $p = 7$, the communication of messages is shown below:

Step	Flow of messages	Time usage
1	P1→P0, P3→P2, P5→P4,	$t_s + t_w m$
2	P2→P0, P6→P4	$t_s + t_w \cdot 2m$
3	P4→P0	$t_s + t_w \cdot 3m$

That is, the total time cost is $3 \cdot t_s + 6m \cdot t_w$.

For the case of $p = 12$, the communication of messages is shown below:

Step	Flow of messages	Time usage
1	P1→P0, P3→P2, P5→P4, P7→P6, P9→P8, P11→P10	$t_s + t_w m$
2	P2→P0, P6→P4, P10→P8,	$t_s + t_w \cdot 2m$
3	P4→P0	$t_s + t_w \cdot 4m$
4	P8→P0	$t_s + t_w \cdot 4m$

That is, the total time cost is $4 \cdot t_s + 11m \cdot t_w$.

Therefore, in general, when p is not a power of 2, the cost model is

$$\lceil \log_2 p \rceil \cdot t_s + (p-1)m \cdot t_w.$$

Problem 4

Problem 4a (10%)

The following pseudo-code describes the bubble sort algorithm that can be used to sort a list of n values: $a_1, a_2, \ldots, a_n$.

```
1.        procedure BUBBLE_SORT($n$)
2.        begin
3.            for $i := n-1$ downto 1 do
4.                for $j := 1$ to $i$ do
5.                    compare-exchange($a_j, a_{j+1}$);
6.        end BUBBLE_SORT
```

Please write a sequential C function

```
void serial_bubble_sort (int n, int* a);
```

which implements the above pseudo-code. (You can assume that n is an even number.) Please allow earlier termination of the `i`-indexed loop when possible.

Suggested solution:

```
int compare_exchange (int* v1, int* v2)
{
  int tmp;
  if (*v1 <= *v2)
    return 0;

  tmp = *v1;
  *v1 = *v2;
  *v2 = tmp;
  return 1;
}


void serial_bubble_sort (int n, int* a)
{
  int i,j,changed;
  for (i=n-2; i>=0; i--) {
    changed = 0;
    for (j=0; j<=i; j++)
      if (compare_exchange(&(a[j]), &(a[j+1])))
        changed = 1;
    if (!changed)
      break;
  }
}
```

We remark that an array in the C programming lanugage always has its lowest index starting from 0 (not 1), therefore the `i` and `j` indices above have been shifted downward by 1.

Problem 4b (10%)

The following pseudo-code describes the odd-even transposition algorithm that allows parallelism when sorting a list:

```
1.      procedure ODD-EVEN($n$)
2.      begin
3.          for $i := 1$ to $n$ do
4.          begin
5.              if $i$ is odd then
6.                  for $j := 0$ to $n/2-1$ do
7.                      compare-exchange($a_{2j+1}, a_{2j+2}$);
8.              if $i$ is even then
9.                  for $j := 1$ to $n/2-1$ do
10.                     compare-exchange($a_{2j}, a_{2j+1}$);
11.         end for
12.     end ODD-EVEN
```

Please write an OpenMP-parallelized C function

```
void para_oddeven_sort (int n, int* a);
```

which implements the above pseudo-code. (You can assume that n is an even number.) Please allow earlier termination of the `i`-indexed loop when possible.

Suggested solution:

```
void para_oddeven_sort (int n, int* a)
{
  int i, j, changed1=0, changed2=0;

#pragma omp parallel default(shared) private(i,j)
 {
  for (i=0; i<n; i+=2) {

    /* odd step */
#pragma omp for reduction(+:changed1)
    for (j=0; j<=n/2-1; j++)
      if (compare_exchange(&(a[2*j]),&(a[2*j+1])))
        changed1 = 1;

    /* even step */
#pragma omp for reduction(+:changed2)
    for (j=1; j<=n/2-1; j++)
      if (compare_exchange(&(a[2*j-1]),&(a[2*j])))
        changed2 = 1;

    if (!changed1 && !changed2)
      break;
#pragma omp barrier

#pragma omp single
    {
      changed1 = changed2 = 0;
    }
  }
 }
}
```

There are a few issues that need to be commented in the above OpenMP code. First, the length of the list, `n`, is assumed to be an even number. Second, each iteration that is indexed by `i` contains both the "odd" and "even" steps. Third, to be able terminate the `i`-indexed for-loop earlier, it requires that *neither* the "odd" step *nor* the "even" step within the same iteration incurs exchanges. Fourth, there is only a single `omp parallel` region for saving the overhead of repeatedly spawning and killing OpenMP threads. Last but not least, the `changed1` and `changed2` flags have to undergo a reduction operation of type summation, in connection with their respective `omp for` directive. (The `omp barrier` and `single` directives are also important to have.)

Probem 5

Problem 5a (10%)

We want to parallelize the matrix-vector multiplication $y = Ax$ with MPI programming, where A is a $n \times n$ matrix, x and y are both vectors of length n. Please implement the following C function, which will be called by every MPI process:

```
void para_matvec (int n, double** A, double* x, double* y);
```

You can assume that n is divisible by the number of MPI processes p, and that the A matrix is already distributed by a row-wise block 1D partitioning among the processes. That is, as input to `para_matvec`, 2D array `A` contains the matching portion of A for each MPI process. (The first dimension of `A` is n/p, whereas its second dimension is n.) However, only process 0 has the entire x vector in its input 1D array `x`, whereas the other processes have dummy values in their input `x` arrays. Upon finish, the output array `y` is supposed to contain the entire y vector only on process 0, whereas the other processes can have dummy values in their output `y` arrays.

Suggested solution: An MPI implementation is as follows (we assume that both the `x` and `y` arrays are already allocated with space for storing `n` values upon input):

```
void para_matvec (int n, double** A, double* x, double* y)
{
  int i,j,p;
  double tmp;
  MPI_COMM_SIZE (&p, MPI_COMM_WORLD);
  MPI_Bcast (x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  for (i=0; i<n/p; i++) {
    tmp=0.0;
    for (j=0; j<n; j++)
      tmp += A[i][j]*x[j];
    y[i] = tmp;
  }
  MPI_Gather (y, n/p, MPI_DOUBLE,
              y, n/p, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

Problem 5b (10%)

Please derive a theoretical formula of the parallel run time of the above function. (You can use the t_s and t_w constants defined in Problem 3b. Moreover, you can use t_c to denote the computational cost of each multiplication or addition operation.)

Suggested solution: The cost of the broadcast step (`MPI_Bcast`) is $\log_2 p\,(t_s + n\cdot t_w)$, whereas the cost of the gather step (`MPI_Gather`) is $\log_2 p\cdot t_s + (p-1)\frac{n}{p}\cdot t_w$. The computational time used by each process will be $2\frac{n^2}{p}t_c$. Therefore, the overal parallel run time is

$$T_P(p,n) = \log_2 p\,(t_s + n\cdot t_w) + 2\frac{n^2}{p}t_c + \log_2 p\cdot t_s + (p-1)\frac{n}{p}\cdot t_w = 2\log_2 p\cdot t_s + \left(\log_2 p\cdot n + \frac{n(p-1)}{p}\right)t_w + 2\frac{n^2}{p}t_c.$$

Problem 5c (5%)

Please carry out the isoefficiency analysis.

Suggested solution: Noticing that $W = T_S = 2n^2 t_c$, we can calculate the overhead function T_O as follows:

$$T_O(p,n) = pT_P(p,n) - T_S = 2p\log_2 p \cdot t_s + p\log_2 p \cdot n \cdot t_w + n(p-1) \cdot t_w. \tag{1}$$

Now, we want to find out how fast W should (at least) grow with respect to p, in order to maintain a same level of parallel efficiency, that is, isoefficiency. From the textbook we know that isoefficiency requires that $W = KT_O$, where K is a constant that is related to the parallel efficiency E as $K = \frac{E}{1-E}$.

If we only focus on the first term on the right-hand of (1), we find that

$$W = K \cdot 2p\log_2 p \cdot t_s,$$

which gives an asymptotic isoefficiency function as $\Theta(p\log p)$.

For the second term on the right-hand of (1), we find that

$$\begin{aligned} W &= K \cdot p\log_2 p \cdot n \cdot t_w & (2)\\ 2n^2 t_c &= K \cdot p\log_2 p \cdot n \cdot t_w & (3)\\ \Rightarrow \quad n &= \frac{K \cdot t_w}{2t_c} p\log_2 p & (4)\\ \Rightarrow \quad W = 2n^2 t_c &= \frac{K^2 \cdot t_w^2}{2t_c}(p\log_2 p)^2 & (5) \end{aligned}$$

which gives an asymptotic isoefficiency function as $\Theta((p\log p)^2)$.

For the last term on the right-hand of (1), we find that

$$\begin{aligned} W &= K \cdot n(p-1) \cdot t_w & (6)\\ 2n^2 t_c &= K \cdot n(p-1) \cdot t_w & (7)\\ \Rightarrow \quad n &= K\frac{t_w}{2t_c}(p-1) & (8)\\ \Rightarrow \quad W = 2n^2 t_c &= \frac{K^2 \cdot t_w^2}{2t_c}(p-1)^2 & (9) \end{aligned}$$

which gives an asymptotic isoefficiency function as $\Theta(p^2)$.

For the current parallel implementation, which is based on a row-wise block1D partitioning, we have to require $n \geq p$. Therefore, with respect to the degree of concurrency, the asymptotic isoefficiency function becomes $\Theta(p^2)$.

Taking the largest asymptotic isoefficiency function from above, the overall isoefficiency function is $\Theta((p\log p)^2)$.

Appendix: The syntax of some commonly used 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)
```