Suggested solutions for the INF3380 exam of spring 2014

Problem 1 (10%)

Suppose a computational problem has been decomposed into many tasks, such that the number of tasks is larger than the number of available processors. What are the two most important principles for mapping the tasks to processors?

Suggested solution: As discussed in Section 3.4 of the textbook, the overall objective of the mapping is that all tasks complete in the shortest amount of time. That is, the overhead of executing the tasks in parallel should be minimized. The two most important principles are therefore **(1) reducing the amount of interaction time between the processors** and **(2) reducing the amount of idle time on the processors**.

Problem 2

Problem 2a (5%)

For a 1D linear array of p processors, explain the following time usage model for broadcasting a message of length m from one processor to all the other processors:

$$T_P = (t_s + t_w m)\lceil \log_2 p \rceil,$$

where t_s and t_w are two constants, while $\lceil \cdot \rceil$ denotes the so-called ceiling function.

Suggested solution: As mentioned in Section 4.1.1 of the textbook, the technique of **recursive doubling** can be used. The first step is to let the source processor send the message to another appropriately chosen processor, such that the p processors are divided into two groups, each now having a source processor. Then, the second step repeats the above action simultaneously within each of the two groups. This recursive process continues until each group has one (or zero) processor, at which time all the processors have already received the message.

The number of steps needed is $\lceil \log_2 p \rceil$. The time used in each step (that is, for sending the message from one processor to another) is $t_s + t_w m$, where t_s stands for the **startup time** and t_w stands for the **per-data-value transfer time**. Therefore, the total time usage is

$$T_P = (t_s + t_w m)\lceil \log_2 p \rceil,$$

Problem 2b (5%)

Suppose there are $p = 12$ processors in the 1D linear array. Explain, step by step, the actual flow of data between the processors.

Suggested solution: Suppose processor 0 is the source. During step 1, the message is sent from processor 0 to processor 6. During step 2, the message is sent from processor 0 to processor 3, while **at the same time** the message is also sent from processor 6 to processor 9. During step 3, the message transfer **simultaneously** happens between the following four pairs:

> processor 0→processor 1, processor 3→processor 4, processor 6→processor 7, processor 9→processor 10.

During step 4, the message transfer **simultaneously** happens between the following four pairs:

> processor 1→processor 2, processor 4→processor 5, processor 7→processor 8, processor 10→processor 11.

Problem 2c (5%)

Let us now consider a 2D mesh of processors, where the number of rows is r and the number of columns is s. Please derive the time usage model of one-to-all broadcast for this case.

Suggested solution: For a 2D mesh of processors, the entire one-to-all broadcast operation can be carried out in two phases. During phase one, the source processor horizontally broadcasts the message to all the processors on the same row. Then, during phase two, each source processor on that row vertically broadcasts the message to all the processors on its column.

Since there are r rows and s columns, the total time usage is

$$T_P = (t_s + t_w m)\left(\lceil \log_2 r \rceil + \lceil \log_2 s \rceil\right).$$

Problem 3 (15%)

Suppose the environment variable `OMP_NUM_THREADS` is set to be 4. What will the following OpenMP code segment produce as the output results (written out by `printf`)? Please explain your answer.

```
  int *result_array = (int*)malloc(100*sizeof(int));
  int i, num_threads;

#pragma omp parallel default(shared)
  {
    int thread_id = omp_get_thread_num();

    if (thread_id==0)
      num_threads = omp_get_num_threads();

#pragma omp for schedule(static,2)
    for (i=1; i<=20; i++)
      result_array[thread_id] += i;

  }

  for (i=0; i<num_threads; i++)
    printf("Result from thread %d is %d\n",i,result_array[i]);
```

Suggested solution: Since the environment variable `OMP_NUM_THREADS` is set to be 4, four OpenMP threads execute side-by-side in the parallel region. Moreover, the `schedule(static,2)` clause will divide the iterations of the `for`-loop among the four threads as follows:

- Thread 0: `i=1,2,9,10,17,18`.
- Thread 1: `i=3,4,11,12,19,20`.
- Thread 2: `i=5,6,13,14`.
- Thread 3: `i=7,8,15,16`.

Therefore, the output results generated by `printf` are as follows:

```
Result from thread 0 is 57
Result from thread 1 is 69
Result from thread 2 is 38
Result from thread 3 is 46
```

Comment: Since the `malloc` function may not always give 0 as initial values to the allocated array, it is safer to add the following loop immediately after the call to `malloc`:

```
for (i=0; i<100; i++)
  result_array[i] = 0;
```

Problem 4

Floyd's algorithm (shown below) can be used to solve the "all-pairs shortest paths" problem for a weighted graph, whose $n \times n$ adjacency matrix is denoted by A.

1. **procedure** FLOYD_ALL_PAIRS_SP(A)
2. **begin**
3. $D^{(0)} = A$;
4. **for** $k := 1$ **to** n **do**
5. **for** $i := 1$ **to** n **do**
6. **for** $j := 1$ **to** n **do**
7. $d_{i,j}^{(k)} := \min\left(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right)$;
8. **end** FLOYD_ALL_PAIRS_SP

Problem 4a (10%)

Suppose the $n \times n$ adjacency matrix A is divided among p processors by a 1D row-wise block partitioning. Describe in detail the communication operations that will be needed in an MPI parallelization of Floyd's algorithm. (You don't have to write the entire MPI program.)

Suggested solution: At the beginning of each `k`-iteration, all the MPI processes should be able to identify the rank (say, `row_k_owner`) of the MPI process that has row `k` of A in its responsibility. Then, the following collectiv MPI call is invoked on each process:

```
MPI_Bcast (buffer, n, MPI_DOUBLE, row_k_owner, MPI_COMM_WORLD);
```

Note, on the MPI process whose rank equals `row_k_owner`, the pointer variable `buffer` should point to the actual row (inside the process' assigned local piece of A) that corresponds to row `k` of the global matrix A. On all the other MPI processes, `buffer` should point to an assistant 1D array that is to receive row `k` from the source MPI process. Thereafter, the local computation per MPI process can proceed independently for this `k`-iteration.

Problem 4b (15%)

Derive the time usage model $T_P(n,p)$ of the MPI parallelization based on a 1D row-wise block partitioning.

Suggested solution: From **Problem 2a** we know that the cost of a single one-to-all broadcast of n values is

$$(t_s + t_w n)\log p.$$

The local computation cost per MPI process in each `k`-iteration is

$$\frac{n^2}{p}.$$

Therefore, recalling that in total n iterations are needed, we can get

$$T_P(n,p) = n\left((t_s + t_w n)\log p + \frac{n^2}{p}\right).$$

Problem 4c (10%)

Derive the associated isoefficiency function.

Suggested solution: By definition we have

$$T_O = pT_P - W.$$

Using the above formula for $T_P(n,p)$, while remembering that $W = n^3$, we can arrive at

$$T_O = t_s np\log p + t_w n^2 p\log p.$$

With respect to maintaining a constant level of parallel efficiency, the first term on the right-hand side of the above formula yields

$$W = Knp\log p \quad \Rightarrow \quad W = \Theta((p\log p)^{1.5})$$

whereas the second term yields

$$W = Kn^2 p\log p \quad \Rightarrow \quad W = \Theta((p\log p)^3)$$

At the same time, since the maximum allowed number of MPI processes for a 1D partitioning of matrix A is n, the **degree of concurrancy** consideration requires that

$$n \leq p \quad \Rightarrow \quad W \geq p^3$$

Therefore, the overall asymptotic isoefficiency function is $\Theta((p\log p)^3)$.

Problem 5

Let A denote an $n \times n$ matrix, whereas b and y are two vectors of length n. The Gauss elimination algorithm can be described as follows:

```
1.      procedure GAUSSIAN_ELIMINATION (A, b, y)
2.      begin
3.          for k := 0 to n − 1 do              /* Outer loop */
4.          begin
5.              for j := k + 1 to n − 1 do
6.                  A[k, j] := A[k, j]/A[k, k];  /* Division step */
7.              y[k] := b[k]/A[k, k];
8.              A[k, k] := 1;
9.              for i := k + 1 to n − 1 do
10.             begin
11.                 for j := k + 1 to n − 1 do
12.                     A[i, j] := A[i, j] − A[i, k] × A[k, j]; /* Elimination step */
13.                 b[i] := b[i] − A[i, k] × y[k];
14.                 A[i, k] := 0;
15.             endfor;             /* Line 9 */
16.         endfor;                 /* Line 3 */
17.     end GAUSSIAN_ELIMINATION
```

Problem 5a (10%)

Write a serial C function

```
void gauss_elim(double **A, double *b, double *y)
```

that implements the serial Gauss elimination algorithm.

Suggested solution:

```
void gauss_elim(double **A, double *b, double *y)
{
  int i,j,k;

  for (k=0; k<n; k++) {

    for (j=k+1; j<n; j++)
      A[k][j] = A[k][j]/A[k][k];  /* Division step */

    y[k] = b[k]/A[k][k];
    A[k][k] = 1;

    for (i=k+1; i<n; i++) {
      for (j=k+1; j<n; j++) {
        A[i][j] = A[i][j]-A[i][k]*A[k][j];  /* Elimination step */
      }
      b[i] = b[i]-A[i][k]*y[k];
```

```
        A[i][k] = 0;
      }

    }
  }
```

Comment: Since the value of n is not passed as an input argument to the `gauss_elim` function, we have to assume that there is a global variable named `n`.

Problem 5b (15%)

Parallelize the serial C function with help of OpenMP. Discuss the quality of your OpenMP parallelization with respect to overhead and load balancing.

Suggested solution: The most important observation is that the outermost `k`-iterations have to be executed in sequence, that is, they cannot be parallelized. Another important observation is that the `i`-iterations can be parallelized.

```
  void gauss_elim(double **A, double *b, double *y)
  {
    int i,j,k;

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

#pragma omp for schedule(static)
      for (j=k+1; j<n; j++)
        A[k][j] = A[k][j]/A[k][k];  /* Division step */

#pragma omp single
{
      y[k] = b[k]/A[k][k];
      A[k][k] = 1;
}

#pragma omp for schedule(static) private(j)
      for (i=k+1; i<n; i++) {
        for (j=k+1; j<n; j++) {
          A[i][j] = A[i][j]-A[i][k]*A[k][j];  /* Elimination step */
        }
        b[i] = b[i]-A[i][k]*y[k];
        A[i][k] = 0;
      }

    }
  }
```

Comments: The reason of having a parallel region that encapsulates the outermost `for`-loop is to reduce the overhead that is related to the master thread repeatedly spawning and terminating other threads. At the same time, it is very important to wrap

```
y[k] = b[k]/A[k][k];
A[k][k] = 1;
```

inside the `single` (or `master`) directive. This is to avoid any **race condition**.