Suggested solutions for the INF3380 exam of spring 2012

Problem 1 (20%)

Suppose there is a 2D rectangular uniform mesh, which has M points in the x-direction and N points in the y-direction. Please describe in detail how to carry out a checkerboard block decomposition of the mesh into $P \times Q$ blocks, as evenly as possible. Note that M may not be divisible by P, and N may not be divisible by Q.

For an MPI process with rank k, which part of the original 2D mesh will be assigned to it?

Suggested solution: A block with 2D-index (p,q), where $0 \leq p < P$ and $0 \leq q < Q$, should cover from row $\lfloor \frac{pM}{P} \rfloor + 1$ to row $\lfloor \frac{(p+1)M}{P} \rfloor$ in the x-direction, and cover from column $\lfloor \frac{qN}{Q} \rfloor + 1$ to column $\lfloor \frac{(q+1)N}{Q} \rfloor$ in the y-direction.

For an MPI process with rank k, where $0 \leq k < PQ$, we can derive its 2D-index (p,q) as follows:

$$p = \mathrm{mod}(k,P), \quad q = \lfloor \frac{k}{P} \rfloor.$$

Problem 2 (15%)

Discuss the differences between Amdahl's Law and Gustafson-Barsis's Law (**ikke lenger pensum 2013**).

Suggested solution: Both laws are concerned with estimating obtainable speedup, where Amdahl's Law states $\Psi(p) \leq \frac{1}{f+\frac{1-f}{p}}$ and Gustafson-Barsis's Law states $\Psi(p) \leq p + (1-p)s$. The f factor in Amdahl's Law is the fraction of inherently sequential operations in $T(1)$, whereas the s factor in Gustafson-Barsis's Law is the fraction of inherently sequential operations in $T(p)$.

Another major difference is that Amdahl's Law considers speedup in the context of a fixed problem size, whereas Gustafson-Barsis's Law encourages increasing the problem size as p increases.

Problem 3 (10%)

What is the latency in connection with point-to-point MPI communication? Can you suggest an experiment to measure the actual size of the latency?

Suggested solution: Latency is the so-called startup time, which does not depend on the actual amount of data communicated, in the context of point-to-point MPI communication.

Let us assume that the entire communication time is of the following form:

$$C(L) = \tau + \frac{L}{\beta},$$

where L denotes the message size, τ denotes latency, and β is a constant representing the inverse of the communication bandwidth. The value of τ (and also the value of β) can be estimated by a so-called pingpong test, which first measures the time usage of bouncing between two MPI processes a message of size L_1. Then it repeats the measurement for several different message sizes $L_2, L_3, \ldots$. The values of τ and β can be found by, for example, a least squares approximation.

Problem 4

We have the following serial code segment:

```
u=(double*)malloc((N+2)*sizeof(double));
u_prev=(double*)malloc((N+2)*sizeof(double));
for (i=0; i<=N+1; i++)
  u_prev[i]=sin(PI*i/(N+1));

t = 0.0;
while (t < final_T) {

  for (i=1; i<=N; i++)
    u[i]=u_prev[i]+a*(u_prev[i-1]-2*u_prev[i]+u_prev[i+1]);

  u[0] = 0.0;
  u[N+1] = 0.0;

  t += dt;
  tmp_ptr = u_prev;
  u_prev = u;
  u = tmp_ptr;
}
```

Problem 4a (10%)

Write an OpenMP implementation of the above serial code segment.

Suggested solution:

```
u=(double*)malloc((N+2)*sizeof(double));
u_prev=(double*)malloc((N+2)*sizeof(double));
#pragma omp parallel for
for (i=0; i<=N+1; i++)
  u_prev[i]=sin(PI*i/(N+1));

t = 0.0;
while (t < final_T) {

#pragma omp parallel for
  for (i=1; i<=N; i++)
    u[i]=u_prev[i]+a*(u_prev[i-1]-2*u_prev[i]+u_prev[i+1]);
```

```
  u[0] = 0.0;
  u[N+1] = 0.0;

  t += dt;
  tmp_ptr = u_prev;
  u_prev = u;
  u = tmp_ptr;
}
```

Problem 4b (10%)

Write an MPI implementation of the above serial code segment. Please note that `u` and `u_prev` should be evenly distributed among the MPI processes, and inter-process communication is to be carried out at every time step.

Suggested solution:

```
MPI_Status status;
int my_rank, num_procs;
int my_offset, my_N;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_size (MPI_COMM_WORLD, &num_procs);
my_offset = my_rank*N/num_procs;
my_N = (my_rank+1)*N/num_procs - my_offset;

u=(double*)malloc((my_N+2)*sizeof(double));
u_prev=(double*)malloc((my_N+2)*sizeof(double));
for (i=0; i<=my_N+1; i++)
  u_prev[i]=sin(PI*(i+my_offset)/(N+1));

t = 0.0;
while (t < final_T) {

  for (i=1; i<=my_N; i++)
    u[i]=u_prev[i]+a*(u_prev[i-1]-2*u_prev[i]+u_prev[i+1]);

  if (my_rank==0)
    u[0] = 0.0;
  else {
    MPI_Send (&(u[1]), 1, MPI_DOUBLE, my_rank-1, 100, MPI_COMM_WORLD);
    MPI_Recv (&(u[0]), 1, MPI_DOUBLE, my_rank-1, 200, MPI_COMM_WORLD, &status);
  }

  if (my_rank==num_procs-1)
    u[my_N+1] = 0.0;
  else {
    MPI_Send (&(u[my_N]), 1, MPI_DOUBLE, my_rank+1, 200, MPI_COMM_WORLD);
    MPI_Recv (&(u[my_N+1]), 1, MPI_DOUBLE, my_rank+1, 100, MPI_COMM_WORLD, &status);
  }
```

```
  t += dt;
  tmp_ptr = u_prev;
  u_prev = u;
  u = tmp_ptr;
}
```

Problem 4c (10%)

If inter-process communication is now only allowed every second time step, how should the above MPI implementation be modified? Please also explain the advantages and disadvantages of this modified MPI implementation.

Suggested solution: First, the two arrays now need to be of length `my_N+4` instead of `my_N+2`. Then, during an odd-numbered time step, computation goes over `my_N+2` points, no communication is needed. Thereafter, during the subsequent even-numbered time step, computation goes over `my_N` points, and two values (instead of one value) are exchanged between each pair of neighboring MPI processes.

The advantage is the reduced latency overhead, whereas the disadvantage is a slightly longer array length and two extra data points in computation during all the odd-numbered time steps.

Problem 5a (15%)

Chapter 14 of the textbook (Quinn: *Parallel Programming in C with MPI and OpenMP*) discusses three parallelizations of the serial quicksort algorithm. Please pick any one of the three parallel algorithms and explain it in detail.

Suggested solution: Let us assume distributed memory, and that the number of processes is a power of 2.

- To begin with, each process is assigned with a piece of the unsorted global list.
- We randomly choose a pivot from one of the processes and broadcast it to every process.
- Each process divides its unsorted list into two parts: those smaller than (or equal) the pivot, those greater than the pivot.
- Each process in the upper half of the process list sends its "low list" to a partner process in the lower half of the process list and receives a "high list" in return.
- Now, the upper-half processes have only values greater than the pivot, and the lower-half processes have only values smaller than the pivot.
- Thereafter, the processes divide themselves into two groups and the algorithm recurses.
- After $\log P$ recursions, every process has an unsorted list of values completely disjoint from the values held by the other processes.
- Finally, each process independently sorts its local list.

Problem 5b (10%)

Suppose we have the following unsorted list of numbers

29, 14, 37, 2, 99, 45, 21, 19, 77, 63, 9, 88, 34, 56, 28, 71

and want to sort them using four processors. Please use the above chosen parallel algorithm and demonstrate, step by step, the entire parallel sorting procedure.

Suggested solution:

1. Process 0 takes four values: (29,14,37,2). Process 1 takes four values: (99,45,21,19). Process 2 takes four values: (77,63,9,88). Process 3 takes four values: (34,56,28,71).

2. The value of 29 is chosen as the pivot and broadcast to Processes 1,2,3.

3. Process 0 splits its local list into two parts: (29,14,2) (37). Process 1 splits its local list into two parts: (21,19) (99,45). Process 2 splits its local list into two parts: (9) (77,63,88). Process 3 splits its local list into two parts: (28) (34,56,71).

4. Process 0 sends its upper part to Process 2, while Process 2 sends its lower part to Process 0. Process 1 sends its upper part to Process 3, where Process 3 sends its lower part to Process 1. The result is: Process 0 has (29,14,2,9). Process 1 has (21,19,28). Process 2 has (37,77,63,88). Process 3 has (99,45,34,56,71).

5. Process 0 and Process 1 now form one group, Process 2 and Process 3 form another group.

6. The value of 21 is chosen as pivot for the first group, with the result that Process 0 splits its local list into two parts: (14,2,9) (29), and Process 1 splits its local list into two parts: (21,19) (28). Then, Process 0 sends its upper part to Process 1, while Process 1 sends its lower part to Process 0. The result is that Process 0 gets (14,2,9,21,19) and Process 1 gets (29,28).

7. The value of 37 is chosen as pivot for the second group, with the result that Process 2 splits its local list into two parts: (37) (77,63,88), and Process 3 splits its local list into two parts: (34) (99,45,56,71). Then, Process 2 sends its upper part to Prices 3, while Process 3 sends its lower part to Process 2. The result is that Process 2 gets (37,34) and Process 1 gets (77,63,88,99,45,56,71).

8. Each process independently sorts its local list, with the final result as: Process 0 has (2,9,14,19,21). Process 1 has (28,29). Process 2 has (34,37). Process 3 has (45,56,63,71,77,88,99).