Suggested solutions for the INF3380 exam of spring 2013

Problem 1 (10%)

If a computational problem has 10% of its work that must be carried out serially, prove that the maximum obtainable speedup cannot exceed 10 by any parallelization.

Suggested solution: The obtainable speedup can be calculated as

$$S(p) = \frac{T(1)}{T(p)} \leq \frac{f+(1-f)}{f+\frac{1-f}{p}},$$

where f is the fraction of inherently serial work in $T(1)$. It is therefore clear that $\max S(p) = \lim S(p)_{p\to\infty} = \frac{1}{f}$. For the current case, where we have $f = 10\%$, the maximum speedup can thus not exceed 10.

Comment: The reason of having $\leq$ in the above formula is due to consideration of likely parallelization overhead and possible load imbalance.

Problem 2 (15%)

```
for (k=0; k<n; k++)
  for (j=0; j<k; j++)
    A[k][j] = A[j][k];
```

Write an OpenMP parallelization of the above code segment. Discuss your solution with respect to load balancing and parallelization overhead.

Suggested solution: First of all, the above nested double `for`-loop is parallelizable, without the danger of race condition. However, the difficulty is that the work amount with each `k`-iteration increases (because of `for (j=0; j<k; j++)`).

If `#pragma omp parallel for` is inserted before the `for`-loop with index `k`, load imbalance will arise. This is because the default scheduler is `static` and uses a largest possible `chunksize` value by default.

If `#pragma omp parallel for` is inserted before the `for`-loop with index `j`, load imbalance will no longer be a problem. However, the overhead due to repeatedly forking and joining threads will be excessive.

The best solution is as follows:

```
#pragma omp parallel for schedule(dynamic, chunksize)
for (k=0; k<n; k++)
  for (j=0; j<k; j++)
    A[k][j] = A[j][k];
```

Comment: The value of `chunksize` should neither be too large or too small, depending on the actual size of n. Another possibility is to use the `guided` scheduler. A third possibility is to use `schedule(static,1)`, for which load imbalance will not be very severe.

Problem 3 (20%)

In Oblig-1 we have looked at the problem of "image denoising", where the computation at each pixel is of the following form:

$$\bar{u}_{i,j} = u_{i,j} + \kappa \left(u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + u_{i,j+1} + u_{i+1,j} \right).$$

Suppose MPI is used to parallelize "image denoising" and that `MPI_Send` and `MPI_Recv` are used to exchange data between two and two neighbors. Moreover, we assume that the time taken to exchange an MPI message of size m is

$$t_s + t_w m,$$

where t_s and t_w are two known constant values.

For the case of a picture that has $n \times n$ pixels and there are P MPI processes, discuss when it pays off to use a 2D block-partitioning instead of a 1D block-partitioning. (Hint: you are supposed to derive a relation between n, t_s, t_w and P.)

Suggested solution: For the 1D block-partitioning, most of the MPI processes will have two neighbors that need to exchange data with. The size of each message in such a case is n. Therefore, the total communication overhead per process is

$$2\left(t_s + t_w n\right).$$

For the 2D block-partitioning, most of the MPI processes will have four neighbors that need to exchange data with. The size of each message in such a case is $n/\sqrt{P}$. Therefore, the total communication overhead per process is

$$4\left(t_s + t_w \frac{n}{\sqrt{P}}\right).$$

In order for the 2D block-partitioning to pay off, we need to have

$$4\left(t_s + t_w \frac{n}{\sqrt{P}}\right) < 2\left(t_s + t_w n\right),$$

which can give the following relationship:

$$n - \frac{2n}{\sqrt{P}} > \frac{t_s}{t_w}.$$

Problem 4

We want to compute $y = Ax$, where A is an $n \times n$ matrix, and x and y are two vectors of length n.

Problem 4a (10%)

Explain how the matrix-vector multiplication can be parallelized, if we assume a 1D rowwise block-partitioning of A, x and y.

Suggested solution: The 1D rowwise block-partitioning means that the rows of matrix A are equally distributed among the p processes, each having n/p rows of A. Moreover, the x vector is also equally distributed among the p processes, each having n/p values of x.

Therefore, the first step of parallelization is to do an all-to-all broadcast among the p processes, such that each process gets the entire x vector. Thereafter, each process can independently carry out a local matrix-vector multiplication to produce the desired segment of the y vector.

Problem 4b (15%)

According to the textbook, the time usage of the above parallelization will be

$$T_P = \frac{n^2}{p} + t_s \log p + t_w n$$

where p is the number of processing elements, t_s and t_w are two known constant values.

Carry out a scalability analysis with help of the "isoefficiency" metric.

Suggested solution: Recall that the "isoefficiency" metric is about finding a guidance about how fast the problem size W should (asymptotically) grow as p increases, such that the parallel efficiency is maintained at a constant level. The exact formula of "isoefficiency" metric is expressed as

$$W = KT_O(W, p),$$

where $K = E/(1-E)$ is a desirable constant and T_O is the total overhead.

For the above case of matrix-vector multiplication, we have $W = T_S = n^2$ and

$$T_O(W, p) = pT_p - T_S = t_s p \log p + t_w np.$$

For the first term of T_O, the "isoefficiency" metric requires $n^2 = Kt_s p \log p$, whereas the second term of T_O requires

$$n^2 = Kt_w np \quad \Rightarrow \quad n = Kt_w p.$$

It can be seen that the second term makes a higher demand on the problem size, which in term means

$$W = n^2 = K^2 t_w^2 p^2 = O(p^2).$$

Problem 5

The problem of "all-pairs shortest paths" is about finding the shortest path between any pair of nodes in a graph. As the starting point we have a matrix A that shows all the direct paths between the nodes. As the result we want to compute a matrix D such that $d_{i,j}$ is the length of the shortest path from node i to node j.

Problem 5a (10%)

Compute D if A is as follows:

$$\begin{bmatrix} 0 & 4 & \infty & \infty \\ 2 & 0 & 3 & 3 \\ \infty & 4 & 0 & 3 \\ \infty & 2 & 4 & 0 \end{bmatrix}$$

Suggested solution:

$$D = \begin{bmatrix} 0 & 4 & 7 & 7 \\ 2 & 0 & 3 & 3 \\ 6 & 4 & 0 & 3 \\ 4 & 2 & 4 & 0 \end{bmatrix}$$

Problem 5b (10%)

Explain how Floyd's algorithm can be used to solve the "all-pairs shortest paths"-problem in general.

Suggested solution: Floyd's Algorithm:

> **procedure** FLOYD_ALL_PAIRS_SP(A)
> **begin**
> $D^{(0)} = A$;
>   **for** $k := 1$ **to** n **do**
>   **for** $i := 1$ **to** n **do**
>   **for** $j := 1$ **to** n **do**
> $d_{i,j}^{(k)} := \min\left(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right)$;
> **end** FLOYD_ALL_PAIRS_SP

Or simply implemented as the following code segment:

```
for (k=0; k<n; k++)
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      if ( (d[i][k]+d[k][j]) < d[i][j] )
        d[i][j] = d[i][k]+d[k][j];
```

Problem 5c (10%)

Parallelize Floyd's algorithm with help of OpenMP programming. (You can assume that matrix A is given as input and the number of nodes is n.)

Suggested solution:

```
#pragma omp parallel default(shared) private(i,j,k)
{
for (k=0; k<n; k++)
  #pragma omp for
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      if ( (d[i][k]+d[k][j]) < d[i][j] )
        d[i][j] = d[i][k]+d[k][j];
}
```

Comment: It is very important to use the `private(i,j,k)` clause, otherwise the OpenMP parallelization won't work.