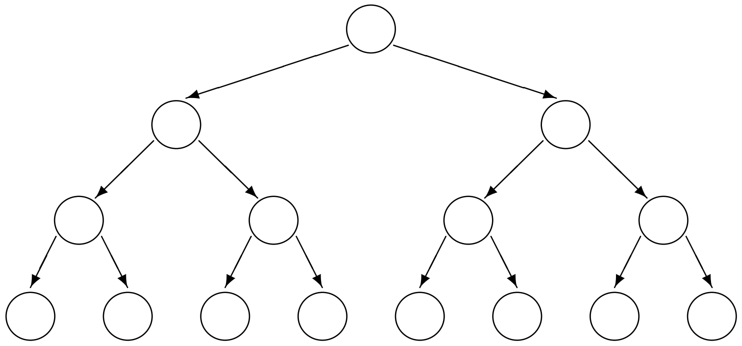
Suggested solutions for the INF3380 exam of spring 2011

Problem 1 (10%)



The above figure shows 15 tasks, each taking 10 minutes to be carried out by one worker. (Using more workers on the same task won't save any time.) Each arrow in the above figure means that the task being pointed cannot start before the pointing task has finished. What will be the shortest time for 4 workers to finish all the 15 tasks? (You need to explain your answer.) Repeat the same question for the case of 3 workers.

Suggested solution: During the first 10 minutes, only one worker can be in action to finish the task on the top level. During the second 10 minutes, two workers can concurrently finish the two tasks on the second level. During the third 10 minutes, all the four workers can concurrently finish the four tasks on the third level. The remaining 8 tasks on the bottom level will require 20 minutes for the four workers to finish. Therefore, the total time usage will be 50 minutes in the case of four workers.

In the case of three workers, the total time usage will be 60 minutes.

Problem 2 (20%)

Gustafson-Barsis's law (**ikke lenger pensum for 2013**) can be expressed as the following formula:

$$\Psi(n,p) \leq p + (1-p)s$$

- What do the symbols Ψ, n, p and s represent?
- What can Gustafson-Barsis's law be used to?
- Derive Gustafson-Barsis's law.

Suggested solution: Ψ represents the obtainable speedup, n represents the problem size, and

$$s = \frac{\sigma(n)}{\sigma(n) + \varphi(n)/p}$$

represents the fraction of time spent in the parallel computation performing inherently sequential operations.

Gustafson-Barsis's law can be used to estimate the speedup Ψ in case s is known.

Due to the above definition of s we can have

$$\begin{aligned}\sigma(n) &= (\sigma(n)+\varphi(n)/p)s,\\ \varphi(n) &= (\sigma(n)+\varphi(n)/p)(1-s)p.\end{aligned}$$

Therefore

$$\begin{aligned} &\Psi(n,p) \leq \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p}\\ \Rightarrow\ &\Psi(n,p) \leq \frac{(\sigma(n)+\varphi(n)/p)(s+(1-s)p)}{\sigma(n)+\varphi(n)/p}\\ \Rightarrow\ &\Psi(n,p) \leq s+(1-s)p = p+(1-p)s\end{aligned}$$

Problem 3 (20%)

```
double trapezoidal (int n) {
  double result = 0.0;
  double h = 1.0/n;
  double x;
  int i;

  x = 0.0;
  for (i=1; i<n; i++) {
    x += h;
    result += exp(5.0*x)+sin(x)-x*x;
  }

  x = 0.;
  result += 0.5*(exp(5.0*x)+sin(x)-x*x);

  x = 1.0;
  result += 0.5*(exp(5.0*x)+sin(x)-x*x);

  return (h*result);
}
```

Write a parallel version of the above `trapezoidal` function using MPI. (Hint: The `for`-loop needs to be modified a little bit to possess parallelism.) Write another parallel version using OpenMP.

Suggested solution: MPI parallelization:

```
double trapezoidal_MPI (int n) {
  double result = 0.0, global_sum;
  double h = 1.0/n;
  double x;
  int i;
  int my_rank, num_procs;
  MPI_Comm_Rank (MPI_COMM_WORLD, &my_rank);
  MPI_Comm_Size (MPI_COMM_WORLD, &num_procs);

  for (i=my_rank; i<n; i+=num_procs) {
    x = i*h;
    result += exp(5.0*x)+sin(x)-x*x;
  }

  MPI_Allreduce (&result, &global_sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

  x = 0.;
  result = global_sum + 0.5*(exp(5.0*x)+sin(x)-x*x);

  x = 1.0;
  result += 0.5*(exp(5.0*x)+sin(x)-x*x);

  return (h*result);
}
```

OpenMP parallelization:

```
double trapezoidal_OMP (int n) {
  double result = 0.0;
  double h = 1.0/n;
  double x;
  int i;

#pragma omp parallel for private(x) reduction(+: result)
  for (i=1; i<n; i++) {
    x = i*h;
    result += exp(5.0*x)+sin(x)-x*x;
  }

  x = 0.;
  result += 0.5*(exp(5.0*x)+sin(x)-x*x);

  x = 1.0;
  result += 0.5*(exp(5.0*x)+sin(x)-x*x);

  return (h*result);
}
```

Problem 4 (30%)

Floyd’s algorithm, which for example can be used to find the shortest distance between two and two cities, uses the following triple `for`-loop:

```
for (k=0; k<n; k++)
  for (i=0; i<n; i++)
```

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

- Explain where parallelism lies in the `for`-loop and why.
- Parallelize the above `for`-loop using OpenMP.
- In the context of an MPI parallelization, where the rows of the 2D-array `a` are distributed as rowwise block stripes, explain how often and how MPI communications should happen. (You don't need to write a complete MPI code.)

Suggested solution: The outermost for-loop that uses index `k` has to carry out the iterations sequentially one after another, but the two nested for-loops using `i` and `j` as index can be parallelized.

An OpenMP parallelization can be as follows:

```
for (k=0; k<n; k++)
#pragma omp parallel for private(j)
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      if ( a[i][j] > (a[i][k]+a[k][j]) )
        a[i][j] = a[i][k]+a[k][j];
```

In case of an MPI parallelization, in the beginning of each `k`-iteration, the MPI process that owns the *k*th row of matrix `a` should broadcast the entire row to all the other MPI processes.

Problem 5 (20%)

```
t = 0.0;
while (t < T) {
  t += dt;

  /* compute all the interior points */
  for (i=1; i<=M; i++)
    up[i] = 2*u[i]-um[i]
           +((dt*dt)/(dx*dx))*(u[i-1]-2*u[i]+u[i+1]);

  up[0] = value_of_left_BC(t);   // left boundary point
  up[M+1] = value_of_rigt_BC(t); // right boundary point

  /* preparation for next time step: array shuffle */
  tmp = um;
  um = u;
  u = up;
  up = tmp;
}
```

The above `while`-loop implements a finite difference method that solves the 1D wave equation (in a similar fashion as oblig 1 in spring 2011). Sketch an MPI parallelization that has the possibility of hiding the communication overhead, by using non-blocking MPI commands for sending and receiving messages.

Suggested solution: To hide the communication overhead, the following approach can be used. First, the value of `up[1]` should be calculated and a non-blocking MPI send command (`MPI_Isend`) and a non-blocking MPI receive command (`MPI_Irecv`) should be initiated. Second, the value of `up[my_M]` should be calculated and another pair of `MPI_Isend` and `MPI_Irecv` commands should be initiated. (Note that `my_M` is the local number of computation points.) Third, the remaining points are calculated using the `for`-loop, with index `i` starting from `2` and stopping at `my_M-1`. Fourth, all the four non-blocking MPI commands should be ensured to finish (by using `MPI_Wait`).